You are right to demand that an artist engage his work consciously, but you confuse two different things: solving the problem and correctly posing the question.
— Anton Chekhov, in a letter to A. S. Suvorin (October 27, 1888)

The more we reduce ourselves to machines in the lower things,
the more force we shall set free to use in the higher.
— Anna C. Brackett, *The Technique of Rest* (1892)

Arithmetic had entered the picture, with its many legs, its many spines and heads, its pitiless eyes made of zeroes. Two and two made four, was its message. But what if you didn't have two and two? Then things wouldn't add up.
— Margaret Atwood, *The Blind Assassin* (2000)

0 Introduction

0.1 What is an algorithm?

An algorithm is an unambiguous sequence of simple, mechanically executable instructions. Note that the word 'computer' doesn't appear anywhere in this definition; algorithms don't necessarily have anything to do with computers! For example, here is an algorithm for singing that annoying song '99 Bottles of Beer on the Wall', for arbitrary values of 99:

```
BottlesOfBeer($n$):
  For $i \leftarrow n$ down to $1$
      Sing "$i$ bottles of beer on the wall, $i$ bottles of beer,"
      Sing "Take one down, pass it around, $i - 1$ bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, $n$ bottles of beer on the wall."
```

The word 'algorithm' does *not* derive, as classically-trained algorithmophobes might guess, from the Greek root *algos* ($\alpha\lambda\gamma o\varsigma$), meaning 'pain'.[1] Rather, it is a corruption of the name of the 9th century Persian mathematician Abu 'Abd Allâh Muḥammad ibn Mûsâ al-Khwârizmî, which literally translates as "Mohammad, father of Adbdulla, son of Moses, the Kwârizmian".[2] (Kwârizm is an ancient city located in what is now the Xorazm Province of Uzbekistan.) Al-Khwârizmî is perhaps best known as the writer of the treatise *Kitab al-jabr wa'l-Muqâbala*, from which the modern word *algebra* derives. The word algorithm is a corruption of the older word *algorism* (by false connection to the Greek *arithmos* ($\alpha\rho\iota\theta\mu o\varsigma$), meaning 'number', and the aforementioned $\alpha\lambda\gamma o\varsigma$), used to describe the modern decimal system for writing and manipulating numbers—in particular, the use of a small circle or *sifr* to represent a missing quantity—which al-Khwârizmî brought into Persia from India. Thanks to the efforts of the medieval Italian mathematician Leonardo of Pisa, better known as Fibonacci, algorism began to replace the abacus as the preferred system of commercial calculation[3] in Europe in the late 12th century, although it was several more centuries before cyphers became truly ubiquitous. (Counting boards were used by the English and Scottish royal exchequers well into the 1600s.) So the word *algorithm* used to refer exclusively to mechanical pencil-and-paper methods for numerical calculations. People trained in the reliable execution of these methods were called—you guessed it—*computers*.

[1] Really. An *analgesic* is a medicine to remove pain; irrational fear of pain is called *algophobia*.

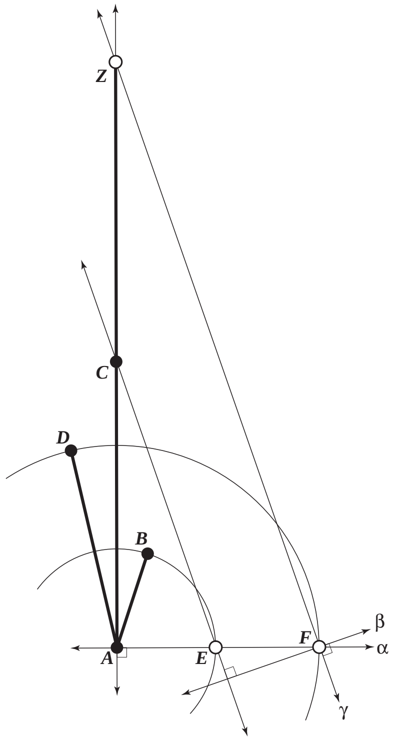
[2] Donald Knuth. Algorithms in modern mathematics and computer science. Chapter 4 in *Selected Papers on Computer Science*, Cambridge University Press, 1996. Originally published in 1981.

[3] from the Latin word *calculus*, meaning literally 'small rock', referring to the stones on a counting board, or abacus

Multiplication by compass and straightedge. However, algorithms have been with us since the dawn of civilization, centuries before Al-Khwârizmî and Fibonacci popularized the cypher. Here is an algorithm, popularized (but almost certainly not discovered) by Euclid about 2500 years ago, for multiplying or dividing numbers using a ruler and compass. The Greek geometers represented numbers using line segments of the appropriate length. In the pseudo-code below, CIRCLE(p, q) represents the circle centered at a point p and passing through another point q. Hopefully the other instructions are obvious.[4]

```
⟨⟨Construct the line perpendicular to $\ell$ and passing through $P$.⟩⟩
RIGHTANGLE($\ell, P$):
  Choose a point $A \in \ell$
  $A, B \leftarrow$ INTERSECT(CIRCLE($P, A$), $\ell$)
  $C, D \leftarrow$ INTERSECT(CIRCLE($A, B$), CIRCLE($B, A$))
  return LINE($C, D$)

⟨⟨Construct a point $Z$ such that $|AZ| = |AC||AD|/|AB|$.⟩⟩
MULTIPLYORDIVIDE($A, B, C, D$):
  $\alpha \leftarrow$ RIGHTANGLE(LINE($A, C$), $A$)
  $E \leftarrow$ INTERSECT(CIRCLE($A, B$), $\alpha$)
  $F \leftarrow$ INTERSECT(CIRCLE($A, D$), $\alpha$)
  $\beta \leftarrow$ RIGHTANGLE(LINE($E, C$), $F$)
  $\gamma \leftarrow$ RIGHTANGLE($\beta, F$)
  return INTERSECT($\gamma$, LINE($A, C$))
```



Multiplying or dividing using a compass and straightedge.

This algorithm breaks down the difficult task of multiplication into simple primitive steps: drawing a line between two points, drawing a circle with a given center and boundary point, and so on. The primitive steps need not be quite this primitive, but each primitive step must be something that the person or machine executing the algorithm already knows how to do. Notice in this example that Euclid made constructing a right angle a primitive operation in the MULTIPLYORDIVIDE algorithm by (as modern programmers would put it) writing a subroutine.

Multiplication by duplation and mediation. Here is an even older algorithm for multiplying large numbers, sometimes called *(Russian) peasant multiplication*. A variant of this method was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650 BC, from a document he claimed was (then) about 350 years old. This was the most common method of calculation by Europeans before Fibonacci's introduction of Arabic numerals. According to some sources, it was still being used in Russia well into the 20th century (along with the Julian calendar). This algorithm was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.[5]

[4] Euclid and his students almost certainly drew their constructions on an $\alpha\beta\alpha\xi$, a table covered in sand (or very small rocks). Over the next several centuries, the Greek *abax* evolved into the medieval European *abacus*.

[5] like the Apple II

```
PeasantMultiply($x, y$):
    $prod \leftarrow 0$
    while $x > 0$
        if $x$ is odd
            $prod \leftarrow prod + y$
        $x \leftarrow \lfloor x/2 \rfloor$
        $y \leftarrow y + y$
    return $p$
```

x	y	$prod$
		0
123	+456	= 456
61	+912	= 1368
30	~~1824~~	
15	+3648	= 5016
7	+7296	= 12312
3	+14592	= 26904
1	+29184	= 56088

The peasant multiplication algorithm breaks the difficult task of general multiplication into four simpler operations: (1) determining parity (even or odd), (2) addition, (3) duplation (doubling a number), and (4) mediation (halving a number, rounding down).[6] Of course a full specification of this algorithm requires describing how to perform those four 'primitive' operations. When executed by hand, peasant multiplication requires (a constant factor!) more paperwork, but the necessary operations are easier for humans to remember than the 10×10 multiplication table required by the American grade school algorithm.[7]

The correctness of peasant multiplication follows from the following recursive identity, which holds for any non-negative integers x and y:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

A bad example. Consider "Martin's algorithm":[8]

```
BecomeAMillionaireAndNeverPayTaxes:
    Get a million dollars.
    Don't pay taxes.
    If you get caught,
        Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy. A group of billionaire CEOs would consider this an algorithm, since for them the first step is both unambiguous and trivial, but for the rest of us poor slobs who don't have a million dollars handy, Martin's procedure is too vague to be considered an algorithm. On the other hand, this is a perfect example of a *reduction*—it *reduces* the problem of being a millionaire and never paying taxes to the 'easier' problem of acquiring a million dollars. We'll see reductions over and over again in this class. As hundreds of businessmen and politicians have demonstrated, if you know how to solve the easier problem, a reduction tells you how to solve the harder one.

Martin's algorithm, like many of our previous examples, is not the kind of algorithm that computer scientists are used to thinking about, because it is phrased in terms of operations that are

[6] The ancient Egyptian version of this algorithm does not use mediation or parity, but it does use comparisons. To avoid halving, the algorithm pre-computes two tables by repeated doubling: one containing all the powers of 2 not exceeding x, the other containing the same powers of 2 multiplied by y. The powers of 2 that sum to x are found by subtraction, and the corresponding entries in the other table are added together to form the product. Egyptian scribes made large tables of powers of 2 to speed up these calculations.

[7] American school kids learn a variant of the *lattice* multiplication algorithm developed by Indian mathematicians and described by Fibonacci in *Liber Abaci*. The two algorithms are equivalent if the input numbers are represented in binary.

[8] S. Martin, "You Can Be A Millionaire", Saturday Night Live, January 21, 1978. Reprinted in *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

difficult for computers to perform. In this class, we'll focus (almost!) exclusively on algorithms that can be reasonably implemented on a computer. In other words, each step in the algorithm must be something that either is directly supported by common programming languages (such as arithmetic, assignments, loops, or recursion) or is something that you've already learned how to do in an earlier class (like sorting, binary search, or depth first search).

Congressional apportionment. Here is another good example of an algorithm that comes from outside the world of computing. Article I, Section 2 of the US Constitution requires that

> Representatives and direct Taxes shall be apportioned among the several States which may be included within this Union, according to their respective Numbers.... The Number of Representatives shall not exceed one for every thirty Thousand, but each State shall have at Least one Representative....

Since there are a limited number of seats available in the House of Representatives, exact proportional representation is impossible without either shared or fractional representatives, neither of which are legal. As a result, several different apportionment algorithms have been proposed and used to round the fractional solution fairly. The algorithm actually used today, called *the Huntington-Hill method* or *the method of equal proportions*, was first suggested by Census Bureau statistician Joseph Hill in 1911, refined by Harvard mathematician Edward Huntington in 1920, adopted into Federal law (2 U.S.C. §§2a and 2b) in 1941, and survived a Supreme Court challenge in 1992.[9] The input array $P[1..n]$ stores the populations of the n states, and R is the total number of representatives. Currently, $n = 50$ and $R = 435$.[10]

```
ApportionCongress($P[1..n], R$):
  $H \leftarrow$ NewMaxHeap
  for $i \leftarrow 1$ to $n$
      $r[i] \leftarrow 1$
      Insert$\big(H, i, P[i]/\sqrt{2}\big)$
  $R \leftarrow R - n$

  while $R > 0$
      $s \leftarrow$ ExtractMax$(H)$
      $r[s] \leftarrow r[s] + 1$
      Insert$\big(H, i, P[i]/\sqrt{r[i](r[i]+1)}\,\big)$
      $R \leftarrow R - 1$

  return $r[1..n]$
```

Note that this description assumes that you know how to implement a max-heap and its basic operations NewMaxHeap, Insert, and ExtractMax. Moreover, the correctness of the algorithm doesn't depend at all on how these operations are implemented. The Census Bureau implements the max-heap as an unsorted array inside an Excel spreadsheet; you should have learned a more efficient solution in your undergraduate data structures class.

[9] Overruling an earlier ruling by a federal district court, the Supreme Court unanimously held that *any* apportionment method adopted in good faith by Congress is constitutional (*United States Department of Commerce v. Montana*). The congressional apportionment algorithm is described in detail at the U.S. Census Department web site http://www.census.gov/population/www/censusdata/apportionment/computing.html. A good history of the apportionment problem can be found at http://www.thirty-thousand.org/pages/Apportionment.htm. A report by the Congressional Research Service describing various apportionment methods is available at http://www.rules.house.gov/archives/RL31074.pdf.

[10] The DC Fair and Equal House Voting Rights Act of 2006, if it becomes law, would permanently increase the number of representatives to 437, exactly one of which would be allocated to to the District of Columbia. Thus, starting in 2010, the apportionment algorithm would be run with $n = 50$ and $R = 436$.

Combinatorial versus numerical. This class will focus specifically on *combinatorial* algorithms, as opposed to *numerical* algorithms. The distinction is fairly artificial, but essentially, numerical algorithms are used to approximate computation with ideal real numbers on finite precision computers. For example, here's a numerical algorithm to compute the square root of a number to a given precision. This algorithm works remarkably quickly—every iteration doubles the number of correct digits.

```
SquareRoot($x, \varepsilon$):
  $s \leftarrow 1$
  while $|s - x/s| > \varepsilon$
      $s \leftarrow (s + x/s)/2$
  return $s$
```

The output of a numerical algorithm is necessarily an approximation to some ideal mathematical object. Any number that's close enough to the ideal answer is a correct answer. Combinatorial algorithms, on the other hand, manipulate discrete objects like arrays, lists, trees, and graphs that can be represented *exactly* on a digital computer.

0.2 Writing down algorithms

Algorithms are *not* programs; they should not be described in a particular programming language. The whole *point* of this course is to develop computational techniques that can be used in *any* programming language.[11] The idiosyncratic syntactic details of C, Java, Python, Scheme, Visual Basic, ML, Smalltalk, Javascript, Forth, TEX, COBOL, Intercal, or Brainfuck[12] are of absolutely no importance in algorithm design, and focusing on them will only distract you from what's really going on.[13] What we really want is closer to what you'd write in the *comments* of a real program than the code itself.

On the other hand, a plain English prose description is usually not a good idea either. Algorithms have a lot of structure—especially conditionals, loops, and recursion—that are far too easily hidden by unstructured prose. Like any language spoken by humans, English is full of ambiguities, subtleties, and shades of meaning, but algorithms must be described as accurately as possible. Finally and more seriously, many people have a tendency to describe loops informally: "Do this first, then do this second, and so on." As anyone who has taken one of those 'what comes next in this sequence?' tests already knows, specifying what happens in the first couple of iterations of a loop doesn't say much about what happens later on.[14] Phrases like 'and so on' or 'do X over and over'

[11] See http://www.ionet.net/~timtroyr/funhouse/beer.html for implementations of the BottlesOfBeer algorithm in over 200 different programming languages.

[12] Brainfuck is the well-deserved name of a programming language invented by Urban Mueller in 1993. Brainfuck programs are written entirely using the punctuation characters `<>+-,.[]`, each representing a different operation (roughly: shift left, shift right, increment, decrement, input, output, begin loop, end loop). See http://www.catseye.mb.ca/esoteric/bf/ for a complete definition, sample programs, an interpreter (written in just 230 characters of C), and related shit.

[13] This is, of course, a matter of religious conviction. Linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states that people think only in the categories imposed by their languages. According to this hypothesis, some concepts in one language simply cannot be understood by speakers of other languages, not just because of technological advancement—How would you translate "jump the shark" or "blog" into ancient Greek?—but because of inherent structural differences between languages and cultures. Anne Rice espoused a similar idea in her later Lestat books. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is some strength to this idea when applied to programming languages. (What's the Y combinator, again? How do templates work?) Fortunately, those differences are generally too subtle to have much impact in *this* class.

[14] See http://www.research.att.com/~njas/sequences/.

or 'et cetera' are a good indication that the algorithm *should* have been described in terms of loops or recursion, and the description should have specified what happens in a *generic* iteration of the loop. Similarly, the appearance of the phrase 'and so on' in a proof is a good indication that the proof *should* have been done by induction!

The best way to write down an algorithm is using pseudocode. Pseudocode uses the structure of formal programming languages and mathematics to break the algorithm into one-sentence steps, but those sentences can be written using mathematics, pure English, or some mixture of the two. Exactly how to structure the pseudocode is a personal choice, but the overriding goal should be clarity and precision. Here are the basic rules I follow:

- Use standard imperative programming keywords (if/then/else, while, for, repeat/until, case, return) and notation (variable←value, array[index], pointer→field, function(args), etc.)

- The block structure should be visible from across the room. Indent everything carefully and consistently. Don't use syntactic sugar (like C/C++/Java braces or Pascal/Algol begin/end tags) unless the pseudocode is absolutely unreadable without it.

- *Don't* typeset keywords in a different **font** or `style`. Changing type style emphasizes the keywords, making the reader think the syntactic sugar is actually important—it isn't!

- Each statement should fit on one line, and each line should contain only one statement. (The only exception is extremely short and similar statements like $i \leftarrow i + 1;\ j \leftarrow j - 1;\ k \leftarrow 0$.)

- Put each structuring statement (for, while, if) on its own line. The order of nested loops matters a great deal; make it absolutely obvious.

- Use short but mnemonic algorithm and variable names. Absolutely *never* use pronouns!

A good description of an algorithm reveals the internal structure, hides irrelevant details, and can be implemented easily by any competent programmer in any programming language, even if they don't understand why the algorithm works. Good pseudocode, like good code, makes the algorithm much easier to understand and analyze; it also makes mistakes much easier to spot. The algorithm descriptions in the textbooks and lecture notes are good examples of what we want to see on your homeworks and exams.

0.3 Analyzing algorithms

It's not enough just to write down an algorithm and say 'Behold!' We also need to convince ourselves (and our graders) that the algorithm does what it's supposed to do, and that it does it quickly.

Correctness: In the real world, it is often acceptable for programs to behave correctly most of the time, on all 'reasonable' inputs. Not in this class; our standards are higher[15]. We need to *prove* that our algorithms are correct on *all possible* inputs. Sometimes this is fairly obvious, especially for algorithms you've seen in earlier courses. But many of the algorithms we will discuss in this course will require some extra work to prove. Correctness proofs almost always involve induction. We *like* induction. Induction is our *friend*.[16]

Before we can formally prove that our algorithm does what we want it to, we have to formally state what we want the algorithm to do! Usually problems are given to us in real-world terms,

[15] or at least different

[16] If induction is *not* your friend, you will have a hard time in this course.

not with formal mathematical descriptions. It's up to us, the algorithm designer, to restate the problem in terms of mathematical objects that we can prove things about: numbers, arrays, lists, graphs, trees, and so on. We also need to determine if the problem statement makes any hidden assumptions, and state those assumptions explicitly. (For example, in the song "n Bottles of Beer on the Wall", n is always a positive integer.) Restating the problem formally is not only required for proofs; it is also one of the best ways to really understand what the problems is asking for. The hardest part of solving a problem is figuring out the right way to ask the question!

An important distinction to keep in mind is the distinction between a problem and an algorithm. A problem is a task to perform, like "Compute the square root of x" or "Sort these n numbers" or "Keep n algorithms students awake for t minutes". An algorithm is a set of instructions that you follow if you want to execute this task. The same problem may have hundreds of different algorithms.

Running time: The usual way of distinguishing between different algorithms for the same problem is by how fast they run. Ideally, we want the fastest possible algorithm for our problem. In the real world, it is often acceptable for programs to run efficiently most of the time, on all 'reasonable' inputs. Not in this class; our standards are different. We require algorithms that *always* run efficiently, even in the worst case.

But how do we measure running time? As a specific example, how long does it take to sing the song BottlesOfBeer(n)? This is obviously a function of the input value n, but it also depends on how quickly you can sing. Some singers might take ten seconds to sing a verse; others might take twenty. Technology widens the possibilities even further. Dictating the song over a telegraph using Morse code might take a full minute per verse. Ripping an mp3 over the Web might take a tenth of a second per verse. Duplicating the mp3 in a computer's main memory might take only a few microseconds per verse.

What's important here is how the singing time changes as n grows. Singing BottlesOfBeer$(2n)$ takes about twice as long as singing BottlesOfBeer(n), no matter what technology is being used. This is reflected in the asymptotic singing time $\Theta(n)$. We can measure time by counting how many times the algorithm executes a certain instruction or reaches a certain milestone in the 'code'. For example, we might notice that the word 'beer' is sung three times in every verse of BottlesOfBeer, so the number of times you sing 'beer' is a good indication of the total singing time. For this question, we can give an exact answer: BottlesOfBeer(n) uses exactly $3n+3$ beers.

There are plenty of other songs that have non-trivial singing time. This one is probably familiar to most English-speakers:

```
NDaysOfChristmas($gifts[2..n]$):
    for $i \leftarrow 1$ to $n$
        Sing "On the $i$th day of Christmas, my true love gave to me"
        for $j \leftarrow i$ down to $2$
            Sing "$j$ $gifts[j]$"
        if $i > 1$
            Sing "and"
        Sing "a partridge in a pear tree."
```

The input to NDaysOfChristmas is a list of $n-1$ gifts. It's quite easy to show that the singing time is $\Theta(n^2)$; in particular, the singer mentions the name of a gift $\sum_{i=1}^{n} i = n(n+1)/2$ times (counting the partridge in the pear tree). It's also easy to see that during the first n days of Christmas, my true love gave to me exactly $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$ gifts. Other songs that take quadratic time to sing are "Old MacDonald", "There Was an Old Lady Who Swallowed a Fly", "Green Grow the Rushes O", "The Barley Mow", "Echad Mi Yode'a" ("Who knows one?"), "Allouette", "Ist das nicht ein Schnitzelbank?"[17] etc. For details, consult your nearest preschooler.

```
OLDMACDONALD($animals[1..n]$, $noise[1..n]$):
  for $i \leftarrow 1$ to $n$
      Sing "Old MacDonald had a farm, E I E I O"
      Sing "And on this farm he had some $animals[i]$, E I E I O"
      Sing "With a $noise[i]$ $noise[i]$ here, and a $noise[i]$ $noise[i]$ there"
      Sing "Here a $noise[i]$, there a $noise[i]$, everywhere a $noise[i]$ $noise[i]$"
      for $j \leftarrow i-1$ down to 1
          Sing "$noise[j]$ $noise[j]$ here, $noise[j]$ $noise[j]$ there"
          Sing "Here a $noise[j]$, there a $noise[j]$, everywhere a $noise[j]$ $noise[j]$"
      Sing "Old MacDonald had a farm, E I E I O."
```

```
ALLOUETTE($lapart[1..n]$):
  Chantez „Allouette, gentille allouette, allouette, je te plumerais."
  pour tout $i$ de 1 á $n$
      Chantez „Je te plumerais $lapart[i]$, je te plumerais $lapart[i]$."
      pour tout $j$ de $i-1$ á bas á 1
          Chantez „Et $lapart[j]$, et $lapart[j]$,"
      Chantez „Ooooooo!"
      Chantez „Allouette, gentille alluette, allouette, je te plumerais."
```

For a slightly more complicated example, consider the algorithm APPORTIONCONGRESS. Here the running time obviously depends on the implementation of the max-heap operations, but we can certainly bound the running time as $O(N+RI+(R-n)E)$, where N is the time for a NEWMAXHEAP, I is the time for an INSERT, and E is the time for an EXTRACTMAX. Under the reasonable assumption that $R > 2n$ (on average, each state gets at least two representatives), this simplifies to $O(N+R(I+E))$. The Census Bureau uses an unsorted array of size n, for which $N = I = \Theta(1)$ (since we know a priori how big the array is), and $E = \Theta(n)$, so the overall running time is $\Theta(Rn)$. This is fine for the federal government, but if we want to be more efficient, we can implement the heap as a perfectly balanced n-node binary tree (or a heap-ordered array). In this case, we have $N = \Theta(1)$ and $I = R = O(\log n)$, so the overall running time is $\Theta(R \log n)$.

Incidentally, there is a faster algorithm for apportioning Congress. I'll give extra credit to the first student who can find the faster algorithm, analyze its running time, and prove that it always gives exactly the same results as APPORTIONCONGRESS.

Sometimes we are also interested in other computational resources: space, randomness, page faults, inter-process messages, and so forth. We use the same techniques to analyze those resources as we use for running time.

0.4 Why are we here, anyway?

This class is ultimately about learning two skills that are crucial for computer scientists: how to *think* about algorithms and how to *talk* about algorithms. Along the way, you'll pick up a bunch of algorithmic facts—mergesort runs in $\Theta(n \log n)$ time; the amortized time to search in a splay tree is $O(\log n)$; greedy algorithms usually don't produce optimal solutions; the traveling salesman problem is NP-hard—but these aren't the point of the course. You can always look up mere facts in a textbook or on the web, provided you have enough intuition and experience to know what to look

[17] Wakko: Ist das nicht Otto von Schnitzelpusskrankengescheitmeyer?
Yakko and Dot: Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!!

for. That's why we let you bring cheat sheets to the exams; we don't want you wasting your study time trying to memorize all the facts you've seen. You'll also practice a lot of algorithm design and analysis skills—finding useful (counter)examples, developing induction proofs, solving recurrences, using big-Oh notation, using probability, giving problems crisp mathematical descriptions, and so on. These skills are *very* useful, but they aren't really the point of the course either. At this point in your educational career, you should be able to pick up those skills on your own, once you know what you're trying to do.

The first main goal of this course is to help you develop algorithmic *intuition*. How do various algorithms really work? When you see a problem for the first time, how should you attack it? How do you tell which techniques will work at all, and which ones will work best? How do you judge whether one algorithm is better than another? How do you tell whether you have the best possible solution?

Our second main goal is to help you develop algorithmic *language*. It's not enough just to understand how to solve a problem; you also have to be able to explain your solution to somebody else. I don't mean just how to turn your algorithms into working code—despite what many students (and inexperienced programmers) think, 'somebody else' is *not* just a computer. Nobody programs alone. Code is read far more often than it is written, or even compiled. Perhaps more importantly in the short term, explaining something to somebody else is one of the best ways of clarifying your own understanding. As Richard Feynman apocryphally put it, "If you can't explain what you're doing to your grandmother, you don't understand it."

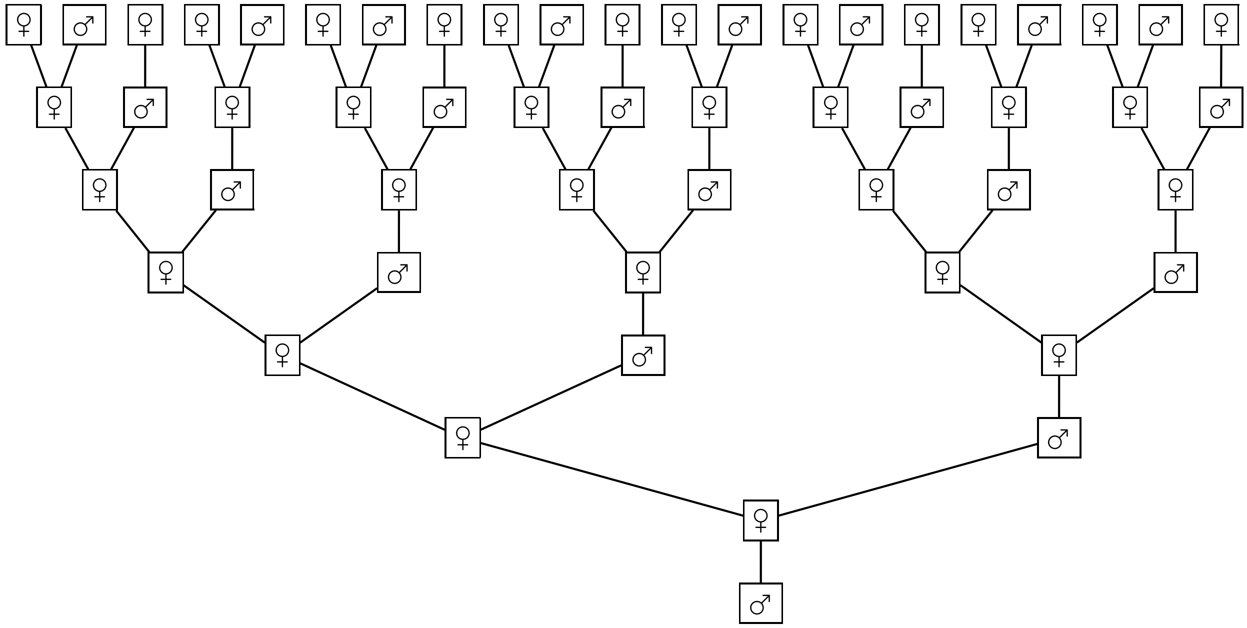
Unfortunately, there is no systematic procedure—no algorithm—to determine which algorithmic techniques are most effective at solving a given problem, or finding good ways to explain, analyze, optimize, or implement a given algorithm. Like many other human activities (music, writing, juggling, acting, martial arts, sports, cooking, programming, teaching, etc.), experts will disagree on the relative values of different techniques. Ultimately, the *only* way to master these skills is to make them your own, through practice, practice, and more practice. We *can't* teach you how to do well in this class. All we can do is lay out a few tools, show you how to use them, create opportunities for you to practice, and give you feedback based on our own experience and intuition. The rest is up to you.

Good algorithms are extremely useful, elegant, surprising, deep, even beautiful. But most importantly, algorithms are *fun*!! I hope this course will inspire at least some you to come play!

R Solving Recurrences

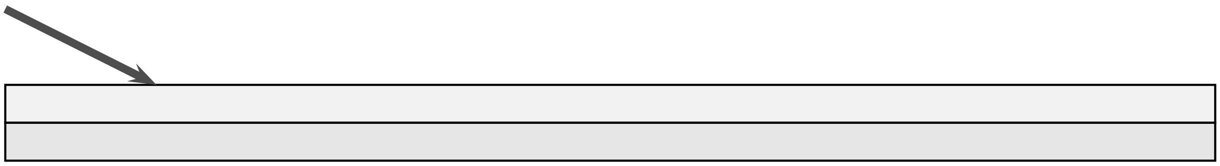
R.1 Fun with Fibonacci numbers

Consider the reproductive cycle of bees. Each male bee has a mother but no father; each female bee has both a mother and a father. If we examine the generations we see the following family tree:

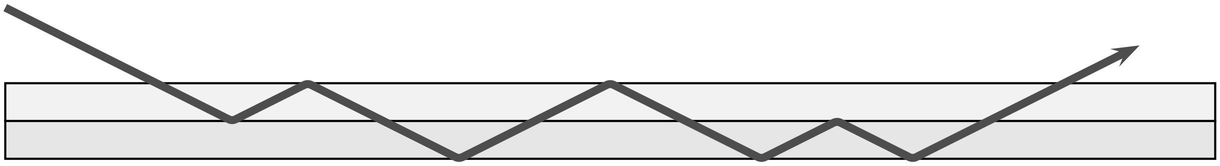


We easily see that the number of ancestors in each generation is the sum of the two numbers before it. For example, our male bee has three great-grandparents, two grandparents, and one parent, and $3 = 2 + 1$. The number of ancestors a bee has in generation n is defined by the Fibonacci sequence; we can also see this by applying the rule of sum.

As a second example, consider light entering two adjacent planes of glass:

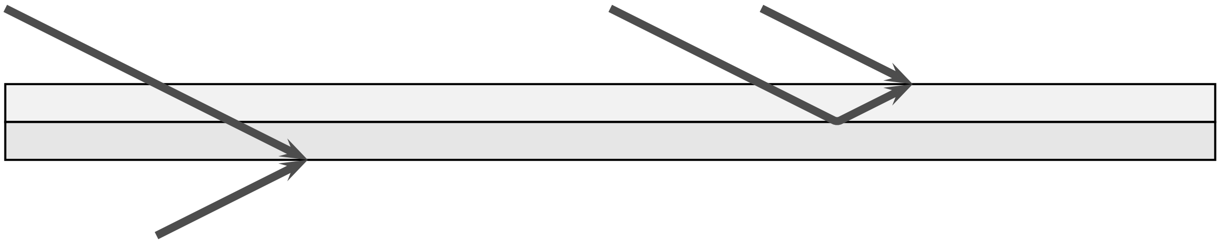


At any meeting surface (between the two panes of glass, or between the glass and air), the light may either reflect or continue straight through (refract). For example, here is the light bouncing seven times before it leaves the glass.



In general, how many different paths can the light take if we are told that it bounces n times before leaving the glass?

The answer to the question (in case you haven't guessed) rests with the Fibonacci sequence. We can divide the set of paths with n reflections into two subsets, depending on where the first reflection happens.



- Suppose the first bounce is on the boundary between the two panes. After the bounce, the light either leaves the class immediately (so $n = 1$), or bounces again off the top of the upper pane. After the *second* bounce, if any, the path is equivalent to a path that enters from the top and bounces $n - 2$ times.
- Suppose the first bounce is *not* at the boundary between the two panes. Then either there are no bounces at all (so $n = 0$) or the first bounce is off the bottom pane. After the first bounce, the path is equivalent to a path that enters from the bottom and bounces $n - 1$ times. Entering through the bottom pane is the same as entering through the top pane (but flipped over).

Thus, we obtain the following recurrence relation for F_n, the number of paths with exactly n bounces. There is exactly one way for the light to travel with no bounces—straight through—and exactly two ways for the light to travel with only one bounce—off the bottom and off the middle. For any $n > 1$, there are F_{n-1} paths where the light bounces off the bottom of the glass, and F_{n-2} paths where the light bounces off the middle and then off the top.

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 2 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

R.2 Sequences, sequence operators, and annihilators

We have shown that several different problems can be expressed in terms of Fibonacci sequences, but we don't yet know how to explicitly compute the nth Fibonacci number, or even (and more importantly) roughly how big it is. We can easily write a program to compute the nth Fibonacci number, but that doesn't help us much here. What we really want is a *closed form solution* for the Fibonacci recurrence—an explicit algebraic formula without conditionals, loops, or recursion.

In order to solve recurrences like the Fibonacci recurrence, we first need to understand *operations* on infinite sequences of numbers. Although these sequences are formally defined as *functions* of the form $A : \mathbb{N} \to \mathbb{R}$, we will write them either as $A = \langle a_0, a_1, a_2, a_3, a_4, \ldots \rangle$ when we want to emphasize the entire sequence[1], or as $A = \langle a_i \rangle$ when we want to emphasize a generic element. For example, the Fibonacci sequence is $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \rangle$.

We can naturally define several sequence operators:

- We can add or subtract any two sequences:

$$\begin{aligned} \langle a_i \rangle + \langle b_i \rangle &= \langle a_0, a_1, a_2, \ldots \rangle + \langle b_0, b_1, b_2, \ldots \rangle = \langle a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots \rangle = \langle a_i + b_i \rangle \\ \langle a_i \rangle - \langle b_i \rangle &= \langle a_0, a_1, a_2, \ldots \rangle - \langle b_0, b_1, b_2, \ldots \rangle = \langle a_0 - b_0, a_1 - b_1, a_2 - b_2, \ldots \rangle = \langle a_i - b_i \rangle \end{aligned}$$

[1] It really doesn't matter whether we start a sequence with a_0 or a_1 or a_5 or even a_{-17}. Zero is often a convenient starting point for many recursively defined sequences, so we'll usually start there.

- We can multiply any sequence by a constant:

$$c \cdot \langle a_i \rangle = c \cdot \langle a_0, a_1, a_2, \ldots \rangle = \langle c \cdot a_0, c \cdot a_1, c \cdot a_2, \ldots \rangle = \langle c \cdot a_i \rangle$$

- We can shift any sequence to the left by removing its initial element:

$$\mathbf{E}\langle a_i \rangle = \mathbf{E}\langle a_0, a_1, a_2, a_3, \ldots \rangle = \langle a_1, a_2, a_3, a_4, \ldots \rangle = \langle a_{i+1} \rangle$$

Example: We can understand these operators better by looking at some specific examples, using the sequence T of powers of two.

$$\begin{aligned} T &= \langle 2^0, 2^1, 2^2, 2^3, \ldots \rangle = \langle 2^i \rangle \\ \mathbf{E}T &= \langle 2^1, 2^2, 2^3, 2^4, \ldots \rangle = \langle 2^{i+1} \rangle \\ 2T &= \langle 2 \cdot 2^0, 2 \cdot 2^1, 2 \cdot 2^2, 2 \cdot 2^3, \ldots \rangle = \langle 2^1, 2^2, 2^3, 2^4, \ldots \rangle = \langle 2^{i+1} \rangle \\ 2T - \mathbf{E}T &= \langle 2^1 - 2^1, 2^2 - 2^2, 2^3 - 2^3, 2^4 - 2^4, \ldots \rangle = \langle 0, 0, 0, 0, \ldots \rangle = \langle 0 \rangle \end{aligned}$$

R.2.1 Properties of operators

It turns out that the distributive property holds for these operators, so we can rewrite $\mathbf{E}T - 2T$ as $(\mathbf{E} - 2)T$. Since $(\mathbf{E} - 2)T = \langle 0, 0, 0, 0, \ldots \rangle$, we say that the operator $(\mathbf{E} - 2)$ *annihilates* T, and we call $(\mathbf{E} - 2)$ an *annihilator* of T. Obviously, we can trivially annihilate any sequence by multiplying it by zero, so as a technical matter, we do not consider multiplication by 0 to be an annihilator.

What happens when we apply the operator $(E - 3)$ to our sequence T?

$$(\mathbf{E} - 3)T = \mathbf{E}T - 3T = \langle 2^{i+1} \rangle - 3\langle 2^i \rangle = \langle 2^{i+1} - 3 \cdot 2^i \rangle = \langle -2^i \rangle = -T$$

The operator $(\mathbf{E} - 3)$ did very little to our sequence T; it just flipped the sign of each number in the sequence. In fact, we will soon see that *only* $(\mathbf{E} - 2)$ will annihilate T, and all other simple operators will affect T in very minor ways. Thus, if we know how to annihilate the sequence, we know what the sequence must look like.

In general, $(\mathbf{E} - c)$ annihilates any geometric sequence $A = \langle a_0, a_0 c, a_0 c^2, a_0 c^3, \ldots \rangle = \langle a_0 c^i \rangle$:

$$(\mathbf{E} - c)\langle a_0 c^i \rangle = \mathbf{E}\langle a_0 c^i \rangle - c\langle a_0 c_i \rangle = \langle a_0 c^{i+1} \rangle - \langle c \cdot a_0 c_i \rangle = \langle a_0 c^{i+1} - a_0 c^{i+1} \rangle = \langle 0 \rangle$$

To see that this is the only operator of this form that annihilates A, let's see the effect of operator $(\mathbf{E} - d)$ for some $d \neq c$:

$$(\mathbf{E} - d)\langle a_0 c^i \rangle = \mathbf{E}\langle a_0 c^i \rangle - d\langle a_0 c_i \rangle = \langle a_0 c^{i+1} - d a_0 c_i \rangle = \langle (c - d) a_0 c^i \rangle = (c - d)\langle a_0 c^i \rangle$$

So we have a more rigorous confirmation that an annihilator annihilates exactly one type of sequence, but multiplies other similar sequences by a constant.

We can use this fact about annihilators of geometric sequences to solve certain recurrences. For example, consider the sequence $R = \langle r_0, r_1, r_2, \ldots \rangle$ defined recursively as follows:

$$\begin{aligned} r_0 &= 3 \\ r_{i+1} &= 5 r_i \end{aligned}$$

We can easily prove that the operator $(\mathbf{E} - 5)$ annihilates R:

$$(\mathbf{E} - 5)\langle r_i \rangle = \mathbf{E}\langle r_i \rangle - 5\langle r_i \rangle = \langle r_{i+1} \rangle - \langle 5 r_i \rangle = \langle r_{i+1} - 5 r_i \rangle = \langle 0 \rangle$$

Since $(\mathbf{E} - 5)$ is an annihilator for R, we must have the closed form solution $r_i = r_0 5^i = 3 \cdot 5^i$. We can easily verify this by induction, as follows:

$$\begin{aligned} r_0 &= 3 \cdot 5^0 = 3 \quad \checkmark && \text{[definition]} \\[1ex] r_i &= 5r_{i-1} && \text{[definition]} \\ &= 5 \cdot (3 \cdot 5^{i-1}) && \text{[induction hypothesis]} \\ &= 5^i \cdot 3 \quad \checkmark && \text{[algebra]} \end{aligned}$$

R.2.2 Multiple operators

An operator is a function that transforms one sequence into another. Like any other function, we can apply operators one after another to the same sequence. For example, we can multiply a sequence $\langle a_i \rangle$ by a constant d and then by a constant c, resulting in the sequence $c(d\langle a_i \rangle) = \langle c \cdot d \cdot a_i \rangle = (cd)\langle a_i \rangle$. Alternatively, we may multiply the sequence by a constant c and then shift it to the left to get $\mathbf{E}(c\langle a_i \rangle) = \mathbf{E}\langle c \cdot a_i \rangle = \langle c \cdot a_{i+1} \rangle$. This is exactly the same as applying the operators in the reverse order: $c(\mathbf{E}\langle a_i \rangle) = c\langle a_{i+1} \rangle = \langle c \cdot a_{i+1} \rangle$. We can also shift the sequence twice to the left: $\mathbf{E}(\mathbf{E}\langle a_i \rangle) = \mathbf{E}\langle a_{i+1} \rangle = \langle a_{i+2} \rangle$. We will write this in shorthand as $\mathbf{E}^2\langle a_i \rangle$. More generally, the operator $\mathbf{E}^k$ shifts a sequence k steps to the left: $\mathbf{E}^k\langle a_i \rangle = \langle a_{i+k} \rangle$.

We now have the tools to solve a whole host of recurrence problems. For example, what annihilates $C = \langle 2^i + 3^i \rangle$? Well, we know that $(\mathbf{E} - 2)$ annihilates $\langle 2^i \rangle$ while leaving $\langle 3^i \rangle$ essentially unscathed. Similarly, $(\mathbf{E} - 3)$ annihilates $\langle 3^i \rangle$ while leaving $\langle 2^i \rangle$ essentially unscathed. Thus, if we apply both operators one after the other, we see that $(\mathbf{E} - 2)(\mathbf{E} - 3)$ annihilates our sequence C.

In general, for any integers $a \neq b$, the operator $(\mathbf{E} - a)(\mathbf{E} - b)$ annihilates any sequence of the form $\langle c_1 a^i + c_2 b^i \rangle$ but nothing else. We will often 'multiply out' the operators into the shorthand notation $\mathbf{E}^2 - (a + b)\mathbf{E} + ab$. It is left as an exhilarating exercise to the student to verify that this shorthand actually makes sense—the operators $(\mathbf{E} - a)(\mathbf{E} - b)$ and $\mathbf{E}^2 - (a + b)\mathbf{E} + ab$ have the same effect on every sequence.

We now know finally enough to solve the recurrence for Fibonacci numbers. Specifically, notice that the recurrence $F_i = F_{i-1} + F_{i-2}$ is annihilated by $\mathbf{E}^2 - \mathbf{E} - 1$:

$$\begin{aligned} (\mathbf{E}^2 - \mathbf{E} - 1)\langle F_i \rangle &= \mathbf{E}^2\langle F_i \rangle - \mathbf{E}\langle F_i \rangle - \langle F_i \rangle \\ &= \langle F_{i+2} \rangle - \langle F_{i+1} \rangle - \langle F_i \rangle \\ &= \langle F_{i-2} - F_{i-1} - F_i \rangle \\ &= \langle 0 \rangle \end{aligned}$$

Factoring $\mathbf{E}^2 - \mathbf{E} - 1$ using the quadratic formula, we obtain

$$\mathbf{E}^2 - \mathbf{E} - 1 = (\mathbf{E} - \phi)(\mathbf{E} - \hat{\phi})$$

where $\phi = (1 + \sqrt{5})/2 \approx 1.618034$ is the golden ratio and $\hat{\phi} = (1 - \sqrt{5})/2 = 1 - \phi = -1/\phi$. Thus, the operator $(E - \phi)(E - \hat{\phi})$ annihilates the Fibonacci sequence, so F_i must have the form

$$F_i = c\phi^i + \hat{c}\hat{\phi}^i$$

for some constants c and $\hat{c}$. We call this the *generic solution* to the recurrence, since it doesn't depend at all on the base cases. To compute the constants c and $\hat{c}$, we use the base cases $F_0 = 0$ and $F_1 = 1$ to obtain a pair of linear equations:

$$\begin{aligned} F_0 &= 0 = c + \hat{c} \\ F_1 &= 1 = c\phi + \hat{c}\hat{\phi} \end{aligned}$$

Solving this system of equations gives us $c = 1/(2\phi - 1) = 1/\sqrt{5}$ and $\hat{c} = -1/\sqrt{5}$.

We now have a closed-form expression for the ith Fibonacci number:

$$\boxed{F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i}$$

With all the square roots in this formula, it's quite amazing that Fibonacci numbers are integers. However, if we do all the math correctly, all the square roots cancel out when i is an integer. (In fact, this is pretty easy to prove using the binomial theorem.)

R.2.3 Degenerate cases

We can't quite solve *every* recurrence yet. In our above formulation of $(\mathbf{E} - a)(\mathbf{E} - b)$, we assumed that $a \neq b$. What about the operator $(\mathbf{E} - a)(\mathbf{E} - a) = (\mathbf{E} - a)^2$? It turns out that this operator annihilates sequences such as $\langle ia^i \rangle$:

$$\begin{aligned}
(\mathbf{E} - a)\langle ia^i \rangle &= \langle (i+1)a^{i+1} - (a)ia^i \rangle \\
&= \langle (i+1)a^{i+1} - ia^{i+1} \rangle \\
&= \langle a^{i+1} \rangle \\
(\mathbf{E} - a)^2\langle ia^i \rangle &= (\mathbf{E} - a)\langle a^{i+1} \rangle = \langle 0 \rangle
\end{aligned}$$

More generally, the operator $(\mathbf{E} - a)^k$ annihilates any sequence $\langle p(i) \cdot a^i \rangle$, where $p(i)$ is any polynomial in i of degree $k - 1$. As an example, $(\mathbf{E} - 1)^3$ annihilates the sequence $\langle i^2 \cdot 1^i \rangle = \langle i^2 \rangle = \langle 1, 4, 9, 16, 25, \ldots \rangle$, since $p(i) = i^2$ is a polynomial of degree $n - 1 = 2$.

As a review, try to explain the following statements:

- $(\mathbf{E} - 1)$ annihilates any constant sequence $\langle \alpha \rangle$.
- $(\mathbf{E} - 1)^2$ annihilates any arithmetic sequence $\langle \alpha + \beta i \rangle$.
- $(\mathbf{E} - 1)^3$ annihilates any quadratic sequence $\langle \alpha + \beta i + \gamma i^2 \rangle$.
- $(\mathbf{E} - 3)(\mathbf{E} - 2)(\mathbf{E} - 1)$ annihilates any sequence $\langle \alpha + \beta 2^i + \gamma 3^i \rangle$.
- $(\mathbf{E} - 3)^2(\mathbf{E} - 2)(\mathbf{E} - 1)$ annihilates any sequence $\langle \alpha + \beta 2^i + \gamma 3^i + \delta i 3^i \rangle$.

R.2.4 Summary

In summary, we have learned several operators that act on sequences, as well as a few ways of combining operators.

Operator	Definition
Addition	$\langle a_i \rangle + \langle b_i \rangle = \langle a_i + b_i \rangle$
Subtraction	$\langle a_i \rangle + \langle b_i \rangle = \langle a_i + b_i \rangle$
Scalar multiplication	$c\langle a_i \rangle = \langle ca_i \rangle$
Shift	$\mathbf{E}\langle a_i \rangle = \langle a_{i+1} \rangle$
Composition of operators	$(\mathbf{X} + \mathbf{Y})\langle a_i \rangle = \mathbf{X}\langle a_i \rangle + \mathbf{Y}\langle a_i \rangle$
	$(\mathbf{X} - \mathbf{Y})\langle a_i \rangle = \mathbf{X}\langle a_i \rangle - \mathbf{Y}\langle a_i \rangle$
	$\mathbf{XY}\langle a_i \rangle = \mathbf{X}(\mathbf{Y}\langle a_i \rangle) = \mathbf{Y}(\mathbf{X}\langle a_i \rangle)$
k-fold shift	$\mathbf{E}^k\langle a_i \rangle = \langle a_{i+k} \rangle$

Notice that we have not defined a multiplication operator for two sequences. This is usually accomplished by *convolution*:

$$\langle a_i \rangle * \langle b_i \rangle = \left\langle \sum_{j=0}^{i} a_j b_{i-j} \right\rangle.$$

Fortunately, convolution is unnecessary for solving the recurrences we will see in this course.

We have also learned some things about annihilators, which can be summarized as follows:

Sequence	Annihilator
$\langle \alpha \rangle$	$\mathbf{E} - 1$
$\langle \alpha a^i \rangle$	$\mathbf{E} - a$
$\langle \alpha a^i + \beta b^i \rangle$	$(\mathbf{E} - a)(\mathbf{E} - b)$
$\langle \alpha_0 a_0^i + \alpha_1 a_1^i + \cdots + \alpha_n a_n^i \rangle$	$(\mathbf{E} - a_0)(\mathbf{E} - a_1) \cdots (\mathbf{E} - a_n)$
$\langle \alpha i + \beta \rangle$	$(\mathbf{E} - 1)^2$
$\langle (\alpha i + \beta) a^i \rangle$	$(\mathbf{E} - a)^2$
$\langle (\alpha i + \beta) a^i + \gamma b^i \rangle$	$(\mathbf{E} - a)^2 (\mathbf{E} - b)$
$\langle (\alpha_0 + \alpha_1 i + \cdots \alpha_{n-1} i^{n-1}) a^i \rangle$	$(\mathbf{E} - a)^n$

If $\mathbf{X}$ annihilates $\langle a_i \rangle$, then $\mathbf{X}$ also annihilates $c\langle a_i \rangle$ for any constant c.

If $\mathbf{X}$ annihilates $\langle a_i \rangle$ and $\mathbf{Y}$ annihilates $\langle b_i \rangle$, then $\mathbf{XY}$ annihilates $\langle a_i \rangle \pm \langle b_i \rangle$.

R.3 Solving Linear Recurrences

R.3.1 Homogeneous Recurrences

The general expressions in the annihilator box above are really the most important things to remember about annihilators because they help you to solve any recurrence for which you can write down an annihilator. The general method is:

1. Write down the annihilator for the recurrence
2. Factor the annihilator
3. Determine the sequence annihilated by each factor
4. Add these sequences together to form the generic solution
5. Solve for constants of the solution by using initial conditions

Example: Let's show the steps required to solve the following recurrence:

$$\begin{aligned} r_0 &= 1 \\ r_1 &= 5 \\ r_2 &= 17 \\ r_i &= 7r_{i-1} - 16r_{i-2} + 12r_{i-3} \end{aligned}$$

1. *Write down the annihilator.* Since $r_{i+3} - 7r_{i+2} + 16r_{i+1} - 12r_i = 0$, the annihilator is $\mathbf{E}^3 - 7\mathbf{E}^2 + 16\mathbf{E} - 12$.

2. *Factor the annihilator.* $\mathbf{E}^3 - 7\mathbf{E}^2 + 16\mathbf{E} - 12 = (\mathbf{E} - 2)^2(\mathbf{E} - 3)$.

3. *Determine sequences annihilated by each factor.* $(\mathbf{E} - 2)^2$ annihilates $\langle (\alpha i + \beta) 2^i \rangle$ for any constants α and β, and $(\mathbf{E} - 3)$ annihilates $\langle \gamma 3^i \rangle$ for any constant γ.

4. *Combine the sequences.* $(\mathbf{E}-2)^2(\mathbf{E}-3)$ annihilates $\langle(\alpha i+\beta)2^i+\gamma 3^i\rangle$ for any constants α, β, γ.

5. *Solve for the constants.* The base cases give us three equations in the three unknowns α, β, γ:

$$\begin{aligned} r_0 = 1 &= (\alpha \cdot 0+\beta)2^0+\gamma \cdot 3^0 = \beta+\gamma \\ r_1 = 5 &= (\alpha \cdot 1+\beta)2^1+\gamma \cdot 3^1 = 2\alpha+2\beta+3\gamma \\ r_2 = 17 &= (\alpha \cdot 2+\beta)2^2+\gamma \cdot 3^2 = 8\alpha+4\beta+9\gamma \end{aligned}$$

We can solve these equations to get $\alpha = 1$, $\beta = 0$, $\gamma = 1$. Thus, our final solution is $\boxed{r_i = i2^i + 3^i}$, which we can verify by induction.

R.3.2 Non-homogeneous Recurrences

A *height balanced tree* is a binary tree, where the heights of the two subtrees of the root differ by at most one, and both subtrees are also height balanced. To ground the recursive definition, the empty set is considered a height balanced tree of height -1, and a single node is a height balanced tree of height 0.

Let T_n be the smallest height-balanced tree of height n—how many nodes does T_n have? Well, one of the subtrees of T_n has height $n-1$ (since T_n has height n) and the other has height either $n-1$ or $n-2$ (since T_n is height-balanced and as small as possible). Since both subtrees are themselves height-balanced, the two subtrees must be T_{n-1} and T_{n-2}.

We have just derived the following recurrence for t_n, the number of nodes in the tree T_n:

$$\begin{aligned} t_{-1} &= 0 && \text{[the empty set]} \\ t_0 &= 1 && \text{[a single node]} \\ t_n &= t_{n-1}+t_{n-2}+1 \end{aligned}$$

The final '+1' is for the root of T_n.

We refer to the terms in the equation involving t_i's as the *homogeneous* terms and the rest as the *non-homogeneous* terms. (If there were no non-homogeneous terms, we would say that the recurrence itself is homogeneous.) We know that $\mathbf{E}^2-\mathbf{E}-1$ annihilates the homogeneous part $t_n = t_{n-1}+t_{n-2}$. Let us try applying this annihilator to the entire equation:

$$\begin{aligned} (\mathbf{E}^2-\mathbf{E}-1)\langle t_i\rangle &= \mathbf{E}^2\langle t_i\rangle - \mathbf{E}\langle a_i\rangle - 1\langle a_i\rangle \\ &= \langle t_{i+2}\rangle - \langle t_{i+1}\rangle - \langle t_i\rangle \\ &= \langle t_{i+2}-t_{i+1}-t_i\rangle \\ &= \langle 1\rangle \end{aligned}$$

The leftover sequence $\langle 1,1,1,\ldots\rangle$ is called the *residue*. To obtain the annihilator for the entire recurrence, we compose the annihilator for its homogeneous part with the annihilator of its residue. Since $\mathbf{E}-1$ annihilates $\langle 1\rangle$, it follows that $(\mathbf{E}^2-\mathbf{E}-1)(\mathbf{E}-1)$ annihilates $\langle t_n\rangle$. We can factor the annihilator into

$$(\mathbf{E}-\phi)(\mathbf{E}-\hat{\phi})(\mathbf{E}-1),$$

so our annihilator rules tell us that

$$t_n = \alpha\phi^n + \beta\hat{\phi}^n + \gamma$$

for some constants α, β, γ. We call this the *generic solution* to the recurrence. Different recurrences can have the same generic solution.

To solve for the unknown constants, we need three equations in three unknowns. Our base cases give us two equations, and we can get a third by examining the next nontrivial case $t_1 = 2$:

$$\begin{aligned} t_{-1} = 0 &= \alpha\phi^{-1} + \beta\hat{\phi}^{-1} + \gamma = \alpha/\phi + \beta/\hat{\phi} + \gamma \\ t_0 = 1 &= \quad \alpha\phi^0 + \beta\hat{\phi}^0 + \gamma \quad = \alpha + \beta + \gamma \\ t_1 = 2 &= \quad \alpha\phi^1 + \beta\hat{\phi}^1 + \gamma \quad = \alpha\phi + \beta\hat{\phi} + \gamma \end{aligned}$$

Solving these equations, we find that $\alpha = \frac{\sqrt{5}+2}{\sqrt{5}}$, $\beta = \frac{\sqrt{5}-2}{\sqrt{5}}$, and $\gamma = -1$. Thus,

$$\boxed{t_n = \frac{\sqrt{5}+2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{\sqrt{5}-2}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n - 1}$$

Here is the general method for non-homogeneous recurrences:

1. Write down the homogeneous annihilator, directly from the recurrence
$1\frac{1}{2}$. ‘Multiply’ by the annihilator for the residue
2. Factor the annihilator
3. Determine what sequence each factor annihilates
4. Add these sequences together to form the generic solution
5. Solve for constants of the solution by using initial conditions

R.3.3 Some more examples

In each example below, we use the base cases $a_0 = 0$ and $a_1 = 1$.

- $\boldsymbol{a_n = a_{n-1} + a_{n-2} + 2}$
  - The homogeneous annihilator is $\mathbf{E}^2 - \mathbf{E} - 1$.
  - The residue is the constant sequence $\langle 2, 2, 2, \ldots \rangle$, which is annihilated by $\mathbf{E} - 1$.
  - Thus, the annihilator is $(\mathbf{E}^2 - \mathbf{E} - 1)(\mathbf{E} - 1)$.
  - The annihilator factors into $(\mathbf{E} - \phi)(\mathbf{E} - \hat{\phi})(\mathbf{E} - 1)$.
  - Thus, the generic solution is $a_n = \alpha\phi^n + \beta\hat{\phi}^n + \gamma$.
  - The constants α, β, γ satisfy the equations

$$\begin{aligned} a_0 &= 0 = \alpha + \beta + \gamma \\ a_1 &= 1 = \alpha\phi + \beta\hat{\phi} + \gamma \\ a_2 &= 3 = \alpha\phi^2 + \beta\hat{\phi}^2 + \gamma \end{aligned}$$

  - Solving the equations gives us $\alpha = \frac{\sqrt{5}+2}{\sqrt{5}}$, $\beta = \frac{\sqrt{5}-2}{\sqrt{5}}$, and $\gamma = -2$
  - So the final solution is $\boxed{a_n = \frac{\sqrt{5}+2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{\sqrt{5}-2}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n - 2}$

(In the remaining examples, I won’t explicitly enumerate the steps like this.)

- $\boldsymbol{a_n = a_{n-1} + a_{n-2} + 3}$

  The homogeneous annihilator $(\mathbf{E}^2 - \mathbf{E} - 1)$ leaves a constant residue $\langle 3, 3, 3, \ldots \rangle$, so the annihilator is $(\mathbf{E}^2 - \mathbf{E} - 1)(\mathbf{E} - 1)$, and the generic solution is $a_n = \alpha\phi^n + \beta\hat{\phi}^n + \gamma$. Solving the equations

$$\begin{aligned} a_0 &= 0 = \alpha + \beta + \gamma \\ a_1 &= 1 = \alpha\phi + \beta\hat{\phi} + \gamma \\ a_2 &= 4 = \alpha\phi^2 + \beta\hat{\phi}^2 + \gamma \end{aligned}$$

  gives us the final solution $\boxed{a_n = \frac{\sqrt{5}+3}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{\sqrt{5}-3}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n - 3}$

- $\boldsymbol{a_n = a_{n-1} + a_{n-2} + 2^n}$

  The homogeneous annihilator $(\mathbf{E}^2 - \mathbf{E} - 1)$ leaves an exponential residue $\langle 4, 8, 16, 32, \ldots \rangle = \langle 2^{i+2} \rangle$, which is annihilated by $\mathbf{E} - 2$. Thus, the annihilator is $(\mathbf{E}^2 - \mathbf{E} - 1)(\mathbf{E} - 2)$, and the generic solution is $a_n = \alpha\phi^n + \beta\hat{\phi}^n + \gamma 2^n$. The constants α, β, γ satisfy the following equations:

$$\begin{aligned} a_0 &= 0 = \alpha + \beta + \gamma \\ a_1 &= 1 = \alpha\phi + \beta\hat{\phi} + 2\gamma \\ a_2 &= 5 = \alpha\phi^2 + \beta\hat{\phi}^2 + 4\gamma \end{aligned}$$

- $\boldsymbol{a_n = a_{n-1} + a_{n-2} + n}$

  The homogeneous annihilator $(\mathbf{E}^2 - \mathbf{E} - 1)$ leaves a linear residue $\langle 2, 3, 4, 5 \ldots \rangle = \langle i + 2 \rangle$, which is annihilated by $(\mathbf{E} - 1)^2$. Thus, the annihilator is $(\mathbf{E}^2 - \mathbf{E} - 1)(\mathbf{E} - 1)^2$, and the generic solution is $a_n = \alpha\phi^n + \beta\hat{\phi}^n + \gamma + \delta n$. The constants $\alpha, \beta, \gamma, \delta$ satisfy the following equations:

$$\begin{aligned} a_0 &= 0 = \alpha + \beta + \gamma \\ a_1 &= 1 = \alpha\phi + \beta\hat{\phi} + \gamma + \delta \\ a_2 &= 3 = \alpha\phi^2 + \beta\hat{\phi}^2 + \gamma + 2\delta \\ a_3 &= 7 = \alpha\phi^3 + \beta\hat{\phi}^3 + \gamma + 3\delta \end{aligned}$$

- $\boldsymbol{a_n = a_{n-1} + a_{n-2} + n^2}$

  The homogeneous annihilator $(\mathbf{E}^2 - \mathbf{E} - 1)$ leaves a quadratic residue $\langle 4, 9, 16, 25 \ldots \rangle = \langle (i+2)^2 \rangle$, which is annihilated by $(\mathbf{E} - 1)^3$. Thus, the annihilator is $(\mathbf{E}^2 - \mathbf{E} - 1)(\mathbf{E} - 1)^3$, and the generic solution is $a_n = \alpha\phi^n + \beta\hat{\phi}^n + \gamma + \delta n + \varepsilon n^2$. The constants $\alpha, \beta, \gamma, \delta, \varepsilon$ satisfy the following equations:

$$\begin{aligned} a_0 &= 0 = \alpha + \beta + \gamma \\ a_1 &= 1 = \alpha\phi + \beta\hat{\phi} + \gamma + \delta + \varepsilon \\ a_2 &= 5 = \alpha\phi^2 + \beta\hat{\phi}^2 + \gamma + 2\delta + 4\varepsilon \\ a_3 &= 15 = \alpha\phi^3 + \beta\hat{\phi}^3 + \gamma + 3\delta + 9\varepsilon \\ a_4 &= 36 = \alpha\phi^4 + \beta\hat{\phi}^4 + \gamma + 4\delta + 16\varepsilon \end{aligned}$$

- $\boldsymbol{a_n = a_{n-1} + a_{n-2} + n^2 - 2^n}$

  The homogeneous annihilator $(\mathbf{E}^2 - \mathbf{E} - 1)$ leaves the residue $\langle (i+2)^2 - 2^{i-2} \rangle$. The quadratic part of the residue is annihilated by $(\mathbf{E}-1)^3$, and the exponential part is annihilated by $(\mathbf{E}-2)$. Thus, the annihilator for the whole recurrence is $(\mathbf{E}^2 - \mathbf{E} - 1)(\mathbf{E}-1)^3(\mathbf{E}-2)$, and so the generic solution is $a_n = \alpha\phi^n + \beta\hat{\phi}^n + \gamma + \delta n + \varepsilon n^2 + \eta 2^i$. The constants $\alpha, \beta, \gamma, \delta, \varepsilon, \eta$ satisfy a system of six equations in six unknowns determined by $a_0, a_1, \ldots, a_5$.

- $\boldsymbol{a_n = a_{n-1} + a_{n-2} + \phi^n}$

  The annihilator is $(\mathbf{E}^2 - \mathbf{E} - 1)(\mathbf{E} - \phi) = (\mathbf{E} - \phi)^2(\mathbf{E} - \hat{\phi})$, so the generic solution is $a_n = \alpha\phi^n + \beta n\phi^n + \gamma\hat{\phi}^n$. (Other recurrence solving methods will have a “interference” problem with this equation, while the operator method does not.)

Our method does not work on recurrences like $a_n = a_{n-1} + \frac{1}{n}$ or $a_n = a_{n-1} + \lg n$, because the functions $\frac{1}{n}$ and $\lg n$ do not have annihilators. Our tool, as it stands, is limited to linear recurrences.

R.4 Divide and Conquer Recurrences

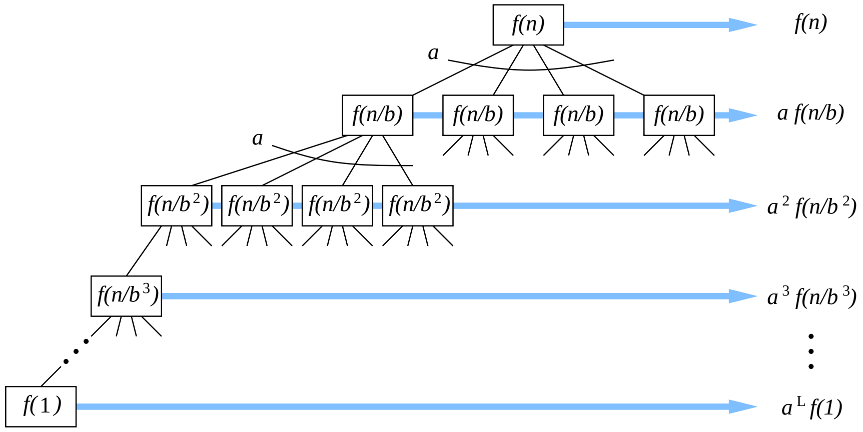
Divide and conquer algorithms often give us running-time recurrences of the form

$$T(n) = a\,T(n/b) + f(n) \tag{1}$$

where a and b are constants and $f(n)$ is some other function. The so-called ‘Master Theorem’ gives us a general method for solving such recurrences when $f(n)$ is a simple polynomial.

Unfortunately, the Master Theorem doesn’t work for all functions $f(n)$, and many useful recurrences don’t look like (**??**) at all. Fortunately, there’s a more general technique to solve most divide-and-conquer recurrences, even if they don’t have this form. This technique is used to *prove* the Master Theorem, so if you remember this technique, you can forget the Master Theorem entirely (which is what I did). Throw off your chains!

I’ll illustrate the technique using the generic recurrence (**??**). We start by drawing a *recursion tree*. The root of the recursion tree is a box containing the value $f(n)$, it has a children, each of which is the root of a recursion tree for $T(n/b)$. Equivalently, a recursion tree is a complete a-ary tree where each node at depth i contains the value $a^i f(n/b^i)$. The recursion stops when we get to the base case(s) of the recurrence. Since we’re looking for asymptotic bounds, it turns out not to matter much what we use for the base case; for purposes of illustration, I’ll assume that $T(1) = f(1)$.



A recursion tree for the recurrence $T(n) = a\,T(n/b) + f(n)$

Now $T(n)$ is just the sum of all values stored in the tree. Assuming that each level of the tree is full, we have

$$T(n) = f(n) + a\,f(n/b) + a^2\,f(n/b^2) + \cdots + a^i\,f(n/b^i) + \cdots + a^L\,f(n/b^L)$$

where L is the depth of the recursion tree. We easily see that $L = \log_b n$, since $n/b^L = 1$. Since $f(1) = \Theta(1)$, the last non-zero term in the summation is $\Theta(a^L) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$.

Now we can easily state and prove the Master Theorem, in a slightly different form than it's usually stated.

The Master Theorem. The recurrence $T(n) = aT(n/b) + f(n)$ can be solved as follows.

- If $a\,f(n/b) = \kappa\,f(n)$ for some constant $\kappa < 1$, then $T(n) = \Theta(f(n))$.
- If $a\,f(n/b) = K\,f(n)$ for some constant $K > 1$, then $T(n) = \Theta(n^{\log_b a})$.
- If $a\,f(n/b) = f(n)$, then $T(n) = \Theta(f(n)\log_b n)$.
- If none of these three cases apply, you're on your own.

Proof: If $f(n)$ is a *constant factor larger* than $a\,f(b/n)$, then by induction, the sum is a descending geometric series. The sum of any geometric series is a constant times its largest term. In this case, the largest term is the first term $f(n)$.

If $f(n)$ is a *constant factor smaller* than $a\,f(b/n)$, then by induction, the sum is an ascending geometric series. The sum of any geometric series is a constant times its largest term. In this case, this is the last term, which by our earlier argument is $\Theta(n^{\log_b a})$.

Finally, if $a\,f(b/n) = f(n)$, then by induction, each of the $L+1$ terms in the summation is equal to $f(n)$. □

Here are a few canonical examples of the Master Theorem in action:

- **Randomized selection:** $\boldsymbol{T(n) = T(3n/4) + n}$

  Here $a\,f(n/b) = 3n/4$ is smaller than $f(n) = n$ by a factor of $4/3$, so $\boxed{T(n) = \Theta(n)}$

- **Karatsuba's multiplication algorithm:** $\boldsymbol{T(n) = 3T(n/2) + n}$

  Here $a\,f(n/b) = 3n/2$ is bigger than $f(n) = n$ by a factor of $3/2$, so $\boxed{T(n) = \Theta(n^{\log_2 3})}$

- **Mergesort:** $\boldsymbol{T(n) = 2T(n/2) + n}$

  Here $a\,f(n/b) = f(n)$, so $\boxed{T(n) = \Theta(n \log n)}$

- $\boldsymbol{T(n) = 4T(n/2) + n\lg n}$

  In this case, we have $a\,f(n/b) = 2n\lg n - 2n$, which is not quite twice $f(n) = n\lg n$. However, for sufficiently large n (which is all we care about with asymptotic bounds) we have $2f(n) > a\,f(n/b) > 1.9f(n)$. Since the level sums are bounded both above and below by ascending geometric series, the solution is $T(n) = \Theta(n^{\log_2 4}) = \boxed{\Theta(n^2)}$. (This trick will *not* work in the second or third cases of the Master Theorem!)

Using the same recursion-tree technique, we can also solve recurrences where the Master Theorem doesn't apply.

- $\boldsymbol{T(n) = 2T(n/2) + n/\lg n}$

  We can't apply the Master Theorem here, because $a\,f(n/b) = n/(\lg n - 1)$ isn't equal to $f(n) = n/\lg n$, but the difference isn't a constant factor. So we need to compute each of the level sums and compute their total in some other way. It's not hard to see that the sum of all the nodes in the ith level is $n/(\lg n - i)$. In particular, this means the depth of the tree is at most $\lg n - 1$.

  $$T(n) = \sum_{i=0}^{\lg n-1} \frac{n}{\lg n - i} = \sum_{j=1}^{\lg n} \frac{n}{j} = nH_{\lg n} = \boxed{\Theta(n \lg\lg n)}$$

- **Randomized quicksort:** $\boldsymbol{T(n) = T(3n/4) + T(n/4) + n}$

  In this case, nodes in the same level of the recursion tree have different values. This makes the tree lopsided; different leaves are at different levels. However, it's not to hard to see that the nodes in any *complete* level (*i.e.*, above any of the leaves) sum to n, so this is like the last case of the Master Theorem, and that every leaf has depth between $\log_4 n$ and $\log_{4/3} n$. To derive an upper bound, we overestimate $T(n)$ by ignoring the base cases and extending the tree downward to the level of the *deepest* leaf. Similarly, to derive a lower bound, we overestimate $T(n)$ by counting only nodes in the tree up to the level of the *shallowest* leaf. These observations give us the upper and lower bounds $n \log_4 n \le T(n) \le n \log_{4/3} n$. Since these bound differ by only a constant factor, we have $\boxed{T(n) = \Theta(n \log n)}$.

- **Deterministic selection:** $\boldsymbol{T(n) = T(n/5) + T(7n/10) + n}$

  Again, we have a lopsided recursion tree. If we look only at complete levels of the tree, we find that the level sums form a descending geometric series $T(n) = n + 9n/10 + 81n/100 + \cdots$, so this is like the first case of the master theorem. We can get an upper bound by ignoring the base cases entirely and growing the tree out to infinity, and we can get a lower bound by only counting nodes in complete levels. Either way, the geometric series is dominated by its largest term, so $\boxed{T(n) = \Theta(n)}$.

- $\boldsymbol{T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n}$

  In this case, we have a complete recursion tree, but the *degree* of the nodes is no longer constant, so we have to be a bit more careful. It's not hard to see that the nodes in any level sum to n, so this is like the third case of the Master Theorem. The depth L satisfies the identity $n^{2^{-L}} = 2$ (we can't get all the way down to 1 by taking square roots), so $L = \lg\lg n$ and $\boxed{T(n) = \Theta(n \lg\lg n)}$.

- $\boldsymbol{T(n) = 4\sqrt{n} \cdot T(\sqrt{n}) + n}$

  We still have at most $\lg\lg n$ levels, but now the nodes in level i sum to $4^i n$. We have an increasing geometric series of level sums, like the second Master case, so $T(n)$ is dominated by the sum over the deepest level: $T(n) = \Theta(4^{\lg\lg n} n) = \boxed{\Theta(n \log^2 n)}$

R.5 Transforming Recurrences

R.5.1 An analysis of mergesort: domain transformation

Previously we gave the recurrence for mergesort as $T(n) = 2T(n/2) + n$, and obtained the solution $T(n) = \Theta(n \log n)$ using the Master Theorem (or the recursion tree method if you, like me, can't remember the Master Theorem). This is fine is n is a power of two, but for other values values of n, this recurrence is incorrect. When n is odd, then the recurrence calls for us to sort a fractional number of elements! Worse yet, if n is not a power of two, we will *never* reach the base case $T(1) = 0$.

To get a recurrence that's valid for *all* integers n, we need to carefully add ceilings and floors:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n.$$

We have almost no hope of getting an exact solution here; the floors and ceilings will eventually kill us. So instead, let's just try to get a tight asymptotic upper bound for $T(n)$ using a technique called *domain transformation*. A domain transformation rewrites a function $T(n)$ with a difficult recurrence as a nested function $S(f(n))$, where $f(n)$ is a simple function and $S()$ has an easier recurrence.

First we overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \le 2T\big(\lceil n/2 \rceil\big) + n \le 2T(n/2 + 1) + n.$$

Now we define a new function $S(n) = T(n + \alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-ready recurrence $S(n) \le 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n + \alpha)$:

$$\begin{aligned} S(n) \le 2S(n/2) + O(n) \quad &\Longrightarrow \quad T(n+\alpha) \le 2T(n/2+\alpha) + O(n) \\ T(n) \le 2T(n/2+1) + n \quad &\Longrightarrow \quad T(n+\alpha) \le 2T((n+\alpha)/2+1) + n + \alpha \end{aligned}$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 + 1$, which implies that $\alpha = 2$. The Master Theorem now tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n).$$

A similar argument gives a matching lower bound $T(n) = \Omega(n \log n)$. So $\boxed{T(n) = \Theta(n \log n)}$ after all, just as though we had ignored the floors and ceilings from the beginning!

Domain transformations are useful for removing floors, ceilings, and lower order terms from the arguments of any recurrence that otherwise looks like it ought to fit either the Master Theorem or the recursion tree method. But now that we know this, we don't need to bother grinding through the actual gory details!

R.5.2 A less trivial example

There is a data structure in computational geometry called *ham-sandwich trees*, where the cost of doing a certain search operation obeys the recurrence $\boldsymbol{T(n) = T(n/2) + T(n/4) + 1}$. This doesn't fit the Master theorem, because the two subproblems have different sizes, and using the recursion tree method only gives us the loose bounds $\sqrt{n} \ll T(n) \ll n$.

Domain transformations save the day. If we define the new function $t(k) = T(2^k)$, we have a new recurrence

$$t(k) = t(k-1) + t(k-2) + 1$$

which should immediately remind you of Fibonacci numbers. Sure enough, after a bit of work, the annihilator method gives us the solution $t(k) = \Theta(\phi^k)$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio. This implies that

$$T(n) = t(\lg n) = \Theta(\phi^{\lg n}) = \boxed{\Theta(n^{\lg \phi})} \approx \Theta(n^{0.69424}).$$

It's possible to solve this recurrence without domain transformations and annihilators—in fact, the inventors of ham-sandwich trees did so—but it's much more difficult.

R.5.3 Secondary recurrences

Consider the recurrence $\boldsymbol{T(n) = 2T(\frac{n}{3} - 1) + n}$ with the base case $T(1) = 1$. We already know how to use domain transformations to get the tight asymptotic bound $T(n) = \Theta(n)$, but how would we we obtain an *exact* solution?

First we need to figure out how the parameter n changes as we get deeper and deeper into the recurrence. For this we use a *secondary recurrence*. We define a sequence n_i so that

$$T(n_i) = 2T(n_{i-1}) + n_i,$$

So n_i is the argument of $T()$ when we are i recursion steps away from the base case $n_0 = 1$. The original recurrence gives us the following secondary recurrence for n_i:

$$n_{i-1} = \frac{n_i}{3} - 1 \implies n_i = 3n_{i-3} + 3.$$

The annihilator for this recurrence is $(\mathbf{E}-1)(\mathbf{E}-3)$, so the generic solution is $n_i = \alpha 3^i + \beta$. Plugging in the base cases $n_0 = 1$ and $n_1 = 6$, we get the exact solution

$$n_i = \frac{5}{2} \cdot 3^i - \frac{3}{2}.$$

Notice that our original function $T(n)$ is only well-defined if $n = n_i$ for some integer $i \geq 0$.

Now to solve the original recurrence, we do a range transformation. If we set $t_i = T(n_i)$, we have the recurrence $t_i = 2t_{i-1} + \frac{5}{2} \cdot 3^i - \frac{3}{2}$, which by now we can solve using the annihilator method. The annihilator of the recurrence is $(\mathbf{E}-2)(\mathbf{E}-3)(\mathbf{E}-1)$, so the generic solution is $\alpha' 3^i + \beta' 2^i + \gamma'$. Plugging in the base cases $t_0 = 1$, $t_1 = 8$, $t_2 = 37$, we get the exact solution

$$t_i = \frac{15}{2} \cdot 3^i - 8 \cdot 2^i + \frac{3}{2}$$

Finally, we need to substitute to get a solution for the original recurrence in terms of n, by inverting the solution of the secondary recurrence. If $n = n_i = \frac{5}{2} \cdot 3^i - \frac{3}{2}$, then (after a little algebra) we have

$$i = \log_3\left(\frac{2}{5}n + \frac{3}{5}\right).$$

Substituting this into the expression for t_i, we get our exact, closed-form solution.

$$\begin{aligned}
T(n) &= \frac{15}{2} \cdot 3^i - 8 \cdot 2^i + \frac{3}{2} \\
&= \frac{15}{2} \cdot 3^{\left(\frac{2}{5}n + \frac{3}{5}\right)} - 8 \cdot 2^{\log_3\left(\frac{2}{5}n + \frac{3}{5}\right)} + \frac{3}{2} \\
&= \frac{15}{2}\left(\frac{2}{5}n + \frac{3}{5}\right) - 8 \cdot \left(\frac{2}{5}n + \frac{3}{5}\right)^{\log_3 2} + \frac{3}{2} \\
&= 3n - 8 \cdot \left(\frac{2}{5}n + \frac{3}{5}\right)^{\log_3 2} + 6
\end{aligned}$$

Isn't that special? Now you know why we stick to asymptotic bounds for most recurrences.

R.6 The Ultimate Method: Guess and Confirm

Ultimately, there is one failsafe method to solve *any* recurrence:

Guess the answer, and then prove it correct by induction.

The annihilator method, the recursion-tree method, and transformations are good ways to generate guesses that are guaranteed to be correct, provided you use them correctly. But if you're faced with a recurrence that doesn't seem to fit any of these methods, or if you've forgotten how those techniques work, don't despair! If you guess a closed-form solution and then try to verify your guess inductively, usually either the proof succeeds and you're done, or the proof fails in a way that lets you refine your guess. Where you get your initial guess is utterly irrelevant[2]—from a classmate, from a textbook, on the web, from the answer to a different problem, scrawled on a bathroom wall in Siebel, dictated by the machine elves, whatever. If you can prove that the answer is correct, then it's correct!

R.6.1 Tower of Hanoi

The classical Tower of Hanoi problem gives us the recurrence $\boldsymbol{T(n) = 2T(n-1) + 1}$ with base case $\boldsymbol{T(0) = 0}$. Just looking at the recurrence we can guess that $T(n)$ is something like 2^n. If we write out the first few values of $T(n)$ all the values are one less than a power of two.

$$T(0) = 0,\ T(1) = 1,\ T(2) = 3,\ T(3) = 7,\ T(4) = 15,\ T(5) = 31,\ T(6) = 63, \dots,$$

It looks like $\boxed{\boldsymbol{T(n) = 2^n - 1}}$ might be the right answer. Let's check.

$$\begin{aligned} T(0) &= 0 = 2^0 - 1 \quad \checkmark \\ T(n) &= 2T(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 && \text{[induction hypothesis]} \\ &= 2^n - 1 \quad \checkmark && \text{[algebra]} \end{aligned}$$

We were right!

R.6.2 Fibonacci numbers

Let's try a less trivial example: the Fibonacci numbers $\boldsymbol{F_n = F_{n-1} + F_{n-2}}$ with base cases $\boldsymbol{F_0 = 0}$ and $\boldsymbol{F_1 = 1}$. There is no obvious pattern (besides the obvious one) in the first several values, but we can reasonably guess that F_n is exponential in n. Let's try to prove that $F_n \le a \cdot c^n$ for some constants $a > 0$ and $c > 1$ and see how far we get.

$$F_n \ = \ F_{n-1} + F_{n-2} \ \le \ a \cdot c^{n-1} + a \cdot c^{n-2} \ \le \ a \cdot c^n \ ???$$

The last inequality is satisfied if $c^n \ge c^{n-1} + c^{n-2}$, or more simply, if $c^2 - c - 1 \ge 0$. The smallest value of c that works is $\phi = (1 + \sqrt{5})/2 \approx 1.618034$; the other root of the quadratic equation is negative, so we can ignore it.

[2] ...except of course during exams, where you aren't supposed to use *any* outside sources

So we have *most* of an inductive proof that $F_n \le a \cdot \phi^n$ for *any* constant a. All that we're missing are the base cases, which (we can easily guess) must determine the value of the coefficient a. We quickly compute

$$\frac{F_0}{\phi^0} = 0 \quad \text{and} \quad \frac{F_1}{\phi^1} = \frac{1}{\phi} \approx 0.618034 > 0,$$

so the base cases of our induction proof are correct as long as $a \ge 1/\phi$. It follows that $\boldsymbol{F_n \le \phi^{n-1}}$ for all $n \ge 0$.

What about a matching lower bound? Well, the same inductive proof implies that $F_n \ge b \cdot \phi^n$ for some constant b, but the only value of b that works for *all* n is the trivial $b = 0$. We could try to find some lower-order term that makes the base case non-trivial, but an easier approach is to recall that $\Omega()$ bounds only have to work for sufficiently large n. So let's ignore the trivial base case $F_0 = 0$ and assume that $F_2 = 1$ is a base case instead. Some more calculation gives us

$$\frac{F_2}{\phi^2} = \frac{1}{\phi^2} \approx 0.381966 < \frac{1}{\phi}.$$

Thus, the new base cases of our induction proof are correct as long as $b \le 1/\phi^2$, which implies that $\boldsymbol{F_n \ge \phi^{n-2}}$ for all $n \ge 1$.

Putting the upper and lower bounds together, we correctly conclude that $\boxed{\boldsymbol{F_n = \Theta(\phi^n)}}$. It *is* possible to get a more exact solution by speculatively refining and conforming our current bounds, but it's not easy; you're better off just using annihilators.

R.6.3 A divide-and-conquer example

Consider the divide-and-conquer recurrence $\boldsymbol{T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n}$. It doesn't fit into the form required by the Master Theorem, but it still sort of resembles the Mergesort recurrence—the total size of the subproblems at the first level of recursion is n—so let's *guess* that $T(n) = O(n \log n)$, and then try to prove that our guess is correct. Specifically, let's conjecture that $T(n) \le a\,n \lg n$ for all sufficiently large n and some constant a to be determined later:

$$\begin{aligned} T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\le \sqrt{n} \cdot a\,\sqrt{n} \lg \sqrt{n} + n && \text{[induction hypothesis]} \\ &= (a/2) n \lg n + n && \text{[algebra]} \\ &\le an \lg n \checkmark \end{aligned}$$

The last inequality assumes only that $1 \le (a/2) \log n$,or equivalently, that $n \ge 2^{2/a}$. In other words, the induction proof is correct if n is sufficiently large. So we were right!

But before you break out the champaign, what about the multiplicative constant a? The proof worked for *any* constant a, no matter how small. This strongly suggests that our upper bound $T(n) = O(n \log n)$ is not tight. Indeed, if we try to prove a matching lower bound $T(n) \ge b\,n \log n$ for sufficiently large n, we run into trouble.

$$\begin{aligned} T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\ge \sqrt{n} \cdot b\,\sqrt{n} \log \sqrt{n} + n && \text{[induction hypothesis]} \\ &= (b/2) n \log n + n \\ &\not\ge bn \log n \end{aligned}$$

The last inequality would be correct only if $1 > (b/2)\log n$, but that inequality is false for large values of n, no matter which constant b we choose. Okay, so $\Theta(n \log n)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n \geq n \checkmark$$

But an inductive proof of the lower bound fails.

$$\begin{aligned} T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\leq \sqrt{n} \cdot a\sqrt{n} + n && \text{[induction hypothesis]} \\ &= (a+1)n && \text{[algebra]} \\ &\not\leq an \end{aligned}$$

Hmmm. So what's bigger than n and smaller than $n \lg n$? How about $n\sqrt{\lg n}$?

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\leq \sqrt{n} \cdot a\sqrt{n}\sqrt{\lg \sqrt{n}} + n && \text{[induction hypothesis]} \\ &= (a/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\ &\leq a\, n\sqrt{\lg n} \quad \text{for large enough } n \checkmark \end{aligned}$$

Okay, the upper bound checks out; how about the lower bound?

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\geq \sqrt{n} \cdot b\sqrt{n}\sqrt{\lg \sqrt{n}} + n && \text{[induction hypothesis]} \\ &= (b/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\ &\not\geq b\, n\sqrt{\lg n} \end{aligned}$$

No, the last step doesn't work. So $\Theta(n\sqrt{\lg n})$ doesn't work. Hmmm...what else is between n and $n \lg n$? How about $n \lg \lg n$?

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\leq \sqrt{n} \cdot a\sqrt{n} \lg\lg \sqrt{n} + n && \text{[induction hypothesis]} \\ &= a\, n \lg\lg n - a\, n + n && \text{[algebra]} \\ &\leq a\, n \lg\lg n \qquad \textbf{if } \boldsymbol{a \geq 1} \checkmark \end{aligned}$$

Hey look at that! For once, our upper bound proof requires a constraint on the hidden constant a. This is an good indication that we've found the right answer. Let's try the lower bound:

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\geq \sqrt{n} \cdot b\sqrt{n} \lg\lg \sqrt{n} + n && \text{[induction hypothesis]} \\ &= b\, n \lg\lg n - b\, n + n && \text{[algebra]} \\ &\geq b\, n \lg\lg n \qquad \textbf{if } \boldsymbol{b \leq 1} \checkmark \end{aligned}$$

Hey, it worked! We have most of an inductive proof that $T(n) \leq an \lg\lg n$ for any $a \geq 1$ and most of an inductive proof that $T(n) \geq bn \lg\lg n$ for any $b \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $\boxed{\boldsymbol{T(n) = \Theta(n \log\log n)}}$.

R.7 References

Methods for solving recurrences by annihilators, domain transformations, and secondary recurrences are nicely described in George Lueker's paper "Some techniques for solving recurrences" (*ACM Computing Surveys* 12(4):419-436, 1980). The Master Theorem is presented in Chapters 4.3 and 4.4 of CLR. Sections 1–3 and 5 of this handout were initially written by Ed Reingold and Ari Trachtenberg and substantially revised by Jeff Erickson. Sections 4 and 6 are entirely Jeff's fault.

Our life is frittered away by detail.
Simplify, simplify.

— Henry David Thoreau

When you come to a fork in the road, take it.

— Yogi Berra

Ha ha! Cookies on dowels!

— Phil Ken Sebben [played by Stephen Colbert]
"Harvey Birdman, Attorney at Law"

1 Recursion

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem (or set of problems) Y means to write an algorithm for X, using an algorithm or Y as a subroutine or black box. For example, the congressional apportionment algorithm described in the previous lecture reduces the problem of apportioning Congress to the problem of maintaining a priority queue under the operations INSERT and EXTRACTMAX. In this class, we'll generally treat primitive data structures like arrays, linked lists, stacks, queues, hash tables, binary search trees, and priority queues as black boxes, adding them to the basic vocabulary of our model of computation. When we design algorithms, we may not know—and we should not care—how these basic building blocks will actually be implemented.

In some sense, *every* algorithm is simply a reduction to some underlying model of computation. Whenever you write a C program, you're really just reducing some problem to the "simpler" problems of compiling C, allocating memory, formatting output, scheduling jobs, and so forth. Even machine language programs are just reductions to the hardware-implemented problems of retrieving and storing memory and performing basic arithmetic. The underlying hardware implementation reduces those problems to timed Boolean logic; low-level hardware design reduces Boolean logic to basic physical devices such as wires and transistors; and the laws of physics reduce wires and transistors to underlying mathematical principles. At least, that's what the people who actually build hardware have to assume.[1]

A particularly powerful kind of reduction is *recursion*, which can be defined loosely as a reducing a problem to one or more **simpler instances of the *same* problem**. If the self-reference is confusing, it's useful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions. Your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will magically take care of the rest.[2]

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly, namely, that there is no infinite sequence of reductions to 'simpler' and 'simpler'

[1] The situation is exactly analogous to that of mathematical proof. Normally, when we prove a theorem, we implicitly rely on a several earlier results. These eventually trace back to axioms, definitions, and the rules of logic, but it is extremely rare for any proof to be expanded to pure symbol manipulation. Even when proofs are written in excruciating Bourbakian formal detail, the accuracy of the proof depends on the consistency of the formal system in which the proof is written. This consistency is simply taken for granted. In some cases (for example, first-order logic and the first-order theory of the reals) it is possible to prove consistency, but this consistency proof necessarily (thanks be to Gödel) relies on some different formal system (usually some extension of Zermelo-Fraenkel set theory), which is itself assumed to be consistent. Yes, it's turtles all the way down.

[2] I used to refer to 'elves' instead of the Recursion Fairy, referring to the traditional fairy tale in which an old shoemaker repeatedly leaves his work half-finished when he goes to bed, only to discover upon waking that elves have finished his work while he slept.

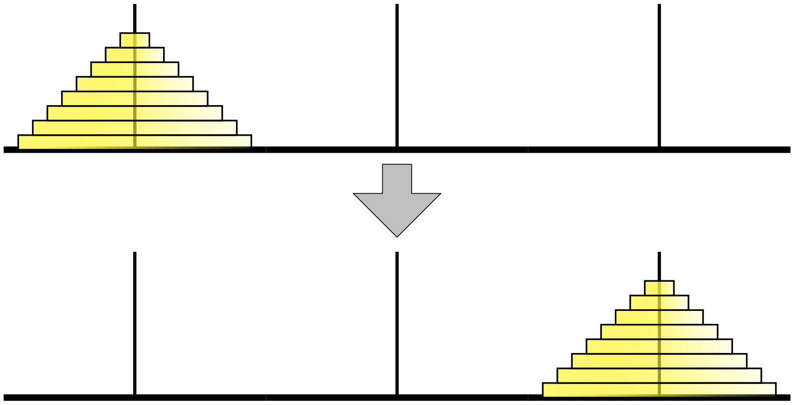
subproblems. Eventually, the recursive reductions must stop with an elementary *base case* that is solved by some other method; otherwise, the algorithm will never terminate. This finiteness condition is *usually* easy to satisfy, but we should always be wary of 'obvious' recursive algorithms that actually recurse forever.

1.1 Tower of Hanoi

The Tower of Hanoi puzzle was first published by the French mathematician François Éduoard Anatole Lucas in 1883, under the pseudonym 'N. Claus (de Siam)' (an anagram of 'Lucas d'Amiens'). The following year, the French scientist Henri de Parville described the puzzle with the following remarkable story:[3]

> In the great temple at Benares beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

Of course, being good computer scientists, we read this story and immediately substitute n for the hardwired constant sixty-four.[4] How can we move a tower of n disks from one needle to another, using a third needles as an occasional placeholder, never placing any disk on top of a smaller disk?



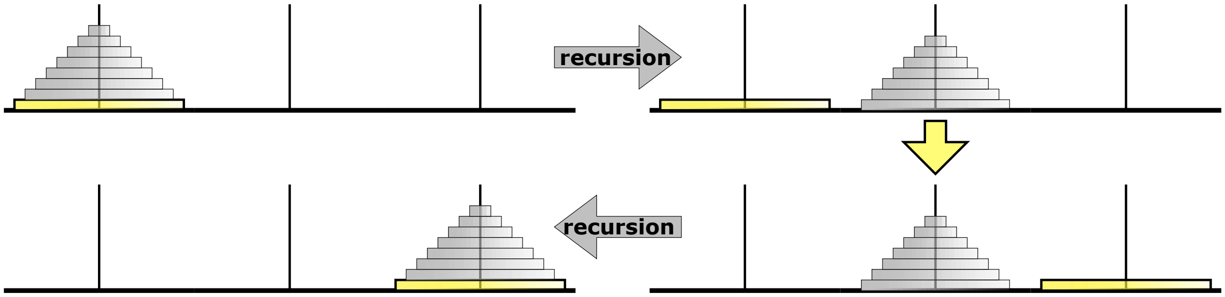
The Tower of Hanoi puzzle

The trick to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle all at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are covering it; we have to move those $n-1$ disks to the third needle before we can move the nth disk. And then after we move the nth disk, we have to move those $n-1$ disks back on top of it. So now all we have to figure out is how to...

[3] This English translation is from W. W. Rouse Ball and H. S. M. Coxeter's book *Mathematical Recreations and Essays*.

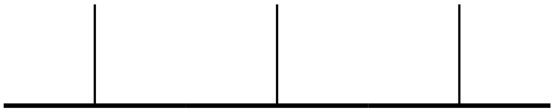
[4] Recognizing that the underlying mathematical abstraction would be unchanged, we may also freely use 'cookies' and 'dowels' instead of 'discs' and 'needles'. Ha ha... underlying!

STOP!! That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy (or, to carry the original story further, to the junior monks at the temple).



The Tower of Hanoi algorithm; ignore everything but the bottom disk

Our algorithm does make one subtle but important assumption: *there is a largest disk*. In other words, our recursive algorithm works for any $n \geq 1$, but it breaks down when $n = 0$. We must handle that base case directly. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one needle to another.



The base case for the Tower of Hanoi algorithm; there is no bottom disk

While it's tempting to think about how all those smaller disks get moved—in other words, what happens when the recursion is unfolded—it's not necessary. In fact, for more complicated problems, opening up the recursion is a distraction. Our *only* task is to reduce the problem to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our algorithm is trivially correct when $n = 0$. For any $n \geq 1$, the Recursion Fairy correctly moves (or more formally, the inductive hypothesis implies that our algorithm correctly moves) the top $n-1$ disks, so our algorithm is clearly correct.

Here's the recursive Hanoi algorithm in more typical pseudocode.

```
HANOI($n, src, dst, tmp$):
  if $n > 0$
      HANOI($n, src, tmp, dst$)
      move disk $n$ from $src$ to $dst$
      HANOI($n, tmp, dst, src$)
```

Let $T(n)$ denote the number of moves required to transfer n disks—the running time of our algorithm. Our vacuous base case implies that $T(0) = 0$, and the more general recursive algorithm implies that $T(n) = 2T(n-1) + 1$ for any $n \geq 1$. The annihilator method lets us quickly derive a closed form solution $\boxed{T(n) = 2^n - 1}$. In particular, moving a tower of 64 disks requires $2^{64} - 1 =$ 18,446,744,073,709,551,615 individual moves. Thus, even at the impressive rate of one move per second, the monks at Benares will be at work for approximately 585 billion years before, with a thunderclap, the world will vanish.

The Hanoi algorithm has two very simple non-recursive formulations, for those of us who do not have an army of assistants to take care of smaller piles. Let's label the needles 0, 1, and 2, and suppose the problem is to move n disks from needle 0 to needle 2 (as shown on the previous page). The non-recursive algorithm can be described with four simple rules. The proof that these rules force the same behavior as the recursive algorithm is a straightforward exercise in induction. (Hint, hint.)[5]

- If n is even, always move the smallest disk forward ($\cdots \to 0 \to 1 \to 2 \to 0 \to \cdots$).
- If n is odd, always move the smallest disk backward ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).
- Never move the same disk twice in a row.
- When there is no legal move, the puzzle is solved.

An even shorter formulation ties the algorithm more closely with its analysis. Let $\rho(n)$ denote the smallest integer k such that $n/2^k$ is not an integer. For example, $\rho(42) = 2$, because $42/2^1$ is an integer but $42/2^2$ is not. (Equivalently, $\rho(n)$ is one more than the position of the least significant 1 bit in the binary representation of n.) The function $\rho(n)$ is sometimes called the 'ruler' function, because its behavior resembles the marks on a ruler:

$$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 6, 1, 2, 1, 3, 1, 2, 1, 4, 1 \ldots.$$

The Hanoi algorithm can now be described in one line:

In step i, move disk $\rho(i)$ forward if $n - i$ is even, backward if $n - i$ is odd.

On move 2^n, this rule requires us to move disk $n+1$, which doesn't exist, so the algorithm ends. At this point, the puzzle is solved. Again, the proof that this algorithm is equivalent to our recursive formulation is a simple exercise in induction. (Hint, hint.)

1.2 Subset Sum

Let's start with a concrete example, the *subset sum* problem: Given a set X of positive integers and *target* integer T, is there a subset of element sin X that add up to T? Notice that there can be more than one such subset. For example, if $X = \{8, 6, 7, 5, 3, 10, 9\}$ and $T = 15$, the answer is TRUE, thanks to the subset $\{8, 7\}$ or $\{7, 5, 3\}$ or $\{6, 9\}$ or $\{5, 10\}$.[6] On the other hand, if $X = \{11, 6, 5, 1, 7, 13, 12\}$ and $T = 15$, the answer is FALSE.

There are two trivial cases. If the target value T is zero, then we can immediately return TRUE, since the elements of the empty set add up to zero.[7] On the other hand, if $T < 0$, or if $T \neq 0$ but the set X is empty, then we can immediately return FALSE.

For the general case, consider an arbitrary element $x \in X$. (We've already handled the case where X is empty.) There are two possibilities to consider.

- There is a subset of X that *includes* x and sums to T. Equivalently, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$.
- There is a subset of X that *excludes* x and sums to T. Equivalently, there must be a subset of $X \setminus \{x\}$ that sums to T.

[5] This means **Pay attention! This might show up on an exam! You might want to do this!**

[6] See http://www.cribbage.org/rules/ for more possibilities.

[7] The empty set is always the best base case!

So we can solve SUBSETSUM(X, T) by reducing it to two simpler instances: SUBSETSUM$(X \setminus \{x\}, T - x)$ and SUBSETSUM$(X \setminus \{x\}, T)$. Here's how the resulting recusive algorithm might look if X is stored in an array.

```
SUBSETSUM($X[1..n], T$):
  if $T = 0$
      return TRUE
  else if $T < 0$ or $n = 0$
      return FALSE
  else
      return SUBSETSUM($X[2..n], T$) $\vee$ SUBSETSUM($X[2..n], T - X[1]$)
```

Proving this algorithm correct is a straightforward exercise in induction. If $T = 0$, then the elements of the empty subset sum to T. Otherwise, if T is negative or the set X is empty, then no subset of X sums to T. Otherwise, if there is a subset that sums to T, then either it contains $X[1]$ or it doesn't, and the Recursion Fairy correctly checks for each of those possibilities. Done.

The running time $T(n)$ clearly satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$, so the running time is $T(n) = O(2^n)$ by the annihilator method.

Along similar lines, here's a recursive algorithm for actually *constructing* a subset of X that sums to T, if one exists. This algorithm also runs in $O(2^n)$ time.

