

Algorithms

Department of Computer Science
University of Illinois at Urbana-Champaign

Instructor: Jeff Erickson

Teaching Assistants:

- **Spring 1999:** Mitch Harris and Shripad Thite
- **Summer 1999 (IMCS):** Mitch Harris
- **Summer 2000 (IMCS):** Mitch Harris
- **Fall 2000:** Chris Neihengen, Ekta Manaktala, and Nick Hurlburt
- **Spring 2001:** Brian Ensink, Chris Neihengen, and Nick Hurlburt
- **Summer 2001 (I2CS):** Asha Seetharam and Dan Bullok
- **Fall 2002:** Erin Wolf, Gio Kao, Kevin Small, Michael Bond, Rishi Talreja, Rob McCann, and Yasutaka Furakawa
- **Spring 2004:** Dan Cranston, Johnathon Fischer, Kevin Milans, and Lan Chen
- **Fall 2005:** Erin Chambers, Igor Gammer, and Aditya Ramani
- **Fall 2006:** Dan Cranston, Nitish Korula, and Kevin Milans
- **Spring 2007:** Kevin Milans
- **Fall 2008:** Reza Zamani-Nasab
- **Spring 2009:** Alina Ene, Ben Moseley, and Amir Nayyeri

© Copyright 1999–2009 Jeff Erickson. Last update August 16, 2009.

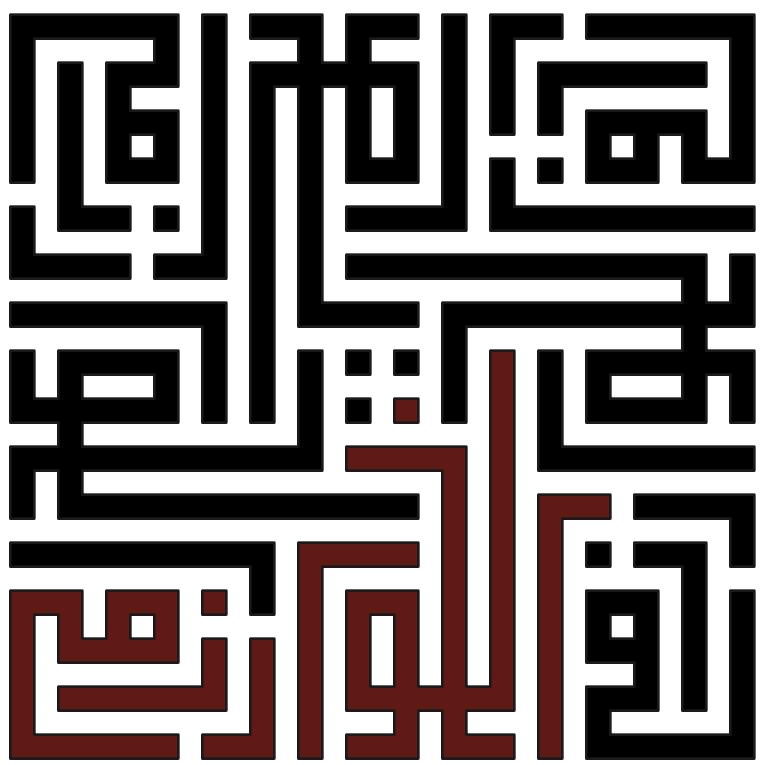
This work may be freely copied and distributed, either electronically or on paper.

It may not be sold for more than the actual cost of reproduction, storage, or transmittal.

This work is licensed under a Creative Commons Attribution-NonCommercial-Share Alike 3.0 United States License.

For license details, see <http://creativecommons.org/licenses/by-nc-sa/3.0/us/>.

For the most recent edition of this work, see <http://www.uiuc.edu/~jeffe/teaching/algorithms/>.