```
CONSTRUCTSUBSET($X[1..n], T$):
  if $T = 0$
      return $\varnothing$
  if $T < 0$ or $n = 0$
      return NONE

  $Y \leftarrow$ CONSTRUCTSUBSET($X[2..n], T$)
  if $Y \neq$ NONE
      return $Y$

  $Y \leftarrow$ CONSTRUCTSUBSET($X[2..n], T - X[1]$)
  if $Y \neq$ NONE
      return $Y \cup \{X[1]\}$

  return NONE
```

These two algorithms are examples of a general algorithmic technique called *backtracking*. You can imagine the algorithm searching through a binary tree of recursive possibilities like a maze, trying to find a hidden treasure ($T = 0$), and backtracking whenever it reaches a dead end ($T < 0$ or $n = 0$). For *some* problems, there are tricks that allow the recursive algorithm to recognize some branches dead ends without exploring them directly, thereby speeding up the algorithm; two such problems are described later in these notes. Alas, SUBSETSUM is not one of the those problems; in the worst case, our algorithm explicitly considers *every* subset of X.

1.3 Longest Increasing Subsequence

Suppose we want to find the longest increasing subsequence of a sequence of n integers. That is, we are given an array $A[1..n]$ of integers, and we want to find the longest sequence of indices $1 \leq i_1 < i_2 < \cdots i_k \leq n$ such that $A[i_j] < A[i_{j+1}]$ for all j.

To derive a recursive algorithm for this problem, we start with a recursive definition of the kinds of objects we're playing with: sequences and subsequences.

A *sequence of integers* is either empty
or an integer followed by a sequence of integers.

This definition suggests the following strategy for devising a recursive algorithm. If the input sequence is empty, there's nothing to do. Otherwise, we should figure out what to do with the first element of the input sequence, and let the Recursion Fairy take care of everything else. We can formalize this strategy somewhat by giving a recursive definition of subsequence (using array notation to represent sequences):

The only *subsequence* of the empty sequence is the empty sequence.
A *subsequence* of $A[1\,..\,n]$ is either a subsequence of $A[2\,..\,n]$
or $A[1]$ followed by a subsequence of $A[2\,..\,n]$.

We're not just looking for just *any* subsequence, but a *longest* subsequence with the property that elements are in *increasing* order. So let's try to add those two conditions to our definition. (I'll omit the familiar vacuous base case.)

The *LIS* of $A[1\,..\,n]$ is either the LIS of $A[2\,..\,n]$
or $A[1]$ followed by the LIS of $A[2\,..\,n]$ *with elements larger than* $A[1]$,
whichever is longer.

This definition is correct, but it's not quite recursive—we're defining 'longest increasing subsequence' in terms of the *different* 'longest increasing subsequence with elements larger than x', which we haven't properly defined yet. Fortunately, this second object has a very similar recursive definition. (Again, I'm omitting the vacuous base case.)

If $A[1] \le x$, the LIS of $A[1\,..\,n]$ with elements larger than x must be
the LIS of $A[2\,..\,n]$ with elements larger than x.

Otherwise, the LIS of $A[1\,..\,n]$ with elements larger than x is
either the LIS of $A[2\,..\,n]$ with elements larger than x
or $A[1]$ followed by the LIS of $A[2\,..\,n]$ with elements larger than $A[1]$,
whichever is longer.

The longest increasing subsequence without restrictions can now be redefined as the longest increasing subsequence with elements larger than $-\infty$. Rewriting this recursive definition into pseudocode gives us the following recursive algorithm.

```
LIS($A[1\,..\,n]$):
  return LISBIGGER($-\infty, A[1\,..\,n]$)
```

```
LISBIGGER($prev, A[1\,..\,n]$):
  if $n = 0$
      return $0$
  else
      $max \gets$ LISBIGGER($prev, A[2\,..\,n]$)
      if $A[1] > prev$
          $L \gets 1 +$ LISBIGGER($A[1], A[2\,..\,n]$)
          if $L > max$
              $max \gets L$
      return $max$
```

The running time of this algorithm satisfies the recurrence

$$T(n) \leq O(1) + 2T(n-1),$$

which implies that $T(n) = O(2^n)$ by the annihilator method. We really shouldn't be surprised by this running time; in the worst case, the algorithm examines each of the 2^n subsequences of the input array.

In lecture, a student suggested the following alternate strategy, which avoids defining a new object with the 'larger than x' constraint. We still only have to decide whether to include or exclude the first element $A[1]$. We consider the case where $A[1]$ is excluded exactly the same way, but to consider the case where $A[1]$ is included, we remove any elements of $A[2\,..\,n]$ that are larger than $A[1]$ *before* we recurse. This modified strategy gives us the following algorithm:

```
FILTER($A[1\,..\,n], x$):
  $j \leftarrow 1$
  for $i \leftarrow 1$ to $n$
      if $A[i] > x$
          $B[j] \leftarrow A[i]$;  $j \leftarrow j + 1$
  return $B[1\,..\,j]$
```

```
LIS($A[1\,..\,n]$):
  if $n = 0$
      return 0
  else
      $max \leftarrow$ LIS($prev, A[2\,..\,n]$)
      $L \leftarrow 1 +$ LIS($A[1]$, FILTER($A[2\,..\,n], A[1]$))
      if $L > max$
          $max \leftarrow L$
      return $max$
```

The FILTER subroutine clearly runs in $O(n)$ time, so the running time of LIS satisfies the recurrence $T(n) \leq 2T(n-1) + O(n)$, which solves to $T(n) \leq O(2^n)$ by the annihilator method. This upper bound pessimistically assumes that FILTER never actually removes any elements, but this can actually happen in the worst case, when the input sequence is sorted.

*1.4 3SAT

This section assumes you are already familiar with NP-completeness.

Now let's consider the mother of all NP-hard problems, 3SAT. Given a boolean formula in conjunctive normal form, with at most three literals in each clause, our task is to determine whether any assignment of values of the variables makes the formula true. Yes, this problem is NP-hard, which means that a polynomial algorithm is almost certainly impossible. Too bad; we have to solve the problem anyway.

The trivial solution is to try every possible assignment. We'll evaluate the running time of our 3SAT algorithms in terms of the number of variables in the formula, so let's call that n. Provided any clause appears in our input formula at most once—a condition that we can easily enforce in polynomial time—the overall input size is $O(n^3)$. There are 2^n possible assignments, and we can evaluate each assignment in $O(n^3)$ time, so the overall running time is $O(2^n n^3)$.

Since polynomial factors like n^3 are essentially noise when the overall running time is exponential, from now on I'll use $\mathrm{poly}(n)$ to represent some arbitrary polynomial in n; in other words, $\mathrm{poly}(n) = n^{O(1)}$. For example, the trivial algorithm for 3SAT runs in time $O(2^n \,\mathrm{poly}(n))$.

We can make this algorithm smarter by exploiting the special recursive structure of 3CNF formulas:

A 3CNF formula is either nothing
or a clause with three literals $\wedge$ a 3CNF formula

Suppose we want to decide whether some 3CNF formula Φ with n variables is satisfiable. Of course this is trivial if Φ is the empty formula, so suppose

$$\Phi = (x \vee y \vee z) \wedge \Phi'$$

for some literals x, y, z and some 3CNF formula Φ'. By distributing the $\wedge$ across the $\vee$s, we can rewrite Φ as follows:

$$\Phi = (x \wedge \Phi') \vee (y \wedge \Phi') \vee (z \wedge \Phi')$$

For any boolean formula Ψ and any literal x, let $\Psi|x$ (pronounced "sigh given eks") denote the simpler boolean formula obtained by assuming x is true. It's not hard to prove by induction (hint, hint) that $x \wedge \Psi = x \wedge \Psi|x$, which implies that

$$\Phi = (x \wedge \Phi'|x) \vee (y \wedge \Phi'|y) \vee (z \wedge \Phi'|z).$$

Thus, in any satisfying assignment for Φ, either x is true and $\Phi'|x$ is satisfiable, or y is true and $\Phi'|y$ is satisfiable, or z is true and $\Phi'|z$ is satisfiable. Each of the smaller formulas has at most $n-1$ variables. If we recursively check all three possibilities, we get the running time recurrence

$$T(n) \leq 3T(n-1) + \operatorname{poly}(n),$$

whose solution is $O(3^n \operatorname{poly}(n))$. So we've actually done *worse!*

But these three recursive cases are not mutually exclusive! If $\Phi'|x$ is *not* satisfiable, then x *must* be false in any satisfying assignment for Φ. So instead of recursively checking $\Phi'|y$ in the second step, we can check the even simpler formula $\Phi'|\bar{x}y$. Similarly, if $\Phi'|\bar{x}y$ is not satisfiable, then we know that y must be false in any satisfying assignment, so we can recursively check $\Phi'|\bar{x}\bar{y}z$ in the third step.

```
3SAT($\Phi$):
  if $\Phi = \varnothing$
      return TRUE
  $(x \vee y \vee z) \wedge \Phi' \leftarrow \Phi$
  if 3SAT($\Phi|x$)
      return TRUE
  if 3SAT($\Phi|\bar{x}y$)
      return TRUE
  return 3SAT($\Phi|\bar{x}\bar{y}z$)
```

The running time off this algorithm obeys the recurrence

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \operatorname{poly}(n),$$

where $\operatorname{poly}(n)$ denotes the polynomial time required to simplify boolean formulas, handle control flow, move stuff into and out of the recursion stack, and so on. The annihilator method gives us the solution

$$T(n) = O(\lambda^n \operatorname{poly}(n)) = \boxed{O(1.83928675522^n)}$$

where $\lambda \approx 1.83928675521\ldots$ is the largest root of the characteristic polynomial $r^3 - r^2 - r - 1$. (Notice that we cleverly eliminated the polynomial noise by increasing the base of the exponent ever so slightly.)

We can improve this algorithm further by eliminating *pure* literals from the formula before recursing. A literal x is *pure* in if it appears in the formula Φ but its negation $\bar{x}$ does not. It's not hard to prove (hint, hint) that if Φ has a satisfying assignment, then it has a satisfying assignment where every pure literal is true. If $\Phi = (x \vee y \vee z) \wedge \Phi'$ has no pure literals, then some in Φ contains the literal $\bar{x}$, so we can write

$$\Phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi'$$

for some literals u and v (each of which might be y, $\bar{y}$, z, or $\bar{z}$). It follows that the first recursive formula $\Phi|x$ has contains the clause $(u \vee v)$. We can recursively eliminate the variables u and v just as we tested the variables y and x in the second and third cases of our previous algorithm:

$$\Phi|x = (u \vee v) \wedge \Phi'|x = (u \wedge \Phi'|xu) \vee (v \wedge \Phi'|x\bar{u}v).$$

Here is our new faster algorithm:

```
3SAT($\Phi$):
  if $\Phi = \varnothing$
      return TRUE
  if $\Phi$ has a pure literal $x$
      return 3SAT($\Phi|x$)
  $(x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi' \leftarrow \Phi$
  if 3SAT($\Phi|xu$)
      return TRUE
  if 3SAT($\Phi|x\bar{u}v$)
      return TRUE
  if 3SAT($\Phi|\bar{x}y$)
      return TRUE
  return 3SAT($\Phi|\bar{x}\bar{y}z$)
```

The running time $T(n)$ of this new algorithm satisfies the recurrence

$$T(n) = 2T(n-2) + 2T(n-3) + \operatorname{poly}(n),$$

and the annihilator method implies that

$$T(n) = O(\mu^n \operatorname{poly}(n)) = \boxed{O(1.76929235425^n)}$$

where $\mu \approx 1.76929235424\ldots$ is the largest root of the characteristic polynomial $r^3 - 2r - 2$.

Naturally, this approach can be extended much further. As of 2004, the fastest (deterministic) algorithm for 3SAT runs in $O(1.473^n)$ time[8], but there is absolutely no reason to believe that this is the best possible.

*1.5 Maximum Independent Set

This section assumes you are already familiar with graphs and NP-completeness.

Finally, suppose we are asked to find the largest independent set in an undirected graph G. Once again, we have an obvious, trivial algorithm: Try every subset of nodes, and return the largest subset with no edges. Expressed recursively, the algorithm might look like this.

[8] Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science* 329(1–3):303–313, 2004.

```
MaximumIndSetSize($G$):
  if $G = \varnothing$
      return $0$
  else
      $v \leftarrow$ any node in $G$
      $withv \leftarrow 1 + $ MaximumIndSetSize($G \setminus N(v)$)
      $withoutv \leftarrow$ MaximumIndSetSize($G \setminus \{v\}$)
      return $\max\{withv, withoutv\}$.
```

Here, $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors. Our algorithm is exploiting the fact that if an independent set contains v, then by definition it contains none of v's neighbors. In the worst case, v has no neighbors, so $G \setminus \{v\} = G \setminus N(v)$. Thus, the running time of this algorithm satisfies the recurrence $T(n) = 2T(n-1) + \operatorname{poly}(n) = O(2^n \operatorname{poly}(n))$. Surprise, surprise.

This algorithm is mirroring a crude recursive upper bound for the number of *maximal* independent sets in a graph. If the graph is non-empty, then every maximal independent set either includes or excludes each vertex. Thus, the number of maximal independent sets satisfies the recurrence $M(n) \leq 2M(n-1)$, with base case $M(1) = 1$. The annihilator method gives us $M(n) \leq 2^n - 1$. The only subset that we aren't counting with this upper bound is the empty set!

We can improve this upper bound by more carefully examining the worst case of the recurrence. If v has no neighbors, then $N(v) = \{v\}$, and both recursive calls consider a graph with $n-1$ nodes. But in this case, v is in *every* maximal independent set, so one of the recursive calls is redundant. On the other hand, if v has at least one neighbor, then $G \setminus N(v)$ has at most $n-2$ nodes. So now we have the following recurrence.

$$M(n) \leq \max \left\{ \begin{array}{l} M(n-1) \\ M(n-1) + M(n-2) \end{array} \right\} = O(1.61803398875^n)$$

The upper bound is derived by solving each case separately using the annihilator method and taking the worst of the two cases. The first case gives us $M(n) = O(1)$; the second case yields our old friends the Fibonacci numbers.

We can improve this bound even more by examining the new worst case: v has exactly one neighbor w. In this case, either v or w appears in any maximal independent set. Thus, instead of recursively searching in $G \setminus \{v\}$, we should recursively search in $G \setminus N(w)$, which has at most $n-1$ nodes. On the other hand, if G has no nodes with degree 1, then $G \setminus N(v)$ has at most $n-3$ nodes.

$$M(n) \leq \max \left\{ \begin{array}{l} M(n-1) \\ 2M(n-2) \\ M(n-1) + M(n-3) \end{array} \right\} = O(1.46557123188^n)$$

The base of the exponent is the largest root of the characteristic polynomial $r^3 - r^2 - 1$. The second case implies a bound of $O(\sqrt{2}^n) = O(1.41421356237^n)$, which is smaller.

We can apply this improvement technique one more time. If G has a node v with degree 3 or more, then $G \setminus N(v)$ has at most $n-4$ nodes. Otherwise (since we have already considered nodes of degree 0 and 1), every node in the graph has degree 2. Let u, v, w be a path of three nodes in G (possibly with u adjacent to w). In any maximal independent set, either v is present and u, w are absent, or u is present and its two neighbors are absent, or w is present and its two neighbors are absent. In all three cases, we recursively count maximal independent sets in a graph with $n-3$

nodes.

$$M(n) \leq \max \left\{ \begin{array}{l} M(n-1) \\ 2M(n-2) \\ M(n-1) + M(n-4) \\ 3M(n-3) \end{array} \right\} = O(3^{n/3}) = O(1.44224957031^n)$$

The third case implies a bound of $O(1.3802775691^n)$, where the base is the largest root of $r^4 - r^3 - 1$.

Unfortunately, we cannot apply the same improvement trick again. A graph consisting of $n/3$ triangles (cycles of length three) has exactly $3^{n/3}$ maximal independent sets, so our upper bound is tight in the worst case.

Now from this recurrence, we can derive an efficient algorithm to compute the largest independent set in G in $O(3^{n/3}\operatorname{poly}(n)) = O(1.44224957032^n)$ time.

```
MAXIMUMINDSETSIZE($G$):
  if $G = \varnothing$
      return 0

  else if $G$ has a node $v$ with degree 0
      return $1 +$ MAXIMUMINDSETSIZE($G \setminus \{v\}$)        ⟨⟨$n-1$⟩⟩

  else if $G$ has a node $v$ with degree 1
      $w \leftarrow v$'s neighbor
      $withv \leftarrow 1 +$ MAXIMUMINDSETSIZE($G \setminus N(v)$)      ⟨⟨$n-2$⟩⟩
      $withw \leftarrow 1 +$ MAXIMUMINDSETSIZE($G \setminus N(w)$)      ⟨⟨$\leq n-2$⟩⟩
      return $\max\{withv, withw\}$

  else if $G$ has a node $v$ with degree greater than 2
      $withv \leftarrow 1 +$ MAXIMUMINDSETSIZE($G \setminus N(v)$)      ⟨⟨$\leq n-4$⟩⟩
      $withoutv \leftarrow$ MAXIMUMINDSETSIZE($G \setminus \{v\}$)        ⟨⟨$\leq n-1$⟩⟩
      return $\max\{withv, withoutv\}$

  else ⟨⟨every node in $G$ has degree 2⟩⟩
      $v \leftarrow$ any node;   $u, w \leftarrow v$'s neighbors
      $withu \leftarrow 1 +$ MAXIMUMINDSETSIZE($G \setminus N(u)$)      ⟨⟨$\leq n-3$⟩⟩
      $withv \leftarrow 1 +$ MAXIMUMINDSETSIZE($G \setminus N(v)$)      ⟨⟨$\leq n-3$⟩⟩
      $withw \leftarrow 1 +$ MAXIMUMINDSETSIZE($G \setminus N(w)$)      ⟨⟨$\leq n-3$⟩⟩
      return $\max\{withu, withv, withw\}$
```

1.6 Generalities

Recursion is reduction from a problem to one or more simpler instances of the *same* problem. Almost every recursive algorithm (and inductive proof) closely follows a recursive definition for the object being computed. Here are a few simple recursive definitions that can be used to derive recursive algorithms:

- A natural number is either 0, or the successor of a natural number.
- A sequence is either empty, or an atom followed by a sequence.
- A sequence is either empty, an atom, or the concatenation of two shorter sequences.
- A set is either empty, or the union of a set and an atom.
- A nonempty set either is a singleton, or the union of two nonempty sets.
- A rooted tree is either nothing, or a node pointing to zero or more rooted trees.

- A binary tree is either nothing, or a node pointing to two binary trees.
- A triangulated polygon is nothing, or a triangle glued to a triangulated polygon [*not* obvious!]
- $\displaystyle\sum_{i=1}^{n} a_i = \begin{cases} 0 & \text{if } n = 0 \\ \displaystyle\sum_{i=1}^{n-1} a_i + a_n & \text{otherwise} \end{cases}$
- $\displaystyle\sum_{i=1}^{n} a_i = \begin{cases} 0 & \text{if } n = 0 \\ a_1 & \text{if } n = 1 \\ \displaystyle\sum_{i=1}^{k} a_i + \sum_{i=k+1}^{n} a_i & \text{otherwise, for some } 1 \le k \le n-1 \end{cases}$

> *The control of a large force is the same principle as the control of a few men:*
> *it is merely a question of dividing up their numbers.*
> — Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

2 Divide and Conquer

2.1 MergeSort

Mergesort is one of the earliest algorithms proposed for sorting. According to Knuth, it was suggested by John von Neumann as early as 1945.

1. Divide the array $A[1\,..\,n]$ into two subarrays $A[1\,..\,m]$ and $A[m+1\,..\,n]$, where $m = \lfloor n/2 \rfloor$.
2. Recursively mergesort the subarrays $A[1\,..\,m]$ and $A[m+1\,..\,n]$.
3. Merge the newly-sorted subarrays $A[1\,..\,m]$ and $A[m+1\,..\,n]$ into a single sorted list.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse:	I	N	O	S	R	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	S	R	T	X

A Mergesort example.

The first step is completely trivial; we only need to compute the median index m. The second step is also trivial, thanks to our friend the recursion fairy. All the real work is done in the final step; the two sorted subarrays $A[1\,..\,m]$ and $A[m+1\,..\,n]$ can be merged using a simple linear-time algorithm. Here's a complete specification of the Mergesort algorithm; for simplicity, we separate out the merge step as a subroutine.

```
MERGESORT($A[1..n]$):
  if ($n > 1$)
      $m \leftarrow \lfloor n/2 \rfloor$
      MERGESORT($A[1..m]$)
      MERGESORT($A[m+1..n]$)
      MERGE($A[1..n], m$)
```

```
MERGE($A[1..n], m$):
  $i \leftarrow 1$;  $j \leftarrow m+1$
  for $k \leftarrow 1$ to $n$
      if $j > n$
          $B[k] \leftarrow A[i]$;  $i \leftarrow i+1$
      else if $i > m$
          $B[k] \leftarrow A[j]$;  $j \leftarrow j+1$
      else if $A[i] < A[j]$
          $B[k] \leftarrow A[i]$;  $i \leftarrow i+1$
      else
          $B[k] \leftarrow A[j]$;  $j \leftarrow j+1$

  for $k \leftarrow 1$ to $n$
      $A[k] \leftarrow B[k]$
```

To prove that the algorithm is correct, we use our old friend induction. We can prove that MERGE is correct using induction on the total size of the two subarrays $A[i\,..\,m]$ and $A[j\,..\,n]$ left to be merged into $B[k\,..\,n]$. The base case, where at least one subarray is empty, is straightforward; the algorithm just copies it into B. Otherwise, the smallest remaining element is either $A[i]$ or $A[j]$, since both subarrays are sorted, so $B[k]$ is assigned correctly. The remaining subarrays—either $A[i+1\,..\,m]$ and $A[j\,..\,n]$, or $A[i\,..\,m]$ and $A[j+1\,..\,n]$—are merged correctly into $B[k+1\,..\,n]$ by the inductive hypothesis.[1] This completes the proof.

[1] "The inductive hypothesis" is just a technical nickname for our friend the recursion fairy.

Now we can prove MERGESORT correct by another round of straightforward induction.[2] The base cases $n \leq 1$ are trivial. Otherwise, by the inductive hypothesis, the two smaller subarrays $A[1 \mathinner{..} m]$ and $A[m+1 \mathinner{..} n]$ are sorted correctly, and by our earlier argument, merged into the correct sorted output.

What's the running time? Since we have a recursive algorithm, we're going to get a recurrence of some sort. MERGE clearly takes linear time, since it's a simple for-loop with constant work per iteration. We get the following recurrence for MERGESORT:

$$T(1) = O(1), \qquad T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

2.2 Aside: Domain Transformations

Except for the floor and ceiling, this recurrence falls into case (b) of the Master Theorem [CLR, §4.3]. If we simply ignore the floor and ceiling, the Master Theorem suggests the solution $T(n) = O(n \log n)$. We can easily check that this answer is correct using induction, but there is a simple method for solving recurrences like this directly, called *domain transformation*.

First we overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \leq 2T\big(\lceil n/2 \rceil\big) + O(n) \leq 2T(n/2+1) + O(n).$$

Now we define a new function $S(n) = T(n+\alpha)$, where α is a constant chosen so that $S(n)$ satisfies the Master-ready recurrence $S(n) \leq 2S(n/2) + O(n)$. To figure out the appropriate value for α, we compare two versions of the recurrence for $T(n+\alpha)$:

$$\begin{aligned} S(n) \leq 2S(n/2) + O(n) \quad &\implies \quad T(n+\alpha) \leq 2T(n/2+\alpha) + O(n) \\ T(n) \leq 2T(n/2+1) + O(n) \quad &\implies \quad T(n+\alpha) \leq 2T((n+\alpha)/2+1) + O(n+\alpha) \end{aligned}$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n+\alpha)/2 + 1$, which implies that $\alpha = 2$. The Master Theorem tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n).$$

We can use domain transformations to remove floors, ceilings, and lower order terms from any recurrence. But now that we know this, we won't bother actually grinding through the details!

2.3 QuickSort

Quicksort was discovered by Tony Hoare in 1962. In this algorithm, the hard work is splitting the array into subsets so that merging the final result is trivial.

1. Choose a *pivot* element from the array.
2. Split the array into three subarrays containing the items less than the pivot, the pivot itself, and the items bigger than the pivot.
3. Recursively quicksort the first and last subarray.

[2] Many textbooks draw an artificial distinction between several different flavors of induction: standard/weak ('the principle of mathematical induction'), strong ('the second principle of mathematical induction'), complex, structural, transfinite, decaffeinated, etc. Those textbooks would call this proof "strong" induction. I don't. *All* induction proofs have precisely the same structure: Pick an arbitrary object, make one or more simpler objects from it, apply the inductive hypothesis to the simpler object(s), infer the required property for the original object, and check the base cases. Induction is just recursion for proofs.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	$\boxed{\text{N}}$	G	E	X	A	M	P	L
Partition:	M	A	E	G	I	L	N	R	X	O	S	P	T
Recurse:	A	E	G	I	L	M	N	O	P	S	R	T	X

A Quicksort example.

Here's a more formal specification of the Quicksort algorithm. The separate PARTITION subroutine takes the original position of the pivot element as input and returns the post-partition pivot position as output.

```
QUICKSORT($A[1..n]$):
  if ($n > 1$)
      Choose a pivot element $A[p]$
      $k \leftarrow$ PARTITION($A, p$)
      QUICKSORT($A[1..k-1]$)
      QUICKSORT($A[k+1..n]$)
```

```
PARTITION($A[1..n], p$):
  if ($p \neq n$)
      swap $A[p] \leftrightarrow A[n]$
  $i \leftarrow 0$;  $j \leftarrow n$
  while ($i < j$)
      repeat $i \leftarrow i + 1$ until ($i = j$ or $A[i] \geq A[n]$)
      repeat $j \leftarrow j - 1$ until ($i = j$ or $A[j] \leq A[n]$)
      if ($i < j$)
          swap $A[i] \leftrightarrow A[j]$
  if ($i \neq n$)
      swap $A[i] \leftrightarrow A[n]$
  return $i$
```

Just as we did for mergesort, we need two induction proofs to show that QUICKSORT is correct—weak induction to prove that PARTITION correctly partitions the array, and then straightforward strong induction to prove that QUICKSORT correctly sorts assuming PARTITION is correct. I'll leave the gory details as an exercise for the reader.

The analysis is also similar to mergesort. PARTITION runs in $O(n)$ time: $j - i = n$ at the beginning, $j - i = 0$ at the end, and we do a constant amount of work each time we increment i or decrement j. For QUICKSORT, we get a recurrence that depends on k, the rank of the chosen pivot:

$$T(n) = T(k-1) + T(n-k) + O(n)$$

If we could choose the pivot to be the median element of the array A, we would have $k = \lceil n/2 \rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) = 2T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + O(n) \leq 2T(n/2) + O(n),$$

and we'd have $T(n) = O(n \log n)$ by the Master Theorem.

Unfortunately, although it is *theoretically* possible to locate the median of an unsorted array in linear time, the algorithm is fairly complicated, and the hidden constant in the $O()$ notation is quite large. So in practice, programmers settle for something simple, like choosing the first or last element of the array. In this case, k can be anything from 1 to n, so we have

$$T(n) = \max_{1 \leq k \leq n} \big(T(k-1) + T(n-k) + O(n)\big)$$

In the worst case, the two subproblems are completely unbalanced—either $k = 1$ or $k = n$—and the recurrence becomes $T(n) \leq T(n-1) + O(n)$. The solution is $T(n) = O(n^2)$. Another common heuristic is 'median of three'—choose three elements (usually at the beginning, middle, and end of the array), and take the middle one as the pivot. Although this is better in practice than just choosing one element, we can still have $k = 2$ or $k = n - 1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n) \le T(1) + T(n-2) + O(n)$, whose solution is still $T(n) = O(n^2)$.

Intuitively, the pivot element will 'usually' fall somewhere in the middle of the array, say between $n/10$ and $9n/10$. This suggests that the *average-case* running time is $O(n \log n)$. Although this intuition is correct, we are still far from a *proof* that quicksort is usually efficient. I'll formalize this intuition about average cases in a later lecture.

2.4 The Pattern

Both mergesort and and quicksort follow the same general three-step pattern of all divide and conquer algorithms:

1. **Split** the problem into several *smaller independent* subproblems.
2. **Recurse** to get a subsolution for each subproblem.
3. **Merge** the subsolutions together into the final solution.

If the size of any subproblem falls below some constant threshold, the recursion bottoms out. Hopefully, at that point, the problem is trivial, but if not, we switch to a different algorithm instead.

Proving a divide-and-conquer algorithm correct usually involves strong induction. Analyzing the running time requires setting up and solving a recurrence, which often (but unfortunately not always!) can be solved using the Master Theorem, perhaps after a simple domain transformation.

2.5 Multiplication

Adding two n-digit numbers takes $O(n)$ time by the standard iterative 'ripple-carry' algorithm, using a lookup table for each one-digit addition. Similarly, multiplying an n-digit number by a one-digit number takes $O(n)$ time, using essentially the same algorithm.

What about multiplying two n-digit numbers? At least in the United States, every grade school student (supposedly) learns to multiply by breaking the problem into n one-digit multiplications and n additions:

$$\begin{array}{r}
31415962 \\
\times\ 27182818 \\
\hline
251327696 \\
31415962 \\
251327696 \\
62831924 \\
251327696 \\
31415962 \\
219911734 \\
62831924 \\
\hline
853974377340916
\end{array}$$

We could easily formalize this algorithm as a pair of nested for-loops. The algorithm runs in $O(n^2)$ time—altogether, there are $O(n^2)$ digits in the partial products, and for each digit, we spend constant time.

We can do better by exploiting the following algebraic formula:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

Here is a divide-and-conquer algorithm that computes the product of two n-digit numbers x and y, based on this formula. Each of the four sub-products e, f, g, h is computed recursively. The last line does not involve any multiplications, however; to multiply by a power of ten, we just shift the digits and fill in the right number of zeros.

```
MULTIPLY($x, y, n$):
  if $n = 1$
      return $x \cdot y$
  else
      $m \leftarrow \lceil n/2 \rceil$
      $a \leftarrow \lfloor x/10^m \rfloor$;  $b \leftarrow x \bmod 10^m$
      $d \leftarrow \lfloor y/10^m \rfloor$;  $c \leftarrow y \bmod 10^m$
      $e \leftarrow$ MULTIPLY($a, c, m$)
      $f \leftarrow$ MULTIPLY($b, d, m$)
      $g \leftarrow$ MULTIPLY($b, c, m$)
      $h \leftarrow$ MULTIPLY($a, d, m$)
      return $10^{2m}e + 10^m(g + h) + f$
```

You can easily prove by induction that this algorithm is correct. The running time for this algorithm is given by the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1,$$

which solves to $T(n) = O(n^2)$ by the Master Theorem (after a simple domain transformation). Hmm... I guess this didn't help after all.

But there's a trick, first published by Anatoliĭ Karatsuba in 1962.[3] We can compute the middle coefficient $bc + ad$ using only *one* recursive multiplication, by exploiting yet another bit of algebra:

$$ac + bd - (a - b)(c - d) = bc + ad$$

This trick lets use replace the last three lines in the previous algorithm as follows:

```
FASTMULTIPLY($x, y, n$):
  if $n = 1$
      return $x \cdot y$
  else
      $m \leftarrow \lceil n/2 \rceil$
      $a \leftarrow \lfloor x/10^m \rfloor$;  $b \leftarrow x \bmod 10^m$
      $d \leftarrow \lfloor y/10^m \rfloor$;  $c \leftarrow y \bmod 10^m$
      $e \leftarrow$ FASTMULTIPLY($a, c, m$)
      $f \leftarrow$ FASTMULTIPLY($b, d, m$)
      $g \leftarrow$ FASTMULTIPLY($a - b, c - d, m$)
      return $10^{2m}e + 10^m(e + f - g) + f$
```

The running time of Karatsuba's FASTMULTIPLY algorithm is given by the recurrence

$$T(n) \leq 3T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1.$$

After a domain transformation, we can plug this into the Master Theorem to get the solution $T(n) = O(n^{\lg 3}) = O(n^{1.585})$, a significant improvement over our earlier quadratic-time algorithm.[4]

[3] However, the same basic trick was used non-recursively by Gauss in the 1800s to multiply complex numbers using only three real multiplications.

[4] Karatsuba actually proposed an algorithm based on the formula $(a + c)(b + d) - ac - bd = bc + ad$. This algorithm also runs in $O(n^{\lg 3})$ time, but the actual recurrence is a bit messier: $a - b$ and $c - d$ are still m-digit numbers, but $a + b$ and $c + d$ might have $m + 1$ digits. The simplification presented here is due to Donald Knuth.

Of course, in practice, all this is done in binary instead of decimal.

We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to get even faster multiplication algorithms. Ultimately, this idea leads to the development of the *Fast Fourier transform*, a more complicated divide-and-conquer algorithm that can be used to multiply two n-digit numbers in $O(n \log n)$ time.[5] We'll talk about Fast Fourier transforms in detail in the next lecture.

2.6 Exponentiation

Given a number a and a positive integer n, suppose we want to compute a^n. The standard naïve method is a simple for-loop that does $n-1$ multiplications by a:

```
SLOWPOWER($a, n$):
  $x \leftarrow a$
  for $i \leftarrow 2$ to $n$
      $x \leftarrow x \cdot a$
  return $x$
```

This iterative algorithm requires n multiplications.

Notice that the input a could be an integer, or a rational, or a floating point number. In fact, it doesn't need to be a number at all, as long as it's something that we know how to multiply. For example, the same algorithm can be used to compute powers modulo some finite number (an operation commonly used in cryptography algorithms) or to compute powers of matrices (an operation used to evaluate recurrences and to compute shortest paths in graphs). All that's required is that a belong to a multiplicative group.[6] Since we don't know what kind of things we're mutliplying, we can't know how long a multiplication takes, so we're forced analyze the running time in terms of the number of multiplications.

There is a much faster divide-and-conquer method, using the simple formula $a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}$. What makes this approach more efficient is that once we compute the first factor $a^{\lfloor n/2 \rfloor}$, we can compute the second factor $a^{\lceil n/2 \rceil}$ using at most one more multiplication.

```
FASTPOWER($a, n$):
  if $n = 1$
      return $a$
  else
      $x \leftarrow$ FASTPOWER($a, \lfloor n/2 \rfloor$)
      if $n$ is even
          return $x \cdot x$
      else
          return $x \cdot x \cdot a$
```

The total number of multiplications is given by the recurrence $T(n) \leq T(\lfloor n/2 \rfloor) + 2$, with the base case $T(1) = 0$. After a domain transformation, the Master Theorem gives us the solution $T(n) = O(\log n)$.

[5] This fast algorithm for multiplying integers using FFTs was discovered by Arnold Schönhange and Volker Strassen in 1971.

[6] A *multiplicative group* $(G, \otimes)$ is a set G and a function $\otimes : G \times G \rightarrow G$, satisfying three axioms:

1. There is a *unit* element $1 \in G$ such that $1 \otimes g = g \otimes 1$ for any element $g \in G$.
2. Any element $g \in G$ has a *inverse* element $g^{-1} \in G$ such that $g \otimes g^{-1} = g^{-1} \otimes g = 1$
3. The function is *associative*: for any elements $f, g, h \in G$, we have $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

Incidentally, this algorithm is asymptotically optimal—any algorithm for computing a^n must perform $\Omega(\log n)$ multiplications. In fact, when n is a power of two, this algorithm is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, our divide-and-conquer algorithm computes a^{15} in six multiplications ($a^{15} = a^7 \cdot a^7 \cdot a$; $a^7 = a^3 \cdot a^3 \cdot a$; $a^3 = a \cdot a \cdot a$), but only five multiplications are necessary ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). Nobody knows of an algorithm that always uses the minimum possible number of multiplications.

2.7 Optimal Binary Search Trees

This last example combines the divide-and-conquer strategy with recursive backtracking.

You all remember that the cost of a successful search in a binary search tree is proportional to the number of ancestors of the target node.[7] As a result, the worst-case search time is proportional to the depth of the tree. To minimize the worst-case search time, the height of the tree should be as small as possible; ideally, the tree is perfectly balanced.

In many applications of binary search trees, it is more important to minimize the total cost of several searches than to minimize the worst-case cost of a single search. If x is a more 'popular' search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y, even if that means increasing the overall depth of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree of depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of n keys $A[1\,..\,n]$ and an array of corresponding *access frequencies* $f[1\,..\,n]$. Over the lifetime of the search tree, we will search for the key $A[i]$ exactly $f[i]$ times. Our task is to build the binary search tree that minimizes the *total* search time.

Before we think about how to solve this problem, we should first come up with a good recursive definition of the function we are trying to optimize! Suppose we are also given a binary search tree T with n nodes; let v_i denote the node that stores $A[i]$. Up to constant factors, the total cost of performing all the binary searches is given by the following expression:

$$\begin{aligned} \mathit{Cost}(T, f[1\,..\,n]) &= \sum_{i=1}^{n} f[i] \cdot \#\text{ancestors of } v_i \\ &= \sum_{i=1}^{n} f[i] \cdot \sum_{j=1}^{n} \big[v_j \text{ is an ancestor of } v_i\big] \end{aligned}$$

Here I am using the extremely useful *Iverson bracket notation* to transform Boolean expressions into integers—for any Boolean expression X, we define $[X] = 1$ if X is true, and $[X] = 0$ if X is false.[8] Since addition is commutative, we can swap the order of the two summations.

$$\mathit{Cost}(T, f[1\,..\,n]) = \sum_{j=1}^{n} \sum_{i=1}^{n} f[i] \cdot \big[v_j \text{ is an ancestor of } v_i\big] \tag{*}$$

Finally, we can exploit the recursive structure of the binary tree by splitting the outer sum into a sum over the left subtree of T, a sum over the right subtree of T, and a 'sum' over the root of T.

[7] An *ancestor* of a node v is either the node itself or an ancestor of the parent of v. A *proper* ancestor of v is either the parent of v or a proper ancestor of the parent of v.

[8] In other words, $[X]$ has precisely the same semantics as $!!X$ in C.

Suppose v_r is the root of T.

$$\begin{aligned}
\mathit{Cost}(T, f[1\,..\,n]) = &\sum_{j=1}^{r-1}\sum_{i=1}^{n} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]\\
&+\sum_{i=1}^{n} f[i]\cdot\big[v_r \text{ is an ancestor of } v_i\big]\\
&+\sum_{j=1}^{r+1}\sum_{i=1}^{n} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]
\end{aligned}$$

We can simplify all three of these partial sums. Since any node in the left subtree is an ancestor only of nodes in the left subtree, we can change the upper limit of the first inner summation from n to $r-1$. Similarly, we can change the lower limit of the last inner summation from 1 to $r+1$. Finally, the root v_r is an ancestor of *every* node in T, so we can remove the bracket expression from the middle summation.

$$\begin{aligned}
\mathit{Cost}(T, f[1\,..\,n]) = &\sum_{j=1}^{r-1}\sum_{i=1}^{r} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]\\
&+\sum_{i=1}^{n} f[i]\\
&+\sum_{j=1}^{r+1}\sum_{i=r+1}^{n} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]
\end{aligned}$$

Now the first and third summations look exactly like our earlier expression (*) for $\mathit{Cost}(T, f[1\,..\,n])$. We finally have our recursive definition!

$$\boxed{\mathit{Cost}(T, f[1\,..\,n]) = \mathit{Cost}(\mathit{left}(T), f[1\,..\,r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; \mathit{Cost}(\mathit{right}(T), f[r+1\,..\,n])}$$

The base case for this recurrence is, as usual, $n=0$; the cost of the empty tree, over the empty set of frequency counts, is zero.

Now our task is to compute the tree T_{opt} that minimizes this cost function. Suppose we somehow magically knew that the root of T_{opt} is v_r. Then the recursive definition of $\mathit{Cost}(T, f)$ immediately implies that the left subtree left(T_{opt}) must be the optimal search tree for the keys $A[1\,..\,r-1]$ and access frequencies $f[1\,..\,r-1]$. Similarly, the right subtree right(T_{opt}) must be the optimal search tree for the keys $A[r+1\,..\,n]$ and access frequencies $f[r+1\,..\,n]$. **Once we choose the correct key to store at the root, the Recursion Fairy will automatically construct the rest of the optimal tree for us.** More formally, let $\mathit{OptCost}(f[1\,..\,n])$ denote the total cost of the optimal search tree for the given frequency counts. We immediately have the following recursive definition.

$$\boxed{\mathit{OptCost}(f[1\,..\,n]) = \min_{1\le r\le n}\left\{\mathit{OptCost}(f[1\,..\,r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; \mathit{OptCost}(f[r+1\,..\,n])\right\}}$$

Again, the base case is $\mathit{OptCost}(f[1\,..\,0]) = 0$; the best way to organize zero keys, given an empty set of frequencies, is by storing them in the empty tree!

This recursive definition can be translated mechanically into a recursive algorithm, whose running time $T(n)$ satisfies the recurrence

$$T(n) = \Theta(n) + \sum_{k=1}^{n} \bigl(T(k-1) + T(n-k)\bigr).$$

The $\Theta(n)$ term comes from computing the total number of searches $\sum_{i=1}^{n} f[i]$. The recurrence looks harder than it really is. To transform it into a more familiar form, we regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

$$\begin{aligned} T(n) &= \Theta(n) + 2\sum_{k=0}^{n-1} T(k) \\ T(n-1) &= \Theta(n-1) + 2\sum_{k=0}^{n-2} T(k) \\ T(n) - T(n-1) &= \Theta(1) + 2T(n-1) \\ T(n) &= 3T(n-1) + \Theta(1) \end{aligned}$$

The solution $\boxed{T(n) = \Theta(3^n)}$ now follows from the annihilator method.

It's worth emphasizing here that our recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with n nodes satisfies the recurrence

$$N(n) = \sum_{r=1}^{n-1} N(r-1) \cdot N(n-r), \quad N(0) = 1,$$

which has the closed-from solution $N(n) = \Theta(4^n/\sqrt{n})$. Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees. A full enumeration of binary search trees would consider all possible *pairings* of left and right subtrees; hence the product in the recurrence for $N(n)$.

> **Calvin:** *Here's another math problem I can't figure out. What's 9+4?*
>
> **Hobbes:** *Ooh, that's a tricky one. You have to use calculus and imaginary numbers for this.*
>
> **Calvin:** *IMAGINARY NUMBERS?!*
>
> **Hobbes:** *You know, eleventeen, thirty-twelve, and all those. It's a little confusing at first.*
>
> **Calvin:** *How did YOU learn all this? You've never even gone to school!*
>
> **Hobbes:** *Instinct. Tigers are born with it.*
>
> — "Calvin and Hobbes" (January 6, 1998)

> *Blech! Ack! Oop! THPPFFT!*
>
> — Bill the Cat, "Bloom County" (1980)

3 Fast Fourier Transforms

3.1 Polynomials

In this lecture we'll talk about algorithms for manipulating *polynomials*: functions of one variable built from additions subtractions, and multiplications (but no divisions). The most common representation for a polynomial $p(x)$ is as a sum of weighted powers of a variable x:

$$\boxed{p(x) = \sum_{j=0}^{n} a_j x^j.}$$

The numbers a_j are called *coefficients*. The *degree* of the polynomial is the largest power of x; in the example above, the degree is n. Any polynomial of degree n can be specified by a sequence of $n+1$ coefficients. Some of these coefficients may be zero, but not the nth coefficient, because otherwise the degree would be less than n.

Here are three of the most common operations that are performed with polynomials:

- **Evaluate:** Give a polynomial p and a number x, compute the number $p(x)$.
- **Add:** Give two polynomials p and q, compute a polynomial $r = p+q$, so that $r(x) = p(x)+q(x)$ for all x. If p and q both have degree n, then their sum $p+q$ also has degree n.
- **Multiply:** Give two polynomials p and q, compute a polynomial $r = p \cdot q$, so that $r(x) = p(x) \cdot q(x)$ for all x. If p and q both have degree n, then their product $p \cdot q$ has degree $2n$.

Suppose we represent a polynomial of degree n as an array of $n+1$ coefficients $P[0\,..\,n]$, where $P[j]$ is the coefficient of the x^j term. We learned simple algorithms for all three of these operations in high-school algebra:

```
EVALUATE($P[0..n], x$):
  $X \gets 1$    $\langle\langle X = x^j \rangle\rangle$
  $y \gets 0$
  for $j \gets 0$ to $n$
      $y \gets y + P[j] \cdot X$
      $X \gets X \cdot x$
  return $y$
```

```
ADD($P[0..n], Q[0..n]$):
  for $j \gets 0$ to $n$
      $R[j] \gets P[j] + Q[j]$
  return $R[0..n]$
```

```
MULTIPLY($P[0..n], Q[0..m]$):
  for $j \gets 0$ to $n+m$
      $R[j] \gets 0$
  for $j \gets 0$ to $n$
      for $k \gets 0$ to $m$
          $R[j+k] \gets R[j+k] + P[j] \cdot Q[k]$
  return $R[0..n+m]$
```

EVALUATE uses $O(n)$ arithmetic operations.[1] This is the best we can hope for, but we can cut the number of multiplications in half using *Horner's rule*:

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)).$$

```
HORNER($P[0\,..\,n], x$):
    $y \leftarrow P[n]$
    for $i \leftarrow n-1$ downto $0$
        $y \leftarrow x \cdot y + P[i]$
    return $y$
```

The addition algorithm also runs in $O(n)$ time, and this is clearly the best we can do.

The multiplication algorithm, however, runs in $O(n^2)$ time. In the previous lecture, we saw a divide and conquer algorithm (due to Karatsuba) for multiplying two n-bit integers in only $O(n^{\lg 3})$ steps; precisely the same algorithm can be applied here. Even cleverer divide-and-conquer strategies lead to multiplication algorithms whose running times are arbitrarily close to linear—$O(n^{1+\varepsilon})$ for your favorite value $e > 0$—but with great cleverness comes great confusion. These algorithms are difficult to understand, even more difficult to implement correctly, and not worth the trouble in practice thanks to large constant factors.

3.2 Alternate Representations

Part of what makes multiplication so much harder than the other two operations is our input representation. Coefficients vectors are the most common representation for polynomials, but there are at least two other useful representations.

3.2.1 Roots

The Fundamental Theorem of Algebra states that every polynomial p of degree n has exactly n *roots* $r_1, r_2, \ldots r_n$ such that $p(r_j) = 0$ for all j. Some of these roots may be irrational; some of these roots may by complex; and some of these roots may be repeated. Despite these complications, this theorem implies a unique representation of any polynomial of the form

$$\boxed{p(x) = s\prod_{j=1}^{n}(x - r_j)}$$

where the r_j's are the roots and s is a scale factor. Once again, to represent a polynomial of degree n, we need a list of $n+1$ numbers: one scale factor and n roots.

Given a polynomial in this root representation, we can clearly evaluate it in $O(n)$ time. Given two polynomials in root representation, we can easily multiply them in $O(n)$ time by multiplying their scale factors and just concatenating the two root sequences.

Unfortunately, if we want to add two polynomials in root representation, we're pretty much out of luck. There's essentially *no* correlation between the roots of p, the roots of q, and the roots

[1] I'm going to assume in this lecture that each arithmetic operation takes $O(1)$ time. This may not be true in practice; in fact, one of the most powerful applications of FFTs is fast *integer* multiplication. One of the fastest integer multiplication algorithms, due to Schönhage and Strassen, multiplies two n-bit binary numbers in $O(n \log n \log\log n \log\log\log n \log\log\log\log n \cdots)$ time. The algorithm uses an n-element Fast Fourier Transform, which requires several $O(\log n)$-nit integer multiplications. These smaller multiplications are carried out recursively (of course!), which leads to the cascade of logs in the running time. Needless to say, this is a can of worms.

of $p+q$. We could convert the polynomials to the more familiar coefficient representation first—this takes $O(n^2)$ time using the high-school algorithms—but there's no easy way to convert the answer back. In fact, for most polynomials of degree 5 or more in coefficient form, it's *impossible* to compute roots exactly.[2]

3.2.2 Samples

Our third representation for polynomials comes from a different consequence of the Fundamental Theorem of Algebra. Given a list of $n+1$ pairs $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\ \}$, there is *exactly one* polynomial p of degree n such that $p(x_j) = y_j$ for all j. This is just a generalization of the fact that any two points determine a unique line, since a line is (the graph of) a polynomial of degree 1. We say that the polynomial p *interpolates* the points (x_j, y_j). As long as we agree on the sample locations x_j in advance, we once again need exactly $n+1$ numbers to represent a polynomial of degree n.

Adding or multiplying two polynomials in this sample representation is easy, as long as they use the same sample locations x_j. To add the polynomials, just add their sample values. To multiply two polynomials, just multiply their sample values; however, if we're multiplying two polynomials of degree n, we need to *start* with $2n+1$ sample values for each polynomial, since that's how many we need to uniquely represent the product polynomial. Both algorithms run in $O(n)$ time.

Unfortunately, evaluating a polynomial in this representation is no longer trivial. The following formula, due to Lagrange, allows us to compute the value of any polynomial of degree n at any point, given a set of $n+1$ samples.

$$p(x) = \sum_{j=0}^{n-1} \left(y_j \frac{\prod_{k\neq j}(x - x_k)}{\prod_{k\neq j}(x_j - x_k)} \right) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k\neq j}(x_j - x_k)} \prod_{k\neq j}(x - x_k) \right)$$

Hopefully it's clear that formula actually describes a polynomial, since each term in the rightmost sum is written as a scaled product of monomials. It's also not hard to check that $p(x_j) = y_j$ for all j. As I mentioned earlier, the fact that this is *the only* polynomial that interpolates the points $\{(x_j, y_j)\}$ is an easy consequence of the Fundamental Theorem of Algebra. We can easily transform Lagrange's formula into an $O(n^2)$-time algorithm.

3.2.3 Summary

We find ourselves in the following frustrating situation. We have three representations for polynomials and three basic operations. Each representation allows us to almost trivially perform a different pair of operations in linear time, but the third takes at least quadratic time, if it can be done at all!

	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	$\boldsymbol{O(n^2)}$
roots + scale	$O(n)$	∞	$O(n)$
samples	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$

[2] This is where numerical analysis comes from.

3.3 Converting Between Representations

What we need are fast algorithms to convert quickly from one representation to another. That way, when we need to perform an operation that's hard for our default representation, we can switch to a different representation that makes the operation easy, perform that operation, and then switch back. This strategy immediately rules out the root representation, since (as I mentioned earlier) finding roots of polynomials is impossible in general, at least if we're interested in exact results.

So how do we convert from coefficients to samples and back? Clearly, once we choose our sample positions x_j, we can compute each sample value $y_j = p(x_j)$ in $O(n)$ time from the coefficients using Horner's rule. So we can convert a polynomial of degree n from coefficients to samples in $O(n^2)$ time. The Lagrange formula gives us an explicit conversion algorithm from the sample representation back to the more familiar coefficient representation. If we use the naïve algorithms for adding and multiplying polynomials (in coefficient form), this conversion takes $O(n^3)$ time.

We can improve the cubic running time by observing that *both* conversion problems boil down to computing the product of a matrix and a vector. The explanation will be slightly simpler if we assume the polynomial has degree $n-1$, so that n is the number of coefficients or samples. Fix a sequence $x_0, x_1, \ldots, x_{n-1}$ of sample *positions,* and let V be the $n \times n$ matrix where $v_{ij} = x_i^j$ (indexing rows and columns from 0 to $n-1$):

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}$$

The matrix V is called a *Vandermonde* matrix. The vector of coefficients $\vec{a} = (a_0, a_1, \ldots, a_{n-1})$ and the vector of sample *values* $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$ are related by the equation

$$\boxed{V\vec{a} = \vec{y}},$$

or in more detail:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Given this formulation, we can clearly transform any coefficient vector $\vec{a}$ into the corresponding sample vector $\vec{y}$ in $O(n^2)$ time.

Conversely, if we know the sample values $\vec{y}$, we can recover the coefficients by solving a system of n linear equations in n unknowns; this takes $O(n^3)$ time if we use Gaussian elimination. But we can speed this up by remembering that **we get to choose the sample positions**. In other words, the matrix V is hard-coded into the algorithm. To convert from samples to coefficients, we can simply multiply the sample vector by the inverse of V, again in $O(n^2)$ time.

$$\boxed{\vec{a} = V^{-1}\vec{y}}$$

Computing V^{-1} would take $O(n^3)$ time if we had to do it from scratch using Gaussian elimination, but because we fixed the set of sample positions in advance, the matrix V^{-1} can be hard-coded into the algorithm.[3]

So we can convert from coefficients to sample value and back in $O(n^2)$ time, which is pointless, because we can ad, multiply, or evaluate directly in either representation in $O(n^2)$ time. But wait! There's still a degree of freedom he haven't exploited. ***We get to choose the sample values!*** Our conversion algorithm may be slow only because we're trying to be too general. Perhaps, if we choose a set of sample points with just the right kind of recursive structure, we can do the conversion more quickly. In fact, there is a set of sample points that's perfect for the job.

3.4 The Discrete Fourier Transform

Given a polynomial of degree $n-1$, we'd like to find n sample points that are somehow as symmetric as possible. The most natural choice for those n points are the *nth roots of unity*; these are the roots of the polynomial $x^n - 1 = 0$. These n roots are spaced exactly evenly around the unit circle in the complex plane.[4] Every nth root of unity is a power of the *primitive* root

$$\boxed{\omega_n = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}.}$$

A typical nth root of unity has the form

$$\omega_n^j = e^{(2\pi i/n)j} = \cos\left(\frac{2\pi}{n}j\right) + i\sin\left(\frac{2\pi}{n}j\right).$$

These complex numbers have several useful properties for any integers n and k:

- There are only n different nth roots of unity: $\omega_n^k = \omega_n^{k \bmod n}$.
- If n is even, then $\omega_n^{k+n/2} = -\omega_n^k$; in particular, $\omega_n^{n/2} = -\omega_n^0 = -1$.
- $1/\omega_n^k = \omega_n^{-k} = \overline{\omega_n^k} = (\overline{\omega_n})^k$, where the bar represents complex conjugation: $\overline{a+bi} = a - bi$
- $\omega_n = \omega_{kn}^k$. Thus, every nth root of unity is also a (kn)th root of unity.

If we sample a polynomial of degree $n-1$ at the nth roots of unity, the resulting list of sample values is called the *discrete Fourier transform* of the polynomial (or more formally, of the coefficient vector). Thus, given an array $P[0\,..\,n-1]$ of coefficients, the discrete Fourier transform computes a new vector $P^*[0\,..\,n-1]$ where

$$\boxed{\boxed{P^*[j] = p(\omega_n^j) = \sum_{k=0}^{n-1} P[k]\cdot\omega_n^{jk}}}$$

[3] Actually, it is possible to invert an $n \times n$ matrix in $o(n^3)$ time, using fast matrix multiplication algorithms that closely resemble Karatsuba's sub-quadratic divide-and-conquer algorithm for integer/polynomial multiplication.

[4] In this lecture, i always represents the square root of -1. Most computer scientists are used to thinking of i as an integer index into a sequence, an array, or a for-loop, but we obviously can't do that here. The physicist's habit of using $j = \sqrt{-1}$ just delays the problem (how do physicists write quaternions?), and typographical tricks like I or $\mathbf{i}$ or Mathematica's ⅈ are just stupid.

We can obviously compute P^* in $O(n^2)$ time, but the structure of the nth roots of unity lets us do better. But before we describe that faster algorithm, let's think about how we might invert this transformation.

Recall that transforming coefficients into sample values is a *linear* transformation; the sample vector is the product of a Vandermonde matrix V and the coefficient vector. For the discrete Fourier transform, each entry in V is an nth root of unity; specifically, $\boxed{v_{jk} = \omega_n^{jk}}$ for all j, k.

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

To invert the discrete Fourier transform, converting sample values back to coefficients,, we just have to multiply P^* by the inverse matrix V^{-1}. But this is almost the same as multiplying by V itself, because of the following amazing fact:

$$\boxed{V^{-1} = \overline{V}/n}$$

In other words, if $W = V^{-1}$ then $w_{jk} = \overline{v_{jk}}/n = \overline{\omega_n^{jk}}/n = \omega_n^{-jk}/n$. It's not hard to prove this fact with a little linear algebra.

Proof: We just have to show that $M = VW$ is the identity matrix. We can compute a single entry in this matrix as follows:

$$m_{jk} = \sum_{l=0}^{n-1} v_{jl} \cdot w_{lk} = \sum_{l=0}^{n-1} \omega_n^{jl} \cdot \overline{\omega_n^{lk}}/n = \frac{1}{n}\sum_{l=0}^{n-1} \omega_n^{jl-lk} = \frac{1}{n}\sum_{l=0}^{n-1} (\omega_n^{j-k})^l$$

If $j = k$, then $\omega_n^{j-k} = 1$, so

$$m_{jk} = \frac{1}{n}\sum_{l=0}^{n-1} 1 = \frac{n}{n} = 1,$$

and if $j \neq k$, we have a geometric series

$$m_{jk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l = \frac{(\omega_n^{j-k})^n - 1}{\omega_n^{j-k} - 1} = \frac{(\omega_n^n)^{j-k} - 1}{\omega_n^{j-k} - 1} = \frac{1^{j-k} - 1}{\omega_n^{j-k} - 1} = 0.$$

That's it! □

What this means for us computer scientists is that any algorithm for computing the discrete Fourier transform can be easily modified to compute the inverse transform as well.

3.5 Divide and Conquer!

The symmetry in the roots of unity also allow us to compute the discrete Fourier transform efficiently using a divide and conquer strategy. The basic structure of the algorithm is almost the same as MergeSort, and the $O(n \log n)$ running time will ultimately follow from the same recurrence. The *Fast Fourier Transform* algorithm, popularized by Cooley and Tukey in 1965[5], assumes that n is a power of two; if necessary, we can just pad the coefficient vector with zeros.

Let $p(x)$ be a polynomial of degree $n-1$, represented by an array $P[0\,..\,n-1]$ of coefficients. The FFT algorithm begins by splitting p into two smaller polynomials u and v, each with degree $n/2-1$. The coefficients of u are precisely the the even-degree coefficients of p; the coefficients of v are the odd-degree coefficients of p. For example, if $p(x) = 3x^3 - 4x^2 + 7x + 5$, then $u(x) = -4x + 5$ and $v(x) = 3x + 7$. These three polynomials are related by the equation

$$\boxed{p(x) = u(x^2) + x \cdot v(x^2).}$$

In particular, if x is an nth root of unity, we have

$$p(\omega_n^k) \;=\; u(\omega_n^{2k}) + \omega_n^k \cdot v(\omega_n^{2k}).$$

Now we can exploit those roots of unity again. Since n is a power of two, n must be even, so we have $\omega_n^{2k} = \omega_{n/2}^k = \omega_{n/2}^{k \bmod n/2}$. In other words, the values of p at the nth roots of unity depend on the values of u and v at $(n/2)$th roots of unity.

$$p(\omega_n^k) \;=\; u(\omega_{n/2}^{k \bmod n/2}) + \omega_n^k \cdot v(\omega_{n/2}^{k \bmod n/2}).$$

But those are just coefficients in the DFTs of u and v! We conclude that the DFT coefficients of P are defined by the following recurrence:

$$\boxed{P^*[k] = U^*[k \bmod n/2] + \omega_n^k \cdot V^*[k \bmod n/2]}$$

Once the Recursion Fairy give us U^* and V^*, we can compute P^* in linear time. The base case for the recurrence is $n = 1$: if $p(x)$ has degree 0, then $P^*[0] = P[0]$.

Here's the complete FFT algorithm, along with its inverse.

[5] Actually, the FFT algorithm was previously published by Runge and König in 1924, and again by Yates in 1932, and again by Stumpf in 1937, and again by Danielson and Lanczos in 1942. But it was first *used* by Gauss in the 1800s for calculating the paths of asteroids from a finite number of equally-spaced observations. By hand. Fourier always did it the hard way. Cooley and Tukey apparently developed their algorithm to help detect Soviet nuclear tests without actually visiting Soviet nuclear facilities, by interpolating off-shore seismic readings. Without their rediscovery of the FFT algorithm, the nuclear test ban treaty would never have been ratified, and we'd all be speaking Russian, or more likely, whatever language radioactive glass speaks.

```
FFT($P[0\,..\,n-1]$):
  if $n = 1$
      return $P$

  for $j \leftarrow 0$ to $n/2 - 1$
      $U[j] \leftarrow P[2j]$
      $V[j] \leftarrow P[2j+1]$

  $U^* \leftarrow$ FFT($U[0\,..\,n/2-1]$)
  $V^* \leftarrow$ FFT($V[0\,..\,n/2-1]$)

  $\omega_n \leftarrow \cos(\frac{2\pi}{n}) + i\sin(\frac{2\pi}{n})$
  $\omega \leftarrow 1$

  for $j \leftarrow 0$ to $n/2 - 1$
      $P^*[j]       \leftarrow U^*[j] + \omega \cdot V^*[j]$
      $P^*[j + n/2] \leftarrow U^*[j] - \omega \cdot V^*[j]$
      $\omega \leftarrow \omega \cdot \omega_n$

  return $P^*[0\,..\,n-1]$
```

```
InverseFFT($P^*[0\,..\,n-1]$):
  if $n = 1$
      return $P$

  for $j \leftarrow 0$ to $n/2 - 1$
      $U^*[j] \leftarrow P^*[2j]$
      $V^*[j] \leftarrow P^*[2j+1]$

  $U \leftarrow$ InverseFFT($U[0\,..\,n/2-1]$)
  $V \leftarrow$ InverseFFT($V[0\,..\,n/2-1]$)

  $\overline{\omega_n} \leftarrow \cos(\frac{2\pi}{n}) - i\sin(\frac{2\pi}{n})$
  $\omega \leftarrow 1$

  for $j \leftarrow 0$ to $n/2 - 1$
      $P[j]       \leftarrow 2(U[j] + \omega \cdot V[j])$
      $P[j + n/2] \leftarrow 2(U[j] - \omega \cdot V[j])$
      $\omega \leftarrow \omega \cdot \overline{\omega_n}$

  return $P[0\,..\,n-1]$
```

The overall running time of this algorithm satisfies the recurrence $T(n) = \Theta(n) + 2T(n/2)$, which as we all know solves to $T(n) = \Theta(n \log n)$.

3.6 Fast Multiplication

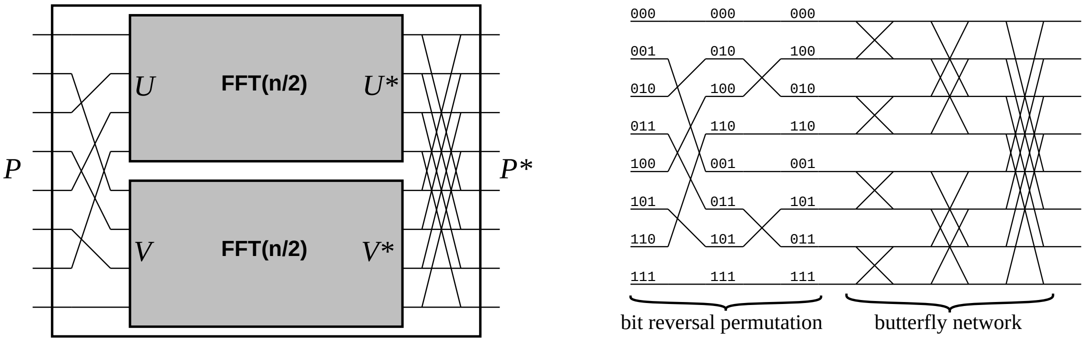
Given two polynomials p and q, each represented by an array of coefficients, we can multiply them in $\Theta(n \log n)$ arithmetic operations as follows. First, pad the coefficient vectors and with zeros until the size is a power of two greater than or equal to the sum of the degrees. Then compute the DFTs of each coefficient vector, multiply the sample values one by one, and compute the inverse DFT of the resulting sample vector.

```
FFTMultiply($P[0\,..\,n-1], Q[0\,..\,m-1]$):
  $\ell \leftarrow \lceil \lg(n+m) \rceil$
  for $j \leftarrow n$ to $2^\ell - 1$
      $P[j] \leftarrow 0$
  for $j \leftarrow m$ to $2^\ell - 1$
      $Q[j] \leftarrow 0$

  $P^* \leftarrow FFT(P)$
  $Q^* \leftarrow FFT(Q)$
  for $j \leftarrow 0$ to $2^\ell - 1$
      $R^*[j] \leftarrow P^*[j] \cdot Q^*[j]$
  return InverseFFT($R^*$)
```

3.7 Inside the FFT

FFTs are often implemented in hardware as circuits. To see the recursive structure of the circuit, let's connect the top-level inputs and outputs to the inputs and outputs of the recursive calls. On the left we split the input P into two recursive inputs U and V. On the right, we combine the outputs U^* and V^* to obtain the final output P^*.



The recursive structure of the FFT algorithm.

If we expand this recursive structure completely, we see that the circuit splits naturally into two parts. The left half computes the *bit-reversal permutation* of the input. To find the position of $P[k]$ in this permutation, write k in binary, and then read the bits backward. For example, in an 8-element bit-reversal permutation, $P[3] = P[011_2]$ ends up in position $6 = 110_2$. The right half of the FFT circuit is a *butterfly network*. Butterfly networks are often used to route between processors in massively-parallel computers, since they allow any processor to communicate with any other in only $O(\log n)$ steps.

Caveat Lector! This presentation is appropriate for graduate students or undergrads with strong math backgrounds, but it leaves most undergrads confused. You may find it less confusing to approach the material in the opposite order, as follows:

First, any polynomial can be split into even-degree and odd-degree parts:

$$p(x) = p_{\textsf{even}}(x^2) + x \cdot p_{\textsf{odd}}(x^2).$$

We can evaluate $p(x)$ by recursively evaluating $p_{\textsf{even}}(x^2)$ and $p_{\textsf{odd}}(x^2)$ and doing $O(1)$ arithmetic operations.

Now suppose our task is to evaluate the degree-n polynomial $p(x)$ at n different points x, as quickly as possible. To exploit the even/odd recursive structure, we must choose the n evaluation points carefully. Call a set X of n values *delicious* if either (1) X has only one element, or (2) the set $X^2 = \{x^2 \mid x \in X\}$ has only $n/2$ elements and X^2 is delicious. Clearly such a set exists only if N is a power of two. If someone magically handed us a delicious set X, we could compute $\{p(x) \mid x \in X\}$ in $O(n \log n)$ time using the even/odd recursive structure. Bit reversal permutation, blah blah blah, butterfly network, yadda yadda yadda.

If n is a power of two, then the set of integers $\{0, 1, \ldots, n-1\}$ is delicious, **provided we perform all arithmetic modulo n**. But that only tells us $p(x) \bmod n$, and we want the actual value of $p(x)$. Of course, we can use larger moduli: $\{0, k, 2k, \ldots, (n-1)k\}$ is delicious mod nk. We can avoid modular arithmetic entirely by using complex roots of unity—the set $\{e^{2\pi i/n} \mid i = 0, 1, \ldots, n-1\}$ is delicious! The sequence of values $p(e^{2\pi i/n})$ is called the *discrete Fourier transform* of p.

Finally, to invert this transformation from coefficients to values, we repeat exactly the same procedure, using the same delicious set *but in the opposite order*. Blardy blardy, linear algebra, hi dee hi dee hi dee ho.

> *Those who cannot remember the past are doomed to repeat it.*
>
> — George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

> *The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word* 'research'*. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term* 'research' *in his presence. You can imagine how he felt, then, about the term* 'mathematical'*. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*
>
> — Richard Bellman, on the origin of his term 'dynamic programming' (1984)

> *If we all listened to the professor, we may be all looking for professor jobs.*
>
> — Pittsburgh Steeler's head coach Bill Cowher, responding to
> David Romer's dynamic-programming analysis of football strategy (2003)

4 Dynamic Programming

4.1 Fibonacci Numbers

The Fibonacci numbers F_n, named after Leonardo Fibonacci Pisano[1], the mathematician who popularized 'algorism' in Europe in the 13th century, are defined as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them:

```
RecFibo($n$):
  if ($n < 2$)
      return $n$
  else
      return RecFibo($n-1$) + RecFibo($n-2$)
```

How long does this algorithm take? Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. If $T(n)$ represents the number of recursive calls to RecFibo, we have the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1.$$

This looks an awful lot like the recurrence for Fibonacci numbers! The annihilator method gives us an asymptotic bound of $\Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803398875$, the so-called *golden ratio*, is the largest root of the polynomial $r^2 - r - 1$. But it's fairly easy to prove (hint, hint) the exact solution $\boxed{T(n) = 2F_{n+1} - 1}$. In other words, computing F_n using this algorithm takes more than twice as many steps as just counting to F_n!

Another way to see this is that the RecFibo is building a big binary tree of additions, with nothing but zeros and ones at the leaves. Since the eventual output is F_n, our algorithm must call RecRibo(1) (which returns 1) exactly F_n times. A quick inductive argument implies that RecFibo(0) is called exactly F_{n-1} times. Thus, the recursion tree has $F_n + F_{n-1} = F_{n+1}$ leaves, and therefore, because it's a full binary tree, it must have $2F_{n+1} - 1$ nodes.

[1] literally, "Leonardo, son of Bonacci, of Pisa"

4.2 Memo(r)ization and Dynamic Programming

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RECURSIVEFIBO(n) results in one recursive call to RECURSIVEFIBO($n-1$), two recursive calls to RECURSIVEFIBO($n-2$), three recursive calls to RECURSIVEFIBO($n-3$), five recursive calls to RECURSIVEFIBO($n-4$), and in general, F_{k-1} recursive calls to RECURSIVEFIBO($n-k$), for any $0 \leq k < n$. For each call, we're recomputing some Fibonacci number from scratch.

We can speed up the algorithm considerably just by writing down the results of our recursive calls and looking them up again if we need them later. This process is called *memoization*.[2]

```
MEMFIBO($n$):
  if ($n < 2$)
      return $n$
  else
      if $F[n]$ is undefined
          $F[n] \leftarrow$ MEMFIBO($n-1$) + MEMFIBO($n-2$)
      return $F[n]$
```

If we actually trace through the recursive calls made by MEMFIBO, we find that the array $F[\,]$ gets filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. Once we realize this, we can replace the recursion with a simple for-loop that just fills up the array in that order, instead of relying on the complicated recursion to do it for us. This gives us our first explicit *dynamic programming* algorithm.

```
ITERFIBO($n$):
  $F[0] \leftarrow 0$
  $F[1] \leftarrow 1$
  for $i \leftarrow 2$ to $n$
      $F[i] \leftarrow F[i-1] + F[i-2]$
  return $F[n]$
```

ITERFIBO clearly takes only $O(n)$ time and $O(n)$ space to compute F_n, an exponential speedup over our original recursive algorithm. We can reduce the space to $O(1)$ by noticing that we never need more than the last two elements of the array:

```
ITERFIBO2($n$):
  prev $\leftarrow 1$
  curr $\leftarrow 0$
  for $i \leftarrow 1$ to $n$
      next $\leftarrow$ curr + prev
      prev $\leftarrow$ curr
      curr $\leftarrow$ next
  return curr
```

(This algorithm uses the non-standard but perfectly consistent base case $F_{-1} = 1$.)

But even this isn't the fastest algorithm for computing Fibonacci numbers. There's a faster algorithm defined in terms of matrix multiplication, using the following wonderful fact:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

[2] "My name is Elmer J. Fudd, millionaire. I own a mansion and a yacht."

In other words, multiplying a two-dimensional vector by the matrix $\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right]$ does exactly the same thing as one iteration of the inner loop of IterFibo2. This might lead us to believe that multiplying by the matrix n times is the same as iterating the loop n times:

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}^n\begin{bmatrix}1\\0\end{bmatrix}=\begin{bmatrix}F_{n-1}\\F_n\end{bmatrix}.$$

A quick inductive argument proves this. So if we want to compute the nth Fibonacci number, all we have to do is compute the nth power of the matrix $\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right]$.

If we use repeated squaring, computing the nth power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$ 2×2 matrix multiplications, but one matrix multiplications can be done with only a constant number of integer multiplications and additions. Thus, we can compute F_n in only $O(\log n)$ integer arithmetic operations.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm. Right?

4.3 Uh. . . wait a minute.

Well, not exactly. Fibonacci numbers grow exponentially fast. The nth Fibonacci number is approximately $n\log_{10}\phi \approx n/5$ decimal digits long, or $n\log_2\phi \approx 2n/3$ bits. So we can't possibly compute F_n in logarithmic time — we need $\Omega(n)$ time just to write down the answer!

I've been cheating by assuming we can do arbitrary-precision arithmetic in constant time. As we discussed last time, multiplying two n-digit numbers takes $O(n\log n)$ time. That means that the matrix-based algorithm's actual running time is given by the recurrence

$$T(n) = T(\lfloor n/2\rfloor) + O(n\log n),$$

which solves to $T(n) = O(n\log n)$ by the Master Theorem.

Is this slower than our "linear-time" iterative algorithm? No! Addition isn't free, either. Adding two n-digit numbers takes $O(n)$ time, so the running time of the iterative algorithm is $O(n^2)$. (Do you see why?) So our matrix algorithm really is faster than our iterative algorithm, but not exponentially faster.

Incidentally, in the original recursive algorithm, the extra cost of arbitrary-precision arithmetic is overwhelmed by the huge number of recursive calls. The correct recurrence is

$$T(n) = T(n-1) + T(n-2) + O(n),$$

which still has the solution $O(\phi^n)$ by the annihilator method.

4.4 The Pattern: Smart Recursion

In a nutshell, dynamic programming is *recursion without repetition*. Developing a dynamic programming algorithm almost always requires two distinct steps.

1. **Formulate the problem recursively.** Write down a formula for the whole problem as a simple combination of the answers to smaller subproblems.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution by considering the intermediate subproblems in the correct order. This is usually easier than the first step.

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

Dynamic programming algorithms store the solutions of intermediate subproblems, often *but not always* done with some kind of array or table. One common mistake that lots of students make is to be distracted by the table (because tables are easy and familiar) and miss the *much* more important (and difficult) task of finding a correct recurrence. ***Dynamic programming isn't about filling in tables; it's about smart recursion.*** As long as we memoize the correct recurrence, an explicit table isn't necessary, but if the recursion is incorrect, nothing works.

4.5 The Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions dont cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string.

If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution. Nevertheless, for many problems involving dynamic programming, many student's first intuition is to apply a greedy strategy. This almost never works; problems that can be solved correctly by a greedy algorithm are *very* rare. Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never? No, never! What, *never*? Well. . . hardly ever.[3]

A different lecture note describes the effort required to prove that greedy algorithms are correct, in the rare instances when they are. **You will not receive *any* credit for *any* greedy algorithm for *any* problem in this class without a *formal* proof of correctness.** We'll push through the formal proofs for two specific problems—minimum spanning trees and shortest paths—but those will be the only greedy algorithms we will consider this semester.

4.6 Edit Distance

The *edit distance* between two words—sometimes also called the *Levenshtein distance*—is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between FOOD and MONEY is at most four:

$$\texttt{\underline{F}OOD} \longrightarrow \texttt{MO\underline{O}D} \longrightarrow \texttt{MON}_{\wedge}\texttt{D} \longrightarrow \texttt{MONE\underline{D}} \longrightarrow \texttt{MONEY}$$

A better way to display this editing process is to place the words one above the other, with a gap in the first word for every insertion, and a gap in the second word for every deletion. Columns with two *different* characters correspond to substitutions. Thus, the number of editing steps is just the number of columns that don't contain the same character twice.

[3] He's hardly ever sick at sea! Then give three cheers, and one cheer more, for the hardy Captain of the *Pinafore*! Then give three cheers, and one cheer more, for the Captain of the *Pinafore*!

```
F  O  O     D
M  O  N  E  Y
```

It's fairly obvious that you can't get from `FOOD` to `MONEY` in three steps, so their edit distance is exactly four. Unfortunately, this is not so easy in general. Here's a longer example, showing that the distance between `ALGORITHM` and `ALTRUISTIC` is at most six. Is this optimal?

```
A  L  G  O  R     I     T  H  M
A  L     T  R  U  I  S  T  I  C
```

To develop a dynamic programming algorithm to compute the edit distance between two strings, we first need to develop a recursive definition. Let's say we have an m-character string A and an n-character string B. Then define $E(i, j)$ to be the edit distance between the first i characters of A and the first j characters of B. The edit distance between the entire strings A and B is $E(m, n)$.

This gap representation for edit sequences has a crucial "optimal substructure" property. Suppose we have the gap representation for the shortest edit sequence for two strings. **If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining substrings.** We can easily prove this by contradiction. If the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings. Once we figure out what should go in the last column, the Recursion Fairy will magically give us the rest of the optimal gap representation.

There are a couple of obvious base cases. The only way to convert the empty string into a string of j characters is by performing j insertions, and the only way to convert a string of i characters into the empty string is with i deletions:

$$E(i, 0) = i, \qquad E(0, j) = j.$$

If neither string is empty, there are three possibilities for the last column in the shortest edit sequence:

- **Insertion:** The last entry in the bottom row is empty. In this case, $E(i, j) = E(i - 1, j) + 1$.
- **Deletion:** The last entry in the top row is empty. In this case, $E(i, j) = E(i, j - 1) + 1$.
- **Substitution:** Both rows have characters in the last column. If the characters are the same, we don't actually have to pay for the substitution, so $E(i, j) = E(i - 1, j - 1)$. If the characters are different, then $E(i, j) = E(i - 1, j - 1) + 1$.

To summarize, the edit distance $E(i, j)$ is the smallest of these three possibilities:[4]

$$\boxed{E(i,j) = \min \left\{ \begin{array}{l} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1) + \big[A[i] \neq B[j]\big] \end{array} \right\}}$$

If we turned this recurrence directly into a recursive algorithm, we would have the following double recurrence for the running time:

$$T(m, n) = \begin{cases} O(1) & \text{if } n = 0 \text{ or } m = 0, \\ T(m, n-1) + T(m-1, n) + T(n-1, m-1) + O(1) & \text{otherwise.} \end{cases}$$

[4] Once again, I'm using Iverson's bracket notation $[P]$ to denote the *indicator variable* for the logical proposition P, which has value 1 if P is true and 0 if P is false.

I don't know of a general closed-form solution for this mess, but we can derive an upper bound by defining a new function

$$T'(N) = \max_{n+m=N} T(n,m) = \begin{cases} O(1) & \text{if } N = 0, \\ 2T(N-1) + T(N-2) + O(1) & \text{otherwise.} \end{cases}$$

The annihilator method implies that $T'(N) = O((1+\sqrt{2})^N)$. Thus, the running time of our recursive edit-distance algorithm is at most $T'(n+m) = O((1+\sqrt{2})^{n+m})$.

We can bring the running time of this algorithm down to a polynomial by building an $m \times n$ table of all possible values of $E(i,j)$. We begin by filling in the base cases, the entries in the 0th row and 0th column, each in constant time. To fill in any other entry, we need to know the values directly above it, directly to the left, and both above and to the left. If we fill in our table in the standard way—row by row from top down, each row from left to right—then whenever we reach an entry in the matrix, the entries it depends on are already available.

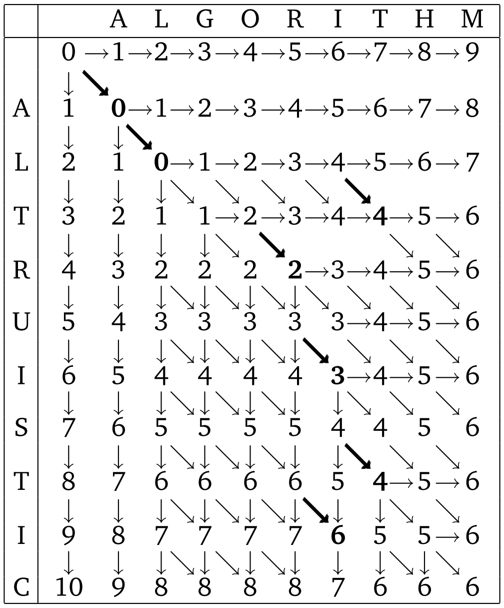
```
EDITDISTANCE($A[1..m], B[1..n]$):
  for $i \leftarrow 1$ to $m$
      Edit$[i,0] \leftarrow i$
  for $j \leftarrow 1$ to $n$
      Edit$[0,j] \leftarrow j$

  for $i \leftarrow 1$ to $m$
      for $j \leftarrow 1$ to $n$
          if $A[i] = B[j]$
              Edit$[i,j] \leftarrow \min\{$ Edit$[i-1,j]+1$,
                                   Edit$[i,j-1]+1$,
                                   Edit$[i-1,j-1]$ $\}$
          else
              Edit$[i,j] \leftarrow \min\{$ Edit$[i-1,j]+1$,
                                   Edit$[i,j-1]+1$,
                                   Edit$[i-1,j-1]+1$ $\}$

  return Edit$[m,n]$
```

Since there are $\Theta(mn)$ entries in the table, and each entry takes $\Theta(1)$ time once we know its predecessors, the total running time is $\Theta(mn)$. The algorithm uses $O(mn)$ space.

Here's the resulting table for `ALGORITHM` → `ALTRUISTIC`. Bold numbers indicate places where characters in the two strings are equal. The arrows represent the predecessor(s) that actually define each entry. Each direction of arrow corresponds to a different edit operation: horizontal=deletion, vertical=insertion, and diagonal=substitution. Bold diagonal arrows indicate "free" substitutions of a letter for itself. Any path of arrows from the top left corner to the bottom right corner of this table represents an optimal edit sequence between the two strings. (There can be many such paths.) Moreover, since we can compute these arrows in a postprocessing phase from the values stored in the table, we can reconstruct the actual optimal editing sequence in $O(n+m)$ additional time.



		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

The edit distance between `ALGORITHM` and `ALTRUISTIC` is indeed six. There are three paths through this table from the top left to the bottom right, so there are three optimal edit sequences:

```
  A L G O R I   T H M
  A L T R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C

A L G O R   I   T H M
A L T   R U I S T I C
```

*4.7 Saving Space: Divide and Conquer

Just as we did for the Fibonacci recurrence, we can reduce the space complexity of our algorithm to $O(m)$ by only storing the current and previous rows of the memoization table. However, if we throw away most of the rows in the table, we no longer have enough information to reconstruct the actual editing sequence. Now what?

Fortunately for memory-misers, in 1975 Dan Hirshberg discovered a simple divide-and-conquer strategy that allows us to compute the optimal editing sequence in $O(mn)$ time, using just $O(m)$ space. The trick is to record not just the edit distance for each pair of prefixes, but also a single position in the middle of the editing sequence for that prefix. Specifically, the optimal editing sequence that transforms $A[1\,..\,m]$ into $B[1\,..\,n]$ can be split into two smaller editing sequences, one transforming $A[1\,..\,m/2]$ into $B[1\,..\,h]$ for some integer h, the other transforming $A[m/2+1\,..\,m]$ into $B[h+1\,..\,n]$. To compute this breakpoint h, we define a second function Half(i,j) as follows:

$$\text{Half}(i,j) = \begin{cases} \infty & \text{if } i < m/2 \\ j & \text{if } i = m/2 \\ \text{Half}(i-1,j) & \text{if } i > m/2 \text{ and } \text{Edit}(i,j) = \text{Edit}(i-1,j)+1 \\ \text{Half}(i,j-1) & \text{if } i > m/2 \text{ and } \text{Edit}(i,j) = \text{Edit}(i,j-1)+1 \\ \text{Half}(i-1,j-1) & \text{otherwise} \end{cases}$$

A simple inductive argument implies that Half(m,n) is the correct value of h. We can easily modify our earlier algorithm so that it computes Half(m,n) at the same time as the edit distance Edit(m,n), all in $O(mn)$ time, using only $O(m)$ space.

Now, to compute the optimal editing sequence that transforms A into B, we recursively compute the optimal subsequences. The recursion bottoms out when one string has only constant length, in which case we can determine the optimal editing sequence by our old dynamic programming algorithm. Overall the running time of our recursive algorithm satisfies the following recurrence:

$$T(m,n) = \begin{cases} O(n) & \text{if } m \le 1 \\ O(m) & \text{if } n \le 1 \\ O(mn) + T(m/2,h) + T(m/2,n-h) & \text{otherwise} \end{cases}$$

It's easy to prove inductively that $T(m,n) = O(mn)$, no matter what the value of h is. Specifically, the entire algorithm's running time is at most twice the time for the initial dynamic programming phase.

$$\begin{aligned} T(m,n) &\le \alpha mn + T(m/2,h) + T(m/2,n-h) \\ &\le \alpha mn + 2\alpha mh/2 + 2\alpha m(n-h)/2 && \text{[inductive hypothesis]} \\ &= 2\alpha mn \end{aligned}$$

A similar inductive argument implies that the algorithm uses only $O(n+m)$ space.

4.8 Optimal Binary Search Trees

A few lectures ago we developed a recursive algorithm for the optimal binary search tree problem. We are given a sorted array $A[1\,..\,n]$ of search keys and an array $f[1\,..\,n]$ of frequency counts, where $f[i]$ is the number of searches to $A[i]$. Our task is to construct a binary search tree for that set such that the total cost of all the searches is as small as possible. We developed the following recurrence for this problem:

$$\mathit{OptCost}(f[1\,..\,n]) = \min_{1\le r\le n} \left\{ \mathit{OptCost}(f[1\,..\,r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; \mathit{OptCost}(f[r+1\,..\,n]) \right\}$$

To put this recurrence in more standard form, fix the frequency array f, and let $S(i,j)$ denote the total search time in the optimal search tree for the subarray $A[i\,..\,j]$. To simplify notation a bit, let $F(i,j)$ denote the total frequency count for all the keys in the interval $A[i\,..\,j]$:

$$F(i,j) = \sum_{k=i}^{j} f[k]$$

We can now write

$$\boxed{S(i,j) = \begin{cases} 0 & \text{if } j < i \\ F(i,j) + \displaystyle\min_{i\le r\le j} \big(S(i,r-1) + S(r+1,j)\big) & \text{otherwise} \end{cases}}$$

The base case might look a little weird, but all it means is that the total cost for searching an empty set of keys is zero.

The algorithm will be somewhat simpler and more efficient if we precompute all possible values of $F(i,j)$ and store them in an array. Computing each value $F(i,j)$ using a separate for-loop would $O(n^3)$ time. A better approach is to turn the recurrence

$$F(i,j) = \begin{cases} f[i] & \text{if } i = j \\ F(i,j-1) + f[j] & \text{otherwise} \end{cases}$$

into the following $O(n^2)$-time dynamic programming algorithm:

```
InitF($f[1\,..\,n]$):
  for $i \leftarrow 1$ to $n$
      $F[i, i-1] \leftarrow 0$
      for $j \leftarrow i$ to $n$
          $F[i,j] \leftarrow F[i,j-1] + f[i]$
```

This will be used as an initialization subroutine in our final algorithm.

So now let's compute the optimal search tree cost $S(1,n)$ from the bottom up. We can store all intermediate results in a table $S[1\,..\,n, 0\,..\,n]$. Only the entries $S[i,j]$ with $j \geq i-1$ will actually be used. The base case of the recurrence tells us that any entry of the form $S[i,i-1]$ can immediately be set to 0. For any other entry $S[i,j]$, we can use the following algorithm fragment, which comes directly from the recurrence:

```
ComputeS($i, j$):
  $S[i,j] \leftarrow \infty$
  for $r \leftarrow i$ to $j$
      $tmp \leftarrow S[i,r-1] + S[r+1,j]$
      if $S[i,j] > tmp$
          $S[i,j] \leftarrow tmp$
  $S[i,j] \leftarrow S[i,j] + F[i,j]$
```

The only question left is what order to fill in the table.

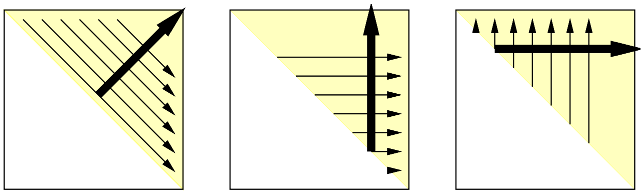
Each entry $S[i,j]$ depends on all entries $S[i,r-1]$ and $S[r+1,j]$ with $i \leq k \leq j$. In other words, every entry in the table depends on all the entries directly to the left or directly below. In order to fill the table efficiently, we must choose an order that computes all those entries before $S[i,j]$. There are at least three different orders that satisfy this constraint. The one that occurs to most people first is to scan through the table one diagonal at a time, starting with the trivial base cases $S[i,i-1]$. The complete algorithm looks like this:

```
OptimalSearchTree($f[1\,..\,n]$):
  InitF($f[1\,..\,n]$)
  for $i \leftarrow 1$ to $n$
      $S[i, i-1] \leftarrow 0$
  for $d \leftarrow 0$ to $n-1$
      for $i \leftarrow 1$ to $n-d$
          ComputeS($i, i+d$)
  return $S[1,n]$
```

We could also traverse the array row by row from the bottom up, traversing each row from left to right, or column by column from left to right, traversing each columns from the bottom up. These two orders give us the following algorithms:

```
OPTIMALSEARCHTREE2(f[1 .. n]):
  INITF(f[1 .. n])
  for i ← n downto 1
      S[i, i − 1] ← 0
      for j ← i to n
          COMPUTES(i, j)
  return S[1, n]
```

```
OPTIMALSEARCHTREE3(f[1 .. n]):
  INITF(f[1 .. n])
  for j ← 0 to n
      S[j + 1, j] ← 0
      for i ← j downto 1
          COMPUTES(i, j)
  return S[1, n]
```



Three different orders to fill in the table $S[i, j]$.

No matter which of these three orders we actually use, the resulting algorithm runs in $\boxed{\Theta(n^3) \text{ time}}$ and uses $\boxed{\Theta(n^2) \text{ space}}$.

We could have predicted this from the original recursive formulation.

$$S(i,j) = \begin{cases} 0 & \text{if } j = i - i \\ F(i,j) + \min\limits_{i \le r \le j} \big(S(i, r-1) + S(r+1, j)\big) & \text{otherwise} \end{cases}$$

First, the function has two arguments, each of which can take on any value between 1 and n, so we probably need a table of size $O(n^2)$. Next, there are *three* variables in the recurrence (i, j, and r), each of which can take any value between 1 and n, so it should take us $O(n^3)$ time to fill the table.

In general, you can get an easy estimate of the time and space bounds for any dynamic programming algorithm by looking at the recurrence. The time bound is determined by how many values *all* the variables can have, and the space bound is determined by how many values the parameters of the function can have. For example, the (completely made up) recurrence

$$F(i,j,k,l,m) = \min_{0 \le p \le i} \max_{0 \le q \le j} \sum_{r=1}^{k-m} F(i-p, j-q, r, l-1, m-r)$$

should immediately suggest a dynamic programming algorithm to compute $F(n,n,n,n,n)$ in $O(n^8)$ time and $O(n^5)$ space. This simple rule of thumb usually gives us the right time bound to shoot for.

*4.9 Montonicity Helps

But not always! In fact, the algorithm I've described is *not* the most efficient algorithm for computing optimal binary search trees. Let $R[i,j]$ denote the root of the optimal search tree for $A[i \mathinner{..} j]$. Donald Knuth proved the following nice monotonicity property for optimal subtrees: if we move either end of the subarray, the optimal root moves in the same direction or not at all, or more formally:

$$\boxed{R[i, j-1] \le R[i,j] \le R[i+1, j] \text{ for all } i \text{ and } j.}$$

This (nontrivial!) observation leads to the following more efficient algorithm:

```
FASTEROPTIMALSEARCHTREE($f[1..n]$):
  INITF($f[1..n]$)
  for $i \leftarrow n$ downto 1
      $S[i, i-1] \leftarrow 0$
      $R[i, i-1] \leftarrow i$
      for $j \leftarrow i$ to $n$
          COMPUTESANDR($i, j$)
          return $S[1, n]$
```

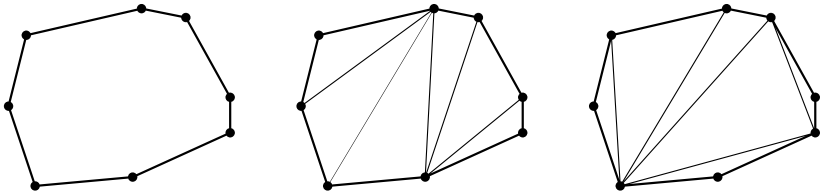
```
COMPUTESANDR($f[1..n]$):
  $S[i, j] \leftarrow \infty$
  for $r \leftarrow R[i, j-1]$ to $j$
      $tmp \leftarrow S[i, r-1] + S[r+1, j]$
      if $S[i, j] > tmp$
          $S[i, j] \leftarrow tmp$
          $R[i, j] \leftarrow r$
  $S[i, j] \leftarrow S[i, j] + F[i, j]$
```

It's not hard to see the r increases monotonically from i to n during each iteration of the *outermost* for loop. Consequently, the innermost for loop iterates at most n times during a single iteration of the outermost loop, so the total running time of the algorithm is $O(n^2)$.

If we formulate the problem slightly differently, this algorithm can be improved even further. Suppose we require the optimum *external* binary tree, where the keys $A[1..n]$ are all stored at the leaves, and intermediate pivot values are stored at the internal nodes. An algorithm due to Te Ching Hu and Alan Tucker[5] computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

4.10 Optimal Triangulations of Convex Polygons

A *convex polygon* is a circular chain of line segments, arranged so none of the corners point inwards—imagine a rubber band stretched around a bunch of nails. (This is technically not the best definition, but it'll do for now.) A *diagonal* is a line segment that cuts across the interior of the polygon from one corner to another. A simple induction argument (hint, hint) implies that any n-sided convex polygon can be split into $n-2$ triangles by cutting along $n-3$ different diagonals. This collection of triangles is called a *triangulation* of the polygon. Triangulations are incredibly useful in computer graphics—most graphics hardware is built to draw triangles incredibly quickly, but to draw anything more complicated, you usually have to break it into triangles first.



A convex polygon and two of its many possible triangulations.

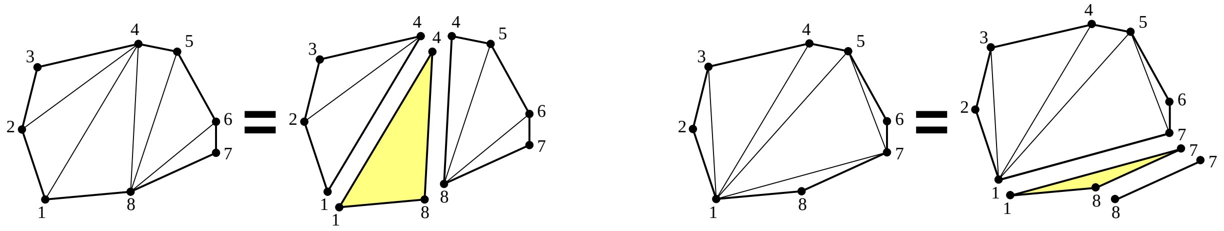
There are several different ways to triangulate any convex polygon. Suppose we want to find the triangulation that requires the least amount of ink to draw, or in other words, the triangulation where the total perimeter of the triangles is as small as possible. To make things concrete, let's label the corners of the polygon from 1 to n, starting at the bottom of the polygon and going clockwise. We'll need the following subroutines to compute the perimeter of a triangle joining three corners using their x- and y-coordinates:

```
$\Delta(i, j, k)$ :
  return DIST($i, j$) + DIST($j, k$) + DIST($i, k$)
```

```
DIST($i, j$) :
  return $\sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}$
```

[5] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Applied Math.* 21:514–532, 1971. For a slightly simpler algorithm with the same running time, see A. M. Garsia and M. L. Wachs, A new algorithms for minimal binary search trees, *SIAM J. Comput.* 6:622–642, 1977. The original correctness proofs for both algorithms are rather intricate; for simpler proofs, see Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science*, 180:309-324, 1997.

In order to get a dynamic programming algorithm, we first need a recursive formulation of the minimum-length triangulation. To do that, we really need some kind of recursive definition of a *triangulation*! Notice that in any triangulation, exactly one triangle uses both the first corner and the last corner of the polygon. If we remove that triangle, what's left over is two smaller triangulations. The base case of this recursive definition is a 'polygon' with just two corners. Notice that at any point in the recursion, we have a polygon joining a contiguous subset of the original corners.



Two examples of the recursive definition of a triangulation.

Building on this recursive definition, we can now recursively define the total length of the minimum-length triangulation. In the best triangulation, if we remove the 'base' triangle, what remains must be the optimal triangulation of the two smaller polygons. So we just have choose the best triangle to attach to the first and last corners, and let the recursion fairy take care of the rest:

$$M(i,j) = \begin{cases} 0 & \text{if } j = i+1 \\ \displaystyle\min_{i<k<j} \big(\Delta(i,j,k) + M(i,k) + M(k,j)\big) & \text{otherwise} \end{cases}$$

What we're looking for is $M(1,n)$.

If you think this looks similar to the recurrence for $S(i,j)$, the cost of an optimal binary search tree, you're absolutely right. We can build up intermediate results in a two-dimensional table, starting with the base cases $M[i,i+1] = 0$ and working our way up. We can use the following algorithm fragment to compute a generic entry $M[i,j]$:

```
COMPUTEM($(i,j)$):
  $M[i,j] \leftarrow \infty$
  for $k \leftarrow i+1$ to $j-1$
      $tmp \leftarrow \Delta(i,j,k) + M[i,k] + M[k,j]$
      if $M[i,j] > tmp$
          $M[i,j] \leftarrow tmp$
```

As in the optimal search tree problem, each table entry $M[i,j]$ depends on all the entries directly to the left or directly below, so we can use any of the orders described earlier to fill the table.

```
MINTRIANGULATION:
  for $i \leftarrow 1$ to $n-1$
      $M[i,i+1] \leftarrow 0$
  for $d \leftarrow 2$ to $n-1$
      for $i \leftarrow 1$ to $n-d$
          COMPUTEM($(i,i+d)$)
  return $M[1,n]$
```

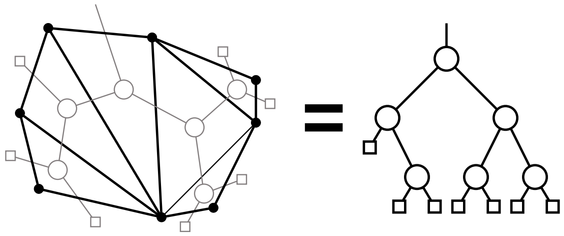
```
MINTRIANGULATION2:
  for $i \leftarrow n$ downto 1
      $M[i,i+1] \leftarrow 0$
      for $j \leftarrow i+2$ to $n$
          COMPUTEM($(i,j)$)
  return $M[1,n]$
```

```
MINTRIANGULATION3:
  for $j \leftarrow 2$ to $n$
      $M[j-1,j] \leftarrow 0$
      for $i \leftarrow j-1$ downto 1
          COMPUTEM($(i,j)$)
  return $M[1,n]$
```

In all three cases, the algorithm runs in $\Theta(n^3)$ time and uses $\Theta(n^2)$ space, just as we should have guessed from the recurrence.

4.11 It's the same problem!

Actually, the last two problems are both special cases of the same meta-problem: computing optimal *Catalan* structures. There is a straightforward one-to-one correspondence between the set of triangulations of a convex n-gon and the set of binary trees with $n-2$ nodes. In effect, these two problems differ only in the cost function for a single node/triangle.



A polygon triangulation and the corresponding binary tree. (Squares represent null pointers.)

A third problem that fits into the same mold is the infamous matrix chain multiplication problem. Using the standard algorithm, we can multiply a $p \times q$ matrix by a $q \times r$ matrix using $O(pqr)$ arithmetic operations; the result is a $p \times r$ matrix. If we have three matrices to multiply, the cost depends on which pair we multiply first. For example, suppose A and C are 1000×2 matrices and B is a 2×1000 matrix. There are two different ways to compute the threefold product ABC:

- $\boldsymbol{(AB)C}$**:** Computing AB takes $1000{\cdot}2{\cdot}1000 = 2\,000\,000$ operations and produces a $1000{\times}1000$ matrix. Multiplying this matrix by C takes $1000 \cdot 1000 \cdot 2 = 2\,000\,000$ additional operations. So the total cost of $(AB)C$ is $4\,000\,000$ operations.
- $\boldsymbol{A(BC)}$**:** Computing BC takes $2 \cdot 1000 \cdot 2 = 4000$ operations and produces a 2×2 matrix. Multiplying A by this matrix takes $1000 \cdot 2 \cdot 2 = 4\,000$ additional operations. So the total cost of $A(BC)$ is only 8000 operations.

Now suppose we are given an array $D[0\,..\,n]$ as input, indicating that each matrix M_i has $D[i-1]$ rows and $D[i]$ columns. We have an exponential number of possible ways to compute the n-fold product $\prod_{i=1}^{n} M_i$. The following dynamic programming algorithm computes the number of arithmetic operations for the best possible parenthesization:

```
MATRIXCHAINMULT:
  for $i \leftarrow n$ downto $1$
      $M[i, i+1] \leftarrow 0$
      for $j \leftarrow i+2$ to $n$
          COMPUTEM$(i, j)$
  return $M[1, n]$
```

```
COMPUTEM$(i, j)$:
  $M[i, j] \leftarrow \infty$
  for $k \leftarrow i+1$ to $j-1$
      $tmp \leftarrow (D[i] \cdot D[j] \cdot D[k]) + M[i, k] + M[k, j]$
      if $M[i, j] > tmp$
          $M[i, j] \leftarrow tmp$
```

The derivation of this algorithm is left as a simple exercise.

4.12 More Examples

We've already seen two other examples of recursive algorithms that we can significantly speed up via dynamic programming.

4.12.1 Subset Sum

Recall from the very first lecture that the *Subset Sum* problem asks, given a set X of positive integers (represented as an array $X[1\,..\,n]$ and an integer T, whether any subset of X sums to T. In that lecture, we developed a recursive algorithm which can be reformulated as follows. Fix the original input array $X[1\,..\,n]$ and the original target sum T, and define the boolean function

$$S(i,t) = \text{some subset of } X[i\,..\,n] \text{ sums to } t.$$

Our goal is to compute $S(1,T)$, using the recurrence

$$S(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0, \\ \textsc{False} & \text{if } t < 0 \text{ or } i > n, \\ S(i+1,t) \vee S(i+1, t - X[i]) & \text{otherwise.} \end{cases}$$

Observe that there are only nT possible values for the input parameters that lead to the interesting case of this recurrence, so storing the results of all such subproblems requires $\boxed{O(mn)\text{ space}}$. If $S(i+1,t)$ and $S(i+1,t-X[i])$ are already known, we can compute $S(i,t)$ in constant time, so memoizing this recurrence gives us and algorithm that runs in $\boxed{O(nT)\text{ time}}$.[6] To turn this into an explicit dynamic programming algorithm, we only need to consider the subproblems $S(i,t)$ in the proper order:

```
SUBSETSUM($X[1..n], T$):
  $S[n+1, 0] \gets$ TRUE
  for $t \gets 1$ to $T$
      $S[n+1, t] \gets$ FALSE

  for $i \gets n$ downto $1$
      $S[i, 0] =$ TRUE
      for $t \gets 1$ to $X[i] - 1$
          $S[i,t] \gets S[i+1,t]$        ⟨⟨Avoid the case $t < 0$⟩⟩
      for $t \gets X[i]$ to $T$
          $S[i,t] \gets S[i+1,t] \vee S[i+1, t - X[i]]$

  return $S[1, T]$
```

This direct algorithm clearly always uses $\boxed{O(nT)\text{ time and space}}$. In particular, if T is significantly larger than 2^n, this algorithm is actually slower than our naïve recursive algorithm. Dynamic programming isn't *always* an improvement!

4.12.2 Longest Increasing Subsequence

We also developed a recurrence for the longest increasing subsequence problem. Fix the original input array $A[1\,..\,n]$ with a sentinel value $A[0] = -\infty$. Let $L(i,j)$ denote the length of the longest increasing subsequence of $A[j\,..\,n]$ with all elements larger than $A[i]$. Our goal is to compute $L(0,1) - 1$. (The -1 removes the sentinel $-\infty$.) For any $i < j$, our recurrence can be stated as follows:

$$L(i,j) = \begin{cases} 0 & \text{if } j > n \\ L(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{L(i,j+1),\ 1 + L(j, j+1)\} & \text{otherwise} \end{cases}$$

[6] This does not contradict our earlier upper bound of $O(2^n)$. Both upper bounds are correct. Which bound is actually better depends on the size of T.

The recurrence suggests that our algorithm should use $O(n^2)$ time and space, since the input parameters i and j each can take n different values. To get an explicit dynamic programming algorithm, we only need to ensure that both $L(i, j+1)$ and $L(j, j+1)$ are considered before $L(i, j)$, for all i and j.

```
LIS($A[1..n]$):
  $A[0] \leftarrow -\infty$                ⟨⟨Add a sentinel⟩⟩
  for $i \leftarrow 0$ to $n$               ⟨⟨Base cases⟩⟩
      $L[i, n+1] \leftarrow 0$
  for $j \leftarrow n$ downto $1$
      for $i \leftarrow 0$ to $j-1$
          if $A[i] \geq A[j]$
              $L[i, j] \leftarrow L[i, j+1]$
          else
              $L[i, j] \leftarrow \max\{L[i, j+1],\ 1 + L[j, j+1]\}$
  return $L[0, 1] - 1$               ⟨⟨Don't count the sentinel⟩⟩
```

As predicted, this algorithm clearly uses $\boxed{O(n^2) \text{ time and space}}$. We can reduce the space to $O(n)$ by only maintaining the two most recent columns of the table, $L[\cdot, j]$ and $L[\cdot, j+1]$.

This is not the only recursive strategy we could use for computing longest increasing subsequences. Here is another recurrence that gives us the $O(n)$ space bound for free. Let $L'(i)$ denote the length of the longest increasing subsequence of $A[i..n]$ that starts with $A[i]$. Our goal is to compute $L'(0) - 1$. To define $L'(i)$ recursively, we only need to specify the *second* element in subsequence; the Recursion Fairy will do the rest.

$$L'(i) = 1 + \max\left\{L'(j) \mid j > i \text{ and } A[j] > A[i]\right\}$$

Here, I'm assuming that $\max \varnothing = 0$, so that the base case is $L'(n) = 1$ falls out of the recurrence automatically. Memoizing this recurrence requires $O(n)$ space, and the resulting algorithm runs in $O(n^2)$ time. To transform this into a dynamic programming algorithm, we only need to guarantee that $L'(j)$ is computed before $L'(i)$ whenever $i < j$.

```
LIS2($A[1..n]$):
  $A[0] = -\infty$                  ⟨⟨Add a sentinel⟩⟩
  for $i \leftarrow n$ downto $0$
      $L'[i] \leftarrow 1$
      for $j \leftarrow i+1$ to $n$
          if $A[j] > A[i]$ and $1 + L'[j] > L'[i]$
              $L'[i] \leftarrow 1 + L'[j]$
  return $L'[0] - 1$               ⟨⟨Don't count the sentinel⟩⟩
```

> *The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind. And greed—mark my words—will save not only Teldar Paper but the other malfunctioning corporation called the USA.*
>
> — Michael Douglas as Gordon Gekko, *Wall Street* (1987)

> *There is always an easy solution to every human problem—*
> *neat, plausible, and wrong.*
>
> — H. L. Mencken, *New York Evening Mail* (November 16, 1917)

A Greedy Algorithms

A.1 Storing Files on Tape

Suppose we have a set of n files that we want to store on a tape. In the future, users will want to read those files from the tape. Reading a file from tape isn't like reading from disk; first we have to fast-forward past all the other files, and that takes a significant amount of time. Let $L[1 \,..\, n]$ be an array listing the lengths of each file; specifically, file i has length $L[i]$. If the files are stored in order from 1 to n, then the cost of accessing the kth file is

$$\mathit{cost}(k) = \sum_{i=1}^{k} L[i].$$

The cost reflects the fact that before we read file k we must first scan past all the earlier files on the tape. If we assume for the moment that each file is equally likely to be accessed, then the *expected* cost of searching for a random file is

$$\mathrm{E}[\mathit{cost}] = \sum_{k=1}^{n} \frac{\mathit{cost}(k)}{n} = \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[i]}{n}.$$

If we change the order of the files on the tape, we change the cost of accessing the files; some files become more expensive to read, but others become cheaper. Different file orders are likely to result in different expected costs. Specifically, let $\pi(i)$ denote the index of the file stored at position i on the tape. Then the expected cost of the permutation π is

$$\mathrm{E}[\mathit{cost}(\pi)] = \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[\pi(i)]}{n}.$$

Which order should we use if we want the expected cost to be as small as possible? The answer is intuitively clear; we should store the files in order from shortest to longest. So let's prove this.

Lemma 1. $\mathrm{E}[\mathit{cost}(\pi)]$ *is minimized when* $L[\pi(i)] \le L[\pi(i+1)]$ *for all* i.

Proof: Suppose $L[\pi(i)] > L[\pi(i+1)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the expected cost by $(L[b] - L[a])/n$. But this change is an improvement, because $L[b] < L[a]$. Thus, if the files are out of order, we can improve the expected cost by swapping some mis-ordered adjacent pair. □

This example gives us our first *greedy algorithm*. To minimize the *total* expected cost of accessing the files, we put the file that is cheapest to access first, and then recursively write everything else; no backtracking, no dynamic programming, just make the best local choice and blindly plow ahead. If we use an efficient sorting algorithm, the running time is clearly $O(n \log n)$, plus the time required to actually write the files. To prove the greedy algorithm is actually correct, we simply prove that the output of any other algorithm can be improved by some sort of swap.

Let's generalize this idea further. Suppose we are also given an array $f[1 \,..\, n]$ of *access frequencies* for each file; file i will be accessed exactly $f[i]$ times over the lifetime of the tape. Now the *total* cost of accessing all the files on the tape is

$$\Sigma \mathit{cost}(\pi) = \sum_{k=1}^{n} \left(f[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)] \right) = \sum_{k=1}^{n} \sum_{i=1}^{k} \big(f[\pi(k)] \cdot L[\pi(i)] \big).$$

Now what order should store the files if we want to minimize the total cost?

We've already proved that if all the frequencies are equal, then we should sort the files by increasing size. If the frequencies are all different but the file lengths $L[i]$ are all equal, then intuitively, we should sort the files by *decreasing* access frequency, with the most-accessed file first. And in fact, this is not hard to prove by modifying the proof of Lemma 1. But what if the sizes and the frequencies are both different? In this case, we should sort the files by the ratio L/f.

Lemma 2. *$\Sigma \mathit{cost}(\pi)$ is minimized when $\dfrac{L[\pi(i)]}{F[\pi(i)]} \le \dfrac{L[\pi(i+1)]}{F[\pi(i+1)]}$ for all i.*

Proof: Suppose $L[\pi(i)]/F[\pi(i)] > L[\pi(i+1)]/F[\pi(i+i)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the total cost by $L[b]F[a] - L[a]F[a]$. But this change is an improvement, since

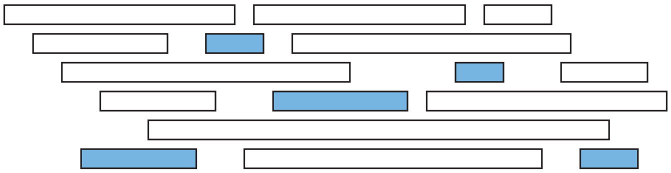
$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \implies L[b]F[a] - L[a]F[a] < 0.$$

Thus, if the files are out of order, we can improve the total cost by swapping some mis-ordered adjacent pair. □

A.2 Scheduling Classes

The next example is slightly less trivial. Suppose you decide to drop out of computer science at the last minute and change your major to Applied Chaos. The Applied Chaos department has all of its classes on the same day every week, referred to as "Soberday" by the students (but interestingly, *not* by the faculty). Every class has a different start time and a different ending time: AC 101 ('Toilet Paper Landscape Architecture') starts at 10:27pm and ends at 11:51pm; AC 666 ('Immanentizing the Eschaton') starts at 4:18pm and ends at 7:06pm, and so on. In the interests of graduating as quickly as possible, you want to register for as many classes as you can. (Applied Chaos classes don't require any actual *work*.) The University's registration computer won't let you register for overlapping classes, and no one in the department knows how to override this 'feature'. Which classes should you take?

More formally, suppose you are given two arrays $S[1 \,..\, n]$ and $F[1 \,..\, n]$ listing the start and finish times of each class. Your task is to choose the largest possible subset $X \in \{1, 2, \ldots, n\}$ so that for any pair $i, j \in X$, either $S[i] > F[j]$ or $S[j] > F[i]$. We can illustrate the problem by drawing each class as a rectangle whose left and right x-coordinates show the start and finish times. The goal is to find a largest subset of rectangles that do not overlap vertically.



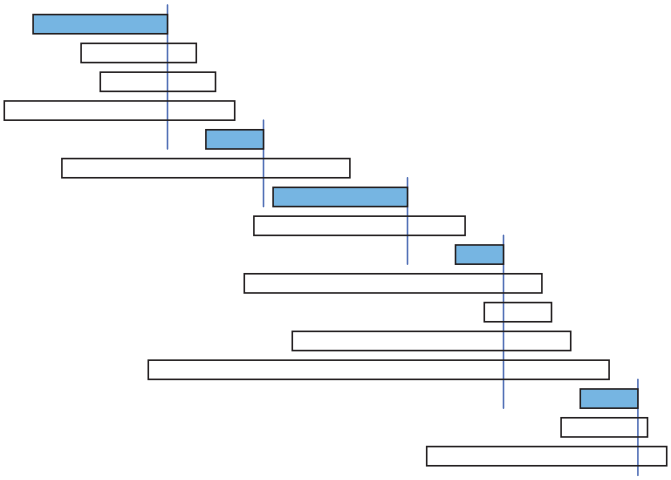
A maximal conflict-free schedule for a set of classes.

This problem has a fairly simple recursive solution, based on the observation that either you take class 1 or you don't. Let B_4 be the set of classes that end *before* class 1 starts, and let L_8 be the set of classes that start *late*r than class 1 ends:

$$B_4 = \{i \mid 2 \leq i \leq n \text{ and } F[i] < S[1]\} \qquad L_8 = \{i \mid 2 \leq i \leq n \text{ and } S[i] > F[1]\}$$

If class 1 is in the optimal schedule, then so are the optimal schedules for B_4 and L_8, which we can find recursively. If not, we can find the optimal schedule for $\{2, 3, \ldots, n\}$ recursively. So we should try both choices and take whichever one gives the better schedule. Evaluating this recursive algorithm from the bottom up gives us a dynamic programming algorithm that runs in $O(n^2)$ time. I won't bother to go through the details, because we can do better.[1]

Intuitively, we'd like the first class to finish as early as possible, because that leaves us with the most remaining classes. If this greedy strategy works, it suggests the following very simple algorithm. Scan through the classes in order of finish time; whenever you encounter a class that doesn't conflict with your latest class so far, take it!



The same classes sorted by finish times and the greedy schedule.

We can write the greedy algorithm somewhat more formally as follows. (Hopefully the first line is understandable.)

```
GREEDYSCHEDULE($S[1..n], F[1..n]$):
  sort $F$ and permute $S$ to match
  $count \leftarrow 1$
  $X[count] \leftarrow 1$
  for $i \leftarrow 2$ to $n$
      if $S[i] > F[X[count]]$
          $count \leftarrow count + 1$
          $X[count] \leftarrow i$
  return $X[1..count]$
```

[1] But you should still work out the details yourself. The dynamic programming algorithm can be used to find the "best" schedule for any definition of "best", but the greedy algorithm I'm about to describe only works that "best" means "biggest". Also, you need the practice.

This algorithm clearly runs in $O(n \log n)$ time.

To prove that this algorithm actually gives us a maximal conflict-free schedule, we use an exchange argument, similar to the one we used for tape sorting. We are not claiming that the greedy schedule is the *only* maximal schedule; there could be others. (See the figures on the previous page.) All we can claim is that at least one of the maximal schedules is the one that the greedy algorithm produces.

Lemma 3. *At least one maximal conflict-free schedule includes the class that finishes first.*

Proof: Let f be the class that finishes first. Suppose we have a maximal conflict-free schedule X that does not include f. Let g be the first class in X to finish. Since f finishes before g does, f cannot conflict with any class in the set $S \setminus \{g\}$. Thus, the schedule $X' = X \cup \{f\} \setminus \{g\}$ is also conflict-free. Since X' has the same size as X, it is also maximal. □

To finish the proof, we call on our old friend, induction.

Theorem 4. *The greedy schedule is an optimal schedule.*

Proof: Let f be the class that finishes first, and let L be the subset of classes the start after f finishes. The previous lemma implies that some optimal schedule contains f, so the best schedule that contains f is an optimal schedule. The best schedule that includes f must contain an optimal schedule for the classes that do not conflict with f, that is, an optimal schedule for L. The greedy algorithm chooses f and then, by the inductive hypothesis, computes an optimal schedule of classes from L. □

The proof might be easier to understand if we unroll the induction slightly.

Proof: Let $\langle g_1, g_2, \ldots, g_k \rangle$ be the sequence of classes chosen by the greedy algorithm. Suppose we have a maximal conflict-free schedule of the form

$$\langle g_1, g_2, \ldots, g_{j-1}, c_j, c_{j+1}, \ldots, c_m \rangle,$$

where the classes c_i are different from the classes chosen by the greedy algorithm. By construction, the jth greedy choice g_j does not conflict with any earlier class $g_1, g_2, \ldots, g_{j-1}$, and since our schedule is conflict-free, neither does c_j. Moreover, g_j has the *earliest* finish time among all classes that don't conflict with the earlier classes; in particular, g_j finishes before c_j. This implies that g_j does not conflict with any of the later classes $c_{j+1}, \ldots, c_m$. Thus, the schedule

$$\langle g_1, g_2, \ldots, g_{j-1}, g_j, c_{j+1}, \ldots, c_m \rangle,$$

is conflict-free. (This is just a generalization of Lemma 3, which considers the case $j = 1$.) By induction, it now follows that there is an optimal schedule $\langle g_1, g_2, \ldots, g_k, c_{k+1}, \ldots, c_m \rangle$ that includes every class chosen by the greedy algorithm. But this is impossible unless $k = m$; if there were a class c_{k+1} that does not conflict with g_k, the greedy algorithm would choose more than k classes. □

A.3 General Structure

The basic structure of this correctness proof is exactly the same as for the tape-sorting problem: an inductive exchange argument.

- Assume that there is an optimal solution that is different from the greedy solution.
- Find the 'first' difference between the two solutions.
- Argue that we can exchange the optimal choice for the greedy choice without degrading the solution.

This argument implies by induction that there is an optimal solution that contains the entire greedy solution. Sometimes, as in the scheduling problem, an additional step is required to show no optimal solution *strictly* improves the greedy solution.

A.4 Huffman codes

A *binary code* assigns a string of 0s and 1s to each character in the alphabet. A binary code is *prefix-free* if no code is a prefix of any other. 7-bit ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code, but it is not prefix-free; for example, the code for S $(\cdots)$ includes the code for E $(\cdot)$ as a prefix. Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; 0 for left, 1 for right. The length of a codeword for a symbol is the depth of the corresponding leaf. (Note that the code tree is *not* a binary search tree. We don't care at all about the sorted order of symbols at the leaves. (In fact. the symbols may not have a well-defined order!)

Suppose we want to encode messages in an n-character alphabet so that the encoded message is as short as possible. Specifically, given an array frequency counts $f[1\,..\,n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message:[2]

$$\sum_{i=1}^{n} f[i] \cdot \mathit{depth}(i).$$

In 1952, David Huffman developed the following greedy algorithm to produce such an optimal code:

HUFFMAN: Merge the two least frequent letters and recurse.

For example, suppose we want to encode the following helpfully self-descriptive sentence, discovered by Lee Sallows:[3]

> This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

To keep things simple, let's forget about the forty-four spaces, nineteen apostrophes, nineteen commas, three hyphens, and one period, and just encode the letters. Here's the frequency table:

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

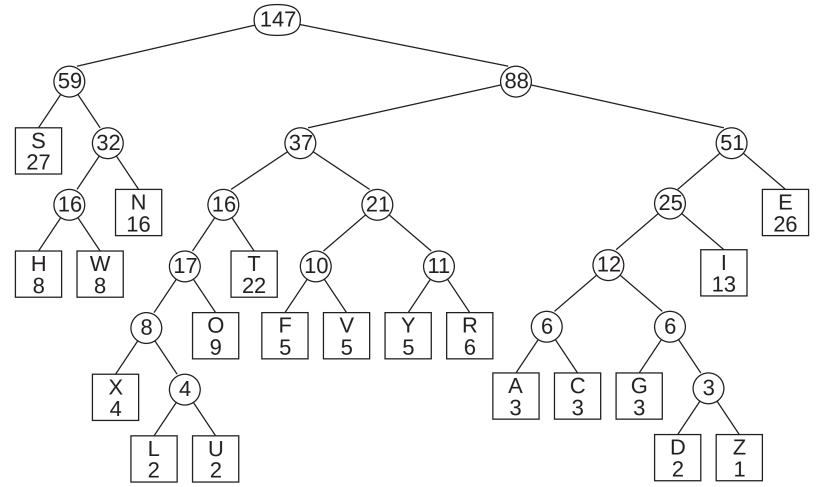
[2] This looks almost exactly like the cost of a binary search tree, but the optimization problem is very different: code trees are **not** search trees!

[3] A. K. Dewdney. Computer recreations. *Scientific American*, October 1984. Douglas Hofstadter published a few earlier examples of Lee Sallows' self-descriptive sentences in his *Scientific American* column in January 1982.

Huffman's algorithm picks out the two least frequent letters, breaking ties arbitrarily—in this case, say, Z and D—and merges them together into a single new character DZ with frequency 3. This new character becomes an internal node in the code tree we are constructing, with Z and D as its children; it doesn't matter which child is which. The algorithm then recursively constructs a Huffman code for the new frequency table

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

After 19 merges, all 20 characters have been merged together. The record of merges gives us our code tree. The algorithm makes a number of arbitrary choices; as a result, there are actually several different Huffman codes. One such code is shown below.



A Huffman code for Lee Sallows' self-descriptive sentence; the numbers are frequencies for merged characters

For example, the code for A is 110000, and the code for S is 00. The encoded message starts like this:

$$\underbrace{1001}_{\mathsf{T}}\,\underbrace{0100}_{\mathsf{H}}\,\underbrace{1101}_{\mathsf{I}}\,\underbrace{00}_{\mathsf{S}}\,\underbrace{00}_{\mathsf{S}}\,\underbrace{111}_{\mathsf{E}}\,\underbrace{011}_{\mathsf{N}}\,\underbrace{1001}_{\mathsf{T}}\,\underbrace{111}_{\mathsf{E}}\,\underbrace{011}_{\mathsf{N}}\,\underbrace{110001}_{\mathsf{C}}\,\underbrace{111}_{\mathsf{E}}\,\underbrace{110001}_{\mathsf{C}}\,\underbrace{10001}_{\mathsf{O}}\,\underbrace{011}_{\mathsf{N}}\,\underbrace{1001}_{\mathsf{T}}\,\underbrace{110000}_{\mathsf{A}}\,\underbrace{1101}_{\mathsf{I}}\ldots$$

Here is the list of costs for encoding each character, along with that character's contribution to the total length of the encoded message:

char.	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq.	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
× depth	6	6	7	3	5	6	4	4	7	3	4	4	2	4	7	5	4	6	5	7
= total	18	18	14	78	25	18	32	52	14	48	36	24	54	88	14	25	32	24	25	7

Altogether, the encoded message is 646 bits long. Different Huffman codes would assign different codes, possibly with different lengths, to various characters, but the overall length of the encoded message is the same for any Huffman code: 646 bits.

Given the simple structure of Huffman's algorithm, it's rather surprising that it produces an *optimal* prefix-free binary code. Encoding Lee Sallows' sentence using *any* prefix-free code requires at least 646 bits! Fortunately, the recursive structure makes this claim easy to prove using an exchange argument, similar to our earlier optimality proofs. We start by proving that the algorithm's very first choice is correct.

Lemma 5. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: I'll actually prove a stronger statement: There is an optimal code in which x and y are siblings *and* have the largest depth of any leaf.

Let T be an optimal code tree with depth d. Since T is a full binary tree, it has at least two leaves at depth d that are siblings. (Verify this by induction!) Suppose those two leaves are *not* x and y, but some other characters a and b.

Let T' be the code tree obtained by swapping x and a. The depth of x increases by some amount Δ, and the depth of a decreases by the same amount. Thus,

$$\mathit{cost}(T') = \mathit{cost}(T) - (f[a] - f[x])\Delta.$$

By assumption, x is one of the two least frequent characters, but a is not, which implies that $f[a] \geq f[x]$. Thus, swapping x and a does not increase the total cost of the code. Since T was an optimal code tree, swapping x and a does not decrease the cost, either. Thus, T' is also an optimal code tree (and incidentally, $f[a]$ actually equals $f[x]$).

Similarly, swapping y and b must give yet another optimal code tree. In this final optimal code tree, x and y as maximum-depth siblings, as required. □

Now optimality is guaranteed by our dear friend the Recursion Fairy! Essentially we're relying on the following recursive definition for a full binary tree: either a single node, or a full binary tree where some leaf has been replaced by an internal node with two leaf children.

Theorem 6. *Huffman codes are optimal prefix-free binary codes.*

Proof: If the message has only one or two different characters, the theorem is trivial.

Otherwise, let $f[1 \,..\, n]$ be the original input frequencies, where without loss of generality, $f[1]$ and $f[2]$ are the two smallest. To keep things simple, let $f[n+1] = f[1] + f[2]$. By the previous lemma, we know that some optimal code for $f[1 \,..\, n]$ has characters 1 and 2 as siblings.

Let T' be the Huffman code tree for $f[3 \,..\, n+1]$; the inductive hypothesis implies that T' is an optimal code tree for the smaller set of frequencies. To obtain the final code tree T, we replace the leaf labeled $n+1$ with an internal node with two children, labelled 1 and 2. I claim that T is optimal for the original frequency array $f[1 \,..\, n]$.

To prove this claim, we can express the cost of T in terms of the cost of T' as follows. (In these equations, *depth*(i) denotes the depth of the leaf labelled i in either T or T'; if the leaf appears in both T and T', it has the same depth in both trees.)

$$\begin{aligned}
\mathit{cost}(T) &= \sum_{i=1}^{n} f[i] \cdot \mathit{depth}(i) \\
&= \sum_{i=3}^{n+1} f[i] \cdot \mathit{depth}(i) + f[1] \cdot \mathit{depth}(1) + f[2] \cdot \mathit{depth}(2) - f[n+1] \cdot \mathit{depth}(n+1) \\
&= \mathit{cost}(T') + f[1] \cdot \mathit{depth}(1) + f[2] \cdot \mathit{depth}(2) - f[n+1] \cdot \mathit{depth}(n+1) \\
&= \mathit{cost}(T') + (f[1] + f[2]) \cdot \mathit{depth}(T) - f[n+1] \cdot (\mathit{depth}(T) - 1) \\
&= \mathit{cost}(T') + f[1] + f[2]
\end{aligned}$$

This equation implies that minimizing the cost of T is equivalent to minimizing the cost of T'; in particular, attaching leaves labeled 1 and 2 to the leaf in T' labeled $n+1$ gives an optimal code tree for the original frequencies. □

Oh, I almost forgot! To actually implement Huffman codes efficiently, we keep the characters in a min-heap, keyed by frequency. We also construct the code tree by keeping three arrays of indices, listing the left and right children and the parent of each node. The root of the tree is the node with index $2n - 1$.

```
BuildHuffman($f[1\,..\,n]$):
  for $i \leftarrow 1$ to $n$
      $L[i] \leftarrow 0$;  $R[i] \leftarrow 0$
      Insert($i, f[i]$)
  for $i \leftarrow n$ to $2n-1$
      $x \leftarrow$ ExtractMin( )
      $y \leftarrow$ ExtractMin( )
      $f[i] \leftarrow f[x] + f[y]$
      $L[i] \leftarrow x$;  $R[i] \leftarrow y$
      $P[x] \leftarrow i$;  $P[y] \leftarrow i$
      Insert($i, f[i]$)
  $P[2n-1] \leftarrow 0$
```

The algorithm performs $O(n)$ min-heap operations. If we use a balanced binary tree as the heap, each operation requires $O(\log n)$ time, so the total running time of BuildHuffman is $O(n \log n)$.

Finally, here are simple algorithms to encode and decode messages:

```
HuffmanEncode($A[1\,..\,k]$):
  $m \leftarrow 1$
  for $i \leftarrow 1$ to $k$
      HuffmanEncodeOne($A[i]$)

HuffmanEncodeOne($x$):
  if $x < 2n-1$
      HuffmanEncodeOne($P[x]$)
      if $x = L[P[x]]$
          $B[m] \leftarrow 0$
      else
          $B[m] \leftarrow 1$
      $m \leftarrow m + 1$
```

```
HuffmanDecode($B[1\,..\,m]$):
  $k \leftarrow 1$
  $v \leftarrow 2n-1$
  for $i \leftarrow 1$ to $m$
      if $B[i] = 0$
          $v \leftarrow L[v]$
      else
          $v \leftarrow R[v]$
      if $L[v] = 0$
          $A[k] \leftarrow v$
          $k \leftarrow k + 1$
          $v \leftarrow 2n-1$
```

The first nuts and bolts appeared in the middle 1400's. The bolts were just screws with straight sides and a blunt end. The nuts were hand-made, and very crude. When a match was found between a nut and a bolt, they were kept together until they were finally assembled.

In the Industrial Revolution, it soon became obvious that threaded fasteners made it easier to assemble products, and they also meant more reliable products. But the next big step came in 1801, with Eli Whitney, the inventor of the cotton gin. The lathe had been recently improved. Batches of bolts could now be cut on different lathes, and they would all fit the same nut.

Whitney set up a demonstration for President Adams, and Vice-President Jefferson. He had piles of musket parts on a table. There were 10 similar parts in each pile. He went from pile to pile, picking up a part at random. Using these completely random parts, he quickly put together a working musket.

— Karl S. Kruszelnicki ('Dr. Karl'), *Karl Trek*, December 1997

Dr Neumann in his Theory of Games and Economic Behavior *introduces the cut-up method of random action into game and military strategy: Assume that the worst has happened and act accordingly. If your strategy is at some point determined... by random factor your opponent will gain no advantage from knowing your strategy since he cannot predict the move. The cut-up method could be used to advantage in processing scientific data. How many discoveries have been made by accident? We cannot produce accidents to order.*

— William S. Burroughs, "The Cut-Up Method of Brion Gysin"
in *The Third Mind* by William S. Burroughs and Brion Gysin (1978)

5 Randomized Algorithms

5.1 Nuts and Bolts

Suppose we are given n nuts and n bolts of different sizes. Each nut matches exactly one bolt and vice versa. The nuts and bolts are all almost exactly the same size, so we can't tell if one bolt is bigger than the other, or if one nut is bigger than the other. If we try to match a nut witch a bolt, however, the nut will be either too big, too small, or just right for the bolt.

Our task is to match each nut to its corresponding bolt. But before we do this, let's try to solve some simpler problems, just to get a feel for what we can and can't do.

Suppose we want to find the nut that matches a particular bolt. The obvious algorithm — test every nut until we find a match — requires exactly $n-1$ tests in the worst case. We might have to check every bolt except one; if we get down the the last bolt without finding a match, we know that the last nut is the one we're looking for.[1]

Intuitively, in the 'average' case, this algorithm will look at approximately $n/2$ nuts. But what exactly does 'average case' mean?

5.2 Deterministic vs. Randomized Algorithms

Normally, when we talk about the running time of an algorithm, we mean the *worst-case* running time. This is the maximum, over all problems of a certain size, of the running time of that algorithm on that input:

$$T_{\text{worst-case}}(n) = \max_{|X|=n} T(X).$$

On extremely rare occasions, we will also be interested in the *best-case* running time:

$$T_{\text{best-case}}(n) = \min_{|X|=n} T(X).$$

[1] "Whenever you lose something, it's always in the last place you look. So why not just look there first?"

The *average-case* running time is best defined by the *expected value*, over all inputs X of a certain size, of the algorithm's running time for X:[2]

$$T_{\text{average-case}}(n) = \operatorname*{E}_{|X|=n}[T(X)] = \sum_{|X|=n} T(x) \cdot \Pr[X].$$

The problem with this definition is that we rarely, if ever, know what the probability of getting any particular input X is. We could compute average-case running times by assuming a particular probability distribution—for example, every possible input is equally likely—but this assumption doesn't describe reality very well. Most real-life data is decidedly non-random (or at least random in some unpredictable way).

Instead of considering this rather questionable notion of average case running time, we will make a distinction between two kinds of algorithms: *deterministic* and *randomized*. A deterministic algorithm is one that always behaves the same way given the same input; the input completely *determines* the sequence of computations performed by the algorithm. Randomized algorithms, on the other hand, base their behavior not only on the input but also on several *random* choices. The same randomized algorithm, given the same input multiple times, may perform different computations in each invocation.

This means, among other things, that the running time of a randomized algorithm on a given input is no longer fixed, but is itself a random variable. When we analyze randomized algorithms, we are typically interested in the *worst-case expected* running time. That is, we look at the average running time for each input, and then choose the maximum over all inputs of a certain size:

$$T_{\text{worst-case expected}}(n) = \max_{|X|=n} \operatorname{E}[T(X)].$$

It's important to note here that we are making *no* assumptions about the probability distribution of possible inputs. All the randomness is inside the algorithm, where we can control it!

5.3 Back to Nuts and Bolts

Let's go back to the problem of finding the nut that matches a given bolt. Suppose we use the same algorithm as before, but at each step we choose a nut *uniformly at random* from the untested nuts. 'Uniformly' is a technical term meaning that each nut has exactly the same probability of being chosen.[3] So if there are k nuts left to test, each one will be chosen with probability $1/k$. Now what's the expected number of comparisons we have to perform? Intuitively, it should be about $n/2$, but let's formalize our intuition.

Let $T(n)$ denote the number of comparisons our algorithm uses to find a match for a single bolt out of n nuts.[4] We still have some simple base cases $T(1) = 0$ and $T(2) = 1$, but when $n > 2$, $T(n)$ is a random variable. $T(n)$ is always between 1 and $n-1$; it's actual value depends on our algorithm's random choices. We are interested in the *expected value* or *expectation* of $T(n)$, which is defined as follows:

$$\operatorname{E}[T(n)] = \sum_{k=1}^{n-1} k \cdot \Pr[T(n) = k]$$

[2] The notation $\operatorname{E}[\,]$ for expectation has nothing to do with the shift operator $\mathbf{E}$ used in the annihilator method for solving recurrences!

[3] This is what most people think 'random' means, but they're wrong.

[4] Note that for this algorithm, the input is completely specified by the number n. Since we're choosing the nuts to test at random, even the order in which the nuts and bolts are presented doesn't matter. That's why I'm using the simpler notation $T(n)$ instead of $T(X)$.

If the target nut is the kth nut tested, our algorithm performs $\min\{k, n-1\}$ comparisons. In particular, if the target nut is the last nut chosen, we don't actually test it. Because we choose the next nut to test uniformly at random, the target nut is equally likely—with probability exactly $1/n$—to be the first, second, third, or kth bolt tested, for any k. Thus:

$$\Pr[T(n) = k] = \begin{cases} 1/n & \text{if } k < n-1, \\ 2/n & \text{if } k = n-1. \end{cases}$$

Plugging this into the definition of expectation gives us our answer.

$$\begin{aligned} \mathrm{E}[T(n)] &= \sum_{k=1}^{n-2} \frac{k}{n} + \frac{2(n-1)}{n} \\ &= \sum_{k=1}^{n-1} \frac{k}{n} + \frac{n-1}{n} \\ &= \frac{n(n-1)}{2n} + 1 - \frac{1}{n} \\ &= \frac{n+1}{2} - \frac{1}{n} \end{aligned}$$

We can get exactly the same answer by thinking of this algorithm recursively. We always have to perform at least one test. With probability $1/n$, we successfully find the matching nut and halt. With the remaining probability $1 - 1/n$, we recursively solve the same problem but with one fewer nut. We get the following recurrence for the expected number of tests:

$$T(1) = 0, \qquad \mathrm{E}[T(n)] = 1 + \frac{n-1}{n}\,\mathrm{E}[T(n-1)]$$

To get the solution, we define a new function $t(n) = n\,\mathrm{E}[T(n)]$ and rewrite:

$$t(1) = 0, \qquad t(n) = n + t(n-1)$$

This recurrence translates into a simple summation, which we can easily solve.

$$\begin{aligned} t(n) = \sum_{k=2}^{n} k &= \frac{n(n+1)}{2} - 1 \\ \implies \mathrm{E}[T(n)] = \frac{t(n)}{n} &= \frac{n+1}{2} - \frac{1}{n} \end{aligned}$$

5.4 Finding All Matches

Not let's go back to the problem introduced at the beginning of the lecture: finding the matching nut for every bolt. The simplest algorithm simply compares every nut with every bolt, for a total of n^2 comparisons. The next thing we might try is repeatedly finding an arbitrary matched pair, using our very first nuts and bolts algorithm. This requires

$$\sum_{i=1}^{n} (i-1) = \frac{n^2 - n}{2}$$

comparisons in the worst case. So we save roughly a factor of two over the really stupid algorithm. Not very exciting.

Here's another possibility. Choose a *pivot* bolt, and test it against every nut. Then test the matching pivot nut against every other bolt. After these $2n-1$ tests, we have one matched pair, and the remaining nuts and bolts are partitioned into two subsets: those smaller than the pivot pair and those larger than the pivot pair. Finally, recursively match up the two subsets. The worst-case number of tests made by this algorithm is given by the recurrence

$$\begin{aligned} T(n) &= 2n-1+\max_{1\le k\le n}\{T(k-1)+T(n-k)\} \\ &= 2n-1+T(n-1) \end{aligned}$$

Along with the trivial base case $T(0)=0$, this recurrence solves to

$$T(n)=\sum_{i=1}^{n}(2n-1)=n^2.$$

In the worst case, this algorithm tests *every* nut-bolt pair! We could have been a little more clever—for example, if the pivot bolt is the smallest bolt, we only need $n-1$ tests to partition everything, not $2n-1$—but cleverness doesn't actually help that much. We still end up with about $n^2/2$ tests in the worst case.

However, since this recursive algorithm looks almost exactly like quicksort, and everybody 'knows' that the 'average-case' running time of quicksort is $\Theta(n\log n)$, it seems reasonable to guess that the average number of nut-bolt comparisons is also $\Theta(n\log n)$. As we shall see shortly, if the pivot bolt is always chosen *uniformly at random*, this intuition is exactly right.

5.5 Reductions to and from Sorting

The second algorithm for mathing up the nuts and bolts looks exactly like quicksort. The algorithm not only matches up the nuts and bolts, but also sorts them by size.

In fact, the problems of sorting and matching nuts and bolts are equivalent, in the following sense. If the bolts were sorted, we could match the nuts and bolts in $O(n\log n)$ time by performing a binary search with each nut. Thus, if we had an algorithm to sort the bolts in $O(n\log n)$ time, we would immediately have an algorithm to match the nuts and bolts, starting from scratch, in $O(n\log n)$ time. This process of *assuming* a solution to one problem and using it to solve another is called *reduction*—we can *reduce* the matching problem to the sorting problem in $O(n\log n)$ time.

There is a reduction in the other direction, too. If the nuts and bolts were matched, we could sort them in $O(n\log n)$ time using, for example, merge sort. Thus, if we have an $O(n\log n)$ time algorithm for either sorting or matching nuts and bolts, we automatically have an $O(n\log n)$ time algorithm for the other problem.

Unfortunately, since we aren't allowed to directly compare two bolts or two nuts, we can't use heapsort or mergesort to sort the nuts and bolts in $O(n\log n)$ worst case time. In fact, the problem of sorting nuts and bolts *deterministically* in $O(n\log n)$ time was only 'solved' in 1995[5], but both the algorithms and their analysis are incredibly technical and the constant hidden in the $O(\cdot)$ notation is quite large.

Reductions will come up again later in the course when we start talking about lower bounds and NP-completeness.

[5] János Komlós, Yuan Ma, and Endre Szemerédi, Sorting nuts and bolts in $O(n\log n)$ time, *SIAM J. Discrete Math* 11(3):347–372, 1998. See also Phillip G. Bradford, Matching nuts and bolts optimally, Technical Report MPI-I-95-1-025, Max-Planck-Institut für Informatik, September 1995. Bradford's algorithm is *slightly* simpler.

5.6 Recursive Analysis

Intuitively, we can argue that our quicksort-like algorithm will usually choose a bolt of approximately median size, and so the average numbers of tests should be $O(n \log n)$. We can now finally formalize this intuition. To simplify the notation slightly, I'll write $\overline{T}(n)$ in place of $\mathrm{E}[T(n)]$ everywhere.

Our randomized matching/sorting algorithm chooses its pivot bolt *uniformly at random* from the set of unmatched bolts. Since the pivot bolt is equally likely to be the smallest, second smallest, or kth smallest for any k, the expected number of tests performed by our algorithm is given by the following recurrence:

$$\begin{aligned} \overline{T}(n) &= 2n - 1 + \mathrm{E}_k\left[\overline{T}(k-1) + \overline{T}(n-k)\right] \\ &= \boxed{2n - 1 + \frac{1}{n}\sum_{k=1}^{n}\left(\overline{T}(k-1) + \overline{T}(n-k)\right)} \end{aligned}$$

The base case is $T(0) = 0$. (We can save a few tests by setting $T(1) = 0$ instead of 1, but the analysis will be easier if we're a little stupid.)

Yuck. At this point, we could simply *guess* the solution, based on the incessant rumors that quicksort runs in $O(n \log n)$ time in the average case, and prove our guess correct by induction. A similar inductive proof appears in [CLR, pp. 166–167], but it was removed from the new edition [CLRS]. That's okay; nobody ever really understood that proof anyway. (See Section 5.8 below for details.)

However, if we're only interested in asymptotic bounds, we can afford to be a little conservative. What we'd *really* like is for the pivot bolt to be the median bolt, so that half the bolts are bigger and half the bolts are smaller. This isn't very likely, but there is a good chance that the pivot bolt is close to the median bolt. Let's say that a pivot bolt is *good* if it's in the middle half of the final sorted set of bolts, that is, bigger than at least $n/4$ bolts and smaller than at least $n/4$ bolts. If the pivot bolt is good, then the *worst* split we can have is into one set of $3n/4$ pairs and one set of $n/4$ pairs. If the pivot bolt is bad, then our algorithm is still better than starting over from scratch. Finally, a randomly chosen pivot bolt is good with probability $1/2$.

These simple observations give us the following simple recursive *upper bound* for the expected running time of our algorithm:

$$\overline{T}(n) \le 2n - 1 + \frac{1}{2}\left(\overline{T}\left(\frac{3n}{4}\right) + \overline{T}\left(\frac{n}{4}\right)\right) + \frac{1}{2}\cdot\overline{T}(n)$$

A little algebra simplifies this even further:

$$\overline{T}(n) \le 4n - 2 + \overline{T}\left(\frac{3n}{4}\right) + \overline{T}\left(\frac{n}{4}\right)$$

We can solve this recurrence using the recursion tree method, giving us the unsurprising upper bound $\overline{T}(n) = O(n \log n)$. A similar argument gives us the matching lower bound $\overline{T}(n) = \Omega(n \log n)$.

Unfortunately, while this argument is convincing, it is *not* a formal proof, because it relies on the unproven assumption that $\overline{T}(n)$ is a *convex* function, which means that $\overline{T}(n+1) + \overline{T}(n-1) \ge 2\overline{T}(n)$ for all n. $\overline{T}(n)$ is actually convex, but we never proved it. Convexity follows form the closed-form solution of the recurrence, but using that fact would be circular logic. Sadly, formally proving convexity seems to be almost as hard as solving the recurrence. If we want a *proof* of the expected cost of our algorithm, we need another way to proceed.

5.7 Iterative Analysis

By making a simple change to our algorithm, which has no effect on the number of tests, we can analyze it much more directly and exactly, without solving a recurrence or relying on hand-wavy intuition.

The recursive subproblems solved by quicksort can be laid out in a binary tree, where each node corresponds to a subset of the nuts and bolts. In the usual recursive formulation, the algorithm partitions the nuts and bolts at the root, then the left child of the root, then the leftmost grandchild, and so forth, recursively sorting everything on the left before starting on the right subproblem.

But we don't have to solve the subproblems in this order. In fact, we can visit the nodes in the recursion tree in any order we like, as long as the root is visited first, and any other node is visited after its parent. Thus, we can recast quicksort in the following iterative form. Choose a pivot bolt, find its match, and partition the remaining nuts and bolts into two subsets. Then pick a second pivot bolt and partition whichever of the two subsets contains it. At this point, we have two matched pairs and three subsets of nuts and bolts. Continue choosing new pivot bolts and partitioning subsets, each time finding one match and increasing the number of subsets by one, until every bolt has been chosen as the pivot. At the end, every bolt has been matched, and the nuts and bolts are sorted.

Suppose we always choose the next pivot bolt *uniformly at random* from the bolts that haven't been pivots yet. Then no matter which subset contains this bolt, the pivot bolt is equally likely to be any bolt *in that subset*. That implies (by induction) that our randomized iterative algorithm performs *exactly* the same set of tests as our randomized recursive algorithm, but possibly in a different order.

Now let B_i denote the ith smallest bolt, and N_j denote the jth smallest nut. For each i and j, define an indicator variable X_{ij} that equals 1 if our algorithm compares B_i with N_j and zero otherwise. Then the total number of nut/bolt comparisons is exactly

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}.$$

We are interested in the expected value of this double summation:

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{E}[X_{ij}].$$

This equation uses a crucial property of random variables called *linearity of expectation*: for any random variables X and Y, the sum of their expectations is equal to the expectation of their sum: $E[X+Y] = E[X] + E[Y]$.

To analyze our algorithm, we only need to compute the expected value of each X_{ij}. By definition of expectation,

$$\mathrm{E}[X_{ij}] = 0 \cdot \Pr[X_{ij} = 0] + 1 \cdot \Pr[X_{ij} = 1] = \Pr[X_{ij} = 1],$$

so we just need to calculate $\Pr[X_{ij} = 1]$ for all i and j.

First let's assume that $i < j$. The only comparisons our algorithm performs are between some pivot bolt (or its partner) and a nut (or bolt) in the same subset. The only thing that can prevent us from comparing B_i and N_j is if some intermediate bolt B_k, with $i < k < j$, is chosen as a pivot before B_i or B_j. In other words:

Our algorithm compares B_i and N_j if and only if the first pivot chosen from the set $\{B_i, B_{i+1}, \ldots, B_j\}$ is either B_i or B_j.

Since the set $\{B_i, B_{i+1}, \ldots, B_j\}$ contains $j-i+1$ bolts, each of which is equally likely to be chosen first, we immediately have

$$\mathrm{E}[X_{ij}] = \frac{2}{j-i+1} \qquad \text{for all } i<j.$$

Symmetric arguments give us $\mathrm{E}[X_{ij}] = \frac{2}{i-j+1}$ for all $i>j$. Since our algorithm is a little stupid, every bolt is compared with its partner, so $X_{ii}=1$ for all i. (In fact, if a pivot bolt is the only bolt in its subset, we don't need to compare it against its partner, but this improvement complicates the analysis.)

Putting everything together, we get the following summation.

$$\begin{aligned}
\mathrm{E}[T(n)] &= \sum_{i=1}^{n}\sum_{j=1}^{n}\mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n}\mathrm{E}[X_{ii}] + 2\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}] \\
&= \boxed{n + 4\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j-i+1}}
\end{aligned}$$

This is quite a bit simpler than the recurrence we got before. In fact, with just a few lines of algebra, we can turn it into an exact, closed-form expression for the expected number of comparisons.

$$\begin{aligned}
\mathrm{E}[T(n)] &= n + 4\sum_{i=1}^{n}\sum_{j=2}^{n-i+1}\frac{1}{k} && [\text{substitute } k=j-i+1] \\
&= n + 4\sum_{k=2}^{n}\sum_{i=1}^{n-k+1}\frac{1}{k} && [\text{reorder summations}] \\
&= n + 4\sum_{k=2}^{n}\frac{n-k+1}{k} \\
&= n + 4\left((n-1)\sum_{k=2}^{n}\frac{1}{k} - \sum_{k=2}^{n}1\right) \\
&= n + 4((n+1)(H_n-1) - (n-1)) \\
&= \boxed{4nH_n - 7n + 4H_n}
\end{aligned}$$

Sure enough, it's $\Theta(n\log n)$.

*5.8 Masochistic Analysis

If we're feeling particularly masochistic, we can actually solve the recurrence directly, all the way to an exact closed-form solution. [I'm including this only to show you it can be done; this won't be on the test.] First we simplify the recurrence slightly by combining symmetric terms.

$$\begin{aligned}
\overline{T}(n) &= 2n - 1 + \frac{1}{n}\sum_{k=1}^{n}\left(\overline{T}(k-1) + \overline{T}(n-k)\right) \\
&= 2n - 1 + \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}(k)
\end{aligned}$$

We then convert this 'full history' recurrence into a 'limited history' recurrence by shifting, and subtracting away common terms. (I call this "Magic step #1".) To make this slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

$$\begin{aligned}
n\overline{T}(n) &= 2n^2 - n + 2\sum_{k=0}^{n-1}\overline{T}(k)\\
(n-1)\overline{T}(n-1) &= \underbrace{2(n-1)^2-(n-1)}_{2n^2-5n+3} + 2\sum_{k=0}^{n-2}\overline{T}(k)\\
n\overline{T}(n)-(n-1)\overline{T}(n-1) &= 4n-3+2T(n-1)\\
\overline{T}(n) &= 4-\frac{3}{n}+\frac{n+1}{n}\overline{T}(n-1)
\end{aligned}$$

To solve this limited-history recurrence, we define a new function $t(n) = \overline{T}(n)/(n+1)$. (I call this "Magic step #2".) This gives us an even simpler recurrence for $t(n)$ in terms of $t(n-1)$:

$$\begin{aligned}
t(n) &= \frac{\overline{T}(n)}{n+1}\\
&= \frac{1}{n+1}\left(4-\frac{3}{n}+(n+1)\frac{T(n-1)}{n}\right)\\
&= \frac{4}{n+1}-\frac{3}{n(n+1)}+t(n-1)\\
&= \frac{7}{n+1}-\frac{3}{n}+t(n-1)
\end{aligned}$$

I used the technique of partial fractions (remember calculus?) to replace $\frac{1}{n(n+1)}$ with $\frac{1}{n}-\frac{1}{n+1}$ in the last step. The base case for this recurrence is $t(0)=0$. Once again, we have a recurrence that translates directly into a summation, which we can solve with just a few lines of algebra.

$$\begin{aligned}
t(n) &= \sum_{i=1}^{n}\left(\frac{7}{i+1}-\frac{3}{i}\right)\\
&= 7\sum_{i=1}^{n}\frac{1}{i+1}-3\sum_{i=1}^{n}\frac{1}{i}\\
&= 7(H_{n+1}-1)-3H_n\\
&= 4H_n-7+\frac{7}{n+1}
\end{aligned}$$

The last step uses the recursive definition of the harmonic numbers: $H_{n+1} = H_n + \frac{1}{n+1}$. Finally, substituting $\overline{T}(n) = (n+1)t(n)$ and simplifying gives us the exact solution to the original recurrence.

$$\overline{T}(n) = 4(n+1)H_n - 7(n+1) + 7 = \boxed{4nH_n - 7n + 4H_n}$$

Surprise, surprise, we get exactly the same solution!

> *I thought the following four [rules] would be enough, provided that I made a firm and constant resolution not to fail even once in the observance of them. The first was never to accept anything as true if I had not evident knowledge of its being so. . . . The second, to divide each problem I examined into as many parts as was feasible, and as was requisite for its better solution. The third, to direct my thoughts in an orderly way. . . establishing an order in thought even when the objects had no natural priority one to another. And the last, to make throughout such complete enumerations and such general surveys that I might be sure of leaving nothing out.*
>
> — René Descartes, *Discours de la Méthode* (1637)

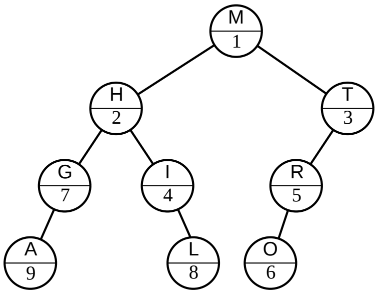
> *For example, creating shortcuts by sprinkling a few diversely connected individuals throughout a large organization could dramatically speed up information flow between departments. On the other hand, because only a few random shortcuts are necessary to make the world small, subtle changes to networks have alarming consequences for the rapid spread of computer viruses, pernicious rumors, and infectious diseases.*
>
> — Ivars Peterson, *Science News*, August 22, 1998.

6 Treaps and Skip Lists

In this lecture, we consider two randomized alternatives to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.

6.1 Treaps

A *treap* is a binary tree in which every node has both a *search key* and a *priority*, where the inorder sequence of search keys is sorted and each node's priority is smaller than the priorities of its children.[1] In other words, a treap is simultaneously a binary search tree for the search keys and a (min-)heap for the priorities. In our examples, we will use letters for the search keys and numbers for the priorities.



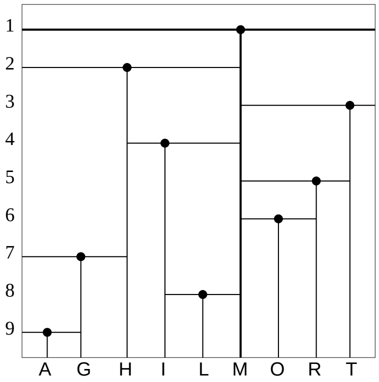
A treap. The top half of each node shows its search key and the bottom half shows its priority.

I'll assume from now on that all the keys and priorities are distinct. Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the search keys and priorities of its nodes. Since it's a heap, the node v with highest priority must be the root. Since it's also a binary search tree, any node u with $key(u) < key(v)$ must be in the left subtree, and any node w with $key(w) > key(v)$ must be in the right subtree. Finally, since the subtrees are treaps, by induction, their structures are completely determined. The base case is the trivial empty treap.

[1] Sometimes I hate English. Normally, 'higher priority' means 'more important', but 'first priority' is also more important than 'second priority'. Maybe 'posteriority' would be better; one student suggested 'unimportance'.

Another way to describe the structure is that a treap is exactly the binary tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the usual insertion algorithm. This is also easy to prove by induction.

A third way interprets the keys and priorities as the coordinates of a set of points in the plane. The root corresponds to a T whose joint lies on the topmost point. The T splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively. This interpretation has some interesting applications in computational geometry, which (unfortunately) we probably won't have time to talk about.



A geometric interpretation of the same treap.

Treaps were first discovered by Jean Vuillemin in 1980, but he called them *Cartesian trees*.[2] The word 'treap' was first used by Edward McCreight around 1980 to describe a slightly different data structure, but he later switched to the more prosaic name *priority search trees*.[3] Treaps were rediscovered and used to build randomized search trees by Cecilia Aragon and Raimund Seidel in 1989.[4] A different kind of randomized binary search tree, which uses random rebalancing instead of random priorities, was later discovered and analyzed by Conrado Martínez and Salvador Roura in 1996.[5]

6.1.1 Binary Search Tree Operations

The search algorithm is the usual one for binary search trees. The time for a successful search is proportional to the depth of the node. The time for an unsuccessful search is proportional to the depth of either its successor or its predecessor.

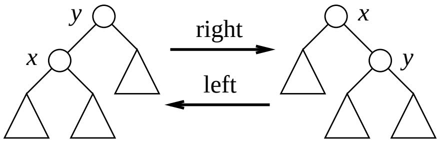
To insert a new node z, we start by using the standard binary search tree insertion algorithm to insert it at the bottom of the tree. At the point, the search keys still form a search tree, but the priorities may no longer form a heap. To fix the heap property, as long as z has smaller priority than its parent, perform a *rotation* at z, a local operation that decreases the depth of z by one and increases its parent's depth by one, while maintaining the search tree property. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

[2] J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

[3] E. M. McCreight. Priority search trees. *SIAM J. Comput.* 14(2):257–276, 1985.

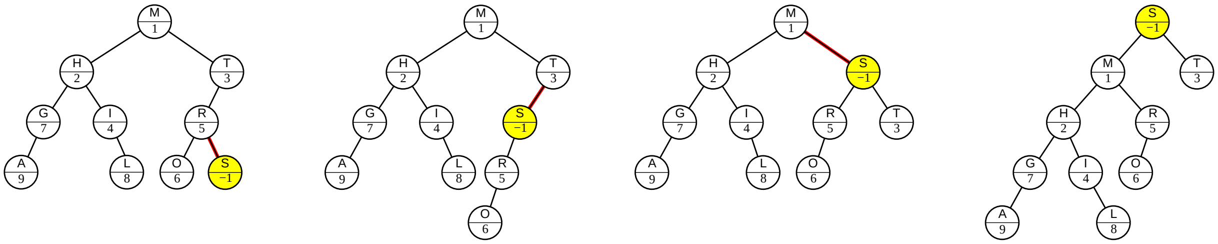
[4] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica* 16:464–497, 1996.

[5] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.



A right rotation at x and a left rotation at y are inverses.

The overall time to insert z is proportional to the depth of z before the rotations—we have to walk down the treap to insert z, and then walk back up the treap doing rotations. Another way to say this is that the time to insert z is roughly twice the time to perform an unsuccessful search for $key(z)$.



Left to right: After inserting $(S, 10)$, rotate it up to fix the heap property.
Right to left: Before deleting $(S, 10)$, rotate it down to make it a leaf.

Deleting a node is *exactly* like inserting a node, but in reverse order. Suppose we want to delete node z. As long as z is not a leaf, perform a rotation at the child of z with smaller priority. This moves z down a level and its smaller-priority child up a level. The choice of which child to rotate preserves the heap property everywhere except at z. When z becomes a leaf, chop it off.

We sometimes also want to *split* a treap T into two treaps $T_<$ and $T_>$ along some pivot key π, so that all the nodes in $T_<$ have keys less than π and all the nodes in $T_>$ have keys bigger then π. A simple way to do this is to insert a new node z with $key(z) = \pi$ and $priority(z) = -\infty$. After the insertion, the new node is the root of the treap. If we delete the root, the left and right sub-treaps are exactly the trees we want. The time to split at π is roughly twice the time to (unsuccessfully) search for π.

Similarly, we may want to *merge* two treaps $T_<$ and $T_>$, where every node in $T_<$ has a smaller search key than any node in $T_>$, into one super-treap. Merging is just splitting in reverse—create a dummy root whose left sub-treap is $T_<$ and whose right sub-treap is $T_>$, rotate the dummy node down to a leaf, and then cut it off.

6.1.2 Analysis

The cost of each of these operations is proportional to the depth of some node v in the treap.

- **Search:** A successful search for key k takes $O(\text{depth}(v))$ time, where v is the node with $key(v) = k$. For an unsuccessful search, let v^- be the inorder *predecessor* of k (the node whose key is just barely smaller than k), and let v^+ be the inorder *successor* of k (the node whose key is just barely larger than k). Since the last node examined by the binary search is either v^- or v^+, the time for an unsuccessful search is either $O(\text{depth}(v^+))$ or $O(\text{depth}(v^-))$.

- **Insert/Delete:** Inserting a new node with key k takes either $O(\text{depth}(v^+))$ time or $O(\text{depth}(v^-))$ time, where v^+ and v^- are the predecessor and successor of the new node. Deletion is just insertion in reverse.

- **Split/Merge:** Splitting a treap at pivot value k takes either $O(\text{depth}(v^+))$ time or $O(\text{depth}(v^-))$ time, since it costs the same as inserting a new dummy root with search key k and priority $-\infty$. Merging is just splitting in reverse.

Since the depth of a node in a treap is $\Theta(n)$ in the worst case, each of these operations has a worst-case running time of $\Theta(n)$.

6.1.3 Random Priorities

A *randomized binary search tree* is a treap in which the priorities are *independently and uniformly distributed continuous random variables*. That means that whenever we insert a new search key into the treap, we generate a random real number between (say) 0 and 1 and use that number as the priority of the new node. The only reason we're using real numbers is so that the probability of two nodes having the same priority is zero, since equal priorities make the analysis messy. In practice, we could just choose random integers from a large range, like 0 to $2^{31}-1$, or random floating point numbers. Also, since the priorities are independent, each node is equally likely to have the smallest priority.

The cost of all the operations we discussed—search, insert, delete, split, join—is proportional to the depth of some node in the tree. Here we'll see that the *expected* depth of *any* node is $O(\log n)$, which implies that the expected running time for any of those operations is also $O(\log n)$.

Let x_k denote the node with the kth smallest search key. To analyze the expected depth, we define an indicator variable

$$A_k^i = \big[x_i \text{ is a proper ancestor of } x_k\big].$$

(The superscript doesn't mean power in this case; it just a reminder of which node is supposed to be further up in the tree.) Since the depth of v is just the number of proper ancestors of v, we have the following identity:

$$\text{depth}(x_k) = \sum_{i=1}^{n} A_k^i.$$

Now we can express the *expected* depth of a node in terms of these indicator variables as follows.

$$\mathrm{E}[\text{depth}(x_k)] = \sum_{i=1}^{n} \Pr[A_k^i = 1]$$

(Just as in our analysis of matching nuts and bolts in Lecture 3, we're using linearity of expectation and the fact that $\mathrm{E}[X] = \Pr[X = 1]$ for any indicator variable X.) So to compute the expected depth of a node, we just have to compute the probability that some node is a proper ancestor of some other node.

Fortunately, we can do this easily once we prove a simple structural lemma. Let $X(i,k)$ denote either the subset of treap nodes $\{x_i, x_{i+1}, \ldots, x_k\}$ or the subset $\{x_k, x_{k+1}, \ldots, x_i\}$, depending on whether $i < k$ or $i > k$. $X(i,k)$ and $X(k,i)$ always denote prceisly the same subset, and this subset contains $|k-i|+1$ nodes. $X(1,n) = X(n,1)$ contains all n nodes in the treap.

Lemma 1. *For all $i \neq k$, x_i is a proper ancestor of x_k if and only if x_i has the smallest priority among all nodes in $X(i,k)$.*

Proof: If x_i is the root, then it is an ancestor of x_k, and by definition, it has the smallest priority of *any* node in the treap, so it must have the smallest priority in $X(i,k)$.

On the other hand, if x_k is the root, then x_i is not an ancestor of x_k, and indeed x_i does not have the smallest priority in $X(i,k)$ — x_k does.

On the gripping hand[6], suppose some other node x_j is the root. If x_i and x_k are in different subtrees, then either $i < j < k$ or $i > j > k$, so $x_j \in X(i,k)$. In this case, x_i is not an ancestor of x_k, and indeed x_i does not have the smallest priority in $X(i,k)$ — x_j does.

Finally, if x_i and x_k are in the same subtree, the lemma follows inductively (or, if you prefer, recursively), since the subtree is a smaller treap. The empty treap is the trivial base case. □

Since each node in $X(i,k)$ is equally likely to have smallest priority, we immediately have the probability we wanted:

$$\Pr[A_k^i = 1] = \frac{[i \neq k]}{|k-i|+1} = \begin{cases} \dfrac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \dfrac{1}{i-k+1} & \text{if } i > k \end{cases}$$

To compute the expected depth of a node x_k, we just plug this probability into our formula and grind through the algebra.

$$\begin{aligned} \text{E}[\text{depth}(x_k)] = \sum_{i=1}^{n} \Pr[A_k^i = 1] &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\ &= \sum_{j=2}^{k} \frac{1}{j} + \sum_{i=2}^{n-k+1} \frac{1}{j} \\ &= H_k - 1 + H_{n-k+1} - 1 \\ &< \ln k + \ln(n-k+1) - 2 \\ &< 2\ln n - 2. \end{aligned}$$

In conclusion, every search, insertion, deletion, split, and merge operation in an n-node randomized binary search tree takes $O(\log n)$ expected time.

Since a treap is exactly the binary tree that results when you insert the keys in order of increasing priority, a randomized treap is the result of inserting the keys in *random* order. So our analysis also automatically gives us the expected depth of any node in a binary tree built by random insertions (without using priorities).

6.1.4 Randomized Quicksort (Again?!)

We've already seen two completely different ways of describing randomized quicksort. The first is the familiar recursive one: choose a random pivot, partition, and recurse. The second is a less familiar iterative version: repeatedly choose a new random pivot, partition whatever subset contains it, and continue. But there's a third way to describe randomized quicksort, this time in terms of binary search trees.

[6] See Larry Niven and Jerry Pournelle, *The Gripping Hand*, Pocket Books, 1994.

```
RandomizedQuicksort:
  $T \leftarrow$ an empty binary search tree
  insert the keys into $T$ in random order
  output the inorder sequence of keys in $T$
```

Our treap analysis tells us is that this algorithm will run in $O(n \log n)$ expected time, since each key is inserted in $O(\log n)$ expected time.

Why is this quicksort? Just like last time, all we've done is rearrange the order of the comparisons. Intuitively, the binary tree is just the recursion tree created by the normal version of quicksort. In the recursive formulation, we compare the initial pivot against everything else and then recurse. In the binary tree formulation, the first "pivot" becomes the root of the tree without any comparisons, but then later as each other key is inserted into the tree, it is compared against the root. Either way, the first pivot chosen is compared with everything else. The partition splits the remaining items into a left subarray and a right subarray; in the binary tree version, these are exactly the items that go into the left subtree and the right subtree. Since both algorithms define the same two subproblems, by induction, both algorithms perform the same comparisons.

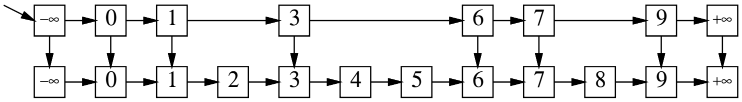
We even saw the probability $\frac{1}{|k-i|+1}$ before, when we were talking about sorting nuts and bolts with a variant of randomized quicksort. In the more familiar setting of sorting an array of numbers, the probability that randomized quicksort compares the ith largest and kth largest elements is exactly $\frac{2}{|k-i|+1}$. The binary tree version compares x_i and x_k if and only if x_i is an ancestor of x_k or vice versa, so the probabilities are exactly the same.

6.2 Skip Lists

Skip lists, which were first discovered by Bill Pugh in the late 1980's,[7] have many of the usual desirable properties of balanced binary search trees, but their structure is completely different.

6.2.1 Random Shortcuts

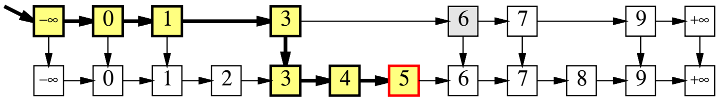
At a high level, a skip list is just a sorted, singly linked list with some shortcuts. To do a search in a normal singly-linked list of length n, we obviously need to look at n items in the worst case. To speed up this process, we can make a second-level list that contains roughly half the items from the original list. Specifically, for each item in the original list, we duplicate it with probability $1/2$. We then string together all the duplicates into a second sorted linked list, and add a pointer from each duplicate back to its original. Just to be safe, we also add sentinel nodes at the beginning and end of both lists.



A linked list with some randomly-chosen shortcuts.

Now we can find a value x in this augmented structure using a two-stage algorithm. First, we scan for x in the shortcut list, starting at the $-\infty$ sentinel node. If we find x, we're done. Otherwise, we reach some value bigger than x and we know that x is not in the shortcut list. Let w be the largest item less than x in the shortcut list. In the second phase, we scan for x in the original list, starting from w. Again, if we reach a value bigger than x, we know that x is not in the data structure.

[7] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33(6):668–676, 1990.

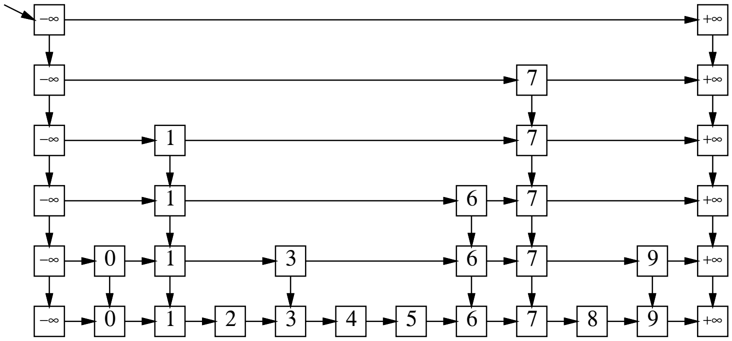


Searching for 5 in a list with shortcuts.

Since each node appears in the shortcut list with probability $1/2$, the expected number of nodes examined in the first phase is at most $n/2$. Only one of the nodes examined in the second phase has a duplicate. The probability that any node is followed by k nodes without duplicates is 2^{-k}, so the expected number of nodes examined in the second phase is at most $1 + \sum_{k\geq 0} 2^{-k} = 2$. Thus, by adding these random shortcuts, we've reduced the cost of a search from n to $n/2 + 2$, roughly a factor of two in savings.

6.2.2 Recursive Random Shortcuts

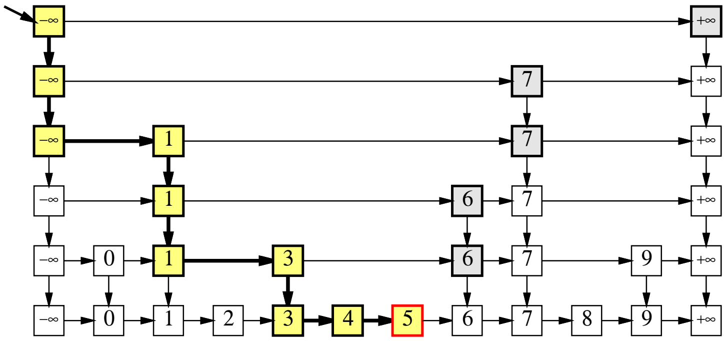
Now there's an obvious improvement—add shortcuts to the shortcuts, and repeat recursively. That's exactly how skip lists are constructed. For each node in the original list, we flip a coin over and over until we get tails. For each heads, we make a duplicate of the node. The duplicates are stacked up in levels, and the nodes on each level are strung together into sorted linked lists. Each node v stores a search key (key(v)), a pointer to its next lower copy (down(v)), and a pointer to the next node in its level (right(v)).



A skip list is a linked list with recursive random shortcuts.

The search algorithm for skip lists is very simple. Starting at the leftmost node L in the highest level, we scan through each level as far as we can without passing the target value x, and then proceed down to the next level. The search ends when we either reach a node with search key x or fail to find x on the lowest level.

```
SkipListFind($x, L$):
  $v \leftarrow L$
  while ($v \neq$ Null and key($v$) $\neq x$)
      if key(right($v$)) $> x$
          $v \leftarrow$ down($v$)
      else
          $v \leftarrow$ right($v$)
  return $v$
```



Searching for 5 in a skip list.

Intuitively, Since each level of the skip lists has about half the number of nodes as the previous level, the total number of levels should be about $O(\log n)$. Similarly, each time we add another level of random shortcuts to the skip list, we cut the search time in half except for a constant overhead. So after $O(\log n)$ levels, we should have a search time of $O(\log n)$. Let's formalize each of these two intuitive observations.

6.2.3 Number of Levels

The actual values of the search keys don't affect the skip list analysis, so let's assume the keys are the integers 1 through n. Let $L(x)$ be the number of levels of the skip list that contain some search key x, not counting the bottom level. Each new copy of x is created with probability $1/2$ from the previous level, essentially by flipping a coin. We can compute the expected value of $L(x)$ recursively—with probability $1/2$, we flip tails and $L(x) = 0$; and with probability $1/2$, we flip heads, increase $L(x)$ by one, and recurse:

$$E[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2}\big(1 + E[L(x)]\big)$$

Solving this equation gives us $E[L(x)] = 1$.

In order to analyze the expected worst-case cost of a search, however, we need a bound on the *number of levels* $L = \max_x L(x)$. Unfortunately, we can't compute the average of a maximum the way we would compute the average of a sum. Instead, we will derive a stronger result, showing that the depth is $O(\log n)$ *with high probability*. 'High probability' is a technical term that means the probability is at least $1 - 1/n^c$ for some constant $c \geq 1$; the hidden constant in the $O(\log n)$ bound could depend on c.

In order for a search key x to appear on the kth level, we must have flipped k heads in a row, so $\Pr[L(x) \geq k] = 2^{-k}$. In particular,

$$\Pr[L(x) \geq 2\lg n] = \frac{1}{n^2}.$$

(There's nothing special about the number 2 here.) The skip list has at least $2\lg n$ levels if and only if $L(x) \geq 2\lg n$ for at least one of the n search keys.

$$\Pr[L \geq 2\lg n] = \Pr\big[(L(1) \geq 2\lg n) \;\vee\; (L(2) \geq 2\lg n) \;\vee \cdots \vee\; (L(n) \geq 2\lg n)\big]$$

Since $\Pr[A \vee B] \leq Pr[A] + \Pr[B]$ for any random events A and B, we can simplify this as follows:

$$\Pr[L \geq 2\lg n] \leq \sum_{x=1}^{n} \Pr[L(x) \geq 2\lg n] = \sum_{x=1}^{n} \frac{1}{n^2} = \frac{1}{n}.$$

So with high probability, a skip list has $O(\log n)$ levels.

6.2.4 Logarithmic Search Time

It's a little easier to analyze the cost of a search if we imagine running the algorithm backwards. ᗡИIᖷTƧIᒧᑫIꓘƧ takes the output from SKIPLISTFIND as input and traces back through the data structure to the upper left corner. Skip lists don't really have up and left pointers, but we'll pretend that they do so we don't have to write '(ʋ)nwob → ʋ' or '(ʋ)Ɉʜϱiɿ → ʋ'.[8]

```
ᗡИIᖷTƧIᒧᑫIꓘƧ($v$):
  while ($v \neq L$)
      if up($v$) exists
          $v \leftarrow$ up($v$)
      else
          $v \leftarrow$ left($v$)
```

Now for *every* node v in the skip list, up(v) exists with probability $1/2$. So for purposes of analysis, ᗡИIᖷTƧIᒧᑫIꓘƧ is equivalent to the following algorithm:

```
FLIPWALK($v$):
  while ($v \neq L$)
      if COINFLIP = HEADS
      $v \leftarrow$ up($v$)
  else
      $v \leftarrow$ left($v$)
```

Obviously, the expected number of heads is exactly the same as the expected number of TAILS. Thus, the expected running time of this algorithm is twice the expected number of upward jumps. Since we already know that the number of upward jumps is $O(\log n)$ with high probability, we can conclude that the expected worst-case search time is $O(\log n)$.

*6.3 High-Probability Bounds for Treaps

The simple recursive structure of skip lists make it relatively easy to derive an upper bound on the expected *worst-case* search time, by way of a stronger high-probability upper bound on the worst-case search time. We can prove similar results for treaps, but because of the more complex recursive structure, we need slightly more sophisticated probabilistic tools. These tools are usually cell *tail inequalities*; intuitively, they bound the probability that a random variable with a bell-shaped distribution takes a value in the *tails* of the distribution, far away from the mean.

The simplest such bound is called Markov's Inequality.

Markov's Inequality. *If X is a non-negative integer random variable, then* $\Pr[X \geq t] \leq \mathrm{E}[X]/t$ *for any* $t > 0$.

[8] Leonardo da Vinci used to write everything this way, but not because he wanted to keep his discoveries secret. He just had really bad arthritis in his right hand!

Proof: This follows from the definition of expectation by simple algebraic manipulation.

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k=0}^{\infty} k \cdot \Pr[X = k] && \text{[definition of } \mathrm{E}[X]\text{]} \\
&= \sum_{k=0}^{\infty} \Pr[X \geq k] && \text{[algebra]} \\
&\geq \sum_{k=0}^{t-1} \Pr[X \geq k] && \text{[since } k < \infty\text{]} \\
&\geq \sum_{k=0}^{t-1} \Pr[X \geq t] && \text{[since } k < t\text{]} \\
&= t \cdot \Pr[X \geq t] && \text{[algebra]}
\end{aligned}$$

□

6.3.1 Chernoff Bounds

Unfortunately, the bounds that Markov's inequality implies (at least directly) are often very weak, even useless. (For example, Markov's inequality implies that with high probability, every node in an n-node treap has depth $O(n^2 \log n)$. Well, *duh!*) To get stronger bounds, we need to exploit some additional structure in our random variables.

Recall that random variables $X_1, X_2, \ldots, X_n$ are *mutually independent* if and only if

$$\Pr\left[\bigwedge_{i=1}^{n} (X_i = x_i)\right] = \prod_{i=1}^{n} \Pr[X_i = x_i]$$

for all possible values $x_1, x_2, \ldots, x_n$. For examples, different flips of the same fair coin are mutually independent, but the number of heads and the number of tails in a sequence of n coin flips are not independent (since they must add to n). Mutual independence of the X_i's implies that

$$\mathrm{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n} \mathrm{E}[X_i].$$

Suppose $X = \sum_{i=1}^{n} X_i$ is the sum of n mutually independent random *indicator* variables X_i. For each i, let $p_i = \Pr[X_i = 1]$, and let $\mu = \mathrm{E}[X] = \sum_i \mathrm{E}[X_i] = \sum_i p_i$.

Chernoff Bound (Upper Tail). $\boxed{\Pr[X > (1+\delta)\mu] < \left(\dfrac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}}$ *for any* $\delta > 0$.

Proof: The proof is fairly long, but it replies on just a few basic components: a clever substitution, Markov's inequality, the independence of the X_i's, The World's Most Useful Inequality $e^x > 1 + x$, a tiny bit of calculus, and lots of high-school algebra.

We start by introducing a variable t, whose role will become clear shortly.

$$Pr[X > (1+\delta)\mu] = \Pr[e^{tX} > e^{t(1+\delta)\mu}]$$

To cut down on the superscripts, I'll usually write $\exp(x)$ instead of e^x in the rest of the proof. Now apply Markov's inequality to the right side of this equation:

$$Pr[X > (1+\delta)\mu] < \frac{\mathrm{E}[\exp(tX)]}{\exp(t(1+\delta)\mu)}.$$

We can simplify the expectation on the right using the fact that the terms X_i are independent.

$$\mathrm{E}\left[\exp(tX)\right] = \mathrm{E}\left[\exp\left(t\sum_i X_i\right)\right] = \mathrm{E}\left[\prod_i \exp(tX_i)\right] = \prod_i \mathrm{E}\left[\exp(tX_i)\right]$$

We can bound the individual expectations $\mathrm{E}\left[e^{tX_i}\right]$ using The World's Most Useful Inequality:

$$\mathrm{E}[\exp(tX_i)] = p_i e^t + (1 - p_i) = 1 + (e^t - 1)p_i < \exp\big((e^t - 1)p_i\big)$$

This inequality gives us a simple upper bound for $\mathrm{E}[e^{tX}]$:

$$\mathrm{E}\left[\exp(tX)\right] < \prod_i \exp((e^t - 1)p_i) < \exp\left(\sum_i (e^t - 1)p_i\right) = \exp((e^t - 1)\mu)$$

Substituting this back into our original fraction from Markov's inequality, we obtain

$$Pr[X > (1+\delta)\mu] < \frac{\mathrm{E}[\exp(tX)]}{\exp(t(1+\delta)\mu)} < \frac{\exp((e^t - 1)\mu)}{\exp(t(1+\delta)\mu)} = \big(\exp(e^t - 1 - t(1+\delta))\big)^{\mu}$$

Notice that this last inequality holds for *all* possible values of t. To obtain the final tail bound, we will choose t to make this bound as tight as possible. To minimize $e^t - 1 - t - t\delta$, we take its derivative with respect to t and set it to zero:

$$\frac{d}{dt}(e^t - 1 - t(1+\delta)) = e^t - 1 - \delta = 0.$$

(And you thought calculus would never be useful!) This equation has just one solution $t = \ln(1+\delta)$. Plugging this back into our bound gives us

$$Pr[X > (1+\delta)\mu] < \big(\exp(\delta - (1+\delta)\ln(1+\delta))\big)^{\mu} = \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

And we're done! □

This form of the Chernoff bound can be a bit clumsy to use. A more complicated argument gives us the bound

$$\boxed{\Pr[X > (1+\delta)\mu] < e^{-\mu\delta^2/3}}$$

for any $0 < \delta < 1$.

A similar argument gives us an inequality bounding the probability that X is significantly *less* than its expected value:

Chernoff Bound (Lower Tail). $\boxed{\Pr[X < (1-\delta)\mu] < \left(\dfrac{e^{\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} < e^{-\mu\delta^2/2}}$ *for any* $\delta > 0$.

6.3.2 Back to Treaps

In our analysis of randomized treaps, we defined the indicator variable A_k^i to have the value 1 if and only if the node with the ith smallest key ('node i') was a proper ancestor of the node with the kth smallest key ('node k'). We argued that

$$\Pr[A_k^i = 1] = \frac{[i \neq k]}{|k - i| + 1},$$

and from this we concluded that the expected depth of node k is

$$\mathrm{E}[\text{depth}(k)] = \sum_{i=1}^{n} \Pr[A_k^i = 1] = H_k + H_{n-k} - 2 < 2\ln n.$$

To prove a worst-case expected bound on the depth of the tree, we need to argue that the *maximum* depth of any node is small. Chernoff bounds make this argument easy, once we establish the following fact, whose proof we leave as an easy exercise (hint, hint).

Observation. *For any index k, the $k-1$ random variables X_k^i such that $i < k$ are mutually independent. Similarly, for any index k, the $n-k$ random variables X_k^i such that $i > k$ are mutually independent.*

Lemma: *The depth of a randomized treap with n nodes is $O(\log n)$ with high probability.*

Proof: First let's bound the probability that the depth of node k is at most $10\ln n$. There's nothing special about the constant 10 here; I'm being somewhat generous to make the analysis easier.

The depth is a sum of n indicator variables A_k^i, as i ranges from 1 to n. Our Observation allows us to partition these variables into two mutually independent subsets. Let $d_<(k) = \sum_{i<k} A_k^i$ and $d_>(k) = \sum_{i<k} A_k^i$, so that depth$(k) = d_<(k) + d_>(k)$. Our earlier analysis implies that

$$\mathrm{E}[d_<(k)] = H_k - 1 \quad \text{and} \quad \mathrm{E}[d_>(k)] = H_{n-k+1} - 1$$

If depth$(k) > 10\ln n$, then either $d_<(k) > 5\ln n$ or $d_>(k) > 5\ln n$. We can bound the probability that $d_<(k) > 5\ln n$ by applying Chernoff's inequality with $\mu = \mathrm{E}[d_<(k)] = H_k - 1 < \ln n$ and $\delta = 4$.

$$\begin{aligned}
\Pr[d_<(k) > 5\ln n] &< \Pr[d_<(k) > 5\mu] \\
&< \left(\frac{e^2}{5^5}\right)^{\mu} \\
&< \left(\frac{e^2}{5^5}\right)^{\ln n} = n^{\ln(e^2/5^5)} = n^{2-5\ln 5} < \frac{1}{n^6}.
\end{aligned}$$

(The last step uses the fact that $5\ln 5 \approx 8.04781 > 8$.) The same analysis implies that $\Pr[d_<(k) > 5\ln n] < 1/n^6$. These inequalities imply the very crude upper bound $\Pr[\text{depth}(k) > 10\ln n] < 2/n^6$.

Now consider the probability that the treap has depth greater than $10\ln n$. Even though the distributions of different nodes' depths are *not* independent, we can conservatively bound the probability of failure as follows:

$$\Pr\left[\max_k \text{depth}(k) > 10\ln n\right] < \sum_{k=1}^{n} \Pr[\text{depth}(k) > 10\ln n] < \frac{2}{n^5}.$$

More generally, this same argument implies that for any constant Δ, the depth of the treap is less than $2\Delta\ln n$ with probability at most $2/n^{1-\Delta\ln\Delta}$. □

This lemma implies that any search, insertion, deletion, or merge operation on an n-node treap requires $O(\log n)$ time with high probability. In particular, the expected worst-case time for each of these operations is $O(\log n)$.

Aitch
Are
Ay
Bee
Cue
Dee
Double U
Ee
Ef
El
Em
En
Ess
Ex
Eye
Gee
Jay
Kay
Oh
Pea
See
Tee
Vee
Wy
Yu
Zee

— Sidney Harris, "The Alphabet in Alphabetical Order"

Calvin: *There! I finished our secret code!*
Hobbes: *Let's see.*
Calvin: *I assigned each letter a totally random number, so the code will be hard to crack. For letter "A", you write 3,004,572,688. "B" is 28,731,569$^1/_2$.*
Hobbes: *That's a good code all right.*
Calvin: *Now we just commit this to memory.*
Calvin: *Did you finish your map of our neighborhood?*
Hoobes: *Not yet. How many bricks does the front walk have?*

— Bill Watterson, "Calvin and Hobbes" (August 23, 1990)

7 Hash Tables

7.1 Introduction

A *hash table* is a data structure for storing a set of items, so that we can quickly determine whether an item is or is not in the set. The basic idea is to pick a *hash function* h that maps every possible item x to a small integer $h(x)$. Then we store x in slot $h(x)$ in an array. The array is the hash table.

Let's be a little more specific. We want to store a set of n items. Each item is an element of some finite[1] set $\mathcal{U}$ called the *universe*; we use u to denote the size of the universe, which is just the number of items in $\mathcal{U}$. A hash table is an array $T[1\,..\,m]$, where m is another positive integer, which we call the *table size*. Typically, m is much smaller than u. A *hash function* is a function

$$h\colon \mathcal{U} \to \{0, 1, \dots, m-1\}$$

that maps each possible item in $\mathcal{U}$ to a slot in the hash table. We say that an item x *hashes* to the slot $T[h(x)]$.

Of course, if $u = m$, then we can always just use the trivial hash function $h(x) = x$. In other words, use the item itself as the index into the table. This is called a *direct access table* (or more commonly, an *array*). In most applications, though, the universe of possible keys is orders of magnitude too large for this approach to be practical. Even when it is possible to allocate enough

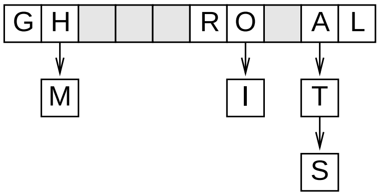
[1] This finiteness assumption is necessary for several of the technical details to work out, but can be ignored in practice. To hash elements from an infinite universe (for example, the positive integers), pretend that the universe is actually finite but very very large. In fact, in *real* practice, the universe actually *is* finite but very very large. For example, on most modern computers, there are only 2^{64} integers (unless you use a big integer package like GMP, in which case the number of integers is closer to $2^{2^{32}}$.)

memory, we usually need to store only a small fraction of the universe. Rather than wasting lots of space, we should make m roughly equal to n, the number of items in the set we want to maintain.

What we'd like is for every item in our set to hash to a different position in the array. Unfortunately, unless $m = u$, this is too much to hope for, so we have to deal with *collisions*. We say that two items x and y *collide* if the have the same hash value: $h(x) = h(y)$. Since we obviously can't store two items in the same slot of an array, we need to describe some methods for *resolving* collisions. The two most common methods are called *chaining* and *open addressing*.

7.2 Chaining

In a *chained* hash table, each entry $T[i]$ is not just a single item, but rather (a pointer to) a linked list of all the items that hash to $T[i]$. Let $\ell(x)$ denote the length of the list $T[h(x)]$. To see if an item x is in the hash table, we scan the entire list $T[h(x)]$. The worst-case time required to search for x is $O(1)$ to compute $h(x)$ plus $O(1)$ for every element in $T[h(x)]$, or $O(1 + \ell(x))$ overall. Inserting and deleting x also take $O(1 + \ell(x))$ time.



A chained hash table with load factor 1.

In the worst case, every item would be hashed to the same value, so we'd get just one long list of n items. In principle, for any deterministic hashing scheme, a malicious adversary can always present a set of items with exactly this property. In order to defeat such malicious behavior, we'd like to use a hash function that is as random as possible. Choosing a truly random hash function is completely impractical, but there are several heuristics for producing hash functions that behave randomly, or at least close to randomly on real data. Thus, we will analyze the performance as though our hash function were truly random. More formally, we make the following assumption.

Simple uniform hashing assumption: $\boxed{\text{If } x \neq y \text{ then } \Pr\left[h(x) = h(y)\right] = 1/m.}$

In the next section, I'll describe a small set of functions with the property that a random hash function in this set satisfies the simple uniform hashing assumption. Most actual implementations of has tables use *deterministic* hash functions. These clearly violate the uniform hashing assumption—the collision probability is either 0 or 1, depending on the pair of items! Nevertheless, it is common practice to adopt the uniform hashing assumption as a convenient fiction for purposes of analysis.

Let's compute the expected value of $\ell(x)$ under this assumption; this will immediately imply a bound on the expected time to search for an item x. To be concrete, let's suppose that x is not already stored in the hash table. For all items x and y, we define the indicator variable

$$C_{x,y} = \big[h(x) = h(y)\big].$$

(In case you've forgotten the bracket notation, $C_{x,y} = 1$ if $h(x) = h(y)$ and $C_{x,y} = 0$ if $h(x) \neq h(y)$.) Since the length of $T[h(x)]$ is precisely equal to the number of items that collide with x, we have

$$\ell(x) = \sum_{y \in T} C_{x,y}.$$

We can rewrite the simple uniform hashing assumption as follows:

$$x \neq y \implies \mathrm{E}[C_{x,y}] = \Pr[C_{x,y} = 1] = \frac{1}{m}.$$

Now we just have to grind through the definitions.

$$\mathrm{E}[\ell(x)] = \sum_{y \in T} \mathrm{E}[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}$$

We call this fraction n/m the *load factor* of the hash table. Since the load factor shows up everywhere, we will give it its own symbol α.

$$\boxed{\alpha = \frac{n}{m}}$$

Our analysis implies that the expected time for an unsuccessful search in a chained hash table is $\Theta(1+\alpha)$. As long as the number if items n is only a constant factor bigger than the table size m, the search time is a constant. A similar analysis gives the same expected time bound (with a slightly smaller constant) for a successful search.

Obviously, linked lists are not the only data structure we could use to store the chains; any data structure that can store a set of items will work. For example, if the universe $\mathcal{U}$ has a total ordering, we can store each chain in a balanced binary search tree. This reduces the expected time for any search to $O(1 + \log \ell(x))$, and under the simple uniform hashing assumption, the expected time for any search is $O(1 + \log \alpha)$.

Another natural possibility is to work recursively! Specifically, for each $T[i]$, we maintain a hash table T_i containing all the items with hash value i. Collisions in those secondary tables are resolved recursively, by storing secondary overflow lists in tertiary hash tables, and so on. The resulting data structure is a tree of hash tables, whose leaves correspond to items that (at some level of the tree) are hashed without any collisions. If every hash table in this tree has size m, then the expected time for any search is $O(\log_m n)$. In particular, if we set $m = \sqrt{n}$, the expected time for any search is *constant*. On the other hand, there is no inherent reason to use the same hash table size everywhere; after all, hash tables deeper in the tree are storing fewer items.

Caveat Lector![2] The preceding analysis does *not* imply bounds on the expected *worst-case* search time is constant. The expected worst-case search time is $O(1 + L)$, where $L = \max_x \ell(x)$. Under the uniform hashing assumption, the maximum list size L is *very* likely to grow faster than any constant, unless the load factor α is *significantly* smaller than 1. For example, $\mathrm{E}[L] = \Theta(\log n / \log\log n)$ when $\alpha = 1$. We've stumbled on a powerful but counterintuitive fact about probability: When several individual items are distributed independently and uniformly at random, the resulting distribution is *not* uniform in the traditional sense! In a later section, I'll describe how to achieve constant expected worst-case search time using secondary hash tables.

7.3 Universal Hashing

Now I'll describe a method to generate random hash functions that satisfy the simple uniform hashing assumption. We say that a set $\mathcal{H}$ of hash function is *universal* if it satisfies the following property: For any items $x \neq y$, if a hash function h is chosen *uniformly at random* from the set $\mathcal{H}$, then $\Pr[h(x) = h(y)] = 1/m$. Note that this probability holds for *any* items x and y; the randomness is entirely in choosing a hash function from the set $\mathcal{H}$.

[2] No, this is not the name of Hannibal's brother. It's Latin for "Reader beware!"

To simplify the following discussion, I'll assume that the universe $\mathcal{U}$ contains exactly m^2 items, each represented as a pair (x, x') of integers between 0 and $m-1$. (Think of the items as two-digit numbers in base m.) I will also assume that m is a prime number.

For any integers $0 \le a, b \le m-1$, define the function $h_{a,b} \colon \mathcal{U} \to \{0, 1, \ldots, m-1\}$ as follows:

$$h_{a,b}(x, x') = (ax + bx') \bmod m.$$

Then the set

$$\mathcal{H} = \{h_{a,b} \mid 0 \le a, b \le m-1\}$$

of all such functions is universal. To prove this, we need to show that for any pair of distinct items $(x, x') \ne (y, y')$, exactly m of the m^2 functions in $\mathcal{H}$ cause a collision.

Choose two items $(x, x') \ne (y, y')$, and assume without loss of generality[3] that $x \ne y$. A function $h_{a,b} \in \mathcal{H}$ causes a collision between (x, x') and (y, y') if and only if

$$\begin{aligned} h_{a,b}(x, x') &= h_{a,b}(y, y') \\ (ax + bx') \bmod m &= (ay + by') \bmod m \\ ax + bx' &\equiv ay + by' \pmod{m} \\ a(x - y) &\equiv b(y' - x') \pmod{m} \\ a &\equiv \frac{b(y' - x')}{x - y} \pmod{m}. \end{aligned}$$

In the last step, we are using the fact that m is prime and $x - y \ne 0$, which implies that $x - y$ has a unique multiplicative inverse modulo m. (For example, the multiplicative inverse of 12 modulo 17 is 10, since $12 \cdot 10 = 120 \equiv 1 \pmod{17}$.) For each possible value of b, the last identity defines a *unique* value of a such that $h_{a,b}$ causes a collision. Since there are m possible values for b, there are exactly m hash functions $h_{a,b}$ that cause a collision, which is exactly what we needed to prove.

Thus, if we want to achieve the constant expected time bounds described earlier, we should choose a random element of $\mathcal{H}$ as our hash function, by generating two numbers a and b uniformly at random between 0 and $m-1$. This is *precisely* the same as choosing a element of $\mathcal{U}$ uniformly at random.

One perhaps undesirable 'feature' of this construction is that we have a small chance of choosing the trivial hash function $h_{0,0}$, which maps everything to 0. So in practice, if we happen to pick $a = b = 0$, we should reject that choice and pick new random numbers. By taking $h_{0,0}$ out of consideration, we reduce the probability of a collision from $1/m$ to $(m-1)/(m^2-1) = 1/(m+1)$. In other words, the set $\mathcal{H} \setminus \{h_{0,0}\}$ is slightly *better* than universal.

This construction can be easily generalized to larger universes. Suppose $u = m^r$ for some constant r, so that each element $x \in \mathcal{U}$ can be represented by a vector $(x_0, x_1, \ldots, x_{r-1})$ of integers between 0 and $m-1$. (Think of x as an r-digit number written in base m.) Then for each vector $a = (a_0, a_1, \ldots, a_{r-1})$, define the corresponding hash function h_a as follows:

$$h_a(x) = (a_0 x_0 + a_1 x_1 + \cdots + a_{r-1} x_{r-1}) \bmod m.$$

Then the set of all m^r such functions is universal.

[3] 'Without loss of generality' is a phrase that appears (perhaps too) often in combinatorial proofs. What it means is that we are considering one of many possible cases, but once we see the proof for one case, the proofs for all the other cases are obvious thanks to some inherent symmetry. For this proof, we are not explicitly considering what happens when $x = y$ and $x' \ne y'$.

*7.4 High Probability Bounds: Balls and Bins

Although any particular search in a chained hash tables requires only constant expected time, but what about the *worst* search time? Under a stronger version[4] of the uniform hashing assumption, this is equivalent to the following more abstract problem. Suppose we toss n balls independently and uniformly at random into one of n bins. Can we say anything about the number of balls in the fullest bin?

Lemma 1. *If n balls are thrown independently and uniformly into n bins, then with high probability, the fullest bin contains $O(\log n/\log\log n)$ balls.*

Proof: Let X_j denote the number of balls in bin j, and let $\hat{X} = \max_j X_j$ be the maximum number of balls in any bin. Clearly, $\mathrm{E}[X_j] = 1$ for all j.

Now consider the probability that bin j contains exactly k balls. There are $\binom{n}{k}$ choices for those k balls; each chosen ball has probability $1/n$ of landing in bin j; and each of the remaining balls has probability $1 - 1/n$ of missing bin j. Thus,

$$\begin{aligned}\Pr[X_j = k] &= \binom{n}{k}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k}\\ &\le \frac{n^k}{k!}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k}\\ &< \frac{1}{k!}\end{aligned}$$

This bound shrinks super-exponentially as k increases, so we can *very* crudely bound the probability that bin 1 contains *at least* k balls as follows:

$$\Pr[X_j \ge k] < \frac{n}{k!}$$

To prove the high-probability bound, we need to choose a value for k such that $n/k! \approx 1/n^c$ for some constant c. Taking logs of both sides of the desired approximation and applying Stirling's approximation, we find

$$\begin{aligned}\ln k! \approx k\ln k - k &\approx (c+1)\ln n\\ \iff k &\approx \frac{(c+1)\ln n}{\ln k - 1}\\ &\approx \frac{(c+1)\ln n}{\ln\frac{(c+1)\ln n}{\ln k-1} - 1}\\ &= \frac{(c+1)\ln n}{\ln\ln n + \ln(c+1) - \ln(\ln k - 1) - 1}\\ &\approx \frac{(c+1)\ln n}{\ln\ln n}.\end{aligned}$$

We have shown (modulo some algebraic hand-waving that is easy but tedious to clean up) that

$$\Pr\left[X_j \ge \frac{(c+1)\ln n}{\ln\ln n}\right] < \frac{1}{n^c}.$$

[4] The simple uniform hashing assumption requires only *pairwise* independence, but the following analysis requires *full* independence.

This probability bound holds for every bin j. Thus, by the union bound, we conclude that

$$\begin{aligned}\Pr\left[\max_j X_j > \frac{(c+1)\ln n}{\ln\ln n}\right] &= \Pr\left[X_j > \frac{(c+1)\ln n}{\ln\ln n} \text{ for all } j\right] \\ &\le \sum_{j=1}^{n} \Pr\left[X_j > \frac{(c+1)\ln n}{\ln\ln n}\right] \\ &< \frac{1}{n^{c-1}}.\end{aligned}$$ □

A similar analysis shows that if we throw n balls randomly into n bins, then with high probability, at least one bin contains $\Omega(\log n/\log\log n)$ balls.

However, if we make the hash table large enough, we can expect every ball to land in a different bin. Suppose there are m bins. Let C_{ij} be the indicator variable that equals 1 if and only if $i \neq j$ and ball i and ball j land in the same bin, and let $C = \sum_{i<j} C_{ij}$ be the total number of pairwise collisions. Since the balls are thrown uniformly at random, the probability of a collision is exactly $1/m$, so $\mathrm{E}[C] = \binom{n}{2}/m$. In particular, if $m = n^2$, the expected number of collisions is less than $1/2$.

To get a high probability bound, let X_j denote the number of balls in bin j, as in the previous proof. We can easily bound the probability that bin j is empty, by taking the two most significant terms in a binomial expansion:

$$\Pr[X_j = 0] \;=\; \left(1-\frac{1}{m}\right)^n \;=\; \sum_{i=1}^{n}\binom{n}{i}\left(\frac{-1}{m}\right)^i \;=\; 1-\frac{n}{m}+\Theta\left(\frac{n^2}{m^2}\right) \;>\; 1-\frac{n}{m}$$

We can similarly bound the probability that bin j contains exactly one ball:

$$\Pr[X_j = 1] \;=\; n\cdot\frac{1}{m}\left(1-\frac{1}{m}\right)^{n-1} \;=\; \frac{n}{m}\left(1-\frac{n-1}{m}+\Theta\left(\frac{n^2}{m^2}\right)\right) \;>\; \frac{n}{m}-\frac{n(n-1)}{m^2}$$

It follows immediately that $\Pr[X_j > 1] < n(n-1)/m^2$. The union bound now implies that $\Pr[\hat{X} > 1] < n(n-1)/m$. If we set $m = n^{2+\varepsilon}$ for any constant $\varepsilon > 0$, then the probability that no bin contains more than one ball is at least $1 - 1/n^{\varepsilon}$.

Lemma 2. *For any $\varepsilon > 0$, if n balls are thrown independently and uniformly into $n^{2+\varepsilon}$ bins, then with high probability, no bin contains more than one ball.*

7.5 Perfect Hashing

So far we are faced with two alternatives. If we use a small hash table, to keep the space usage down, the resulting worst-case expected search time is $\Theta(\log n/\log\log n)$ with high probability, which is not much better than a binary search tree. On the other hand, we can get constant worst-case search time, at least in expectation, by using a table of quadratic size, but that seems unduly wasteful.

Fortunately, there is a fairly simple way to combine these two ideas to get a data structure of linear expected size, whose expected worst-case search time is constant. At the top level, we use a hash table of size n, but instead of linked lists, we use secondary hash tables to resolve collisions. Specifically, the jth secondary hash table has size n_j^2, where n_j is the number of items whose primary hash value is j. The expected worst-case search time in any secondary hash table is $O(1)$, by our earlier analysis.

Although this data structure needs significantly more memory for each secondary structure, the overall increase in space is insignificant, at least in expectation.

Lemma 3. *Under the simple uniform hashing assumption,* $\mathrm{E}[n_j^2] < 2$.

Proof: Let X_{ij} be the indicator variable that equals 1 if item i hashes to slot j in the primary hash table.

$$\begin{aligned}
\mathrm{E}[n_j^2] &= \mathrm{E}\left[\left(\sum_{i=1}^{n} X_{ij}\right)\left(\sum_{k=1}^{n} X_{kj}\right)\right] \\
&= \mathrm{E}\left[\sum_{i=1}^{n}\sum_{k=1}^{n} X_{ij}X_{kj}\right] \\
&= \sum_{i=1}^{n}\sum_{k=1}^{n} \mathrm{E}[X_{ij}X_{kj}] \qquad \text{by linearity of expectation} \\
&= \sum_{i=1}^{n} \mathrm{E}[X_{ij}^2] + 2\sum_{i=1}^{n}\sum_{k=i+1}^{n} \mathrm{E}[X_{ij}X_{kj}]
\end{aligned}$$

Because X_{ij} is an indicator variable, we have $X_{ij}^2 = X_{ij}$, which implies that $\mathrm{E}[X_{ij}^2] = \mathrm{E}[X_{ij}] = 1/n$ by the uniform hashing assumption. The uniform hashing assumption also implies that X_{ij} and X_{kj} are independent whenever $i \neq k$, so $\mathrm{E}[X_{ij}X_{kj}] = \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{kj}] = 1/n^2$. Thus,

$$\mathrm{E}[n_j^2] = \sum_{i=1}^{n}\frac{1}{n} + 2\sum_{i=1}^{n}\sum_{k=i+1}^{n}\frac{1}{n^2} = 1 + 2\binom{n}{2}\frac{1}{n^2} = 2 - \frac{1}{n}. \qquad \square$$

This lemma implies that the expected size of our two-level hash table is $O(n)$. By our earlier analysis, the expected worst-case search time is $O(1)$.

7.6 Open Addressing

Another method used to resolve collisions in hash tables is called *open addressing*. Here, rather than building secondary data structures, we resolve collisions by looking elsewhere in the table. Specifically, we have a sequence of hash functions $\langle h_0, h_1, h_2, \ldots, h_{m-1}\rangle$, such that for any item x, the *probe sequence* $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x)\rangle$ is a permutation of $\langle 0, 1, 2, \ldots, m-1\rangle$. In other words, different hash functions in the sequence always map x to different locations in the hash table.

We search for x using the following algorithm, which returns the array index i if $T[i] = x$, 'absent' if x is not in the table but there is an empty slot, and 'full' if x is not in the table and there no no empty slots.