*Shall I tell you, my friend, how you will come to understand it?
Go and write a book on it.*

— Henry Home, Lord Kames (1696–1782), to Sir Gilbert Elliot

*You know, I could write a book.
And this book would be thick enough to stun an ox.*

— Laurie Anderson, “Let X=X”, *Big Science* (1982)

*I’m writing a book. I’ve got the page numbers done,
so now I just have to fill in the rest.*

— Stephen Wright

About These Notes

This course packet includes lecture notes, homework questions, and exam questions from algorithms courses I taught at the University of Illinois at Urbana-Champaign in Spring 1999, Fall 2000, Spring 2001, Fall 2002, Spring 2004, Fall 2005, Fall 2006, Spring 2007, Fall 2008, and Spring 2009. These lecture notes and my videotaped lectures were also offered over the web in Summer 1999, Summer 2000, Summer 2001, Fall 2002, and Fall 2005 as part of the UIUC computer science department’s online master’s program. Lecture notes were posted to the course web site a few days (on average) after each lecture. Homeworks, exams, and solutions were also distributed over the web.

I wrote a significant fraction of these lecture notes in Spring 1999; I revise them and add a few new notes every time I teach the course. The recurrences ‘pre-lecture’ is very loosely based on a handout written by Ari Trachtenberg, based on a paper by George Lueker, from an earlier semester (Fall 1998?) taught by Ed Reingold, but I have essentially rewritten it from scratch.

Most (but not all) of the exercises at the end of each lecture note have been used at least once in a homework assignment, discussion section, or exam. You can also find a near-complete collection of homeworks and exams from past semesters of my class online at <http://www.uiuc.edu/~jeffe/teaching/algorithms/>. A large fraction of these exercises were contributed by some amazing teaching assistants:

Aditya Ramani, Alina Ene, Amir Nayyeri, Asha Seetharam, Ben Moseley, Brian Ensink, Chris Neihengen, Dan Bullok, Dan Cranston, Johnathon Fischer, Ekta Manaktala, Erin Wolf Chambers, Igor Gammer, Gio Kao, Kevin Milans, Kevin Small, Lan Chen, Michael Bond, Mitch Harris, Nick Hurlburt, Nitish Korula, Reza Zamani-Nasab, Rishi Talreja, Rob McCann, Shripad Thite, and Yasu Furakawa.

*Stars indicate more challenging problems; many of these appeared on qualifying exams for the algorithms PhD students at UIUC. A small number of *really* hard problems are marked with a ★ larger star; one or two *open* problems are indicated by ★enormous stars.

Please do not ask me for solutions to the exercises. If you’re a student, seeing the solution will rob you of the experience of solving the problem yourself, which is the only way to learn the material. If you’re an instructor, you shouldn’t assign problems that you can’t solve yourself! (I do not always follow my own advice; some of these problems have serious bugs.)

Prerequisites

For the most part, these notes assume that the reader has mastered the material covered in the first two years of a typical undergraduate computer science curriculum. (Mastery is not the same thing as ‘exposure’ or ‘a good grade’; this is why I start every semester with Homework Zero.) Specific prerequisites include:

- Proof techniques: direct proof, indirect proof, proof by contradiction, combinatorial proof, and induction (including its “strong”, “structural”, and “recursive” forms). Lecture 0 requires induction, and whenever Lecture $n - 1$ requires induction, so does Lecture n .
- Discrete mathematics: Boolean algebra, predicate logic, sets, functions, relations, recursive definitions, trees (as abstract objects, not just data structures), graphs
- Elementary discrete probability: expectation, linearity of expectation, independence
- Iterative programming concepts: variables, conditionals, iteration, subroutines, indirection (addresses/pointers/references), recursion. Programming experience in any language that supports pointers and recursion is a plus.
- Fundamental data structures: arrays/vectors, linked lists, search trees, heaps
- Fundamental abstract data types: dictionaries, stacks, queues, priority queues; the difference between this list and the previous list.
- Fundamental algorithms: searching, sorting (selection, insertion, merge, heap, quick, anything but bubble), pre-/post-/inorder tree traversal, depth/breadth first search
- Basic algorithm analysis: Asymptotic notation (o , O , Θ , Ω , ω), translating loops into sums and recursive calls into recurrences, evaluating simple sums and recurrences.
- Mathematical maturity: facility with abstraction, formal (especially recursive) definitions, and (especially inductive) proofs; following mathematical arguments; recognizing syntactic, semantic, and/or logical nonsense; writing the former rather than the latter