```
OpenAddressSearch($x$):
  for $i \leftarrow 0$ to $m-1$
      if $T[h_i(x)] = x$
          return $h_i(x)$
      else if $T[h_i(x)] = \varnothing$
          return 'absent'
  return 'full'
```

The algorithm for inserting a new item into the table is similar; only the second-to-last line is changed to $T[h_i(x)] \leftarrow x$. Notice that for an open-addressed hash table, the load factor is never bigger than 1.

Just as with chaining, we'd like to pretend that the sequence of hash values is random. For purposes of analysis, there is a stronger uniform hashing assumption that gives us constant expected search and insertion time.

Strong uniform hashing assumption:

For any item x, the probe sequence $\langle h_0(x), h_1(x), \dots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the set $\{0, 1, 2, \dots, m-1\}$.

Let's compute the expected time for an unsuccessful search using this stronger assumption. Suppose there are currently n elements in the hash table. Our strong uniform hashing assumption has two important consequences:

- The initial hash value $h_0(x)$ is equally likely to be any integer in the set $\{0, 1, 2, \dots, m-1\}$.
- If we ignore the first probe, the remaining probe sequence $\langle h_1(x), h_2(x), \dots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the smaller set $\{0, 1, 2, \dots, m-1\} \setminus \{h_0(x)\}$.

The first sentence implies that the probability that $T[h_0(x)]$ is occupied is exactly n/m. The second sentence implies that if $T[h_0(x)]$ is occupied, *our search algorithm recursively searches the rest of the hash table!* Since the algorithm will never again probe $T[h_0(x)]$, for purposes of analysis, we might as well pretend that slot in the table no longer exists. Thus, we get the following recurrence for the expected number of probes, as a function of m and n:

$$\mathrm{E}[T(m,n)] = 1 + \frac{n}{m}\,\mathrm{E}[T(m-1, n-1)].$$

The trivial base case is $T(m, 0) = 1$; if there's nothing in the hash table, the first probe always hits an empty slot. We can now easily prove by induction that $\boxed{\mathrm{E}[T(m,n)] \le m/(m-n)}$:

$$\begin{aligned}
\mathrm{E}[T(m,n)] &= 1 + \frac{n}{m}\,\mathrm{E}[T(m-1, n-1)] \\
&\le 1 + \frac{n}{m} \cdot \frac{m-1}{m-n} && \text{[induction hypothesis]} \\
&< 1 + \frac{n}{m} \cdot \frac{m}{m-n} && [m-1 < m] \\
&= \frac{m}{m-n} \checkmark && \text{[algebra]}
\end{aligned}$$

Rewriting this in terms of the load factor $\alpha = n/m$, we get $\boxed{\mathrm{E}[T(m,n)] \le 1/(1-\alpha)}$. In other words, the expected time for an unsuccessful search is $O(1)$, unless the hash table is almost completely full.

In practice, however, we can't generate truly random probe sequences, so we use one of the following heuristics:

- **Linear probing:** We use a single hash function $h(x)$, and define $h_i(x) = (h(x) + i) \bmod m$. This is nice and simple, but collisions tend to make items in the table clump together badly, so this is not really a good idea.
- **Quadratic probing:** We use a single hash function $h(x)$, and define $h_i(x) = (h(x) + i^2) \bmod m$. Unfortunately, for certain values of m, the sequence of hash values $\langle h_i(x)\rangle$ does not hit every possible slot in the table; we can avoid this problem by making m a prime number. (That's often a good idea anyway.) Although quadratic probing does not suffer from the same clumping problems as linear probing, it does have a weaker clustering problem: If two items have the same initial hash value, their entire probe sequences will be the same.

- **Double hashing:** We use two hash functions $h(x)$ and $h'(x)$, and define h_i as follows:

  $$h_i(x) = (h(x) + i \cdot h'(x)) \bmod m$$

  To guarantee that this can hit every slot in the table, the *stride* function $h'(x)$ and the table size m must be relatively prime. We can guarantee this by making m prime, but a simpler solution is to make m a power of 2 and choose a stride function that is always odd. Double hashing avoids the clustering problems of linear and quadratic probing. In fact, the actual performance of double hashing is almost the same as predicted by the uniform hashing assumption, at least when m is large and the component hash functions h and h' are sufficiently random. This is the method of choice![5]

7.7 Deleting from an Open-Addressed Hash Table

This section assumes familiarity with amortized analysis.

Deleting an item x from an open-addressed hash table is a bit more difficult than in a chained hash table. We can't simply clear out the slot in the table, because we may need to know that $T[h(x)]$ is occupied in order to find some other item!

Instead, we should delete more or less the way we did with scapegoat trees. When we delete an item, we mark the slot that used to contain it as a *wasted* slot. A sufficiently long sequence of insertions and deletions could eventually fill the table with marks, leaving little room for any real data and causing searches to take linear time.

However, we can still get good *amortized* performance by using two rebuilding rules. First, if the number of items in the hash table exceeds $m/4$, double the size of the table ($m \leftarrow 2m$) and rehash everything. Second, if the number of wasted slots exceeds $m/2$, clear all the marks and rehash everything in the table. Rehashing everything takes m steps to create the new hash table and $O(n)$ expected steps to hash each of the n items. By charging a \$4 tax for each insertion and a \$2 tax for each deletion, we expect to have enough money to pay for any rebuilding.

In conclusion, the *expected amortized* cost of any insertion or deletion is $O(1)$, under the uniform hashing assumption. Notice that we're doing two very different kinds of averaging here. On the one hand, we are averaging the possible costs of *each individual search* over all possible probe sequences ('expected'). On the other hand, we are also averaging the costs of *the entire sequence* of operations to 'smooth out' the cost of rebuilding ('amortized'). Both randomization and amortization are necessary to get this constant time bound.

[5] ...unless your hash tables are really huge, in which case linear probing has *far* better cache behavior, especially when the load factor is small.

> *The goode workes that men don whil they ben in good lif al amortised by synne folwyng.*
>
> — Geoffrey Chaucer, "The Persones [Parson's] Tale" (c.1400)
>
> *I will gladly pay you Tuesday for a hamburger today.*
>
> — J. Wellington Wimpy, "Thimble Theatre" (1931)
>
> *I want my two dollars!*
>
> — Johnny Gasparini [Demian Slade], "Better Off Dead" (1985)

8 Amortized Analysis

8.1 Incrementing a Binary Counter

It is a straightforward exercise in induction, which often appears on Homework 0, to prove that any non-negative integer n can be represented as the sum of distinct powers of two. Although some students correctly use induction on the number of bits—pulling off either the least significant bit or the most significant bit in the binary representation and letting the Recursion Fairy convert the remainder—the most commonly submitted proof uses induction on the value of the integer, as follows:

Proof: The base case $n = 0$ is trivial. For any $n > 0$, the inductive hypothesis implies that there is set of distinct powers of two whose sum is $n - 1$. If we add 2^0 to this set, we obtain a 'set' of powers of two whose sum is n, but there might be two copies of 2^0. To fix this, as long as there are two copies of any 2^i, we remove them both from the 'set' and insert 2^{i+1} in their place. The sum of the powers of two in our 'set' is unchanged by this replacement, since $2^{i+1} = 2^i + 2^i$. Each iteration decreases the number of powers of two in our 'set', so this replacement process must eventually terminate. When it does terminate, we have a set of distinct powers of two whose sum is n. □

This proof is describing an algorithm to increment a binary counter from $n - 1$ to n. Here's a more formal (and shorter!) description of the algorithm to add one to a binary counter. The input B is an (infinite) array of bits, where $B[i] = 1$ if and only if 2^i appears in the sum.

```
INCREMENT($B$):
  $i \leftarrow 0$
  while $B[i] = 1$
      $B[i] \leftarrow 0$
      $i \leftarrow i + 1$
  $B[i] \leftarrow 1$
```

We've already argued that INCREMENT must terminate, but how quickly? Obviously, the running time depends on the array of bits passed as input. If the first k bits are all 1s, then INCREMENT takes $\Theta(k)$ time. Thus, if the number represented by B is between 0 and n, INCREMENT takes $\Theta(\log n)$ time in the worst case, since the binary representation for n is exactly $\lfloor \lg n \rfloor + 1$ bits long.

8.2 Counting from 0 to n

Now suppose we want to use INCREMENT to count from 0 to n. If we only use the worst-case running time for each call, we get an upper bound of $O(n \log n)$ on the total running time. Although this bound is correct, it isn't the best we can do. There are several general methods for proving that the total running time is actually only $O(n)$. Many of these methods are logically equivalent, but different formulations are more natural for different problems.

8.2.1 The Aggregate Method

The easiest way to get a tighter bound is to observe that we don't need to flip $\Theta(\log n)$ bits *every* time INCREMENT is called. The least significant bit $B[0]$ does flip every time, but $B[1]$ only flips every other time, $B[2]$ flips every 4th time, and in general, $B[i]$ flips every 2^ith time. If we start from an array full of zeros, a sequence of n INCREMENTs flips each bit $B[i]$ exactly $\lfloor n/2^i \rfloor$ times. Thus, the total number of bit-flips for the entire sequence is

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

Thus, *on average*, each call to INCREMENT flips only two bits, and so runs in constant time.

This 'on average' is quite different from the averaging we did in the previous lecture. There is no probability involved; we are averaging over a sequence of operations, not the possible running times of a single operation. This averaging idea is called *amortization*—the *amortized* cost of each INCREMENT is $O(1)$. Amortization is a ~~sleazy~~ clever trick used by accountants to average large one-time costs over long periods of time; the most common example is calculating uniform payments for a loan, even though the borrower is paying interest on less and less capital over time.

There are several different methods for deriving amortized bounds for a sequence of operations. Most textbooks call the technique we just used the *aggregate* method, but this is just a name for adding up the total cost of the sequence and dividing by the number of operations.

> **The Aggregate Method.** *Find the worst case running time $T(n)$ for a sequence of n operations. The amortized cost of each operation is $T(n)/n$.*

8.2.2 The Taxation (Accounting) Method

A second method we can use to derive amortized bounds is called either the *accounting* method or the *taxation* method. Suppose it costs us a dollar to toggle a bit, so we can measure the running time of our algorithm in dollars. Time is money!

Instead of paying for each bit flip when it happens, the Increment Revenue Service charges a two-dollar *increment tax* whenever we want to set a bit from zero to one. One of those dollars is spent changing the bit from zero to one; the other is stored away as *credit* until we need to reset the same bit to zero. The key point here is that we always have enough credit saved up to pay for the next INCREMENT. The amortized cost of an INCREMENT is the total tax it incurs, which is exactly 2 dollars, since each INCREMENT changes just one bit from 0 to 1.

It is often useful to assign various parts of the tax income to specific pieces of the data structure. For example, for each INCREMENT, we could store one of the two dollars on the single bit that is set for 0 to 1, so that *that* bit can pay to reset itself back to zero later on.

> **Taxation Method 1.** *Certain steps in the algorithm charge you taxes, so that the total cost of the algorithm is never more than the total taxes you pay. The amortized cost of an operation is the overall tax charged to you during that operation.*

A different way to schedule the taxes is for *every* bit to charge us a tax at *every* operation, regardless of whether the bit changes of not. Specifically, each bit $B[i]$ charges a tax of $1/2^i$ dollars for each INCREMENT. The total tax we are charged during each INCREMENT is $\sum_{i\geq 0} 2^{-i} = 2$ dollars. Every time a bit $B[i]$ actually needs to be flipped, it has collected exactly \$1, which is just enough for us to pay for the flip.

> **Taxation Method 2.** *Certain portions of the data structure charge you taxes at each operation, so that the total cost of maintaining the data structure is never more than the total taxes you pay. The amortized cost of an operation is the overall tax you pay during that operation.*

In both of the taxation methods, our task as algorithm analysts is to come up with an appropriate 'tax schedule'. Different 'schedules' can result in different amortized time bounds. The tightest bounds are obtained from tax schedules that *just barely* stay in the black.

8.2.3 The Charging Method

Another common method of amortized analysis involves *charging* the cost of some steps to some other, earlier steps. The method is similar to taxation, except that we focus on where each unit of tax is (or will be) spent, rather than where is it collected. By charging the cost of some operations to earlier operations, we are overestimating the total cost of any sequence of operations, since we pay for some charges from future operations that may never actually occur.

For example, in our binary counter, suppose we charge the cost of clearing a bit (changing its value from 1 to 0) to the previous operation that sets that bit (changing its value from 0 to 1). If we flip k bits during an INCREMENT, we charge $k-1$ of those bit-flips to earlier bit-flips. Conversely, the single operation that sets a bit receives at most one unit of charge from the next time that bit is cleared. So instead of paying for k bit-flips, we pay for at most two: one for actually setting a bit, plus at most one charge from the future for clearing that same bit. Thus, the total amortized cost of the INCREMENT is at most two bit-flips.

> **Charging Method.** *Charge the cost of some steps of the algorithm to earlier steps, or to steps in some earlier operation. The amortized cost of the algorithm is its actual running time, minus its total charges to past operations, plus the total charge from future operations.*

8.2.4 The Potential Method

The most powerful method (and the hardest to use) builds on a physics metaphor of 'potential energy'. Instead of associating costs or taxes with particular operations or pieces of the data structure, we represent prepaid work as *potential* that can be spent on later operations. The potential is a function of the entire data structure.

Let D_i denote our data structure after i operations, and let Φ_i denote its potential. Let c_i denote the actual cost of the ith operation (which changes D_{i-1} into D_i). Then the *amortized* cost of the ith operation, denoted a_i, is defined to be the actual cost plus the change in potential:

$$\boxed{a_i = c_i + \Phi_i - \Phi_{i-1}}$$

So the *total* amortized cost of n operations is the actual total cost plus the total change in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

Our task is to define a potential function so that $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i. Once we do this, the total *actual* cost of any sequence of operations will be less than the total *amortized* cost:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

For our binary counter example, we can define the potential Φ_i after the ith INCREMENT to be the number of bits with value 1. Initially, all bits are equal to zero, so $\Phi_0 = 0$, and clearly $\Phi_i > 0$ for all $i > 0$, so this is a legal potential function. We can describe both the actual cost of an INCREMENT and the change in potential in terms of the number of bits set to 1 and reset to 0.

$$\begin{aligned} c_i &= \#\text{bits changed from } 0 \text{ to } 1 + \#\text{bits changed from } 1 \text{ to } 0 \\ \Phi_i - \Phi_{i-1} &= \#\text{bits changed from } 0 \text{ to } 1 - \#\text{bits changed from } 1 \text{ to } 0 \end{aligned}$$

Thus, the amortized cost of the ith INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits changed from } 0 \text{ to } 1$$

Since INCREMENT changes only *one* bit from 0 to 1, the amortized cost INCREMENT is 2.

> **The Potential Method.** *Define a potential function for the data structure that is initially equal to zero and is always nonnegative. The amortized cost of an operation is its actual cost plus the change in potential.*

For this particular example, the potential is exactly equal to the total unspent taxes paid using the taxation method, so not too surprisingly, we have exactly the same amortized cost. In general, however, there may be no way of interpreting the change in potential as 'taxes'.

Different potential functions will lead to different amortized time bounds. The trick to using the potential method is to come up with the best possible potential function. A good potential function goes up a little during any cheap/fast operation, and goes down a lot during any expensive/slow operation. Unfortunately, there is no general technique for doing this other than playing around with the data structure and trying lots of different possibilities.

8.3 Incrementing and Decrementing

Now suppose we wanted a binary counter that we could both increment and decrement efficiently. A standard binary counter won't work, even in an amortized sense, since alternating between 2^k and $2^k - 1$ costs $\Theta(k)$ time per operation.

A nice alternative is represent a number as a pair of bit strings (P, N), where for any bit position i, at most one of the bits $P[i]$ and $N[i]$ is equal to 1. The actual value of the counter is $P - N$. Here are algorithms to increment and decrement our double binary counter.

```
INCREMENT($P, N$):
    $i \leftarrow 0$
    while $P[i] = 1$
        $P[i] \leftarrow 0$
        $i \leftarrow i + 1$
    if $N[i] = 1$
        $N[i] \leftarrow 0$
    else
        $P[i] \leftarrow 1$
```

```
DECREMENT($P, N$):
    $i \leftarrow 0$
    while $N[i] = 1$
        $N[i] \leftarrow 0$
        $i \leftarrow i + 1$
    if $P[i] = 1$
        $P[i] \leftarrow 0$
    else
        $N[i] \leftarrow 1$
```

Here's an example of these algorithms in action. Notice that any number other than zero can be represented in multiple (in fact, infinitely many) ways.

$$\begin{array}{ccccccccccccc}
P = 10001 & & P = 100\mathbf{10} & & P = 1001\mathbf{1} & & P = 100\mathbf{00} & & P = 10000 & & P = 10000 & & P = 1000\mathbf{1} \\
N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01\mathbf{0}00 & \xrightarrow{--} & N = 0100\mathbf{1} & \xrightarrow{--} & N = 010\mathbf{10} & \xrightarrow{++} & N = 01010 \\
P - N = 5 & & P - N = 6 & & P - N = 7 & & P - N = 8 & & P - N = 7 & & P - N = 6 & & P - N = 7
\end{array}$$

Incrementing and decrementing a double-binary counter.

Now suppose we start from $(0, 0)$ and apply a sequence of n INCREMENTs and DECREMENTs. In the worst case, operation takes $\Theta(\log n)$ time, but what is the amortized cost? We can't use the aggregate method here, since we don't know what the sequence of operations looks like.

What about the taxation method? It's not hard to prove (by induction, of course) that after either $P[i]$ or $N[i]$ is set to 1, there must be at least 2^i operations, either INCREMENTs or DECREMENTs, before that bit is reset to 0. So if each bit $P[i]$ and $N[i]$ pays a tax of 2^{-i} at each operation, we will always have enough money to pay for the next operation. Thus, the amortized cost of each operation is at most $\sum_{i \geq 0} 2(\cdot 2^{-i}) = 4$.

We can get even better bounds using the potential method. Define the potential Φ_i to be the number of 1-bits in both P and N after i operations. Just as before, we have

$$\begin{aligned}
c_i &= \#\text{bits changed from } 0 \text{ to } 1 + \#\text{bits changed from } 1 \text{ to } 0 \\
\Phi_i - \Phi_{i-1} &= \#\text{bits changed from } 0 \text{ to } 1 - \#\text{bits changed from } 1 \text{ to } 0 \\
\Longrightarrow \qquad a_i &= 2 \times \#\text{bits changed from } 0 \text{ to } 1
\end{aligned}$$

Since each operation changes *at most* one bit to 1, the ith operation has amortized cost $a_i \leq 2$.

Exercise: Modify the binary double-counter to support a new operation SIGN, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant. *[Hint: If P has p significant bits, and N has n significant bits, then $p - n$ always has the same sign as $P - N$. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. But how do you store p and n??]*

Exercise: Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of *fits*, where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fitstring 101110_F represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a single fitstring in constant amortized time. *[Hint: Most numbers can be represented by more than one fitstring!]*

*8.4 Gray Codes

An attractive alternate solution to the increment/decrement problem was independently suggested by several students. *Gray codes* (named after Frank Gray, who discovered them in the 1950s) are methods for representing numbers as bit strings so that successive numbers differ by only one bit. For example, here is the four-bit *binary reflected* Gray code for the integers 0 through 15:

$$0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$

The general rule for incrementing a binary reflected Gray code is to invert the bit that would be set from 0 to 1 by a normal binary counter. In terms of bit-flips, this is the perfect solution; each increment of decrement *by definition* changes only one bit. Unfortunately, the naïve algorithm to *find* the single bit to flip still requires $\Theta(\log n)$ time in the worst case. Thus, so the total cost of maintaining a Gray code, using the obvious algorithm, is the same as that of maintaining a normal binary counter.

Fortunately, this is only true of the naïve algorithm. The following algorithm, discovered by Gideon Ehrlich[1] in 1973, maintains a Gray code counter in constant *worst-case* time per increment! The algorithm uses a separate 'focus' array $F[0\,..\,n]$ in addition to a Gray-code bit array $G[0\,..\,n-1]$.

```
EHRLICHGRAYINIT($n$):
  for $i \leftarrow 0$ to $n-1$
      $G[i] \leftarrow 0$
  for $i \leftarrow 0$ to $n$
      $F[i] \leftarrow i$
```

```
EHRLICHGRAYINCREMENT($n$):
  $j \leftarrow F[0]$
  $F[0] \leftarrow 0$
  if $j = n$
      $G[n-1] \leftarrow 1 - G[n-1]$
  else
      $G[j] = 1 - G[j]$
      $F[j] \leftarrow F[j+1]$
      $F[j+1] \leftarrow j+1$
```

The EHRLICHGRAYINCREMENT algorithm obviously runs in $O(1)$ time, even in the worst case. Here's the algorithm in action with $n = 4$. The first line is the Gray bit-vector G, and the second line shows the focus vector F, both in reverse order:

$$G : 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$
$$F : 3210, 3211, 3220, 3212, 3310, 3311, 3230, 3213, 4210, 4211, 4220, 4212, 3410, 3411, 3240, 3214$$

Voodoo! I won't explain in detail how Ehrlich's algorithm works, except to point out the following invariant. Let $B[i]$ denote the ith bit in the *standard* binary representation of the current number. **If $B[j] = 0$ and $B[j-1] = 1$, then $F[j]$ is the smallest integer $k > j$ such that $B[k] = 1$; otherwise, $F[j] = j$.** Got that?

But wait — this algorithm only handles increments; what if we also want to decrement? Sorry, I don't have a clue. Extra credit, anyone?

8.5 Generalities and Warnings

Although computer scientists usually apply amortized analysis to understand the efficiency of maintaining and querying data structures, you should remember that amortization can be applied to *any* sequence of numbers. Banks have been using amortization to calculate fixed payments for interest-bearing loans for centuries. The IRS allows taxpayers to amortize business expenses or gambling losses across several years for purposes of computing income taxes. Some cell phone contracts let you to apply amortization to calling time, by rolling unused minutes from one month into the next month.

It's important to keep in mind when you're doing amortized analysis is that *amortized time bounds are not unique*. For a data structure that supports multiple operations, different amortization schemes can assign different costs to *exactly the same* algorithms. For example, consider a generic data structure that can be modified by three algorithms: FOLD, SPINDLE, and *Mutilate*. One amortization scheme might imply that FOLD and SPINDLE each run in $O(\log n)$ amortized time, while MUTILATE runs in $O(n)$ amortized time. Another scheme might imply that FOLD runs in $O(\sqrt{n})$ amortized time, while SPINDLE and MUTILATE each run in $O(1)$ amortized time. These two results are not necessarily inconsistent! Moreover, there is no general reason to prefer one of these sets of amortized time bounds over the other; our preference may depend on the context in which the data structure is used.

[1] Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.* 20:500–513, 1973.

> *Everything was balanced before the computers went off line. Try and adjust something, and you unbalance something else. Try and adjust that, you unbalance two more and before you know what's happened, the ship is out of control.*
>
> — Blake, *Blake's 7*, "Breakdown" (March 6, 1978)

> *A good scapegoat is nearly as welcome as a solution to the problem.*
>
> — Anonymous

> *Let's play.*
>
> — El Mariachi [Antonio Banderas], *Desperado* (1992)

> *CAPTAIN: TAKE OFF EVERY 'ZIG'!!*
> *CAPTAIN: YOU KNOW WHAT YOU DOING.*
> *CAPTAIN: MOVE 'ZIG'.*
> *CAPTAIN: FOR GREAT JUSTICE.*
>
> — *Zero Wing* (1992)

9 Scapegoat and Splay Trees

9.1 Definitions

I'll assume that everyone is already familiar with the standard terminology for binary search trees—node, search key, edge, root, internal node, leaf, right child, left child, parent, descendant, sibling, ancestor, subtree, preorder, postorder, inorder, etc.—as well as the standard algorithms for searching for a node, inserting a node, or deleting a node. Otherwise, consult your favorite data structures textbook.

For this lecture, we will consider only *full* binary trees—where every internal node has *exactly* two children—where only the *internal* nodes actually store search keys. In practice, we can represent the leaves with null pointers.

Recall that the *depth* of a node is its distance from the root, and its *height* is the distance to the farthest leaf in its subtree. The height (or depth) of the tree is just the height of the root. The *size* of a node is the number of nodes in its subtree. The size n of the whole tree is just the total number of nodes.

A tree with height h has at most 2^h leaves, so the minimum height of an n-leaf binary tree is $\lceil \lg n \rceil$. In the worst case, the time required for a search, insertion, or deletion to the height of the tree, so in general we would like keep the height as close to $\lg n$ as possible. The best we can possibly do is to have a *perfectly balanced* tree, in which each subtree has as close to half the leaves as possible, and both subtrees are perfectly balanced. The height of a perfectly balanced tree is $\lceil \lg n \rceil$, so the worst-case search time is $O(\log n)$. However, even if we started with a perfectly balanced tree, a malicious sequence of insertions and/or deletions could make the tree arbitrarily unbalanced, driving the search time up to $\Theta(n)$.

To avoid this problem, we need to periodically modify the tree to maintain 'balance'. There are several methods for doing this, and depending on the method we use, the search tree is given a different name. Examples include AVL trees, red-black trees, height-balanced trees, weight-balanced trees, bounded-balance trees, path-balanced trees, B-trees, treaps, randomized binary search trees, skip lists,[1] and jumplists. Some of these trees support searches, insertions, and deletions, in $O(\log n)$ *worst-case* time, others in $O(\log n)$ *amortized* time, still others in $O(\log n)$ *expected* time.

[1] Yeah, yeah. Skip lists aren't really binary search trees. Whatever you say, Mr. Picky.

In this lecture, I'll discuss two binary search tree data structures with good *amortized* performance. The first is the *scapegoat tree*, discovered by Arne Andersson* in 1989 [1, 2] and independently[2] by Igal Galperin* and Ron Rivest in 1993 [9]. The second is the *splay tree*, discovered by Danny Sleator and Bob Tarjan in 1981 [13, 11].

9.2 Lazy Deletions: Global Rebuilding

First let's consider the simple case where we start with a perfectly-balanced tree, and we only want to perform searches and deletions. To get good search and delete times, we will use a technique called *global rebuilding*. When we get a delete request, we locate and mark the node to be deleted, *but we don't actually delete it*. This requires a simple modification to our search algorithm—we still use marked nodes to guide searches, but if we search for a marked node, the search routine says it isn't there. This keeps the tree more or less balanced, but now the search time is no longer a function of the amount of data currently stored in the tree. To remedy this, we also keep track of how many nodes have been marked, and then apply the following rule:

> **Global Rebuilding Rule.** *As soon as half the nodes in the tree have been marked, rebuild a new perfectly balanced tree containing only the unmarked nodes.*[3]

With this rule in place, a search takes $O(\log n)$ time in the worst case, where n is the number of unmarked nodes. Specifically, since the tree has at most n marked nodes, or $2n$ nodes altogether, we need to examine at most $\lg n + 1$ keys. There are several methods for rebuilding the tree in $O(n)$ time, where n is the size of the new tree. (Homework!) So a single deletion can cost $\Theta(n)$ time in the worst case, but only if we have to rebuild; most deletions take only $O(\log n)$ time.

We spend $O(n)$ time rebuilding, but only after $\Omega(n)$ deletions, so the *amortized* cost of rebuilding the tree is $O(1)$ per deletion. (Here I'm using a simple version of the 'taxation method'. For each deletion, we charge a $\$1$ tax; after n deletions, we've collected $\$n$, which is just enough to pay for rebalancing the tree containing the remaining n nodes.) Since we also have to find and mark the node being 'deleted', the total amortized time for a deletion is $\boxed{O(\log n)}$.

9.3 Insertions: Partial Rebuilding

Now suppose we only want to support searches and insertions. We can't 'not really insert' new nodes into the tree, since that would make them unavailable to the search algorithm.[4] So instead, we'll use another method called *partial rebuilding*. We will insert new nodes normally, but whenever a *subtree* becomes unbalanced enough, we rebuild it. The definition of 'unbalanced enough' depends on an arbitrary constant $\alpha > 1$.

Each node v will now also store *height*(v) and *size*(v). We now modify our insertion algorithm with the following rule:

> **Partial Rebuilding Rule.** *After we insert a node, walk back up the tree updating the heights and sizes of the nodes on the search path. If we encounter a node v where height$(v) > \alpha \cdot \lg($size$(v))$, rebuild its subtree into a perfectly balanced tree (in $O($size$(v))$ time).*

[2] The claim of independence is Andersson's [2]. The two papers actually describe very slightly different rebalancing algorithms. The algorithm I'm using here is closer to Andersson's, but my analysis is closer to Galperin and Rivest's.

[3] Alternately: When the number of unmarked nodes is one less than an exact power of two, rebuild the tree. This rule ensures that the tree is always *exactly* balanced.

[4] Well, we could use the Bentley-Saxe* logarithmic method [3], but that would raise the query time to $O(\log^2 n)$.

If we always follow this rule, then after an insertion, the height of the tree is at most $\alpha \cdot \lg n$. Thus, since α is a constant, the worst-case search time is $O(\log n)$. In the worst case, insertions require $\Theta(n)$ time—we might have to rebuild the entire tree. However, the *amortized* time for each insertion is again only $O(\log n)$. Not surprisingly, the proof is a little bit more complicated than for deletions.

Define the *imbalance* $I(v)$ of a node v to be one less than the absolute difference between the sizes of its two subtrees, or zero, whichever is larger:

$$\boxed{I(v) = \max\big\{0,\ |\mathit{size}(\mathit{left}(v)) - \mathit{size}(\mathit{right}(v))| - 1\big\}}$$

A simple induction proof implies that $I(v) = 0$ for every node v in a perfectly balanced tree. So immediately after we rebuild the subtree of v, we have $I(v) = 0$. On the other hand, each insertion into the subtree of v increments either $\mathit{size}(\mathit{left}(v))$ or $\mathit{size}(\mathit{right}(v))$, so $I(v)$ changes by at most 1.

The whole analysis boils down to the following lemma.

Lemma 1. *Just before we rebuild v's subtree, $I(v) = \Omega(\mathit{size}(v))$.*

Before we prove this, let's first look at what it implies. If $I(v) = \Omega(\mathit{size}(v))$, then $\Omega(\mathit{size}(v))$ keys have been inserted in the v's subtree since the last time it was rebuilt from scratch. On the other hand, rebuilding the subtree requires $O(\mathit{size}(v))$ time. Thus, if we amortize the rebuilding cost across all the insertions since the last rebuilding, v is charged *constant* time for each insertion into its subtree. Since each new key is inserted into at most $\alpha \cdot \lg n = O(\log n)$ subtrees, the total amortized cost of an insertion is $\boxed{O(\log n)}$.

Proof: Since we're about to rebuild the subtree at v, we must have $\mathit{height}(v) > \alpha \cdot \lg \mathit{size}(v)$. Without loss of generality, suppose that the node we just inserted went into v's left subtree. Either we just rebuilt this subtree or we didn't have to, so we also have $\mathit{height}(\mathit{left}(v)) \le \alpha \cdot \lg \mathit{size}(\mathit{left}(v))$. Combining these two inequalities with the recursive definition of height, we get

$$\alpha \cdot \lg \mathit{size}(v) \ <\ \mathit{height}(v) \ \le\ \mathit{height}(\mathit{left}(v)) + 1 \ \le\ \alpha \cdot \lg \mathit{size}(\mathit{left}(v)) + 1.$$

After some algebra, this simplifies to $\mathit{size}(\mathit{left}(v)) > \mathit{size}(v)/2^{1/\alpha}$. Combining this with the identity $\mathit{size}(v) = \mathit{size}(\mathit{left}(v)) + \mathit{size}(\mathit{right}(v)) + 1$ and doing some more algebra gives us the inequality

$$\mathit{size}(\mathit{right}(v)) < \big(1 - 1/2^{1/\alpha}\big)\mathit{size}(v) - 1.$$

Finally, we combine these two inequalities using the recursive definition of imbalance.

$$I(v) \ \ge\ \mathit{size}(\mathit{left}(v)) - \mathit{size}(\mathit{right}(v)) - 1 \ >\ \big(2/2^{1/\alpha} - 1\big)\mathit{size}(v)$$

Since α is a constant bigger than 1, the factor in parentheses is a positive constant. □

9.4 Scapegoat Trees

Finally, to handle both insertions and deletions efficiently, *scapegoat trees* use both of the previous techniques. We use partial rebuilding to re-balance the tree after insertions, and global rebuilding to re-balance the tree after deletions. Each search takes $O(\log n)$ time in the worst case, and the amortized time for any insertion or deletion is also $O(\log n)$. There are a few small technical details left (which I won't describe), but no new ideas are required.

Once we've done the analysis, we can actually simplify the data structure. It's not hard to prove that at most one subtree (the *scapegoat*) is rebuilt during any insertion. Less obviously, we can even get the same amortized time bounds (except for a small constant factor) if we only maintain the three integers in addition to the actual tree: the size of the entire tree, the height of the entire tree, and the number of marked nodes. Whenever an insertion causes the tree to become unbalanced, we can compute the sizes of all the subtrees on the search path, starting at the new leaf and stopping at the scapegoat, in time proportional to the size of the scapegoat subtree. Since we need that much time to re-balance the scapegoat subtree, this computation increases the running time by only a small constant factor! Thus, unlike almost every other kind of balanced trees, scapegoat trees require only $O(1)$ extra space.

9.5 Rotations, Double Rotations, and Splaying

Another method for maintaining balance in binary search trees is by adjusting the shape of the tree locally, using an operation called a *rotation*. A rotation at a node x decreases its depth by one and increases its parent's depth by one. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

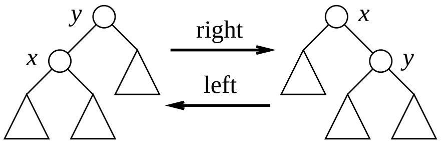


Figure 1. A right rotation at x and a left rotation at y are inverses.

For technical reasons, we will need to use rotations two at a time. There are two types of double rotations, which might be called *zig-zag* and *roller-coaster*. A zig-zag at x consists of two rotations at x, in opposite directions. A roller-coaster at a node x consists of a rotation at x's parent followed by a rotation at x, both in the same direction. Each double rotation decreases the depth of x by two, leaves the depth of its parent unchanged, and increases the depth of its grandparent by either one or two, depending on the type of double rotation. Either type of double rotation can be performed in constant time.

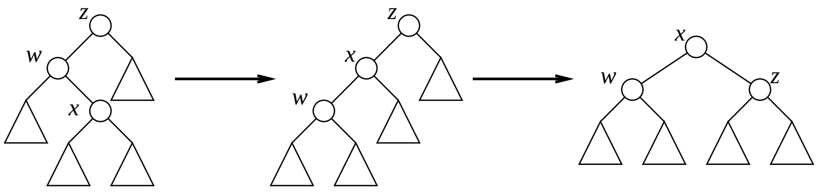


Figure 2. A zig-zag at x. The symmetric case is not shown.

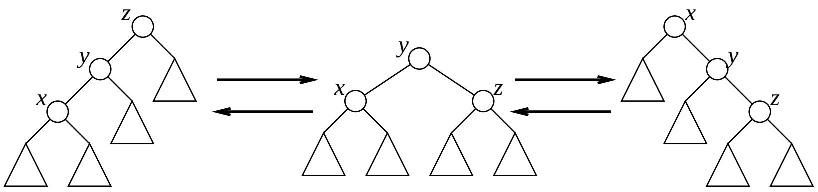


Figure 3. A right roller-coaster at x and a left roller-coaster at z.

Finally, a *splay* operation moves an arbitrary node in the tree up to the root through a series of double rotations, possibly with one single rotation at the end. Splaying a node v requires time proportional to *depth*(v). (Obviously, this means the depth *before* splaying, since after splaying v is the root and thus has depth zero!)

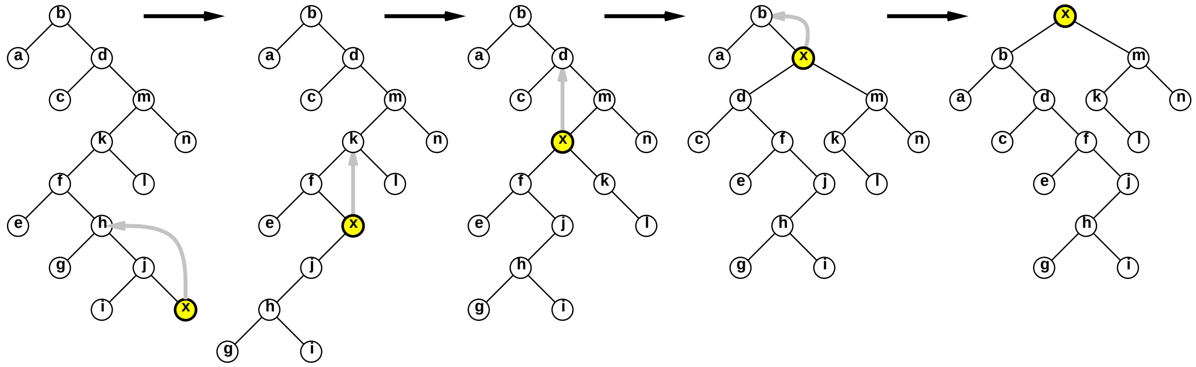

b a d c m k n f l e h g j i x
b a d c m k n f l e x j h g i
b a d c m x n f k e j l h g i
b a x d m c f k n e j l h g i
x b m a d k n c f l e j h g i

Figure 4. Splaying a node. Irrelevant subtrees are omitted for clarity.

9.6 Splay Trees

A *splay tree* is a binary search tree that is kept more or less balanced by splaying. Intuitively, after we access any node, we move it to the root with a splay operation. In more detail:

- **Search:** Find the node containing the key using the usual algorithm, or its predecessor or successor if the key is not present. Splay whichever node was found.
- **Insert:** Insert a new node using the usual algorithm, then splay that node.
- **Delete:** Find the node x to be deleted, splay it, and then delete it. This splits the tree into two subtrees, one with keys less than x, the other with keys bigger than x. Find the node w in the left subtree with the largest key (*i.e.*, the inorder predecessor of x in the original tree), splay it, and finally join it to the right subtree.

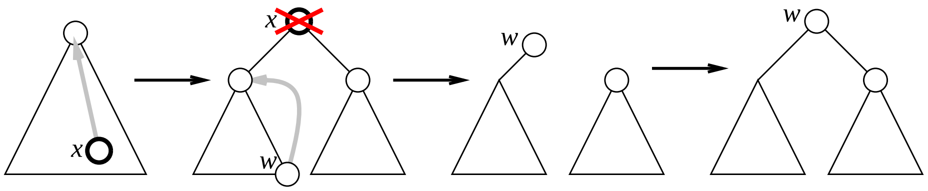


Figure 5. Deleting a node in a splay tree.

Each search, insertion, or deletion consists of a constant number of operations of the form *walk down to a node, and then splay it up to the root*. Since the walk down is clearly cheaper than the splay up, all we need to get good amortized bounds for splay trees is to derive good amortized bounds for a single splay.

Believe it or not, the easiest way to do this uses the potential method. We define the *rank* of a node v to be $\lfloor \lg \mathit{size}(v) \rfloor$, and the *potential* of a splay tree to be the sum of the ranks of its nodes:

$$\boxed{\Phi = \sum_v \mathit{rank}(v) = \sum_v \lfloor \lg \mathit{size}(v) \rfloor}$$

It's not hard to observe that a perfectly balanced binary tree has potential $\Theta(n)$, and a linear chain of nodes (a perfectly *unbalanced* tree) has potential $\Theta(n \log n)$.

The amortized analysis of splay trees boils down to the following lemma. Here, $\mathit{rank}(v)$ denotes the rank of v before a (single or double) rotation, and $\mathit{rank}'(v)$ denotes its rank afterwards. Recall that the amortized cost is defined to be the number of rotations plus the drop in potential.

The Access Lemma. *The amortized cost of a single rotation at any node v is at most $1+3\,\mathit{rank}'(v)-3\,\mathit{rank}(v)$, and the amortized cost of a double rotation at any node v is at most $3\,\mathit{rank}'(v)-3\,\mathit{rank}(v)$.*

Proving this lemma is a straightforward but tedious case analysis of the different types of rotations. For the sake of completeness, I'll give a proof (of a generalized version) in the next section.

By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node v is at most $1 + 3\,\mathit{rank}'(v) - 3\,\mathit{rank}(v)$, where $\mathit{rank}'(v)$ is the rank of v after the entire splay. (The intermediate ranks cancel out in a nice telescoping sum.) But after the splay, v is the root! Thus, $\mathit{rank}'(v) = \lfloor \lg n \rfloor$, which implies that the amortized cost of a splay is at most $3 \lg n - 1 = O(\log n)$.

We conclude that every insertion, deletion, or search in a splay tree takes $O(\log n)$ amortized time.

*9.7 Other Optimality Properties

In fact, splay trees are optimal in several other senses. Some of these optimality properties follow easily from the following generalization of the Access Lemma.

Let's arbitrarily assign each node v a non-negative real *weight* $w(v)$. These weights are not actually stored in the splay tree, nor do they affect the splay algorithm in any way; they are only used to help with the analysis. We then redefine the *size* $s(v)$ of a node v to be the sum of the weights of the descendants of v, including v itself: $s(v) = w(v) + s(\mathit{right}(v)) + s(\mathit{left}(v))$. If $w(v) = 1$ for every node v, then the size of anode is just the numebr of nodes in its subtree, as in the previous section. As before, we define the *rank* of any node v is as $r(v) = \lfloor \lg s(v) \rfloor$, and the emphpotential of a splay tree to be the sum of the ranks of all its nodes:

$$\Phi = \sum_v r(v) = \sum_v \lfloor \lg s(v) \rfloor$$

In the following lemma, $r(v)$ denotes the rank of v before a (single or double) rotation, and $r'(v)$ denotes its rank afterwards.

The Generalized Access Lemma. *For **any** assignment of non-negative weights to the nodes, the amortized cost of a single rotation at any node x is at most $1 + 3r'(x) - 3r(x)$, and the amortized cost of a double rotation at any node v is at most $3r'(x) - 3r(x)$.*

Proof: First consider a single rotation, as shown in Figure 1.

$$\begin{aligned}
1 + \Phi' - \Phi &= 1 + r'(x) + r'(y) - r(x) - r(y) && [\text{only } x \text{ and } y \text{ change rank}]\\
&\le 1 + r'(x) - r(x) && [r'(y) \le r(y)]\\
&\le 1 + 3r'(x) - 3r(x) && [r'(x) \ge r(x)]
\end{aligned}$$

Now consider a zig-zag, as shown in Figure 2. Only w, x, and z change rank.

$$\begin{aligned}
&2 + \Phi' - \Phi \\
&\qquad = 2 + r'(w) + r'(x) + r'(z) - r(w) - r(x) - r(z) && \text{[only } w, x, z \text{ change rank]} \\
&\qquad \le 2 + r'(w) + r'(z) - 2r(x) && [r(x) \le r(w) \text{ and } r'(x) = r(z)] \\
&\qquad = 2(r'(x) - r(x)) + 2 + r'(w) + r'(z) - 2r'(x) \\
&\qquad = 2(r'(x) - r(x)) + 2 + \lg\frac{s'(w)}{s'(x)} + \lg\frac{s'(z)}{s'(x)} \\
&\qquad \le 2(r'(x) - r(x)) + 2 + 2\lg\frac{s'(x)/2}{s'(x)} && [s'(w) + s'(z) \le s'(x), \text{ lg is concave}] \\
&\qquad = 2(r'(x) - r(x)) \\
&\qquad \le 3(r'(x) - r(x)) && [r'(x) \ge r(x)]
\end{aligned}$$

Finally, consider a roller-coaster, as shown in Figure 3. Only x, y, and z change rank.

$$\begin{aligned}
&2 + \Phi' - \Phi \\
&\qquad = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) && \text{[only } x, y, z \text{ change rank]} \\
&\qquad \le 2 + r'(x) + r'(z) - 2r(x) && [r'(y) \le r(z) \text{ and } r(x) \ge r(y)] \\
&\qquad = 3(r'(x) - r(x)) + 2 + r(x) + r'(z) - 2r'(x) \\
&\qquad = 3(r'(x) - r(x)) + 2 + \lg\frac{s(x)}{s'(x)} + \lg\frac{s'(z)}{s'(x)} \\
&\qquad \le 3(r'(x) - r(x)) + 2 + 2\lg\frac{s'(x)/2}{s'(x)} && [s(x) + s'(z) \le s'(x), \text{ lg is concave}] \\
&\qquad = 3(r'(x) - r(x)) && \square
\end{aligned}$$

Observe that this argument works for *arbitrary* non-negative vertex weights. By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node x is at most $1 + 3r(\mathit{root}) - 3r(x)$. (The intermediate ranks cancel out in a nice telescoping sum.)

This analysis has several immediate corollaries. The first corollary is that the amortized search time in a splay tree is within a constant factor of the search time in the best possible *static* binary search tree. Thus, if some nodes are accessed more often than others, the standard splay algorithm *automatically* keeps those more frequent nodes closer to the root, at least most of the time.

Static Optimality Theorem. *Suppose each node x is accessed at least $t(x)$ times, and let $T = \sum_x t(x)$. The amortized cost of accessing x is $O(\log T - \log t(x))$.*

Proof: Set $w(x) = t(x)$ for each node x. □

For any nodes x and z, let $\mathit{dist}(x, z)$ denote the *rank distance* between x and y, that is, the number of nodes y such that $\mathit{key}(x) \le \mathit{key}(y) \le \mathit{key}(z)$ or $\mathit{key}(x) \ge \mathit{key}(y) \ge \mathit{key}(z)$. In particular, $\mathit{dist}(x, x) = 1$ for all x.

Static Finger Theorem. *For any fixed node f ('the finger'), the amortized cost of accessing x is $O(\lg \mathit{dist}(f, x))$.*

Proof: Set $w(x) = 1/\mathit{dist}(x, f)^2$ for each node x. Then $s(\mathit{root}) \le \sum_{i=1}^{\infty} 2/i^2 = \pi^2/3 = O(1)$, and $r(x) \ge \lg w(x) = -2\lg \mathit{dist}(f, x)$. □

Here are a few more interesting properties of splay trees, which I'll state without proof.[5] The proofs of these properties (especially the dynamic finger theorem) are considerably more complicated than the amortized analysis presented above.

Working Set Theorem [11]. *The amortized cost of accessing node x is $O(\log D)$, where D is the number of distinct items accessed since the last time x was accessed. (For the first access to x, we set $D = n$.)*

Scanning Theorem [14]. *Splaying all nodes in a splay tree in order, starting from any initial tree, requires $O(n)$ total rotations.*

Dynamic Finger Theorem [7, 6]. *Immediately after accessing node y, the amortized cost of accessing node x is $O(\lg \mathit{dist}(x, y))$.*

*9.8 Splay Tree Conjectures

The *Deque Conjecture* [14] considers the cost of dynamically maintaining two fingers l and r, starting on the left and right ends of the tree. Suppose at each step, we can move one of these two fingers either one step left or one step right; in other words, we are using the splay tree as a doubly-ended queue. Sundar* proved that the total cost of m deque operations on an n-node splay tree is $O((m+n)\alpha(m+n))$ [12]; the Deque Conjecture states that the total cost is actually $O(m+n)$.

The *Traversal Conjecture* [11] states that accessing the nodes in a splay tree, in the order specified by a *preorder* traversal of any other binary tree with the same keys, takes $O(n)$ time. This is generalization of the Scanning Theorem.

The *Unified Conjecture* [10] states that the time to access node x is $O(\lg \min_y(D(y) + d(x, y)))$, where $D(y)$ is the number of *distinct* nodes accessed since the last time y was accessed. This would immediately imply both the Dynamic Finger Theorem, which is about spatial locality, and the Working Set Theorem, which is about temporal locality. Two other structures are known that satisfy the unified bound [4, 10].

Finally, the most important conjecture about splay trees is that they are *dynamically optimal* [11]. Specifically, the cost of any sequence of accesses to a splay tree is conjectured to be at most a constant factor more than the cost of the best possible dynamic binary search tree *that knows the entire access sequence in advance*. To make the rules concrete, we consider binary search trees that can undergo *arbitrary* rotations after a search; the cost of a search is the number of key comparisons plus the number of rotations. We do not require that the rotations be on or even near the search path. This is an extremely strong conjecture! No dynamically optimal binary search tree is known, even in the offline setting. The closest partial result known is a recently-discovered pair of $O(\log \log n)$-competitive structures, *Tango trees* [8] and *multisplay trees* [15].

References

[1] A. Andersson*. Improving partial rebuilding by using simple balance criteria. *Proc. Workshop on Algorithms and Data Structures*, 393–402, 1989. Lecture Notes Comput. Sci. 382, Springer-Verlag.

[2] A. Andersson. General balanced trees. *J. Algorithms* 30:1–28, 1999.

[3] J. L. Bentley and J. B. Saxe*. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms* 1(4):301–358, 1980.

[4] M. Bădiou* and E. D. Demaine. A simplified and dynamic unified structure. *Proc. 6th Latin American Sympos. Theoretical Informatics*, 466–473, 2004. Lecture Notes Comput. Sci. 2976, Springer-Verlag.

[5] This list and the following section are taken almost directly from Erik Demaine's lecture notes [5].

[5] J. Cohen* and E. Demaine. 6.897: Advanced data structures (Spring 2005), Lecture 3, February 8 2005. ⟨http://theory.csail.mit.edu/classes/6.897/spring05/lec.html⟩.

[6] R. Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM J. Comput.* 30(1):44–85, 2000.

[7] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM J. Comput.* 30(1):1–43, 2000.

[8] E. D. Demaine, D. Harmon*, J. Iacono, and M. Pătraşcu**. Dynamic optimality—almost. *Proc. 45th Annu. IEEE Sympos. Foundations Comput. Sci.*, 484–490, 2004.

[9] I. Galperin* and R. R. Rivest. Scapegoat trees. *Proc. 4th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 165–174, 1993.

[10] J. Iacono*. Alternatives to splay trees with $O(\log n)$ worst-case access times. *nth Annu. ACM-SIAM Sympos. Discrete Algorithms*, 516–522, 2001.

[11] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM* 32(3):652–686, 1985.

[12] R. Sundar*. On the Deque conjecture for the splay algorithm. *Combinatorica* 12(1):95–124, 1992.

[13] R. E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics 44. SIAM, 1983.

[14] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica* 5(5):367–378, 1985.

[15] C. C. Wang*, J. Derryberry*, and D. D. Sleator. $O(\log \log n)$-competitive dynamic binary search trees. *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 374–383, 2006.

*Starred authors were PhD students at the time that the cited work was published, except for **Mihai Pătraşcu, who was a junior at MIT.

E pluribus unum (Out of many, one)

— Official motto of the United States of America

John: *Who's your daddy? C'mon, you know who your daddy is! Who's your daddy? D'Argo, tell him who his daddy is!"*
D'Argo: *I'm your daddy.*

— *Farscape*, "Thanks for Sharing" (June 15, 2001)

What rolls down stairs, alone or in pairs, rolls over your neighbor's dog?
What's great for a snack, and fits on your back? It's Log, Log, Log!

It's Log! It's Log! It's big, it's heavy, it's wood!
It's Log! It's Log! It's better than bad, it's good!
— *Ren & Stimpy*, "Stimpy's Big Day/The Big Shot" (August 11, 1991)
lyrics by John Kricfalusi

The thing's hollow - it goes on forever - and - oh my God! - it's full of stars!

— Capt. David Bowman's last words(?)
2001: A Space Odyssey by Arthur C. Clarke (1968)

10 Data Structures for Disjoint Sets

In this lecture, we describe some methods for maintaining a collection of disjoint sets. Each set is represented as a pointer-based data structure, with one node per element. We will refer to the elements as either 'objects' or 'nodes', depending on whether we want to emphasize the set abstraction or the actual data structure. Each set has a unique 'leader' element, which identifies the set. (Since the sets are always disjoint, the same object cannot be the leader of more than one set.) We want to support the following operations.

- MAKESET(x): Create a new set $\{x\}$ containing the single element x. The object x must not appear in any other set in our collection. The leader of the new set is obviously x.
- FIND(x): Find (the leader of) the set containing x.
- UNION(A, B): Replace two sets A and B in our collection with their union $A \cup B$. For example, UNION(A, MAKESET(x)) adds a new element x to an existing set A. The sets A and B are specified by arbitrary elements, so UNION(x, y) has exactly the same behavior as UNION(FIND(x), FIND(y)).

Disjoint set data structures have lots of applications. For instance, Kruskal's minimum spanning tree algorithm relies on such a data structure to maintain the components of the intermediate spanning forest. Another application is maintaining the connected components of a graph as new vertices and edges are added. In both these applications, we can use a disjoint-set data structure, where we maintain a set for each connected component, containing that component's vertices.

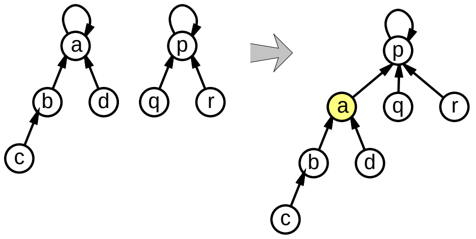
10.1 Reversed Trees

One of the easiest ways to store sets is using trees, in which each node represents a single element of the set. Each node points to another node, called its *parent*, except for the leader of each set, which points to itself and thus is the root of the tree. MAKESET is trivial. FIND traverses parent pointers up to the leader. UNION just redirects the parent pointer of one leader to the other. Unlike most tree data structures, nodes do *not* have pointers down to their children.

```
MAKESET($x$):
    $parent(x) \leftarrow x$
```

```
FIND($x$):
    while $x \neq parent(x)$
        $x \leftarrow parent(x)$
    return $x$
```

```
UNION($x, y$):
    $\overline{x} \leftarrow$ FIND($x$)
    $\overline{y} \leftarrow$ FIND($y$)
    $parent(\overline{y}) \leftarrow \overline{x}$
```



Merging two sets stored as trees. Arrows point to parents. The shaded node has a new parent.

MAKE-SET clearly takes $\Theta(1)$ time, and UNION requires only $O(1)$ time in addition to the two FINDS. The running time of FIND(x) is proportional to the depth of x in the tree. It is not hard to come up with a sequence of operations that results in a tree that is a long chain of nodes, so that FIND takes $\boxed{\Theta(n)\text{ time}}$ in the worst case.

However, there is an easy change we can make to our UNION algorithm, called *union by depth*, so that the trees always have logarithmic depth. Whenever we need to merge two trees, we always make the root of the *shallower* tree a child of the *deeper* one. This requires us to also maintain the depth of each tree, but this is quite easy.

```
MAKESET($x$):
    $parent(x) \leftarrow x$
    $depth(x) \leftarrow 0$
```

```
FIND($x$):
    while $x \neq parent(x)$
        $x \leftarrow parent(x)$
    return $x$
```

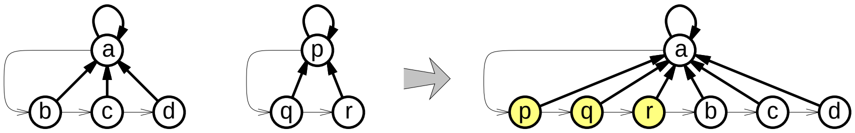
```
UNION($x, y$)
    $\overline{x} \leftarrow$ FIND($x$)
    $\overline{y} \leftarrow$ FIND($y$)
    if $depth(\overline{x}) > depth(\overline{y})$
        $parent(\overline{y}) \leftarrow \overline{x}$
    else
        $parent(\overline{x}) \leftarrow \overline{y}$
        if $depth(\overline{x}) = depth(\overline{y})$
            $depth(\overline{y}) \leftarrow depth(\overline{y}) + 1$
```

With this new rule in place, it's not hard to prove by induction that for any set leader $\overline{x}$, the size of $\overline{x}$'s set is at least $2^{depth(\overline{x})}$. If $depth(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set. For any $d > 0$, when $depth(\overline{x})$ becomes d for the first time, $\overline{x}$ is becoming the leader of the union of two sets, both of whose leaders had depth $d - 1$. By the inductive hypothesis, both component sets had at least 2^{d-1} elements, so the new set has at least 2^d elements. Later UNION operations might ad elements to $\overline{x}$'s set without changing its depth, but that only helps us.

Since there are only n elements altogether, the maximum depth of any set is $\lg n$. We conclude that if we use union by depth, both FIND and UNION run in $\boxed{\Theta(\log n)}$ time in the worst case.

10.2 Shallow Threaded Trees

Alternately, we could just have every object keep a pointer to the leader of its set. Thus, each set is represented by a shallow tree, where the leader is the root and all the other elements are its children. With this representation, MAKESET and FIND are completely trivial. Both operations clearly run in constant time. UNION is a little more difficult, but not much. Our algorithm sets all the leader pointers in one set to point to the leader of the other set. To do this, we need a method to visit every element in a set; we will 'thread' a linked list through each set, starting at the set's leader. The two threads are merged in the UNION algorithm in constant time.



Merging two sets stored as threaded trees.
Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.

```
MAKESET($x$):
    leader($x$) ← $x$
    next($x$) ← $x$
```

```
FIND($x$):
    return leader($x$)
```

```
UNION($x, y$):
    $\overline{x}$ ← FIND($x$)
    $\overline{y}$ ← FIND($y$)

    $y$ ← $\overline{y}$
    leader($y$) ← $\overline{x}$
    while (next($y$) ≠ NULL)
        $y$ ← next($y$)
        leader($y$) ← $\overline{x}$

    next($y$) ← next($\overline{x}$)
    next($\overline{x}$) ← $\overline{y}$
```

The worst-case running time of UNION is a constant times the size of the *larger* set. Thus, if we merge a one-element set with another n-element set, the running time can be $\Theta(n)$. Generalizing this idea, it is quite easy to come up with a sequence of n MAKESET and $n-1$ UNION operations that requires $\Theta(n^2)$ time to create the set $\{1, 2, \ldots, n\}$ from scratch.

```
WORSTCASESEQUENCE($n$):
    MAKESET(1)
    for $i$ ← 2 to $n$
        MAKESET($i$)
        UNION($1, i$)
```

We are being stupid in two different ways here. One is the order of operations in WORSTCASESEQUENCE. Obviously, it would be more efficient to merge the sets in the other order, or to use some sort of divide and conquer approach. Unfortunately, we can't fix this; we don't get to decide how our data structures are used! The other is that we always update the leader pointers in the larger set. To fix this, we add a comparison inside the UNION algorithm to determine which set is smaller. This requires us to maintain the size of each set, but that's easy.

```
MAKEWEIGHTEDSET($x$):
    leader($x$) ← $x$
    next($x$) ← $x$
    size($x$) ← 1
```

```
WEIGHTEDUNION($x, y$)
    $\overline{x}$ ← FIND($x$)
    $\overline{y}$ ← FIND($y$)
    if size($\overline{x}$) > size($\overline{y}$)
        UNION($\overline{x}, \overline{y}$)
        size($\overline{x}$) ← size($\overline{x}$) + size($\overline{y}$)
    else
        UNION($\overline{y}, \overline{x}$)
        size($\overline{x}$) ← size($\overline{x}$) + size($\overline{y}$)
```

The new WEIGHTEDUNION algorithm still takes $\Theta(n)$ time to merge two n-element sets. However, in an amortized sense, this algorithm is much more efficient. Intuitively, before we can merge two large sets, we have to perform a large number of MAKEWEIGHTEDSET operations.

Theorem 1. *A sequence of m MAKEWEIGHTEDSET operations and n WEIGHTEDUNION operations takes $O(m + n \log n)$ time in the worst case.*

Proof: Whenever the leader of an object x is changed by a WEIGHTEDUNION, the size of the set containing x increases by at least a factor of two. By induction, if the leader of x has changed k times, the set containing x has at least 2^k members. After the sequence ends, the largest set contains at most n members. (Why?) Thus, the leader of any object x has changed at most $\lfloor \lg n \rfloor$ times.

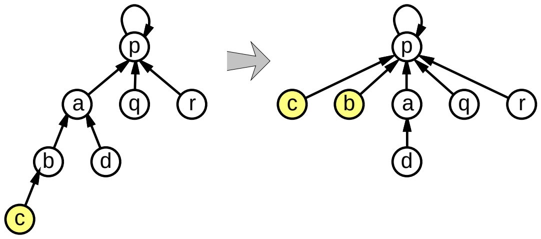
Since each WEIGHTEDUNION reduces the number of sets by one, there are $m - n$ sets at the end of the sequence, and at most n objects are *not* in singleton sets. Since each of the non-singleton objects had $O(\log n)$ leader changes, the total amount of work done in updating the leader pointers is $O(n \log n)$. □

The aggregate method now implies that each WEIGHTEDUNION has $\boxed{\text{amortized cost } O(\log n)}$.

10.3 Path Compression

Using unthreaded tress, FIND takes logarithmic time and everything else is constant; using threaded trees, UNION takes logarithmic amortized time and everything else is constant. A third method allows us to get both of these operations to have *almost* constant running time.

We start with the original unthreaded tree representation, where every object points to a parent. The key observation is that in any FIND operation, once we determine the leader of an object x, we can speed up future FINDs by redirecting x's parent pointer directly to that leader. In fact, we can change the parent pointers of all the ancestors of x all the way up to the root; this is easiest if we use recursion for the initial traversal up the tree. This modification to FIND is called *path compression*.



Path compression during FIND(c). Shaded nodes have a new parent.

```
FIND(x)
    if $x \neq$ parent($x$)
        parent($x$) $\leftarrow$ FIND(parent($x$))
    return parent($x$)
```

If we use path compression, the 'depth' field we used earlier to keep the trees shallow is no longer correct, and correcting it would take way too long. But this information still ensures that FIND runs in $\Theta(\log n)$ time in the worst case, so we'll just give it another name: *rank*.

```
MAKESET(x):
    parent($x$) $\leftarrow x$
    rank($x$) $\leftarrow 0$
```

```
UNION(x, y)
    $\overline{x} \leftarrow$ FIND($x$)
    $\overline{y} \leftarrow$ FIND($y$)
    if rank($\overline{x}$) $>$ rank($\overline{y}$)
        parent($\overline{y}$) $\leftarrow \overline{x}$
    else
        parent($\overline{x}$) $\leftarrow \overline{y}$
        if rank($\overline{x}$) $=$ rank($\overline{y}$)
            rank($\overline{y}$) $\leftarrow$ rank($\overline{y}$) $+ 1$
```

FIND still runs in $O(\log n)$ time in the worst case; path compression increases the cost by only most a constant factor. But we have good reason to suspect that this upper bound is no longer tight. Our new algorithm memoizes the results of each FIND, so if we are asked to FIND the same item twice in a row, the second call returns in constant time. Splay trees used a similar strategy to achieve their optimal amortized cost, but our up-trees have fewer constraints on their structure than binary search trees, so we should get even better performance.

This intuition is exactly correct, but it takes a bit of work to define precisely *how* much better the performance is. As a first approximation, we will prove below that the amortized cost of a FIND operation is bounded by the *iterated logarithm* of n, denoted $\log^* n$, which is the number of times one must take the logarithm of n before the value is less than 1:

$$\lg^* n = \begin{cases} 1 & \text{if } n \le 2, \\ 1 + \lg^*(\lg n) & \text{otherwise.} \end{cases}$$

Our proof relies on several useful properties of ranks, which follow directly from the UNION and FIND algorithms.

- If a node x is not a set leader, then the rank of x is smaller than the rank of its parent.
- Whenever *parent*(x) changes, the new parent has larger rank than the old parent.
- Whenever the leader of x's set changes, the new leader has larger rank than the old leader.
- The size of any set is exponential in the rank of its leader: $\mathit{size}(\overline{x}) \ge 2^{\mathit{rank}(\overline{x})}$. (This is easy to prove by induction, hint, hint.)
- In particular, since there are only n objects, the highest possible rank is $\lfloor \lg n \rfloor$.
- For any integer r, there are at most $n/2^r$ objects of rank r.

Only the last property requires a clever argument to prove. Fix your favorite integer r. Observe that only set leaders can change their rank. Whenever the rank of any set leader $\overline{x}$ changes from $r-1$ to r, mark all the objects in $\overline{x}$'s set. Since leader ranks can only increase over time, each object is marked at most once. There are n objects altogether, and any object with rank r marks at least 2^r objects. It follows that there are at most $n/2^r$ objects with rank r, as claimed.

*10.4 $O(\log^* n)$ Amortized Time

The following analysis of path compression was discovered just a few years ago by Raimund Seidel and Micha Sharir.[1] Previous proofs[2] relied on complicated charging schemes; Seidel and Sharir's analysis relies on a comparatively simple recursive decomposition.

Seidel and Sharir phrase their analysis in terms of two more general operations on set forests. Their more general COMPRESS operation compresses *any* directed path, not just paths that lead to the root. The new SHATTER operation makes every node on a root-to-leaf path into its own parent.

[1] Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM Journal on Computing* 34(3):515–525, 2005.

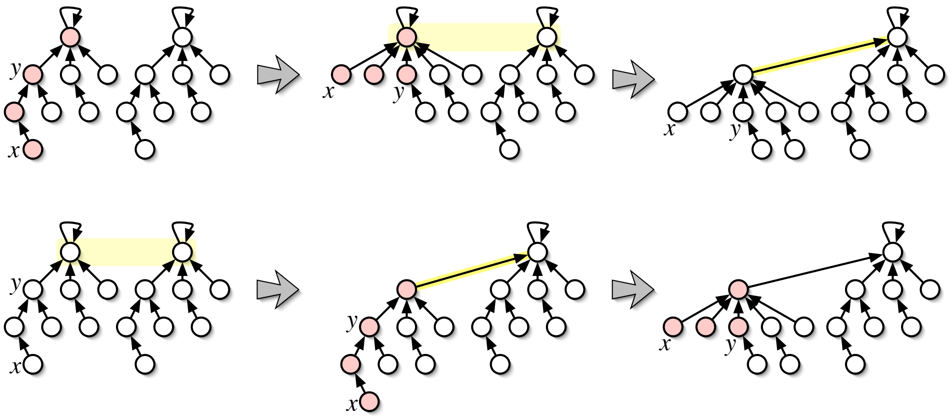
[2] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.* 22:215–225, 1975.

```
COMPRESS$(x, y)$:
⟨⟨$y$ must be an ancestor of $x$⟩⟩
  if $x \neq y$
      COMPRESS$(parent(x), y)$
      $parent(x) \leftarrow parent(y)$
```

```
SHATTER$(x)$:
  if $parent(x) \neq x$
      SHATTER$(parent(x))$
      $parent(x) \leftarrow x$
```

Clearly, the running time of FIND(x) operation is dominated by the running time of COMPRESS(x, y), where y is the leader of the set containing x. This implies that we can prove the upper bound by analyzing an arbitrary sequence of UNION and COMPRESS operations. Moreover, we can assume that the arguments to each UNION operation are set leaders, so that each UNION takes only constant worst-case time.

Finally, since each call to COMPRESS specifies the top node in the path to be compressed, we can reorder the sequence of operations, so that every UNION occurs before any COMPRESS, without changing the number of pointer assignments.



Top row: A COMPRESS followed by a UNION. Bottom row: The same operations in the opposite order.

Each UNION requires only constant time in the worst case, so we only need to analyze the amortized cost of COMPRESS. The running time of COMPRESS is proportional to the number of parent pointer assignments, plus $O(1)$ overhead, so we will phrase our analysis in terms of pointer assignments. Let $\boxed{T(m, n, r)}$ denote the worst case number of pointer assignments in any sequence of at most m COMPRESS operations, executed on a forest of at most n nodes, with maximum rank at most r.

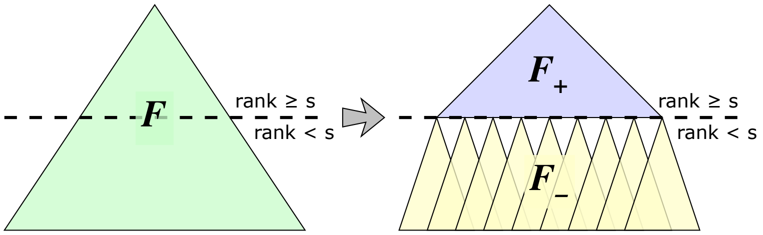
The following trivial upper bound will be the base case for our recursive argument.

Theorem 2. $\boxed{T(m, n, r) \leq nr}$

Proof: Each node can change parents at most r times, because each new parent has higher rank than the previous parent. □

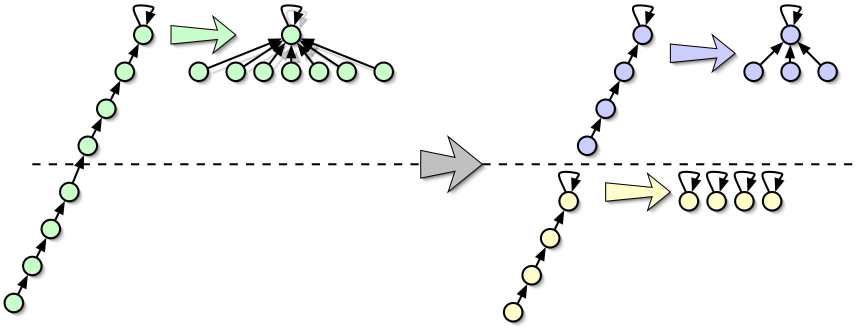
Fix a forest F of n nodes with maximum rank r, and a sequence C of m COMPRESS operations on F, and let $T(F, C)$ denote the total number of pointer assignments executed by this sequence.

Let s be an arbitrary positive rank. Partition F into two sub-forests: a 'low' forest F_- containing all nodes with rank at most s, and a 'high' forest F_+ containing all nodes with rank greater than s. Since ranks increase as we follow parent pointers, every ancestor of a high node is another high node. Let n_- and n_+ denote the number of nodes in F_- and F_+, respectively. Finally, let m_+ denote the number of COMPRESS operations that involve any node in F_+, and let $m_- = m - m_+$.



Splitting the forest F (in this case, a single tree) into sub-forests F_+ and F_- at rank s.

Any sequence of COMPRESS operations on F can be decomposed into a sequence of COMPRESS operations on F_+, plus a sequence of COMPRESS and SHATTER operations on F_-, with the same total cost. This requires only one small modification to the code: We forbid any low node from having a high parent. Specifically, if x is a low node and y is a high node, we replace any assignment *parent*$(x) \leftarrow y$ with *parent*$(x) \leftarrow x$.



A COMPRESS operation in F splits into a COMPRESS operation in F_+ and a SHATTER operation in F_-

This modification is equivalent to the following reduction:

```
COMPRESS($x, y, F$):    ⟨⟨$y$ is an ancestor of $x$⟩⟩
  if rank$(x) > r$
        COMPRESS$(x, y, F_+)$    ⟨⟨in $C_+$⟩⟩
  else if rank$(y) \le r$
        COMPRESS$(x, y, F_-)$    ⟨⟨in $C_-$⟩⟩
  else
        $z \leftarrow$ highest ancestor of $x$ in $F$ with rank at most $r$
        COMPRESS$(parent_F(z), y, F_+)$    ⟨⟨in $C_+$⟩⟩
        SHATTER$(x, z, F_-)$
        $parent(z) \leftarrow z$    $(*)$
```

The pointer assignment in the last line looks redundant, but it is actually necessary for the analysis. Each execution of line $(*)$ mirrors an assignment of the form *parent*$(x) \leftarrow y$, where x is a low node, y is a high node, and the previous parent of x was a high node. Each of these 'redundant' assignments happens immediately after a COMPRESS in the top forest, so we perform at most m_+ redundant assignments.

Each node x is touched by at most one SHATTER operation, so the total number of pointer reassignments in all the SHATTER operations is at most n.

Thus, by partitioning the forest F into F_+ and F_-, we have also partitioned the sequence C of COMPRESS operations into subsequences C_+ and C_-, with respective lengths m_+ and m_-, such that the following inequality holds:

$$\boxed{T(F, C) \le T(F_+, C_+) + T(F_-, C_-) + m_+ + n}.$$

Since there are only $n/2^i$ nodes of any rank i, we have $n_+ \le \sum_{i>s} n/2^i = n/2^s$. The number of different ranks in F_+ is $r - s < r$. Thus, Theorem 2 implies the upper bound

$$T(F_+, C_+) < rn/2^s.$$

Let us fix $\boxed{s = \lg r}$, so that $T(F_+, C_+) \le n$. We can now simplify our earlier recurrence to

$$T(F, C) \le T(F_-, C_-) + m_+ + 2n,$$

or equivalently,

$$T(F, C) - m \le T(F_-, C_-) - m_- + 2n.$$

Since this argument applies to *any* forest F and *any* sequence C, we have just proved that

$$\boxed{T'(m, n, r) \le T'(m, n, \lfloor \lg r \rfloor) + 2n},$$

where $T'(m, n, r) = T(m, n, r) - m$. The solution to this recurrence is $T'(n, m, r) \le 2n \lg^* r$. Voila!

Theorem 3. $\boxed{T(m, n, r) \le m + 2n \lg^* r}$

*10.5 Turning the Crank

There is one place in the preceding analysis where we have significant room for improvement. Recall that we bounded the total cost of the operations on F_+ using the trivial upper bound from Theorem 2. But we just proved a better upper bound in Theorem 3! We can apply precisely the same strategy, using Theorem 3 instead of Theorem 2, to improve the bound even more.

Suppose we fix $s = \lg^* r$, so that $n_+ = n/2^{\lg^* r}$. Theorem 3 implies that

$$T(F_+, C_+) \le m_+ + 2n \frac{\lg^* r}{2^{\lg^* r}} \le m_+ + 2n.$$

This implies the recurrence

$$T(F, C) \le T(F_-, C_-) + 2m_+ + 3n,$$

which in turn implies that

$$T''(m, n, r) \le T''(m, n, \lg^* r) + 3n,$$

where $T''(m, n, r) = T(m, n, r) - 2m$. The solution to this equation is $\boxed{T(m, n, r) \le 2m + 3n \lg^{**} r}$, where $\lg^{**} r$ is the *iterated* iterated logarithm of r:

$$\lg^{**} r = \begin{cases} 1 & \text{if } r \le 2, \\ 1 + \lg^{**}(\lg^* r) & \text{otherwise.} \end{cases}$$

Naturally we can apply the same improvement strategy again, and again, as many times as we like, each time producing a tighter upper bound. Applying the reduction c times, for any positive integer c, gives us

$$\boxed{T(m, n, r) \le cm + (c+1) n \lg^{*^c} r}$$

where

$$\lg^{*^c} r = \begin{cases} \lg r & \text{if } c = 0, \\ 1 & \text{if } r \le 2, \\ 1 + \lg^{*^c}(\lg^{*^{c-1}} r) & \text{otherwise.} \end{cases}$$

Each time we 'turn the crank', the dependence on m increases, while the dependence on n and r decreases. For sufficiently large values of c, the cm term dominates the time bound, and further iterations only make things worse. The point of diminishing returns can be estimated by *the minimum number of stars* such that $\lg^{**\cdots*} r$ is smaller than a constant:

$$\alpha(r) = \min\left\{c \geq 1 \;\middle|\; \lg^{*^c} n \leq 3\right\}.$$

(The threshold value 3 is used here because $\lg^{*^c} 5 \geq 2$ for all c.) By setting $c = \alpha(r)$, we obtain our final upper bound.

Theorem 4. $\boxed{T(m,n,r) \leq m\alpha(r) + 3n(\alpha(r)+1)}$

We can assume without loss of generality that $m \geq n$ by ignoring any singleton sets, so this upper bound can be further simplified to $T(m,n,r) = O(m\alpha(r)) = O(m\alpha(n))$. It follows that if we use union by rank, FIND with path compression runs in $\boxed{O(\alpha(n)) \text{ amortized time}}$.

Even this upper bound is somewhat conservative if m is larger than n. A closer estimate is given by the function

$$\alpha(m,n) = \min\left\{c \geq 1 \;\middle|\; \log^{*^c}(\lg n) \leq m/n\right\}.$$

It's not hard to prove that if $m = \Theta(n)$, then $\alpha(m,n) = \Theta(\alpha(n))$. On the other hand, $m \geq n \lg^{*****} n$ (for any constant number of stars) already implies that $\alpha(m,n) = O(1)$. So even if the number of FIND operations is only *slightly* larger than the number of nodes, the amortized cost of each FIND is *constant*.

$O(\alpha(m,n))$ is actually a *tight* upper bound for the amortized cost of path compression; there are no more tricks that will improve the analysis further. More surprisingly, this is the best amortized bound we obtain for *any* pointer-based data structure for maintaining disjoint sets; the amortized cost of *every* FIND algorithm is at least $\Omega(\alpha(m,n))$. The proof of the matching lower bound is, unfortunately, far beyond the scope of this class.[3]

10.6 The Ackermann Function and its Inverse

The iterated logarithms that fell out of our analysis of path compression are the inverses of a hierarchy of recursive functions defined by Wilhelm Ackermann in 1928.[4]

$$\boxed{2 \uparrow^c n = \begin{cases} 2 & \text{if } n = 1 \\ 2n & \text{if } c = 0 \\ 2 \uparrow^{c-1} (2 \uparrow^c (n-1)) & \text{otherwise} \end{cases}}$$

For each fixed c, the function $2 \uparrow^c n$ is monotonically increasing in n, and these functions grow *incredibly* faster as the index c increases. $2 \uparrow n$ is the familiar power function 2^n. $2 \uparrow\uparrow n$ is the *tower* function $\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}^{n}$; this function is also sometimes called *tetration*. John Conway named $2 \uparrow\uparrow\uparrow n$ the *wower* function: $2 \uparrow\uparrow\uparrow n = \underbrace{2 \uparrow\uparrow 2 \uparrow\uparrow \cdots \uparrow\uparrow 2}_{n}$. And so on, *et cetera, ad infinitum*.

[3] Robert E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Comput. Syst. Sci.* 19:110–127, 1979.

[4] Actually, Ackermann didn't define his functions this way—I'm actually describing a slightly cleaner hierarchy defined 35 years later by R. Creighton Buck—but the exact details of the definition are surprisingly irrelevant! The mnemonic up-arrow notation for these functions was introduced by Don Knuth in the 1970s.

$2\uparrow^c n$	$n=1$	2	$n=3$	$n=4$	$n=5$
$2n$	2	4	6	8	10
$2\uparrow n$	2	4	8	16	32
$2\uparrow\uparrow n$	2	4	16	65536	2^{65536}
$2\uparrow\uparrow\uparrow n$	2	4	65536	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536$	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}_{\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536}$
$2\uparrow\uparrow\uparrow\uparrow n$	2	4	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536$	$\left.2^{2^{\cdots 2}}\big\}^{2^{\cdots 2}}\big\}^{\cdot^{\cdot^{\cdot^{2^{\cdots 2}\}65536}}}}\right\}_{\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536}$	⟨⟨*Yeah, right.*⟩⟩
$2\uparrow\uparrow\uparrow\uparrow\uparrow n$	2	4	$\left.2^{2^{\cdots 2}}\big\}^{2^{\cdots 2}}\big\}^{\cdot^{\cdot^{\cdot^{2^{\cdots 2}\}65536}}}}\right\}_{\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536}$	⟨⟨*Very funny.*⟩⟩	⟨⟨*Argh! My eyes!*⟩⟩

Small (!!) values of Ackermann's functions.

For any fixed c, the function $\log^{*^c} n$ is the inverse of the function $2\uparrow^{c+1} n$, the $(c+1)$th row in the Ackerman hierarchy. Thus, for any remotely reasonable values of n, say $n \le 2^{256}$, we have $\log^* n \le 5$, $\log^{**} n \le 4$, and $\log^{*^c} n \le 3$ for any $c \ge 3$.

The function $\alpha(n)$ is usually called the *inverse Ackerman function*.[5] Our earlier definition is equivalent to $\alpha(n) = \min\{c \ge 1 \mid 2\uparrow^{c+2} 3 \ge n\}$; in other words, $\alpha(n)+2$ is the inverse of the third column in the Ackermann hierarchy. The function $\alpha(n)$ grows *much* more slowly than $\log^{*^c} n$ for any fixed c; we have $\alpha(n) \le 3$ for all even *remotely imaginable* values of n. Nevertheless, the function $\alpha(n)$ is eventually larger than any constant, so it is *not* $O(1)$.

The inverse Ackerman function is the smallest super-constant function that appears in algorithm analysis. Of course, one can define arbitrarily smaller functions, starting with the *iterated* inverse Ackerman function $\alpha^*(n)$, but (*pace* Hofstadter) those functions don't seem to appear in running times of algorithms.[6]

[5] Strictly speaking, the name 'inverse Ackerman function' is inaccurate. One good formal definition of the true inverse Ackerman function is $\tilde{\alpha}(n) = \min\left\{c \ge 1 \;\middle|\; \lg^{*^c} n \le c\right\} = \min\left\{c \ge 1 \;\middle|\; 2\uparrow^{c+2} c \ge n\right\}$. However, it's not hard to prove that $\tilde{\alpha}(n) \le \alpha(n) \le \tilde{\alpha}(n)+1$ for all sufficiently large n, so the inaccuracy is completely forgivable. As I said in the previous footnote, the exact details of the definition are surprisingly irrelevant!

[6] However, some bizarre functions that grow much more slowly than any iterated logarithm but much faster than $\alpha(n)$, such as $2^{\alpha(n)}$ and $\alpha(n)^{\alpha(n)}$, show up in surprising places. Google for 'Davenport-Schinzel sequence'!

> *A little and a little, collected together, become a great deal; the heap in the barn consists of single grains, and drop and drop makes an inundation.*
>
> — Saadi (1184–1291)

> *The trees that are slow to grow bear the best fruit.*
>
> — Molière (1622–1673)

> *Promote yourself but do not demote another.*
>
> — Rabbi Israel Salanter (1810–1883)

> *Fall is my favorite season in Los Angeles, watching the birds change color and fall from the trees.*
>
> — David Letterman

B Fibonacci Heaps

B.1 Mergeable Heaps

A *mergeable heap* is a data structure that stores a collection of *keys*[1] and supports the following operations.

- **INSERT:** Insert a new key into a heap. This operation can also be used to create a new heap containing just one key.
- **FINDMIN:** Return the smallest key in a heap.
- **DELETEMIN:** Remove the smallest key from a heap.
- **MERGE:** Merge two heaps into one. The new heap contains all the keys that used to be in the old heaps, and the old heaps are (possibly) destroyed.

If we never had to use DELETEMIN, mergeable heaps would be completely trivial. Each "heap" just stores to maintain the single record (if any) with the smallest key. INSERTs and MERGEs require only one comparison to decide which record to keep, so they take constant time. FINDMIN obviously takes constant time as well.

If we need DELETEMIN, but we don't care how long it takes, we can still implement mergeable heaps so that INSERTs, MERGEs, and FINDMINs take constant time. We store the records in a circular doubly-linked list, and keep a pointer to the minimum key. Now deleting the minimum key takes $\Theta(n)$ time, since we have to scan the linked list to find the new smallest key.

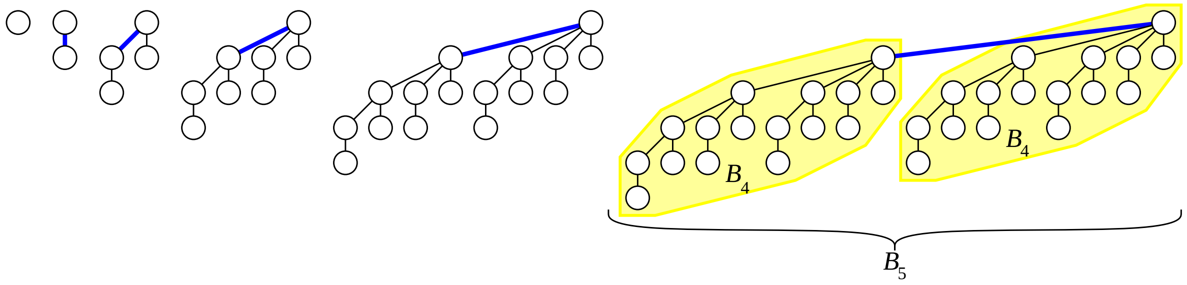
In this lecture, I'll describe a data structure called a *Fibonacci heap* that supports INSERTs, MERGEs, and FINDMINs in constant time, even in the worst case, and also handles DELETEMIN in $O(\log n)$ *amortized* time. That means that any sequence of n INSERTs, m MERGEs, f FINDMINs, and d DELETEMINs takes $O(n + m + f + d \log n)$ time.

B.2 Binomial Trees and Fibonacci Heaps

A *Fibonacci heap* is a circular doubly linked list, with a pointer to the minimum key, but the elements of the list are not single keys. Instead, we collect keys together into structures called *binomial heaps*. Binomial heaps are trees that satisfy the *heap property*—every node has a smaller key than its children—and have the following special recursive structure.

[1] In the earlier lecture on treaps, I called these keys *priorities* to distinguish them from search keys.

A *kth order binomial tree*, which I'll abbreviate B_k, is defined recursively. B_0 is a single node. For all $k > 0$, B_k consists of two copies of B_{k-1} that have been *linked* together, meaning that the root of one B_{k-1} has become a new child of the other root.

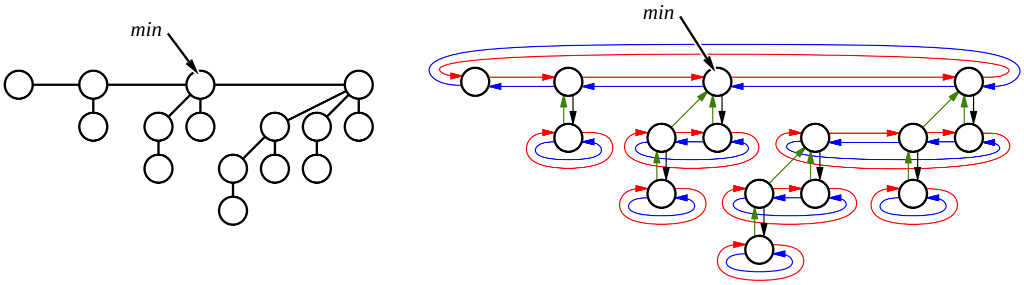


Binomial trees of order 0 through 5.

Binomial trees have several useful properties, which are easy to prove by induction (hint, hint).

- The root of B_k has degree k.
- The children of the root of B_k are the roots of $B_0, B_1, \ldots, B_{k-1}$.
- B_k has height k.
- B_k has 2^k nodes.
- B_k can be obtained from B_{k-1} by adding a new child to every node.
- B_k has $\binom{k}{d}$ nodes at depth d, for all $0 \leq d \leq k$.
- B_k has 2^{k-h-1} nodes with height h, for all $0 \leq h < k$, and one node (the root) with height k.

Although we normally don't care in this class about the low-level details of data structures, we need to be specific about how Fibonacci heaps are actually implemented, so that we can be sure that certain operations can be performed quickly. Every node in a Fibonacci heap points to four other nodes: its parent, its 'next' sibling, its 'previous' sibling, and one of its children. The sibling pointers are used to join the roots together into a circular doubly-linked *root list*. In each binomial tree, the children of each node are also joined into a circular doubly-linked list using the sibling pointers.



A high-level view and a detailed view of the same Fibonacci heap. Null pointers are omitted for clarity.

With this representation, we can add or remove nodes from the root list, merge two root lists together, link one two binomial tree to another, or merge a node's list of children with the root list, in constant time, and we can visit every node in the root list in constant time per node. Having established that these primitive operations can be performed quickly, we never again need to think about the low-level representation details.

B.3 Operations on Fibonacci Heaps

The Insert, Merge, and FindMin algorithms for Fibonacci heaps are exactly like the corresponding algorithms for linked lists. Since we maintain a pointer to the minimum key, FindMin is trivial. To insert a new key, we add a single node (which we should think of as a B_0) to the root list and (if necessary) update the pointer to the minimum key. To merge two Fibonacci heaps, we just merge the two root lists and keep the pointer to the smaller of the two minimum keys. Clearly, all three operations take $O(1)$ time.

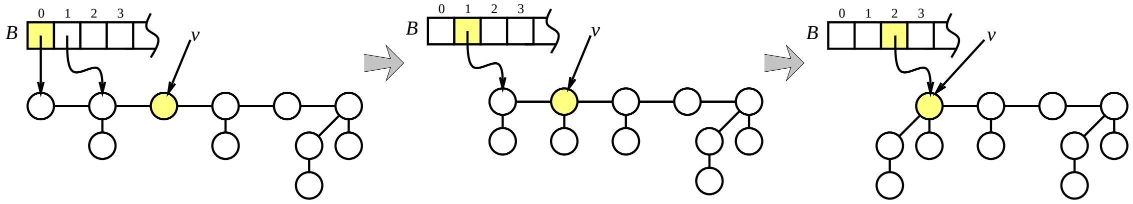
Deleting the minimum key is a little more complicated. First, we remove the minimum key from the root list and splice its children into the root list. Except for updating the parent pointers, this takes $O(1)$ time. Then we scan through the root list to find the new smallest key and update the parent pointers of the new roots. This scan could take $\Theta(n)$ time in the worst case. To bring down the *amortized* deletion time, we apply a Cleanup algorithm, which links pairs of equal-size binomial heaps until there is only one binomial heap of any particular size.

Let me describe the Cleanup algorithm in more detail, so we can analyze its running time. The following algorithm maintains a global array $B[1..\lfloor \lg n \rfloor]$, where $B[i]$ is a pointer to some previously-visited binomial heap of order i, or Null if there is no such binomial heap. Notice that Cleanup simultaneously resets the parent pointers of all the new roots and updates the pointer to the minimum key. I've split off the part of the algorithm that merges binomial heaps of the same order into a separate subroutine MergeDupes.

```
Cleanup:
  $newmin \leftarrow$ some node in the root list
  for $i \leftarrow 0$ to $\lfloor \lg n \rfloor$
      $B[i] \leftarrow$ Null

  for all nodes $v$ in the root list
      $parent(v) \leftarrow$ Null        $(\star)$
      if $key(newmin) > key(v)$
          $newmin \leftarrow v$
      MergeDupes($v$)
```

```
MergeDupes($v$):
  $w \leftarrow B[\deg(v)]$
  while $w \neq$ Null
      $B[\deg(v)] \leftarrow$ Null
      if $key(v) \leq key(w)$
          swap $v \leftrightarrows w$
      remove $w$ from the root list        $(\star\star)$
      link $w$ to $v$
      $w \leftarrow B[\deg(v)]$
  $B[\deg(v)] \leftarrow v$
```



MergeDupes(v), ensuring that no earlier root has the same degree as v.

Notices that MergeDupes is careful to merge heaps so that the heap property is maintained—the heap whose root has the larger key becomes a new child of the heap whose root has the smaller key. This is handled by swapping v and w if their keys are in the wrong order.

The running time of Cleanup is $O(r')$, where r' is the length of the root list just before Cleanup is called. The easiest way to see this is to count the number of times the two starred lines can be executed: line $(\star)$ is executed once for every node v on the root list, and line $(\star\star)$ is executed *at most* once for every node w on the root list. Since DeleteMin does only a constant amount of work before calling Cleanup, $\boxed{\text{the running time of DeleteMin is } O(r') = O(r + \deg(\min))}$ where r is the number of roots before DeleteMin begins, and $\min$ is the node deleted.

Although $\deg(\min)$ is at most $\lg n$, we can still have $r = \Theta(n)$ (for example, if nothing has been deleted yet), so the worst-case time for a DeleteMin is $\Theta(n)$. After a DeleteMin, the root list has length $O(\log n)$, since all the binomial heaps have unique orders and the largest has order at most $\lfloor \lg n \rfloor$.

B.4 Amortized Analysis of DeleteMin

To bound the amortized cost, observe that each insertion increments r. If we charge a constant 'cleanup tax' for each insertion, and use the collected tax to pay for the Cleanup algorithm, the unpaid cost of a DeleteMin is only $O(\deg(\min)) = O(\log n)$.

More formally, define the *potential* of the Fibonacci heap to be the number of roots. Recall that the amortized time of an operation can be defined as its actual running time plus the increase in potential, provided the potential is initially zero (it is) and we never have negative potential (we never do). Let r be the number of roots before a DeleteMin, and let r'' denote the number of roots afterwards. The actual cost of DeleteMin is $r + \deg(\mathit{min})$, and the number of roots increases by $r'' - r$, so the amortized cost is $r'' + \deg(\min)$. Since $r'' = O(\log n)$ and the degree of any node is $O(\log n)$, $\boxed{\text{the amortized cost of DeleteMin is } O(\log n).}$

Each Insert adds only one root, so its amortized cost is still constant. A Merge actually doesn't change the number of roots, since the new Fibonacci heap has all the roots from its constituents and no others, so its amortized cost is also constant.

B.5 Decreasing Keys

In some applications of heaps, we also need the ability to delete an arbitrary node. The usual way to do this is to decrease the node's key to $-\infty$, and then use DeleteMin. Here I'll describe how to decrease the key of a node in a Fibonacci heap; the algorithm will take $O(\log n)$ time in the worst case, but the amortized time will be only $O(1)$.

Our algorithm for decreasing the key at a node v follows two simple rules.

1. Promote v up to the root list. (This moves the whole subtree rooted at v.)

2. As soon as two children of any node w have been promoted, immediately promote w.

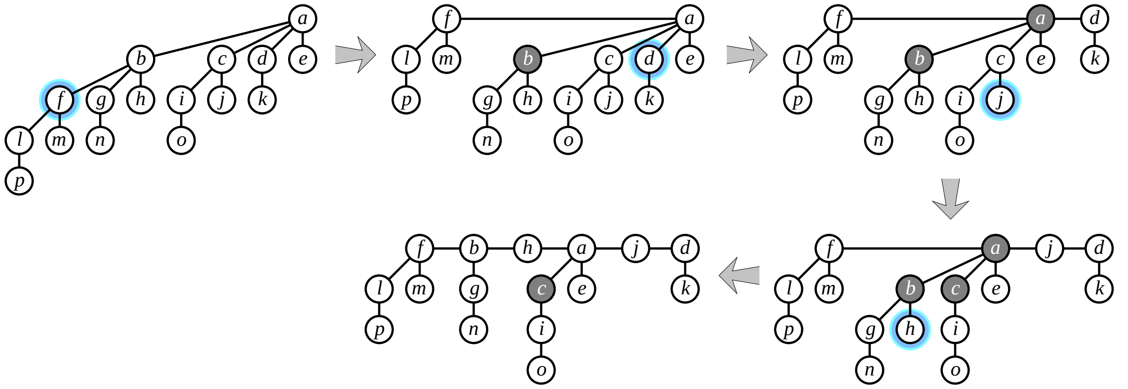
In order to enforce the second rule, we now *mark* certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well. Whenever we promote a marked node, we unmark it; this is the*only* way to unmark a node. (Specifically, splicing nodes into the root list during a DeleteMin is not considered a promotion.)

Here's a more formal description of the algorithm. The input is a pointer to a node v and the new value k for its key.