Some of this material is covered briefly, but more as a reminder than a good introduction. In the future, I hope to write review notes that cover more of these topics. Alas, the future has not yet arrived.

Acknowledgments

The lecture notes and exercises draw heavily on the creativity, wisdom, and experience of thousands of algorithms students, teachers, and researchers. In particular, I am immensely grateful to the almost 1200 Illinois students who have used these notes as a primary reference, offered useful (if sometimes painful) criticism, and suffered through some truly awful first drafts. I’m also grateful for the contributions and feedback from teaching assistants, all listed above.

Naturally, these notes owe a great deal to the people who taught me this algorithms stuff in the first place: Bob Bixby and Michael Perlman at Rice; David Eppstein, Dan Hirshberg, and George Lueker at UC Irvine; and Abhiram Ranade, Dick Karp, Manuel Blum, Mike Luby, and Raimund Seidel at UC Berkeley. I’ve also been helped tremendously by many discussions with faculty colleagues at UIUC—Chandra Chekuri, Edgar Ramos, Herbert Edelsbrunner, Jason Zych, Lenny Pitt, Mahesh Viswanathan, Margaret Fleck, Shang-Hua Teng, Steve LaValle, and especially Ed Reingold and Sariel Har-Peled. I stole the first iteration of the overall course structure, and the idea to write up my own lecture notes, from Herbert Edelsbrunner.

“Johnny’s” multi-colored crayon homework was found under the TA office door among the other Fall 2000 Homework 1 submissions. The square Kufi rendition of the name “al-Khwārizmī” on the back of the cover page is mine.

The following sources have been invaluable references. (This list is incomplete!)

- Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (I used this textbook as an undergrad at Rice, and again as a masters student at UC Irvine.)
- Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 2000.
- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.
- Thomas Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein. *Introduction to Algorithms*, second edition. MIT Press/McGraw-Hill, 2000. (This was my recommended textbook until 2005. I also used the first edition as a teaching assistant at Berkeley.)
- Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006. (This is the current recommended textbook for the undergraduate section of my class.)
- Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.
- Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Molecular Biology*. Cambridge University Press, 1997.
- Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005. (This is the current recommended textbook for the graduate section of my class.)
- Donald Knuth. *The Art of Computer Programming*, volumes 1–3. Addison-Wesley, 1997. (My parents gave me these for Christmas when I was 14. I actually read it *much* later.)
- Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989. (I used this textbook as a teaching assistant at Berkeley.)
- Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- Ian Parberry. *Problems on Algorithms*. Prentice-Hall, 1995. (This book is out of print, but it can be downloaded for ‘free’ from <http://www.eng.unt.edu/ian/books/free/license.html>.)
- Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.
- Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988. (This book and its sequels have by far the best algorithm *illustrations* I’ve seen anywhere.)
- Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
- Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 2001.
- Class notes from my own algorithms classes at Berkeley, especially those taught by Dick Karp and Raimund Seidel.
- The Source of All Knowledge (Google) and The Source of All Lies (Wikipedia).

Caveat Lector!

With few exceptions, each of these notes contains far too much material to cover in a single lecture. In a typical 75-minute lecture, I tend to cover 4 to 5 pages of material—a bit more if I'm lecturing to graduate students than to undergraduates. Your mileage may vary! (Arguably, that means that as I continue to add material, the label “lecture notes” becomes less and less accurate.)

Despite several rounds of revision, these notes still contain lots of mistakes, errors, bugs, gaffes, omissions, snafus, kludges, typos, mathos, grammaros, thinkos, brain farts, nonsense, garbage, cruft, junk, and outright lies, *all of which are entirely Steve Skiena's fault*. I revise and update these notes every time I teach the course, so please let me know if you find a bug. (Steve is unlikely to care.)

Whenever I teach the algorithms class, I award extra credit points to the first student to post an explanation and correction of any error in the lecture notes to the course newsgroup. Obviously, the number of extra credit points depends on the severity of the error and the quality of the correction. If I'm not teaching the course, encourage your instructor to set up a similar extra-credit scheme, and forward the bug reports to **Steve** me!

Of course, any other feedback is also welcome!

Enjoy!

— *Jeff*

It is traditional for the author to magnanimously accept the blame for whatever deficiencies remain. I don't. Any errors, deficiencies, or problems in this book are somebody else's fault, but I would appreciate knowing about them so as to determine who is to blame.

— Steven S. Skiena, *The Algorithm Design Manual* (1997)

We should explain, before proceeding, that it is not our object to consider this program with reference to the actual arrangement of the data on the Variables of the engine, but simply as an abstract question of the nature and number of the operations required to be performed during its complete solution.

— Ada Augusta Byron King, Countess of Lovelace, translator's notes for Luigi F. Menabrea,
"Sketch of the Analytical Engine invented by Charles Babbage, Esq." (1843)

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)

*The moment a man begins to talk about technique
that's proof that he is fresh out of ideas.*

— Raymond Chandler

0 Introduction

0.1 What is an algorithm?

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions. 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 algorithmophobic classicists might guess, from the Greek root *algos* ($\alpha\lambda\gammaος$), meaning ‘pain’. Rather, it is a corruption of the name of the 9th century Persian mathematician Abū 'Abd Allāh Muḥammad ibn Mūsā al-Khwārizmī.¹ Al-Khwārizmī is perhaps best known as the writer of the treatise *Al-Kitāb al-mukhtaṣar fīhīsāb al-ğabr wa'l-muqābala*², 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* ($\alpha\rhoι\thetaμος$), meaning ‘number’, and perhaps the aforementioned $\alpha\lambda\gammaος$), 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-Khwārizmī 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 calculation³ 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.) Thus, until very recently, the word *algorithm* referred exclusively to pencil-and-paper

¹Mohammad, father of Adbdulla, son of Moses, the Kwārizmian'. Kwārizm is an ancient city, now called Khiva, in the Khorezm Province of Uzbekistan.

²The Compendious Book on Calculation by Completion and Balancing'.

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

methods for numerical calculations. People trained in the reliable execution of these methods were called—you guessed it—*computers*.

0.2 A Few Simple Examples

Multiplication by compass and straightedge. Although they have only been an object of formal study for a few decades, algorithms have been with us since the dawn of civilization, for centuries before Al-Khwārizmī 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, $\text{CIRCLE}(p, q)$ 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 ℓ and passing through P .

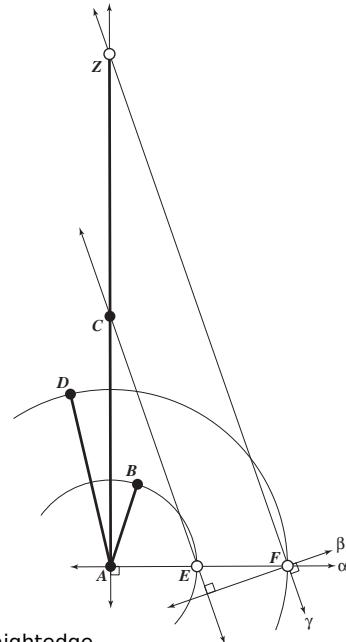
RIGHTANGLE(ℓ, P):

Choose a point $A \in \ell$
 $A, B \leftarrow \text{INTERSECT}(\text{CIRCLE}(P, A), \ell)$
 $C, D \leftarrow \text{INTERSECT}(\text{CIRCLE}(A, B), \text{CIRCLE}(B, A))$
return LINE(C, D)

Construct a point Z such that $|AZ| = |AC||AD|/|AB|$.