```
DecreaseKey($v, k$):
  $key(v) \leftarrow k$
  update the pointer to the smallest key
  Promote($v$)
```

```
Promote($v$):
  unmark $v$
  if $parent(v) \neq$ Null
      remove $v$ from $parent(v)$'s list of children
      insert $v$ into the root list
      if $parent(v)$ is marked
          Promote($parent(v)$)
      else
          mark $parent(v)$
```

The PROMOTE algorithm calls itself recursively, resulting in a 'cascading promotion'. Each consecutive marked ancestor of v is promoted to the root list and unmarked, otherwise unchanged. The lowest unmarked ancestor is then marked, since one of its children has been promoted.



Decreasing the keys of four nodes: first f, then d, then j, and finally h. Dark nodes are marked. DECREASEKEY(h) causes nodes b and a to be recursively promoted.

The time to decrease the key of a node v is $O(1+\#\text{consecutive marked ancestors of } v)$. Binomial heaps have logarithmic depth, so if we still had only full binomial heaps, the running time would be $O(\log n)$. Unfortunately, promoting nodes destroys the nice binomial tree structure; our trees no longer have logarithmic depth! In fact, $\boxed{\text{DECREASEKEY runs in } \Theta(n) \text{ time in the worst case.}}$

To compute the amortized cost of DECREASEKEY, we'll use the potential method, just as we did for DELETEMIN. We need to find a potential function Φ that goes up a little whenever we do a little work, and goes down a lot whenever we do a lot of work. DECREASEKEY unmarks several marked ancestors and possibly also marks one node. So *the number of marked nodes* might be an appropriate potential function here. Whenever we do a little bit of work, the number of marks goes up by at most one; whenever we do a lot of work, the number of marks goes down a lot.

More precisely, let m and m' be the number of marked nodes before and after a DECREASEKEY operation. The actual time (ignoring constant factors) is

$$t = 1 + \#\text{consecutive marked ancestors of } v$$

and if we set $\Phi = m$, the increase in potential is

$$m' - m \leq 1 - \#\text{consecutive marked ancestors of } v.$$

Since $t + \Delta\Phi \leq 2$, $\boxed{\text{the amortized cost of DECREASEKEY is } O(1)}$.

B.6 Bounding the Degree

But now we have a problem with our earlier analysis of DELETEMIN. The amortized time for a DELETEMIN is still $O(r + \deg(\min))$. To show that this equaled $O(\log n)$, we used the fact that the maximum degree of any node is $O(\log n)$, which implies that after a CLEANUP the number of roots is $O(\log n)$. But now that we don't have complete binomial heaps, this 'fact' is no longer obvious!

So let's prove it. For any node v, let $|v|$ denote the number of nodes in the subtree of v, including v itself. Our proof uses the following lemma, which *finally* tells us why these things are called Fibonacci heaps.

Lemma 1. *For any node v in a Fibonacci heap, $|v| \geq F_{\deg(v)+2}$.*

Proof: Label the children of v in the chronological order in which they were linked to v. Consider the situation just before the ith oldest child w_i was linked to v. At that time, v had at least $i-1$ children (possibly more). Since CLEANUP only links trees with the same degree, we had $\deg(w_i) = \deg(v) \geq i-1$. Since that time, at most one child of w_i has been promoted away; otherwise, w_i would have been promoted to the root list by now. So currently we have $\deg(w_i) \geq i-2$.

We also quickly observe that $\deg(w_i) \geq 0$. (Duh.)

Let s_d be the minimum possible size of a tree with degree d in any Fibonacci heap. Clearly $s_0 = 1$; for notational convenience, let $s_{-1} = 1$ also. By our earlier argument, the ith oldest child of the root has degree at least $\max\{0, i-2\}$, and thus has size at least $\max\{1, s_{i-2}\} = s_{i-2}$. Thus, we have the following recurrence:

$$s_d \geq 1 + \sum_{i=1}^{d} s_{i-2}$$

If we assume inductively that $s_i \geq F_{i+2}$ for all $-1 \leq i < d$ (with the easy base cases $s_{-1} = F_1$ and $s_0 = F_2$), we have

$$s_d \geq 1 + \sum_{i=1}^{d} F_i = F_{d+2}.$$

(The last step was a practice problem in Homework 0.) By definition, $|v| \geq s_{\deg(v)}$. □

You can easily show (using either induction or the annihilator method) that $F_{k+2} > \phi^k$ where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. Thus, Lemma 1 implies that

$$\deg(v) \leq \log_\phi |v| = O(\log |v|).$$

Thus, since the size of any subtree in an n-node Fibonacci heap is obviously at most n, the degree of any node is $O(\log n)$, which is exactly what we wanted. Our earlier analysis is still good.

B.7 Analyzing Everything Together

Unfortunately, our analyses of DELETEMIN and DECREASEKEY used two different potential functions. Unless we can find a *single* potential function that works for *both* operations, we can't claim both amortized time bounds simultaneously. So we need to find a potential function Φ that goes up a little during a cheap DELETEMIN or a cheap DECREASEKEY, and goes down a lot during an expensive DELETEMIN or an expensive DECREASEKEY.

Let's look a little more carefully at the cost of each Fibonacci heap operation, and its effect on both the number of roots and the number of marked nodes, the things we used as out earlier potential functions. Let r and m be the numbers of roots and marks before each operation, and let r' and m' be the numbers of roots and marks after the operation.

operation	actual cost	$r' - r$	$m' - m$
INSERT	1	1	0
MERGE	1	0	0
DELETEMIN	$r + \deg(min)$	$r' - r$	0
DECREASEKEY	$1 + m - m'$	$1 + m - m'$	$m' - m$

In particular, notice that promoting a node in DECREASEKEY requires constant time and increases the number of roots by one, and that we promote (at most) one unmarked node.

If we guess that the correct potential function is a linear combination of our old potential functions r and m and play around with various possibilities for the coefficients, we will eventually stumble across the correct answer:

$$\boxed{\Phi = r + 2m}$$

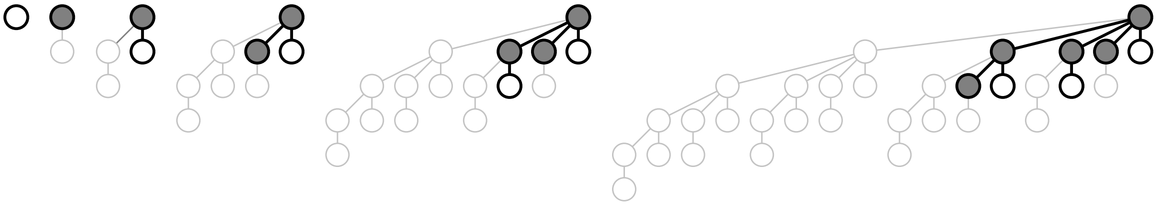
To see that this potential function gives us good amortized bounds for every Fibonacci heap operation, let's add two more columns to our table.

operation	actual cost	$r' - r$	$m' - m$	$\Phi' - \Phi$	amortized cost
INSERT	1	1	0	1	2
MERGE	1	0	0	0	1
DELETEMIN	$r + \deg(\min)$	$r' - r$	0	$r' - r$	$r' + \deg(\min)$
DECREASEKEY	$1 + m - m'$	$1 + m - m'$	$m' - m$	$1 + m' - m$	2

Since Lemma 1 implies that $r' + \deg(\min) = O(\log n)$, we're finally done! (Whew!)

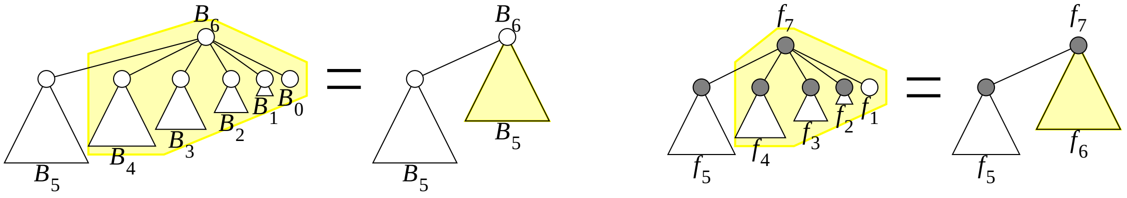
B.8 Fibonacci Trees

To give you a little more intuition about how Fibonacci heaps behave, let's look at a worst-case construction for Lemma 1. Suppose we want to remove as many nodes as possible from a binomial heap of order k, by promoting various nodes to the root list, but without causing any cascading promotions. The most damage we can do is to promote the largest subtree of every node. Call the result a *Fibonacci tree* of order $k+1$, and denote it f_{k+1}. As a base case, let f_1 be the tree with one (unmarked) node, that is, $f_1 = B_0$. The reason for shifting the index should be obvious after a few seconds.



Fibonacci trees of order 1 through 6. Light nodes have been promoted away; dark nodes are marked.

Recall that the root of a binomial tree B_k has k children, which are roots of $B_0, B_1, \ldots, B_{k-1}$. To convert B_k to f_{k+1}, we promote the root of B_{k-1}, and recursively convert each of the other subtrees B_i to f_{i+1}. The root of the resulting tree f_{k+1} has degree $k-1$, and the children are the roots of smaller Fibonacci trees $f_1, f_2, \ldots, f_{k-1}$. We can also consider B_k as two copies of B_{k-1} linked together. It's quite easy to show that an order-k Fibonacci tree consists of an order $k-2$ Fibonacci tree linked to an order $k-1$ Fibonacci tree. (See the picture below.)



Comparing the recursive structures of B_6 and f_7.

Since f_1 and f_2 both have exactly one node, the number of nodes in an order-k Fibonacci tree is exactly the kth Fibonacci number! (That's why we changed in the index.) Like binomial trees, Fibonacci trees have lots of other nice properties that easy to prove by induction (hint, hint):

- The root of f_k has degree $k-2$.
- f_k can be obtained from f_{k-1} by adding a new unmarked child to every marked node and then marking all the old unmarked nodes.
- f_k has height $\lceil k/2 \rceil - 1$.
- f_k has F_{k-2} unmarked nodes, F_{k-1} marked nodes, and thus F_k nodes altogether.
- f_k has $\binom{k-d-2}{d-1}$ unmarked nodes, $\binom{k-d-2}{d}$ marked nodes, and $\binom{k-d-1}{d}$ total nodes at depth d, for all $0 \le d \le \lfloor k/2 \rfloor - 1$.
- f_k has F_{k-2h-1} nodes with height h, for all $0 \le h \le \lfloor k/2 \rfloor - 1$, and one node (the root) with height $\lceil k/2 \rceil - 1$.

I stopped covering Fibonacci heaps in my undergraduate algorithms class a few years ago, even though they are a great illustration of data structure design principles and amortized analysis. My main reason for skipping them is that the algorithms are relatively complicated, which both hinders understanding and limits any practical advantages over regular binary heaps. (The popular algorithms textbook CLRS dismisses Fibonacci heaps as "predominantly of theoretical interest" because of programming complexity and large constant factors in its running time.)

Nevertheless, students interested in data structures are strongly advised to become familiar with binomial and Fibonacci heaps, since they share key structural properties with other data structures (such as union-find trees and Bentley-Saxe dynamization) and illustrate important data structure techniques (like lazy rebuilding). Also, Fibonacci heaps (and more recent extensions like Chazelle's soft heaps) are key ingredients in the fastest algorithms known for some problems.

Obie looked at the seein' eye dog. Then at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one... and then he looked at the seein' eye dog. And then at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one and began to cry.

Because Obie came to the realization that it was a typical case of American blind justice, and there wasn't nothin' he could do about it, and the judge wasn't gonna look at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one explainin' what each one was, to be used as evidence against us.

And we was fined fifty dollars and had to pick up the garbage. In the snow.

But that's not what I'm here to tell you about.

— Arlo Guthrie, "Alice's Restaurant" (1966)

First I shake the whole Apple tree, that the ripest might fall. Then I climb the tree and shake each limb, and then each branch and then each twig, and then I look under each leaf.

— Martin Luther

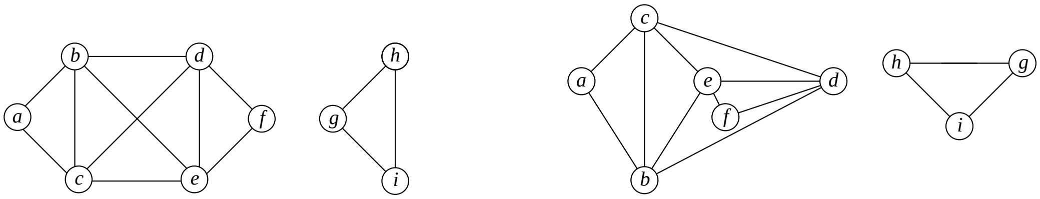
11 Basic Graph Properties

11.1 Definitions

A *graph* G is a pair of sets (V, E). V is a set of arbitrary objects that we call *vertices*[1] or *nodes*. E is a set of vertex pairs, which we call *edges* or occasionally *arcs*. In an *undirected* graph, the edges are unordered pairs, or just sets of two vertices. In a *directed* graph, the edges are ordered pairs of vertices. We will only be concerned with *simple* graphs, where there is no edge from a vertex to itself and there is at most one edge from any vertex to any other.

Following standard (but admittedly confusing) practice, I'll also use V to denote the *number* of vertices in a graph, and E to denote the *number* of edges. Thus, in an undirected graph, we have $0 \le E \le \binom{V}{2}$, and in a directed graph, $0 \le E \le V(V-1)$.

We usually visualize graphs by looking at an *embedding*. An embedding of a graph maps each vertex to a point in the plane and each edge to a curve or straight line segment between the two vertices. A graph is *planar* if it has an embedding where no two edges cross. The same graph can have many different embeddings, so it is important not to confuse a particular embedding with the graph itself. In particular, planar graphs can have non-planar embeddings!

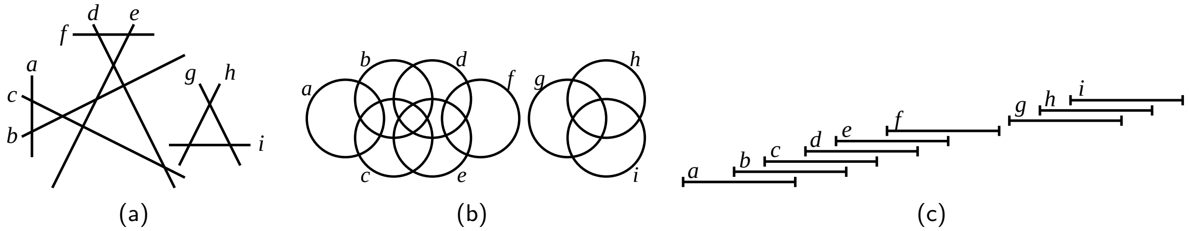


A non-planar embedding of a planar graph with nine vertices, thirteen edges, and two connected components, and a planar embedding of the same graph.

There are other ways of visualizing and representing graphs that are sometimes also useful. For example, the *intersection graph* of a collection of objects has a node for every object and an edge for every intersecting pair. Whether a particular graph can be represented as an intersection graph

[1] The singular of 'vertices' is **vertex**. The singular of 'matrices' is **matrix**. Unless you're speaking Italian, there is no such thing as a vertice, a matrice, an indice, an appendice, a helice, an apice, a vortice, a radice, a simplice, an apice, a codice, a directrice, a dominatrice, a Unice, a Kleenice, an Asterice, an Obelice, a Dogmatice, a Getafice, a Cacofonice, a Vitalstatistice, a Geriatrice, or Jimi Hendrice! You *will* lose points for using any of these so-called words.

depends on what kind of object you want to use for the vertices. Different types of objects—line segments, rectangles, circles, etc.—define different classes of graphs. One particularly useful type of intersection graph is an *interval graph*, whose vertices are intervals on the real line, with an edge between any two intervals that overlap.



The example graph is also the intersection graph of (a) a set of line segments, (b) a set of circles, or (c) a set of intervals on the real line (stacked for visibility).

If (u, v) is an edge in an undirected graph, then u is a *neighbor* or v and vice versa. The *degree* of a node is the number of neighbors. In directed graphs, we have two kinds of neighbors. If $u \to v$ is a directed edge, then u is a *predecessor* of v and v is a *successor* of u. The *in-degree* of a node is the number of predecessors, which is the same as the number of edges going into the node. The *out-degree* is the number of successors, or the number of edges going out of the node.

A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A *path* is a sequence of edges, where each successive pair of edges shares a vertex, and all other edges are disjoint. A graph is *connected* if there is a path from any vertex to any other vertex. A disconnected graph consists of several *connected components*, which are maximal connected subgraphs. Two vertices are in the same connected component if and only if there is a path between them.

A *cycle* is a path that starts and ends at the same vertex, and has at least one edge. A graph is *acyclic* if no subgraph is a cycle; acyclic graphs are also called *forests*. *Trees* are special graphs that can be defined in several different ways. You can easily prove by induction (hint, hint, hint) that the following definitions are equivalent.

- A tree is a connected acyclic graph.
- A tree is a connected component of a forest.
- A tree is a connected graph with *at most* $V - 1$ edges.
- A tree is a minimal connected graph; removing any edge makes the graph disconnected.
- A tree is an acyclic graph with *at least* $V - 1$ edges.
- A tree is a maximal acyclic graph; adding an edge between any two vertices creates a cycle.

A *spanning tree* of a graph G is a subgraph that is a tree and contains every vertex of G. Of course, a graph can only have a spanning tree if it's connected. A *spanning forest* of G is a collection of spanning trees, one for each connected component of G.

11.2 Explicit Representations of Graphs

There are two common data structures used to explicitly represent graphs: *adjacency matrices*[2] and *adjacency lists*.

[2] See footnote 1.

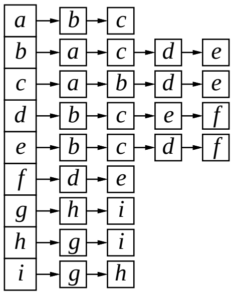
The adjacency matrix of a graph G is a $V \times V$ matrix of indicator variables. Each entry in the matrix indicates whether a particular edge is or is not in the graph:

$$A[i,j] = \big[(i,j) \in E\big].$$

For undirected graphs, the adjacency matrix is always *symmetric*: $A[i,j] = A[j,i]$. Since we don't allow edges from a vertex to itself, the diagonal elements $A[i,i]$ are all zeros.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\Theta(V)$ time by scanning the corresponding row (or column). This is optimal in the worst case, since a vertex can have up to $V-1$ neighbors; however, if a vertex has few neighbors, we may still have to examine every entry in the row to see them all. Similarly, adjacency matrices require $\Theta(V^2)$ space, regardless of how many edges the graph actually has, so it is only space-efficient for very *dense* graphs.

$$\begin{array}{c|ccccccccc}
 & a & b & c & d & e & f & g & h & i \\
\hline
a & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
b & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
c & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
d & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
e & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
f & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
h & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
i & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}$$



Adjacency matrix and adjacency list representations for the example graph.

For *sparse* graphs—graphs with relatively few edges—we're better off using adjacency lists. An adjacency list is an array of linked lists, one list per vertex. Each linked list stores the neighbors of the corresponding vertex.

For undirected graphs, each edge (u,v) is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge is stores only once. Either way, the overall space required for an adjacency list is $O(V+E)$. Listing the neighbors of a node v takes $O(1+\deg(v))$ time; just scan the neighbor list. Similarly, we can determine whether (u,v) is an edge in $O(1+\deg(u))$ time by scanning the neighbor list of u. For undirected graphs, we can speed up the search by simultaneously scanning the neighbor lists of both u and v, stopping either we locate the edge or when we fall of the end of a list. This takes $O(1+\min\{\deg(u),\deg(v)\})$ time.

The adjacency list structure should immediately remind you of hash tables with chaining. Just as with hash tables, we can make adjacency list structure more efficient by using something besides a linked list to store the neighbors. For example, if we use a hash table with constant load factor, when we can detect edges in $O(1)$ expected time, just as with an adjacency list. In practice, this will only be useful for vertices with large degree, since the constant overhead in both the space and search time is larger for hash tables than for simple linked lists.

You might at this point ask why anyone would ever use an adjacency matrix. After all, if you use hash tables to store the neighbors of each vertex, you can do everything as fast or faster with an adjacency list as with an adjacency matrix, only using less space. The answer is that many graphs are only represented *implicitly*. For example, intersection graphs are usually represented implicitly by simply storing the list of objects. As long as we can test whether two objects overlap in constant time, we can apply any graph algorithm to an intersection graph by *pretending* that it is stored explicitly as an adjacency matrix. On the other hand, any data structure build from records with pointers between them can be seen as a directed graph. Algorithms for searching graphs can be applied to these data structures by *pretending* that the graph is represented explicitly using an adjacency list.

To keep things simple, we'll consider only undirected graphs for the rest of this lecture, although the algorithms I'll describe also work for directed graphs.

11.3 Traversing connected graphs

Suppose we want to visit every node in a connected graph (represented either explicitly or implicitly). The simplest method to do this is an algorithm called *depth-first search*, which can be written either recursively or iteratively. It's exactly the same algorithm either way; the only difference is that we can actually see the 'recursion' stack in the non-recursive version. Both versions are initially passed a *source* vertex s.

```
RecursiveDFS($v$):
  if $v$ is unmarked
      mark $v$
      for each edge $(v, w)$
          RecursiveDFS($w$)
```

```
IterativeDFS($s$):
  Push($s$)
  while stack not empty
      $v \leftarrow$ Pop
      if $v$ is unmarked
          mark $v$
          for each edge $(v, w)$
              Push($w$)
```

Depth-first search is one (perhaps the most common) instance of a general family of graph traversal algorithms. The generic graph traversal algorithm stores a set of candidate edges in some data structure that I'll call a 'bag'. The only important properties of a 'bag' are that we can put stuff into it and then later take stuff back out. (In C++ terms, think of the 'bag' as a template for a real data structure.) Here's the algorithm:

```
Traverse($s$):
  put $(\varnothing, s)$ in bag
  while the bag is not empty
      take $(p, v)$ from the bag          $(\star)$
      if $v$ is unmarked
          mark $v$
          parent($v$) $\leftarrow p$
          for each edge $(v, w)$        $(\dagger)$
              put $(v, w)$ into the bag       $(\star\star)$
```

Notice that we're keeping *edges* in the bag instead of *vertices*. This is because we want to remember, whenever we visit a vertex v for the first time, which previously-visited vertex p put v into the bag. The vertex p is called the *parent* of v.

Lemma 1. Traverse(s) *marks every vertex in any connected graph exactly once, and the set of edges* $(v, \mathit{parent}(v))$ *with* $\mathit{parent}(v) \neq \varnothing$ *form a spanning tree of the graph.*

Proof: First, it should be obvious that no node is marked more than once.

Clearly, the algorithm marks s. Let $v \neq s$ be a vertex, and let $s \to \cdots \to u \to v$ be the path from s to v with the minimum number of edges. Since the graph is connected, such a path always exists. (If s and v are neighbors, then $u = s$, and the path has just one edge.) If the algorithm marks u, then it must put (u, v) into the bag, so it must later take (u, v) out of the bag, at which point v

must be marked (if it isn't already). Thus, by induction on the shortest-path distance from s, the algorithm marks every vertex in the graph.

Call an edge $(v, \mathit{parent}(v))$ with $\mathit{parent}(v) \neq \varnothing$ a *parent edge*. For any node v, the path of parent edges $v \rightarrow \mathit{parent}(v) \rightarrow \mathit{parent}(\mathit{parent}(v)) \rightarrow \cdots$ eventually leads back to s, so the set of parent edges form a connected graph. Clearly, both endpoints of every parent edge are marked, and the number of parent edges is exactly one less than the number of vertices. Thus, the parent edges form a spanning tree. □

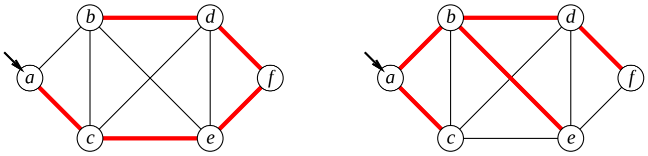
The exact running time of the traversal algorithm depends on how the graph is represented and what data structure is used as the 'bag', but we can make a few general observations. Since each vertex is visited at most once, the for loop (†) is executed at most V times. Each edge is put into the bag exactly twice; once as (u, v) and once as (v, u), so line ($\star\star$) is executed at most $2E$ times. Finally, since we can't take more things out of the bag than we put in, line ($\star$) is executed at most $2E + 1$ times.

11.4 Examples

Let's first assume that the graph is represented by an adjacency list, so that the overhead of the for loop (†) is only a constant per edge.

- If we implement the 'bag' by using a *stack*, we have depth-first search. Each execution of ($\star$) or ($\star\star$) takes constant time, so the overall running time is $O(V + E)$. Since the graph is connected, $V \leq E + 1$, so we can simplify the running time to $O(E)$. The spanning tree formed by the parent edges is called a *depth-first spanning tree*. The exact shape of the tree depends on the order in which neighbor edges are pushed onto the stack, but the in general, depth-first spanning trees are long and skinny.

- If we use a *queue* instead of a stack, we have *breadth-first search*. Again, each execution of ($\star$) or ($\star\star$) takes constant time, so the overall running time is still $O(E)$. In this case, the *breadth-first spanning tree* formed by the parent edges contains *shortest paths* from the start vertex s to every other vertex in its connected component. The exact shape of the shortest path tree depends on the order in which neighbor edges are pushed onto the queue, but the in general, shortest path trees are short and bushy. We'll see shortest paths again next week.



A depth-first spanning tree and a breadth-first spanning tree of one component of the example graph, with start vertex a.

- Suppose the edges of the graph are weighted. If we implement the 'bag' using a *priority queue*, always extracting the minimum-weight edge in line ($\star$), then we we have what might be called *shortest-first search*. In this case, each execution of ($\star$) or ($\star\star$) takes $O(\log E)$ time, so the overall running time is $O(V + E \log E)$, which simplifies to $O(E \log E)$ if the graph is connected. For this algorithm, the set of parent edges form the *minimum spanning tree* of the connected component of s. We'll see minimum spanning trees again in the next lecture.

If the graph is represented using an adjacency matrix, the finding all the neighbors of each vertex in line (†) takes $O(V)$ time. Thus, depth- and breadth-first search take $O(V^2)$ time overall, and 'shortest-first search' takes $O(V^2 + E \log E) = O(V^2 \log V)$ time overall.

11.5 Searching disconnected graphs

If the graph is disconnected, then Traverse(s) only visits the nodes in the connected component of the start vertex s. If we want to visit all the nodes in every component, we can use the following 'wrapper' around our generic traversal algorithm. Since Traverse computes a spanning tree of one component, TraverseAll computes a spanning *forest* of the entire graph.

```
TraverseAll($s$):
  for all vertices $v$
      if $v$ is unmarked
          Traverse($v$)
```

There is a rather unfortunate mistake on page 477 of CLR:

> Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by depth-first search may be composed of several trees, because the search may be repeated from multiple sources.

This statement seems to imply that depth-first search is always called with the TraverseAll, and breadth-first search never is, **but this is not true!** The choice of whether to use a stack or a queue is completely independent of the choice of whether to use TraverseAll or not.

We must all hang together, gentlemen, or else we shall most assuredly hang separately.

— Benjamin Franklin, at the signing of the Declaration of Independence (July 4, 1776)

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde

Computers are useless. They can only give you answers.

— Pablo Picasso

A ship in port is safe, but that is not what ships are for.

— Rear Admiral Grace Murray Hopper

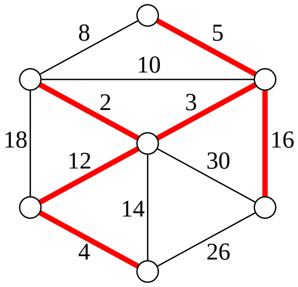
12 Minimum Spanning Trees

12.1 Introduction

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w \colon E \to \mathbb{R}$ that assigns a *weight* $w(e)$ to each edge e. For this lecture, we'll assume that the weights are real numbers. Our task is to find the *minimum spanning tree* of G, *i.e.*, the spanning tree T minimizing the function

$$w(T) = \sum_{e \in T} w(e).$$

To keep things simple, I'll assume that all the edge weights are distinct: $w(e) \neq w(e')$ for any pair of edges e and e'. Distinct weights guarantee that the minimum spanning tree of the graph is unique. Without this condition, there may be several different minimum spanning trees. For example, if all the edges have weight 1, then *every* spanning tree is a minimum spanning tree with weight $V - 1$.



A weighted graph and its minimum spanning tree.

If we have an algorithm that assumes the edge weights are unique, we can still use it on graphs where multiple edges have the same weight, as long as we have a consistent method for breaking ties. One way to break ties consistently is to use the following algorithm in place of a simple comparison. ShorterEdge takes as input four integers i, j, k, l, and decides which of the two edges (i, j) and (k, l) has 'smaller' weight.

```
SHORTEREDGE($i, j, k, l$)
    if $w(i,j) < w(k,l)$ return $(i,j)$
    if $w(i,j) > w(k,l)$ return $(k,l)$
    if $\min(i,j) < \min(k,l)$ return $(i,j)$
    if $\min(i,j) > \min(k,l)$ return $(k,l)$
    if $\max(i,j) < \max(k,l)$ return $(i,j)$
    ⟨⟨if max(i,j) < max(k,l)⟩⟩ return $(k,l)$
```

12.2 The Only Minimum Spanning Tree Algorithm

There are several different methods for computing minimum spanning trees, but really they are all instances of the following generic algorithm. The situation is similar to the previous lecture, where we saw that depth-first search and breadth-first search were both instances of a single generic traversal algorithm.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G, which we will call an *intermediate spanning forest*. F is a subgraph of the minimum spanning tree of G, and every component of F is a minimum spanning tree of its vertices. Initially, F consists of n one-node trees. The generic algorithm merges trees together by adding certain edges between them. When the algorithm halts, F consists of a single n-node tree, which must be the minimum spanning tree. Obviously, we have to be careful about *which* edges we add to the evolving forest, since not every edge is in the minimum spanning tree.

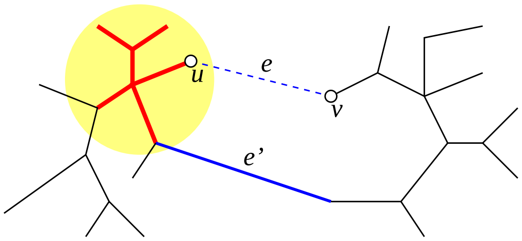
The intermediate spanning forest F induces two special types of edges. An edge is *useless* if it is not an edge of F, but both its endpoints are in the same component of F. For each component of F, we associate a *safe* edge—the minimum-weight edge with exactly one endpoint in that component. Different components might or might not have different safe edges. Some edges are neither safe nor useless—we call these edges *undecided*.

All minimum spanning tree algorithms are based on two simple observations.

Lemma 1. *The minimum spanning tree contains every safe edge and no useless edges.*[1]

Proof: Let T be the minimum spanning tree. Suppose F has a 'bad' component whose safe edge $e = (u, v)$ is not in T. Since T is connected, it contains a unique path from u to v, and at least one edge e' on this path has exactly one endpoint in the bad component. Removing e' from the minimum spanning tree and adding e gives us a new spanning tree. Since e is the bad component's safe edge, we have $w(e') > w(e)$, so the the new spanning tree has smaller total weight than T. But this is impossible—T is the *minimum* spanning tree. So T must contain every safe edge.

Adding any useless edge to F would introduce a cycle. □



Proving that every safe edge is in the minimum spanning tree. The 'bad' component of F is highlighted.

[1] This is actually a special case of two more general theorems: First, for any partition of the vertices of G into two disjoint subsets, the minimum-weight edge with one endpoint in each subset **is** in the minimum spanning tree. Second, the maximum-weight edge in any cycle in G is **not** in the minimum spanning tree. A few minimum spanning tree algorithms require this more general result, but we won't talk about them here.

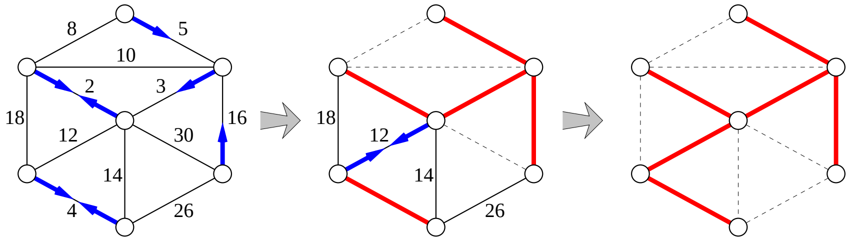
So our generic minimum spanning tree algorithm repeatedly adds one or more safe edges to the evolving forest F. Whenever we add new edges to F, some undecided edges become safe, and others become useless. To specify a particular algorithm, we must decide which safe edges to add, and how to identify new safe and new useless edges, at each iteration of our generic template.

12.3 Borůvka's Algorithm

The oldest and possibly simplest minimum spanning tree algorithm was discovered by Borůvka in 1926, long before computers even existed, and practically before the invention of graph theory![2] The algorithm was rediscovered by Choquet in 1938; again by Florek, Łukaziewicz, Perkal, Stienhaus, and Zubrzycki in 1951; and again by Sollin some time in the early 1960s. Because Sollin was the only Western computer scientist in this list—Choquet was a civil engineer; Florek and his co-authors were anthropologists—this algorithm is frequently but incorrectly called 'Sollin's algorithm', especially in the parallel computing literature.[3]

The Borůvka/Choquet/Florek/Łukaziewicz/Perkal/Stienhaus/Zubrzycki/Sollin algorithm can be summarized in one line:

BORŮVKA: Add all the safe edges and recurse.



Borůvka's algorithm run on the example graph. Thick edges are in F.
Arrows point along each component's safe edge. Dashed edges are useless.

At the beginning of each phase of the Borůvka algorithm, each component elects an arbitrary 'leader' node. The simplest way to hold these elections is a depth-first search of F; the first node we visit in any component is that component's leader. Once the leaders are elected, we find the safe edges for each component, essentially by brute force. Finally, we add these safe edges to F.

```
BORŮVKA($V, E$):
    $F = (V, \varnothing)$
    while $F$ has more than one component
        choose leaders using DFS
        FINDSAFEEDGES($V, E$)
        for each leader $\overline{v}$
            add safe($\overline{v}$) to $F$
```

```
FINDSAFEEDGES($V, E$):
    for each leader $\overline{v}$
        safe($\overline{v}$) ← $\infty$
    for each edge $(u, v) \in E$
        $\overline{u}$ ← leader($u$)
        $\overline{v}$ ← leader($v$)
        if $\overline{u} \neq \overline{v}$
            if $w(u, v) < w(\text{safe}(\overline{u}))$
                safe($\overline{u}$) ← $(u, v)$
            if $w(u, v) < w(\text{safe}(\overline{v}))$
                safe($\overline{v}$) ← $(u, v)$
```

[2] Leonard Euler published the first graph theory result, his famous theorem about the bridges of Königsburg, in 1736. However, the first textbook on graph theory, written by Dénes König, was not published until 1936. Alas, König was not from Königsburg.

[3] Algorithms are quite commonly named after their *last* discoverers.

Each call to FINDSAFEEDGES takes $O(E)$ time, since it examines every edge. Since the graph is connected, it has at most $E+1$ vertices. Thus, each iteration of the while loop in BORŮVKA takes $O(E)$ time, assuming the graph is represented by an adjacency list. Each iteration also reduces the number of components of F by at least a factor of two—the worst case occurs when the components coalesce in pairs. Since there are initially V components, the while loop iterates $O(\log V)$ times. Thus, the overall running time of Borůvka's algorithm is $\boxed{O(E \log V)}$.

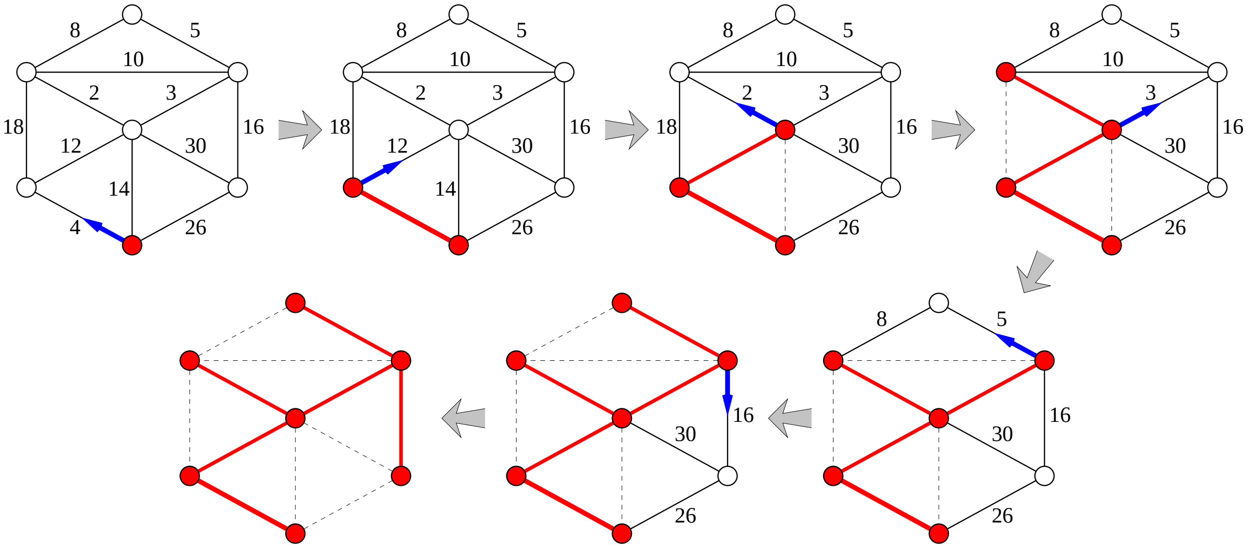
Despite its relatively obscure origin, early algorithms researchers were aware of Borůvka's algorithm, but dismissed it as being "too complicated"! As a result, despite its simplicity and efficiency, Borůvka's algorithm is rarely mentioned in algorithms and data structures textbooks.

12.4 Jarník's ('Prim's') Algorithm

The next oldest minimum spanning tree algorithm was first described by the Polish mathematician Vojtěch Jarník in a 1929 letter to Borůvka. The algorithm was independently rediscovered by Kruskal in 1956, by Prim in 1957, by Loberman and Weinberger in 1957, and finally by Dijkstra in 1958. Prim, Loberman, Weinberger, and Dijkstra all (eventually) knew of and even cited Kruskal's paper, but since Kruskal also described two other minimum-spanning-tree algorithms in the same paper, *this* algorithm is usually (incorrectly) called 'Prim's algorithm', or sometimes even 'the Prim/Dijkstra algorithm'[4], even though by 1958 Dijkstra already had another algorithm (inappropriately) named after him.

In Jarník's algorithm, the forest F contains only one nontrivial component T; all the other components are isolated vertices. Initially, T consists of an arbitrary vertex of the graph. The algorithm repeats the following step until T spans the whole graph:

JARNÍK: Find T's safe edge and add it to T.



Jarník's algorithm run on the example graph, starting with the bottom vertex.
At each stage, thick edges are in T, an arrow points along T's safe edge, and dashed edges are useless.

To implement Jarník's algorithm, we keep all the edges adjacent to T in a heap. When we pull the minimum-weight edge off the heap, we first check whether both of its endpoints are in T.

[4] following the Last Discoverer Rule

If not, we add the edge to T and then add the new neighboring edges to the heap. In other words, Jarník's algorithm is just another instance of the generic graph traversal algorithm we saw last time, using a heap as the 'bag'! If we implement the algorithm this way, its running time is $O(E \log E) = O(E \log V)$.

However, we can speed up the implementation by observing that the graph traversal algorithm visits each vertex only once. Rather than keeping edges in the heap, we can keep a heap of vertices, where the key of each vertex v is the length of the minimum-weight edge between v and T (or ∞ if there is no such edge). Each time we add a new edge to T, we may need to decrease the key of some neighboring vertices.

To make the description easier, we break the algorithm into two parts. JarníkInit initializes the vertex heap. JarníkLoop is the main algorithm. The input consists of the vertices and edges of the graph, plus the start vertex s.

```
Jarník($V, E, s$):
    JarníkInit($V, E, s$)
    JarníkLoop($V, E, s$)
```

```
JarníkInit($V, E, s$):
    for each vertex $v \in V \setminus \{s\}$
        if $(v, s) \in E$
            edge($v$) ← $(v, s)$
            key($v$) ← $w(v, s)$
        else
            edge($v$) ← Null
            key($v$) ← $\infty$
        Insert($v$)
```

```
JarníkLoop($V, E, s$):
    $T \leftarrow (\{s\}, \varnothing)$
    for $i \leftarrow 1$ to $|V| - 1$
        $v \leftarrow$ ExtractMin
        add $v$ and edge($v$) to $T$
        for each edge $(u, v) \in E$
            if $u \notin T$ and key($u$) $> w(u, v)$
                edge($u$) ← $(u, v)$
                DecreaseKey($u, w(u, v)$)
```

The running time of Jarník is dominated by the cost of the heap operations Insert, ExtractMin, and DecreaseKey. Insert and ExtractMin are each called $O(V)$ times once for each vertex except s, and DecreaseKey is called $O(E)$ times, at most twice for each edge. If we use a Fibonacci heap[5], the amortized costs of Insert and DecreaseKey is $O(1)$, and the amortized cost of ExtractMin is $O(\log n)$. Thus, the overall running time of Jarník is $\boxed{O(E + V \log V)}$. This is faster than Borůvka's algorithm unless $E = O(V)$.

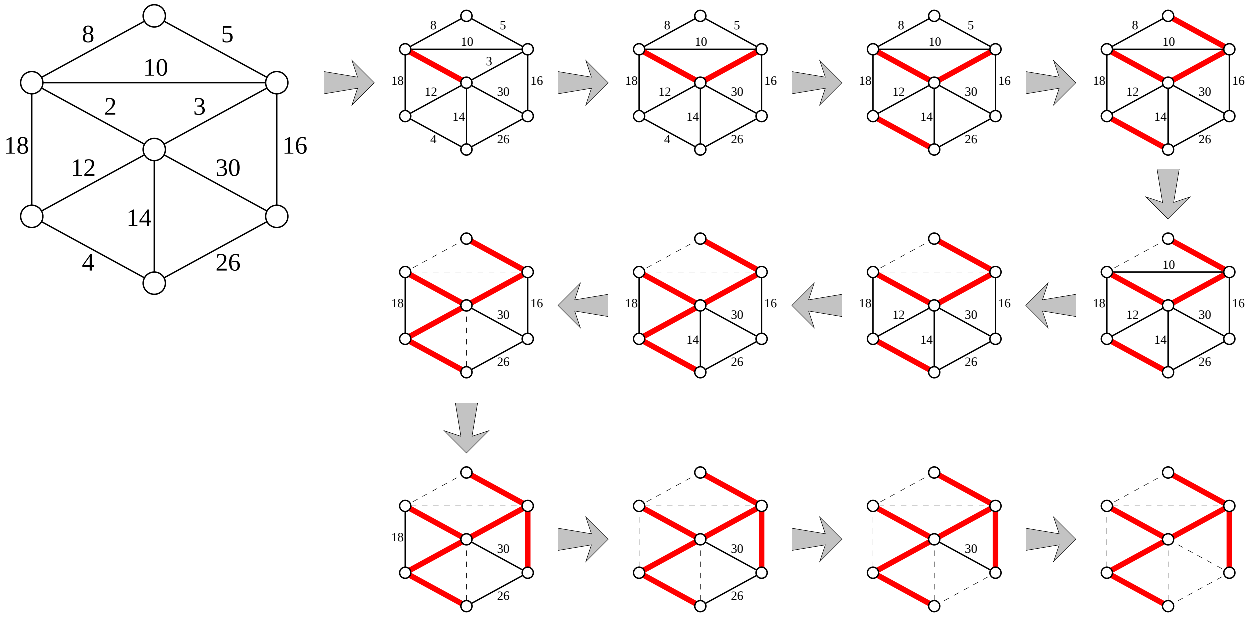
12.5 Kruskal's Algorithm

The last minimum spanning tree algorithm I'll discuss was first described by Kruskal in 1956, in the same paper where he rediscovered Jarnik's algorithm. Kruskal was motivated by 'a typewritten translation (of obscure origin)' of Borůvka's original paper, claiming that Borůvka's algorithm was 'unnecessarily elaborate'.[6] This algorithm was also rediscovered in 1957 by Loberman and Weinberger, but somehow avoided being renamed after them.

$\boxed{\text{Kruskal: Scan all edges in increasing weight order; if an edge is safe, add it to } F.}$

[5] See Non-Lecture B.

[6] To be fair, Borůvka's original paper *was* unnecessarily elaborate, but in his followup paper, also published in 1927, simplified his algorithm essentially to the form presented in this lecture note. Kruskal was apparently unaware of Borůvka's second paper. Stupid Iron Curtain.



Kruskal's algorithm run on the example graph. Thick edges are in F. Dashed edges are useless.

Since we examine the edges in order from lightest to heaviest, any edge we examine is safe if and only if its endpoints are in different components of the forest F. To prove this, suppose the edge e joins two components A and B but is not safe. Then there would be a lighter edge e' with exactly one endpoint in A. But this is impossible, because (inductively) any previously examined edge has both endpoints in the same component of F.

Just as in Borůvka's algorithm, each component of F has a 'leader' node. An edge joins two components of F if and only if the two endpoints have different leaders. But unlike Borůvka's algorithm, we do not recompute leaders from scratch every time we add an edge. Instead, when two components are joined, the two leaders duke it out in a nationally-televised no-holds-barred steel-cage grudge match.[7] One of the two emerges victorious as the leader of the new larger component. More formally, we will use our earlier algorithms for the Union-Find problem, where the vertices are the elements and the components of F are the sets. Here's a more formal description of the algorithm:

```
Kruskal($V, E$):
  sort $E$ by wieght
  $F \leftarrow \varnothing$
  for each vertex $v \in V$
      MakeSet($v$)
  for $i \leftarrow 1$ to $|E|$
      $(u, v) \leftarrow i$th lightest edge in $E$
      if Find($u$) $\neq$ Find($v$)
          Union($u, v$)
          add $(u, v)$ to $F$
  return $F$
```

In our case, the sets are components of F, and $n = V$. Kruskal's algorithm performs $O(E)$ Find operations, two for each edge in the graph, and $O(V)$ Union operations, one for each edge in the

[7] Live at the Assembly Hall! Only $49.95 on Pay-Per-View!

minimum spanning tree. Using union-by-rank and path compression allows us to perform each UNION or FIND in $O(\alpha(E,V))$ time, where α is the not-quite-constant inverse-Ackerman function. So ignoring the cost of sorting the edges, the running time of this algorithm is $O(E\,\alpha(E,V))$.

We need $O(E\log E) = O(E\log V)$ additional time just to sort the edges. Since this is bigger than the time for the UNION-FIND data structure, the overall running time of Kruskal's algorithm is $\boxed{O(E\log V)}$, exactly the same as Borůvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

> *Well, ya turn left by the fire station in the village and take the old post road by the reservoir and. . . no, that won't do.*
>
> *Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on the road to Bennett's Lake until. . . no, that won't work either.*
>
> *East Millinocket, ya say? Come to think of it, you can't get there from here.*
>
> — Robert Bryan and Marshall Dodge,
> *Bert and I and Other Stories from Down East* (1961)

> *Hey farmer! Where does this road go?*
> *Been livin' here all my life, it ain't gone nowhere yet.*
>
> *Hey farmer! How do you get to Little Rock?*
> *Listen stranger, you can't get there from here.*
>
> *Hey farmer! You don't know very much do you?*
> *No, but I ain't lost.*
>
> — Michelle Shocked, "Arkansas Traveler" (1992)

13 Shortest Paths

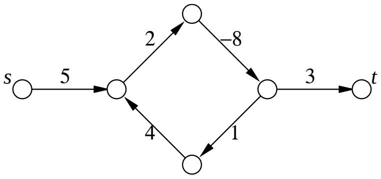
13.1 Introduction

Given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, a *source* s and a *target* t, we want to find the shortest directed path from s to t. In other words, we want to find the path p starting at s and ending at t minimizing the function

$$w(p) = \sum_{e \in p} w(e).$$

For example, if I want to answer the question 'What's the fastest way to drive from my old apartment in Champaign, Illinois to my wife's old apartment in Columbus, Ohio?', we might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Champaign, and t is Columbus.[1] The graph is directed since the driving times along the same road might be different in different directions.[2]

Perhaps counter to intuition, we will allow the weights on the edges to be negative. Negative edges make our lives complicated, since the presence of a negative cycle might mean that there is no shortest path. In general, a shortest path from s to t exists if and only if there is *at least one* path from s to t, but there is no path from s to t that touches a negative cycle. If there is a negative cycle between s and t, then se can always find a shorter path by going around the cycle one more time.



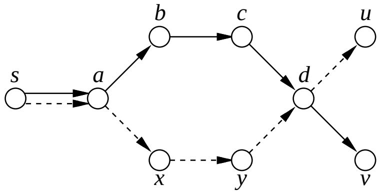
There is no shortest path from s to t.

[1] West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We both live in Urbana now.

[2] There is a speed trap on I-70 just inside the Ohio border, but only for eastbound traffic.

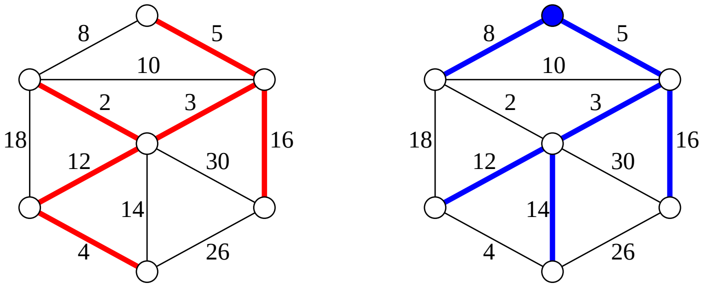
Every algorithm known for solving this problem actually solves (large portions of) the following more general *single source shortest path* or *SSSP* problem: find the shortest path from the source vertex s to *every* other vertex in the graph. In fact, the problem is usually solved by finding a *shortest path tree* rooted at s that contains all the desired shortest paths.

It's not hard to see that if shortest paths are unique, then they form a tree. To prove this, it's enough to observe that any subpath of a shortest path is also a shortest path. If there are multiple shortest paths to the same vertices, we can always choose one path to each vertex so that the union of the paths is a tree. If there are shortest paths to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths so that the two paths only diverge once.



If $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are both shortest paths, then $s \to a \to b \to c \to d \to u$ is also a shortest path.

Shortest path trees and minimum spanning trees are usually very different. For one thing, there is only one minimum spanning tree, but in general, there is a different shortest path tree for every source vertex. Moreover, in general, *all* of these shortest path trees are different from the minimum spanning tree.



A minimum spanning tree and a shortest path tree (rooted at the topmost vertex) of the same graph.

All of the algorithms I'm describing in this lecture also work for undirected graphs, with some slight modifications. Most importantly, we must specifically prohibit alternating back and forth across the same undirected negative-weight edge. Our unmodified algorithms would interpret any such edge as a negative cycle of length 2.

To emphasize the direction, I will consistently use the nonstandard notation $u \to v$ to denote a directed edge from u to v.

13.2 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, there are several different SSSP algorithms, but they are all special cases of the a single generic algorithm, first proposed by Ford in 1956, and independently by Dantzig in 1957.[3] Each vertex v in the graph stores two values, which (inductively) describe a *tentative* shortest path from s to v.

[3] Specifically, Dantzig showed that the shortest path problem can be phrased as a linear programming problem, and then described an interpretation of the simplex method (which Dantzig discovered) in terms of the original graph. His description was equivalent to Ford's relaxation strategy.

- *dist*(v) is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- *pred*(v) is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or NULL if there is no such vertex.

The predecessor pointers automatically define a tentative shortest path tree; they play the same role as the 'parent' pointers in our generic graph traversal algorithm. We already know that *dist*$(s) = 0$ and *pred*$(s) =$ NULL. For every vertex $v \neq s$, we initially set *dist*$(v) = \infty$ and *pred*$(v) =$ NULL to indicate that we do not know of *any* path from s to v.

We call an edge $u \to v$ *tense* if *dist*$(u) + w(u \to v) <$ *dist*(v). If $u \to v$ is tense, then the tentative shortest path $s \rightsquigarrow v$ is incorrect, since the path $s \rightsquigarrow u \to v$ is shorter. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

```
RELAX($u \to v$):
    $dist(v) \leftarrow dist(u) + w(u \to v)$
    $pred(v) \leftarrow u$
```

If there are no tense edges, our algorithm is finished, and we have our desired shortest path tree.

The correctness of the relaxation algorithm follows directly from three simple claims:

1. If *dist*$(v) \neq \infty$, then *dist*(v) is the total weight of the predecessor chain ending at v:

$$s \to \cdots \to pred(pred(v)) \to pred(v) \to v.$$

This is easy to prove by induction on the number of relaxation steps. (Hint, hint.)

2. If the algorithm halts, then *dist*$(v) \leq w(s \rightsquigarrow v)$ for *any* path $s \rightsquigarrow v$. This is easy to prove by induction on the number of edges in the path $s \rightsquigarrow v$. (Hint, hint.)

3. The algorithm halts if and only if there is no negative cycle reachable from s. The 'only if' direction is easy—if there is a reachable negative cycle, then after the first edge in the cycle is relaxed, the cycle *always* has at least one tense edge. The 'if' direction follows from the fact that every relaxation step reduces either the number of vertices with *dist*$(v) = \infty$ by 1 or reduces the sum of the finite shortest path lengths by the difference between two edge weights.

I haven't said anything about how we detect which edges can be relaxed, or in what order we relax them. In order to make this easier, we can refine the relaxation algorithm slightly, into something closely resembling the generic graph traversal algorithm. We maintain a 'bag' of vertices, initially containing just the source vertex s. Whenever we take a vertex u out of the bag, we scan all of its outgoing edges, looking for something to relax. Whenever we successfully relax an edge $u \to v$, we put v into the bag. Unlike our generic graph traversal algorithm, the same vertex might be visited many times.

```
INITSSSP($s$):
    $dist(s) \leftarrow 0$
    $pred(s) \leftarrow$ NULL
    for all vertices $v \neq s$
        $dist(v) \leftarrow \infty$
        $pred(v) \leftarrow$ NULL
```

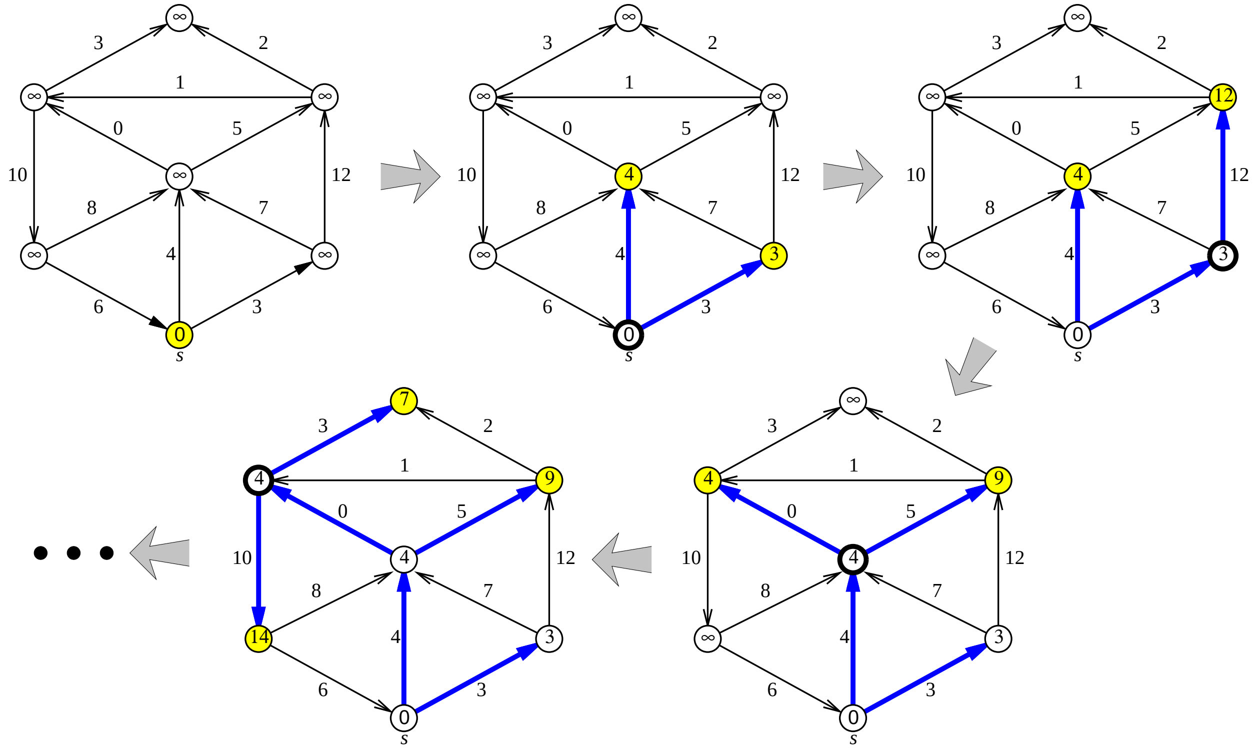
```
GENERICSSSP($s$):
    INITSSSP($s$)
    put $s$ in the bag
    while the bag is not empty
        take $u$ from the bag
        for all edges $u \to v$
            if $u \to v$ is tense
                RELAX($u \to v$)
                put $v$ in the bag
```

Just as with graph traversal, using different data structures for the 'bag' gives us different algorithms. There are three obvious choices to try: a stack, a queue, and a heap. Unfortunately, if we use a stack, we have to perform $\Theta(2^V)$ relaxation steps in the worst case! (Proving this is a good homework problem.) The other two possibilities are much more efficient.

13.3 Dijkstra's Algorithm

If we implement the bag as a heap, where the key of a vertex v is *dist*(v), we obtain an algorithm first 'published'[4] by Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, and Seitz in 1957, and then later independently rediscovered by Edsger Dijkstra in 1959. A very similar algorithm was also described by Dantzig in 1958.

Dijkstra's algorithm, as it is universally known[5], is particularly well-behaved if the graph has no negative-weight edges. In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s. It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once. Since the key of each vertex in the heap is its tentative distance from s, the algorithm performs a DECREASEKEY operation every time an edge is relaxed. Thus, the algorithm performs at most E DECREASEKEYS. Similarly, there are at most V INSERT and EXTRACTMIN operations. Thus, if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $\boxed{O(E + V \log V)}$.



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

[4] in the first annual report on a research project performed for the Combat Development Department of the Army Electronic Proving Ground

[5] I will follow this common convention, despite the historical inaccuracy, because I don't think anybody wants to read about the "Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-Seitz algorithm".

This analysis assumes that no edge has negative weight. Dijkstra's algorithm (in the form I'm presenting here) is still *correct* if there are negative edges[6], but the worst-case running time could be exponential. (Proving this unfortunate fact is a good homework problem.)

13.4 The A^* Heuristic

A slight generalization of Dijkstra's algorithm, commonly known as the A^* algorithm, is frequently used to find a shortest path from a single source node s to a single target node t. A^* uses a black-box function GUESSDISTANCE(v, t) that returns an estimate of the distance from v to t. The only difference between Dijkstra and A^* is that the key of a vertex v is *dist*(v) + GUESSDISTANCE(v, t).

The function GUESSDISTANCE is called *admissible* if GUESSDISTANCE(v, t) never overestimates the actual shortest path distance from v to t. If GUESSDISTANCE is admissible and the actual edge weights are all non-negative, the A^* algorithm computes the actual shortest path from s to t at least as quickly as Dijkstra's algorithm. The closer GUESSDISTANCE(v, t) is to the real distance from v to t, the faster the algorithm. However, in the worst case, the running time is still $O(E+V\log V)$.

The heuristic is especially useful in situations where the actual graph is not known. For example, A^* can be used to solve puzzles (15-puzzle, Freecell, Shanghai, Sokoban, Atomix, Rush Hour, Rubik's Cube, ...) and other path planning problems where the starting and goal configurations are given, but the graph of all possible configurations and their connections is not given explicitly.

13.5 Shimbel's Algorithm ('Bellman-Ford')

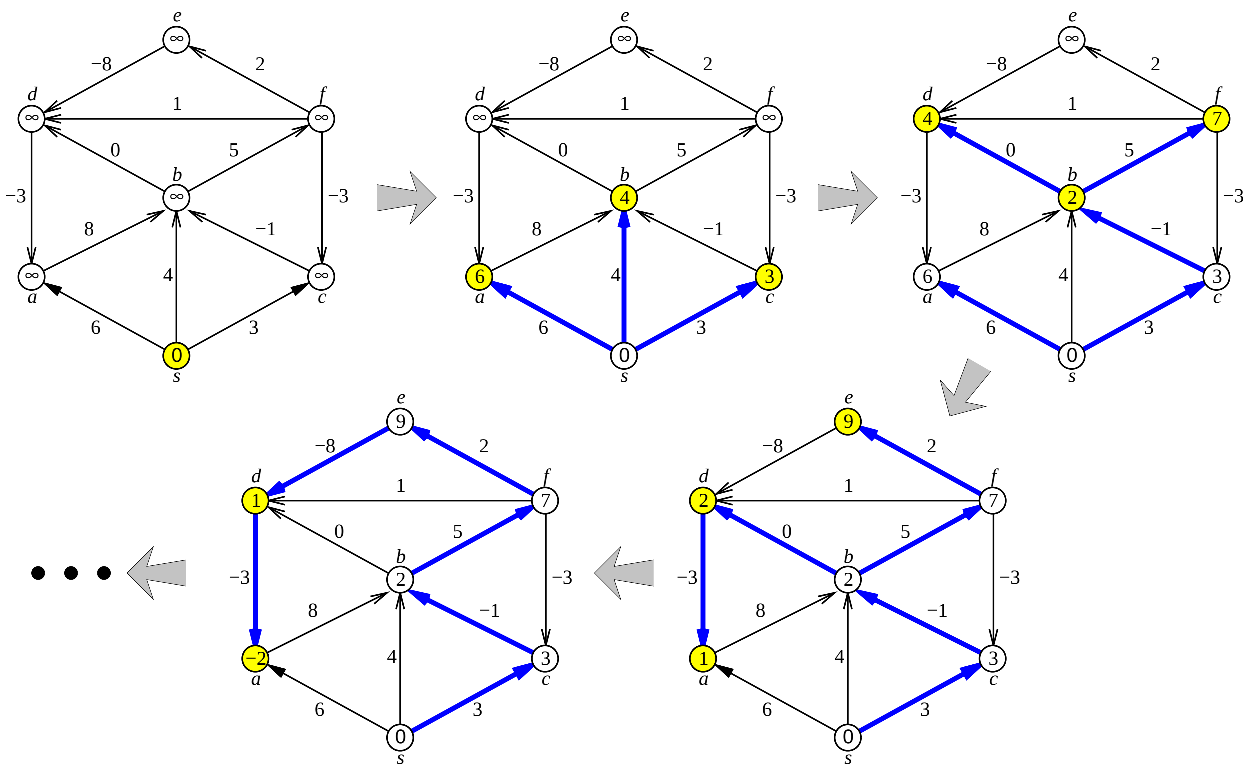
If we replace the heap in Dijkstra's algorithm with a queue, we get an algorithm that was first published by Shimbel in 1955, then independently rediscovered by Moore in 1957, by Woodbury and Dantzig in 1957, and by Bellman in 1958. Since Bellman (the Last Discoverer) used the idea of relaxing edges, which was first proposed by Ford in 1956, this is usually called the 'Bellman-Ford' algorithm. Shimbel's algorithm is efficient even if there are negative edges, and it can be used to quickly detect the presence of negative cycles. If there are no negative edges, however, Dijkstra's algorithm is faster. (In fact, in practice, Dijkstra's algorithm is often faster even for graphs with negative edges.)

The easiest way to analyze the algorithm is to break the execution into *phases*, by introducing an imaginary *token*. Before we even begin, we insert the token into the queue. The current phase ends when we take the token out of the queue; we begin the next phase by reinserting the token into the queue. The 0th phase consists entirely of scanning the source vertex s. The algorithm ends when the queue contains *only* the token. A simple inductive argument (hint, hint) implies the following invariant:

> At the end of the ith phase, for every vertex v, *dist*(v) is less than or equal to the length of the shortest path $s \rightsquigarrow v$ consisting of i or fewer edges.

Since a shortest path can only pass through each vertex once, either the algorithm halts before the Vth phase, or the graph contains a negative cycle. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the running time of a single phase is $O(E)$. Thus, the overall running time of Shimbel's algorithm is $\boxed{O(VE)}$.

[6] Some textbooks (like CLRS) present a version of Dijkstra's algorithm that gives incorrect results for graphs with negative edges.



Four phases of Shimbel's algorithm run on a directed graph with negative edges.
Nodes are taken from the queue in the order $s \diamond a\ b\ c \diamond d\ f\ b \diamond a\ e\ d \diamond d\ a \diamond \diamond$, where $\diamond$ is the token.
Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path tree.

Once we understand how the phases of Shimbel's algorithm behave, we can simplify the algorithm considerably. Instead of using a queue to perform a partial breadth-first search of the graph in each phase, we can simply scan through the adjacency list directly and try to relax every edge in the graph.

```
ShimbelSSSP($s$)
  InitSSSP($s$)
  repeat $V$ times:
      for every edge $u \to v$
          if $u \to v$ is tense
              Relax($u \to v$)

  for every edge $u \to v$
      if $u \to v$ is tense
          return 'Negative cycle!'
```

This is how most textbooks present the 'Bellman-Ford' algorithm.[7] The $O(VE)$ running time of this version of the algorithm should be obvious, but it may not be clear that the algorithm is still correct. To prove correctness, we just have to show that our earlier invariant holds; as before, this can be proved by induction on i.

[7] In fact, this is closer to the description that Shimbel and Bellman used. Bob Tarjan recognized in the early 1980s that Shimbel's algorithm is equivalent to Dijkstra's algorithm with a queue instead of a heap.

13.6 Greedy Relaxation?

Here's another algorithm that fits our generic framework, but which I've never seen analyzed.

$$\boxed{\text{Repeatedly relax the tensest edge.}}$$

Specifically, let's define the 'tension' of an edge $u \to v$ as follows:

$$\textit{tension}(u \to v) = \max\{0,\, \textit{dist}(v) - \textit{dist}(u) - w(u \to v)\}$$

(This is defined even when $\textit{dist}(v) = \infty$ or $\textit{dist}(u) = \infty$, as long as we treat ∞ just like some indescribably large but finite number.) If an edge has zero tension, it's not tense. If we relax an edge $u \to v$, then $\textit{dist}(v)$ decreases $\textit{tension}(u \to v)$ and $\textit{tension}(u \to v)$ becomes zero.

Intuitively, we can keep the edges of the graph in some sort of heap, where the key of each edge is its tension. Then we repeatedly pull out the tensest edge $u \to v$ and relax it. Then we need to recompute the tension of other edges adjacent to v. Edges leaving v possibly become more tense, and edges coming into v possibly become less tense. So we need a heap that efficiently supports the operations INSERT, EXTRACTMAX, INCREASEKEY, and DECREASEKEY.

If there are no negative cycles, this algorithm eventually halts with a shortest path tree, but how quickly? Can the same edge be relaxed more than once, and if so, how many times? Is it faster if all the edge weights are positive? Hmm.... This sounds like a good extra credit problem![8]

[8] I first proposed this bizarre algorithm in 1998, the very first time I taught an algorithms class. As far as I know, nobody has even seriously attempted an analysis. Or maybe it *has* been analyzed, but it requires an exponential (or even unbounded) number of relaxation steps in the worst case, so nobody's ever bothered to publish it.

> *The tree which fills the arms grew from the tiniest sprout;*
> *the tower of nine storeys rose from a (small) heap of earth;*
> *the journey of a thousand li commenced with a single step.*
>
> — Lao-Tzu, *Tao Te Ching*, chapter 64 (6th century BC),
> translated by J. Legge (1891)

> *And I would walk five hundred miles,*
> *And I would walk five hundred more,*
> *Just to be the man who walks a thousand miles*
> *To fall down at your door.*
>
> — The Proclaimers, "Five Hundred Miles (I'm Gonna Be)",
> from *Sunshine on Leith* (2001)

> *Almost there... Almost there...*
>
> — Red Leader [Drewe Henley], *Star Wars* (1977)

14 All-Pairs Shortest Paths

14.1 The Problem

In the previous lecture, we saw algorithms to find the shortest path from a source vertex s to a target vertex t in a directed graph. As it turns out, the best algorithms for this problem actually find the shortest path from s to every possible target (or from every possible source to t) by constructing a shortest path tree. The shortest path tree specifies two pieces of information for each node v in the graph:

- *dist*(v) is the length of the shortest path (if any) from s to v;
- *pred*(v) is the second-to-last vertex (if any) the shortest path (if any) from s to v.

In this lecture, we want to generalize the shortest path problem even further. In the *all pairs shortest path* problem, we want to find the shortest path from *every* possible source to *every* possible destination. Specifically, for every pair of vertices u and v, we need to compute the following information:

- *dist*(u, v) is the length of the shortest path (if any) from u to v;
- *pred*(u, v) is the second-to-last vertex (if any) on the shortest path (if any) from u to v.

For example, for any vertex v, we have *dist*$(v, v) = 0$ and *pred*$(v, v) = \text{NULL}$. If the shortest path from u to v is only one edge long, then *dist*$(u, v) = w(u \to v)$ and *pred*$(u, v) = u$. If there is *no* shortest path from u to v—either because there's no path at all, or because there's a negative cycle—then *dist*$(u, v) = \infty$ and *pred*$(v, v) = \text{NULL}$.

The output of our shortest path algorithms will be a pair of $V \times V$ arrays encoding all V^2 distances and predecessors. Many maps include a distance matrix—to find the distance from (say) Champaign to (say) Columbus, you would look in the row labeled 'Champaign' and the column labeled 'Columbus'. In these notes, I'll focus almost exclusively on computing the distance array. The predecessor array, from which you would compute the actual shortest paths, can be computed with only minor additions to the algorithms I'll describe (hint, hint).

14.2 Lots of Single Sources

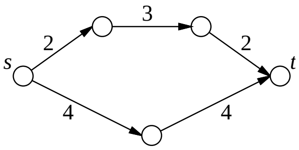
The obvious solution to the all-pairs shortest path problem is just to run a single-source shortest path algorithm V times, once for every possible source vertex! Specifically, to fill in the one-dimensional subarray *dist*$[s][\,]$, we invoke either Dijkstra's or Shimbel's algorithm starting at the source vertex s.

```
ObviousAPSP($V, E, w$):
  for every vertex $s$
      $dist[s][\,] \leftarrow \mathrm{SSSP}(V, E, w, s)$
```

The running time of this algorithm depends on which single-source shortest path algorithm we use. If we use Shimbel's algorithm, the overall running time is $\boldsymbol{\Theta(V^2E)} = O(V^4)$. If all the edge weights are non-negative, we can use Dijkstra's algorithm instead, which decreases the running time to $\boldsymbol{\Theta(VE + V^2 \log V)} = O(V^3)$. For graphs with negative edge weights, Dijkstra's algorithm can take exponential time, so we can't get this improvement directly.

14.3 Reweighting

One idea that occurs to most people is increasing the weights of all the edges by the same amount so that all the weights become positive, and then applying Dijkstra's algorithm. Unfortunately, this simple idea doesn't work. Different paths change by different amounts, which means the shortest paths in the reweighted graph may not be the same as in the original graph.



Increasing all the edge weights by 2 changes the shortest path s to t.

However, there is a more complicated method for reweighting the edges in a graph. Suppose each vertex v has some associated *cost* $c(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

$$w'(u \to v) = c(u) + w(u \to v) - c(v)$$

To give some intuition, imagine that when we leave vertex u, we have to pay an exit tax of $c(u)$, and when we enter v, we get $c(v)$ as an entrance gift.

Now it's not too hard to show that the shortest paths with the new weight function w' are exactly the same as the shortest paths with the original weight function w. In fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

$$w'(u \rightsquigarrow v) = c(u) + w(u \rightsquigarrow v) - c(v).$$

We pay $c(u)$ in exit fees, plus the original weight of of the path, minus the $c(v)$ entrance gift. At every intermediate vertex x on the path, we get $c(x)$ as an entrance gift, but then immediately pay it back as an exit tax!

14.4 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm finds a cost $c(v)$ for each vertex, so that when the graph is reweighted, every edge has non-negative weight.

Suppose the graph has a vertex s that has a path to every other vertex. Johnson's algorithm computes the shortest paths from s to every other vertex, using Shimbel's algorithm (which doesn't care if the edge weights are negative), and then sets

$$c(v) = \mathit{dist}(s, v),$$

so the new weight of every edge is

$$w'(u \to v) = \mathit{dist}(s, u) + w(u \to v) - \mathit{dist}(s, v).$$

Why are all these new weights non-negative? Because otherwise, Shimbel's algorithm wouldn't be finished! Recall that an edge $u \to v$ is *tense* if $\mathit{dist}(s, u) + w(u \to v) < \mathit{dist}(s, v)$, and that single-source shortest path algorithms eliminate all tense edges. The only exception is if the graph has a negative cycle, but then shortest paths aren't defined, and Johnson's algorithm simply aborts.

But what if the graph *doesn't* have a vertex s that can reach everything? No matter where we start Shimbel's algorithm, some of those vertex costs will be infinite. Johnson's algorithm avoids this problem by adding a new vertex s to the graph, with zero-weight edges going from s to every other vertex, but *no* edges going back into s. This addition doesn't change the shortest paths between any other pair of vertices, because there are no paths into s.

So here's Johnson's algorithm in all its glory.

```
JohnsonAPSP($V, E, w$) :
    create a new vertex $s$
    for every vertex $v \in V$
        $w(s \to v) \leftarrow 0$
        $w(v \to s) \leftarrow \infty$
    $dist[s][\,] \leftarrow$ Shimbel($V, E, w, s$)
    abort if Shimbel found a negative cycle
    for every edge $(u, v) \in E$
        $w'(u \to v) \leftarrow dist[s][u] + w(u \to v) - dist[s][v]$
    for every vertex $u \in V$
        $dist[u][\,] \leftarrow$ Dijkstra($V, E, w', u$)
        for every vertex $v \in V$
            $dist[u][v] \leftarrow dist[u][v] - dist[s][u] + dist[s][v]$
```

The algorithm spends $\Theta(V)$ time adding the artificial start vertex s, $\Theta(VE)$ time running Shimbel, $O(E)$ time reweighting the graph, and then $\Theta(VE + V^2 \log V)$ running V passes of Dijkstra's algorithm. Thus, the overall running time is $\boldsymbol{\Theta(VE + V^2 \log V)}$.

14.5 Dynamic Programming

There's a completely different solution to the all-pairs shortest path problem that uses dynamic programming instead of a single-source algorithm. For *dense* graphs where $E = \Omega(V^2)$, the dynamic programming approach eventually leads to the same $O(V^3)$ running time as Johnson's algorithm, but with a much simpler algorithm. In particular, the new algorithm avoids Dijkstra's algorithm, which gets its efficiency from Fibonacci heaps, which are rather easy to screw up in the implementation. **In the rest of this lecture, I will assume that the input graph contains no negative cycles.**

As usual for dynamic programming algorithms, we first need to come up with a recursive formulation of the problem. Here is an "obvious" recursive definition for *dist*(u, v):

$$\mathit{dist}(u,v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{x} \big(\mathit{dist}(u,x) + w(x \to v)\big) & \text{otherwise} \end{cases}$$

In other words, to find the shortest path from u to v, try all possible predecessors x, compute the shortest path from u to x, and then add the last edge $u \to v$. **Unfortunately, this recurrence doesn't work!** In order to compute *dist*(u, v), we first have to compute *dist*(u, x) for every other vertex x, but to compute any *dist*(u, x), we first need to compute *dist*(u, v). We're stuck in an infinite loop!

To avoid this circular dependency, we need an additional parameter that decreases at each recursion, eventually reaching zero at the base case. One possibility is to include the number of edges in the shortest path as this third magic parameter. So let's define *dist*(u, v, k) to be the length of the shortest path from u to v that uses *at most k edges*. Since we know that the shortest path between any two vertices has at most $V - 1$ vertices, what we're really trying to compute is *dist*$(u, v, V - 1)$.

After a little thought, we get the following recurrence.

$$\mathit{dist}(u,v,k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min\limits_{x} \big(\mathit{dist}(u,x,k-1) + w(x \to v)\big) & \text{otherwise} \end{cases}$$

Just like last time, the recurrence tries all possible predecessors of v in the shortest path, but now the recursion actually bottoms out when $k = 0$.

Now it's not difficult to turn this recurrence into a dynamic programming algorithm. Even before we write down the algorithm, though, we can tell that its running time will be $\boldsymbol{\Theta(V^4)}$ simply because recurrence has four variables—u, v, k, and x—each of which can take on V different values. Except for the base cases, the algorithm itself is just four nested for loops. To make the algorithm a little shorter, let's assume that $w(v \to v) = 0$ for every vertex v.

```
DYNAMICPROGRAMMINGAPSP($V, E, w$):
  for all vertices $u \in V$
      for all vertices $v \in V$
          if $u = v$
              $dist[u][v][0] \leftarrow 0$
          else
              $dist[u][v][0] \leftarrow \infty$

  for $k \leftarrow 1$ to $V - 1$
      for all vertices $u \in V$
          for all vertices $v \in V$
              $dist[u][v][k] \leftarrow \infty$
              for all vertices $x \in V$
                  if $dist[u][v][k] > dist[u][x][k-1] + w(x \to v)$
                      $dist[u][v][k] \leftarrow dist[u][x][k-1] + w(x \to v)$
```

The last four lines actually evaluate the recurrence.

In fact, this algorithm is almost exactly the same as running Shimbel's algorithm once for every source vertex. The only difference is the innermost loop, which in Shimbel's algorithm would read "for all edges $x \to v$". This simple change improves the running time to $\boldsymbol{\Theta(V^2E)}$, assuming the graph is stored in an adjacency list.

14.6 Divide and Conquer

But we can make a more significant improvement. The recurrence we just used broke the shortest path into a slightly shorter path and a single edge, by considering all predecessors. Instead, let's break it into two shorter paths at the middle vertex on the path. This idea gives us a different recurrence for $\mathit{dist}(u, v, k)$. Once again, to simplify things, let's assume $w(v \to v) = 0$.

$$\mathit{dist}(u,v,k) = \begin{cases} w(u \to v) & \text{if } k = 1 \\ \min\limits_{x} \big(\mathit{dist}(u,x,k/2) + \mathit{dist}(x,v,k/2)\big) & \text{otherwise} \end{cases}$$

This recurrence only works when k is a power of two, since otherwise we might try to find the shortest path with a fractional number of edges! But that's not really a problem, since $\mathit{dist}(u, v, 2^{\lceil \lg V \rceil})$ gives us the overall shortest distance from u to v. Notice that we use the base case $k = 1$ instead of $k = 0$, since we can't use half an edge.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\boldsymbol{\Theta(V^3 \log V)}$—we consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of k.

```
FastDynamicProgrammingAPSP($(V, E, w)$):
  for all vertices $u \in V$
      for all vertices $v \in V$
          $\mathit{dist}[u][v][0] \leftarrow w(u \to v)$
  for $i \leftarrow 1$ to $\lceil \lg V \rceil$          $\langle\langle k = 2^i \rangle\rangle$
      for all vertices $u \in V$
          for all vertices $v \in V$
              $\mathit{dist}[u][v][i] \leftarrow \infty$
              for all vertices $x \in V$
                  if $\mathit{dist}[u][v][i] > \mathit{dist}[u][x][i-1] + \mathit{dist}[x][v][i-1]$
                      $\mathit{dist}[u][v][i] \leftarrow \mathit{dist}[u][x][i-1] + \mathit{dist}[x][v][i-1]$
```

14.7 Aside: 'Funny' Matrix Multiplication

There is a very close connection (first observed by Shimbel, and later independently by Bellman) between computing shortest paths in a directed graph and computing powers of a square matrix. Compare the following algorithm for multiplying two $n \times n$ matrices A and B with the inner loop of our first dynamic programming algorithm. (I've changed the variable names in the second algorithm slightly to make the similarity clearer.)

```
MatrixMultiply($(A, B)$):
  for $i \leftarrow 1$ to $n$
      for $j \leftarrow 1$ to $n$
          $C[i][j] \leftarrow 0$
          for $k \leftarrow 1$ to $n$
              $C[i][j] \leftarrow C[i][j] + A[i][k] \cdot B[k][j]$
```

```
APSPInnerLoop:
  for all vertices $u$
      for all vertices $v$
          $D'[u][v] \leftarrow \infty$
          for all vertices $x$
              $D'[u][v] \leftarrow \min\{D'[u][v], D[u][x] + w[x][v]\}$
```

The *only* difference between these two algorithms is that we use addition instead of multiplication and minimization instead of addition. For this reason, the shortest path inner loop is often referred to as 'funny' matrix multiplication.

DynamicProgrammingAPSP is the standard iterative algorithm for computing the $(V-1)$th 'funny power' of the weight matrix w. The first set of for loops sets up the 'funny identity matrix', with zeros on the main diagonal and infinity everywhere else. Then each iteration of the second main for loop computes the next 'funny power'. FastDynamicProgrammingAPSP replaces this iterative method for computing powers with repeated squaring, exactly like we saw at the beginning of the semester. The fast algorithm is simplified slightly by the fact that unless there are negative cycles, every 'funny power' after the Vth is the same.

There are faster methods for multiplying matrices, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. (Google for 'Strassen's algorithm'.) Unfortunately, these algorithms us subtraction, and there's no 'funny' equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there seems to be no way to speed up the inner loop of our dynamic programming algorithms.

Fortunately, this isn't true. There is a beautiful randomized algorithm, due to Noga Alon, Zvi Galil, Oded Margalit*, and Moni Noar,[1] that computes all-pairs shortest paths in undirected graphs in $O(M(V)\log^2 V)$ expected time, where $M(V)$ is the time to multiply two $V \times V$ integer matrices. A simplified version of this algorithm for *unweighted* graphs was discovered by Raimund Seidel.[2]

14.8 Floyd and Warshall's Algorithm

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower than Johnson's algorithm. A different formulation due to Floyd and Warshall removes this logarithmic factor. Their insight was to use a different third parameter in the recurrence.

Number the vertices arbitrarily from 1 to V. For every pair of vertices u and v and every integer r, we define a path $\pi(u,v,r)$ as follows:

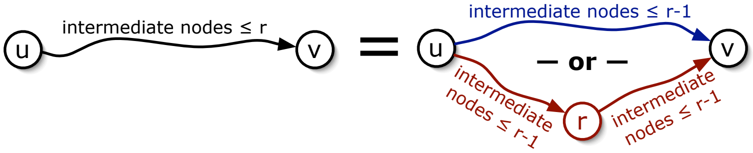
> $\pi(u,v,r) :=$ the shortest path from u to v where every intermediate vertex (that is, every vertex except u and v) is numbered at most r.

If $r = 0$, we aren't allowed to use any intermediate vertices, so $\pi(u,v,0)$ is just the edge (if any) from u to v. If $r > 0$, then either $\pi(u,v,r)$ goes through the vertex numbered r, or it doesn't. If $\pi(u,v,r)$ does contain vertex r, it splits into a subpath from u to r and a subpath from r to v, where every intermediate vertex in these two subpaths is numbered at most $r-1$. Moreover, the subpaths are as short as possible with this restriction, so they must be $\pi(u,r,r-1)$ and $\pi(r,v,r-1)$. On the other hand, if $\pi(u,v,r)$ does not go through vertex r, then every intermediate vertex in $\pi(u,v,r)$

[1] N. Alon, Z. Galil, O. Margalit*, and M. Naor. Witnesses for Boolean matrix multiplication and for shortest paths. *Proc. 33rd FOCS* 417-426, 1992. See also N. Alon, Z. Galil, O. Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

[2] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. This is one of the few algorithms papers where (in the conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper.

is numbered at most $r-1$; since $\pi(u,v,r)$ must be the *shortest* such path, we have $\pi(u,v,r) = \pi(u,v,r-1)$.



Recursive structure of the restricted shortest path $\pi(u,v,r)$.

This recursive structure implies the following recurrence for the length of $\pi(u,v,r)$, which we will denote by $\mathit{dist}(u,v,r)$:

$$\mathit{dist}(u,v,r) = \begin{cases} w(u\to v) & \text{if } r=0 \\ \min\left\{\mathit{dist}(u,v,r-1),\ \mathit{dist}(u,r,r-1)+\mathit{dist}(r,v,r-1)\right\} & \text{otherwise} \end{cases}$$

We need to compute the shortest path distance from u to v with no restrictions, which is just $\mathit{dist}(u,v,V)$.

Once again, we should immediately see that a dynamic programming algorithm that implements this recurrence will run in $\boldsymbol{\Theta(V^3)}$ time: three variables appear in the recurrence (u, v, and r), each of which can take on V possible values. Here's one way to do it:

```
FloydWarshall($V, E, w$):
  for $u \leftarrow 1$ to $V$
      for $v \leftarrow 1$ to $V$
          $dist[u][v][0] \leftarrow w(u \to v)$
  for $r \leftarrow 1$ to $V$
      for $u \leftarrow 1$ to $V$
          for $v \leftarrow 1$ to $V$
              if $dist[u][v][r-1] < dist[u][r][r-1] + dist[r][v][r-1]$
                  $dist[u][v][r] \leftarrow dist[u][v][r-1]$
              else
                  $dist[u][v][r] \leftarrow dist[u][r][r-1] + dist[r][v][r-1]$
```

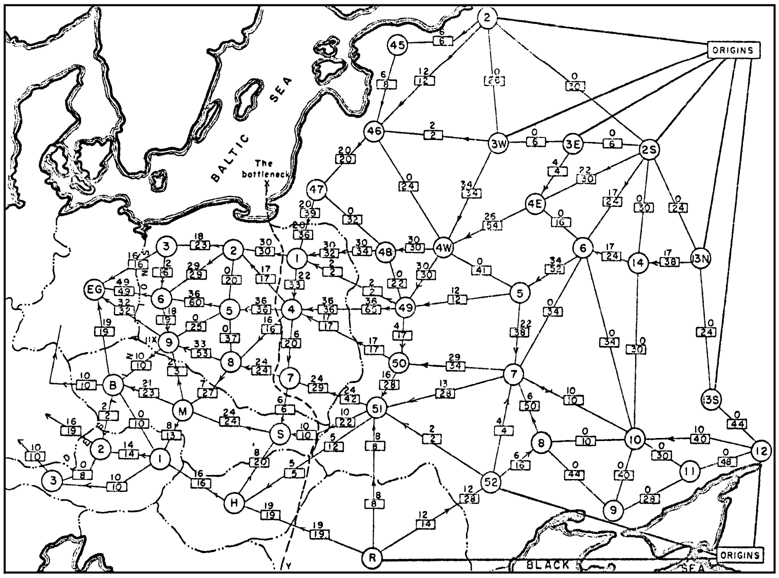
14.9 Homework

All of the algorithms discussed in this lecture fail if the graph contains a negative cycle. Johnson's algorithm detects the negative cycle in the initialization phase (via Shimbel's algorithm) and aborts; the dynamic programming algorithms just return incorrect results. **Describe how to modify these algorithms to return the correct shortest-path distances, even if the graph has negative cycles.** Specifically, if there is a path from vertex u to a negative cycle and a path from that negative cycle to vertex v, then the algorithm should report that $\mathit{dist}[u][v] = -\infty$. If there is no directed path from u to v, the algorithm should return $\mathit{dist}[u][v] = \infty$. Otherwise, $\mathit{dist}[u][v]$ should be the length of the shortest directed path from u to v.

> **Col. Klink:** *What are you waiting for? Cut the wire.*
> **Col. Hogan:** *That's the problem. One of these wires disconnects the fuse, the other one fires the bomb. Which one would you cut, Shultz?*
> **Sgt. Schultz:** *Don't ask me, this is a decision for an officer.*
> **Col. Hogan:** *All right. Which wire, Colonel Klink?*
> **Col. Klink:** *This one. [points to the white wire]*
> **Col. Hogan:** *You're sure?*
> **Col. Klink:** *Yes.*
> *[Hogan cuts the black wire; the bomb stops ticking]*
> **Col. Klink:** *If you knew which wire it was, why did you ask me?*
> **Col. Hogan:** *I wasn't sure which was the right one, but I was certain you'd pick the wrong one.*
>
> — "A Klink, a Bomb, and a Short Fuse", *Hogan's Heroes* (1966)

15 Maximum Flows and Minimum Cuts

In the mid-1950s, Air Force researchers T. E. Harris and F. S. Ross published a classified report studying the rail network that linked the Soviet Union to its satellite countries in Eastern Europe. The network was modeled as a graph with 44 vertices, representing geographic regions, and 105 edges, representing links between those regions in the rail network. Each edge was given a weight, representing the rate at which material could be shipped from one region to the next. Essentially by trial and error, they determined both the maximum amount of stuff that could be moved from Russia into Europe, as well as the cheapest way to disrupt the network by removing links (or in less abstract terms, blowing up train tracks), which they called 'the bottleneck'. Their results (including the figure at the top of the page) were only declassified in 1999.[1]



Harris and Ross's map of the Warsaw Pact rail network

This one of the first recorded applications of the *maximum flow* and *minimum cut* problems, which are defined as follows. Let $G = (V, E)$ be a directed graph, and let s and t be special vertices of G called the *source* and *target*. As in the previous lectures, I will use $u \rightarrow v$ to denote the directed edge from vertex u to vertex v.

[1] Both the map and the story were taken from Alexander Schrijver's fascinating survey 'On the history of combinatorial optimization (till 1960)'.

15.1 Flows

An ***(s, t)*-flow** (or just ***flow*** if the source and target are clear from context) is a function $f\colon E \to \mathbb{R}_{\geq 0}$ that satisfies the following ***balance constraint*** for all vertices v except possibly s and t:

$$\sum_u f(u \to v) = \sum_w f(v \to w).$$

In English, the total flow into any vertex is equal to the total flow out of that vertex. (To keep the notation simple, we assume here that $f(u \to v) = 0$ if there is no edge $u \to v$ in the graph.) The ***value*** of the flow f is defined as the excess flow out of the source vertex s:

$$\boxed{|f| = \sum_w f(s \to w) - \sum_u f(u \to s)}$$

It's not hard to prove that the value $|f|$ is also equal to the excess flow *into* the target vertex t. First we observe that

$$\sum_v \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) = \sum_v \sum_w f(v \to w) - \sum_v \sum_u f(u \to v) = 0$$

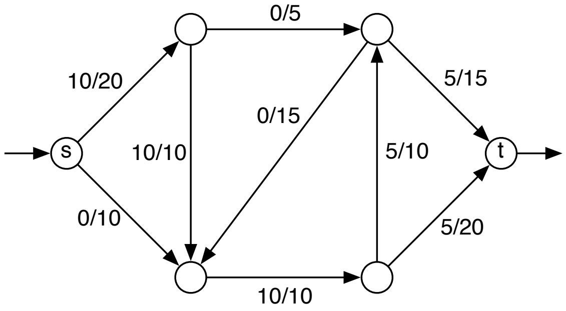
because both summations count the total flow across all edges. On the other hand, the balance constraint implies that

$$\begin{aligned} &\sum_v \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) \\ &\qquad = \left(\sum_w f(t \to w) - \sum_u f(u \to s) \right) + \left(\sum_w f(t \to w) - \sum_u f(u \to t) \right) \\ &\qquad = |f| + \left(\sum_w f(t \to w) - \sum_u f(u \to t) \right). \end{aligned}$$

It follows that

$$|f| = \sum_u f(u \to t) - \sum_w f(t \to w).$$

Now suppose we have another function $c\colon E \to \mathbb{R}_{\geq 0}$ that assigns a non-negative ***capacity*** $c(e)$ to each edge e. We say that a flow f is ***subject to c*** if $f(e) \leq c(e)$ for every edge e. Most of the time we will consider only flows that are subject to some fixed capacity function c. We say that a flow f ***saturates*** edge e if $f(e) = c(e)$, and ***avoids*** edge e if $f(e) = 0$. The ***maximum flow problem*** is to compute an (s, t)-flow in a given directed graph, subject to a given capacity function, whose value is as large as possible.



An (s, t)-flow with value 10. Each edge is labeled with its flow/capacity.

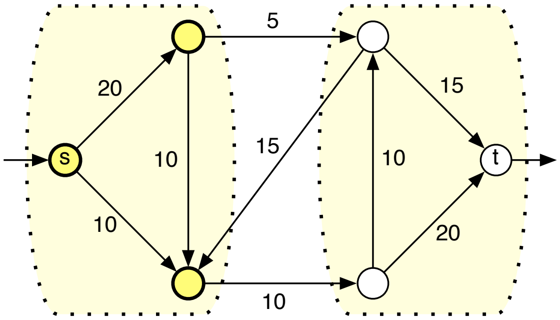
15.2 Cuts

An ***(s, t)*-*cut*** (or just ***cut*** if the source and target are clear from context) is a partition of the vertices into disjoint subsets S and T—meaning $S \cup T = V$ and $S \cap T = \varnothing$—where $s \in S$ and $t \in T$.

If we have a capacity function $c : E \to \mathbb{R}_{\geq 0}$, the ***cost*** of a cut is the sum of the capacities of the edges that start in S and end in T:

$$\boxed{\|S, T\| = \sum_{v \in S} \sum_{w \in T} c(v \to w).}$$

(Again, if $v \to w$ is not an edge in the graph, we assume $c(v \to w) = 0$.) Notice that the definition is asymmetric; edges that start in T and end in S are unimportant. The ***minimum cut problem*** is to compute an (s, t)-cut whose cost, relative to a given capacity function, is as large as possible.



An (s, t)-cut with cost 15. Each edge is labeled with its capacity.

Intuitively, the minimum cut is the cheapest way to disrupt all flow from s to t. Indeed, it is not hard to show that **the value of *any* (s, t)-flow subject to c is at most the cost of *any* (s, t)-cut.** Choose your favorite flow f and your favorite cut (S, T), and then follow the bouncing inequalities:

$$\begin{aligned}
|f| &= \sum_w f(s \to w) - \sum_u f(u \to s) && \text{by definition} \\
&= \sum_{v \in S} \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) && \text{by the balance constraint} \\
&= \sum_{v \in S} \left(\sum_{w \in T} f(v \to w) - \sum_{u \in T} f(u \to v) \right) && \text{removing duplicate edges} \\
&\leq \sum_{v \in S} \sum_{w \in T} f(v \to w) && \text{since } f(u \to v) \geq 0 \\
&\leq \sum_{v \in S} \sum_{w \in T} c(v \to w) && \text{since } f(u \to v) \leq c(v \to w) \\
&= \|S, T\| && \text{by definition}
\end{aligned}$$

Our derivation actually implies the following stronger observation: **$|f| = \|S, T\|$ if and only if f saturates every edge from S to T and avoids every edge from T to S.** Moreover, if we have a flow f and a cut (S, T) that satisfies this equality condition, f must be a maximum cut, and (S, T) must be a minimum flow.

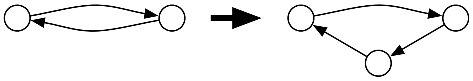
15.3 The Max-Flow Min-Cut Theorem

Surprisingly, for any weighted directed graph, there is always a flow f and a cut (S, T) that satisfy the equality condition. This is the famous *max-flow min-cut theorem*:

The value of the maximum flow is *equal* to the cost of the minimum cut.

The rest of this section gives a proof of this theorem; we will eventually turn this proof into an algorithm.

Fix a graph G, vertices s and t, and a capacity function $c : E \to \mathbb{R}_{\geq 0}$. The proof will be easier if we assume that the capacity function is ***reduced***: For any vertices u and v, either $c(u \to v) = 0$ or $c(v \to u) = 0$, or equivalently, if an edge appears in G, then its reversal does not. This assumption is easy to enforce. Whenever an edge $u \to v$ and its reversal $v \to u$ are both the graph, replace the edge $u \to v$ with a path $u \to x \to v$ of length two, where x is a new vertex and $c(u \to x) = c(x \to v) = c(u \to v)$. The modified graph has the same maximum flow value and minimum cut cost as the original graph.

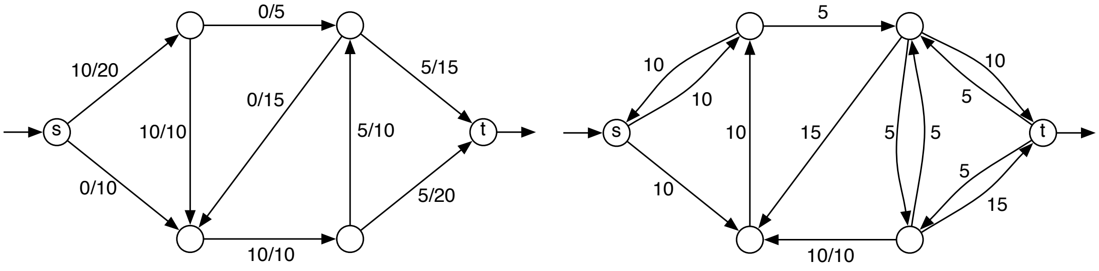


Enforcing the one-direction assumption.

Let f be a flow subject to c. We define a new capacity function $c_f : V \times V \to \mathbb{R}$, called the ***residual capacity***, as follows:

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}.$$

Since $f \geq 0$ and $f \leq c$, the residual capacities are always non-negative. It is possible to have $c_f(u \to v) > 0$ even if $u \to v$ is not an edge in the original graph G. Thus, we define the ***residual graph*** $G_f = (V, E_f)$, where E_f is the set of edges whose residual capacity is positive. Notice that the residual capacities are *not* necessarily reduced; it is quite possible to have both $c_f(u \to v) > 0$ and $c_f(v \to u) > 0$.

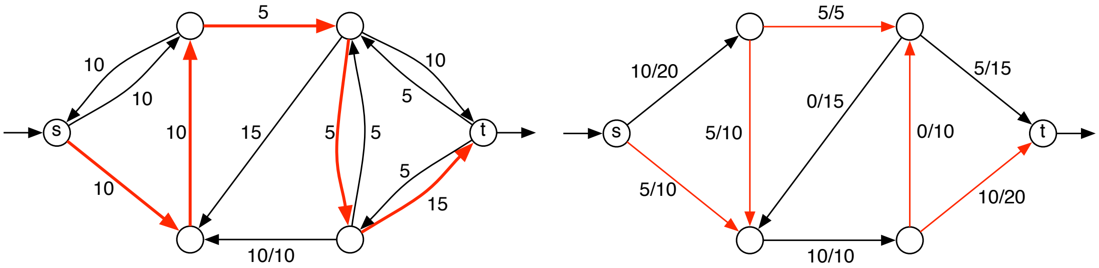


A flow f in a weighted graph G and the corresponding residual graph G_f.

Suppose there is no path from the source s to the target t in the residual graph G_f. Let S be the set of vertices that are reachable from s in G_f, and let $T = V \setminus S$. The partition (S, T) is clearly an (s, t)-cut. For every vertex $u \in S$ and $v \in T$, we have

$$c_f(u \to v) = (c(u \to v) - f(u \to v)) + f(v \to u) = 0,$$

which implies that $c(u \to v) - f(u \to v) = 0$ and $f(v \to u) = 0$. In other words, our flow f saturates every edge from S to T and avoids every edge from T to S. It follows that $|f| = \|S, T\|$. Moreover, f is a maximum flow and (S, T) is a minimum cut.



An augmenting path in G_f with value $F = 5$ and the augmented flow f'.

On the other hand, suppose there is a path $s = v_0 \to v_1 \to \cdots \to v_r = t$ in G_f. We refer to $v_0 \to v_1 \to \cdots \to v_r$ as an ***augmenting path***. Let $F = \min_i c_f(v_i \to v_{i+1})$ denote the maximum amount of flow that we can push through the augmenting path in G_f. We define a new flow function $f' : E \to \mathbb{R}$ as follows:

$$f'(u \to v) = \begin{cases} f(u \to v) + F & \text{if } u \to v \text{ is in the augmenting path} \\ f(u \to v) - F & \text{if } v \to u \text{ is in the augmenting path} \\ f(u \to v) & \text{otherwise} \end{cases}$$

To prove this is a legal flow function subject to the original capacities c, we need to verify that $f' \geq 0$ and $f' \leq c$. Consider an edge $u \to v$ in G. If $u \to v$ is in the augmenting path, then $f'(u \to v) > f(u \to v) \geq 0$ and

$$\begin{aligned} f'(u \to v) &= f(u \to v) + F && \text{by definition of } f' \\ &\leq f(u \to v) + c_f(u \to v) && \text{by definition of } F \\ &= f(u \to v) + c(u \to v) - f(u \to v) && \text{by definition of } c_f \\ &= c(u \to v) && \text{Duh.} \end{aligned}$$

On the other hand, if the reversal $v \to u$ is in the augmenting path, then $f'(u \to v) < f(u \to v) \leq c(u \to v)$ and

$$\begin{aligned} f'(u \to v) &= f(u \to v) - F && \text{by definition of } f' \\ &\geq f(u \to v) - c_f(v \to u) && \text{by definition of } F \\ &= f(u \to v) - f(u \to v) && \text{by definition of } c_f \\ &= 0 && \text{Duh.} \end{aligned}$$

Finally, we observe that (without loss of generality) only the first edge in the augmenting path leaves s, so $|f'| = |f| + F > 0$. In other words, f is *not* a maximum flow.

This completes the proof!

> *A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.*
>
> — The First Law of Mentat, in Frank Herbert's *Dune* (1965)

> *There's a difference between knowing the path and walking the path.*
>
> — Morpheus [Laurence Fishburne], *The Matrix* (1999)

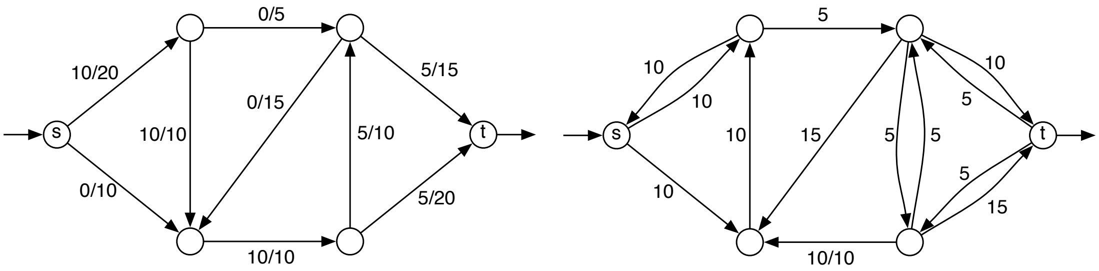
16 Max-Flow Algorithms and Applications

16.1 Recap

Fix a directed graph $G = (V, E)$ that does not contain both an edge $u \to v$ and its reversal $v \to u$, and fix a capacity function $c : E \to \mathbb{R}_{\geq 0}$. For any flow function $f : E \to \mathbb{R}_{\geq 0}$, the *residual capacity* is defined as

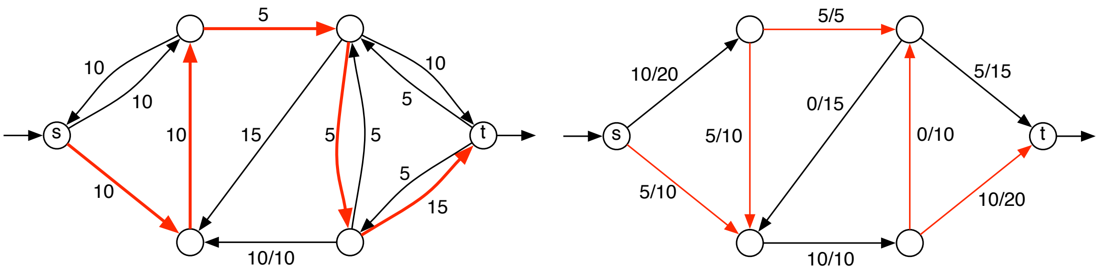
$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}.$$

The *residual graph* $G_f = (V, E_f)$, where E_f is the set of edges whose non-zero residual capacity is positive.



A flow f in a weighted graph G and its residual graph G_f.

In the last lecture, we proved the Max-flow Min-cut Theorem: *In any weighted directed graph network, the value of the maximum (s, t)-flow is equal to the cost of the minimum (s, t)-cut.* The proof of the theorem is constructive. If the residual graph contains a path from s to t, then we can increase the flow by the minimum capacity of the edges on this path, so we must not have the maximum flow. Otherwise, we can define a cut (S, T) whose cost is the same as the flow f, such that every edge from S to T is saturated and every edge from T to S is empty, which implies that f is a maximum flow and (S, T) is a minimum cut.



An augmenting path in G_f and the resulting (maximum) flow f'.

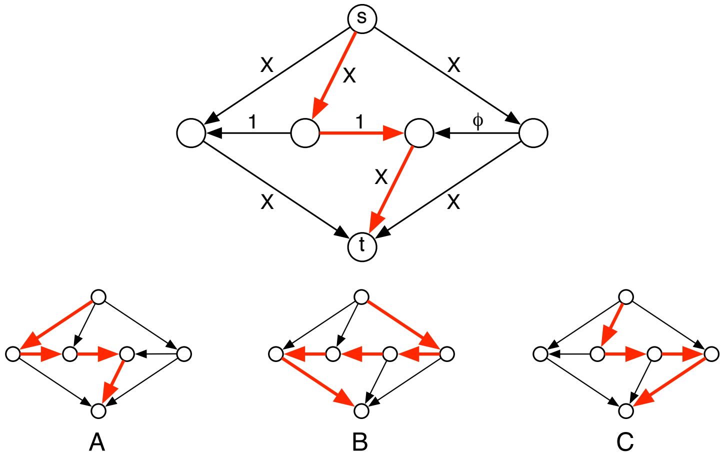
16.2 Ford-Fulkerson

It's not hard to realize that this proof translates almost immediately to an algorithm, first developed by Ford and Fulkerson in the 1950s: Starting with the zero flow, repeatedly augment the flow along **any** path $s \rightsquigarrow t$ in the residual graph, until there is no such path.

If every edge capacity is an integer, then every augmentation step increases the value of the flow by a positive integer. Thus, the algorithm halts after $|f^*|$ iterations, where f^* is the actual maximum flow. Each iteration requires $O(E)$ time, to create the residual graph G_f and perform a whatever-first-search to find an augmenting path. Thus, in the words case, the Ford-Fulkerson algorithm runs in $\boxed{O(E|f^*|)}$ time.

If we multiply all the capacities by the same (positive) constant, the maximum flow increases everywhere by the same constant factor. It follows that if all the edge capacities are *rational*, then the Ford-Fulkerson algorithm eventually halts. However, if we allow irrational capacities, the algorithm can loop forever, always finding smaller and smaller augmenting paths. Worse yet, this infinite sequence of augmentations may not even converge to the maximum flow! Perhaps the simplest example of this effect was discovered by Uri Zwick.

Consider the graph shown below, with six vertices and nine edges. Six of the edges have some large integer capacity X, two have capacity 1, and one has capacity $\phi = (\sqrt{5}-1)/2 \approx 0.618034$, chosen so that $1 - \phi = \phi^2$. To prove that the Ford-Fulkerson algorithm can get stuck, we can watch the residual capacities of the three horizontal edges as the algorithm progresses. (The residual capacities of the other six edges will always be at least $X - 3$.)



Uri Zwick's non-terminating flow example, and three augmenting paths.

The Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown in the large figure above. The three horizontal edges,, in order from left to right, now have residual capacities 1, 0, ϕ. Suppose inductively that the horizontal residual capacities are ϕ^{k-1}, 0, ϕ^k for some non-negative integer k.

1. Augment along B, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, \phi^k, 0$.
2. Augment along C, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^k$.
3. Augment along B, adding ϕ^{k+1} to the flow; the residual capacities are now $0, \phi^{k+1}, \phi^{k+2}$.
4. Augment along A, adding ϕ^{k+1} to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^{k+2}$.

Thus, after $4n+1$ augmentation steps, the residual capacities are $\phi^{2n-2}, 0, \phi^{2n-1}$. As the number of augmentation steps grows to infinity, the value of the flow converges to

$$1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5} < 7,$$

even though the maximum flow value is clearly $2X+1$.

Picky students might wonder at this point why we care about irrational capacities; after all, computers can't represent anything but (small) integers or (dyadic) rationals exactly. Good question! One reason is that the integer restriction is literally *artificial*; it's an *artifact* of actual computational hardware[1], not an inherent feature of the abstract mathematical problem. Another reason, which is probably more convincing to most practical computer scientists, is that the behavior of the algorithm with irrational inputs tells us something about its worst-case behavior *in practice* given floating-point capacities—terrible! Even with very reasonable capacities, a careless implementation of Ford-Fulkerson could enter an infinite loop simply because of round-off error!

16.3 Edmonds-Karp: Fat Pipes

The Ford-Fulkerson algorithm does not specify which alternating path to use if there is more than one. In 1972, Jack Edmonds and Richard Karp analyzed two natural heuristics for choosing the path. The first is essentially a greedy algorithm:

$$\boxed{\text{Choose the augmenting path with largest bottleneck value.}}$$

It's a fairly easy to show that the maximum-bottleneck (s,t)-path in a directed graph can be computed in $O(E \log V)$ time using a variant of Jarník's minimum-spanning-tree algorithm, or of Dijkstra's shortest path algorithm. Simply grow a directed spanning tree T, rooted at s. Repeatedly find the highest-capacity edge leaving T and add it to T, until T contains a path from s to t. Alternately, once could emulate Kruskal's algorithm—insert edges one at a time in decreasing capacity order until there is a path from s to t—although this is less efficient.

We can now analyze the algorithm in terms of the value of the maximum flow f^*. Let f be any flow in G, and let f' be the maximum flow *in the current residual graph* G_f. (At the beginning of the algorithm, $G_f = G$ and $f' = f^*$.) Let e be the bottleneck edge in the next augmenting path. Let S be the set of vertices reachable from s through edges with capacity greater than $c(e)$ and let $T = V \setminus S$. By construction, T is non-empty, and every edge from S to T has capacity at most $c(e)$. Thus, the cost of the cut (S,T) is at most $c(e) \cdot E$. On the other hand, $\|S,T\| \geq |f|$, which implies that $c(e) \geq |f|/E$.

Thus, augmenting f along the maximum-bottleneck path in G_f multiplies the maximum flow value in G_f by a factor of at most $1 - 1/E$. In other words, the residual flow *decays exponentially* with the number of iterations. After $E \cdot \ln|f^*|$ iterations, the maximum flow value in G_f is at most

$$|f^*| \cdot (1 - 1/E)^{E \cdot \ln|f^*|} < |f^*| e^{-\ln|f^*|} = 1.$$

(That's Euler's constant e, not the edge e. Sorry.) In particular, *if all the capacities are integers*, then after $E \cdot \ln|f^*|$ iterations, the maximum capacity of the residual graph is *zero* and f is a maximum flow.

We conclude that for graphs with integer capacities, the Edmonds-Karp 'fat pipe' algorithm runs in $\boxed{O(E^2 \log E \log|f^*|)}$ time.

[1] ...or perhaps the laws of physics. Yeah, right. Whatever. Like *reality* actually matters in this class.

16.4 Dinits/Edmonds-Karp: Short Pipes

The second Edmonds-Karp heuristic was actually proposed by Ford and Fulkerson in their original max-flow paper, and first analyzed by the Russian mathematician Dinits (sometimes transliterated Dinic) in 1970. Edmonds and Karp published their independent and slightly weaker analysis in 1972. So naturally, almost everyone refers to this algorithm as 'Edmonds-Karp'.[2]

Choose the augmenting path with fewest edges.

The correct path can be found in $O(E)$ time by running breadth-first search in the residual graph. More surprisingly, the algorithm halts after a polynomial number of iterations, independent of the actual edge capacities!

The proof of this upper bound relies on two observations about the evolution of the residual graph. Let f_i be the current flow after i augmentation steps, let G_i be the corresponding residual graph. In particular, f_0 is zero everywhere and $G_0 = G$. For each vertex v, let $\mathit{level}_i(v)$ denote the unweighted shortest path distance from s to v in G_i, or equivalently, the *level* of v in a breadth-first search tree of G_i rooted at s.

Our first observation is that these levels can only increase over time.

Lemma 1. *$\mathit{level}_{i+1}(v) \geq \mathit{level}_i(v)$ for all vertices v and integers i.*

Proof: The claim is trivial for $v = s$, since $\mathit{level}_i(s) = 0$ for all i. Choose an arbitrary vertex $v \neq s$, and let $s \to \cdots \to u \to v$ be a shortest path from s to v in G_{i+1}. (If there is no such path, then $\mathit{level}_{i+1}(v) = \infty$, and we're done.) Because this is a shortest path, we have $\mathit{level}_{i+1}(v) = \mathit{level}_{i+1}(u) + 1$, and the inductive hypothesis implies that $\mathit{level}_{i+1}(u) \geq \mathit{level}_i(u)$.

We now have two cases to consider. If $u \to v$ is an edge in G_i, then $\mathit{level}_i(v) \leq \mathit{level}_i(u) + 1$, because the levels are defined by breadth-first traversal.

On the other hand, if $u \to v$ is not an edge in G_i, then $v \to u$ must be an edge in the ith augmenting path. Thus, $v \to u$ must lie on the shortest path from s to t in G_i, which implies that $\mathit{level}_i(v) = \mathit{level}_i(u) - 1 \leq \mathit{level}_i(u) + 1$.

In both cases, we have $\mathit{level}_{i+1}(v) = \mathit{level}_{i+1}(u) + 1 \geq \mathit{level}_i(u) + 1 \geq \mathit{level}_i(v)$. □

Whenever we augment the flow, the bottleneck edge in the augmenting path disappears from the residual graph, and some other edge in the *reversal* of the augmenting path may (re-)appear. Our second observation is that an edge cannot appear or disappear too many times.

Lemma 2. *During the execution of the Dinits/Edmonds-Karp algorithm, any edge $u \to v$ disappears from the residual graph G_f at most $V/2$ times.*

Proof: Suppose $u \to v$ is in two residual graphs G_i and G_{j+1}, but not in any of the intermediate residual graphs $G_{i+1}, \ldots, G_j$, for some $i < j$. Then $u \to v$ must be in the ith augmenting path, so $\mathit{level}_i(v) = \mathit{level}_i(u) + 1$, and $v \to u$ must be on the jth augmenting path, so $\mathit{level}_j(v) = \mathit{level}_j(u) - 1$. By the previous lemma, we have

$$\mathit{level}_j(u) = \mathit{level}_j(v) + 1 \geq \mathit{level}_i(v) + 1 = \mathit{level}_i(u) + 2.$$

[2] To be fair, Edmonds and Karp discovered their algorithm a few years before publication—getting ideas into print takes time, especially in the early 1970s—which is why some authors believe they deserve priority. I don't buy it; Dinits *also* presumably discovered his algorithm a few years before *its* publication. (In Soviet Union, result publish you.) On the gripping hand, Dinits's paper also described an improvement to the algorithm presented here that runs in $O(V^2E)$ time instead of $O(VE^2)$, so maybe *that* ought to be called Dinits's algorithm.

In other words, the distance from s to u increased by at least 2 between the disappearance and reappearance of $u \to v$. Since every level is either less than V or infinite, the number of disappearances is at most $V/2$. □

Now we can derive an upper bound on the number of iterations. Since each edge can disappear at most $V/2$ times, there are at most $EV/2$ edge disappearances overall. But at least one edge disappears on each iteration, so the algorithm must halt after at most $EV/2$ iterations. Finally, since each iteration requires $O(E)$ time, Dinits' algorithm runs in $\boxed{O(VE^2)\text{ time}}$ overall.

16.5 Maximum Matchings in Bipartite Graphs

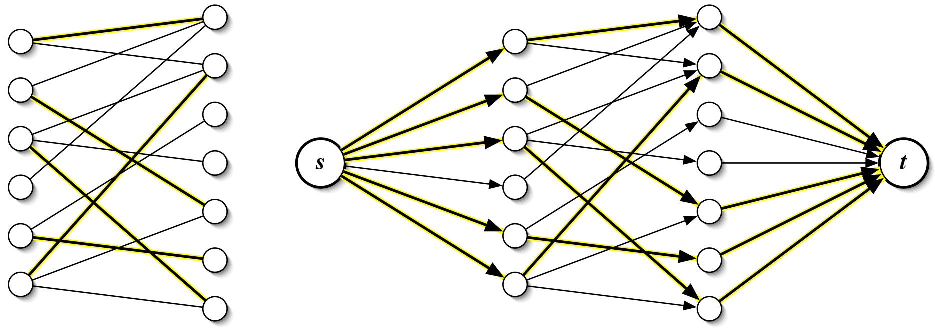
Perhaps one of the simplest applications of maximum flows is in computing a maximum-size *matching* in a bipartite graph. A matching is a subgraph in which every vertex has degree at most one, or equivalently, a collection of edges such that no two share a vertex. The problem is to find the largest matching in a given bipartite graph.

We can solve this problem by reducing it to a maximum flow problem as follows. Let G be the given bipartite graph with vertex set $V = U \cup W$, such that every edge joins a vertex in U to a vertex in W. We create a new *directed* graph G' by (1) orienting each edge from U to W, (2) adding two new vertices s and t, (3) adding edges from s to every vertex in U, and (4) adding edges from each vertex in W to t. Finally, we assign every edge in G' a capacity of 1.

Any matching M in G can be transformed into a flow f_M in G' as follows: For each edge (u, w) in M, push one unit of flow along the path $s \to u \to w \to t$. These paths are disjoint except at s and t, so the resulting flow satisfies the capacity constraints. Moreover, the value of the resulting flow is equal to the number of edges in M.

Conversely, consider any (s, t)-flow f in G' computed using the Ford-Fulkerson augmenting path algorithm. Because the edge capacities are integers, the Ford-Fulkerson algorithm assigns an integer flow to every edge. (This is easy to verify by induction hint hint.) Moreover, since each edge has *unit* capacity, the computed flow either saturates ($f(e) = 1$) or avoids ($f(e) = 0$) every edge in G'. Finally, since at most one unit of flow can enter any vertex in U or leave any vertex in W, the saturated edges from U to W form a matching in G. The size of this matching is exactly $|f|$.

Thus, the size of the maximum matching in G is equal to the value of the maximum flow in G', and provided we compute the maxflow using augmenting paths, we can convert the actual maxflow into a maximum matching. The maximum flow has value at most $\min\{|U|, |W|\} = O(V)$, so the Ford-Fulkerson algorithm runs in $\boxed{O(VE)\text{ time}}$.



A maximum matching in a bipartite graph G, and the corresponding maximum flow in G'

16.6 Edge-Disjoint Paths

Similarly, we can compute the maximum number of edge-disjoint paths between two vertices s and t in a graph using maximum flows. A set of paths in G is *edge-disjoint* if each edge in G appears in at most one of the paths. (Several edge-disjoint paths may pass through the same vertex, however.)

If we give each edge capacity 1, then the maxflow from s to t assigns a flow of either 0 or 1 to every edge. Moreover, even if the original graph is undirected, the maxflow algorithm will assign a direction to every saturated edge. Thus, the subgraph S of saturated edges is the union of several edge-disjoint paths; the number of paths is equal to the value of the flow. Extracting the actual paths from S is easy: Just follow any directed path in S from s to t, remove that path from S, and recurse. The overall running time is $O(VE)$, just like for maximum bipartite matchings.

Conversely, we can transform any collection of edge-disjoint paths into a flow by pushing one unit of flow along each path from s to t; the value of the resulting flow is equal to the number of paths in the collection. It follows that the maxflow algorithm actually computes the largest possible set of edge-disjoint paths.

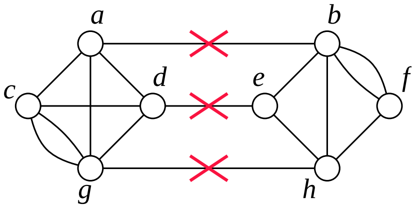
> **Jaques:** *But, for the seventh cause; how did you find the quarrel on the seventh cause?*
> **Touchstone:** *Upon a lie seven times removed:–bear your body more seeming, Audrey:–as thus, sir. I did dislike the cut of a certain courtier's beard: he sent me word, if I said his beard was not cut well, he was in the mind it was: this is called the Retort Courteous. If I sent him word again 'it was not well cut,' he would send me word, he cut it to please himself: this is called the Quip Modest. If again 'it was not well cut,' he disabled my judgment: this is called the Reply Churlish. If again 'it was not well cut,' he would answer, I spake not true: this is called the Reproof Valiant. If again 'it was not well cut,' he would say I lied: this is called the Counter-cheque Quarrelsome: and so to the Lie Circumstantial and the Lie Direct.*
> **Jaques:** *And how oft did you say his beard was not well cut?*
> **Touchstone:** *I durst go no further than the Lie Circumstantial, nor he durst not give me the Lie Direct; and so we measured swords and parted.*
>
> — William Shakespeare, *As You Like It* Act V, Scene 4 (1600)

C Randomized Minimum Cut

C.1 Setting Up the Problem

This lecture considers a problem that arises in robust network design. Suppose we have a connected multigraph[1] G representing a communications network like the UIUC telephone system, the internet, or Al-Qaeda. In order to disrupt the network, an enemy agent plans to remove some of the edges in this multigraph (by cutting wires, placing police at strategic drop-off points, or paying street urchins to 'lose' messages) to separate it into multiple components. Since his country is currently having an economic crisis, the agent wants to remove as few edges as possible to accomplish this task.

More formally, a *cut* partitions the nodes of G into two nonempty subsets. The *size* of the cut is the number of *crossing edges,* which have one endpoint in each subset. Finally, a *minimum* cut in G is a cut with the smallest number of crossing edges. The same graph may have several minimum cuts.



A multigraph whose minimum cut has three edges.

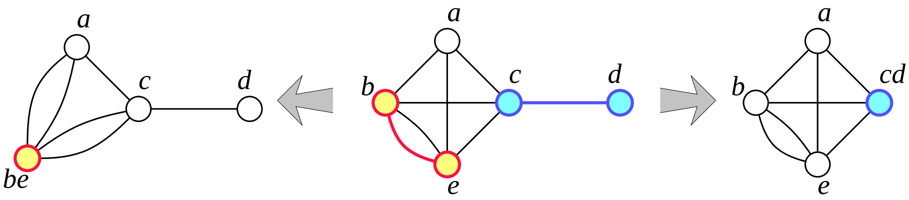
This problem has a long history. The classical deterministic algorithms for this problem rely on *network flow* techniques, which are discussed in another lecture. The fastest such algorithms run in $O(n^3)$ time and are quite complex and difficult to understand (unless you're already familiar with network flows). Here I'll describe a relatively simple randomized algorithm published by David Karger[2], who was a Ph.D. student at Stanford at the time.

Karger's algorithm uses a primitive operation called *collapsing an edge*. Suppose u and v are vertices that are connected by an edge in some multigraph G. To collapse the edge $\{u, v\}$, we

[1] A multigraph allows multiple edges between the same pair of nodes. Everything in this lecture could be rephrased in terms of simple graphs where every edge has a non-negative weight, but this would make the algorithms and analysis slightly more complicated.

[2] David R. Karger*. Random sampling in cut, flow, and network design problems. Proc. 25th STOC, 648–657, 1994.

create a new node called uv, replace any edge of the form u, w or v, w with a new edge uv, w, and then delete the original vertices u and v. Equivalently, collapsing the edge shrinks the edge down to nothing, pulling the two endpoints together. The new collapsed graph is denoted $G/\{u,v\}$. We don't allow self-loops in our multigraphs; if there are multiple edges between u and v, collapsing any one of them deletes them all.



A graph G and two collapsed graphs $G/\{b,e\}$ and $G/\{c,d\}$.

I won't describe how to actually implement collapsing an edge—it will be a homework exercise later in the course—but it can be done in $O(n)$ time. Let's just accept collapsing as a black box subroutine for now.

The correctness of our algorithms will eventually boil down the following simple observation: For any cut in $G/\{u,v\}$, there is cut in G with exactly the same number of crossing edges. In fact, in some sense, the 'same' edges form the cut in both graphs. The converse is not necessarily true, however. For example, in the picture above, the original graph G has a cut of size 1, but the collapsed graph $G/\{c,d\}$ does not.

This simple observation has two immediate but important consequences. First, collapsing an edge cannot decrease the minimum cut size. More importantly, collapsing an edge increases the minimum cut size if and only if that edge is part of *every* minimum cut.

C.2 Blindly Guessing

Let's start with an algorithm that tries to *guess* the minimum cut by randomly collapsing edges until the graph has only two vertices left.

```
GUESSMINCUT($G$):
  for $i \leftarrow n$ downto 2
      pick a random edge $e$ in $G$
      $G \leftarrow G/e$
  return the only cut in $G$
```

Since each collapse takes $O(n)$ time, this algorithm runs in $O(n^2)$ time. Our earlier observations imply that as long as we never collapse an edge that lies in every minimum cut, our algorithm will actually guess correctly. But how likely is that?

Suppose G has only one minimum cut—if it actually has more than one, just pick your favorite—and this cut has size k. Every vertex of G must lie on at least k edges; otherwise, we could separate that vertex from the rest of the graph with an even smaller cut. Thus, the number of incident vertex-edge pairs is at least kn. Since every edge is incident to exactly two vertices, G must have at least $kn/2$ edges. That implies that if we pick an edge in G uniformly at random, the probability of picking an edge in the minimum cut is at most $2/n$. In other words, the probability that we don't screw up on the very first step is at least $1 - 2/n$.

Once we've collapsed the first random edge, the rest of the algorithm proceeds recursively (with independent random choices) on the remaining $(n-1)$-node graph. So the overall probability $P(n)$ that GUESSMINCUT returns the true minimum cut is given by the following recurrence:

$$P(n) \geq \frac{n-2}{n} \cdot P(n-1).$$

The base case for this recurrence is $P(2) = 1$. We can immediately expand this recurrence into a product, most of whose factors cancel out immediately.

$$P(n) \geq \prod_{i=3}^{n} \frac{i-2}{i} \;=\; \frac{\prod_{i=3}^{n}(i-2)}{\prod_{i=3}^{n} i} = \frac{\prod_{i=1}^{n-2} i}{\prod_{i=3}^{n} i} = \boxed{\frac{2}{n(n-1)}}$$

C.3 Blindly Guessing Over and Over

That's not very good. Fortunately, there's a simple method for increasing our chances of finding the minimum cut: run the guessing algorithm many times and return the smallest guess. Randomized algorithms folks like to call this idea *amplification*.

```
KARGERMINCUT($G$):
  $mink \leftarrow \infty$
  for $i \leftarrow 1$ to $N$
      $X \leftarrow$ GUESSMINCUT($G$)
      if $|X| < mink$
          $mink \leftarrow |X|$
          $minX \leftarrow X$
  return $minX$
```

Both the running time and the probability of success will depend on the number of iterations N, which we haven't specified yet.

First let's figure out the probability that KARGERMINCUT returns the actual minimum cut. The only way for the algorithm to return the wrong answer is if GUESSMINCUT fails N times in a row. Since each guess is independent, our probability of success is at least

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^N.$$

We can simplify this using one of the most important (and easiest) inequalities known to mankind:

$$\boxed{\boxed{1 - x \leq e^{-x}}}$$

So our success probability is at least

$$1 - e^{-2N/n(n-1)}.$$

By making N larger, we can make this probability arbitrarily close to 1, but never equal to 1. In particular, if we set $\boxed{N = c\binom{n}{2}\ln n}$ for some constant c, then KARGERMINCUT is correct with probability at least

$$1 - e^{-c \ln n} = 1 - \frac{1}{n^c}.$$

When the failure probability is a polynomial fraction, we say that the algorithm is correct *with high probability*. Thus, KARGERMINCUT computes the minimum cut of any n-node graph in $\boxed{O(n^4 \log n)}$ time.

If we make the number of iterations even larger, say $N = n^2(n-1)/2$, the success probability becomes $1 - e^{-n}$. When the failure probability is exponentially small like this, we say that the algorithm is correct with *very* high probability. In practice, very high probability is usually overkill; high probability is enough. (Remember, there is a small but non-zero probability that your computer will transform itself into a kitten before your program is finished.)

C.4 Not-So-Blindly Guessing

The $O(n^4 \log n)$ running time is actually comparable to some of the simpler flow-based algorithms, but it's nothing to get excited about. But we can improve our guessing algorithm, and thus decrease the number of iterations in the outer loop, by observing that *as the graph shrinks, the probability of collapsing an edge in the minimum cut increases*. At first the probability is quite small, only $2/n$, but near the end of execution, when the graph has only three vertices, we have a $2/3$ chance of screwing up!

A simple technique for working around this increasing probability of error was developed by David Karger and Cliff Stein.[3] Their idea is to group the first several random collapses a 'safe' phase, so that the cumulative probability of screwing up is small—less than 1/2, say—and a 'dangerous' phase, which is much more likely to screw up.

The safe phase shrinks the graph from n nodes to $n/\sqrt{2}+1$ nodes, using a sequence of $n - n/\sqrt{2} - 1$ random collapses. Following our earlier analysis, the probability that *none* of these safe collapses touches the minimum cut is at least

$$\prod_{i=n/\sqrt{2}+2}^{n} \frac{i-2}{i} = \frac{(n/\sqrt{2})(n/\sqrt{2}+1)}{n(n-1)} = \frac{n+\sqrt{2}}{2(n-1)} > \frac{1}{2}.$$

Now, to get around the danger of the dangerous phase, we use amplification. However, instead of running through the dangerous phase once, we run it *twice* and keep the best of the two answers. Naturally, we treat the dangerous phase recursively, so we actually obtain a binary recursion tree, which expands as we get closer to the base case, instead of a single path. More formally, the algorithm looks like this:

```
CONTRACT($G, m$):
    for $i \leftarrow n$ downto $m$
        pick a random edge $e$ in $G$
        $G \leftarrow G/e$
    return $G$
```

```
BETTERGUESS($G$):
    if $G$ has more than $8$ vertices
        $X_1 \leftarrow$ BETTERGUESS(CONTRACT($G, n/\sqrt{2}+1$))
        $X_2 \leftarrow$ BETTERGUESS(CONTRACT($G, n/\sqrt{2}+1$))
        return $\min\{X_1, X_2\}$
    else
        use brute force
```

This might look like we're just doing to same thing twice, but remember that CONTRACT (and thus BETTERGUESS) is randomized. Each call to CONTRACT contracts an independent random set of edges; X_1 and X_2 are almost always different cuts.

BETTERGUESS correctly returns the minimum cut unless *both* recursive calls return the wrong result. X_1 is the minimum cut of G if and only if (1) none of the min cut edges are CONTRACTed

[3] David R. Karger* and Cliff Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. Proc. 25th STOC, 757–765, 1993.

and (2) the recursive BETTERGUESS returns the minimum cut of the CONTRACTed graph. If $P(n)$ denotes the probability that BETTERGUESS returns a minimum cut of an n-node graph, then X_1 is the minimum cut with probability at least $1/2 \cdot P(n/\sqrt{2})$, and X_2 is the minimum cut with the same probability. Since these two events are independent, we have the following recurrence, with base case $P(n) = 1$ for all $n \leq 6$.

$$P(n) \geq 1 - \left(1 - \frac{1}{2}\, P\!\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Using a series of transformations, Karger and Stein prove that $P(n) = \Omega(1/\log n)$. I've included the proof at the end of this note.

For the running time, we get a simple recurrence that is easily solved using recursion trees or the Master theorem (after a domain transformation to remove the $+1$ from the recurrence).

$$T(n) = O(n^2) + 2T\!\left(\frac{n}{\sqrt{2}} + 1\right) = \boxed{O(n^2 \log n)}$$

So all this splitting and recursing has slowed down the guessing algorithm slightly, but the probability of failure is *exponentially* smaller!

Let's express the lower bound $P(n) = \Omega(1/\log n)$ explicitly as $P(n) \geq \alpha/\ln n$ for some constant α. (Karger and Klein's proof implies $\alpha > 2$). If we call BETTERGUESS $N = c \ln^2 n$ times, for some new constant c, the overall probability of success is at least

$$1 - \left(1 - \frac{\alpha}{\ln n}\right)^{c \ln^2 n} \geq 1 - e^{-(c/\alpha)\ln n} = 1 - \frac{1}{n^{c/\alpha}}.$$

By setting c sufficiently large, we can bound the probability of failure by an arbitrarily small polynomial function of n. In other words, we now have an algorithm that computes the minimum cut with high probability in only $\boxed{O(n^2 \log^3 n)}$ time!

*C.5 Solving the Karger/Stein recurrence

Recall the following recurrence for the probability that BETTERGUESS successfully finds a minimum cut of an n-node graph:

$$P(n) \geq 1 - \left(1 - \frac{1}{2}\, P\!\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Karger and Stein solve this rather ugly recurrence through a series of functional transformations. Let $p(k)$ denote the probability of success at the kth level of recursion, counting upward from the base case. This function satisfies the recurrence

$$p(k) \geq 1 - \left(1 - \frac{p(k-1)}{2}\right)^2$$

with base case $p(0) = 1$. Let $\bar{p}(k)$ be the function that satisfies this recurrence with equality; clearly, $p(k) \geq \bar{p}(k)$. Now consider the function $z(k) = 4/\bar{p}(k) - 1$. Substituting $\bar{p}(k) = 4/(z(k) + 1)$ into our old recurrence implies (after a bit of algebra) that

$$z(k) = z(k-1) + 2 + \frac{1}{z(k-1)}.$$

Since clearly $z(k) > 1$ for all k, we have a conservative upper bound

$$z(k) < z(k-1) + 2,$$

which implies inductively that $z(k) \le 2k + 3$, since $z(0) = 3$. It follows that

$$p(k) \ge \bar{p}(k) > \frac{1}{2k+4} = \Omega(1/k).$$

To compute the number of levels of recursion that BetterGuess executes for an n-node graph, we solve the secondary recurrence

$$k(n) = 1 + k\left(\frac{n}{\sqrt{2}} + 1\right)$$

with base cases $k(n) = 0$ for all $n \le 8$. After a domain transformation to remove the $+1$ from the right side, the recursion tree method (or the Master theorem) implies that $k(n) = \Theta(\log n)$.

We conclude that $\boxed{P(n) = p(k(n)) = \Omega(1/\log n)}$, as promised.

> *And it's one, two, three,*
> *What are we fighting for?*
> *Don't tell me, I don't give a damn,*
> *Next stop is Vietnam; [or: This time we'll kill Saddam]*
> *And it's five, six, seven,*
> *Open up the pearly gates,*
> *Well there ain't no time to wonder why,*
> *Whoopee! We're all going to die.*
>
> — Country Joe and the Fish
> "I-Feel-Like-I'm-Fixin'-to-Die Rag" (1967)

> *There are 0 kinds of mathematicians:*
> *Those who can count modulo 2 and those who can't.*
>
> — Anonymous

> *God created the integers; all the rest is the work of man.*[a]
>
> — Kronecker
>
> [a] Kronecker was mistaken. God created the induction fairy, who created the integers, along with everything else.

D Number Theoretic Algorithms (November 12 and 14)

D.1 Greatest Common Divisors

Before we get to any actual algorithms, we need some definitions and preliminary results. **Unless specifically indicated otherwise, *all* variables in this lecture are integers.**

The symbol $\mathbb{Z}$ (from the German word "Zahlen", meaning 'numbers' or 'to count') to denote the set of integers. We say that one integer d *divides* another integer n, or that d is a *divisor* of n, if the quotient n/d is also an integer. Symbolically, we can write this definition as follows:

$$d \mid n \iff \left\lfloor \frac{n}{d} \right\rfloor = \frac{n}{d}$$

In particular, zero is not a divisor of any integer—∞ is *not* an integer—but every other integer is a divisor of zero. If d and n are positive, then $d \mid n$ immediately implies that $d \leq n$.

Any integer n can be written in the form $n = qd + r$ for some non-negative integer $0 \leq r \leq |d-1|$. Moreover, the choices for the quotient q and remainder r are unique:

$$q = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{and} \qquad r = n \bmod d = n - d \left\lfloor \frac{n}{d} \right\rfloor.$$

Note that the remainder $n \bmod d$ is *always* non-negative, even if $n < 0$ or $d < 0$ or both.[1]

If d divides two integers m and n, we say that d is a *common divisor* of m and n. It's trivial to prove (by definition crunching) that any common divisor of m and n also divides any integer linear combination of m and n:

$$(d \mid m) \text{ and } (d \mid n) \implies d \mid (am + bn)$$

The *greatest common divisor* of m and n, written $\gcd(m, n)$,[2] is the largest integer that divides both m and n. Sometimes this is also called the greater common *denominator*. The greatest common divisor has another useful characterization as the *smallest* element of another set.

[1] The sign rules for the C/C++/Java % operator are just plain stupid. I can't count the number of times I've had to write `x = (x+n)%n;` instead of `x %= n;`. Frickin' *idiots*. Gah!

[2] Do *not* use the notation (m, n) for greatest common divisor. *Ever*.

Lemma 1. *$gcd(m,n)$ is the smallest positive integer of the form $am + bn$.*

Proof: Let s be the smallest positive integer of the form $am + bn$. Any common divisor of m and n is also a divisor of $s = am + bn$. In particular, $\gcd(m, n)$ is a divisor of s, which implies that $\boxed{\gcd(m,n) \leq s}$.

To prove the other inequality, let's show that $s \mid m$ by calculating $m \bmod s$.

$$m \bmod s = m - s \left\lfloor \frac{m}{s} \right\rfloor = m - (am + bn) \left\lfloor \frac{m}{s} \right\rfloor = m \left(1 - a \left\lfloor \frac{m}{s} \right\rfloor\right) + n \left(-b \left\lfloor \frac{m}{s} \right\rfloor\right)$$

We observe that $m \bmod s$ is an integer linear combination of m and n. Since $m \bmod s < s$, and s is the smallest *positive* integer linear combination, $m \bmod s$ cannot be positive. So it must be zero, which implies that $s \mid m$, as we claimed. By a symmetric argument, $s \mid n$. Thus, s is a common divisor of m and n. A common divisor can't be greater than the *greatest* common divisor, so $\boxed{s \leq \gcd(m,n)}$.

These two inequalities imply that $s = \gcd(m, n)$, completing the proof. □

D.2 Euclid's GCD Algorithm

We can compute the greatest common divisor of two given integers by recursively applying two simple observations:

$$\gcd(m, n) = \gcd(m, n - m) \qquad \text{and} \qquad \gcd(n, 0) = n$$

The following algorithm uses the first observation to reduce the input and recurse; the second observation provides the base case.

```
SLOWGCD($m, n$):
  $m \leftarrow |m|$; $n \leftarrow |n|$
  if $m < n$
      swap $m \leftrightarrow n$
  while $n > 0$
      $m \leftarrow m - n$
      if $m < n$
          swap $m \leftrightarrow n$
  return $m$
```

The first few lines just ensure that $m \geq n \geq 0$. Each iteration of the main loop decreases one of the numbers by at least 1, so the running time is $O(m + n)$. This bound is tight in the worst case; consider the case $n = 1$. Unfortunately, this is terrible. The input consists of just $\log m + \log n$ bits; as a function of the input size, this algorithm runs in *exponential* time.

Let's think for a moment about what the main loop computes between swaps. We start with two numbers m and n and repeatedly subtract n from m until we can't any more. This is just a (slow) recipe for computing $m \bmod n$! That means we can speed up the algorithm by using mod instead of subtraction.

```
EUCLIDGCD($m, n$):
  $m \leftarrow |m|$; $n \leftarrow |n|$
  if $m < n$
      swap $m \leftrightarrow n$
  while $n > 0$
      $m \leftarrow m \bmod n$    $(\star)$
      swap $m \leftrightarrow n$
  return $m$
```

This algorithm swaps m and n at *every* iteration, because $m \bmod n$ is always less than n. This is almost universally called *Euclid's algorithm*, because the main idea is included in Euclid's *Elements*.[3]

The easiest way to analyze this algorithm is to work backward. First, let's consider the number of iterations of the main loop, or equivalently, the number times line ($\star$) is executed. To keep things simple, let's assume that $m > n > 0$, so the first three lines are redundant, and the algorithm performs at least one iteration. Recall that the Fibonacci numbers(!) are defined as $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for all $k > 1$.

Lemma 2. *If the algorithm performs k iterations, then $m \geq F_{k+2}$ and $n \geq F_{k+1}$.*

Proof (by induction on k): If $k = 1$, we have the trivial bounds $n \geq 1 = F_2$ and $m \geq 2 = F_3$.

Suppose $k > 1$. The first iteration of the loop replaces (m, n) with $(n, m \bmod n)$. The algorithm performs $k - 1$ more iterations, so the inductive hypothesis implies that $n \geq F_{k+1}$ and $m \bmod n \geq F_k$. We've assumed that $m > n$, so $m \geq m + n(1 - \lfloor m/n \rfloor) = n + (m \bmod n)$. We conclude that $m \geq F_{k+1} + F_k = F_{k+1}$. □

Theorem 1. EUCLIDGCD(m, n) *runs in $O(\log m)$ iterations.*

Proof: Let k be the number of iterations. Lemma 2 implies that $m \geq F_{k+2} \geq \phi^{k+2}/\sqrt{5} - 1$, where $\phi = (1 + \sqrt{5})/2$ (by the annihilator method). Thus, $k \leq \log_\phi(\sqrt{5}(m + 1)) - 2 = O(\log m)$. □

What about the actual running time? Every number used by the algorithm has $O(\log m)$ bits. Computing the remainder of one b-bit integer by another using the grade-school long division algorithm requires $O(b^2)$ time. So crudely, the running time is $O(b^2 \log m) = O(\log^3 m)$. More careful analysis reduces the time bound to $\boxed{O(\log^2 m)}$. We can make the algorithm even faster by using a fast integer division algorithm (based on FFTs, for example).

D.3 Modular Arithmetic and Algebraic Groups

Modular arithmetic is familiar to anyone who's ever wondered how many minutes are left in an exam that ends at 9:15 when the clock says 8:54.

When we do arithmetic 'modulo n', what we're really doing is a funny kind of arithmetic on the elements of following set:

$$\boxed{\mathbb{Z}_n = \{0, 1, 2, \ldots, n - 1\}}$$

Modular addition and subtraction satisfies all the axioms that we expect implicitly:

- $\mathbb{Z}_n$ is *closed* under addition mod n: For any $a, b \in \mathbb{Z}_n$, their sum $a + b \bmod n$ is also in $\mathbb{Z}_n$

[3] However, Euclid's exposition was a little, erm, informal by current standards, primarily because the Greeks didn't know about induction. He basically said "Try one iteration. If that doesn't work, try three iterations." In modern language, Euclid's algorithm would be written as follows, assuming $m \geq n > 0$.

```
ACTUALEUCLIDGCD($m, n$):
  if $n \mid m$
      return $n$
  else
      return $n \bmod (m \bmod n)$
```

This algorithm is *obviously* incorrect; consider the input $m = 3$, $n = 2$. Nevertheless, mathematics and algorithms students have applied 'Euclidean induction' to a vast number of problems, only to scratch their heads in dismay when they don't get any credit.

- Addition is *associative*: $(a + b \bmod n) + c \bmod n = a + (b + c \bmod n) \bmod n$.
- Zero is an additive *identity* element: $0 + a \bmod n = a + 0 \bmod n = a \bmod n$.
- Every element $a \in \mathbb{Z}_n$ has an *inverse* $b \in \mathbb{Z}_n$ such that $a + b \bmod n = 0$. Specifically, if $a = 0$, then $b = 0$; otherwise, $b = n - a$.

Any set with a binary operator that satisfies the closure, associativity, identity, and inverse axioms is called a *group*. Since $\mathbb{Z}_n$ is a group under an 'addition' operator, we call it an *additive* group. Moreover, because addition is commutative ($a + b \bmod n = b + a \bmod n$), we can call $(\mathbb{Z}_n, + \bmod n)$ is an *abelian* additive group.[4]

What about multiplication? $\mathbb{Z}_n$ is closed under multiplication mod n, multiplication mod n is associative (and commutative), and 1 is a multiplicative identity, but some elements do not have multiplicative inverses. Formally, we say that $\mathbb{Z}_n$ is a *ring* under addition and multiplication modulo n.

If n is composite, then the following theorem shows that we can factor the ring $\mathbb{Z}_n$ into two smaller rings. The Chinese Remainder Theorem is named a third-century Chinese mathematician and algorismist Sun Tzu (or Sun Zi).[5]

The Chinese Remainder Theorem. *If $p \perp q$, then $\mathbb{Z}_{pq} \cong \mathbb{Z}_p \times \mathbb{Z}_q$.*

Okay, okay, before we prove this, let's define all the notation. The product $\mathbb{Z}_p \times \mathbb{Z}_q$ is the set of ordered pairs $\{(a, b) \mid a \in \mathbb{Z}_p, b \in \mathbb{Z}_q\}$, where addition, subtraction, and multiplication are defined as follows:

$$\begin{aligned}(a, b) + (c, d) &= (a + c \bmod p, b + d \bmod q)\\ (a, b) - (c, d) &= (a - c \bmod p, b - d \bmod q)\\ (a, b) \cdot (c, d) &= (ac \bmod p, bd \bmod q)\end{aligned}$$

It's not hard to check that $\mathbb{Z}_p \times \mathbb{Z}_q$ is a ring under these operations, where $(0, 0)$ is the additive identity and $(1, 1)$ is the multiplicative identity. The funky equal sign $\cong$ means that these two rings are *isomorphic*: there is a bijection between the two sets that is consistent with the arithmetic operations.

As an example, the following table describes the bijection between $\mathbb{Z}_{15}$ and $\mathbb{Z}_3 \times \mathbb{Z}_5$:

	0	1	2	3	4
0	0	6	12	3	9
1	10	1	7	13	4
2	5	11	2	8	14

For instance, we have $8 = (2, 3)$ and $13 = (1, 3)$, and

$$\begin{aligned}(2, 3) + (1, 3) = (2 + 1 \bmod 3, 3 + 3 \bmod 5) = (0, 1) = 6 = 21 \bmod 15 = (8 + 13) \bmod 15.\\ (2, 3) \cdot (1, 3) = (2 \cdot 1 \bmod 3, 3 \cdot 3 \bmod 5) = (2, 4) = 14 = 104 \bmod 15 = (8 \cdot 13) \bmod 15.\end{aligned}$$

Proof: The functions $n \mapsto (n \bmod p, n \bmod q)$ and $(a, b) \mapsto aq(q \bmod p) + bp(p \bmod q)$ are inverses of each other, and each map preserves the ring structure. □

[4] after the Norwegian mathematical prodigy Niels Henrik Abel, who (among many other things) proved the insolubility of quintic equations at the ripe old age of 22.

[5] The author of *The Art of War*, who had the same name, lived more than 500 years earlier.

We can extend the Chinese remainder theorem inductively as follows:

The Real Chinese Remainder Theorem. *Suppose* $n = \prod_{i=1}^{r} p_i$*, where* $p_i \perp p_j$ *for all* i *and* j*. Then* $\mathbb{Z}_n \cong \prod_{i=1}^{r} \mathbb{Z}_{p_i} = \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \cdots \times \mathbb{Z}_{p_r}$.

Thus, if we want to perform modular arithmetic where the modulus n is very large, we can improve the performance of our algorithms by breaking n into several relatively prime factors, and performing modular arithmetic separately modulo each factor.

So we can do modular addition, subtraction, and multiplication; what about division? As I said earlier, not every element of $\mathbb{Z}_n$ has a multiplicative inverse. The most obvious example is 0, but there can be others. For example, 3 has no multiplicative inverse in $\mathbb{Z}_{15}$; there is no integer x such that $3x \bmod 15 = 1$. On the other hand, 0 is the only element of $\mathbb{Z}_7$ without a multiplicative inverse:

$$1 \cdot 1 \equiv 2 \cdot 4 \equiv 3 \cdot 5 \equiv 6 \cdot 6 \equiv 1 \pmod 7$$

These examples suggest (I hope) that x has a multiplicative inverse in $\mathbb{Z}_n$ if and only if a and x are relatively prime. This is easy to prove as follows. If $xy \bmod n = 1$, then $xy + kn = 1$ for some integer k. Thus, 1 is an integer linear combination of x and n, so Lemma 1 implies that $\gcd(x, n) = 1$. On the other hand, if $x \perp n$, then $ax + bn = 1$ for some integers a and b, which implies that $ax \bmod n = 1$.

Let's define the set $\mathbb{Z}_n^*$ to be the set of elements if $\mathbb{Z}_n$ that have multiplicative inverses.

$$\boxed{\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid a \perp n\}}$$

It is a tedious exercise to show that $\mathbb{Z}_n^*$ is an abelian group under multiplication modulo n. As long as we stick to elements of this group, we can reasonably talk about 'division mod n'.

We denote the number of elements in $\mathbb{Z}_n^*$ by $\phi(n)$; this is called Euler's *totient* function. This function is remarkably badly-behaved, but there is a relatively simple formula for $\phi(n)$ (not surprisingly) involving prime numbers and division:

$$\boxed{\phi(n) = n \prod_{p|n} \frac{p-1}{p}}$$

I won't prove this formula, but the following intuition is helpful. If we start with $\mathbb{Z}_n$ and throw out all $n/2$ multiples of 2, all $n/3$ multiples of 3, all $n/5$ multiples of 5, and so on. Whenever we throw out multiples of p, we multiply the size of the set by $(p-1)/p$. At the end of this process, we're left with precisely the elements of $\mathbb{Z}_n^*$. *This is not a proof!* On the one hand, this argument throws out some numbers (like 6) more than once, so our estimate seems too low. On the other hand, there are actually $\lceil n/p \rceil$ multiples of p in $\mathbb{Z}_n$, so our estimate seems too high. Surprisingly, these two errors exactly cancel each other out.

D.4 Toward Primality Testing

In this last section, we discuss algorithms for detecting whether a number is prime. Large prime numbers are used primarily (but not exclusively) in cryptography algorithms.

A positive integer is *prime* if it has exactly two positive divisors, and *composite* if it has more than two positive divisors. The integer 1 is neither prime nor composite. Equivalently, an integer $n \geq 2$ is prime if n is relatively prime with every positive integer smaller than n. We can rephrase this definition yet again: n is prime if and only if $\phi(n) = n - 1$.

The obvious algorithm for testing whether a number is prime is *trial division*: simply try every possible nontrivial divisor between 2 and $\sqrt{n}$.

```
TRIALDIVPRIME($n$) :
  for $d \leftarrow 1$ to $\lfloor\sqrt{n}\rfloor$
    if $n \bmod d = 0$
      return COMPOSITE
  return PRIME
```

Unfortunately, this algorithm is horribly slow. Even if we could do the remainder computation in constant time, the overall running time of this algorithm would be $\Omega(\sqrt{n})$, which is exponential in the number of input bits.

This might seem completely hopeless, but fortunately most composite numbers are quite easy to detect as composite. Consider, for example, the related problem of deciding whether a given integer n, whether $n = m^e$ for any integers $m > 1$ and $e > 1$. We can solve this problem in polynomial time with the following straightforward algorithm. The subroutine ROOT(n, i) computes $\lfloor n^{1/i} \rfloor$ essentially by binary search. (I'll leave the analysis as a simple exercise.)

```
EXACTPOWER?($n$):
  for $i \leftarrow 2$ to $\lg n$
    if $(\text{ROOT}(n, i))^i = n$
      return TRUE
  return FALSE
```

```
ROOT($n, i$):
  $r \leftarrow 0$
  for $\ell \leftarrow \lceil(\lg n)/i\rceil$ down to 1
    if $(r + 2^\ell)^i \le n$
      $r \leftarrow r + 2^\ell$
  return $r$
```

To distinguish between arbitrary prime and composite numbers, we need to exploit some results about $\mathbb{Z}_n^*$ from group theory and number theory. First, we define the *order* of an element $x \in \mathbb{Z}_n^*$ as the smallest positive integer k such that $x^k \equiv 1 \pmod{n}$. For example, in the group

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\},$$

the number 2 has order 4, and the number 11 has order 2. For any $x \in \mathbb{Z}_n^*$, we can partition the elements of $\mathbb{Z}_n^*$ into equivalence classes, by declaring $a \sim_x b$ if $a \equiv b \cdot x^k$ for some integer k. The size of every equivalence class is exactly the order of x. Because the equivalence classes must be disjoint, we can conclude that $\boxed{\text{the order of any element divides the size of the group}}$. We can express this observation more succinctly as follows:

Euler's Theorem. $a^{\phi(n)} \equiv 1 \pmod{n}$.[6]

The most interesting special case of this theorem is when n is prime.

Fermat's Little Theorem. *If p is prime, then* $a^p \equiv a \pmod{p}$.[7]

This theorem leads to the following efficient *pseudo*-primality test.

[6] This is not Euler's only theorem; he had thousands. It's not even his most famous theorem. His *second* most famous theorem is the formula $v + e - f = 2$ relating the vertices, edges and faces of any planar map. His most famous theorem is the magic formula $e^{\pi i} + 1 = 0$. Naming something after a mathematician or physicist (as in 'Euler tour' or 'Gaussian elimination' or 'Avogadro's number') is considered a high compliment. Using a lower case letter ('abelian group') is even better; abbreviating ('volt', 'amp') is better still. The number e was named after Euler.

[7] This is not Fermat's only theorem; he had hundreds, most of them stated without proof. Fermat's Last Theorem wasn't the last one he published, but the last one proved. Amazingly, despite his dislike of writing proofs, Fermat was almost always right. In that respect, he was *very* different from you and me.

```
FermatPseudoPrime($n$) :
    choose an integer $a$ between $1$ and $n-1$
    if $a^n \bmod n \neq a$
        return Composite!
    else
        return Prime?
```

In practice, this algorithm is both fast and effective. The (empirical) probability that a random 100-digit composite number will return Prime? is roughly 10^{-30}, even if we always choose $a = 2$. Unfortunately, there are composite numbers that always pass this test, no matter which value of a we use. A *Carmichael number* is a composite integer n such that $a^n \equiv a \pmod{n}$ for every integer a. Thus, Fermat's Little Theorem can be used to distinguish between two types of numbers: (primes and Carmichael numbers) and everything else. Carmichael numbers are extremely rare; in fact, it was proved only in the 1990s that there are an infinite number of them.

To deal with Carmichael numbers effectively, we need to look more closely at the structure of the group $\mathbb{Z}_n^*$. We say that $\mathbb{Z}_n^*$ is *cyclic* if it contains an element of order $\phi(n)$; such an element is called a *generator*. Successive powers of any generator *cycle* through every element of the group in some order. For example, the group $\mathbb{Z}_9^* = \{1, 2, 4, 5, 7, 8\}$ is cyclic, with two generators: 2 and 5, but $\mathbb{Z}_{15}^*$ is not cyclic. The following theorem completely characterizes which groups $\mathbb{Z}_n^*$ are cyclic.

The Cycle Theorem. *$\mathbb{Z}_n^*$ is cyclic if and only if $n = 2$, 4, p^e, or $2p^e$ for some odd prime p and positive integer e.*

This theorem has two relatively simple corollaries.

The Discrete Log Theorem. *Suppose $\mathbb{Z}_n^*$ is cyclic and g is a generator. Then $g^x \equiv g^y \pmod{n}$ if and only if $x \equiv y \pmod{\phi(n)}$.*

Proof: Suppose $g^x \equiv g^y \pmod{n}$. By definition of 'generator', the sequence $\langle 1, g, g^2, \ldots \rangle$ has period $\phi(n)$. Thus, $x \equiv y \pmod{\phi(n)}$. On the other hand, if $x \equiv y \pmod{\phi(n)}$, then $x = y + k\phi(n)$ for some integer k, so $g^x = g^{y+k\phi(n)} = g^y \cdot (g^{\phi(n)})^k$. Euler's Theorem now implies that $(g^{\phi(n)})^k \equiv 1^k \equiv 1 \pmod{n}$, so $g^x \equiv g^y \pmod{n}$. □

The $\sqrt{1}$ Theorem. *Suppose $n = p^e$ for some odd prime p and positive integer e. The only elements $x \in \mathbb{Z}_n^*$ that satisfy the equation $x^2 \equiv 1 \pmod{n}$ are $x = 1$ and $x = n - 1$.*

Proof: Obviously $1^2 \equiv 1 \pmod{n}$ and $(n-1)^2 = n^2 - 2n + 1 \equiv 1 \pmod{n}$.

Suppose $x^2 \equiv 1 \pmod{n}$ where $n = p^e$. By the Cycle Theorem, $\mathbb{Z}_n^*$ is cyclic. Let g be a generator of $\mathbb{Z}_n^*$, and suppose $x = g^k$. Then we immediately have $x^2 = g^{2k} \equiv g^0 = 1 \pmod{p^e}$. The Discrete Log Theorem implies that $2k \equiv 0 \pmod{\phi(p^e)}$. Because p is and odd prime, we have $\phi(p^e) = (p-1)p^{e-1}$, which is even. Thus, the equation $2k \equiv 0 \pmod{\phi(p^e)}$ has just two solutions: $k = 0$ and $k = \phi(p^e)/2$. By the Cycle Theorem, either $x = 1$ or $x = g^{\phi(n)/2}$. Because $x = n - 1$ is also a solution to the original equation, we must have $g^{\phi(n)/2} \equiv n - 1 \pmod{n}$. □

This theorem leads to a different *pseudo*-primality algorithm:

```
Sqrt1PseudoPrime($n$):
    choose a number $a$ between $2$ and $n-2$
    if $a^2 \bmod n = 1$
        return Composite!
    else
        return Prime?
```

As with the previous pseudo-primality test, there are composite numbers that this algorithm cannot identify as composite: powers of primes, for instance. Fortunately, however, the set of composites that always pass the $\sqrt{1}$ test is disjoint from the set of numbers that always pass the Fermat test. In particular, Carmichael numbers *never* have the form p^e.

D.5 The Miller-Rabin Primality Test

The following randomized algorithm, adapted by Michael Rabin from an earlier deterministic algorithm of Gary Miller*, combines the Fermat test and the $\sqrt{1}$ test. The algorithm repeats the same two tests s times, where s is some user-chosen parameter, each time with a random value of a.

```
MILLERRABIN($n$):
  write $n-1 = 2^t u$ where $u$ is odd
  for $i \leftarrow 1$ to $s$
      $a \leftarrow$ RANDOM$(2, n-2)$
      if EUCLIDGCD$(a, n) \neq 1$
          return COMPOSITE!          ⟨⟨obviously!⟩⟩

      $x_0 \leftarrow a^u \bmod n$
      for $j \leftarrow 1$ to $t$
          $x_j \leftarrow x_{j-1}^2 \bmod n$
          if $x_j = 1$ and $x_{j-1} \neq 1$ and $x_{j-1} \neq n-1$
              return COMPOSITE!      ⟨⟨by the $\sqrt{1}$ Theorem⟩⟩

      if $x_t \neq 1$                ⟨⟨$x_t = a^{n-1} \bmod n$⟩⟩
          return COMPOSITE!          ⟨⟨by Fermat's Little Theorem⟩⟩

  return PRIME?
```

First let's consider the running time; for simplicity, we assume that all integer arithmetic is done using the quadratic-time grade school algorithms. We can compute u and t in $O(\log n)$ time by scanning the bits in the binary representation of n. Euclid's algorithm takes $O(\log^2 n)$ time. Computing $a^u \mod n$ requires $O(\log u) = O(\log n)$ multiplications, each of which takes $O(\log^2 n)$ time. Squaring x_j takes $O(\log^2 n)$ time. Overall, the running time for one iteration of the outer loop is $O(\log^3 n + t \log^2 n) = O(\log^3 n)$, because $t \leq \lg n$. Thus, the total running time of this algorithm is $\boxed{O(s \log^3 n)}$. If we set $s = O(\log n)$, this running time is polynomial in the size of the input.

Fine, so it's fast, but is it correct? Like the earlier pseudoprime testing algorithms, a prime input will always cause MILLERRABIN to return PRIME?. Composite numbers, however, may not always return COMPOSITE!; because we choose the number a at random, there is a small probability of error.[8] Fortunately, the error probability can be made ridiculously small—in practice, less than the probability that random quantum fluctuations will instantly transform your computer into a kitten—by setting $s \approx 1000$.

Theorem 2. *If n is composite,* MILLERRABIN(n) *returns* COMPOSITE! *with probability at least* $1 - 2^{-s}$.

[8] If instead, we try all possible values of a, we obtain an exact primality testing algorithm, but it runs in exponential time. Miller's original deterministic algorithm examined every value of a in a carefully-chosen subset of $\mathbb{Z}_n^*$. If the Extended Riemann Hypothesis holds, this subset has logarithmic size, and Miller's algorithm runs in polynomial time. The Riemann Hypothesis is a century-old open problem about the distribution of prime numbers. A solution would be at least as significant as proving Fermat's Last Theorem or P≠NP.

Proof: First, suppose n is not a Carmichael number. Let F be the set of elements of $\mathbb{Z}_n^*$ that pass the Fermat test:

$$F = \{a \in \mathbb{Z}_n^* \mid a^{n-1} \equiv 1 \pmod{n}\}.$$

Because n is not a Carmichael number, F is a *proper* subset of $\mathbb{Z}_n^*$. Given any two elements $a, b \in F$, their product $a \cdot b \bmod n$ in $\mathbb{Z}_n^*$ is also an element of F:

$$(a \cdot b)^{n-1} \equiv a^{n-1} b^{n-1} \equiv 1 \cdot 1 \equiv 1 \pmod{n}$$

We also easily observe that 1 is an element of F, and the multiplicative inverse (mod n) of any element of F is also in F. Thus, F is a proper *subgroup* of $\mathbb{Z}_n^*$, that is, a proper subset that is also a group under the same binary operation. A standard result in group theory states that if F is a subgroup of a finite group G, the number of elements of F divides the number of elements of G. (We used a special case of this result in our proof of Euler's Theorem.) In our setting, this means that $|F|$ divides $\phi(n)$. Since we already know that $|F| < \phi(n)$, we must have $|F| \leq \phi(n)/2$. Thus, at most half the elements of $\mathbb{Z}_n^*$ pass the Fermat test.

The case of Carmichael numbers is more complicated, but the main idea is the same: at most half the possible values of a pass the $\sqrt{1}$ test. See CLRS for further details. □

> *Why are our days numbered and not, say, lettered?*
>
> — Woody Allen

17 String Matching

17.1 Brute Force

The basic object that we're going to talk about for the next two lectures is a *string*, which is really just an array. The elements of the array come from a set Σ called the *alphabet*; the elements themselves are called *characters*. Common examples are ASCII text, where each character is an seven-bit integer[1], strands of DNA, where the alphabet is the set of nucleotides $\{A, C, G, T\}$, or proteins, where the alphabet is the set of 22 amino acids.

The problem we want to solve is the following. Given two strings, a *text* $T[1 \,..\, n]$ and a *pattern* $P[1 \,..\, m]$, find the first *substring* of the text that is the same as the pattern. (It would be easy to extend our algorithms to find *all* matching substrings, but we will resist.) A substring is just a contiguous subarray. For any *shift* s, let T_s denote the substring $T[s \,..\, s+m-1]$. So more formally, we want to find the smallest shift s such that $T_s = P$, or report that there is no match. For example, if the text is the string 'AMANAPLANACATACANALPANAMA'[2] and the pattern is 'CAN', then the output should be 15. If the pattern is 'SPAM', then the answer should be 'none'. In most cases the pattern is much smaller than the text; to make this concrete, I'll assume that $m < n/2$.

Here's the 'obvious' brute force algorithm, but with one immediate improvement. The inner while loop compares the substring T_s with P. If the two strings are not equal, this loop stops at the first character mismatch.

```
ALMOSTBRUTEFORCE($T[1 \,..\, n], P[1 \,..\, m]$):
  for $s \gets 1$ to $n - m + 1$
      equal $\gets$ true
      $i \gets 1$
      while equal and $i \le m$
          if $T[s + i - 1] \ne P[i]$
              equal $\gets$ false
          else
              $i \gets i + 1$
      if equal
          return $s$
  return 'none'
```

[1] Yes, *seven*. Most computer systems use some sort of 8-bit character set, but there's no universally accepted standard. Java supposedly uses the Unicode character set, which has variable-length characters and therefore doesn't really fit into our framework. Just think, someday you'll be able to write '¶ = ℵ[∞++]/℧;' in your Java code! Joy!

[2] Dan Hoey (or rather, his computer program) found the following 540-word palindrome in 1984. We have better online dictionaries now, so I'm sure you could do better.

A man, a plan, a caret, a ban, a myriad, a sum, a lac, a liar, a hoop, a pint, a catalpa, a gas, an oil, a bird, a yell, a vat, a caw, a pax, a wag, a tax, a nay, a ram, a cap, a yam, a gay, a tsar, a wall, a car, a luger, a ward, a bin, a woman, a vassal, a wolf, a tuna, a nit, a pall, a fret, a watt, a bay, a daub, a tan, a cab, a datum, a gall, a hat, a fag, a zap, a say, a jaw, a lay, a wet, a gallop, a tug, a trot, a trap, a tram, a torr, a caper, a top, a tonk, a toll, a ball, a fair, a sax, a minim, a tenor, a bass, a passer, a capital, a rut, an amen, a ted, a cabal, a tang, a sun, an ass, a maw, a sag, a jam, a dam, a sub, a salt, an axon, a sail, an ad, a wadi, a radian, a room, a rood, a rip, a tad, a pariah, a revel, a reel, a reed, a pool, a plug, a pin, a peek, a parabola, a dog, a pat, a cud, a nu, a fan, a pal, a rum, a nod, an eta, a lag, an eel, a batik, a mug, a mot, a nap, a maxim, a mood, a leek, a grub, a gob, a gel, a drab, a citadel, a total, a cedar, a tap, a gag, a rat, a manor, a bar, a gal, a cola, a pap, a yaw, a tab, a raj, a gab, a nag, a pagan, a bag, a jar, a bat, a way, a papa, a local, a gar, a baron, a mat, a rag, a gap, a tar, a decal, a tot, a led, a tic, a bard, a leg, a bog, a burg, a keel, a doom, a mix, a map, an atom, a gum, a kit, a baleen, a gala, a ten, a don, a mural, a pan, a faun, a ducat, a pagoda, a lob, a rap, a keep, a nip, a gulp, a loop, a deer, a leer, a lever, a hair, a pad, a tapir, a door, a moor, an aid, a raid, a wad, an alias, an ox, an atlas, a bus, a madam, a jag, a saw, a mass, an anus, a gnat, a lab, a cadet, an em, a natural, a tip, a caress, a pass, a baronet, a minimax, a sari, a fall, a ballot, a knot, a pot, a rep, a carrot, a mart, a part, a tort, a gut, a poll, a gateway, a law, a jay, a sap, a zag, a fat, a hall, a gamut, a dab, a can, a tabu, a day, a batt, a waterfall, a patina, a nut, a flow, a lass, a van, a mow, a nib, a draw, a regular, a call, a war, a stay, a gam, a yap, a cam, a ray, an ax, a tag, a wax, a paw, a cat, a valley, a drib, a lion, a saga, a plat, a catnip, a pooh, a rail, a calamus, a dairyman, a bater, a canal—Panama!

In the worst case, the running time of this algorithm is $O((n-m)m) = O(nm)$, and we can actually achieve this running time by searching for the pattern `AAA...AAAB` with $m-1$ `A`'s, in a text consisting of n `A`'s.

In practice, though, breaking out of the inner loop at the first mismatch makes this algorithm quite practical. We can wave our hands at this by assuming that the text and pattern are both random. Then on average, we perform a constant number of comparisons at each position i, so the total expected number of comparisons is $O(n)$. Of course, neither English nor DNA is really random, so this is only a heuristic argument.

17.2 Strings as Numbers

For the rest of the lecture, let's assume that the alphabet consists of the numbers `0` through `9`, so we can interpret any array of characters as either a string or a decimal number. In particular, let p be the numerical value of the pattern P, and for any shift s, let t_s be the numerical value of T_s:

$$p = \sum_{i=1}^{m} 10^{m-i} \cdot P[i] \qquad t_s = \sum_{i=1}^{m} 10^{m-i} \cdot T[s+i-1]$$

For example, if $T = \texttt{3141592653589793}\underline{\texttt{2384}}\texttt{626433832795028841971}$ and $m = 4$, then $t_{17} = 2384$.

Clearly we can rephrase our problem as follows: Find the smallest s, if any, such that $p = t_s$. We can compute p in $O(m)$ arithmetic operations, without having to explicitly compute powers of ten, using *Horner's rule*:

$$p = P[m] + 10\left(P[m-1] + 10\big(P[m-2] + \cdots + 10\big(P[2] + 10 \cdot P[1]\big)\cdots\big)\right)$$

We could also compute any t_s in $O(m)$ operations using Horner's rule, but this leads to essentially the same brute-force algorithm as before. But once we know t_s, we can actually compute t_{s+1} in constant time just by doing a little arithmetic — subtract off the most significant digit $T[s] \cdot 10^{m-1}$, shift everything up by one digit, and add the new least significant digit $T[r+m]$:

$$t_{s+1} = 10\big(t_s - 10^{m-1} \cdot T[s]\big) + T[s+m]$$

To make this fast, we need to precompute the constant 10^{m-1}. (And we know how to do that quickly. Right?) So it seems that we can solve the string matching problem in $O(n)$ worst-case time using the following algorithm:

```
NumberSearch($T[1..n], P[1..m]$):
  $\sigma \leftarrow 10^{m-1}$
  $p \leftarrow 0$
  $t_1 \leftarrow 0$
  for $i \leftarrow 1$ to $m$
      $p \leftarrow 10 \cdot p + P[i]$
      $t_1 \leftarrow 10 \cdot t_1 + T[i]$
  for $s \leftarrow 1$ to $n - m + 1$
      if $p = t_s$
          return $s$
      $t_{s+1} \leftarrow 10 \cdot \big(t_s - \sigma \cdot T[s]\big) + T[s+m]$
  return 'none'
```

Unfortunately, the most we can say is that the number of *arithmetic operations* is $O(n)$. These operations act on numbers with up to m digits. Since we want to handle arbitrarily long patterns, we can't assume that each operation takes only constant time!

17.3 Karp-Rabin Fingerprinting

To make this algorithm efficient, we will make one simple change, discovered by Richard Karp and Michael Rabin in 1981:

Perform all arithmetic modulo some prime number q.

We choose q so that the value $10q$ fits into a standard integer variable, so that we don't need any fancy long-integer data types. The values $(p \bmod q)$ and $(t_s \bmod q)$ are called the *fingerprints* of P and T_s, respectively. We can now compute $(p \bmod q)$ and $(t_1 \bmod q)$ in $O(m)$ *time* using Horner's rule 'mod q'

$$p \bmod q = P[m] + \big(\cdots + \big(10 \cdot \big(P[2] + \big(10 \cdot P[1] \bmod q\big) \bmod q\big) \bmod q\big) \cdots\big)\big) \bmod q$$

and similarly, given $(t_s \bmod q)$, we can compute $(t_{s+1} \bmod q)$ in constant time.

$$t_{s+1} \bmod q = \big(10 \cdot \big(t_s - \big(\big(10^{m-1} \bmod q\big) \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q$$

Again, we have to precompute the value $(10^{m-1} \bmod q)$ to make this fast.

If $(p \bmod q) \neq (t_s \bmod q)$, then certainly $P \neq T_s$. However, if $(p \bmod q) = (t_s \bmod q)$, we can't tell whether $P = T_s$ or not. All we know for sure is that p and t_s differ by some integer multiple of q. If $P \neq T_s$ in this case, we say there is a *false match* at shift s. To test for a false match, we simply do a brute-force string comparison. (In the algorithm below, $\tilde{p} = p \bmod q$ and $\tilde{t}_s = t_s \bmod q$.)

```
KARPRABIN($T[1\,..\,n], P[1\,..\,m]$):
    choose a small prime $q$
    $\sigma \leftarrow 10^{m-1} \bmod q$
    $\tilde{p} \leftarrow 0$
    $\tilde{t}_1 \leftarrow 0$
    for $i \leftarrow 1$ to $m$
        $\tilde{p} \leftarrow (10 \cdot \tilde{p} \bmod q) + P[i] \bmod q$
        $\tilde{t}_1 \leftarrow (10 \cdot \tilde{t}_1 \bmod q) + T[i] \bmod q$
    for $s \leftarrow 1$ to $n - m + 1$
        if $\tilde{p} = \tilde{t}_s$
            if $P = T_s$        ⟨⟨brute-force $O(m)$-time comparison⟩⟩
                return $s$
        $\tilde{t}_{s+1} \leftarrow \big(10 \cdot \big(\tilde{t}_s - \big(\sigma \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q$
    return 'none'
```

The running time of this algorithm is $O(n + Fm)$, where F is the number of false matches.

Intuitively, we expect the fingerprints t_s to jump around between 0 and $q - 1$ more or less at random, so the 'probability' of a false match 'ought' to be $1/q$. This intuition implies that $F = n/q$ 'on average', which gives us an 'expected' running time of $O(n + nm/q)$. If we always choose $q \geq m$, this simplifies to $O(n)$. But of course all this intuitive talk of probabilities is just frantic meaningless handwaving, since we haven't actually done anything random yet.

17.4 Random Prime Number Facts

The real power of the Karp-Rabin algorithm is that by choosing the modulus q *randomly*, we can actually formalize this intuition! The first line of KARPRABIN should really read as follows:

Let q be a random prime number less than $nm^2 \log(nm^2)$.

For any positive integer u, let $\pi(u)$ denote the number of prime numbers less than u. There are $\pi(nm^2 \log nm^2)$ possible values for q, each with the same probability of being chosen.

Our analysis needs two results from number theory. I won't even try to prove the first one, but the second one is quite easy.

Lemma 1 (The Prime Number Theorem). $\pi(u) = \Theta(u/\log u)$.

Lemma 2. *Any integer x has at most $\lfloor \lg x \rfloor$ distinct prime divisors.*

Proof: If x has k distinct prime divisors, then $x \geq 2^k$, since every prime number is bigger than 1. □

Let's assume that there are no true matches, so $p \neq t_s$ for all s. (That's the worst case for the algorithm anyway.) Let's define a strange variable X as follows:

$$X = \prod_{s=1}^{n-m+1} |p - t_s|.$$

Notice that by our assumption, X can't be zero.

Now suppose we have false match at shift s. Then $p \bmod q = t_s \bmod q$, so $p - t_s$ is an integer multiple of q, and this implies that X is also an integer multiple of q. In other words, if there is a false match, then q must one of the prime divisors of X.

Since $p < 10^m$ and $t_s < 10^m$, we must have $X < 10^{nm}$. Thus, by the second lemma, X has $O(mn)$ prime divisors. Since we chose q randomly from a set of $\pi(nm^2 \log(nm^2)) = \Omega(nm^2)$ prime numbers, the probability that q divides X is at most

$$\frac{O(nm)}{\Omega(nm^2)} = O\left(\frac{1}{m}\right).$$

We have just proven the following amazing fact.

$$\boxed{\text{The probability of getting a false match is } O(1/m).}$$

Recall that the running time of KARPRABIN is $O(n + mF)$, where F is the number of false matches. By using the *really* loose upper bound $\mathrm{E}[F] \leq \Pr[F > 0] \cdot n$, we can conclude that the expected number of false matches is $O(n/m)$. Thus, the expected running time of the KARPRABIN algorithm is $O(n)$.

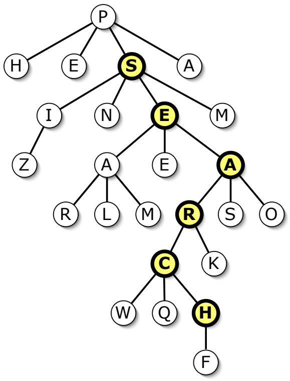
17.5 Random Prime Number?

Actually choosing a random prime number is not particularly easy. The best method known is to repeatedly generate a random integer and test to see if it's prime. In practice, it's enough to choose a random *probable* prime. You can read about probable primes in Non-Lecture D, or in the textbook *Randomized Algorithms* by Rajeev Motwani and Prabhakar Raghavan (Cambridge, 1995).

Exercises

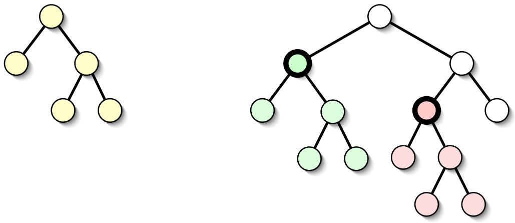
1. Describe and analyze a two-dimensional variant of KARPRABIN that searches for a given two-dimensional pattern $P[1..p][1..q]$ within a given two-dimensional 'text' $T[1..m][1..,n]$. Your algorithm should report *all* index pairs (i, j) such that the subarray $T[i..i+p-1][j..j+q-1]$ is identical to the given pattern, in $O(pq + mn)$ expected time.

2. Describe and analyze a variant of KarpRabin that looks for strings inside labeled rooted trees. The input consists of a *pattern string* $P[1\,..\,m]$ and a rooted *text tree* T with n nodes, each labeled with a single character. Nodes in T can have any number of children. Your algorithm should either return a downward path in T whose labels match the string P, or report that there is no such path. The expected running time of your algorithm should be $O(m+n)$.



The string SEARCH appears on a downward path in the tree.

3. Describe and analyze a variant of KarpRabin that searches for subtrees of ordered rooted binary trees (every node has a left subtree and a right subtree, either or both of which may be empty). The input consists of a *pattern tree* P with m nodes and a *text tree* T with n nodes. Your algorithm should report *all* nodes v in T such that the subtree rooted at v is structurally identical to P. The expected running time of your algorithm should be $O(m+n)$. Ignore all search keys, labels, or other data in the nodes; only the left/right pointer structure matters.



The pattern tree (left) appears exactly twice in the text tree (right).

*4. How important is the requirement that the fingerprint modulus q is prime? Specifically, suppose q is chosen uniformly at random in the range $1\,..\,N$. If $t_s \neq p$, what is the probability that $\tilde{t}_s = \tilde{p}$? What does this imply about the expected number of false matches? How large should N be to guarantee expected running time $O(m+n)$? *[Hint: This will require some additional number theory.]*

> *Philosophers gathered from far and near*
> *To sit at his feat and hear and hear,*
> *Though he never was heard*
> *To utter a word*
> *But* "Abracadabra, abracadab,
> Abracada, abracad,
> Abraca, abrac, abra, ab!"
> *'Twas all he had,*
> *'Twas all they wanted to hear, and each*
> *Made copious notes of the mystical speech,*
> *Which they published next –*
> *A trickle of text*
> *In the meadow of commentary.*
> *Mighty big books were these,*
> *In a number, as leaves of trees;*
> *In learning, remarkably – very!*
>
> — Jamrach Holobom, quoted by Ambrose Bierce,
> *The Devil's Dictionary* (1911)

18 More String Matching

18.1 Redundant Comparisons

Let's go back to the character-by-character method for string matching. Suppose we are looking for the pattern 'ABRACADABRA' in some longer text using the (almost) brute force algorithm described in the previous lecture. Suppose also that when $s = 11$, the substring comparison fails at the fifth position; the corresponding character in the text (just after the vertical line below) is not a C. At this point, our algorithm would increment s and start the substring comparison from scratch.

HOCUSPOCUSABRA|BRACADABRA...
ABRA|C̸~~ADABRA~~
ABR|ACADABRA

If we look carefully at the text and the pattern, however, we should notice right away that there's no point in looking at $s = 12$. We already know that the next character is a B — after all, it matched $P[2]$ during the previous comparison — so why bother even looking there? Likewise, we already know that the next two shifts $s = 13$ and $s = 14$ will also fail, so why bother looking there?

HOCUSPOCUSABRA|BRACADABRA...
ABRA|C̸~~ADABRA~~
A̸~~BR|ACADABRA~~
A̸~~B|RACADABRA~~
A|BRACADABRA

Finally, when we get to $s = 15$, we can't immediately rule out a match based on earlier comparisons. However, for precisely the same reason, we shouldn't start the substring comparison over from scratch — we already know that $T[15] = P[4] = \texttt{A}$. Instead, we should start the substring comparison at the *second* character of the pattern, since we don't yet know whether or not it matches the corresponding text character.

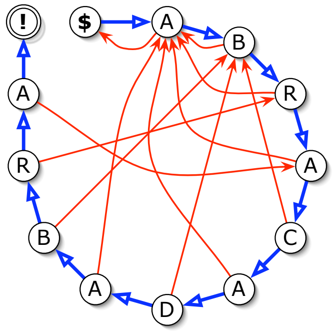
If you play with this idea long enough, you'll notice that the character comparisons should always advance through the text. **Once we've found a match for a text character, we never need to do another comparison with that text character again.** In other words, we should be able to optimize the brute-force algorithm so that it always *advances* through the text.

You'll also eventually notice a good rule for finding the next 'reasonable' shift s. A *prefix* of a string is a substring that includes the first character; a *suffix* is a substring that includes the last character. A prefix or suffix is *proper* if it is not the entire string. Suppose we have just discovered that $T[i] \neq P[j]$. **The next reasonable shift is the smallest value of s such that $T[s\,..\,i-1]$, which is a suffix of the previously-read text, is also a proper prefix of the pattern.**

In this lecture, we'll describe a string matching algorithm, published by Donald Knuth, James Morris, and Vaughn Pratt in 1977, that implements both of these ideas.

18.2 Finite State Machines

If we have a string matching algorithm that follows our first observation (that we always advance through the text), we can interpret it as feeding the text through a special type of *finite-state machine*. A finite state machine is a directed graph. Each node in the graph, or *state*, is labeled with a character from the pattern, except for two special nodes labeled $\textcircled{\$}$ and $\textcircled{!}$. Each node has two outgoing edges, a *success* edge and a *failure* edge. The success edges define a path through the characters of the pattern in order, starting at $\textcircled{\$}$ and ending at $\textcircled{!}$. Failure edges always point to earlier characters in the pattern.



A finite state machine for the string 'ABRADACABRA'.
Thick arrows are the success edges; thin arrows are the failure edges.

We use the finite state machine to search for the pattern as follows. At all times, we have a current text character $T[i]$ and a current node in the graph, which is usually labeled by some pattern character $P[j]$. We iterate the following rules:

- If $T[i] = P[j]$, or if the current label is $\textcircled{\$}$, follow the success edge to the next node and increment i. (So there is no failure edge from the start node $\textcircled{\$}$.)
- If $T[i] \neq P[j]$, follow the failure edge back to an earlier node, but do not change i.

For the moment, let's simply assume that the failure edges are defined correctly—we'll come back to this later. If we ever reach the node labeled $\textcircled{!}$, then we've found an instance of the pattern in the text, and if we run out of text characters ($i > n$) before we reach $\textcircled{!}$, then there is no match.

The finite state machine is really just a (very!) convenient metaphor. In a real implementation, we would not construct the entire graph. Since the success edges always go through the pattern characters in order, we only have to remember where the failure edges go. We can encode this *failure function* in an array $\mathit{fail}[1\,..\,n]$, so that for each j there is a failure edge from node j to node $\mathit{fail}[j]$. Following a failure edge back to an earlier state exactly corresponds, in our earlier formulation, to shifting the pattern forward. The failure function $\mathit{fail}[j]$ tells us how far to shift after a character mismatch $T[i] \neq P[j]$.

Here's what the actual algorithm looks like:

```
KnuthMorrisPratt($T[1\,..\,n], P[1\,..\,m]$):
  $j \leftarrow 1$
  for $i \leftarrow 1$ to $n$
      while $j > 0$ and $T[i] \neq P[j]$
          $j \leftarrow fail[j]$
      if $j = m$      ⟨⟨Found it!⟩⟩
          return $i - m + 1$
      $j \leftarrow j + 1$
  return 'none'
```

Before we discuss computing the failure function, let's analyze the running time of KnuthMorrisPratt under the assumption that a correct failure function is already known. At each character comparison, either we increase i and j by one, or we decrease j and leave i alone. We can increment i at most $n-1$ times before we run out of text, so there are at most $n-1$ successful comparisons. Similarly, there can be at most $n-1$ failed comparisons, since the number of times we decrease j cannot exceed the number of times we increment j. In other words, we can amortize character mismatches against earlier character matches. Thus, the total number of character comparisons performed by KnuthMorrisPratt in the worst case is $O(n)$.

18.3 Computing the Failure Function

We can now rephrase our second intuitive rule about how to choose a reasonable shift after a character mismatch $T[i] \neq P[j]$:

$P[1\,..\,\mathit{fail}[j]-1]$ is the longest proper prefix of $P[1\,..\,j-1]$ that is also a suffix of $T[1\,..\,i-1]$.

Notice, however, that if we are comparing $T[i]$ against $P[j]$, then we must have already matched the first $j-1$ characters of the pattern. In other words, we already know that $P[1\,..\,j-1]$ is a suffix of $T[1\,..\,i-1]$. Thus, we can rephrase the prefix-suffix rule as follows:

$P[1\,..\,\mathit{fail}[j]-1]$ is the longest proper prefix of $P[1\,..\,j-1]$ that is also a suffix of $P[1\,..\,j-1]$.

This is the definition of the Knuth-Morris-Pratt failure function $\mathit{fail}[j]$ for all $j > 1$.[1] By convention we set $\mathit{fail}[1] = 0$; this tells the KMP algorithm that if the first pattern character doesn't match, it should just give up and try the next text character.

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
$\mathit{fail}[i]$	0	1	1	1	2	1	2	1	2	3	4

Failure function for the string 'ABRACADABRA'
(Compare with the finite state machine on the previous page.)

We could easily compute the failure function in $O(m^3)$ time by checking, for each j, whether every prefix of $P[1\,..\,j-1]$ is also a suffix of $P[1\,..\,j-1]$, but this is not the fastest method. The following algorithm essentially uses the KMP search algorithm to look for the pattern inside itself!

[1] Many algorithms textbooks, including CLRS, define a similar *prefix function*, denoted $\pi[j]$, as follows:

$P[1\,..\,\pi[j]]$ is the longest proper prefix of $P[1\,..\,j]$ that is also a suffix of $P[1\,..\,j]$.

These two functions are not the same, but they are related by the simple equation $\pi[j] = \mathit{fail}[j+1]-1$. The off-by-one difference between the two functions adds a few extra $+1$s to the CLRS version of the algorithm.

```
ComputeFailure($P[1\,..\,m]$):
  $j \leftarrow 0$
  for $i \leftarrow 1$ to $m$
      $fail[i] \leftarrow j$        $(*)$
      while $j > 0$ and $P[i] \neq P[j]$
          $j \leftarrow fail[j]$
      $j \leftarrow j + 1$
```

Here's an example of this algorithm in action. In each line, the current values of i and j are indicated by superscripts; $\$$ represents the beginning of the string. (You should imagine pointing at $P[j]$ with your left hand and pointing at $P[i]$ with your right hand, and moving your fingers according to the algorithm's directions.)

$j \leftarrow 0,\ i \leftarrow 1$	$\$^j$	A^i	B	R	A	C	A	D	A	B	R	X . . .
$fail[i] \leftarrow j$		**0**										. . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A^j	B^i	R	A	C	A	D	A	B	R	X . . .
$fail[i] \leftarrow j$		0	**1**									. . .
$j \leftarrow fail[j]$	$\$^j$	A	B^i	R	A	C	A	D	A	B	R	X . . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A^j	B	R^i	A	C	A	D	A	B	R	X . . .
$fail[i] \leftarrow j$		0	1	**1**								. . .
$j \leftarrow fail[j]$	$\$^j$	A	B	R^i	A	C	A	D	A	B	R	X . . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A^j	B	R	A^i	C	A	D	A	B	R	X . . .
$fail[i] \leftarrow j$		0	1	1	**1**							. . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A	B^j	R	A	C^i	A	D	A	B	R	X . . .
$fail[i] \leftarrow j$		0	1	1	1	**2**						. . .
$j \leftarrow fail[j]$	$\$$	A^j	B	R	A	C^i	A	D	A	B	R	X . . .
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C^i	A	D	A	B	R	X . . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A^j	B	R	A	C	A^i	D	A	B	R	X . . .
$fail[i] \leftarrow j$		0	1	1	1	2	**1**					. . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A	B^j	R	A	C	A	D^i	A	B	R	X . . .
$fail[i] \leftarrow j$		0	1	1	1	2	1	**2**				. . .
$j \leftarrow fail[j]$	$\$$	A^j	B	R	A	C	A	D^i	A	B	R	X . . .
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D^i	A	B	R	X . . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A^j	B	R	A	C	A	D	A^i	B	R	X . . .
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	**1**			. . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A	B^j	R	A	C	A	D	A	B^i	R	X . . .
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	**2**		. . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A	B	R^j	A	C	A	D	A	B	R^i	X . . .
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	**3**	. . .
$j \leftarrow j+1,\ i \leftarrow i+1$	$\$$	A	B	R	A^j	C	A	D	A	B	R	X^i. . .
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	**4** . . .
$j \leftarrow fail[j]$	$\$$	A^j	B	R	A	C	A	D	A	B	R	X^i. . .
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D	A	B	R	X^i. . .

ComputeFailure in action. Do this yourself by hand!

Just as we did for KnuthMorrisPratt, we can analyze ComputeFailure by amortizing character mismatches against earlier character matches. Since there are at most m character matches, ComputeFailure runs in $O(m)$ time.

Let's prove (by induction, of course) that ComputeFailure correctly computes the failure function. The base case $fail[1] = 0$ is obvious. Assuming inductively that we correctly computed $fail[1]$ through $fail[i]$ in line $(*)$, we need to show that $fail[i+1]$ is also correct. Just after the ith iteration of line $(*)$, we have $j = fail[i]$, so $P[1\,..\,j-1]$ is the longest proper prefix of $P[1\,..\,i-1]$ that is also a suffix.

Let's define the iterated failure functions $fail^c[j]$ inductively as follows: $fail^0[j] = j$, and

$$fail^c[j] = fail[fail^{c-1}[j]] = \overbrace{fail[fail[\cdots[fail}^{c}[j]]\cdots]].$$

In particular, if $\mathit{fail}^{c-1}[j] = 0$, then $\mathit{fail}^{c}[j]$ is undefined. We can easily show by induction that every string of the form $P[1\,..\,\mathit{fail}^{c}[j] - 1]$ is both a proper prefix and a proper suffix of $P[1\,..\,i-1]$, and in fact, these are the only examples. Thus, the longest proper prefix/suffix of $P[1\,..\,i]$ must be the longest string of the form $P[1\,..\,\mathit{fail}^{c}[j]]$ — *i.e.*, the one with smallest c — such that $P[\mathit{fail}^{c}[j]] = P[i]$. This is exactly what the while loop in ComputeFailure computes; the $(c+1)$th iteration compares $P[\mathit{fail}^{c}[j]] = P[\mathit{fail}^{c+1}[i]]$ against $P[i]$. ComputeFailure is actually a *dynamic programming* implementation of the following recursive definition of $\mathit{fail}[i]$:

$$\mathit{fail}[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max\limits_{c \geq 1} \left\{ \mathit{fail}^{c}[i-1] + 1 \;\middle|\; P[i-1] = P[\mathit{fail}^{c}[i-1]] \right\} & \text{otherwise.} \end{cases}$$

18.4 Optimizing the Failure Function

We can speed up KnuthMorrisPratt slightly by making one small change to the failure function. Recall that after comparing $T[i]$ against $P[j]$ and finding a mismatch, the algorithm compares $T[i]$ against $P[\mathit{fail}[j]]$. With the current definition, however, it is possible that $P[j]$ and $P[\mathit{fail}[j]]$ are actually the same character, in which case the next character comparison will automatically fail. So why do the comparison at all?

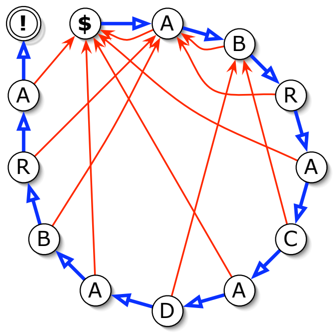
We can optimize the failure function by 'short-circuiting' these redundant comparisons with some simple post-processing:

```
OptimizeFailure($P[1\,..\,m]$, $\mathit{fail}[1\,..\,m]$):
  for $i \leftarrow 2$ to $m$
      if $P[i] = P[\mathit{fail}[i]]$
          $\mathit{fail}[i] \leftarrow \mathit{fail}[\mathit{fail}[i]]$
```

We can also compute the optimized failure function directly by adding three new lines (in bold) to the ComputeFailure function.

```
ComputeOptFailure($P[1\,..\,m]$):
  $j \leftarrow 0$
  for $i \leftarrow 1$ to $m$
      if $P[i] = P[j]$
          $\mathit{fail}[i] \leftarrow \mathit{fail}[j]$
      else
          $\mathit{fail}[i] \leftarrow j$
      while $j > 0$ and $P[i] \neq P[j]$
          $j \leftarrow \mathit{fail}[j]$
      $j \leftarrow j + 1$
```

This optimization slows down the preprocessing slightly, but it may significantly decrease the number of comparisons at each text character. The worst-case running time is still $O(n)$; however, the constant is about half as big as for the unoptimized version, so this could be a significant improvement in practice.



Optimized finite state machine for the string 'ABRADACABRA'

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
fail$[i]$	0	1	1	**0**	2	**0**	2	**0**	**1**	**1**	**0**

Optimized failure function for 'ABRACADABRA', with changes in bold.

Here are the unoptimized and optimized failure functions for a few more patterns:

$P[i]$	A	N	A	N	A	B	A	N	A	N	A	N	A
unoptimized *fail*$[i]$	0	1	1	2	3	4	1	2	3	4	5	6	5
optimized *fail*$[i]$	0	1	0	1	0	4	0	1	0	1	0	6	0

Failure functions for 'ANANABANANANA'.

$P[i]$	A	B	A	B	C	A	B	A	B	C	A	B	C
unoptimized *fail*$[i]$	0	1	1	2	3	1	2	3	4	5	6	7	8
optimized *fail*$[i]$	0	1	0	1	3	0	1	0	1	3	0	1	8

Failure functions for 'ABABCABABCABC'.

$P[i]$	A	B	B	A	B	B	A	B	A	B	B	A	B
unoptimized *fail*$[i]$	0	1	1	1	2	3	4	5	6	2	3	4	5
optimized *fail*$[i]$	0	1	1	0	1	1	0	1	6	1	1	0	1

Failure functions for 'ABBABBABABBAB'.

Exercises

1. A *palindrome* is any string that is the same as its reversal, such as X, ABBA, or REDIVIDER. Describe and analyze an algorithm that computes the longest palindrome that is a (not necessarily proper) prefix of a given string $T[1\,..\,n]$. Your algorithm should run in $O(n)$ time.

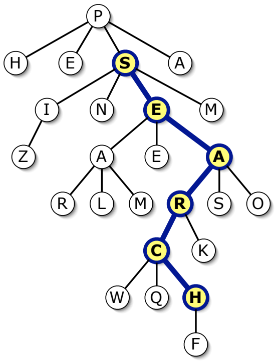
2. Describe a modification of KNUTHMORRISPRATT in which the pattern can contain any number of *wildcard* symbols *****, each of which matches an arbitrary string. For example, the pattern ABR*****CAD*****BRA appears in the text SCHABR**AIN**CADBRANCH; in this case, the second ***** matches the empty string. Your algorithm should run in $O(m+n)$ time, where m is the length of the pattern and n is the length of the text.

3. Describe a modification of KNUTHMORRISPRATT in which the pattern can contain any number of *wildcard* symbols **?**, each of which matches an arbitrary single character. For example, the pattern ABR**?**CAD**?**BRA appears in the text SCHABR**U**CAD**I**BRANCH. Your algorithm should run in $O(m+qn)$ time, where m is the length of the pattern, n is the length of the text., and q is the number of **?**s in the pattern.

*4. Describe another algorithm for the previous problem that runs in time $O(m + kn)$, where k is the number of runs of consecutive non-wildcard characters in the pattern. For example, the pattern **?**FISH**???**B**??**IS**????**CUIT**?** has $k = 4$ runs.

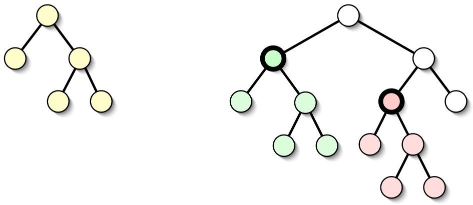
5. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols **=**, each of which matches *the same* arbitrary single character. For example, the pattern **=**HOC**=**SPOC**=**S appears in the texts WH**U**HOC**U**SPOC**U**SOT and ABR**A**HOC**A**SPOC**A**SCADABRA, but *not* in the text FRI**S**HOC**U**SPOC**E**STIX. Your algorithm should run in $O(m + n)$ time, where m is the length of the pattern and n is the length of the text.

6. Describe and analyze a variant of KnuthMorrisPratt that looks for strings inside labeled rooted trees. The input consists of a *pattern string* $P[1 .. m]$ and a rooted *text tree* T with n nodes, each labeled with a single character. Nodes in T can have any number of children. Your algorithm should either return a downward path in T whose labels match the string P, or report that there is no such path. Your algorithm should run in $O(m + n)$ time.



The string SEARCH appears on a downward path in the tree.

7. Describe and analyze a variant of KnuthMorrisPratt that searches for subtrees of ordered rooted binary trees (every node has a left subtree and a right subtree, either or both of which may be empty). The input consists of a *pattern tree* P with m nodes and a *text tree* T with n nodes. Your algorithm should report *all* nodes v in T such that the subtree rooted at v is structurally identical to P. Your algorithm should run in $O(m + n)$ time. Ignore all search keys, labels, or other data in the nodes; only the left/right pointer structure matters.



The pattern tree (left) appears exactly twice in the text tree (right).

8. This problem considers the maximum length of a *failure chain* $j \to \mathit{fail}[j] \to \mathit{fail}[\mathit{fail}[j]] \to \mathit{fail}[\mathit{fail}[\mathit{fail}[j]]] \to \cdots \to 0$, or equivalently, the maximum number of iterations of the inner loop of KnuthMorrisPratt. This clearly depends on which failure function we use: unoptimized or optimized. Let m be an arbitrary positive integer.

(a) Describe a pattern $A[1\,..\,m]$ whose longest *unoptimized* failure chain has length m.

(b) Describe a pattern $B[1\,..\,m]$ whose longest *optimized* failure chain has length $\Theta(\log m)$.

*(c) Describe a pattern $C[1\,..\,m]$ *containing only two different characters*, whose longest optimized failure chain has length $\Theta(\log m)$.

*(d) Prove that for any pattern of length m, the longest optimized failure chain has length at most $O(\log m)$.

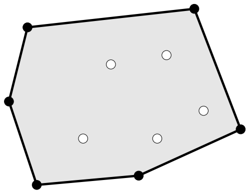
E Convex Hulls

E.1 Definitions

We are given a set P of n points in the plane. We want to compute something called the *convex hull* of P. Intuitively, the convex hull is what you get by driving a nail into the plane at each point and then wrapping a piece of string around the nails. More formally, the convex hull is the smallest convex polygon containing the points:

- **polygon:** A region of the plane bounded by a cycle of line segments, called *edges*, joined end-to-end in a cycle. Points where two successive edges meet are called *vertices*.
- **convex:** For any two points p, q inside the polygon, the line segment $\overline{pq}$ is completely inside the polygon.
- **smallest:** Any convex proper subset of the convex hull excludes at least one point in P. This implies that every vertex of the convex hull is a point in P.

We can also define the convex hull as the *largest* convex polygon whose vertices are all points in P, or the *unique* convex polygon that contains P and whose vertices are all points in P. Notice that P might have *interior* points that are not vertices of the convex hull.



A set of points and its convex hull.
Convex hull vertices are black; interior points are white.

Just to make things concrete, we will represent the points in P by their Cartesian coordinates, in two arrays $X[1..n]$ and $Y[1..n]$. We will represent the convex hull as a circular linked list of vertices in counterclockwise order. if the ith point is a vertex of the convex hull, $next[i]$ is index of the next vertex counterclockwise and $pred[i]$ is the index of the next vertex clockwise; otherwise, $next[i] = pred[i] = 0$. It doesn't matter which vertex we choose as the 'head' of the list. The decision to list vertices counterclockwise instead of clockwise is arbitrary.

To simplify the presentation of the convex hull algorithms, I will assume that the points are in *general position*, meaning (in this context) that *no three points lie on a common line*. This is just like assuming that no two elements are equal when we talk about sorting algorithms. If we wanted to really implement these algorithms, we would have to handle colinear triples correctly, or at least consistently. This is fairly easy, but definitely not trivial.

E.2 Simple Cases

Computing the convex hull of a single point is trivial; we just return that point. Computing the convex hull of two points is also trivial.

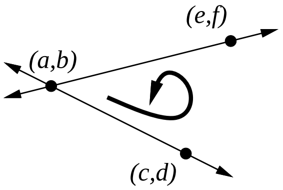
For three points, we have two different possibilities — either the points are listed in the array in clockwise order or counterclockwise order. Suppose our three points are (a, b), (c, d), and (e, f), given in that order, and for the moment, let's also suppose that the first point is furthest to the left, so $a < c$ and $a < f$. Then the three points are in counterclockwise order if and only if the line $\overleftrightarrow{(a,b)(c,d)}$ is less than the slope of the line $\overleftrightarrow{(a,b)(e,f)}$:

$$\text{counterclockwise} \iff \frac{d-b}{c-a} < \frac{f-b}{e-a}$$

Since both denominators are positive, we can rewrite this inequality as follows:

$$\boxed{\text{counterclockwise} \iff (f-b)(c-a) > (d-b)(e-a)}$$

This final inequality is correct even if (a,b) is not the leftmost point. If the inequality is reversed, then the points are in clockwise order. If the three points are colinear (remember, we're assuming that never happens), then the two expressions are equal.



Three points in counterclockwise order.

Another way of thinking about this counterclockwise test is that we're computing the *cross-product* of the two vectors $(c,d) - (a,b)$ and $(e,f) - (a,b)$, which is defined as a 2×2 determinant:

$$\boxed{\text{counterclockwise} \iff \begin{vmatrix} c-a & d-b \\ e-a & f-b \end{vmatrix} > 0}$$

We can also write it as a 3×3 determinant:

$$\boxed{\text{counterclockwise} \iff \begin{vmatrix} 1 & a & b \\ 1 & c & d \\ 1 & e & f \end{vmatrix} > 0}$$

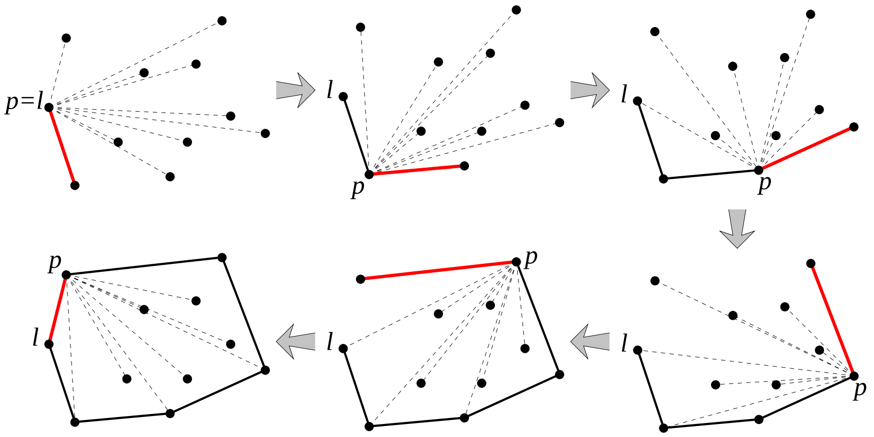
All three boxed expressions are algebraically identical.

This counterclockwise test plays *exactly* the same role in convex hull algorithms as comparisons play in sorting algorithms. Computing the convex hull of of three points is analogous to sorting two numbers: either they're in the correct order or in the opposite order.

E.3 Jarvis's Algorithm (Wrapping)

Perhaps the simplest algorithm for computing convex hulls simply simulates the process of wrapping a piece of string around the points. This algorithm is usually called *Jarvis's march*, but it is also referred to as the *gift-wrapping* algorithm.

Jarvis's march starts by computing the leftmost point ℓ, *i.e.*, the point whose x-coordinate is smallest, since we know that the left most point must be a convex hull vertex. Finding ℓ clearly takes linear time.



The execution of Jarvis's March.

Then the algorithm does a series of *pivoting* steps to find each successive convex hull vertex, starting with ℓ and continuing until we reach ℓ again. The vertex immediately following a point p is the point that appears to be furthest to the right to someone standing at p and looking at the other points. In other words, if q is the vertex following p, and r is any other input point, then the triple p, q, r is in counter-clockwise order. We can find each successive vertex in linear time by performing a series of $O(n)$ counter-clockwise tests.