MULTIPLYORDIVIDE(A, B, C, D):

$\alpha \leftarrow \text{RIGHTANGLE}(\text{LINE}(A, C), A)$
 $E \leftarrow \text{INTERSECT}(\text{CIRCLE}(A, B), \alpha)$
 $F \leftarrow \text{INTERSECT}(\text{CIRCLE}(A, D), \alpha)$
 $\beta \leftarrow \text{RIGHTANGLE}(\text{LINE}(E, C), F)$
 $\gamma \leftarrow \text{RIGHTANGLE}(\beta, F)$
return INTERSECT($\gamma, \text{LINE}(A, C)$)



Multiplying or dividing using a compass and straightedge.

This algorithm breaks down the difficult task of multiplication into a series of simple primitive operations: drawing a line between two points, drawing a circle with a given center and boundary point, and so on. These primitive steps are quite non-trivial to execute on a modern digital computer, but this algorithm wasn't designed for a digital computer; it was designed for the Platonic Ideal Classical Greek Mathematician, wielding the Platonic Ideal Compass and the Platonic Ideal Straightedge. In this example, Euclid first defines a new primitive operation, constructing a right angle, 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; it was still being used in Russia, along with the Julian

⁴Euclid 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*.

calendar, well into the 20th century. This algorithm was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

<u>PEASANTMULTIPLY(x, y):</u>	x	y	$prod$
$prod \leftarrow 0$			0
while $x > 0$			
if x is odd			
$prod \leftarrow prod + y$			
$x \leftarrow \lfloor x/2 \rfloor$			
$y \leftarrow y + y$			
return p			
	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) duplication (doubling a number), and (4) mediation (halving a number, rounding down).⁵ Of course a full specification of this algorithm requires describing how to perform those four ‘primitive’ operations. Peasant multiplication requires (a constant factor!) more paperwork to execute by hand, but the necessary operations are easier (for humans) to remember than the 10×10 multiplication table required by the American grade school algorithm.⁶

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”:⁷

<u>BECOMEAMILLIONAIREANDNEVERPAYTAXES:</u>
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, 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.

⁵The version of this algorithm actually used in ancient Egypt 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 then found by greedy subtraction, and the corresponding entries in the other table are added together to form the product.

⁶American 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.

⁷S. Martin, “You Can Be A Millionaire”, Saturday Night Live, January 21, 1978. Appears on *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

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 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 1992.⁸ 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 $R = 435$.

```

APPORTIONCONGRESS( $P[1..n], R$ ):
     $H \leftarrow \text{NEWMAXHEAP}$ 
    for  $i \leftarrow 1$  to  $n$ 
         $r[i] \leftarrow 1$ 
        INSERT( $H, i, P[i]/\sqrt{2}$ )
     $R \leftarrow R - n$ 
    while  $R > 0$ 
         $s \leftarrow \text{EXTRACTMAX}(H)$ 
         $r[s] \leftarrow r[s] + 1$ 
        INSERT( $H, i, P[i]/\sqrt{r[i](r[i] + 1)}$ )
         $R \leftarrow R - 1$ 
    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. (The actual law doesn't make those assumptions, of course.) 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 stored in a column of an Excel spreadsheet; you should have learned a more efficient solution in your undergraduate data structures class.

⁸Overruling 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 current congressional apportionment algorithm is described in gruesome 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>.

0.3 Writing down algorithms

Computer programs are concrete representations of algorithms, but 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.⁹ The idiosyncratic syntactic details of C, Java, Python, Ruby, Erlang, OcaML, Scheme, Visual Basic, Smalltalk, Javascript, Forth, TeX, COBOL, Intercal, or Brainfuck¹⁰ are of little or no importance in algorithm design, and focusing on them will only distract you from what's *really* going on.¹¹ 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 natural language, 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 few iterations of a loop doesn't say much about what happens in later iterations.¹² Phrases like ‘and so on’ or ‘repeat this for all n ’ are good indicators that the algorithm should have been described in terms of loops or recursion, and the description should have specified a *generic* iteration of the loop. Similarly, the appearance of the phrase ‘and so on’ in a proof almost always means 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 an appropriate 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 guidelines I follow:

- Use standard imperative programming keywords (*if/then/else*, *while*, *for*, *repeat/until*, *case*, *return*) and notation (*variable* \leftarrow *value*, *Array*[*index*], *function(argument)*, *bigger* > *smaller*, etc.). Keywords should be standard English words: write ‘*else if*’ instead of ‘*elif*’.)
- Use standard mathematical notation for standard mathematical stuff. For example, write \sqrt{x} and a^b and π instead of *sqrt(x)* and *power(a, b)* and *pi*. (One exception: *never* use \forall to represent a for-loop!)
- Avoid mathematical notation if English is clearer. For example, ‘Insert a into X ’ is often preferable to *INSERT(X, a)* or $X \leftarrow X \cup \{a\}$.

⁹See <http://www.ionet.net/~timtroyr/funhouse/beer.html> for implementations of the BOTTLESOFBEER algorithm in over 200 different programming languages.

¹⁰Brainfuck is the well-deserved name of a programming language invented by Urban Müller 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://esolangs.org/wiki/Brainfuck> for a complete language description; links to several interpreters, compilers, and sample programs; and lots of related shit.

¹¹This is, of course, a matter of religious conviction. Linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states (more or less) that people think only in the categories imposed by their languages. According to an extreme formulation of this principle, 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 Aramaic?—but because of inherent structural differences between languages and cultures. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is admittedly some strength to this idea when applied to different programming paradigms. (What's the Y combinator, again? How do templates work? What's an Abstract Factory?) Fortunately, those differences are generally too subtle to have much impact in *this* class.

¹²See <http://www.research.att.com/~njas/sequences/>.

- Indent everything carefully and consistently; the block structure should be visible from across the room. This is especially important for nested loops and conditionals. *Don't* use unnecessary syntactic sugar (like braces or begin/end tags).
- *Don't* typeset keywords in a different font or style. Changing type style emphasizes the keywords, making the reader think the syntactic sugar is important. It isn't!
- Each statement should fit on one line, and each line should contain either exactly one statement or exactly one structuring element (for, while, if). (I sometimes make an exception for short and similar statements like $i \leftarrow i + 1$; $j \leftarrow j - 1$; $k \leftarrow 0$.)
- Use short, 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 and correctly by any competent programmer in *their* favorite 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 these lecture notes are (hopefully) good examples of what we want to see on your homeworks and exams.

0.4 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 efficiently.

Correctness: In some application settings, it is acceptable for programs to behave correctly most of the time, on all 'reasonable' inputs. Not in this class; we require algorithms that are correct for *all possible* inputs. Moreover, we must *prove* that they're correct; trusting our instincts, or trying a few test cases, isn't good enough.¹³ Sometimes correctness is fairly obvious, especially for algorithms you've seen in earlier courses. On the other hand, 'obvious' is all too often a synonym for 'wrong'. 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.¹⁴

But before we can formally prove that our algorithm does what we want it to do, we have to formally state what we want it to do! Usually problems are given to us in real-world terms, not with formal mathematical descriptions. It's up to us, the algorithm designers, to restate these problems 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 answering any question is figuring out the right way to ask it!

It is important to remember 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 to follow if you want to accomplish this task. The same problem may have hundreds of different algorithms; the same algorithm may solve hundreds of different problems.

¹³I say we take off and nuke the entire site from orbit. It's the only way to be sure.

¹⁴If induction is *not* your friend, you will have a hard time in this course.

Running time: The most common way of ranking different algorithms for the same problem is by how fast they run. Ideally, we want the fastest possible algorithm for our problem. In many application settings, it is acceptable for programs to run efficiently most of the time, on all ‘reasonable’ inputs. Not in this class; 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 $\text{BOTTLESOFBEER}(n)$? 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. Downloading 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 $\text{BOTTLESOFBEER}(2n)$ takes about twice as long as singing $\text{BOTTLESOFBEER}(n)$, 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: $\text{BOTTLESOFBEER}(n)$ 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 ← 1 to n
    Sing “On the ith day of Christmas, my true love gave to me”
    for j ← 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 had a Farm”, “There Was an Old Lady Who Swallowed a Fly”, “The House that Jack Built”, “Green Grow the Rushes O”, “Eh, Compare!”, “Alouette”, “Echad Mi Yodea”, “Chad Gadya”, and “Ist das nicht ein Schnitzelbank?”¹⁵ For further details, consult your nearest preschooler.