```
JarvisMarch($X[1..n], Y[1..n]$):
    $\ell \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        if $X[i] < X[\ell]$
            $\ell \leftarrow i$

    $p \leftarrow \ell$
    repeat
        $q \leftarrow p + 1$        ⟨⟨Make sure $p \neq q$⟩⟩
        for $i \leftarrow 2$ to $n$
            if CCW$(p, i, q)$
                $q \leftarrow i$
        $next[p] \leftarrow q$;  $prev[q] \leftarrow p$
        $p \leftarrow q$
    until $p = \ell$
```

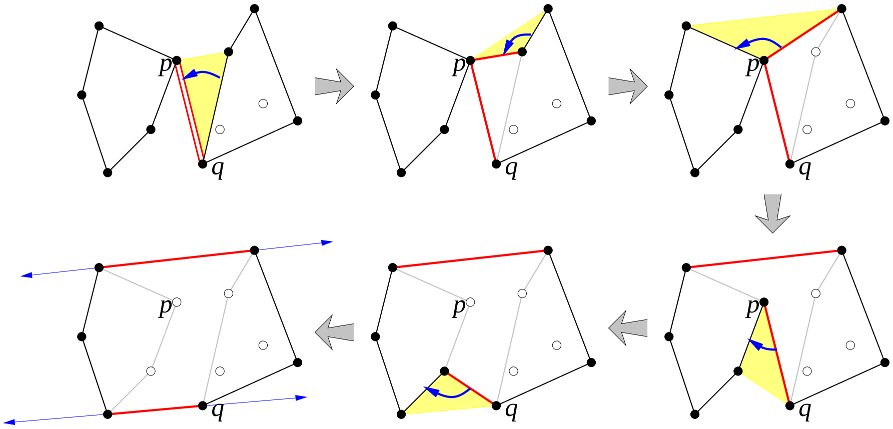
Since the algorithm spends $O(n)$ time for each convex hull vertex, the worst-case running time is $O(n^2)$. However, this naïve analysis hides the fact that if the convex hull has very few vertices, Jarvis's march is extremely fast. A better way to write the running time is $O(nh)$, where h is the number of convex hull vertices. In the worst case, $h = n$, and we get our old $O(n^2)$ time bound, but in the best case $h = 3$, and the algorithm only needs $O(n)$ time. Computational geometers call this an *output-sensitive* algorithm; the smaller the output, the faster the algorithm.

E.4 Divide and Conquer (Splitting)

The behavior of Jarvis's marsh is very much like selection sort: repeatedly find the item that goes in the next slot. In fact, most convex hull algorithms resemble some sorting algorithm.

For example, the following convex hull algorithm resembles quicksort. We start by choosing a *pivot* point p. Partitions the input points into two sets L and R, containing the points to the left of p, including p itself, and the points to the right of p, by comparing x-coordinates. Recursively compute the convex hulls of L and R. Finally, merge the two convex hulls into the final output.

The merge step requires a little explanation. We start by connecting the two hulls with a line segment between the rightmost point of the hull of L with the leftmost point of the hull of R. Call these points p and q, respectively. (Yes, it's the same p.) Actually, let's add *two* copies of the segment $\overline{pq}$ and call them *bridges*. Since p and q can 'see' each other, this creates a sort of dumbbell-shaped polygon, which is convex except possibly at the endpoints off the bridges.

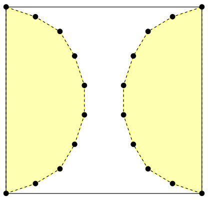


Merging the left and right subhulls.

We now expand this dumbbell into the correct convex hull as follows. As long as there is a clockwise turn at either endpoint of either bridge, we remove that point from the circular sequence of vertices and connect its two neighbors. As soon as the turns at both endpoints of both bridges are counter-clockwise, we can stop. At that point, the bridges lie on the *upper* and *lower common tangent* lines of the two subhulls. These are the two lines that touch both subhulls, such that both subhulls lie below the upper common tangent line and above the lower common tangent line.

Merging the two subhulls takes $O(n)$ time in the worst case. Thus, the running time is given by the recurrence $T(n) = O(n) + T(k) + T(n-k)$, just like quicksort, where k the number of points in R. Just like quicksort, if we use a naïve deterministic algorithm to choose the pivot point p, the worst-case running time of this algorithm is $O(n^2)$. If we choose the pivot point randomly, the expected running time is $O(n \log n)$.

There are inputs where this algorithm is clearly wasteful (at least, clearly to *us*). If we're really unlucky, we'll spend a long time putting together the subhulls, only to throw almost everything away during the merge step. Thus, this divide-and-conquer algorithm is *not* output sensitive.

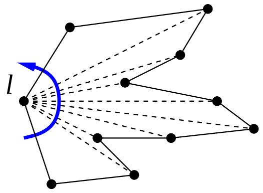


A set of points that shouldn't be divided and conquered.

E.5 Graham's Algorithm (Scanning)

Our third convex hull algorithm, called *Graham's scan*, first explicitly sorts the points in $O(n \log n)$ and then applies a linear-time scanning algorithm to finish building the hull.

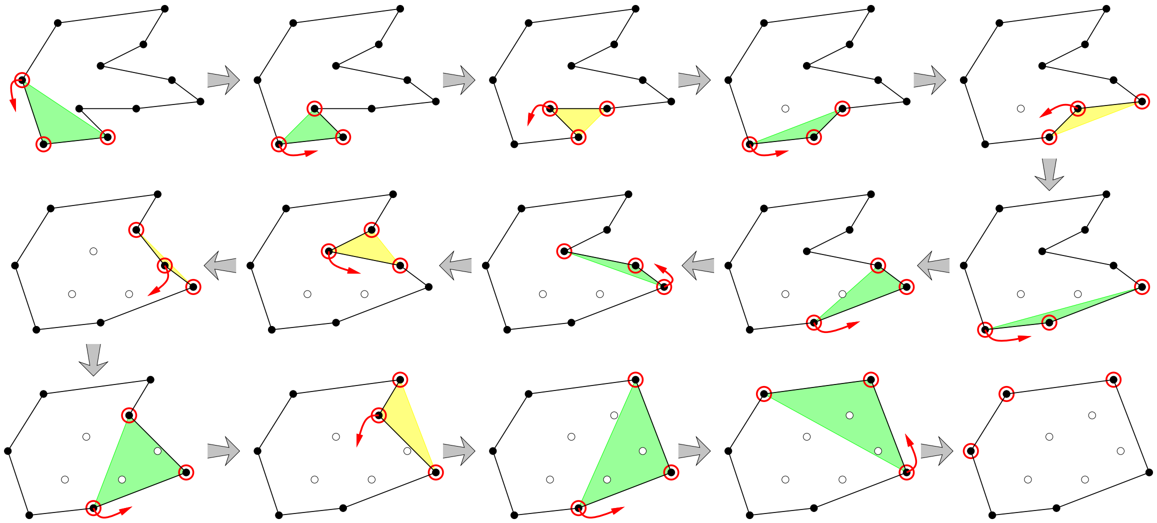
We start Graham's scan by finding the leftmost point ℓ, just as in Jarvis's march. Then we sort the points in counterclockwise order around ℓ. We can do this in $O(n \log n)$ time with any comparison-based sorting algorithm (quicksort, mergesort, heapsort, etc.). To compare two points p and q, we check whether the triple ℓ, p, q is oriented clockwise or counterclockwise. Once the points are sorted, we connected them in counterclockwise order, starting and ending at ℓ. The result is a *simple* polygon with n vertices.



A simple polygon formed in the sorting phase of Graham's scan.

To change this polygon into the convex hull, we apply the following 'three-penny algorithm'. We have three pennies, which will sit on three consecutive vertices p, q, r of the polygon; initially, these are ℓ and the two vertices after ℓ. We now apply the following two rules over and over until a penny is moved forward onto ℓ:

- If p, q, r are in counterclockwise order, move the back penny forward to the successor of r.
- If p, q, r are in clockwise order, remove q from the polygon, add the edge pr, and move the middle penny backward to the predecessor of p.



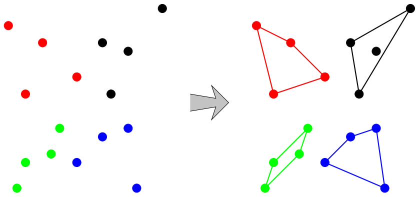
The 'three-penny' scanning phase of Graham's scan.

Whenever a penny moves forward, it moves onto a vertex that hasn't seen a penny before (except the last time), so the first rule is applied $n-2$ times. Whenever a penny moves backwards, a vertex is removed from the polygon, so the second rule is applied exactly $n-h$ times, where h is as usual the number of convex hull vertices. Since each counterclockwise test takes constant time, the scanning phase takes $O(n)$ time altogether.

E.6 Chan's Algorithm (Shattering)

The last algorithm I'll describe is an output-sensitive algorithm that is never slower than either Jarvis's march or Graham's scan. The running time of this algorithm, which was discovered by Timothy Chan in 1993, is $O(n \log h)$. Chan's algorithm is a combination of divide-and-conquer and gift-wrapping.

First suppose a 'little birdie' tells us the value of h; we'll worry about how to implement the little birdie in a moment. Chan's algorithm starts by *shattering* the input points into n/h arbitrary[1] subsets, each of size h, and computing the convex hull of each subset using (say) Graham's scan. This much of the algorithm requires $O((n/h) \cdot h \log h) = O(n \log h)$ time.

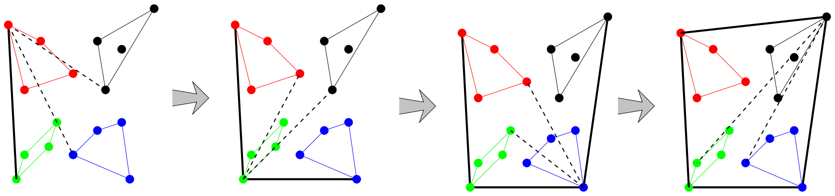


Shattering the points and computing subhulls in $O(n \log h)$ time.

Once we have the n/h subhulls, we follow the general outline of Jarvis's march, 'wrapping a string around' the n/h subhulls. Starting with $p = \ell$, where ℓ is the leftmost input point, we successively find the convex hull vertex the follows p and counterclockwise order until we return back to ℓ again.

The vertex that follows p is the point that appears to be furthest to the right to someone standing at p. This means that the successor of p must lie on a *right tangent line* between p and one of the subhulls—a line from p through a vertex of the subhull, such that the subhull lies completely on the right side of the line from p's point of view. We can find the right tangent line between p and any subhull in $O(\log h)$ time using a variant of binary search. (This is a practice problem in the homework!) Since there are n/h subhulls, finding the successor of p takes $O((n/h) \log h)$ time altogether.

Since there are h convex hull edges, and we find each edge in $O((n/h) \log h)$ time, the overall running time of the algorithm is $O(n \log h)$.



Wrapping the subhulls in $O(n \log h)$ time.

Unfortunately, this algorithm only takes $O(n \log h)$ time if a little birdie has told us the value of h in advance. So how do we implement the 'little birdie'? Chan's trick is to *guess* the correct value of h; let's denote the guess by h^*. Then we shatter the points into n/h^* subsets of size h^*, compute their subhulls, and then find the first h^* edges of the global hull. If $h < h^*$, this algorithm computes the complete convex hull in $O(n \log h^*)$ time. Otherwise, the hull doesn't wrap all the way back around to ℓ, so we know our guess h^* is too small.

[1] In the figures, in order to keep things as clear as possible, I've chosen these subsets so that their convex hulls are disjoint. This is not true in general!

Chan's algorithm starts with the optimistic guess $h^* = 3$. If we finish an iteration of the algorithm and find that h^* is too small, we *square* h^* and try again. Thus, in the ith iteration, we have $h^* = 3^{2^i}$. In the final iteration, $h^* < h^2$, so the last iteration takes $O(n \log h^*) = O(n \log h^2) = O(n \log h)$ time. The total running time of Chan's algorithm is given by the sum

$$\sum_{i=1}^{k} O(n \log 3^{2^i}) = O(n) \cdot \sum_{i=1}^{k} 2^i$$

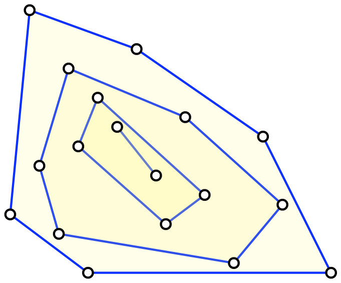
for some integer k. Since any geometric series adds up to a constant times its largest term, the total running time is a constant times the time taken by the last iteration, which is $O(n \log h)$. So Chan's algorithm runs in $O(n \log h)$ time overall, even without the little birdie.

Exercises

1. The *convex layers* of a point set X are defined by repeatedly computing the convex hull of X and removing its vertices from X, until X is empty.

   (a) Describe an algorithm to compute the convex layers of a given set of n points in the plane in $O(n^2)$ time.

   *(b) Describe an algorithm to compute the convex layers of a given set of n points in the plane in $O(n \log n)$ time.



The convex layers of a set of points.

2. Let X be a set of points in the plane. A point p in X is *Pareto-optimal* if no other point in X is both above and to the right of p. The Pareto-optimal points can be connected by horizontal and vertical lines into the *staircase* of X, with a Pareto-optimal point at the top right corner of every step. See the figure on the next page.

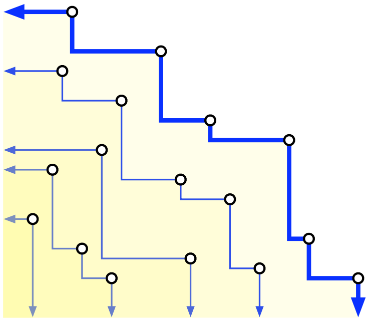
   (a) QUICKSTEP: Describe a divide-and-conquer algorithm to compute the staircase of a given set of n points in the plane in $O(n \log n)$ time.

   (b) SCANSTEP: Describe an algorithm to compute the staircase of a given set of n points in the plane, sorted in left to right order, in $O(n)$ time.

   (c) NEXTSTEP: Describe an algorithm to compute the staircase of a given set of n points in the plane in $O(nh)$ time, where h is the number of Pareto-optimal points.

   (d) SHATTERSTEP: Describe an algorithm to compute the staircase of a given set of n points in the plane in $O(n \log h)$ time, where h is the number of Pareto-optimal points.

   In all these problems, you may assume that no two points have the same x- or y-coordinates.



The staircase (thick line) and staircase layers (all lines) of a set of points.

3. The *staircase layers* of a point set are defined by repeatedly computing the staircase and removing the Pareto-optimal points from the set, until the set becomes empty.

   (a) Describe and analyze an algorithm to compute the staircase layers of a given set of n points in $O(n \log n)$ time.

   (b) An *increasing subsequence* of a point set X is a sequence of points in X such that each point is above and to the right of its predecessor in the sequence. Describe and analyze an algorithm to compute the *longest* increasing subsequence of a given set of n points in the plane in $O(n \log n)$ time. *[Hint: There is a one-line solution that uses part (a). But why is is correct?]*

> **Spengler:** *There's something very important I forgot to tell you.*
> **Venkman:** *What?*
> **Spengler:** *Don't cross the streams.*
> **Venkman:** *Why?*
> **Spengler:** *It would be bad.*
> **Venkman:** *I'm fuzzy on the whole good/bad thing. What do you mean, "bad"?*
> **Spengler:** *Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light.*
> **Stantz:** *Total protonic reversal.*
> **Venkman:** *Right. That's bad. Okay. All right. Important safety tip. Thanks, Egon.*
>
> — *Ghostbusters* (1984)

F Line Segment Intersection

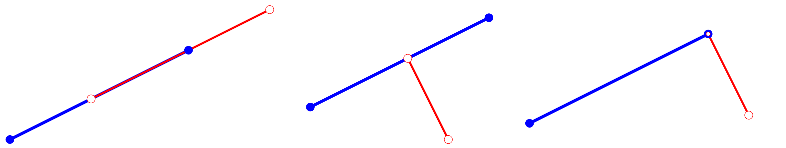
F.1 Introduction

In this lecture, I'll talk about detecting line segment intersections. A line segment is the convex hull of two points, called the *endpoints* (or *vertices*) of the segment. We are given a set of n line segments, each specified by the x- and y-coordinates of its endpoints, for a total of $4n$ real numbers, and we want to know whether any two segments intersect.

To keep things simple, just as in the previous lecture, I'll assume the segments are in *general position*.

- No three endpoints lie on a common line.
- No two endpoints have the same x-coordinate. In particular, no segment is vertical, no segment is just a point, and no two segments share an endpoint.

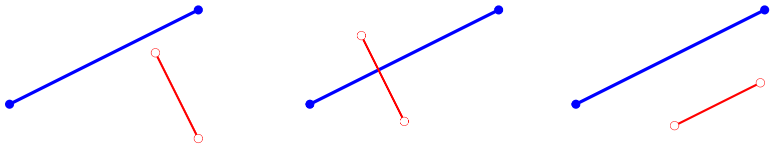
This general position assumption lets us avoid several annoying degenerate cases. Of course, in any real implementation of the algorithm I'm about to describe, you'd have to handle these cases. Real-world data is *full* of degeneracies!



Degenerate cases of intersecting segments that we'll pretend never happen:
Overlapping colinear segments, endpoints inside segments, and shared endpoints.

F.2 Two segments

The first case we have to consider is $n = 2$. (The problem is obviously trivial when $n \leq 1$!) How do we tell whether two line segments intersect? One possibility, suggested by a student in class, is to construct the convex hull of the segments. Two segments intersect if and only if the convex hull is a quadrilateral whose vertices alternate between the two segments. In the figure below, the first pair of segments has a triangular convex hull. The last pair's convex hull is a quadrilateral, but its vertices don't alternate.



Some pairs of segments.

Fortunately, we don't need (or want!) to use a full-fledged convex hull algorithm just to test two segments; there's a much simpler test.

Two segments $\overline{ab}$ and $\overline{cd}$ intersect if and only if

- **the endpoints a and b are on opposite sides of the line $\overleftrightarrow{cd}$, and**
- **the endpoints c and d are on opposite sides of the line $\overleftrightarrow{ab}$.**

To test whether two points are on opposite sides of a line through two other points, we use the same counterclockwise test that we used for building convex hulls. Specifically, a and b are on opposite sides of line $\overleftrightarrow{cd}$ if and only if exactly one of the two triples a, c, d and b, c, d is in counterclockwise order. So we have the following simple algorithm.

```
Intersect$(a, b, c, d)$:
  if CCW$(a, c, d)$ = CCW$(b, c, d)$
      return False
  else if CCW$(a, b, c)$ = CCW$(a, b, d)$
      return False
  else
      return True
```

Or even simpler:

```
Intersect$(a, b, c, d)$:
  return $\big[$CCW$(a, c, d) \neq$ CCW$(b, c, d)\big] \wedge \big[$CCW$(a, b, c) \neq$ CCW$(a, b, d)\big]$
```

F.3 A Sweep Line Algorithm

To detect whether there's an intersection in a set of more than just two segments, we use something called a *sweep line* algorithm. First let's give each segment a unique *label*. I'll use letters, but in a real implementation, you'd probably use pointers/references to records storing the endpoint coordinates.

Imagine sweeping a vertical line across the segments from left to right. At each position of the sweep line, look at the sequence of (labels of) segments that the line hits, sorted from top to bottom. The only times this sorted sequence can change is when the sweep line passes an endpoint or when the sweep line passes an intersection point. In the second case, the order changes because two adjacent labels swap places.[1] Our algorithm will simulate this sweep, looking for potential swaps between adjacent segments.

The sweep line algorithm begins by sorting the $2n$ segment endpoints from left to right by comparing their x-coordinates, in $O(n \log n)$ time. The algorithm then moves the sweep line from left to right, stopping at each endpoint.

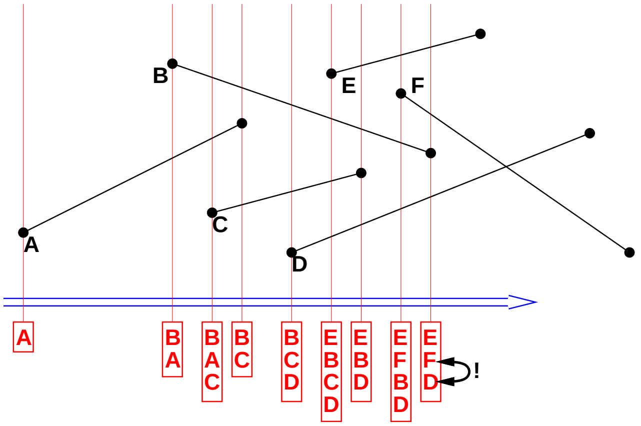
We store the vertical label sequence in some sort of balanced binary tree that supports the following operations in $O(\log n)$ time. Note that the tree does not store any explicit search keys, only segment labels.

- **Insert** a segment label.
- **Delete** a segment label.

[1] Actually, if more than two segments intersect at the same point, there could be a larger reversal, but this won't have any effect on our algorithm.

- Find the **neighbors** of a segment label in the sorted sequence.

$O(\log n)$ amortized time is good enough, so we could use a scapegoat tree or a splay tree. If we're willing to settle for an expected time bound, we could use a treap or a skip list instead.



The sweep line algorithm in action. The boxes show the label sequence stored in the binary tree. The intersection between F and D is detected in the last step.

Whenever the sweep line hits a left endpoint, we insert the corresponding label into the tree in $O(\log n)$ time. In order to do this, we have to answer questions of the form 'Does the new label X go above or below the old label Y?' To answer this question, we test whether the new left endpoint of X is above segment Y, or equivalently, if the triple of endpoints left(Y), right(Y), left(X) is in counterclockwise order.

Once the new label is inserted, we test whether the new segment intersects either of its two neighbors in the label sequence. For example, in the figure above, when the sweep line hits the left endpoint of F, we test whether F intersects either B or E. These tests require $O(1)$ time.

Whenever the sweep line hits a right endpoint, we delete the corresponding label from the tree in $O(\log n)$ time, and then check whether its two neighbors intersect in $O(1)$ time. For example, in the figure, when the sweep line hits the right endpoint of C, we test whether B and D intersect.

If at any time we discover a pair of segments that intersects, we stop the algorithm and report the intersection. For example, in the figure, when the sweep line reaches the right endpoint of B, we discover that F and D intersect, and we halt. Note that we may not discover the intersection until long after the two segments are inserted, and the intersection we discover may not be the one that the sweep line would hit first. It's not hard to show by induction (hint, hint) that the algorithm is correct. Specifically, if the algorithm reaches the nth right endpoint without detecting an intersection, none of the segments intersect.

For each segment endpoint, we spend $O(\log n)$ time updating the binary tree, plus $O(1)$ time performing pairwise intersection tests—at most two at each left endpoint and at most one at each right endpoint. Thus, the entire sweep requires $O(n \log n)$ time in the worst case. Since we also spent $O(n \log n)$ time sorting the endpoints, the overall running time is $O(n \log n)$.

Here's a slightly more formal description of the algorithm. The input $S[1 .. n]$ is an array of line segments. The sorting phase in the first line produces two auxiliary arrays:

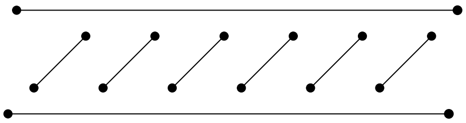
- *label*$[i]$ is the label of the ith leftmost endpoint. I'll use indices into the input array S as the labels, so the ith vertex is an endpoint of $S[$*label*$[i]]$.

- *isleft*$[i]$ is TRUE if the ith leftmost endpoint is a left endpoint and FALSE if it's a right endpoint.

The functions INSERT, DELETE, PREDECESSOR, and SUCCESSOR modify or search through the sorted label sequence. Finally, INTERSECT tests whether two line segments intersect.

```
ANYINTERSECTIONS($S[1..n]$):
  sort the endpoints of $S$ from left to right
  create an empty label sequence
  for $i \leftarrow 1$ to $2n$
      $\ell \leftarrow label[i]$
      if $isleft[i]$
          INSERT($\ell$)
          if INTERSECT($S[\ell]$, $S$[SUCCESSOR($\ell$)])
              return TRUE
          if INTERSECT($S[\ell]$, $S$[PREDECESSOR($\ell$)])
              return TRUE
      else
          if INTERSECT($S$[SUCCESSOR($\ell$)], $S$[PREDECESSOR($\ell$)])
              return TRUE
          DELETE($label[i]$)
  return FALSE
```

Note that the algorithm doesn't try to avoid redundant pairwise tests. In the figure below, the top and bottom segments would be checked $n-1$ times, once at the top left endpoint, and once at the right endpoint of every short segment. But since we've already spent $O(n \log n)$ time just sorting the inputs, $O(n)$ redundant segment intersection tests make no difference in the overall running time.



The same pair of segments might be tested $n-1$ times.

Exercises

1. Let X be a set of n rectangles in the plane, each specified by a left and right x-coordinate and a top and bottom y-coordinate. Thus, the input consists of four arrays $L[1..n]$, $R[1..n]$, $T[1..n]$, and $B[1..n]$, where $L[i] < R[i]$ and $T[i] > B[i]$ for all i.

   (a) Describe and analyze and algorithm to determine whether any two rectangles in X intersect, in $O(n \log n)$ time.

   (b) Describe and analyze an algorithm to find a point that lies inside the largest number of rectangles in X, in $O(n \log n)$ time.

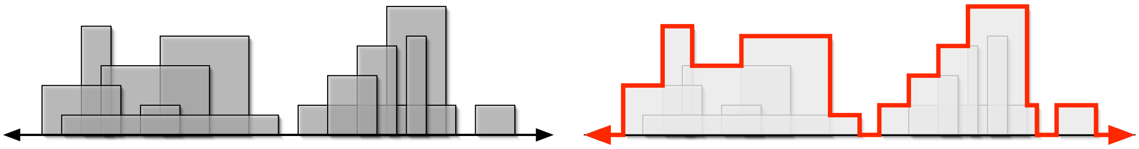
   (c) Describe and analyze an algorithm to compute the area of the union of the rectangles in X, in $O(n \log n)$ time.

2. Describe and analyze a sweepline algorithm to determine, given n circles in the plane, whether any two intersect, in $O(n \log n)$ time. Each circle is specified by its center and its radius, so the input consists of three arrays $X[1..n]$, $Y[1..n]$, and $R[1..n]$. Be careful to correctly implement the low-level primitives.

3. Describe an algorithm to determine, given n line segments in the plane, a list of all intersecting pairs of segments. Your algorithm should run in $O(n \log n + k \log n)$ time, where n is the number of segments, and k is the number of intersecting pairs.

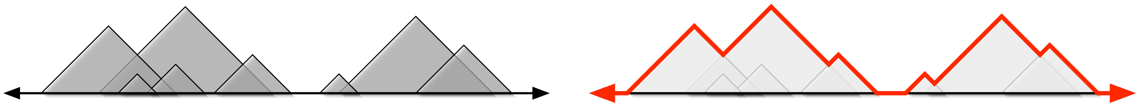
4. This problem asks you to compute *skylines* of various cities. In each city, all the buildings have a signature geometric shape. Describe an algorithm to compute a description of the union of n such shapes in $O(n \log n)$ time.

   (a) Manhattan: Each building is a rectangle whose bottom edge lies on the x-axis, specified by the left and right x-coordinates and the top y-coordinate.



The Manhattan skyline

   (b) Giza: Each building is a right isosceles triangle whose base lies on the x-axis, specified by the (x, y)-coordinates of its apex.



The Egyptian skyline

   (c) Nome: Each building is a semi-circular disk whose center lies on the x-axis, specified by its center x-coordinate and radius.

5. *[Adapted from CLRS, problem 33-3.]* A group of n Ghostbusters are teaming up to fight n ghosts. Each Ghostbuster is armed with a proton pack that shoots a stream along a straight line until it encounters (and neutralizes) a ghost. The Ghostbusters decide that they will fire their proton packs simultaneously, with each Ghostbuster aiming at a different ghost. Crossing the streams would be bad—total protonic reversal, yadda yadda—so it is vital that each Ghostbuster choose his target carefully. Assume that each Ghostbuster and each ghost is a single point in the plane, and that no three of these $2n$ points are collinear.

   (a) Prove that there is a set of n disjoint line segments, each joining one Ghostbuster to one ghost. *[Hint: Consider the matching of minimum total length.]*

   (b) Prove that the non-collinearity assumption is necessary in part (a).

   (c) A *partitioning line* is a line in the plane such that the number of ghosts on one side of the line is equal to the number of Ghostbusters on the same side of that line. Describe an algorithm to compute a partitioning line in $O(n \log n)$ time.

   (d) Prove that there is a partitioning line with exactly $\lfloor n/2 \rfloor$ ghosts and exactly $\lfloor n/2 \rfloor$ Ghostbusters on either side. This is a special case of the so-called *ham sandwich theorem*. Describe an algorithm to compute such a line as quickly as possible.

*(e) Describe a randomized algorithm to find an *approximate* ham-sandwich line—that is, a partitioning line with at least $n/4$ ghosts on each side—in $O(n \log n)$ time.

(f) Describe an algorithm to compute a set of n disjoint line segments, each joining one Ghostbuster to one ghost, as quickly as possible.

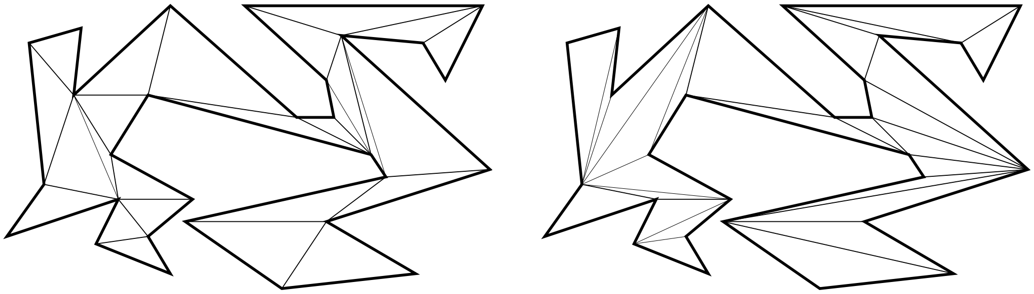
If triangles had a god, they would give him three sides.
— Charles Louis de Secondat Montesquie (1721)

Down with Euclid! Death to triangles!
— Jean Dieudonné (1959)

G Polygon Triangulation

G.1 Introduction

Recall from last time that a *polygon* is a region of the plane bounded by a cycle of straight edges joined end to end. Given a polygon, we want to decompose it into triangles by adding *diagonals*: new line segments between the vertices that don't cross the boundary of the polygon. Because we want to keep the number of triangles small, we don't allow the diagonals to cross. We call this decomposition a *triangulation* of the polygon. Most polygons can have more than one triangulation; we don't care which one we compute.

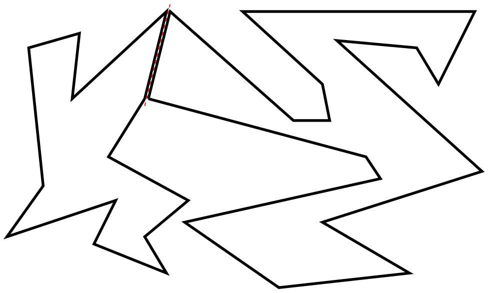


Two triangulations of the same polygon.

Before we go any further, I encourage you to play around with some examples. Draw a few polygons (making sure that the edges are straight and don't cross) and try to break them up into triangles.

G.2 Existence and Complexity

If you play around with a few examples, you quickly discover that every triangulation of an n-sided has $n-2$ triangles. You might even try to prove this observation by induction. The base case $n=3$ is trivial: there is only one triangulation of a triangle, and it obviously has only one triangle! To prove the general case, let P be a polygon with n edges. Draw a diagonal between two vertices. This splits P into two smaller polygons. One of these polygons has k edges of P plus the diagonal, for some integer k between 2 and $n-2$, for a total of $k+1$ edges. So by the induction hypothesis, this polygon can be broken into $k-1$ triangles. The other polygon has $n-k+1$ edges, and so by the induction hypothesis, it can be broken into $n-k-1$ tirangles. Putting the two pieces back together, we have a total of $(k-1)+(n-k-1)=n-2$ triangles.



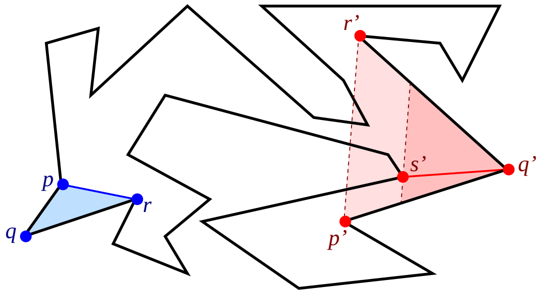
Breaking a polygon into two smaller polygons with a diagonal.

This is a fine induction proof, which any of you could have discovered on your own (right?), except for one small problem. How do we know that every polygon *has* a diagonal? This seems patently obvious, but it's surprisingly hard to prove, and in fact many incorrect proofs were actually published as late as 1975. The following proof is due to Meisters in 1975.

Lemma 1. *Every polygon with more than three vertices has a diagonal.*

Proof: Let P be a polygon with more than three vertices. Every vertex of a P is either *convex* or *concave*, depending on whether it points into or out of P, respectively. Let q be a convex vertex, and let p and r be the vertices on either side of q. For example, let q be the leftmost vertex. (If there is more than one leftmost vertex, let q be the the lowest one.) If $\overline{pr}$ is a diagonal, we're done; in this case, we say that the triangle $\triangle pqr$ is an *ear*.

If pr is not a diagonal, then $\triangle pqr$ must contain another vertex of the polygon. Out of all the vertices inside $\triangle pqr$, let s be the vertex furthest away from the line $\overleftrightarrow{pr}$. In other words, if we take a line parallel to $\overleftrightarrow{pr}$ through q, and translate it towards $\overleftrightarrow{pr}$, then then s is the first vertex that the line hits. Then the line segment $\overline{qs}$ is a diagonal. □



The leftmost vertex q is the tip of an ear, so pr is a diagonal.
The rightmost vertex q' is not, since $\triangle p'q'r'$ contains three other vertices. In this case, $q's'$ is a diagonal.

G.3 Existence and Complexity

Meister's existence proof immediately gives us an algorithm to compute a diagonal in linear time. The input to our algorithm is just an array of vertices in counterclockwise order around the polygon. First, we can find the (lowest) leftmost vertex q in $O(n)$ time by comparing the x-coordinates of the vertices (using y-coordinates to break ties). Next, we can determine in $O(n)$ time whether the triangle $\triangle pqr$ contains any of the other $n-3$ vertices. Specifically, we can check whether one point lies inside a triangle by performing three counterclockwise tests. Finally, if the triangle is not empty, we can find the vertex s in $O(n)$ time by comparing the areas of every triangle $\triangle pqs$; we can compute this area using the counterclockwise determinant.

Here's the algorithm in excruciating detail. We need three support subroutines to compute the area of a polygon, to determine if three poitns are in counterclockwise order, and to determine if a point is inside a triangle.

```
⟨⟨Return twice the signed area of $\triangle P[i]P[j]P[k]$⟩⟩
AREA$(i, j, k)$:
  return $(P[k].y - P[i].y)(P[j].x - P[i].x) - (P[k].x - P[i].x)(P[j].y - P[i].y)$
```

```
⟨⟨Are $P[i], P[j], P[k]$ in counterclockwise order?⟩⟩
CCW$(i, j, k)$:
  return AREA$(i, j, k) > 0$
```

```
⟨⟨Is $P[i]$ inside $\triangle P[p]P[q]P[r]$?⟩⟩
INSIDE$(i, p, q, r)$:
  return CCW$(i, p, q)$ and CCW$(i, q, r)$ and CCW$(i, r, p)$
```

```
FINDDIAGONAL$(P[1..n])$:
  $q \leftarrow 1$
  for $i \leftarrow 2$ to $n$
      if $P[i].x < P[q].x$
          $q \leftarrow i$
  $p \leftarrow q - 1 \bmod n$
  $r \leftarrow q + 1 \bmod n$

  $ear \leftarrow$ TRUE
  $s \leftarrow p$
  for $i \leftarrow 1$ to $n$
      if $i \leq p$ and $i \neq q$ and $i \neq r$ and INSIDE$(i, p, q, r)$
          $ear \leftarrow$ FALSE
          if AREA$(i, r, p) >$ AREA$(s, r, p)$
              $s \leftarrow i$

  if $ear =$ TRUE
      return $(p, r)$
  else
      return $(q, s)$
```

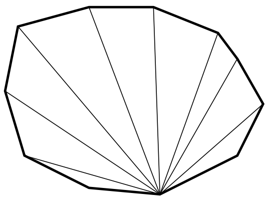
Once we have a diagonal, we can recursively triangulate the two pieces. The worst-case running time of this algorithm satisfies almost the same recurrence as quicksort:

$$T(n) \leq \max_{2 \leq k \leq n-2} T(k+1) + T(n-k+1) + O(n).$$

So we can now triangulate any polygon in $O(n^2)$ time.

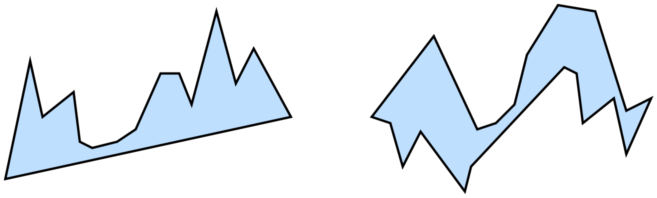
G.4 Faster Special Cases

For certain special cases of polygons, we can do much better than $O(n^2)$ time. For example, we can easily triangulate any convex polygon by connecting any vertex to every other vertex. Since we're given the counterclockwise order of the vertices as input, this takes only $O(n)$ time.



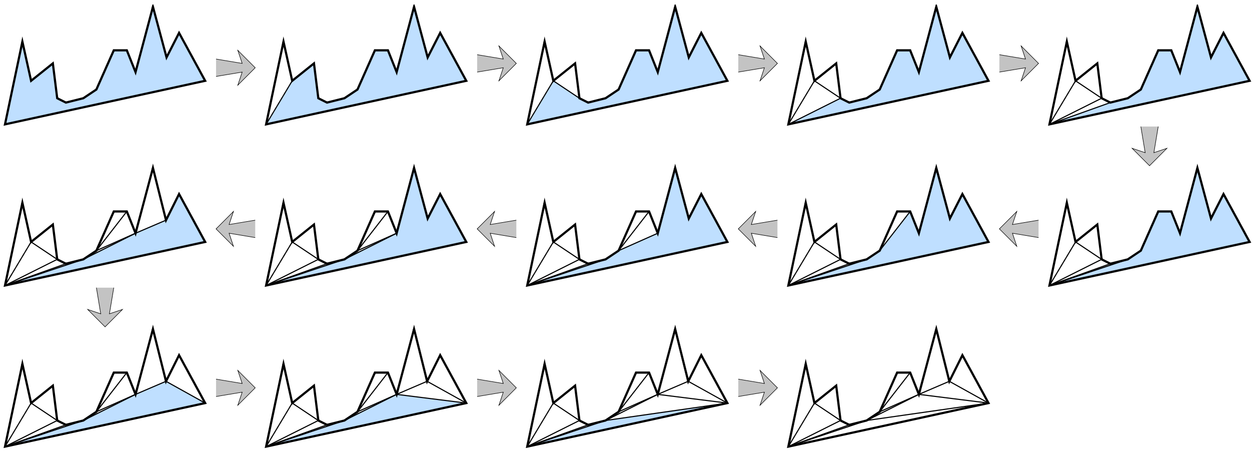
Triangulating a convex polygon is easy.

Another easy special case is *monotone mountains*. A polygon is *monotone* if any vertical line intersects the boundary in at most two points. A monotone polygon is a *mountain* if it contains an edge from the rightmost vertex to the leftmost vertex. Every monotone polygon consists of two chains of edges going left to right between the two extreme vertices; for mountains, one of these chains is a single edge.



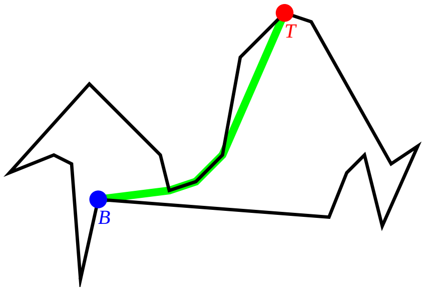
A monotone mountain and a monotone non-mountain.

Triangulating a monotone mounting is extremely easy, since every convex vertex is the tip of an ear, except possibly for the vertices on the far left and far right. Thus, all we have to do is scan through the intermediate vertices, and when we find a convex vertex, cut off the ear. The simplest method for doing this is probably the three-penny algorithm used in the "Graham's scan" convex hull algorithm—instead of filling in the outside of a polygon with triangles, we're filling in the inside, both otherwise it's the same process. This takes $O(n)$ time.



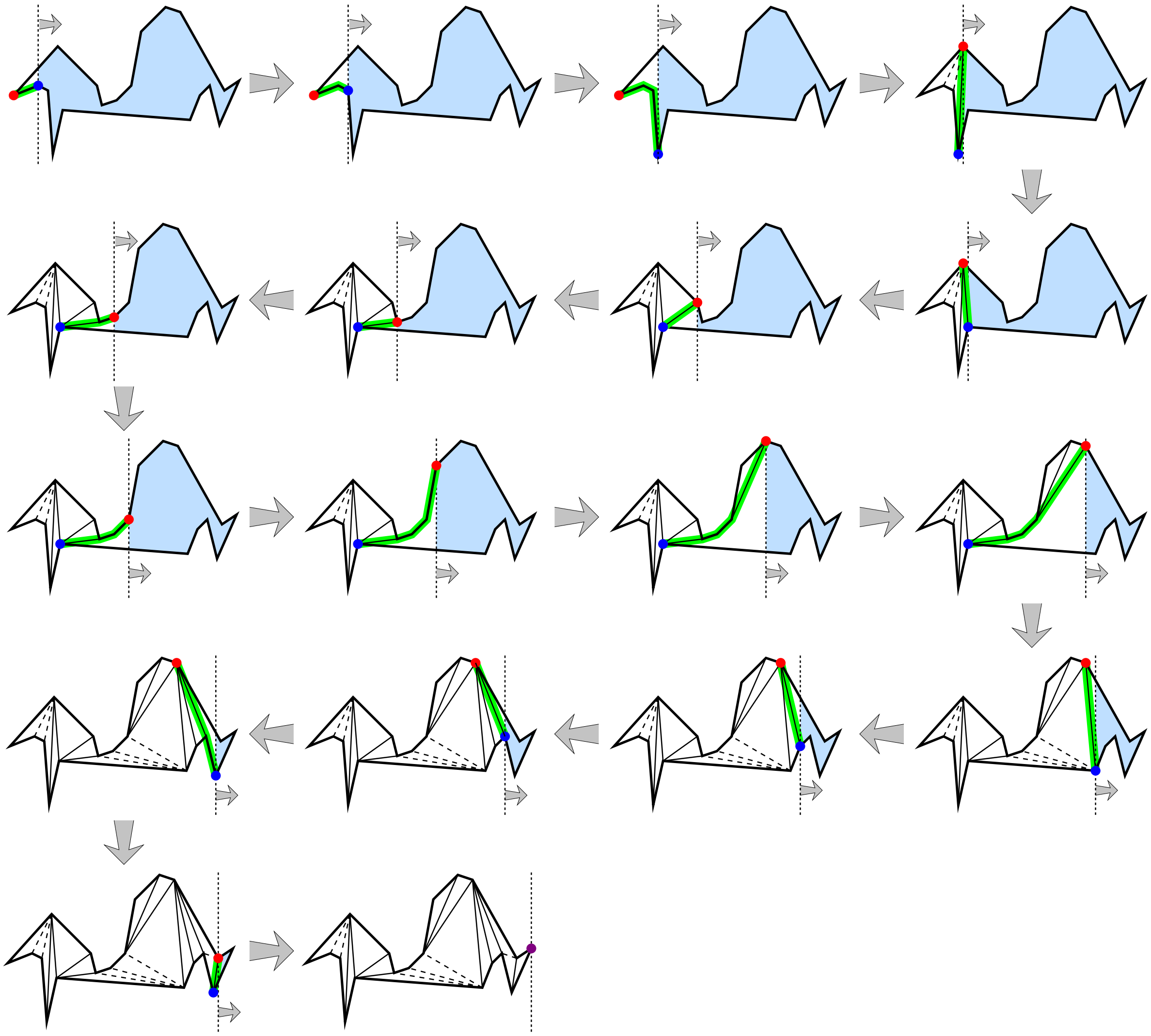
Triangulating a monotone mountain. (Some of the triangles are very thin.)

We can also triangulate general monotone polygons in linear time, but the process is more complicated. A good way to visualize the algorithm is to think of the polygon as a complicated room. Two people named Tom and Bob are walking along the top and bottom walls, both starting at the left end and going to the right. At all times, they have a rubber band stretched between them that can never leave the room.



A rubber band stretched between a vertex on the top and a vertex on the bottom of a monotone polygon.

Now we loop through *all* the vertices of the polygon in order from left to right. Whenever we see a new bottom vertex, Bob moves onto it, and whenever we see a new bottom vertex Tom moves onto it. After either person moves, we cut the polygon along the rubber band. (In fact, this will only cut the polygon along a single diagonal at any step.) When we're done, the polygon is decomposed into triangles and *boomerangs*—nonconvex polygons consisting of two straight edges and a concave chain. A boomerang can only be triangulated in one way, by joining the *apex* to every vertex in the concave chain.



Triangulating a monotone polygon by walking a rubber band from left to right.

I don't want to go into too many implementation details, but a few observations shoudl convince you that this algorithm can be implemented to run in $O(n)$ time. Notice that at all times, the rubber band forms a concave chain. The leftmost edge in the rubber band joins a top vertex to a bottom vertex. If the rubber band has any other vertices, either they are all on top or all on bottom. If all the other vertices are on top, there are just three ways the rubber band can change:

1. The bottom vertex changes, the rubber band straighens out, and we get a new boomerang.

2. The top vertex changes and the rubber band gets a new concave vertex.

3. The top vertex changes, the rubber band loses some vertices, and we get a new boomerang.

Deciding between the first case and the other two requires a simple comparison between x-coordinates. Deciding between the last two requires a counterclockwise test.

> *It was a Game called Yes and No, where Scrooge's nephew had to think of something, and the rest must find out what; he only answering to their questions yes or no, as the case was. The brisk fire of questioning to which he was exposed, elicited from him that he was thinking of an animal, a live animal, rather a disagreeable animal, a savage animal, an animal that growled and grunted sometimes, and talked sometimes, and lived in London, and walked about the streets, and wasn't made a show of, and wasn't led by anybody, and didn't live in a menagerie, and was never killed in a market, and was not a horse, or an ass, or a cow, or a bull, or a tiger, or a dog, or a pig, or a cat, or a bear. At every fresh question that was put to him, this nephew burst into a fresh roar of laughter; and was so inexpressibly tickled, that he was obliged to get up off the sofa and stamp. At last the plump sister, falling into a similar state, cried out:*
>
> *"I have found it out! I know what it is, Fred! I know what it is!"*
>
> *"What is it?" cried Fred.*
>
> *"It's your Uncle Scrooge!"*
>
> *Which it certainly was. Admiration was the universal sentiment, though some objected that the reply to "Is it a bear?" ought to have been "Yes," inasmuch as an answer in the negative was sufficient to have diverted their thoughts from Mr Scrooge, supposing they had ever had any tendency that way.*
>
> — Charles Dickens, *A Christmas Carol*' (1843)

19 Lower Bounds

19.1 What *Are* Lower Bounds?

So far in this class we've been developing algorithms and data structures for solving certain problems and analyzing their time and space complexity.

Let $T_A(X)$ denote the running of algorithm A given input X. Recall that the worst-case running time of A for inputs of size n is defined as follows:

$$T_A(n) = \max_{|X|=n} T_A(X).$$

The worst-case complexity of a *problem* Π is the worst-case running time of the *fastest* algorithm for solving it:

$$T_\Pi(n) = \min_{A \text{ solves } \Pi} T_A(n) = \min_{A \text{ solves } \Pi} \max_{|X|=n} T_A(X).$$

Whenever we describe an algorithm A that solves Π in $O(f(n))$ time, we immediately have an *upper bound* on the complexity of Π:

$$T_\Pi(n) \le T_A(n) = O(f(n)).$$

The faster our algorithm, the better our upper bound. In other words, when we give a running time for an algorithm, what we're really doing—and what most computer scientists devote their entire careers doing[1]—is bragging about how *easy* some problem is.

[1] This sometimes leads to long sequences of results that sound like an obscure version of "Name that Tune":

> Lennes: "I can triangulate that polygon in $O(n^2)$ time."
> Shamos: "I can triangulate that polygon in $O(n \log n)$ time."
> Tarjan: "I can triangulate that polygon in $O(n \log\log n)$ time."
> Seidel: "I can triangulate that polygon in $O(n \log^* n)$ time." [Audience gasps.]
> Chazelle: "I can triangulate that polygon in $O(n)$ time." [Audience gasps and applauds.]
> "Triangulate that polygon!"

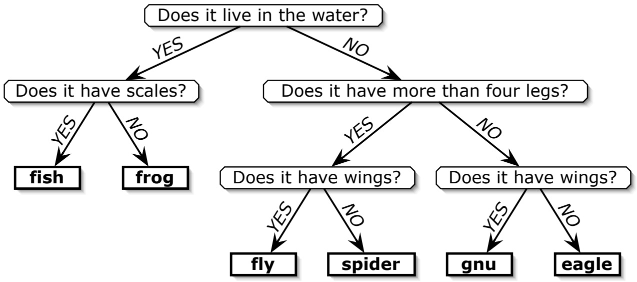
Starting with this lecture, we'll turn the tables. Instead of bragging about how easy problems are, we will argue that certain problems are *hard*, by proving *lower bounds* on their complexity. This is considerably harder than proving an upper bound, because it's no longer enough to examine a single algorithm. To show that $T_\Pi(n) = \Omega(f(n))$, we have to prove that *every* algorithm that solves Π has a worst-case running time $\Omega(f(n))$, or equivalently, that *no* algorithm runs in $o(f(n))$ time.

19.2 Decision Trees

Unfortunately, there is no formal definition of the phrase 'all algorithms'![2] So when we derive lower bounds, we first have to *formally* specify *precisely* what kinds of algorithms we will consider and *precisely* how to measure their running time. This specification is called a *model of computation*.

One rather powerful model of computation—and essentially the only model we'll talk about this class—is *decision trees*. A decision tree is, as the name suggests, a tree. Each internal node in the tree is labeled by a *query*, which is just a question about the input. The edges out of a node correspond to the possible answers to that node's query. Each leaf of the tree is labeled with an *output*. To compute with a decision tree, we start at the root and follow a path down to a leaf. At each internal node, the answer to the query tells us which node to visit next. When we reach a leaf, we output its label.

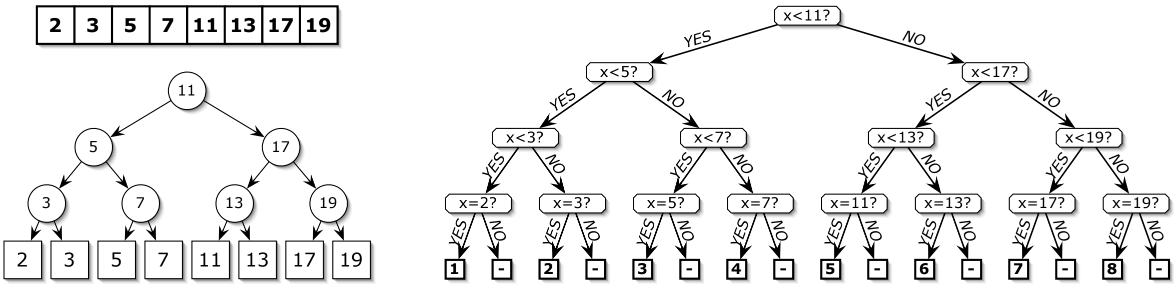
For example, the guessing game where one person thinks of an animal and the other person tries to figure it out with a series of yes/no questions can be modeled as a decision tree. Each internal node is labeled with a question and has two edges labeled 'yes' and 'no'. Each leaf is labeled with an animal.



A decision tree to choose one of six animals.

Here's another simple example, called the *dictionary problem*. Let A be a fixed array with n numbers. Suppose want to determine, given a number x, the position of x in the array A, if any. One solution to the dictionary problem is to sort A (remembering every element's original position) and then use binary search. The (implicit) binary search tree can be used almost directly as a decision tree. Each internal node the the *search* tree stores a key k; the corresponding node in the *decision* tree stores the question 'Is $x < k$?'. Each leaf in the *search* tree stores some value $A[i]$; the corresponding node in the *decision* tree asks 'Is $x = A[i]$?' and has two leaf children, one labeled 'i' and the other 'none'.

[2] Complexity-theory ~~snobs~~ purists will argue that 'all algorithms' is just a synonym for 'all Turing machines'. This is utter nonsense; Turing machines are just another model of computation. Turing machines *might* be a reasonable model of *physically realizable* computation—that's the Church-Turing thesis—but it has a few problems. First, computation is an abstract mathematical process, not a physical process. Algorithms that use physically unrealistic components (like exact real numbers) are still mathematically well-defined and still provide useful intuition about real-world computation. Moreover, Turing machines don't accurately reflect the complexity of physically realizable algorithms, because (for example) they can't do arithmetic or access arbitrary memory locations in constant time. At best, they estimate algorithmic complexity up to polynomial factors (although even that is unknown).



Left: A binary search tree for the first eight primes.
Right: The corresponding binary decision tree for the dictionary problem ($-$ = 'none').

We *define* the running time of a decision tree algorithm for a given input to be the number of queries in the path from the root to the leaf. For example, in the 'Guess the animal' tree above, $T(\text{frog}) = 2$. Thus, the worst-case running time of the algorithm is just the depth of the tree. This definition ignores other kinds of operations that the algorithm might perform that have nothing to do with the queries. (Even the most efficient binary search problem requires more than one machine instruction per comparison!) But the number of decisions is certainly a *lower bound* on the actual running time, which is good enough to prove a lower bound on the complexity of a problem.

Both of the examples describe *binary* decision trees, where every query has only two answers. We may sometimes want to consider decision trees with higher degree. For example, we might use queries like 'Is x greater than, equal to, or less than y?' or 'Are these three points in clockwise order, colinear, or in counterclockwise order?' A *k-ary* decision tree is one where every query has (at most) k different answers. **From now on, I will only consider k-ary decision trees where k is a constant.**

19.3 Information Theory

Most lower bounds for decision trees are based on the following simple observation: *the answers to the queries must give you enough information to specify any possible output.* If a problem has N different outputs, then obviously any decision tree must have at least N leaves. (It's possible for several leaves to specify the same output.) Thus, if every query has at most k possible answers, then the depth of the decision tree must be at least $\lceil \log_k N \rceil = \Omega(\log N)$.

Let's apply this to the dictionary problem for a set S of n numbers. Since there are $n+1$ possible outputs, any decision tree must have at least $n+1$ leaves, and thus any decision tree must have depth at least $\lceil \log_k (n+1) \rceil = \Omega(\log n)$. So the complexity of the dictionary problem, in the decision-tree model of computation, is $\Omega(\log n)$. This matches the upper bound $O(\log n)$ that comes from a perfectly-balanced binary search tree. That means that the standard binary search algorithm, which runs in $O(\log n)$ time, is *optimal*—there is no faster algorithm in this model of computation.

19.4 But wait a second...

We can solve the membership problem in $O(1)$ expected time using hashing. Isn't this inconsistent with the $\Omega(\log n)$ lower bound?

No, it isn't. The reason is that hashing involves a query with more than a constant number of outcomes, specifically 'What is the hash value of x?' In fact, if we don't restrict the degree of the decision tree, we can get constant running time even without hashing, by using the obviously unreasonable query 'For which index i (if any) is $A[i] = x$?'. No, I am *not* cheating — remember that the decision tree model allows us to ask *any* question about the input!

This example illustrates a common theme in proving lower bounds: *choosing the right model of computation is absolutely crucial.* If you choose a model that is too powerful, the problem you're studying may have a completely trivial algorithm. On the other hand, if you consider more restrictive models, the problem may not be solvable at all, in which case any lower bound will be meaningless! (In this class, we'll just tell you the right model of computation to use.)

19.5 Sorting

Now let's consider the *sorting* problem — Given an array of n numbers, arrange them in increasing order. Unfortunately, decision trees don't have any way of describing moving data around, so we have to rephrase the question slightly:

> Given a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ of n distinct numbers, find the permutation π such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$.

Now a k-ary decision-tree lower bound is immediate. Since there are $n!$ possible permutations π, any decision tree for sorting must have at least $n!$ leaves, and so must have depth $\Omega(\log(n!))$. To simplify the lower bound, we apply *Stirling's approximation*

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n.$$

This gives us the lower bound

$$\lceil \log_k(n!) \rceil > \left\lceil \log_k \left(\frac{n}{e}\right)^n \right\rceil = \lceil n \log_k n - n \log_k e \rceil = \Omega(n \log n).$$

This matches the $O(n \log n)$ upper bound that we get from mergesort, heapsort, or quicksort, so those algorithms are optimal. The decision-tree complexity of sorting is $\Theta(n \log n)$.

Well. . . we're not quite done. In order to say that those algorithms are optimal, we have to demonstrate that they fit into our model of computation. A few minutes of thought will convince you that they can be described as a special type of decision tree called a *comparison* tree, where every query is of the form 'Is x_i bigger or smaller than x_j?' These algorithms treat any two input sequences exactly the same way as long as the same comparisons produce exactly the same results. This is a feature of any comparison tree. In other words, *the actual input values don't matter, only their order*. Comparison trees describe almost all sorting algorithms: bubble sort, selection sort, insertion sort, shell sort, quicksort, heapsort, mergesort, and so forth—but *not* radix sort or bucket sort.

19.6 Finding the Maximum and Adversaries

Finally let's consider the *maximum* problem: Given an array X of n numbers, find its largest entry. Unfortunately, there's no hope of proving a lower bound in this formulation, since there are an infinite number of possible answers, so let's rephrase it slightly.

> Given a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ of n distinct numbers, find the index m such that x_m is the largest element in the sequence.

We can get an upper bound of $n - 1$ comparisons in several different ways. The easiest is probably to start at one end of the sequence and do a linear scan, maintaining a current maximum. Intuitively, this seems like the best we can do, but the information-theoretic bound is only $\lceil \log_2 n \rceil$.

And in fact, this bound is tight! We can locate the maximum element by asking only $\lceil \log_2 n \rceil$ 'unreasonable' questions like "Is the index of the maximum element odd?" No, this is *not* cheating—the decision tree model allows *arbitrary* questions.

To prove a non-trivial lower bound for this problem, we must do two things. First, we need to consider a more reasonable model of computation, by restricting the kinds of questions the algorithm is allowed to ask. We will consider the *comparison tree model*, where every query must have the form "Is $x_i > x_j$?". Since most algorithms[3] for finding the maximum rely on comparisons to make control-flow decisions, this does not seem like an unreasonable restriction.

Second, we will use something called an *adversary argument*. The idea is that an all-powerful malicious adversary *pretends* to choose an input for the algorithm. When the algorithm asks a question about the input, the adversary answers in whatever way will make the algorithm do the most work. If the algorithm does not ask enough queries before terminating, then there will be several different inputs, each consistent with the adversary's answers, the should result in different outputs. In this case, whatever the algorithm outputs, the adversary can 'reveal' an input that is consistent with its answers, but contradicts the algorithm's output, and then claim that that was the input that he was using all along.

For the maximum problem, the adversary originally pretends that $x_i = i$ for all i, and answers all comparison queries accordingly. Whenever the adversary reveals that $x_i < x_j$, he *marks* x_i as an item that the algorithm knows (or should know) is not the maximum element. At most one element x_i is marked after each comparison. Note that x_n is never marked. If the algorithm does less than $n-1$ comparisons before it terminates, the adversary must have at least one other unmarked element $x_k \neq x_n$. In this case, the adversary can change the value of x_k from k to $n+1$, making x_k the largest element, without being inconsistent with any of the comparisons that the algorithm has performed. In other words, the algorithm cannot tell that the adversary has cheated. However, x_n is the maximum element in the original input, and x_k is the largest element in the modified input, so the algorithm cannot possibly give the correct answer for both cases. Thus, in order to be correct, any algorithm must perform at least $n-1$ comparisons.

The adversary argument we described has two very important properties. First, no algorithm can distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully. But much more importantly, **the adversary makes *absolutely no* assumptions about the order in which the algorithm performs comparisons.** The adversary forces *any* comparison-based algorithm[4] to either perform $n-1$ comparisons, or to give the wrong answer for at least one input sequence.

Exercises

1. Simon bar Kokhba thinks of an integer between 1 and 1,000,000 (or so he claims). You are trying to determine his number by asking as few yes/no questions as possible. How many yes/no questions are required to determine Simon's number in the worst case? Give both an upper bound (supported by an algorithm) and a lower bound.

2. Consider the following *multi-dictionary* problem. Let $A[1..n]$ be a fixed array of distinct integers. Given an array $X[1..k]$, we want to find the position (if any) of each integer $X[i]$

[3] but not all—see Exercise 3

[4] In fact, the $n-1$ lower bound for finding the maximum holds in a much powerful model called *algebraic* decision trees, which are binary trees where every query is a comparison between two polynomial functions of the input values, such as 'Is $x_1^2 - 3x_2x_3 + x_4^{17}$ bigger or smaller than $5 + x_1x_3^5x_5^2 - 2x_7^{42}$?'

in the array A. In other words, we want to compute an array $I[1\,..\,k]$ where for each i, either $I[i] = 0$ (so zero means 'none') or $A[I[i]] = X[i]$. Determine the *exact* complexity of this problem, as a function of n and k, in the binary decision tree model.

*3. Suppose you want to determine the largest number in an n-element set $X = \{x_1, x_2, \dots, x_n\}$, where each element x_i is an integer between 1 and $2^m - 1$. Describe an algorithm that solves this problem in $O(n + m)$ steps, where at each step, your algorithm compares one of the elements x_i with a *constant*. In particular, your algorithm must never actually compare two elements of X! *[Hint: Construct and maintain a nested set of 'pinning intervals' for the numbers that you have not yet removed from consideration, where each interval but the largest is either the upper half or lower half of the next larger block.]*

An adversary means opposition and competition,
but not having an adversary means grief and loneliness.
— Zhuangzi (Chuang-tsu) c. 300 BC

It is possible that the operator could be hit by an asteroid and your $20 could fall off his cardboard box and land on the ground, and while you were picking it up, $5 could blow into your hand. You therefore could win $5 by a simple twist of fate.
— Penn Jillette, explaining how to win at Three-Card Monte (1999)

20 Adversary Arguments

20.1 Three-Card Monte

Until Times Square was sanitized into TimesSquareLand$^{\text{TM}}$ by Mayor Rudy Guiliani, you could often find dealers stealing tourists' money using a game called 'Three Card Monte' or 'Spot the Lady'. The dealer has three cards, say the Queen of Hearts and the two and three of clubs. The dealer shuffles the cards face down on a table (usually slowly enough that you can follow the Queen), and then asks the tourist to bet on which card is the Queen. In principle, the tourist's odds of winning are at least one in three.

In practice, however, the tourist *never*[1] wins, because the dealer cheats. There are actually *four* cards; before he even starts shuffling the cards, the dealer palms the queen or sticks it up his sleeve. No matter what card the tourist bets on, the dealer turns over a black card. If the tourist gives up, the dealer slides the queen under one of the cards and turns it over, showing the tourist 'where the queen was all along'. If the dealer is really good, the tourist won't see the dealer changing the cards and will think maybe the queen *was* there all along and he just wasn't smart enough to figure that out. As long as the dealer doesn't reveal all the black cards at once, the tourist has no way to prove that the dealer cheated![2]

20.2 n-Card Monte

Now let's consider a similar game, but with an algorithm acting as the tourist and with bits instead of cards. Suppose we have an array of n bits and we want to determine if any of them is a 1. Obviously we can figure this out by just looking at every bit, but can we do better? Is there maybe some complicated tricky algorithm to answer the question "Any ones?" without looking at every bit? Well, of course not, but how do we prove it?

The simplest proof technique is called an *adversary* argument. The idea is that an all-powerful malicious adversary (the dealer) *pretends* to choose an input for the algorithm (the tourist). When the algorithm wants looks at a bit (a card), the adversary sets that bit to whatever value will make the algorithm do the most work. If the algorithm does not look at enough bits before terminating, then there will be several different inputs, each consistent with the bits already seen, the should result in different outputs. Whatever the algorithm outputs, the adversary can 'reveal' an input that is has all the examined bits but contradicts the algorithm's output, and then claim that that was the input that he was using all along. Since the only information the algorithm has is the set of bits it examined, the algorithm cannot distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully.

[1] What, never? No, **NEVER**. Anyone you see winning at Three Card Monte is a shill.

[2] Even if the dealer isn't very good, he cheats anyway. The shills will protect him from any angry tourists who realize they've been ripped off, and shake down any tourist who refuses to pay. You *cannot* win this game.

For the n-card monte problem, the adversary originally pretends that the input array is all zeros—whenever the algorithm looks at a bit, it sees a 0. Now suppose the algorithms stops before looking at all three bits. If the algorithm says 'No, there's no 1,' the adversary changes one of the unexamined bits to a 1 and shows the algorithm that it's wrong. If the algorithm says 'Yes, there's a 1,' the adversary reveals the array of zeros and again proves the algorithm wrong. Either way, the algorithm cannot tell that the adversary has cheated.

One absolutely crucial feature of this argument is that *the adversary makes absolutely* ***no*** *assumptions about the algorithm*. The adversary strategy can't depend on some predetermined order of examining bits, and it doesn't care about anything the algorithm might or might not do when it's not looking at bits. *Any* algorithm that doesn't examine every bit falls victim to the adversary.

20.3 Finding Patterns in Bit Strings

Let's make the problem a little more complicated. Suppose we're given an array of n bits and we want to know if it contains the substring 01, a zero followed immediately by a one. Can we answer this question without looking at every bit?

It turns out that if n is odd, we *don't* have to look at all the bits. First we look the bits in every even position: $B[2], B[4], \ldots, B[n-1]$. If we see $B[i] = 0$ and $B[j] = 1$ for any $i < j$, then we know the pattern 01 is in there somewhere—starting at the last 0 before $B[j]$—so we can stop without looking at any more bits. If we see only 1s followed by 0s, we don't have to look at the bit between the last 0 and the first 1. If every even bit is a 0, we don't have to look at $B[1]$, and if every even bit is a 1, we don't have to look at $B[n]$. In the worst case, our algorithm looks at only $n-1$ of the n bits.

But what if n is even? In that case, we can use the following adversary strategy to show that any algorithm *does* have to look at every bit. The adversary will attempt to produce an 'input' string B *without* the substring 01; all such strings have the form $11\ldots100\ldots0$. The adversary maintains two indices ℓ and r and pretends that the prefix $B[1\,..\,\ell]$ contains only 1s and the suffix $B[r\,..\,n]$ contains only 0s. Initially $\ell = 0$ and $r = n+1$.

$$\mathbf{11111}\underset{\substack{\uparrow\\ \boldsymbol{\ell}}}{\mathbf{1}}\square\square\square\square\square\square\underset{\substack{\uparrow\\ \boldsymbol{r}}}{\mathbf{0}}\mathbf{000}$$

What the adversary is thinking; □ represents an unknown bit.

The adversary maintains the invariant that $r - \ell$, the length of the undecided portion of the 'input' string, is even. When the algorithm looks at a bit between ℓ and r, the adversary chooses whichever value preserves the parity of the intermediate chunk of the array, and then moves either ℓ or r. Specifically, here's what the adversary does when the algorithm examines bit $B[i]$. (Note that I'm specifying the adversary strategy as an algorithm!)

```
HIDE01($i$):
  if $i \le \ell$
      $B[i] \leftarrow 1$
  else if $i \ge r$
      $B[i] \leftarrow 0$
  else if $i - \ell$ is even
      $B[i] \leftarrow 0$
      $r \leftarrow i$
  else
      $B[i] \leftarrow 1$
      $\ell \leftarrow i$
```

It's fairly easy to prove that this strategy forces the algorithm to examine every bit. If the algorithm doesn't look at every bit to the right of r, the adversary could replace some unexamined bit with a 1. Similarly, if the algorithm doesn't look at every bit to the left of ℓ, the adversary could replace some unexamined bit with a zero. Finally, if there are any unexamined bits between ℓ and r, there must be at least two such bits (since $r - \ell$ is always even) and the adversary can put a 01 in the gap.

In general, we say that a bit pattern is *evasive* if we have to look at every bit to decide if a string of n bits contains the pattern. So the pattern 1 is evasive for all n, and the pattern 01 is evasive if and only if n is even. It turns out that the *only* patterns that are evasive for *all* values of n are the one-bit patterns 0 and 1.

20.4 Evasive Graph Properties

Another class of problems for which adversary arguments give good lower bounds is graph problems where the graph is represented by an adjacency matrix, rather than an adjacency list. Recall that the adjacency matrix of an undirected n-vertex graph $G = (V, E)$ is an $n \times n$ matrix A, where $A[i, j] = \big[(i, j) \in E\big]$. We are interested in deciding whether an undirected graph has or does not have a certain *property*. For example, is the input graph connected? Acyclic? Planar? Complete? A tree? We call a graph property *evasive* if we have to look look at all $\binom{n}{2}$ entries in the adjacency matrix to decide whether a graph has that property.

An obvious example of an evasive graph property is *emptiness*: Does the graph have any edges at all? We can show that emptiness is evasive using the following simple adversary strategy. The adversary maintains *two* graphs E and G. E is just the empty graph with n vertices. Initially G is the complete graph on n vertices. Whenever the algorithm asks about an edge, the adversary removes that edge from G (unless it's already gone) and answers 'no'. If the algorithm terminates without examining every edge, then G is not empty. Since both G and E are consistent with all the adversary's answers, the algorithm must give the wrong answer for one of the two graphs.

20.5 Connectedness Is Evasive

Now let me give a more complicated example, *connectedness*. Once again, the adversary maintains two graphs, Y and M ('yes' and 'maybe'). Y contains all the edges that the algorithm knows are definitely in the input graph. M contains all the edges that the algorithm thinks *might* be in the input graph, or in other words, all the edges of Y plus all the unexamined edges. Initially, Y is empty and M is complete.

Here's the strategy that adversary follows when the adversary asks whether the input graph contains the edge e. I'll assume that whenever an algorithm examines an edge, it's in M but not in Y; in other words, algorithms never ask about the same edge more than once.