```
OLDMACDONALD(animals[1..n], noise[1..n]):
for i ← 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 ← 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.”
```

¹⁵Wakko: Ist das nicht Otto von Schnitzelpusskrankengescheitmeyer?

Yakko and Dot: Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!!

```

ALOUETTE(lapart[1..n]):
    Chantez « Alouette, gentille alouette, alouette, je te plumerais. »
    pour tout i de 1 à n
        Chantez « Je te plumerais lapart[i]. Je te plumerais lapart[i]. »
        pour tout j de i - 1 à bas à 1
            Chantez « Et lapart[j] ! Et lapart[j] ! »
            Chantez « Ooooooo ! »
    Chantez « Alouette, gentille alluette, alouette, 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 $\Theta(R \log n)$.

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.

0.5 A Longer Example: Stable Marriage

Every year, thousands of new doctors must obtain internships at hospitals around the United States. During the first half of the 20th century, competition among hospitals for the best doctors led to earlier and earlier offers of internships, sometimes as early as the second year of medical school, along with tighter deadlines for acceptance. In the 1940s, medical schools agreed not to release information until a common date during their students' fourth year. In response, hospitals began demanding faster decisions. By 1950, hospitals would regularly call doctors, offer them an internship, and demand an *immediate* response. Interns were forced to gamble if their third-choice hospital called first—accept and risk losing a better opportunity later, or reject and risk having no position at all.

Finally, a central clearinghouse for internship assignments, now called the National Resident Matching Program, was established in the early 1950s. Each year, doctors submit a ranked list of all hospitals where they would accept an internship, and each hospital submits a ranked list of doctors they would accept as interns. The NRMP then computes an assignment of interns to hospitals that satisfies the following *stability* requirement. For simplicity, let's assume that there are n doctors and n hospitals; each hospital offers exactly one internship; each doctor ranks all hospitals and vice versa; and finally, there are no ties in the doctors' and hospitals' rankings.¹⁶ We say that a matching of doctors to hospitals is *unstable* if there are two doctors α and β and two hospitals A and B , such that

- α is assigned to A , and β is assigned to B ;
- α prefers B to A , and B prefers α to β .

In other words, α and B would both be happier with each other than with their current assignment. The goal of the Resident Match is a *stable* matching, in which no doctor or hospital has an incentive to cheat the system. At first glance, it is not clear that a stable matching exists!

¹⁶In reality, most hospitals offer multiple internships. Each doctor ranks only a subset of the hospitals and vice versa, and there are typically more internships than interested doctors. And then it starts getting complicated.