```
HideConnectedness($e$):
  if $M \setminus \{e\}$ is connected
      remove $(i, j)$ from $M$
      return $0$
  else
      add $e$ to $Y$
      return $1$
```

Notice that the graphs Y and M are both consistent with the adversary's answers at all times. The adversary strategy maintains a few other simple invariants.

- ***Y* is a subgraph of *M*.** This is obvious.
- ***M* is connected.** This is also obvious.
- **If *M* has a cycle, none of its edges are in *Y*.** If M has a cycle, then deleting any edge in that cycle leaves M connected.
- ***Y* is acyclic.** This follows directly from the previous invariant.
- **If $Y \neq M$, then *Y* is disconnected.** The only connected acyclic graph is a tree. Suppose Y is a tree and some edge e is in M but not in Y. Then there is a cycle in M that contains e, all of whose other edges are in Y. This violated our third invariant.

We can also think about the adversary strategy in terms of minimum spanning trees. Recall the anti-Kruskal algorithm for computing the *maximum* spanning tree of a graph: Consider the edges one at a time in increasing order of length. If removing an edge would disconnect the graph, declare it part of the spanning tree (by adding it to Y); otherwise, throw it away (by removing it from M). If the algorithm examines all $\binom{n}{2}$ possible edges, then Y and M are both equal to the maximum spanning tree of the complete n-vertex graph, where the weight of an edge is the time when the algorithm asked about it.

Now, if an algorithm terminates before examining all $\binom{n}{2}$ edges, then there is at least one edge in M that is not in Y. Since the algorithm cannot distinguish between M and Y, even though M is connected and Y is not, the algorithm cannot possibly give the correct output for both graphs. Thus, in order to be correct, any algorithm must examine every edge—*Connectedness is evasive!*

20.6 An Evasive Conjecture

A graph property is *nontrivial* is there is at least one graph with the property and at least one graph without the property. (The only trivial properties are 'Yes' and 'No'.) A graph property is *monotone* if it is closed under taking subgraphs — if G has the property, then any subgraph of G has the property. For example, emptiness, planarity, acyclicity, and *non*-connectedness are monotone. The properties of being a tree and of having a vertex of degree 3 are not monotone.

Conjecture 1 (Aanderraa, Karp, and Rosenberg). *Every nontrivial monotone property of n-vertex graphs is evasive.*

The Aanderraa-Karp-Rosenberg conjecture has been proven when $n = p^e$ for some prime p and positive integer exponent e—the proof uses some interesting results from algebraic topology[3]—but it is still open for other values of n.'

There are non-trivial non-evasive graph properties, but all known examples are non-monotone. One such property—'scorpionhood'—is described in an exercise at the end of this lecture note.

[3] Let Δ be a contractible simplicial complex whose automorphism group $\operatorname{Aut}(\Delta)$ is vertex-transitive, and let Γ be a vertex-transitive subgroup of $\operatorname{Aut}(\Delta)$. If there are normal subgroups $\Gamma_1 \lhd \Gamma_2 \lhd \Gamma$ such that $|\Gamma_1| = p^\alpha$ for some prime p and integer α, $|\Gamma/\Gamma_2| = q^\beta$ for some prime q and integer β, and Γ_2/Γ_1 is cyclic, then Δ is a simplex.

No, this will not be on the final exam.

20.7 Finding the Minimum and Maximum

Last time, we saw an adversary argument that finding the largest element of an unsorted set of n numbers requires at least $n-1$ comparisons. Let's consider the complexity of finding the largest *and* smallest elements. More formally:

> Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find indices i and j such that $x_i = \min X$ and $x_j = \max X$.

How many comparisons do we need to solve this problem? An upper bound of $2n-3$ is easy: find the minimum in $n-1$ comparisons, and then find the maximum of everything else in $n-2$ comparisons. Similarly, a lower bound of $n-1$ is easy, since any algorithm that finds the min and the max certainly finds the max.

We can improve both the upper and the lower bound to $\lceil 3n/2 \rceil - 2$. The upper bound is established by the following algorithm. Compare all $\lfloor n/2 \rfloor$ consecutive pairs of elements x_{2i-1} and x_{2i}, and put the smaller element into a set S and the larger element into a set L. if n is odd, put x_n into both L and S. Then find the smallest element of S and the largest element of L. The total number of comparisons is at most

$$\underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{\text{build } S \text{ and } L} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \min S} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \max L} = \left\lceil \frac{3n}{2} \right\rceil - 2.$$

For the lower bound, we use an adversary argument. The adversary marks each element $+$ if it *might* be the maximum element, and $-$ if it *might* be the minimum element. Initially, the adversary puts both marks $+$ and $-$ on every element. If the algorithm compares two double-marked elements, then the adversary declares one smaller, removes the $+$ mark from the smaller element, and removes the $-$ mark from the larger one. In every other case, the adversary can answer so that at most one mark needs to be removed. For example, if the algorithm compares a double marked element against one labeled $-$, the adversary says the one labeled $-$ is smaller and removes the $-$ mark from the other. If the algorithm compares to $+$'s, the adversary must unmark one of the two.

Initially, there are $2n$ marks. At the end, in order to be correct, exactly one item must be marked $+$ and exactly one other must be marked $-$, since the adversary can make any $+$ the maximum and any $-$ the minimum. Thus, the algorithm must force the adversary to remove $2n-2$ marks. At most $\lfloor n/2 \rfloor$ comparisons remove two marks; every other comparison removes at most one mark. Thus, the adversary strategy forces any algorithm to perform at least $2n-2-\lfloor n/2 \rfloor = \lceil 3n/2 \rceil - 2$ comparisons.

20.8 Finding the Median

Finally, let's consider the *median* problem: Given an unsorted array X of n numbers, find its $n/2$th largest entry. (I'll assume that n is even to eliminate pesky floors and ceilings.) More formally:

> Given a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the index m such that x_m is the $n/2$th largest element in the sequence.

To prove a lower bound for this problem, we can use a combination of information theory and two adversary arguments. We use one adversary argument to prove the following simple lemma:

Lemma 1. *Any comparison tree that correctly finds the median element also identifies the elements smaller than the median and larger than the median.*

Proof: Suppose we reach a leaf of a decision tree that chooses the median element x_m, and there is still some element x_i that isn't known to be larger or smaller than x_m. In other words, we cannot decide based on the comparisons that we've already performed whether $x_i < x_m$ or $x_i > x_m$. Then in particular no element is known to lie between x_i and x_m. This means that there must be an input that is consistent with the comparisons we've performed, in which x_i and x_m are adjacent in sorted order. But then we can swap x_i and x_m, without changing the result of any comparison, and obtain a different consistent input in which x_i is the median, not x_m. Our decision tree gives the wrong answer for this 'swapped' input. □

This lemma lets us rephrase the median-finding problem yet again.

> Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the indices of its $n/2 - 1$ largest elements L and its $n/2$th largest element x_m.

Now suppose a 'little birdie' tells us the set L of elements larger than the median. This information fixes the outcomes of certain comparisons—any item in L is bigger than any element not in L—so we can 'prune' those comparisons from the comparison tree. The pruned tree finds the largest element of $X \setminus L$ (the median of X), and thus must have depth at least $n/2 - 1$. In fact, the adversary argument in the last lecture implies that *every* leaf in the pruned tree must have depth at least $n/2 - 1$, so the pruned tree has at least $2^{n/2-1}$ leaves.

There are $\binom{n}{n/2-1} \approx 2^n/\sqrt{n/2}$ possible choices for the set L. Every leaf in the original comparison tree is also a leaf in exactly one of the $\binom{n}{n/2-1}$ pruned trees, so the original comparison tree must have at least $\binom{n}{n/2-1}2^{n/2-1} \approx 2^{3n/2}/\sqrt{n/2}$ leaves. Thus, any comparison tree that finds the median must have depth at least

$$\left\lceil \frac{n}{2} - 1 + \lg \binom{n}{n/2-1} \right\rceil = \frac{3n}{2} - O(\log n).$$

A more complicated adversary argument (also involving pruning the comparison tree with little birdies) improves this lower bound to $2n - o(n)$.

A similar argument implies that at least $n - k + \left\lceil \lg \binom{n}{k-1} \right\rceil = \Omega((n-k) + k\log(n/k))$ comparisons are required to find the kth largest element in an n-element set. This bound is tight up to constant factors for all $k \le n/2$; there is an algorithm that uses at most $O(n + k\log(n/k))$ comparisons. Moreover, this lower bound is *exactly* tight when $k = 1$ or $k = 2$. In fact, these are the *only* values of $k \le n/2$ for which the exact complexity of the selection problem is known; even the case $k = 3$ is still open.

Exercises

1. (a) Let X be a set containing an odd number of n-bit strings. Prove that any algorithm that decides whether a given n-bit string is an element of X *must* examine every bit of the input string in the worst case.

   (b) Give a one-line proof that the bit pattern 01 is evasive for all even n.

   (c) Prove that the bit pattern 11 is evasive if and only if $n \bmod 3 = 1$.

   *(d) Prove that the bit pattern 111 is evasive if and only if $n \bmod 4 = 0$ or 3.

2. Suppose we are given the adjacency matrix of a *directed* graph G with n vertices. Describe an algorithm that determines whether G has a *sink* by probing only $O(n)$ bits in the input matrix. A sink is a vertex that has an incoming edge from every other vertex, but no outgoing edges.

*3. A *scorpion* is an undirected graph with three special vertices: the *sting*, the *tail*, and the *body*. The sting is connected only to the tail; the tail is connected only to the sting and the body; and the body is connected to every vertex except the sting. The rest of the vertices (the head, eyes, legs, antennae, teeth, gills, flippers, wheels, etc.) can be connected arbitrarily. Describe an algorithm that determines whether a given n-vertex graph is a scorpion by probing only $O(n)$ entries in the adjacency matrix.

4. Prove using an advesary argument that acyclicity is an evasive graph property. *[Hint: Kruskal.]*

5. Prove that finding the second largest element in an n-element array requires *exactly* $n - 2 + \lceil \lg n \rceil$ comparisons in the worst case. Prove the upper bound by describing and analyzing an algorithm; prove the lower bound using an adversary argument.

6. Let T be a perfect ternary tree where every leaf has depth ℓ. Suppose each of the 3^ℓ leaves of T is labeled with a bit, either 0 or 1, and each internal node is labeled with a bit that agrees with the *majority* of its children.

   (a) Prove that any deterministic algorithm that determines the label of the root must examine all 3^ℓ leaf bits in the worst case.

   (b) Describe and analyze a *randomized* algorithm that determines the root label, such that the expected number of leaves examined is $o(3^\ell)$. (You may want to review the notes on randomized algorithms.)

*7. UIUC has just finished constructing the new Reingold Building, the tallest structure on campus. In order to determine how much insurance to buy, the administration needs to determine the highest safe floor in the building. A floor is consdered *safe* if a student can fall from a window on that floor and survive; if the student dies, the floor is considered *unsafe*. The same floors are safe for every student, and any floor that is higher than an unsafe floor is also unsafe. The only way to determine whether a floor is safe is for a student 'volunteer' to jump out of a window on that floor. The University wants to minimize the number of jumps; however, there are only a handful of student 'volunteers' available.

   Let $J(n, s)$ denote the minimum number of jumps required to determine the highest safe floor in an n-story building, using at most s student volunteers. Derive the tightest upper and lower bounds you can for this function. *[Hint: It may help to consider one or both inverse functions $N(j, s)$ and $S(n, j)$. How should these be defined?]*

Reduce big troubles to small ones, and small ones to nothing.
— Chinese proverb

I have yet to see any problem, however complicated, which, when you looked at it in the right way, did not become still more complicated.
— Poul Anderson, *New Scientist* (September 25, 1969)

H Reductions

H.1 Introduction

An extremely common technique for deriving algorithms is *reduction*—instead of solving a problem directly, we use an algorithm for some other related problem as a subroutine or black box.

For example, when we talked about nuts and bolts, we argued that once the nuts and bolts are sorted, we can match each nut to its bolt in linear time. Thus, since we can sort nuts and bolts in $O(n \log n)$ expected time, then we can also match them in $O(n \log n)$ expected time:

$$T_{\text{match}}(n) \leq T_{\text{sort}}(n) + O(n) = O(n \log n) + O(n) = O(n \log n).$$

Let's consider (as we did in the previous lecture) a decision tree model of computation, where every query is a comparison between a nut and a bolt—too big, too small, or just right? The output to the matching problem is a permutation π, where for all i, the ith nut matches the $\pi(i)$th bolt. Since there are $n!$ permutations of n items, any nut/bolt comparison tree that matches n nuts and bolts has at least $n!$ leaves, and thus has depth at least $\lceil \log_3(n!) \rceil = \Omega(n \log n)$.

Now the same reduction from matching to sorting can be used to prove a lower bound for *sorting* nuts and bolts, just by reversing the inequality:

$$T_{\text{sort}}(n) \geq T_{\text{match}}(n) - O(n) = \Omega(n \log n) - O(n) = \Omega(n \log n).$$

Thus, any nut-bolt comparison tree that sorts n nuts and bolts has depth $\Omega(n \log n)$, and our randomized quicksort algorithm is optimal.[1]

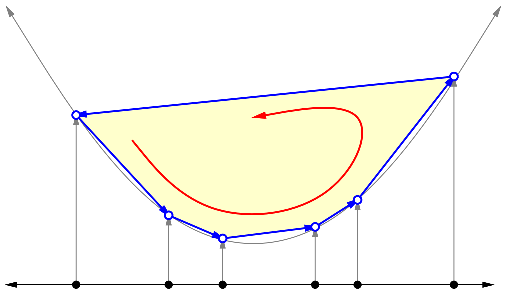
The rest of this lecture assumes some familiarity with computational geometry.

H.2 Sorting to Convex Hulls

Here's a slightly less trivial example. Suppose we want to prove a lower bound for the problem of computing the convex hull of a set of n points in the plane. To do this, we demonstrate a reduction from sorting to convex hulls.

To sort a list of n numbers $\{a, b, c, \ldots\}$, we first transform it into a set of n points $\{(a, a^2), (b, b^2), (c, c^2), \ldots\}$. You can think of the original numbers as a set of points on a horizontal real number line, and the transformation as lifting those point up to the parabola $y = x^2$. Then we compute the convex hull of the parabola points. Finally, to get the final sorted list of numbers, we output the first coordinate of every convex vertex, starting from the leftmost vertex and going in counterclockwise order.

[1] We could have proved this lower bound directly. The output to the sorting problem is *two* permutations, so there are $n!^2$ possible outputs, and we get a lower bound of $\lceil \log_3(n!^2) \rceil = \Omega(n \log n)$.



Reducing sorting to computing a convex hull.

Transforming the numbers into points takes $O(n)$ time. Since the convex hull is output as a circular doubly-linked list of vertices, reading off the final sorted list of numbers also takes $O(n)$ time. Thus, given a black-box convex hull algorithm, we can sort in linear extra time. In this case, we say that *there is a linear time reduction from sorting to convex hulls*. We can visualize the reduction as follows:

$$\begin{tikzcd}
\boxed{\text{set of } n \text{ numbers}} \arrow[r, "O(n)"] & \boxed{\text{set of } n \text{ points}} \arrow[d, Rightarrow, "\textsc{ConvexHull}"] \\
\boxed{\text{sorted list of numbers}} & \boxed{\text{convex polygon}} \arrow[l, "O(n)"']
\end{tikzcd}$$

(I *strongly* encourage you to draw a picture like this whenever you use a reduction argument, at least until you get used to them.) The reduction gives us the following inequality relating the complexities of the two problems:

$$T_{\text{sort}}(n) \leq T_{\text{convex hull}}(n) + O(n)$$

Since we can compute convex hulls in $O(n \log n)$ time, our reduction implies that we can also sort in $O(n \log n)$ time. More importantly, by reversing the inequality, we get a lower bound on the complexity of computing convex hulls.

$$T_{\text{convex hull}}(n) \geq T_{\text{sort}}(n) - O(n)$$

Since any binary decision tree requires $\Omega(n \log n)$ time to sort n numbers, it follows that any binary decision tree requires $\Omega(n \log n)$ time to compute the convex hull of n points.

H.3 Watch the Model!

This result about the complexity of computing convex hulls is often misquoted as follows:

> *Since we need $\Omega(n \log n)$ comparisons to sort, we also need $\Omega(n \log n)$ comparisons (between x-coordinates) to compute convex hulls.*

Although this statement is true, **it's completely trivial**, since it's impossible to compute convex hulls using *any* number of comparisons! In order to compute hulls, we *must* perform counterclockwise tests on triples of points.

The convex hull algorithms we've seen — Graham's scan, Jarvis's march, divide-and-conquer, Chan's shatter — can all be modeled as binary[2] decision trees, where every query is a counterclockwise test on three points. So our binary decision tree lower bound is meaningful, and several of those algorithms are optimal.

[2] or ternary, if we allow colinear triples of points

This is a subtle but important point about deriving lower bounds using reduction arguments. In order for any lower bound to be meaningful, it must hold in a model in which the problem can be solved! Often the problem we are reducing *from* is much simpler than the problem we are reducing *to*, and thus can be solved in a more restrictive model of computation.

H.4 Element Uniqueness (A Bad Example)

The *element uniqueness* problem asks, given a list of n numbers $x_1, x_2, \ldots, x_n$, whether any two of them are equal. There is an obvious and simple algorithm to solve this problem: sort the numbers, and then scan for adjacent duplicates. Since we can sort in $O(n \log n)$ time, we can solve the element uniqueness problem in $O(n \log n)$ time.

We also have an $\Omega(n \log n)$ lower bound for sorting, but our reduction does *not* give us a lower bound for element uniqueness. The reduction goes the wrong way! Inscribe the following on the back of your hand[3]:

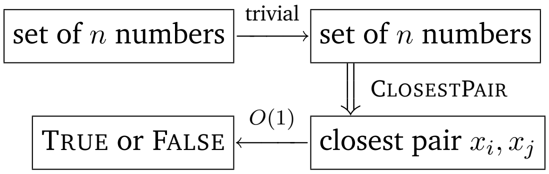
To prove that problem A is harder than problem B, reduce B to A.

There isn't (as far as I know) a reduction from sorting to the element uniqueness problem. However, using other techniques (which I won't talk about), it is possible to prove an $\Omega(n \log n)$ lower bound for the element uniqueness problem. The lower bound applies to so-called *algebraic decision trees*. An algebraic decision tree is a ternary decision tree, where each query asks for the sign of a constant-degree polynomial in the variables $x_1, x_2, \ldots, x_n$. A comparison tree is an example of an algebraic decision tree, using polynomials of the form $x_i - x_j$. The reduction from sorting to element uniqueness implies that any algebraic decision tree requires $\Omega(n \log n)$ time to sort n numbers. But since algebraic decision trees are ternary decision trees, we already knew that.

H.5 Closest Pair

The simplest version of the *closest pair* problem asks, given a list of n numbers $x_1, x_2, \ldots, x_n$, to find the closest pair of elements, that is, the elements x_i and x_j that minimize $|x_i - x_j|$.

There is an obvious reduction from element uniqueness to closest pair, based on the observation that the elements of the input list are distinct if and only if the distance between the closest pair is bigger than zero. This reduction implies that the closest pair problem requires $\Omega(n \log n)$ time in the algebraic decision tree model.



There are also higher-dimensional closest pair problems; for example, given a set of n points in the plane, find the two points that closest together. Since the one-dimensional problem is a special case of the 2d problem — just put all n point son the x-axis — the $\Omega(n \log n)$ lower bound applies to the higher-dimensional problems as well.

[3] right under all those rules about logarithms, geometric series, and recurrences

H.6 3SUM to Colinearity...

Unfortunately, lower bounds are relatively few and far between. There are thousands of computational problems for which we cannot prove any good lower bounds. We can still learn something useful about the complexity of such a problem by from reductions, namely, that it is harder than some other problem.

Here's an example. The problem 3SUM asks, given a sequence of n numbers $x_1, \ldots, x_n$, whether any three of them sum to zero. There is a fairly simple algorithm to solve this problem in $O(n^2)$ time **(hint, hint)**. This is widely believed to be the fastest algorithm possible. There is an $\Omega(n^2)$ lower bound for 3SUM, but only in a fairly weak model of computation.[4]

Now consider a second problem: given a set of n points in the plane, do any three of them lie on a common non-horizontal line? Again, there is an $O(n^2)$-time algorithm, and again, this is believed to be the best possible. The following reduction from 3SUM offers some support for this belief. Suppose we are given an array A of n numbers as input to 3SUM. Replace each element $a \in A$ with three points $(a, 0)$, $(-a/2, 1)$, and $(a, 2)$. Thus, we replace the n numbers with $3n$ points on three horizontal lines $y = 0$, $y = 1$, and $y = 2$.

If any three points in this set lie on a common non-horizontal line, they consist of one point on each of those three lines, say $(a, 0)$, $(-b/2, 1)$, and $(c, 2)$. The slope of the common line is equal to both $-b/2 - a$ and $c + b/2$; since these two expressions are equal, we must have $a + b + c = 0$. Similarly, is any three elements $a, b, c \in A$ sum to zero, then the resulting points $(a, 0)$, $(-b/2, 1)$, and $(c, 2)$ are colinear.

So we have a valid reduction from 3SUM to the colinear-points problem:

$$\begin{tikzcd} \boxed{\text{set of } n \text{ numbers}} \arrow[r, "O(n)"] & \boxed{\text{set of } 3n \text{ points}} \arrow[d, Rightarrow, "\textsc{Colinear?}"] \\ \boxed{\textsc{True or False}} & \boxed{\textsc{True or False}} \arrow[l, "\text{trivial}"'] \end{tikzcd}$$

$$T_{\text{3SUM}}(n) \leq T_{\text{colinear}}(3n) + O(n) \qquad \Longrightarrow \qquad T_{\text{colinear}}(n) \geq T_{\text{3SUM}}(n/3) - O(n).$$

Thus, if we could detect colinear points in $o(n^2)$ time, we could also solve 3SUM in $o(n^2)$ time, which seems unlikely. Conversely, if we could prove an $\Omega(n^2)$ lower bound for 3SUM in a sufficiently powerful model of computation, it would imply an $\Omega(n^2)$ lower bound for the colinear points problem as well.

The existing $\Omega(n^2)$ lower bound for 3SUM does *not* imply a lower bound for finding colinear points, because the model of computation is too weak. It is possible to prove an $\Omega(n^2)$ lower bound directly using an adversary argument, but only in a fairly weak decision-tree model of computation.

Note that in order to prove that the reduction is correct, we have to show that both yes answers and no answers are correct: the numbers sum to zero *if and only if* three points lie on a line. **Even though the reduction itself only goes one way, from the 'easier' problem to the 'harder' problem, the proof of correctness must go both ways.**

Anka Gajentaan and Mark Overmars[5] defined a whole class of computational geometry problems that are harder than 3SUM; they called these problems 3SUM-*hard*. A sub-quadratic algorithm for any 3SUM-hard problem would imply a subquadratic algorithm for 3SUM. I'll finish the lecture with two more examples of 3SUM-hard problems.

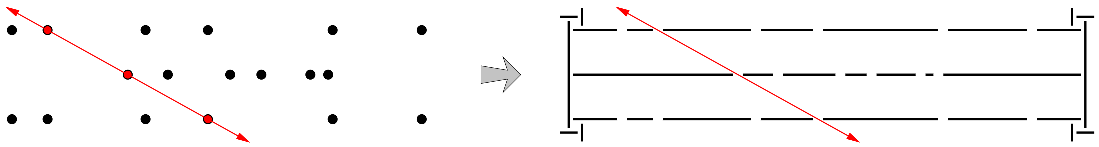
[4] The $\Omega(n^2)$ lower bound holds in a decision tree model where every query asks for the sign of a linear combination of three of the input numbers. For example, 'Is $5x_1 + x_{42} - 17x_5$ positive, negative, or zero?' See my paper 'Lower bounds for linear satisfiability problems' (http://www.uiuc.edu/~jeffe/pubs/linsat.html) for the gory(!) details.

[5] A. Gajentaan and M. Overmars, On a class of $O(n^2)$ problems in computational geometry, *Comput. Geom. Theory Appl.* 5:165–185, 1995. ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1993/1993-15.ps.gz

H.7 ...to Segment Splitting ...

Consider the following *segment splitting problem*: Given a collection of line segments in the plane, is there a line that does not hit any segment and splits the segments into two non-empty subsets?

To show that this problem is 3SUM-hard, we start with the collection of points produced by our last reduction. Replace each point by a 'hole' between two horizontal line segments. To make sure that the only way to split the segments is by passing through three colinear holes, we build two 'gadgets', each consisting of five segments, to cap off the left and right ends as shown in the figure below.



Top: $3n$ points, three on a non-horizontal line.
Bottom: $3n + 13$ segments separated by a line through three colinear holes.

This reduction could be performed in linear time if we could make the holes infinitely small, but computers can't really deal with infinitesimal numbers. On the other hand, if we make the holes too big, we might be able to thread a line through three holes that don't quite line up. I won't go into details, but it is possible to compute a working hole size in $O(n \log n)$ time by first computing the distance between the closest pair of points.

Thus, we have a valid reduction from 3SUM to segment splitting (by way of colinearity):

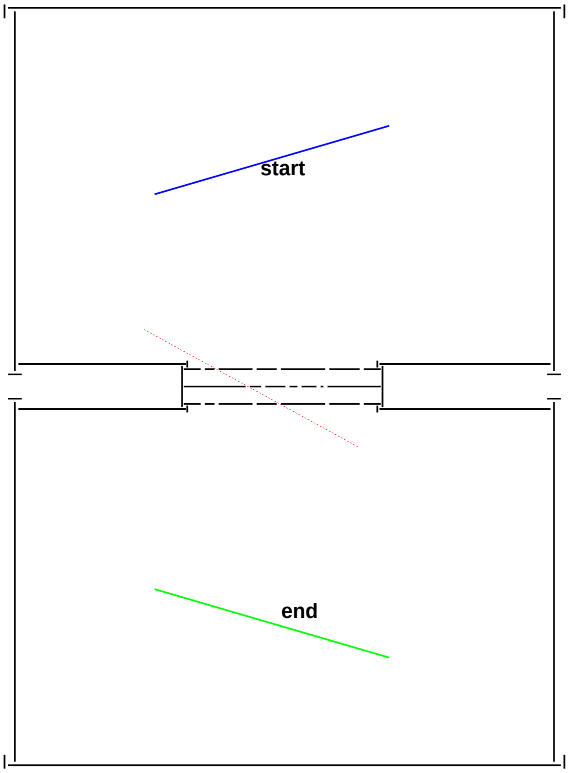
$$\begin{tikzcd}
\boxed{\text{set of } n \text{ numbers}} \arrow[r, "O(n)"] & \boxed{\text{set of } 3n \text{ points}} \arrow[r, dashed, "O(n \log n)"] & \boxed{\text{set of } 3n+13 \text{ segments}} \arrow[d, Rightarrow, "\textsc{Splittable?}"] \\
\boxed{\textsc{True} \text{ or } \textsc{False}} & \boxed{\textsc{True} \text{ or } \textsc{False}} \arrow[l, "\text{trivial}"'] & \boxed{\textsc{True} \text{ or } \textsc{False}} \arrow[l, dashed, "\text{trivial}"']
\end{tikzcd}$$

$$T_{3\text{SUM}}(n) \le T_{\text{split}}(3n+13) + O(n \log n) \implies T_{\text{split}}(n) \ge T_{3\text{SUM}}\left(\frac{n-13}{3}\right) - O(n \log n).$$

H.8 ...to Motion Planning

Finally, suppose we want to know whether a robot can move from one position and location to another. To make things simple, we'll assume that the robot is just a line segment, and the environment in which the robot moves is also made up of non-intersecting line segments. Given an initial position and orientation and a final position and orientation, is there a sequence of translations and rotations that moves the robot from start to finish?

To show that this *motion planning* problem is 3SUM-hard, we do one more reduction, starting from the set of segments output by the previous reduction algorithm. Specifically, we use our earlier set of line segments as a 'screen' between two large rooms. The rooms are constructed so that the robot can enter or leave each room only by passing through the screen. We make the robot long enough that the robot can pass from one room to the other if and only if it can pass through three colinear holes in the screen. (If the robot isn't long enough, it could get between the 'layers' of the screen.) See the figure below:



The robot can move from one room to the other if and only if the screen between the rooms has three colinear holes.

Once we have the screen segments, we only need linear time to compute how big the rooms should be, and then $O(1)$ time to set up the 20 segments that make up the walls. So we have a fast reduction from 3SUM to motion planning (by way of colinearity and segment splitting):

$$\begin{tikzcd}
\boxed{\text{set of } n \text{ numbers}} \arrow[r, "O(n)"] & \boxed{\text{set of } 3n \text{ points}} \arrow[r, "O(n\log n)"] & \boxed{\text{set of } 3n+13 \text{ segments}} \arrow[r, "O(n)"] & \boxed{\text{set of } 3n+33 \text{ segments}} \arrow[d, Rightarrow, "\textsc{Movable?}"] \\
\boxed{\textsc{True or False}} & \boxed{\textsc{True or False}} \arrow[l, "\text{trivial}"'] & \boxed{\textsc{True or False}} \arrow[l, "\text{trivial}"'] & \boxed{\textsc{True or False}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

$$T_{\text{3SUM}}(n) \le T_{\text{motion}}(3n+33) + O(n\log n) \quad\Longrightarrow\quad T_{\text{mtion}}(n) \ge T_{\text{3SUM}}\left(\frac{n-33}{3}\right) - O(n\log n).$$

Thus, a sufficiently powerful $\Omega(n^2)$ lower bound for 3SUM would imply an $\Omega(n^2)$ lower bound for motion planning as well. The existing $\Omega(n^2)$ lower bound for 3SUM does *not* imply a lower bound for this problem — the model of computation in which the lower bound holds is too weak to even solve the motion planning problem. In fact, the best lower bound anyone can prove for this motion planning problem is $\Omega(n \log n)$, using a (different) reduction from element uniqueness. But the reduction does give us *evidence* that motion planning 'should' require quadratic time.

Math class is tough!
— Teen Talk Barbie (1992)

That's why I like it!
— What she should have said next

The wonderful thing about standards is that
there are so many of them to choose from.
— Grace Hopper

If a problem has no solution, it may not be a problem, but a fact —
not to be solved, but to be coped with over time.
— Shimon Peres

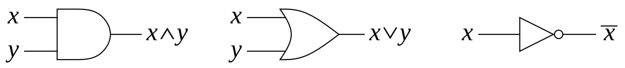
21 NP-Hard Problems

21.1 'Efficient' Problems

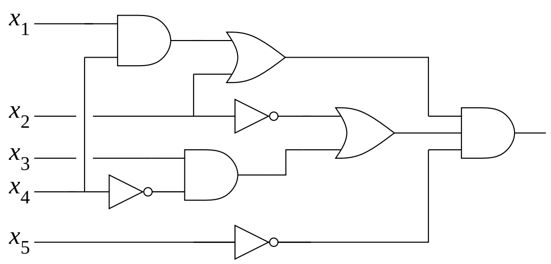
A long time ago[1], theoretical computer scientists like Steve Cook and Dick Karp decided that a minimum requirement of any efficient algorithm is that it runs in polynomial time: $O(n^c)$ for some constant c. People recognized early on that not all problems can be solved this quickly, but we had a hard time figuring out exactly which ones could and which ones couldn't. So Cook, Karp, and others, defined the class of *NP-hard* problems, which most people believe *cannot* be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

Circuit satisfiability is a good example of a problem that we don't know how to solve in polynomial time. In this problem, the input is a *boolean circuit*: a collection of and, or, and not gates connected by wires. We will assume that there are no loops in the circuit (so no delay lines or flip-flops). The input to the *circuit* is a set of m boolean (true/false) values $x_1, \ldots, x_m$. The output is a single boolean value. Given specific input values, we can calculate the output in polynomial (actually, *linear*) time using depth-first-search and evaluating the output of each gate in constant time.

The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output TRUE, or conversely, whether the circuit *always* outputs FALSE. Nobody knows how to solve this problem faster than just trying all 2^m possible inputs to the circuit, but this requires exponential time. On the other hand, nobody has ever proved that this is the best we can do; maybe there's a clever algorithm that nobody has discovered yet!



An AND gate, an OR gate, and a NOT gate.



A boolean circuit. Inputs enter from the left, and the output leaves to the right.

[1] . . . in a galaxy far far away . . .

21.2 P, NP, and co-NP

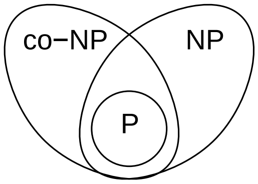
Let me define three classes of problems:

- **P** is the set of yes/no problems[2] that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.
- **NP** is the set of yes/no problems with the following property: If the answer is yes, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of problems where we can verify a YES answer quickly if we have the solution in front of us. For example, the circuit satisfiability problem is in NP. If the answer is yes, then any set of m input values that produces TRUE output is a proof of this fact; we can check the proof by evaluating the circuit in polynomial time.
- **co-NP** is the exact opposite of NP. If the answer to a problem in co-NP is *no*, then there is a proof of this fact that can be checked in polynomial time.

If a problem is in P, then it is also in NP — to verify that the answer is yes in polynomial time, we can just throw away the proof and recompute the answer from scratch. Similarly, any problem in P is also in co-NP.

One of the most important open questions in theoretical computer science is whether or not P=NP. Nobody knows. Intuitively, it should be obvious that P≠NP; the homeworks and exams in this class have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are obvious once you see them. But nobody can prove it.

Notice that the definition of NP (and co-NP) is not symmetric. Just because we can verify every yes answer quickly, we may not be able to check no answers quickly, and vice versa. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. But again, we don't have a proof; everyone believes that NP≠co-NP, but nobody really knows.



What we *think* the world looks like.

21.3 NP-hard, NP-easy, and NP-complete

A problem Π is **NP-hard** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*.[3] In other words:

[2] Technically, I should be talking about *languages*, which are just sets of bit strings. The language associated with a yes/no problem is the set of bit strings for which the answer is yes. For example, if the problem is 'Is the input graph connected?', then the corresponding language is the set of connected graphs, where each graph is represented as a bit string (for example, its adjacency matrix). P is the set of languages that can be *recognized* in polynomial time by a single-tape Turing machine. Take 579 if you want to know more.

[3] More formally, a problem Π is NP-hard if and only if, for any problem Π′ in NP, there is a polynomial-time Turing reduction from Π′ to Π—a Turing reduction just means a reduction that can be executed on a Turing machine. Polynomial-time Turing reductions are also called *Cook reductions*.

For technical reasons, complexity theorists prefer to define NP-hardness in terms of polynomial-time *many-one* reductions, which are also called *Karp reductions*. A *many-one* reduction from one language Π′ to another language Π is an

$$\boxed{\textbf{Π is NP-hard} \iff \textbf{If Π can be solved in polynomial time, then P=NP}}$$

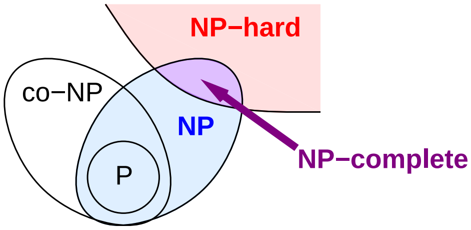
Intuitively, this is like saying that if we could solve one particular NP-hard problem quickly, then we could quickly solve *any* problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. NP-hard problems are at least as hard as any problem in NP.

Saying that a problem is NP-hard is like saying 'If I own a dog, then it can speak fluent English.' You probably don't know whether or not I own a dog, but you're probably pretty sure that I don't own a *talking* dog. Nobody has a mathematical *proof* that dogs can't speak English—the fact that no one has ever heard a dog speak English is evidence, as are the hundreds of examinations of dogs that lacked the proper mouth shape and brainpower, but mere evidence is not a mathematical proof. Nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English. So the statement 'If I own a dog, then it can speak fluent English' has a natural corollary: No one in their right mind should believe that I own a dog! Likewise, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.

The following theorem was proved by Steve Cook in 1971. I won't even sketch the proof, since I've been (deliberately) vague about the definitions.

Cook's Theorem. *Circuit satisfiability is NP-hard.*

Finally, a problem is **NP-complete** if it is both NP-hard and an element of NP (or 'NP-easy'). NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for *every* NP-complete problem. Literally *thousands* of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (*i.e.*, all) of them seems incredibly unlikely.



More of what we *think* the world looks like.

21.4 Reductions and SAT

To prove that a problem is NP-hard, we use a *reduction argument*. Reducing problem A to another problem B means describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You're already used to doing reductions, only you probably call it something else, like writing subroutines or utility functions, or modular programming. To

function $f : \Sigma^* \to \Sigma^*$ such that $x \in \Pi'$ if and only if $f(x) \in \Pi$. Every Karp reduction is a Cook reduction, but not vice versa. Every reduction (between decision problems) in these notes is a Karp reduction.

This definition is preferred partly because NP is closed under Karp reductions, but believed *not* to be closed under Cook reductions. Moreover, the two definitions of NP-hardness are equivalent only if NP=co-NP, which is considered unlikely. In fact, there are natural problems that are (1) NP-hard with respect to Cook reductions, but (2) NP-hard with respect to Karp reductions only if P=NP. On the other hand, the Karp definition *only* applies to decision problems, or more formally, sets of bit-strings.

To make things even more confusing, both Cook and Karp originally defined NP-hardness in terms of *logarithmic-space* reductions. Every logarithmic-space reduction is a polynomial-time reduction, but (we think) not vice versa. It is an open question whether relaxing the set of allowed (Cook or Karp) reductions from logarithmic-space to polynomial-time changes the set of NP-hard problems.

prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.

You should tattoo the following rule of onto the back of your hand.

To prove that problem A is NP-hard, reduce a known NP-hard problem to A.

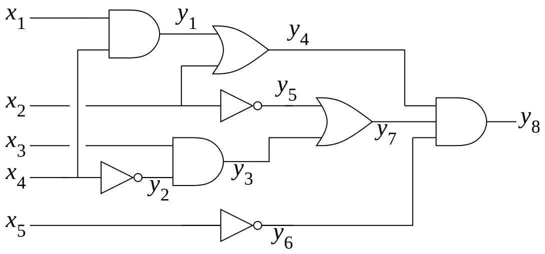
In other words, to prove that your problem is hard, you need to describe an algorithm to solve a *different* problem, which you already know is hard, using a mythical algorithm for *your* problem as a subroutine. The essential logic is a proof by contradiction. Your reduction shows implies that if your problem were easy, then the other problem would be easy, too. Equivalently, since you know the other problem is hard, your problem must also be hard.

For example, consider the *formula satisfiability* problem, usually just called *SAT*. The input to SAT is a boolean *formula* like

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

and the question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the formula evaluates to TRUE.

To show that SAT is NP-hard, we need to give a reduction from a known NP-hard problem. The only problem we know is NP-hard so far is circuit satisfiability, so let's start there. Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate, and then just writing down the list of gates separated by and. For example, we could transform the example circuit into a formula as follows:

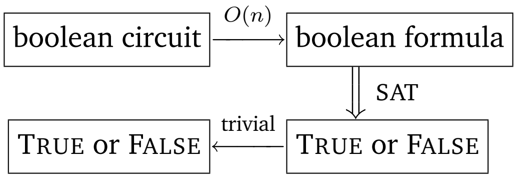


$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

Now the original circuit is satisfiable if and only if the resulting formula is satisfiable. Given a satisfying input to the circuit, we can get a satisfying assignment for the formula by computing the output of every gate. Given a satisfying assignment for the formula, we can get a satisfying input the the circuit by just ignoring the gate variables y_i.

We can transform any boolean circuit into a formula in linear time using depth-first search, and the size of the resulting formula is only a constant factor larger than the size of the circuit. Thus, we have a polynomial-time reduction from circuit satisfiability to SAT:



$$T_{\text{CSAT}}(n) \leq O(n) + T_{\text{SAT}}(O(n)) \quad \Longrightarrow \quad T_{\text{SAT}}(n) \geq T_{\text{CSAT}}(\Omega(n)) - O(n)$$

The reduction implies that if we had a polynomial-time algorithm for SAT, then we'd have a polynomial-time algorithm for circuit satisfiability, which would imply that P=NP. So SAT is NP-hard.

To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula true. We can check the proof in linear time just by reading the formula from left to right, evaluating as we go. So SAT is also in NP, and thus is actually NP-complete.

21.5 3SAT (from SAT)

A special case of SAT that is particularly useful in proving NP-hardness results is called *3SAT*.

A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (AND) of several *clauses*, each of which is the disjunction (OR) of several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

A *3CNF* formula is a CNF formula with exactly three literals per clause; the previous example is not a 3CNF formula, since its first clause has four literals and its last clause has only two. 3SAT is just SAT restricted to 3CNF formulas: Given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to TRUE?

We could prove that 3SAT is NP-hard by a reduction from the more general SAT problem, but it's easier just to start over from scratch, with a boolean circuit. We perform the reduction in several stages.

1. *Make sure every* AND *and* OR *gate has only two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k - 1$ two-input gates.

2. *Write down the circuit as a formula, with one clause per gate.* This is just the previous reduction.

3. *Change every gate clause into a CNF formula.* There are only three types of clauses, one for each type of gate:

$$\begin{aligned} a = b \wedge c &\quad\longmapsto\quad (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\quad\longmapsto\quad (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\quad\longmapsto\quad (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

4. *Make sure every clause has exactly three literals.* Introduce new variables into each one- and two-literal clause, and expand it into two clauses as follows:

$$\begin{aligned} a &\quad\longmapsto\quad (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \\ a \vee b &\quad\longmapsto\quad (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \end{aligned}$$

For example, if we start with the same example circuit we used earlier, we obtain the following 3CNF formula. Although this may look a lot more ugly and complicated than the original circuit at first glance, it's actually only a constant factor larger—every binary gate in the original circuit has been transformed into at most five clauses. Even if the formula size were a large *polynomial* function (like n^{473}) of the circuit size, we would still have a valid reduction.

$$\begin{aligned}
&(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2})\\
\wedge\, &(y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4})\\
\wedge\, &(y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6})\\
\wedge\, &(\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8})\\
\wedge\, &(y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}})\\
\wedge\, &(y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}})\\
\wedge\, &(\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}})\\
\wedge\, &(y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}})\\
\wedge\, &(y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}})\\
\wedge\, &(y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}})
\end{aligned}$$

At the end of this process, we've transformed the circuit into an equivalent 3CNF formula. The formula is satisfiable if and only if the original circuit is satisfiable. As with the more general SAT problem, the formula is only a constant factor larger than then any reasonable description of the original circuit, and the reduction can be carried out in polynomial time. Thus, we have a polynomial-time reduction from circuit satisfiability to 3SAT:

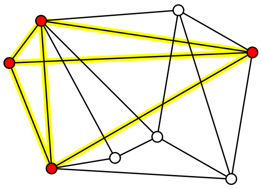
$$\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{3CNF formula}} \arrow[d, Rightarrow, "\text{3SAT}"] \\
\boxed{\text{TRUE or FALSE}} & \boxed{\text{TRUE or FALSE}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{3SAT}}(O(n)) \quad\Longrightarrow\quad T_{\text{3SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

So 3SAT is NP-hard. And since 3SAT is a special case of SAT, it is also in NP. Thus, 3SAT is NP-complete.

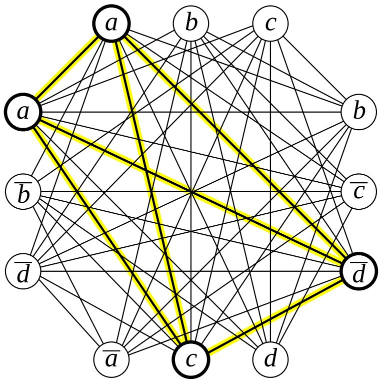
21.6 Maximum Clique Size (from 3SAT)

The next problem we'll consider is a graph problem. A *clique* is another name for a complete graph. The *maximum clique size* problem, or simply MAXCLIQUE, is to compute, given a graph, the number of nodes in its largest complete subgraph.



A graph with maximum clique size 4.

I'll prove that MAXCLIQUE is NP-hard (but not NP-complete, since it isn't a yes/no problem) using a reduction from 3SAT. I'll describe a reduction algorithm that transforms a 3CNF formula into a graph that has a clique of a certain size if and only if the formula is satisfiable. The graph has one node for each instance of each literal in the formula. Two nodes are connected by an edge if (1) they correspond to literals in different clauses and (2) those literals do not contradict each other. In particular, all the nodes that come from the same literal (in different clauses) are joined by edges. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ is transformed into the following graph. (Look for the edges that *aren't* in the graph.)



A graph derived from a 3CNF formula, and a clique of size 4.

Now suppose the original formula had k clauses. Then I claim that the formula is satisfiable if and only if the graph has a clique of size k.

1. ***k*-clique $\Longrightarrow$ satisfying assignment:** If the graph has a clique of k vertices, then each vertex must come from a different clause. To get the satisfying assignment, we declare that each literal in the clique is true. Since we only connect non-contradictory literals with edges, this declaration assigns a consistent value to several of the variables. There may be variables that have no literal in the clique; we can set these to any value we like.

2. **satisfying assignment $\Longrightarrow$ *k*-clique:** If we have a satisfying assignment, then we can choose one literal in each clause that is true. Those literals form a clique in the graph.

Thus, the reduction is correct. Since the reduction from 3CNF formula to graph takes polynomial time, we conclude that MAXCLIQUE is NP-hard. Here's a diagram of the reduction:

$$\begin{tikzcd}
\boxed{\text{3CNF formula with } k \text{ clauses}} \arrow[r, "O(n)"] & \boxed{\text{graph with } 3k \text{ nodes}} \arrow[d, Rightarrow, "\textsc{Clique}"] \\
\boxed{\textsc{True} \text{ or } \textsc{False}} & \boxed{\text{maximum clique size}} \arrow[l, "O(1)"']
\end{tikzcd}$$

$$T_{\text{3SAT}}(n) \le O(n) + T_{\textsc{MaxClique}}(O(n)) \quad\Longrightarrow\quad T_{\textsc{MaxClique}}(n) \ge T_{\text{3SAT}}(\Omega(n)) - O(n)$$

21.7 Independent Set (from Clique)

An *independent set* is a collection of vertices is a graph with no edges between them. The INDEPENDENTSET problem is to find the largest independent set in a given graph.

There is an easy proof that INDEPENDENTSET is NP-hard, using a reduction from CLIQUE. Any graph G has a *complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—(u, v) is an edge in $\overline{G}$ if and only if it is *not* an edge in G. A set of vertices forms a clique in G if and only if the same vertices are an independent set in $\overline{G}$. Thus, we can compute the largest clique in a graph simply by computing the largest independent set in the complement of the graph.

$$\begin{tikzcd}
\boxed{\text{graph } G} \arrow[r, "O(n)"] & \boxed{\text{complement graph } \overline{G}} \arrow[d, Rightarrow, "\textsc{IndependentSet}"] \\
\boxed{\text{largest clique}} & \boxed{\text{largest independent set}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

21.8 Vertex Cover (from Independent Set)

A *vertex cover* of a graph is a set of vertices that touches every edge in the graph. The VertexCover problem is to find the smallest vertex cover in a given graph.

Again, the proof of NP-hardness is simple, and relies on just one fact: If I is an independent set in a graph $G = (V, E)$, then $V \setminus I$ is a vertex cover. Thus, to find the *largest* independent set, we just need to find the vertices that aren't in the *smallest* vertex cover of the same graph.

$$\begin{tikzcd}
\boxed{\text{graph } G = (V, E)} \arrow[r, "\text{trivial}"] & \boxed{\text{graph } G = (V, E)} \arrow[d, Rightarrow, "\textsc{VertexCover}"] \\
\boxed{\text{largest independent set } V \setminus S} & \boxed{\text{smallest vertex cover } S} \arrow[l, "O(n)"']
\end{tikzcd}$$

21.9 Graph Coloring (from 3SAT)

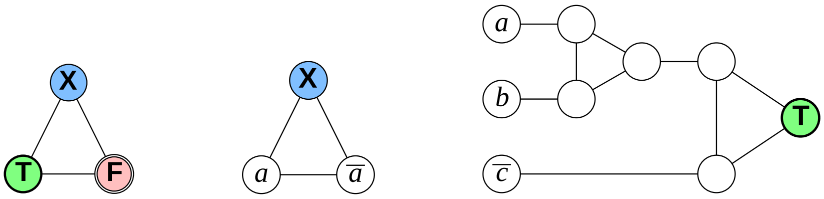
A *c-coloring* of a graph is a map $C : V \to \{1, 2, \ldots, c\}$ that assigns one of c 'colors' to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring. To show that this problem is NP-hard, it's enough to consider the special case 3Colorable: Given a graph, does it have a 3-coloring?

To prove that 3Colorable is NP-hard, we use a reduction from 3SAT. Given a 3CNF formula, we produce a graph as follows. The graph consists of a *truth* gadget, one *variable* gadget for each variable in the formula, and one *clause* gadget for each clause in the formula.

The truth gadget is just a triangle with three vertices T, F, and X, which intuitively stand for True, false, and other. Since these vertices are all connected, they must have different colors in any 3-coloring. For the sake of convenience, we will name those colors True, False, and Other. Thus, when we say that a node is colored True, all we mean is that it must be colored the same as the node T.

The variable gadget for a variable a is also a triangle joining two new nodes labeled a and $\overline{a}$ to node X in the truth gadget. Node a must be colored either True or False, and so node $\overline{a}$ must be colored either False or True, respectively.

Finally, each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabeled nodes and ten edges; see the figure below. If all three literal nodes in the clause gadget are colored False, then the rightmost vertex in the gadget cannot have one of the three colors. Since the variable gadgets force each literal node to be colored either True or False, in any valid 3-coloring, at least one of the three literal nodes is colored True. I need to emphasize here that the final graph contains only *one* node T, only *one* node F, and only *two* nodes a and $\bar{a}$ for each variable.

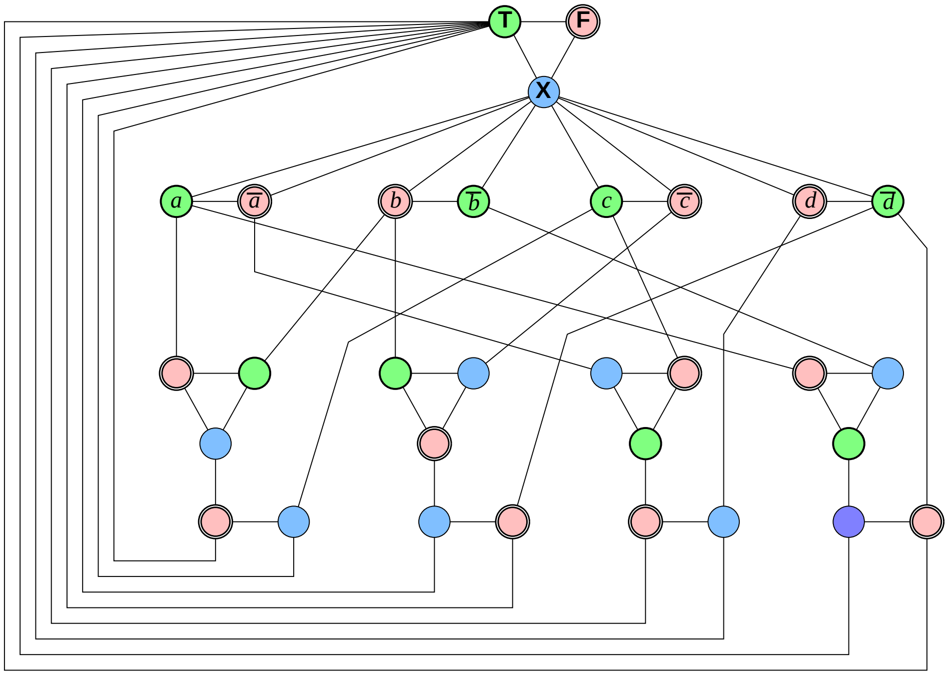


Gadgets for the reduction from 3SAT to 3-Colorability:
The truth gadget, a variable gadget for a, and a clause gadget for $(a \vee b \vee \bar{c})$.

The proof of correctness is just brute force. If the graph is 3-colorable, then we can extract a satisfying assignment from any 3-coloring—at least one of the three literal nodes in every clause gadget is colored TRUE. Conversely, if the formula is satisfiable, then we can color the graph according to any satisfying assignment.

$$\begin{tikzcd}
\boxed{\text{3CNF formula}} \arrow[r, "O(n)"] & \boxed{\text{graph}} \arrow[d, Rightarrow, "\text{3COLORABLE}"] \\
\boxed{\text{TRUE or FALSE}} & \boxed{\text{TRUE or FALSE}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ that I used to illustrate the MAXCLIQUE reduction would be transformed into the following graph. The 3-coloring is one of several that correspond to the satisfying assignment $a = c =$ TRUE, $b = d =$ FALSE.



A 3-colorable graph derived from a satisfiable 3CNF formula.

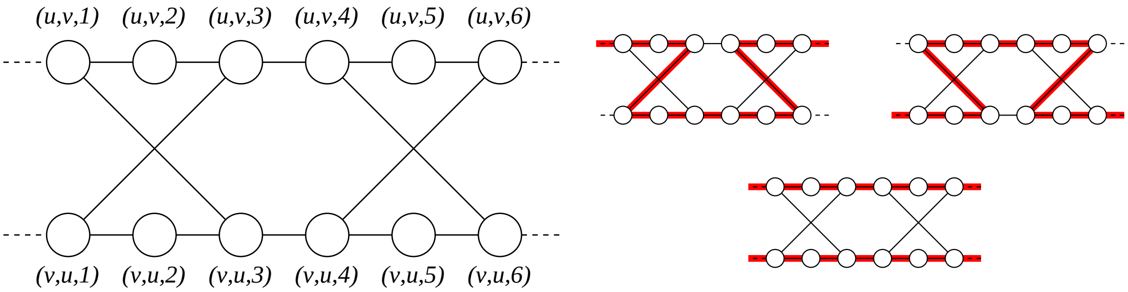
We can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge. Thus, 3COLORING is in NP, and therefore NP-complete. Moreover, since 3COLORING is a special case of the more general graph coloring problem—What is the minimum number of colors?—the more problem is also NP-hard, but *not* NP-complete, because it's not a yes/no problem.

21.10 Hamiltonian Cycle (from Vertex Cover)

A *Hamiltonian cycle* is a graph is a cycle that visits every vertex exactly once. This is very different from an *Eulerian cycle*, which is actually a closed *walk* that traverses every *edge* exactly once. Eulerian cycles are easy to find and construct in linear time using a variant of depth-first search. Finding Hamiltonian cycles, on the other hand, is NP-hard.

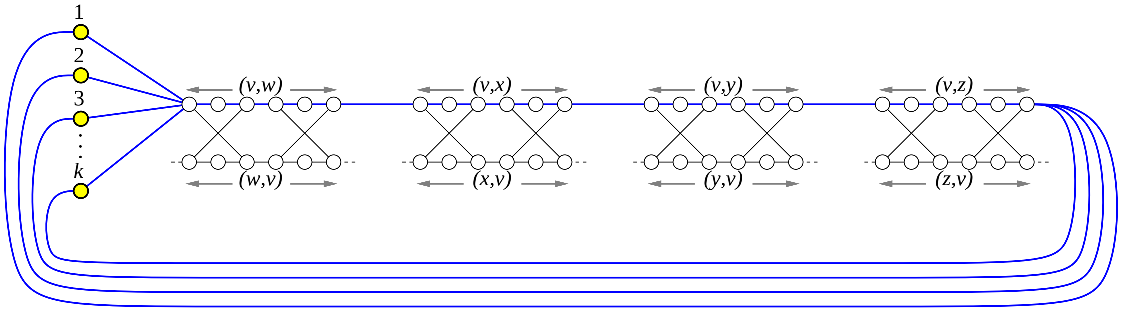
To prove this, we use a reduction from the vertex cover problem. Given a graph G and an integer k, we need to transform it into another graph G', such that G' has a Hamiltonian cycle if and only if G has a vertex cover of size k. As usual, our transformation uses several gadgets.

- For each edge (u,v) in G, we have an *edge gadget* in G' consisting of twelve vertices and fourteen edges, as shown below. The four corner vertices $(u,v,1)$, $(u,v,6)$, $(v,u,1)$, and $(v,u,6)$ each have an edge leaving the gadget. A Hamiltonian cycle can only pass through an edge gadget in one of three ways. Eventually, these will correspond to one or both of the vertices u and v being in the vertex cover.



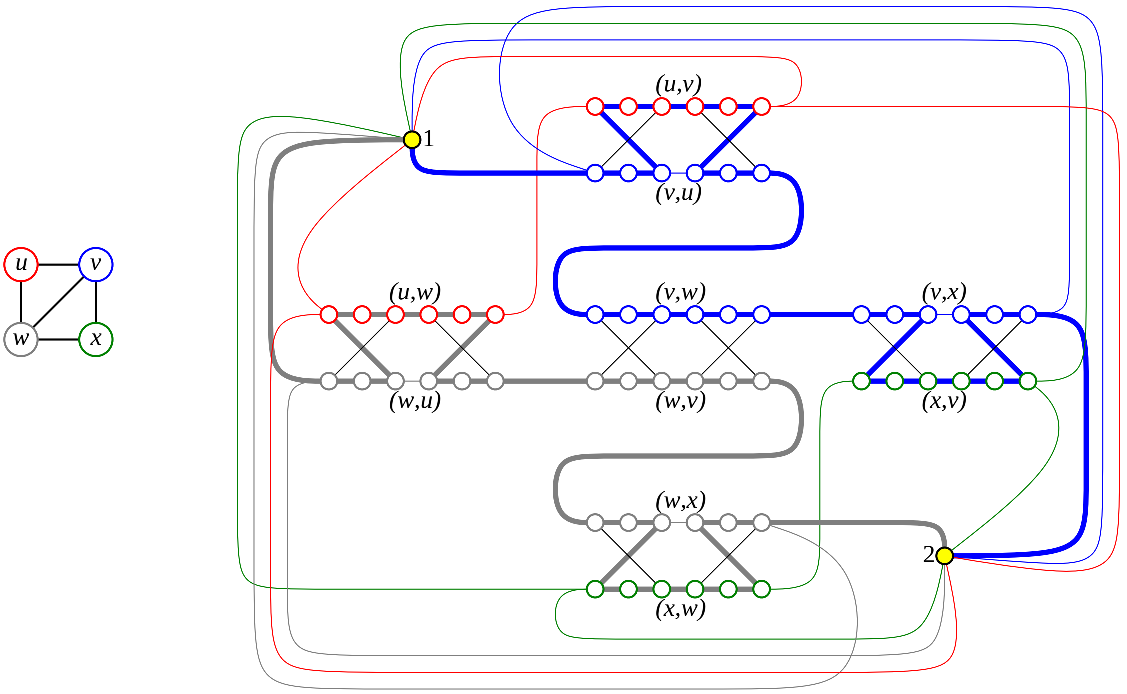
An edge gadget for (u,v) and the only possible Hamiltonian paths through it.

- G' also contains k *cover vertices*, simply numbered 1 through k.
- Finally, for each vertex u in G, we string together all the edge gadgets for edges (u,v) into a single *vertex chain*, and then connect the ends of the chain to all the cover vertices. Specifically, suppose u has d neighbors $v_1, v_2, \ldots, v_d$. Then G' has $d-1$ edges between $(u,v_i,6)$ and $(u,v_{i+1},1)$, plus k edges between the cover vertices and $(u,v_1,1)$, and finally k edges between the cover vertices and $(u,v_d,6)$.



The vertex chain for v: all edge gadgets involving v are strung together and joined to the k cover vertices.

It's not hard to prove that if $\{v_1, v_2, \ldots, v_k\}$ is a vertex cover of G, then G' has a Hamiltonian cycle—start at cover vertex 1, through traverse the vertex chain for v_1, then visit cover vertex 2, then traverse the vertex chain for v_2, and so forth, eventually returning to cover vertex 1. Conversely, any Hamiltonian cycle in G' alternates between cover vertices and vertex chains, and the vertex chains correspond to the k vertices in a vertex cover of G. (This is a little harder to prove.) Thus, G has a vertex cover of size k if and only if G' has a Hamiltonian cycle.



The original graph G with vertex cover $\{v, w\}$, and the transformed graph G' with a corresponding Hamiltonian cycle. Vertex chains are colored to match their corresponding vertices.

The transformation from G to G' takes at most $O(n^2)$ time, so the Hamiltonian cycle problem is NP-hard. Moreover, since we can easily verify a Hamiltonian cycle in linear time, the Hamiltonian cycle problem is in NP, and therefore NP-complete.

A closely related problem to Hamiltonian cycles is the famous *traveling salesman problem*—Given a *weighted* graph G, find the shortest cycle that visits every vertex. Finding the shortest cycle is obviously harder than determining if a cycle exists at all, so the traveling salesman problem is also NP-hard.

21.11 Subset Sum (from Vertex Cover)

The last problem that we will prove NP-hard is the SUBSETSUM problem considered in the very first lecture on recursion: Given a set X of integers and an integer t, determine whether X has a subset whose elements sum to t.

To prove this problem is NP-hard, we apply a reduction from the vertex cover problem. Given a graph G and an integer k, we need to transform it into set of integers X and an integer t, such that X has a subset that sums to t if and only if G has an vertex cover of size k. Our transformation uses just two 'gadgets'; these are *integers* representing vertices and edges in G.

Number the *edges* of G arbitrarily from 0 to $m - 1$. Our set X contains the integer $b_i := 4^i$ for each edge i, and the integer

$$a_v := 4^m + \sum_{i \in \Delta(v)} 4^i$$

for each vertex v, where $\Delta(v)$ is the set of edges that have v has an endpoint. Alternately, we can think of each integer in X as an $(m+1)$-digit number written in base 4. The mth digit is 1 if the integer represents a vertex, and 0 otherwise. For each $i < m$, the ith digit is 1 if the integer represents edge i or one of its endpoints, and 0 otherwise. Finally, we set the target sum

$$t := k \cdot 4^m + \sum_{i=0}^{m-1} 2 \cdot 4^i.$$

Now let's prove that the reduction is correct. First, suppose there is a vertex cover of size k in the original graph G. Consider the subset $X_C \subseteq X$ that includes a_v for every vertex v in the vertex cover, and b_i for every edge i that has *exactly one* vertex in the cover. The sum of these integers, written in base 4, has a 2 in each of the first m digits; in the most significant digit, we are summing exactly k 1's. Thus, the sum of the elements of X_C is exactly t.

On the other hand, suppose there is a subset $X' \subseteq X$ that sums to t. Specifically, we must have

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

for some subsets $V' \subseteq V$ and $E' \subseteq E$. Again, if we sum these base-4 numbers, there are no carries in the first m digits, because for each i there are only three numbers in X whose ith digit is 1. Each edge number b_i contributes only one 1 to the ith digit of the sum, but the ith digit of t is 2. Thus, for each edge in G, at least one of its endpoints must be in V'. In other words, V is a vertex cover. On the other hand, only vertex numbers are larger than 4^m, and $\lfloor t/4^m \rfloor = k$, so V' has at most k elements. (In fact, it's not hard to see that V' has *exactly* k elements.)

For example, given the four-vertex graph used on the previous page to illustrate the reduction to Hamiltonian cycle, our set X might contain the following base-4 integers:

$$\begin{aligned}
b_{uv} &:= 010000_4 = 256 \\
b_{uw} &:= 001000_4 = 64 \\
b_{vw} &:= 000100_4 = 16 \\
b_{vx} &:= 000010_4 = 4 \\
b_{wx} &:= 000001_4 = 1 \\[1ex]
a_u &:= 111000_4 = 1344 \\
a_v &:= 110110_4 = 1300 \\
a_w &:= 101101_4 = 1105 \\
a_x &:= 100011_4 = 1029
\end{aligned}$$

If we are looking for a vertex cover of size 2, our target sum would be $t := 222222_4 = 2730$. Indeed, the vertex cover $\{v, w\}$ corresponds to the subset $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, whose sum is $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$.

The reduction can clearly be performed in polynomial time. Since VertexCover is NP-hard, it follows that SubsetSum is NP-hard.

There is one subtle point that needs to be emphasized here. Way back at the beginning of the semester, we developed a dynamic programming algorithm to solve SubsetSum in time $O(nt)$. Isn't this a polynomial-time algorithm? Nope. True, the running time is polynomial in n and t, but in order to qualify as a true polynomial-time algorithm, the running time must be a polynomial function *of the size of the input*. The *values* of the elements of X and the target sum t could be exponentially larger than the number of input bits. Indeed, the reduction we just described produces exponentially large integers, which would force our dynamic programming algorithm to run in exponential time. Algorithms like this are called *pseudo-polynomial-time*, and any NP-hard problem with such an algorithm is called *weakly* NP-hard.

21.12 Other Useful NP-hard Problems

Literally thousands of different problems have been proved to be NP-hard. I want to close this note by listing a few NP-hard problems that are useful in deriving reductions. I won't describe the NP-hardness for these problems, but you can find most of them in Garey and Johnson's classic *Scary Black Book of NP-Completeness.*[4]

- PLANARCIRCUITSAT: Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output TRUE? This can be proved NP-hard by reduction from the general circuit satisfiability problem, by replacing each crossing with a small series of gates. (This is an easy exercise.)

- NOTALLEQUAL3SAT: Given a 3CNF formula, is there an assignment of values to the variables so that every clause contains at least one true literal *and* at least one false literal? This can be proved NP-hard by reduction from the usual 3SAT.

- PLANAR3SAT: Given a 3CNF boolean formula, consider a bipartite graph whose vertices are the clauses and variables, where an edge indicates that a variable (or its negation) appears in a clause. If this graph is planar, the 3CNF formula is also called planar. The PLANAR3SAT problem asks, given a planar 3CNF formula, whether it has a satisfying assignment. This can be proven NP-hard by reduction from PLANARCIRCUITSAT.

- PLANARNOTALLEQUAL3SAT: You get the idea.

- EXACT3DIMENSIONALMATCHING or X3M: Given a set S and a collection of three-element subsets of S, called *triples,* is there a subcollection of disjoint triples that exactly cover S? This can be proved NP-hard by a reduction from 3SAT.

- PARTITION: Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \varnothing$, and
$$\sum_{a \in A} a = \sum_{b \in B} b?$$
This can be proved NP-hard by a simple reduction from SUBSETSUM. Like SUBSETSUM, the PARTITION problem is only weakly NP-hard.

- 3PARTITION: Given a set S of $3n$ integers, can it be partitioned into n disjoint subsets, each with 3 elements, such that every subset has exactly the same sum? Note that this is *very* different from the PARTITION problem; I didn't make up the names. This can be proved NP-hard by reduction from X3M. Unlike PARTITION, the 3PARTITION problem is *strongly* NP-hard, that is, it remains NP-hard even if the input numbers are less than some polynomial in n. The similar problem of dividing a set of $2n$ integers into n equal-weight *two*-element sets can be solved in $O(n \log n)$ time.

- SETCOVER: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest subcollection of S_i's that contains all the elements of $\bigcup_i S_i$. This is a generalization of both VERTEXCOVER and X3M.

- HITTINGSET: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathcal{S}$. This is also a generalization of VERTEXCOVER.

[4] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

- LONGESTPATH: Given a non-negatively weighted graph G and two vertices u and v, what is the longest simple path from u to v in the graph? A path is *simple* if it visits each vertex at most once. This is a generalization of the HAMILTONIANPATH problem. Of course, the corresponding *shortest* path problem is in P.
- STEINERTREE: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex? If *every* vertex is marked, the minimum Steiner tree is just the minimum spanning tree; if exactly two vertices are marked, the minimum Steiner tree is just the shortest path between them. This can be proved NP-hard by reduction to HAMILTONIANPATH.

Most interesting puzzles and solitaire games have been shown to be NP-hard, or to have NP-hard generalizations. (Arguably, if a game or puzzle isn't at least NP-hard, it isn't interesting!) Some familiar examples include Minesweeper (by reduction from CIRCUITSAT)[5], Tetris (by reduction from 3PARTITION)[6], and Shanghai (by reduction from 3SAT)[7]. Most two-player games[8] like tic-tac-toe, reversi, checkers, chess, go, mancala—or more accurately, appropriate generalizations of these constant-size games to arbitrary board sizes—are not just NP-hard, but PSPACE-hard or even EXP-hard.[9]

Exercises

0. Describe and analyze an algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. *[Hint: There is a one-line solution!]*

1. (a) Describe and analyze and algorithm to solve PARTITION in time $O(nM)$, where n is the size of the input set and M is the sum of the absolute values of its elements.

   (b) Why doesn't this algorithm imply that P=NP?

[5] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf

[6] Ron Breukelaar*, Erik D. Demaine, Susan Hohenberger*, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell*. Tetris is Hard, Even to Approximate. International Journal of Computational Geometry and Applications 14:41–68, 2004. The reduction was *considerably* simplified between its discovery in 2002 and its publication in 2004.

[7] David Eppstein. Computational complexity of games and puzzles. http://www.ics.uci.edu/~eppstein/cgt/hard.html

[8] For a good (but now slightly dated) overview of known results on the computational complexity of games and puzzles, see Erik D. Demaine's survey "Playing Games with Algorithms: Algorithmic Combinatorial Game Theory" at http://arxiv.org/abs/cs.CC/0106019.

[9] PSPACE and EXP are the next two big steps above NP in the complexity hierarchy.

PSPACE is the set of decision problems that can be solved using polynomial space. Every problem in NP (and therefore in P) is also in PSPACE. It is generally believed that NP $\neq$ PSPACE, but nobody can even prove that P $\neq$ PSPACE. A problem Π is PSPACE-hard if, for any problem Π' that can be solved using polynomial *space*, there is a polynomial-*time* many-one reduction from Π' to Π. If any PSPACE-hard problem is in NP, then PSPACE=NP.

EXP (also called EXPTIME) is the set of decision problems that can be solved in exponential time: at most 2^{n^c} for some $c > 0$. Every problem in PSPACE (and therefore in NP (and therefore in P)) is also in EXP. It is generally believed that PSPACE $\subsetneq$ EXP, but nobody can even prove that NP $\neq$ EXP. We *do* know that P $\subsetneq$ EXP, but we do not know of a single natural decision problem in P $\setminus$ EXP. A problem Π is EXP-hard if, for any problem Π' that can be solved in *exponential* time, there is a *polynomial*-time many-one reduction from Π' to Π. If any EXP-hard problem is in PSPACE, then EXP=PSPACE.

Then there's NEXP, then EXPSPACE, then EEXP, then NEEXP, then EEXPSPACE, and so on ad infinitum. Whee!

2. Consider the following problem, called BoxDepth: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BoxDepth to MaxClique.

   (b) Describe and analyze a polynomial-time algorithm for BoxDepth. *[Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.]*

   (c) Why don't these two results imply that P=NP?

3. (a) Describe a polynomial-time reduction from Partition to SubsetSum.

   (b) Describe a polynomial-time reduction from SubsetSum to Partition.

4. (a) Using the gadget in Figure 1(a), prove that deciding whether a given *planar* graph is 3-colorable is NP-complete. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*

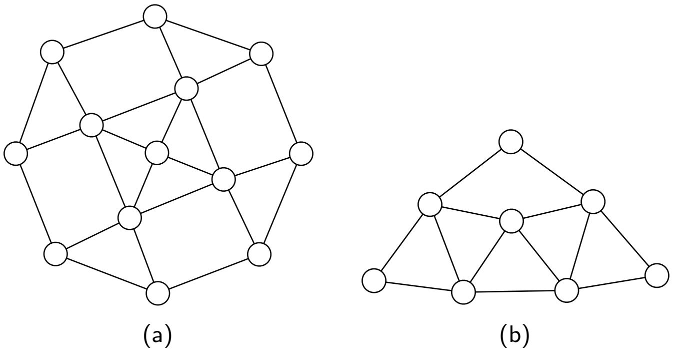


Figure 1. (a) Gadget for planar 3-colorability. (b) Gadget for degree-4 planar 3-colorability.

   (b) Using part (a) and the gadget in Figure 1(b), prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-complete. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*

5. Prove that PlanarCircuitSat is NP-complete.

6. Prove that NotAllEqual3SAT is NP-complete.

7. Prove that the following problems are NP-complete.

   (a) Given two undirected graphs G and H, is G isomorphic to a subgraph of H?

   (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 17?

   (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

8. The RectangleTiling problem asks, given a 'large' rectangle R and several 'small' rectangles $r_1, r_2, \ldots, r_n$, whether the small rectangles can be placed inside the larger rectangle with no gaps or overlaps. Prove that RectangleTiling is NP-complete.

9. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-complete.

   (b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-complete. *[Hint: Use part (a).]*

10. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* consists of removing two pebbles from a vertex v and adding one pebble to an arbitrary neighbor of v. (Obviously, the vertex v must have at least two pebbles before the move.) The PEBBLEDESTRUCTION problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PEBBLEDESTRUCTION is NP-complete.

11. **Reducing Construction Problems to Decision Problems**

    (a) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, the length of the shortest Hamiltonian cycle in G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary weighted graph G, the shortest Hamiltonian cycle in G, using this magic black box as a subroutine.

    (b) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

    (c) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.]*

    (d) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary boolean formula Φ, whether Φ is satisfiable. Describe and analyze a **polynomial-time** algorithm that either computes a satisfying assignment for a given boolean formula or correctly reports that no such assignment exists, using the magic black box as a subroutine.

    *(e) Suppose you are given a magic black box that can determine **in polynomial time**, given an initial Tetris configuration and a finite sequence of Tetris pieces, whether a perfect player can play every piece. (This problem is NP-hard.) Describe and analyze a **polynomial-time** algorithm that computes the shortest Hamiltonian cycle in a given weighted graph in polynomial time, using this magic black box as a subroutine. *[Hint: Use part (a). You do not need to know anything about Tetris to solve this problem.]*

12. **Two is easy; three is hard.**

    (a) Describe and analyze a polynomial-time algorithm for 2COLOR. Given an undirected graph G, your algorithm will determine in polynomial time whether G has a proper coloring that uses only two colors.

    (b) Describe and analyze a polynomial-time algorithm for 2SAT. Given a boolean formula Φ in conjunctive normal form, with exactly *two* literals per clause, your algorithm will determine in polynomial time whether Φ has a satisfying assignment.

    (c) Describe and analyze a polynomial-time algorithm for 2PARTITION. Given a set S of $2n$ positive integers, your algorithm will determine in polynomial time whether the elements of S can be split into n disjoint pairs whose sums are all equal.

Le mieux est l'ennemi du bien. [The best is the enemy of the good.]
— Voltaire, *La Bégueule* (1772)

Who shall forbid a wise skepticism, seeing that there is no practical question on which any thing more than an approximate solution can be had?
— Ralph Waldo Emerson, *Representative Men* (1850)

All progress is precarious, and the solution of one problem
brings us face to face with another problem.
— Martin Luther King Jr., "Strength to Love" (1960)

I Approximation Algorithms

I.1 Load Balancing

On the future smash hit reality-TV game show *Grunt Work*, scheduled to air Thursday nights at 3am (2am Central) on ESPNπ, the contestants are given a series of utterly pointless tasks to perform. Each task has a predetermined time limit; for example, "Sharpen this pencil for 17 seconds", or "Pour pig's blood on your head and sing The Star-Spangled Banner for two minutes", or "Listen to this 75-minute algorithms lecture". The directors of the show want you to assign each task to one of the contestants, so that the last task is completed as early as possible. When your predecessor correctly informed the directors that their problem is NP-hard, he was immediately fired. "Time is money!" they screamed at him. "We don't need perfection. Wake up, dude, this is *television*!"

Less facetiously, suppose we have a set of n jobs, which we want to assign to m machines. We are given an array $T[1\,..\,n]$ of non-negative numbers, where $T[j]$ is the running time of job j. We can describe an *assignment* by an array $A[1\,..\,n]$, where $A[j] = i$ means that job j is assigned to machine i. The *makespan* of an assignment is the maximum time that any machine is busy:

$$\text{makespan}(A) = \max_i \sum_{A[j]=i} T[j]$$

The *load balancing* problem is to compute the assignment with the smallest possible makespan.

It's not hard to prove that the load balancing problem is NP-hard by reduction from PARTITION: The array $T[1\,..\,n]$ can be evenly partitioned if and only if there is an assignment to two machines with makespan exactly $\sum_i T[i]/2$. A slightly more complicated reduction from 3PARTITION implies that the load balancing problem is *strongly* NP-hard. If we really need the optimal solution, there is a dynamic programming algorithm that runs in time $O(nM^m)$, where M is the minimum makespan, but that's just horrible.

There is a fairly natural and efficient greedy heuristic for load balancing: consider the jobs one at a time, and assign each job to the machine with the earliest finishing time.

```
GREEDYLOADBALANCE($T[1\,..\,n], m$):
  for $i \leftarrow 1$ to $m$
      $Total[i] \leftarrow 0$

  for $j \leftarrow 1$ to $n$
      $min \leftarrow 1$
      for $i \leftarrow 2$ to $n$
          if $Total[i] < Total[min]$
              $min \leftarrow i$
      $A[j] \leftarrow min$
      $Total[min] \leftarrow Total[min] + T[j]$

  return $A[1\,..\,m]$
```

Theorem 1. *The makespan of the assignment computed by* GREEDYLOADBALANCE *is at most twice the makespan of the optimal assignment.*

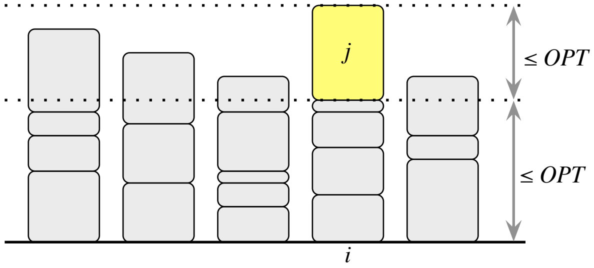
Proof: Fix an arbitrary input, and let OPT denote the makespan of its optimal assignment. The approximation bound follows from two trivial observations. First, the makespan of any assignment (and therefore of the optimal assignment) is at least the duration of the longest job. Second, the makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

$$OPT \geq \max_j T[j] \qquad \text{and} \qquad OPT \geq \frac{1}{m}\sum_{j=1}^{n} T[j]$$

Now consider the assignment computed by GREEDYLOADBALANCE. Suppose machine i has the largest total running time, and let j be the last job assigned to job i. Our first trivial observation implies that $T[j] \leq OPT$. To finish the proof, we must show that $\mathit{Total}[i] - T[j] \leq OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $\mathit{Total}[i] - T[j] \leq \mathit{Total}[k]$ for all k. (Some values $\mathit{Total}[k]$ may have increased since job j was assigned, but that only helps us.) In particular, $\mathit{Total}[i] - T[j]$ is less than or equal to the *average* finishing time over all machines. Thus,

$$\mathit{Total}[i] - T[j] \leq \frac{1}{m}\sum_{i=1}^{m} \mathit{Total}[i] = \frac{1}{m}\sum_{j=1}^{n} T[j] \leq OPT$$

by our second trivial observation. We conclude that the makespan $\mathit{Total}[i]$ is at most $2 \cdot OPT$. □



Proof that GREEDYLOADBALANCE is a 2-approximation algorithm

GREEDYLOADBALANCE is an *online* algorithm: It assigns jobs to machines in the order that the jobs appear in the input array. Online approximation algorithms are useful in settings where inputs arrive in a stream of unknown length—for example, real jobs arriving at a real scheduling algorithm. In this online setting, it may be *impossible* to compute an optimum solution, even in cases where the offline problem (where all inputs are known in advance) can be solved in polynomial time. In particular, there is *no* online load balancing algorithm that achieves an approximation factor better than 2. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

Even though GREEDYLOADBALANCE is the best possible *online* load balancing algorithm, in our original offline setting, we can improve the approximation factor by sorting the jobs before piping them through the greedy algorithm.

```
SORTEDGREEDYLOADBALANCE($T[1..n], m$):
    sort $T$ in decreasing order
    return GREEDYLOADBALANCE($T, m$)
```

Theorem 2. *The makespan of the assignment computed by* SortedGreedyLoadBalance *is at most* $3/2$ *times the makespan of the optimal assignment.*

Proof: Let i be the busiest machine in the schedule computed by SortedGreedyLoadBalance. If only one job is assigned to machine i, then the greedy schedule is actually optimal, and the theorem is trivially true. Otherwise, let j be the last job assigned to machine i. Since each of the first m jobs is assigned to a unique processor, we must have $j \geq m+1$. As in the previous proof, we know that $\mathit{Total}[i] - T[j] \leq OPT$.

In the optimal assignment, at least two of the first $m+1$ jobs, say jobs k and ℓ, must be scheduled on the same processor. Thus, $T[k] + T[\ell] \leq OPT$. Since $\max\{k, \ell\} \leq m+1 \leq j$, and the jobs are sorted in decreasing order by direction, we have

$$T[j] \leq T[m+1] \leq T[\max\{k,\ell\}] = \min\{T[k], T[\ell]\} \leq OPT/2.$$

We conclude that the makespan $\mathit{Total}[i]$ is at most $3 \cdot OPT/2$, as claimed. □

I.2 Generalities

Consider an arbitrary optimization problem. Let $OPT(X)$ denote the value of the optimal solution for a given input X, and let $A(X)$ denote the value of the solution computed by algorithm A given the same input X. We say that A is an **α-approximation algorithm** if

$$\frac{OPT(X)}{A(X)} \leq \alpha \quad \text{and} \quad \frac{A(X)}{OPT(X)} \leq \alpha$$

for all inputs X. Generally, only one of these two inequalities will be important. For maximization problems, where we want to compute a solution whose cost is as small as possible, the first inequality is trivial. For maximization problems, where we want a solution whose value is as large as possible, the second inequality is trivial. A 1-approximation algorithm always returns the exact optimal solution. The approximation factor α may be either a constant or a function of the input size.

I.3 Greedy Vertex Cover

Recall that the *vertex color* problem asks, given a graph G, for the smallest set of vertices of G that cover every edge. This is one of the first NP-hard problems introduced in the first week of class. There is a natural and efficient greedy heuristic[1] for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

```
GreedyVertexCover($G$):
  $C \leftarrow \varnothing$
  while $G$ has at least one edge
      $v \leftarrow$ vertex in $G$ with maximum degree
      $G \leftarrow G \setminus v$
      $C \leftarrow C \cup v$
  return $C$
```

Obviously this algorithm doesn't compute the optimal vertex cover—that would imply P=NP!—but it does compute a reasonably close approximation.

[1] A *heuristic* is an algorithm that doesn't work.

Theorem 3. GreedyVertexCover *is an* $O(\log n)$*-approximation algorithm.*

Proof: For all i, let G_i denote the graph G after i iterations of the main loop, and let d_i denote the maximum degree of any node in G_{i-1}. We can define these variables more directly by adding a few extra lines to our algorithm:

```
GreedyVertexCover($G$):
  $C \leftarrow \varnothing$
  $G_0 \leftarrow G$
  $i \leftarrow 0$
  while $G_i$ has at least one edge
      $i \leftarrow i+1$
      $v_i \leftarrow$ vertex in $G_{i-1}$ with maximum degree
      $d_i \leftarrow \deg_{G_{i-1}}(v_i)$
      $G_i \leftarrow G_{i-1} \setminus v_i$
      $C \leftarrow C \cup v_i$
  return $C$
```

Let $|G_{i-1}|$ denote the number of edges in the graph G_{i-1}. Let C^* denote the optimal vertex cover of G, which consists of OPT vertices. Since C^* is also a vertex cover for G_{i-1}, we have

$$\sum_{v \in C^*} \deg_{G_{i-1}}(v) \geq |G_{i-1}|.$$

In other words, the *average* degree in G_i of any node in C^* is at least $|G_{i-1}|/OPT$. It follows that G_{i-1} has at least one node with degree at least $|G_{i-1}|/OPT$. Since d_i is the maximum degree of any node in G_{i-1}, we have

$$d_i \geq \frac{|G_{i-1}|}{OPT}$$

Moreover, for any $j \geq i-1$, the subgraph G_j has no more edges than G_{i-1}, so $d_i \geq |G_j|/OPT$. This observation implies that

$$\sum_{i=1}^{OPT} d_i \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{i-1}|}{OPT} \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{OPT}|}{OPT} \;=\; |G_{OPT}| \;=\; |G| - \sum_{i=1}^{OPT} d_i.$$

In other words, the first OPT iterations of GreedyVertexCover remove at least half the edges of G. Thus, after at most $OPT \ \lg|G| \leq 2\,OPT \lg n$ iterations, all the edges of G have been removed, and the algorithm terminates. We conclude that GreedyVertexCover computes a vertex cover of size $O(OPT \ \log n)$. □

I.4 Set Cover and Hitting Set

The greedy algorithm for vertex cover can be applied almost immediately to two more general problems: *set cover* and *hitting set*. The input for both of these problems is a *set system* $(X, \mathcal{F})$, where X is a finite *ground set*, and $\mathcal{F}$ is a family of subsets of X. A *set cover* of a set system $(X, \mathcal{F})$ is a subfamily of sets in $\mathcal{F}$ whose union is the entire ground set X. A *hitting set* for $(X, \mathcal{F})$ is a subset of the ground set X that intersects every set in $\mathcal{F}$.

An undirected graph can be cast as a set system in two different ways. In one formulation, the ground set X contains the vertices, and each edge defines a set of two vertices in $\mathcal{F}$. In this formulation, a vertex cover is a hitting set. In the other formulation, the *edges* are the ground set, the *vertices* define the family of subsets, and a vertex cover is a set cover.

Here are the natural greedy algorithms for finding a small set cover and finding a small hitting set. GREEDYSETCOVER finds a set cover whose size is at most $O(\log|\mathcal{F}|)$ times the size of smallest set cover. GREEDYHITTINGSET finds a hitting set whose size is at most $O(\log|X|)$ times the size of the smallest hitting set.

```
GREEDYSETCOVER($X, \mathcal{F}$):
  $\mathcal{C} \leftarrow \varnothing$
  while $X \neq \varnothing$
      $S \leftarrow \arg\max_{S \in \mathcal{F}} |S \cap X|$
      $X \leftarrow X \setminus S$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
  return $\mathcal{C}$
```

```
GREEDYHITTINGSET($X, \mathcal{F}$):
  $H \leftarrow \varnothing$
  while $\mathcal{F} \neq \varnothing$
      $x \leftarrow \arg\max_{x \in X} |\{S \in \mathcal{F} \mid x \in S\}|$
      $\mathcal{F} \leftarrow \mathcal{F} \setminus \{S \in \mathcal{F} \mid x \in S\}$
      $H \leftarrow H \cup \{x\}$
  return $H$
```

The similarity between these two algorithms is no coincidence. For any set system $(X, \mathcal{F})$, there is a *dual* set system $(\mathcal{F}, X^*)$ defined as follows. For any element $x \in X$ in the ground set, let x^* denote the subfamily of sets in $\mathcal{F}$ that contain x:

$$x^* = \{S \in \mathcal{F} \mid x \in S\}.$$

Finally, let X^* denote the collection of all subsets of the form x^*:

$$X^* = \{x^* \mid x \in S\}.$$

As an example, suppose X is the set of letters of alphabet and $\mathcal{F}$ is the set of last names of student taking CS 473G this semester. Then X^* has 26 elements, each containing the subset of CS 473G students whose last name contains a particular letter of the alphabet. For example, m^* is the set of students whose last names contain the letter m.

There is a direct one-to-one correspondence between the ground set X and the dual set family X^*. It is a tedious but instructive exercise to prove that the dual of the dual of any set system is isomorphic to the original set system—$(X^*, \mathcal{F}^*)$ is essentially the same as $(X, \mathcal{F})$. It is also easy to prove that a set cover for any set system $(X, \mathcal{F})$ is also a hitting set for the dual set system $(\mathcal{F}, X^*)$, and therefore a hitting set for any set system $(X, \mathcal{F})$ is isomorphic to a set cover for the dual set system $(\mathcal{F}, X^*)$.

I.5 Vertex Cover, Again

The greedy approach doesn't always lead to the best approximation algorithms. Consider the following alternate heuristic for vertex cover:

```
DUMBVERTEXCOVER($G$):
  $C \leftarrow \varnothing$
  while $G$ has at least one edge
      $(u, v) \leftarrow$ any edge in $G$
      $G \leftarrow G \setminus \{u, v\}$
      $C \leftarrow C \cup \{u, v\}$
  return $C$
```

The minimum vertex cover—in fact, *every* vertex cover—contains at least one of the two vertices u and v chosen inside the while loop. It follows immediately that DUMBVERTEXCOVER is a 2-approximation algorithm!

The same idea can be extended to approximate the minimum hitting set for any set system $(X, \mathcal{F})$, where the approximation factor is the size of the largest set in $\mathcal{F}$.

I.6 Traveling Salesman: The Bad News

The *traveling salesman problem*[2] problem asks for the shortest Hamiltonian cycle in a weighted undirected graph. To keep the problem simple, we can assume without loss of generality that the underlying graph is always the complete graph K_n for some integer n; thus, the input to the traveling salesman problem is just a list of the $\binom{n}{2}$ edge lengths.

Not surprisingly, given its similarity to the Hamiltonian cycle problem, it's quite easy to prove that the traveling salesman problem is NP-hard. Let G be an arbitrary undirected graph with n vertices. We can construct a length function for K_n as follows:

$$\ell(e) = \begin{cases} 1 & \text{if } e \text{ is an edge in } G, \\ 2 & \text{otherwise.} \end{cases}$$

Now it should be obvious that if G has a Hamiltonian cycle, then there is a Hamiltonian cycle in K_n whose length is exactly n; otherwise every Hamiltonian cycle in K_n has length at least $n + 1$. We can clearly compute the lengths in polynomial time, so we have a polynomial time reduction from Hamiltonian cycle to traveling salesman. Thus, the traveling salesman problem is NP-hard, even if all the edge lengths are 1 and 2.

There's nothing special about the values 1 and 2 in this reduction; we can replace them with any values we like. By choosing values that are sufficiently far apart, we can show that even approximating the shortest traveling salesman tour is NP-hard. For example, suppose we set the length of the 'absent' edges to $n + 1$ instead of 2. Then the shortest traveling salesman tour in the resulting weighted graph either has length exactly n (if G has a Hamiltonian cycle) or has length at least $2n$ (if G does not have a Hamiltonian cycle). Thus, if we could approximate the shortest traveling salesman tour within a factor of 2 in polynomial time, we would have a polynomial-time algorithm for the Hamiltonian cycle problem.

Pushing this idea to its limits us the following negative result.

Theorem 4. *For any function $f(n)$ that can be computed in time polynomial in n, there is no polynomial-time $f(n)$-approximation algorithm for the traveling salesman problem on general weighted graphs, unless P=NP.*

I.7 Traveling Salesman: The Good News

Even though the general traveling salesman problem can't be approximated, a common special case can be approximated fairly easily. The special case requires the edge lengths to satisfy the so-called *triangle inequality*:

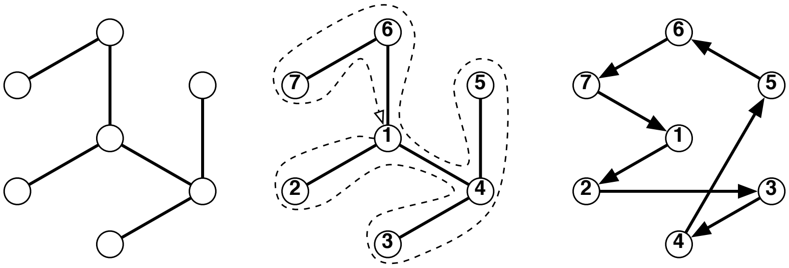
$$\ell(u, w) \le \ell(u, v) + \ell(v, w) \text{ for any vertices } u, v, w.$$

This inequality is satisfied for *geometric graphs*, where the vertices are points in the plane (or some higher-dimensional space), edges are straight line segments, and lengths are measured in the usual Euclidean metric. Notice that the length functions we used above to show that the general TSP is hard to approximate do not (always) satisfy the triangle inequality.

With the triangle inequality in place, we can quickly compute a 2-approximation for the traveling salesman tour as follows. First, we compute the minimum spanning tree T of the weighted input graph; this can be done in $O(n^2 \log n)$ time (where n is the number of vertices of the graph)

[2] This is sometimes bowdlerized into the traveling sales*person* problem. Sorry, no. Who ever heard of a traveling salesperson sleeping with the farmer's child?

using any of several classical algorithms. Second, we perform a depth-first traversal of T, numbering the vertices in the order that we first encounter them. Because T is a spanning tree, every vertex is numbered. Finally, we return the cycle obtained by visiting the vertices according to this numbering.



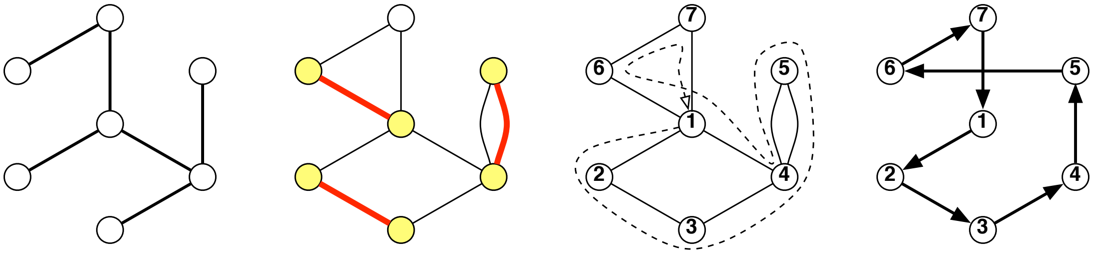
A minimum spanning tree T, a depth-first traversal of T, and the resulting approximate traveling salesman tour.

Let OPT denote the cost of the optimal TSP tour, let MST denote the total length of the minimum spanning tree, and let A be the length of the tour computed by our approximation algorithm. Consider the 'tour' obtained by walking through the minimum spanning tree in depth-first order. Since this tour traverses every edge in the tree exactly twice, its length is $2 \cdot MST$. The final tour can be obtained from this one by removing duplicate vertices, moving directly from each node to the next *unvisited* node.; the triangle inequality implies that taking these shortcuts cannot make the tour longer. Thus, $A \leq 2 \cdot MST$. On the other hand, if we remove any edge from the optimal tour, we obtain a spanning tree (in fact a spanning *path*) of the graph; thus, $MST \geq OPT$. We conclude that $A \leq 2 \cdot OPT$; our algorithm computes a 2-approximation of the optimal tour.

We can improve the approximation factor even further using the following algorithm discovered by Nicos Christofides in 1976. As in the previous algorithm, we start by constructing the minimum spanning tree T. Then let O be the set of vertices with *odd* degree in T; it is an easy exercise (hint, hint) to show that the number of vertices in O is even.

In the next stage of the algorithm, we compute a *minimum-cost perfect matching* M of these odd-degree vertices. A prefect matching is a collection of edges, where each edge has both endpoints in O and each vertex in O is adjacent to exactly one edge; we want the perfect matching of minimum total length. Later in the semester, we will see an algorithm to compute M in polynomial time.

Now consider the multigraph $T \cup M$; any edge in both T and M appears twice in this multigraph. This graph is connected, and every vertex has even degree. Thus, it contains an *Eulerian circuit*: a closed walk that uses every edge exactly once. We can compute such a walk in $O(n)$ time with a simple modification of depth-first search. To obtain the final approximate TSP tour, we number the vertices in the order they first appear in some Eulerian circuit of $T \cup M$, and return the cycle obtained by visiting the vertices according to that numbering.



A minimum spanning tree T, a minimum-cost perfect matching M of the odd vertices in T, an Eulerian circuit of $T \cup M$, and the resulting approximate traveling salesman tour.

Theorem 5. *Given a weighted graph that obeys the triangle inequality, the Christofides heuristic computes a (3/2)-approximation of the minimum traveling salesman tour.*

Proof: Let A denote the length of the tour computed by the Christofides heuristic; let OPT denote the length of the optimal tour; let MST denote the total length of the minimum spanning tree; let MOM denote the total length of the minimum odd-vertex matching.

The graph $T \cup M$, and therefore any Euler tour of $T \cup M$, has total length $MST + MOM$. By the triangle inequality, taking a shortcut past a previously visited vertex can only shorten the tour. Thus, $A \leq MST + MOM$.

By the triangle inequality, the optimal tour of the odd-degree vertices of T cannot be longer than OPT. Any cycle passing through of the odd vertices can be partitioned into two perfect matchings, by alternately coloring the edges of the cycle red and green. One of these two matchings has length at most $OPT/2$. On the other hand, both matchings have length at least MOM. Thus, $MOM \leq OPT/2$.

Finally, recall our earlier observation that $MST \leq OPT$.

Putting these three inequalities together, we conclude that $A \leq 3 \cdot OPT/2$, as claimed. □

> *The greatest flood has the soonest ebb; the sorest tempest the most sudden calm; the hottest love the coldest end; and from the deepest desire oftentimes ensues the deadliest hate.*
>
> — Socrates

> *Th' extremes of glory and of shame,*
> *Like east and west, become the same.*
>
> — Samuel Butler, *Hudibras* Part II, Canto I (c. 1670)

> *Extremes meet, and there is no better example*
> *than the haughtiness of humility.*
>
> — Ralph Waldo Emerson, "Greatness",
> in *Letters and Social Aims* (1876)

J Linear Programming

The maximum flow/minimum cut problem is a special case of a very general class of problems called *linear programming*. Many other optimization problems fall into this class, including minimum spanning trees and shortest paths, as well as several common problems in scheduling, logistics, and economics. Linear programming was used implicitly by Fourier in the early 1800s, but it was first formalized and applied to problems in economics in the 1930s by Leonid Kantorovich. Kantorivich's work was hidden behind the Iron Curtain (where it was largely ignored) and therefore unknown in the West. Linear programming was rediscovered and applied to shipping problems in the early 1940s by Tjalling Koopmans. The first complete algorithm to solve linear programming problems, called the *simplex method*, was published by George Dantzig in 1947. Koopmans first proposed the name "linear programming" in a discussion with Dantzig in 1948. Kantorovich and Koopmans shared the 1975 Nobel Prize in Economics "for their contributions to the theory of optimum allocation of resources". Dantzig did not; his work was apparently too pure. Koopmans wrote to Kantorovich suggesting that they refuse the prize in protest of Dantzig's exclusion, but Kantorovich saw the prize as a vindication of his use of mathematics in economics, which had been written off as "a means for apologists of capitalism".

A linear programming problem asks for a vector $x \in \mathbb{R}^d$ that maximizes (or equivalently, minimizes) a given linear function, among all vectors x that satisfy a given set of linear inequalities. The general form of a linear programming problem is the following:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1 \mathinner{..} p \\
& \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for each } i = p+1 \mathinner{..} p+q \\
& \sum_{j=1}^{d} a_{ij} x_j \ge b_i \quad \text{for each } i = p+q+1 \mathinner{..} n
\end{aligned}$$

Here, the input consists of a matrix $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, a column vector $b \in \mathbb{R}^n$, and a row vector $c \in \mathbb{R}^d$. Each coordinate of the vector x is called a *variable*. Each of the linear inequalities is called a *constraint*. The function $x \mapsto x \cdot b$ is called the *objective function*. I will always use d to denote the number of variables, also known as the *dimension* of the problem. The number of constraints is usually denoted n.

A linear programming problem is said to be in *canonical form*[1] if it has the following structure:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1 \mathinner{..} n \\ & x_j \ge 0 \quad \text{for each } j = 1 \mathinner{..} d \end{aligned}$$

We can express this canonical form more compactly as follows. For two vectors $x = (x_1, x_2, \ldots, x_d)$ and $y = (y_1, y_2, \ldots, y_d)$, the expression $x \ge y$ means that x_i and y_i for every index i.

$$\boxed{\begin{aligned} \max \quad & c \cdot x \\ \text{s.t.} \quad & Ax \le b \\ & x \ge 0 \end{aligned}}$$

Any linear programming problem can be converted into canonical form as follows:

- For each variable x_j, add the equality constraint $x_j = x_j^+ - x_j^-$ and the inequalities $x_j^+ \ge 0$ and $x_j^- \ge 0$.
- Replace any equality constraint $\sum_j a_{ij} x_j = b_i$ with two inequality constraints $\sum_j a_{ij} x_j \ge b_i$ and $\sum_j a_{ij} x_j \le b_i$.
- Replace any upper bound $\sum_j a_{ij} x_j \ge b_i$ with the equivalent lower bound $\sum_j -a_{ij} x_j \le -b_i$.

This conversion potentially triples the number of variables and doubles the number of constraints; fortunately, it is almost never necessary in practice.

Another convenient formulation, especially for describing the simplex algorithm, is *slack form*[2], in which the only inequalities are of the form $x_j \ge 0$:

$$\boxed{\begin{aligned} \max \quad & c \cdot x \\ \text{s.t.} \quad & Ax = b \\ & x \ge 0 \end{aligned}}$$

It's fairly easy to convert any linear programming problem into slack form. This form will be especially useful in describing the simplex algorithm.

J.1 The Geometry of Linear Programming

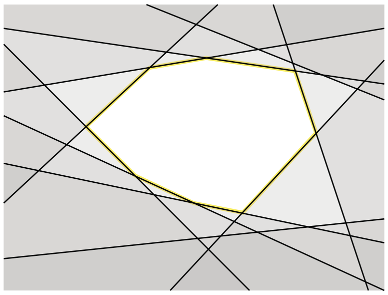
A point $x \in \mathbb{R}^d$ is *feasible* with respect to some linear programming problem if it satisfies all the linear constraints. The set of all feasible points is called the *feasible region* for that linear program.

[1] Confusingly, some authors call this *standard form*.

[2] Confusingly, some authors call this *standard form*.

The feasible region has a particularly nice geometric structure that lands some useful intuition to later linear programming algorithms.

Any linear equation in d variables defines a *hyperplane* in $\mathbb{R}^d$; think of a line when $d = 2$, or a plane when $d = 3$. This hyperplane divides $\mathbb{R}^d$ into two *halfspaces*; each halfspace is the set of points that satisfy some linear inequality. Thus, the set of feasible points is the intersection of several hyperplanes (one for each equality constraint) and halfspaces (one for each inequality constraint). The intersection of a finite number of hyperplanes and halfspaces is called a *polyhedron*. It's not hard to verify that any halfspace, and therefore any polyhedron, is *convex*—if a polyhedron contains two points x and y, then it contains the entire line segment $\overline{xy}$.



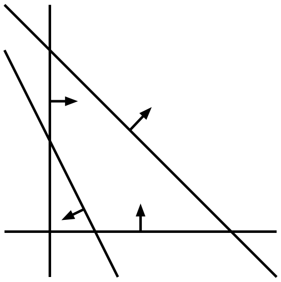
A two-dimensional polyhedron (white) defined by 10 linear inequalities.

By rotating $\mathbb{R}^d$ so that the objective function points downward, we can express *any* linear programming problem in the following geometric form:

$$\boxed{\text{Find the lowest point in a given polyhedron.}}$$

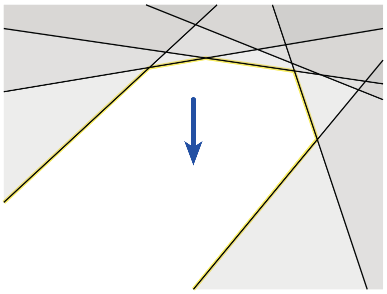
With this geometry in hand, we can easily picture two pathological cases where a given linear programming problem has no solution. The first possibility is that there are no feasible points; in this case the problem is called *infeasible*. For example, the following LP problem is infeasible:

$$\begin{aligned} \text{maximize } & x - y \\ \text{subject to } 2x + y &\le 1 \\ x + y &\ge 2 \\ x, y &\ge 0 \end{aligned}$$



An infeasible linear programming problem; arrows indicate the constraints.

The second possibility is that there are feasible points at which the objective function is arbitrarily large; in this case, we call the problem *unbounded*. The same polyhedron could be unbounded for some objective functions but not others, or it could be unbounded for every objective function.



A two-dimensional polyhedron (white) that is unbounded downward but bounded upward.

J.2 Example 1: Shortest Paths

We can compute the length of the shortest path from s to t in a weighted directed graph by solving the following very simple linear programming problem.

$$\begin{array}{lll} \text{maximize} & d_t & \\ \text{subject to} & d_s = 0 & \\ & d_v - d_u \le \ell_{u\to v} & \text{for every edge } u \to v \end{array}$$

Here, $w_{u\to v}$ is the length of the edge $u \to v$. Each variable d_v represents a tentative shortest-path distance from s to v. The constraints mirror the requirement that every edge in the graph must be relaxed. These relaxation constraints imply that in any feasible solution, d_v is *at most* the shortest path distance from s to v. Thus, somewhat counterintuitively, we are correctly *maximizing* the objective function to compute the *shortest* path! In the optimal solution, the objective function d_t is the actual shortest-path distance from s to t, but for any vertex v that is not on the shortest path from s to t, d_v may be an underestimate of the true distance from s to v. However, we can obtain the true distances from s to every other vertex by modifying the objective function:

$$\begin{array}{lll} \text{maximize} & \sum_v d_v & \\ \text{subject to} & d_s = 0 & \\ & d_v - d_u \le \ell_{u\to v} & \text{for every edge } u \to v \end{array}$$

There is another formulation of shortest paths as an LP minimization problem using indicator variables.

$$\begin{array}{lll} \text{minimize} & \sum_{u\to v} \ell_{u\to v} \cdot x_{u\to v} & \\ \text{subject to} & \sum_u x_{u\to s} - \sum_w x_{s\to w} = 1 & \\ & \sum_u x_{u\to t} - \sum_w x_{t\to w} = -1 & \\ & \sum_u x_{u\to v} - \sum_w x_{v\to w} = 0 & \text{for every vertex } v \ne s, t \\ & x_{u\to v} \ge 0 & \text{for every edge } u \to v \end{array}$$

Intuitively, $x_{u\to v}$ equals 1 if $u \to v$ is in the shortest path from s to t, and equals 0 otherwise. The constraints merely state that the path should start at s, end at t, and either pass through or avoid every other vertex v. Any path from s to t—in particular, the shortest path—clearly implies a feasible point for this linear program, but there are other feasible solutions with non-integral values that do not represent paths.

Nevertheless, there is an optimal solution in which every x_e is either 0 or 1 and the edges e with $x_e = 1$ comprise the shortest path. Moreover, in any optimal solution, the objective function gives the shortest path distance, even if not every x_e is an integer!

J.3 Example 2: Maximum Flows and Minimum Cuts

Recall that the input to the maximum (s,t)-flow problem consists of a weighted directed graph $G = (V, E)$, two special vertices s and t, and a function assigning a non-negative *capacity* c_e to each edge e. Our task is to choose the flow f_e across each edge e, as follows:

$$\begin{array}{lrl}
\text{maximize} & \displaystyle\sum_w f_{s\to w} - \sum_u f_{u\to s} & \\
\text{subject to} & \displaystyle\sum_w f_{v\to w} - \sum_u f_{u\to v} = 0 & \text{for every vertex } v \neq s,t \\
& f_{u\to v} \le c_{u\to v} & \text{for every edge } u\to v \\
& f_{u\to v} \ge 0 & \text{for every edge } u\to v
\end{array}$$

Similarly, the minimum cut problem can be formulated using 'indicator' variables similarly to the shortest path problem. We have a variable S_v for each vertex v, indicating whether $v \in S$ or $v \in T$, and a variable $X_{u\to v}$ for each edge $u \to v$, indicating whether $u \in S$ and $v \in T$, where (S,T) is some (s,t)-cut.[3]

$$\begin{array}{lrl}
\text{minimize} & \displaystyle\sum_{u\to v} c_{u\to v}\cdot X_{u\to v} & \\
\text{subject to} & X_{u\to v} + S_v - S_u \ge 0 & \text{for every edge } u\to v \\
& X_{u\to v} \ge 0 & \text{for every edge } u\to v \\
& S_s = 1 & \\
& S_t = 0 &
\end{array}$$

Like the minimization LP for shortest paths, there can be optimal solutions that assign fractional values to the variables. Nevertheless, the minimum value for the objective function is the cost of the minimum cut, and there is an optimal solution for which every variable is either 0 or 1, representing an actual minimum cut. No, this is not obvious; in particular, my claim is not a proof!

J.4 Linear Programming Duality

Each of these pairs of linear programming problems is related by a transformation called *duality*. For any linear programming problem, there is a corresponding dual linear program that can be obtained by a mechanical translation, essentially by swapping the constraints and the variables. The translation is simplest when the LP is in canonical form:

$$\begin{array}{ccc}
\textbf{Primal } (\Pi) & & \textbf{Dual } (\amalg) \\
\boxed{\begin{array}{rr} \max & c\cdot x \\ \text{s.t.} & Ax \le b \\ & x \ge 0 \end{array}} & \Longleftrightarrow & \boxed{\begin{array}{rr} \min & y\cdot b \\ \text{s.t.} & yA \ge c \\ & y \ge 0 \end{array}}
\end{array}$$

[3] These two linear programs are not quite *syntactic* duals; I've added two redundant variables S_s and S_t to the min-cut program increase readability.

We can also write the dual linear program in exactly the same canonical form as the primal, by swapping the coefficient vector c and the objective vector b, negating both vectors, and replacing the constraint matrix A with its negative transpose.[4]

$$\begin{array}{ccc} \textbf{Primal } (\Pi) & & \textbf{Dual } (\amalg) \\ \boxed{\begin{array}{rr} \max & c\cdot x \\ \text{s.t. } & Ax \le b \\ & x \ge 0 \end{array}} & \Longleftrightarrow & \boxed{\begin{array}{rr} \max & -b^\top \cdot y^\top \\ \text{s.t. } & -A^\top y^\top \le -c \\ & y^\top \ge 0 \end{array}} \end{array}$$

Written in this form, it should be immediately clear that duality is an *involution*: The dual of the dual linear program $\amalg$ is identical to the primal linear program Π. The choice of which LP to call the 'primal' and which to call the 'dual' is totally arbitrary.[5]

The Fundamental Theorem of Linear Programming. *A linear program* Π *has an optimal solution* x^* *if and only if the dual linear program* $\amalg$ *has an optimal solution* y^* *where* $c \cdot x^* = y^* A x^* = y^* \cdot b$.

The weak form of this theorem is trivial to prove.

Weak Duality Theorem. *If* x *is a feasible solution for a canonical linear program* Π *and* y *is a feasible solution for its dual* $\amalg$*, then* $c \cdot x \le yAx \le y \cdot b$.

Proof: Because x is feasible for Π, we have $Ax \le b$. Since y is positive, we can multiply both sides of the inequality to obtain $yAx \le y \cdot b$. Conversely, y is feasible for $\amalg$ and x is positive, so $yAx \ge c \cdot x$. □

It immediately follows that if $c \cdot x = y \cdot b$, then x and y are optimal solutions to their respective linear programs. This is in fact a fairly common way to prove that we have the optimal value for a linear program.

J.5 Duality Example

Before I prove the stronger duality theorem, let me first provide some intuition about where this duality thing comes from in the first place.[6] Consider the following linear programming problem:

$$\begin{array}{rrrrl} \text{maximize} & 4x_1 + & x_2 + & 3x_3 & \\ \text{subject to} & x_1 + & 4x_2 & & \le 1 \\ & 3x_1 - & x_2 + & x_3 & \le 3 \\ & & & x_1, x_2, x_3 & \ge 0 \end{array}$$

Let σ^* denote the optimum objective value for this LP. The feasible solution $x = (1, 0, 0)$ gives us a lower bound $\sigma^* \ge 4$. A different feasible solution $x = (0, 0, 3)$ gives us a better lower bound $\sigma^* \ge 9$.

[4] For the notational purists: In these formulations, x and b are column vectors, and y and c are row vectors. This is a somewhat nonstandard choice. Yes, that means the dot in $c \cdot x$ is redundant. Sue me.

[5] For historical reasons, maximization LPs tend to be called 'primal' and minimization LPs tend to be called 'dual', which is really really stupid, since the only difference is a sign change.

[6] This example is taken from Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions* [Springer, 2001], but the idea appears earlier in Jens Clausen's 1997 paper 'Teaching Duality in Linear Programming: The Multiplier Approach'.

We could play this game all day, finding different feasible solutions and getting ever larger lower bounds. How do we know when we're done? Is there a way to prove an *upper* bound on σ^*?

In fact, there is. Let's multiply each of the constraints in our LP by a new non-negative scalar value y_i:

$$\begin{array}{rl} \text{maximize} & 4x_1 + x_2 + 3x_3 \\ \text{subject to} & y_1(x_1 + 4x_2 \phantom{{}+ x_3}) \le y_1 \\ & y_2(3x_1 - x_2 + x_3) \le 3y_2 \\ & x_1, x_2, x_3 \ge 0 \end{array}$$

Because each y_i is non-negative, we do not reverse any of the inequalities. Any feasible solution (x_1, x_2, x_3) must satisfy both of these inequalities, so it must also satisfy their sum:

$$(y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \le y_1 + 3y_2.$$

Now suppose that each y_i is larger than the ith coefficient of the objective function:

$$y_1 + 3y_2 \ge 4, \qquad 4y_1 - y_2 \ge 1, \qquad y_2 \ge 3.$$

This assumption lets us derive an upper bound on the objective value of *any* feasible solution:

$$4x_1 + x_2 + 3x_3 \le (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \le y_1 + 3y_2.$$

We have just proved that $\sigma^* \le y_1 + 3y_2$.

Now it's natural to ask how tight we can make this upper bound. How small can we make the expression $y_1 + 3y_2$ without violating any of the inequalities we used to prove the upper bound? This is just another linear programming problem!

$$\begin{array}{rr} \text{minimize} & y_1 + 3y_2 \\ \text{subject to} & y_1 + 3y_2 \ge 4 \\ & 4y_1 - y_2 \ge 1 \\ & y_2 \ge 3 \\ & y_1, y_2 \ge 0 \end{array}$$

This is precisely the dual of our original linear program!

J.6 Strong Duality

The Fundamental Theorem can be rephrased in the following form:

Strong Duality Theorem. *If x^* is an optimal solution for a canonical linear program Π, then there is an optimal solution y^* for its dual $\amalg$, such that $c \cdot x^* = y^*Ax^* = y^* \cdot b$.*

Proof (Sketch): I'll prove the theorem only for *non-degenerate* linear programs, in which (a) the optimal solution (if one exists) is a unique vertex of the feasible region, and (b) at most d constraint planes pass through any point. These non-degeneracy assumptions are relatively easy to enforce in practice and can be removed from the proof at the expense of some technical detail. I will also prove the theorem only for the case $n \ge d$; the argument for under-constrained LPs is similar (if not simpler).

Let x^* be the optimal solution for the linear program Π; non-degeneracy implies that this solution is unique, and that exactly d of the n linear constraints are satisfied with equality. Without loss of generality (by permuting the rows of A), we can assume that these are the first d constraints.

So let $A_\bullet$ be the $d \times d$ matrix containing the first d rows of A, and let $A_\circ$ denote the other $n-d$ rows. Similarly, partition b into its first d coordinates $b_\bullet$ and everything else $b_\circ$. Thus, we have partitioned the inequality $Ax^* \leq b$ into a system of equations $A_\bullet x^* = b_\bullet$ and a system of strict inequalities $A_\circ x^* < b_\circ$.

Now let $y^* = (y^*_\bullet, y^*_\circ)$ where $y^*_\bullet = cA_\bullet^{-1}$ and $y^*_\circ = 0$. We easily verify that $y^* \cdot b = c \cdot x^*$:

$$y^* \cdot b = y^*_\bullet \cdot b_\bullet = (cA_\bullet^{-1})b_\bullet = c(A_\bullet^{-1}b_\bullet) = c \cdot x^*.$$

Similarly, it's trivial to verify that $y^* A \geq c$:

$$y^* A = y^*_\bullet A^*_\bullet = c.$$

Once we prove that y^* is non-negative, and therefore feasible, the Weak Duality Theorem implies the result. Clearly $y^*_\circ \geq 0$. As we will see below, the inequality $y^*_\bullet \geq 0$ follows from the fact that x^* is optimal—we had to use that fact somewhere! This is the hardest part of the proof.

The key insight is to give a geometric interpretation to the vector $y^*_\bullet = cA_\bullet^{-1}$. Each row of the linear system $A_\bullet x^* = b_\bullet$ describes a hyperplane $a_i \cdot c^* = b_i$ in $\mathbb{R}^d$. The vector a_i is normal to this hyperplane and points *out* of the feasible region. The vectors $a_1, \ldots, a_d$ are linearly independent (by non-degeneracy) and thus describe a coordinate frame for the vector space $\mathbb{R}^d$. The definition of $y^*_\bullet$ can be rewritten as follows:

$$c = y^*_\bullet A_\bullet = \sum_{i=1}^{d} y^*_i a_i.$$

We are expressing the objective vector c as a linear combination of the constraint normals $a_1, \ldots, a_d$.

Now consider any vertex z of the feasible region that is adjacent to x^*. The vector $z - x^*$ is normal to all but one of the vectors a_i. Thus, we have

$$A_\bullet(z - x^*) = (0, \ldots, \nu, \ldots, 0)^\top$$

where the constant ν is in the ith coordinate. The vector $z - x^*$ points *into* the feasible region, so $\nu \leq 0$. It follows that

$$c \cdot (z - x^*) = y^*_\bullet A_\bullet(z - x^*) = \nu y^*_i.$$

The optimality of x^* implies that $c \cdot x^* \geq c \cdot z$, so we must have $y^*_i \geq 0$. We're done! □

> *When a jar is broken, the space that was inside*
> *Merges into the space outside.*
> *In the same way, my mind has merged in God;*
> *To me, there appears no duality.*
>
> — Avadhuta Gita

K Linear Programming Algorithms

In this lecture, we'll see a few algorithms for actually solving linear programming problems. The most famous of these, the *simplex method*, was proposed by George Dantzig in 1947. Although most variants of the simplex algorithm performs well in practice, there is no sub-exponential upper bound on the running time of any simplex variant. However, if the dimension of the problem is considered a constant, there are several linear programming algorithms that run in *linear* time. I'll describe a particularly simple randomized algorithm due to Raimund Seidel.

My approach to describing these algorithms will rely much more heavily on geometric intuition than the usual linear-algebraic formalism. This works better for me, but your mileage may vary. For a more traditional description of the simplex algorithm, see Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions* [Springer, 2001], which can be freely downloaded (but not legally printed) from the author's website.

K.1 Bases, Feasibility, and Local Optimality

Consider the linear program $\max\{c \cdot x \mid Ax \geq b, x \geq 0\}$, where A is an $n \times d$ constraint matrix, b is an n-dimensional coefficient vector, and c is a d-dimensional objective vector. We will interpret this linear program geometrically as looking for the lowest point in a convex polyhedron in $\mathbb{R}^d$, described as the intersection of $n + d$ halfspaces. As in the last lecture, we will consider only *non-degenerate* linear programs: At most d constraint hyperplanes pass through any point, and no constraint hyperplane is normal to the objective vector.

A *basis* is a subset of d constraints, which by our non-degeneracy assumption must be linearly independent. The *location* of a basis is the unique point x that satisfies all d constraints with equality; geometrically, x is the unique intersection point of the d hyperplanes. The *value* of a basis is $c \cdot x$, where x is the location of the basis. There are precisely $\binom{n+d}{d}$ bases. Geometrically, the set of constraint hyperplanes defines a decomposition of $\mathbb{R}^d$ into convex polyhedra; this cell decomposition is called the *arrangement* of the hyperplanes. Every d-tuple of hyperplanes (*i.e.*, every basis) defines a *vertex* of this arrangement (the location of the basis). I will use the words 'vertex' and 'basis' interchangeably.

A basis is *feasible* if its location x satisfies all the linear constraints, or geometrically, if the point x is a vertex of the polyhedron.

A basis is *locally optimal* if its location x is the optimal solution to the linear program with the same objective function and *only* the constraints in the basis. Geometrically, a basis is locally optimal if its location x is the lowest point in the intersection of those d halfspaces. A careful reading of the proof of the Strong Duality Theorem reveals that local optimality is the dual equivalent of feasibility; a basis is locally feasible for a linear program Π if and only if the same basis is feasible for the dual linear program $\amalg$. For this reason, locally optimal bases are sometimes also called *dual feasible*.

Every $(d-1)$-tuple of hyperplanes defines a line in $\mathbb{R}^d$ that is broken into *edges* by the other hyperplanes. Most of these edges are line segments joining two vertices, but some are infinite rays, which we think of as segments joined to an artificial vertex called ∞. This collection of vertices and edges is called the *1-skeleton* of the *graph* of the hyperplane arrangement. The graph of vertices and edges on the boundary of the feasible region is a subgraph of the arrangement graph.

Two bases are said to be *adjacent* if they have $d-1$ constraints in common; geometrically, two vertices are adjacent if they are joined by an edge in the arrangement graph. A *pivot* changes a basis B into some basis adjacent to B, or equivalently, moves a point from one vertex in the arrangement to an adjacent vertex.

The Weak Duality Theorem implies that the value of every feasible basis is less than or equal to the value of every locally optimal basis; every feasible vertex is higher than every locally optimal vertex. The Strong Duality Theorem implies that (under our non-degeneracy assumption), if a linear program has an optimal solution, it is the *unique* vertex that is *both* feasible and locally optimal. Moreover, the optimal solution is both the lowest feasible vertex and the highest locally optimal vertex.

K.2 The Simplex Algorithm: Primal View

From a geometric standpoint, Dantzig's simplex algorithm is very simple. The input is a set of halfspaces H; we want the lowest vertex in the intersection of these halfspaces.

```
SIMPLEX($H$):
  if $\cap H = \varnothing$
      return INFEASIBLE
  $x \leftarrow$ any feasible vertex
  while $x$ is not locally optimal
      ⟨⟨pivot downward, maintaining feasibility⟩⟩
      $x \leftarrow$ any feasible neighbor of $x$ that is lower than $x$
      if $x = \infty$
          return UNBOUNDED
  return $x$.
```

Let's ignore the first three lines for the moment. The algorithm maintains a feasible vertex x. At each pivot operation, it moves to a *lower* vertex, so the algorithm never visits the same vertex more than once. Thus, after at most $\binom{n+d}{d}$ pivots, the algorithm either finds the optimal solution, or it discovers that the polyhedron is unbounded.

Notice that we have left open the method for choosing *which* neighbor to choose at each pivot. There are several natural pivoting rules, but for most rules, there are input polyhedra that require an exponential number of pivots. *No* pivoting rule is known that guarantees a polynomial number of pivots in the worst case.

K.3 The Simplex Algorithm: Dual View

We can also geometrically interpret the execution of the simplex algorithm on the dual linear program ∐. Algebraically, there is no difference between these two algorithms; the only change is in how we interpret the linear algebra geometrically. Again, the input is a set of halfspaces H, and we want the lowest vertex in the intersection of these halfspaces. By the Strong Duality Theorem, this is the same as the highest locally-optimal vertex in the hyperplane arrangement.

```
SIMPLEX($H$):
  if there is no locally optimal vertex
      return UNBOUNDED
  $x \leftarrow$ any locally optimal vertex
  while $x$ is not feasbile
      ⟨⟨pivot upward, maintaining local optimality⟩⟩
      $x \leftarrow$ any locally-optimal neighbor of $x$ that is higher than $x$
      if $x = \infty$
          return INFEASIBLE
  return $x$.
```

Let's ignore the first three lines for the moment. The algorithm maintains a locally optimal vertex x. At each pivot operation, it moves to a *higher* vertex, so the algorithm never visits the same vertex more than once. Thus, after at most $\binom{n+d}{d}$ pivots, the algorithm either finds the optimal solution, or it discovers that the linear program is infeasible.

K.4 Computing the Initial Basis

To complete our description of the simplex algorithm, we need to describe how to find the initial vertex x. Our algorithm relies on the following simple observations.

First, the feasibility of a vertex does not depend at all on the choice of objective vector; a vertex is either feasible for every objective function or for none. No matter how we rotate the polyhedron, every feasible vertex stays feasible. Conversely (or by duality, equivalently), the local optimality of a vertex does not depend on the exact location of the d hyperplanes, but only on their normal directions and the objective function. No matter how we translate the hyperplanes, every locally optimal vertex stays locally optimal. In terms of the original matrix formulation, feasibility depends on A and b but not c, and local optimality depends on A and c but not b.

The second important observation is that *every* basis is locally optimal for *some* objective function. Specifically, it suffices to choose any vector that has a positive inner product with each of the normal vectors of the d chosen hyperplanes. Equivalently, we can make *any* basis feasible by translating the hyperplanes appropriately. Specifically, it suffices to translate the chosen d hyperplanes so that they pass through the origin, and then translate all the other halfspaces so that they contain the origin.

Our strategy for finding our initial feasible vertex is to choose *any* vertex, choose a new objective function that makes that vertex locally optimal, and then find the optimal vertex for *that* objective function by running the (dual) simplex algorithm. This vertex must be feasible, even after we restore the original objective function! Equivalently, to find an initial locally optimal vertex, we choose *any* vertex, translate the hyperplanes so that that vertex becomes feasible, and then find the optimal vertex for those translated constraints. This vertex must be locally optimal, even after we restore the hyperplanes to their original locations!

Here are more complete descriptions of the simplex algorithm with this initialization rule, in both primal and dual forms. *Argh*(H) denotes the hyperplane arrangement induced by the halfspaces H.

```
SIMPLEX($H$):
    $x \leftarrow$ any vertex
    $\tilde{H} \leftarrow$ rotation of $H$ that makes $x$ locally optimal

    while $x$ is not feasible
        ⟨⟨pivot upward maintaining local optimality⟩⟩
        $x \leftarrow$ any locally optimal neighbor of $x$ in Argh$(\tilde{H})$ that is higher than $x$
        if $x = \infty$
            return INFEASIBLE

    while $x$ is not locally optimal
        ⟨⟨pivot downward, maintaining feasibility⟩⟩
        $x \leftarrow$ any feasible neighbor of $x$ in Argh$(H)$ that is lower than $x$
        if $x = \infty$
            return UNBOUNDED
    return $x$.
```

```
SIMPLEX($H$):
    $x \leftarrow$ any vertex
    $\tilde{H} \leftarrow$ translation of $H$ that makes $x$ feasible

    while $x$ is not locally optimal
        ⟨⟨pivot downward, maintaining feasibility⟩⟩
        $x \leftarrow$ any feasible neighbor of $x$ in Argh$(\tilde{H})$ that is lower than $x$
        if $x = \infty$
            return UNBOUNDED

    while $x$ is not feasible
        ⟨⟨pivot upward maintaining local optimality⟩⟩
        $x \leftarrow$ any locally optimal neighbor of $x$ in Argh$(H)$ that is higher than $x$
        if $x = \infty$
            return INFEASIBLE
    return $x$
```

K.5 Linear Expected Time for Fixed Dimensions

In most geometric applications of linear programming, the number of variables is a small constant, but the number of constraints may still be very large.

The input to the following algorithm is a set H of n halfspaces and a set B of b hyperplanes. (B stands for *basis*.) The algorithm returns the lowest point in the intersection of the halfspaces in H and the hyperplanes B. At the top level of recursion, B is empty. I implicitly assume here that the linear program is bounded; if necessary, we can guarantee boundedness by adding a single halfspace to H. A point x *violates* a constraint h if it is not contained in the corresponding halfspace.

```
SEIDELLP($H, B$) :
    if $|B| = d$
        $x \leftarrow \bigcap B$
        if $x$ violates any constraint in $H$
            return INFEASIBLE
        else
            return $x$
    if $|H \cup B| = d$
        return $\bigcap(H \cup B)$
    $h \leftarrow$ random element of $H$
    $x \leftarrow$ SEIDELLP($H \setminus h, B$)        $(*)$
    if $x =$ INFEASIBLE
        return $x$
    else if $x$ violates $h$
        return SEIDELLP($H \setminus h, B \cup \partial h$)
    else
        return $x$
```

The point x recursively computed in line $(*)$ is not the optimal solution precisely when the random halfspace h is one of the d halfspaces that define the optimal solution. In other words, the probability of calling SEIDELLP$(H, B \cup h)$ is exactly $(d - b)/n$. Thus, we have the following recurrence for the expected number of recursive calls for this algorithm:

$$T(n, b) = \begin{cases} 1 & \text{if } b = d \text{ or } n + b = d \\ T(n-1, b) + \dfrac{d-b}{n} \cdot T(n-1, b+1) & \text{otherwise} \end{cases}$$

The recurrence is somewhat simpler if we let $\delta = d - b$:

$$T(n, \delta) = \begin{cases} 1 & \text{if } \delta = 0 \text{ or } n = \delta \\ T(n-1, \delta) + \dfrac{\delta}{n} \cdot T(n-1, \delta-1) & \text{otherwise} \end{cases}$$

It's easy to prove by induction that $T(n, \delta) = O(\delta!\, n)$:

$$\begin{aligned} T(n, \delta) &= T(n-1, \delta) + \frac{\delta}{n} \cdot T(n-1, \delta-1) \\ &\le \delta!\,(n-1) + \frac{\delta}{n}(\delta-1)! \cdot (n-1) && \text{[ind. hyp.]} \\ &= \delta!\,(n-1) + \delta! \frac{n-1}{n} \\ &\le \delta!\, n \end{aligned}$$

At the top level of recursion, we perform one violation test in $O(d)$ time. In each of the base cases, we spend $O(d^3)$ time computing the intersection point of d hyperplanes, and in the first base case, we spend $O(dn)$ additional time testing for violations. More careful analysis implies that the algorithm runs in $\boxed{O(d! \cdot n) \text{ expected time}}$.