In 1952, the NRMP adopted the “Boston Pool” algorithm to assign interns, so named because it had been previously used by a regional clearinghouse in the Boston area. The algorithm is often inappropriately attributed to David Gale and Lloyd Shapley, who formally analyzed the algorithm and first proved that it computes a stable matching in 1962.¹⁷ The algorithm proceeds in rounds until every position has been filled. In each round:

1. Some unassigned hospital offers its position to the best doctor (according to the hospital’s preference list) who has not already rejected it.
2. Each doctor is always assigned to the best hospital (according to the doctor’s preference list) that has made her an offer so far. Thus, whenever a doctor receives an offer that she likes more than her current assignment, her assignment changes.

For example, suppose there are three doctors $\alpha, \beta, \gamma, \delta$, and three hospitals A, B, C, D with the following preference lists:

α	β	γ	δ		A	B	C	D
A	A	B	D		δ	β	δ	γ
B	D	A	B		γ	δ	α	β
C	C	C	C		β	α	β	α
D	B	D	A		α	γ	γ	δ

The algorithm might proceed as follows:

1. A makes an offer to δ .
2. B makes an offer to β .
3. C makes an offer to δ , who rejects her earlier offer from A .
4. D makes an offer to γ . (From this point on, because there is only one unmatched hospital, the algorithm has no more choices.)
5. A makes an offer to γ , who rejects her earlier offer from D .
6. D makes an offer to β , who rejects her earlier offer from B .
7. B makes an offer to δ , who rejects her earlier offer from C .
8. C makes an offer to α .

At this point we have the assignment $(\alpha, C), (\beta, D), (\gamma, A), (\delta, B)$. You can verify by brute force that this matching is stable, despite the fact that no doctor was hired by her favorite hospital, and no hospital hired its favorite doctor.

Analyzing the algorithm is straightforward. Since each hospital makes an offer to each doctor at most once, the algorithm requires at most n^2 rounds. In an actual implementation, each doctor and hospital can be identified by a unique integer between 1 and n , and the preference lists can be represented as two arrays $DocPref[1..n][1..n]$ and $HosPref[1..n][1..n]$, where $DocPref[\alpha][r]$ represents the r th hospital

¹⁷Gale and Shapely used the metaphor of college admissions. The “Gale-Shapely algorithm” is a prime instance of Stigler’s Law of Eponymy: No scientific discovery is named after its original discoverer. In his 1980 paper that gives the law its name, the statistician Stephen Stigler claimed that this law was first proposed by sociologist Robert K. Merton, although similar sentiments were previously attributed to Vladimir Arnol’d in the 1970’s (“Discoveries are rarely attributed to the correct person.”), Carl Boyer in 1968 (“Clio, the muse of history, often is fickle in attaching names to theorems!”), Alfred North Whitehead in 1917 (“Everything of importance has been said before by someone who did not discover it.”), and even Stephen’s father George Stigler in 1966 (“If we should ever encounter a case where a theory is named for the correct man, it will be noted.”)! The law was dubbed the Zeroth Theorem of the History of Science by historian Ernst Peter Fischer in 2006 (“[E]ine Entdeckung (Regel, Gesetzmäßigkeit, Einsicht), die nach einer Person benannt ist, nicht von dieser Person herrührt.”).

in doctor α 's preference list, and $HosPref[A][r]$ represents the r th doctor in hospital A 's preference list. With the input in this form, the Boston Pool algorithm can be implemented to run in $O(n^2)$ time; we leave the details as an exercise for the reader. A somewhat harder exercise is to prove that there are inputs (and choices of who makes offers when) that force $\Omega(n^2)$ rounds before the algorithm terminates.

But why is it *correct*? Gale and Shapely prove that the Boston Pool algorithm always computes a stable matching as follows. The algorithm must terminate, because each hospital makes an offer to each doctor at most once. When the algorithm terminates, every internship has been filled. Now suppose in the final matching, doctor α is assigned to hospital A but prefers B . Since every doctor accepts the best offer she receives, α received no offer she liked more than A . In particular, B never made an offer to α . On the other hand, B made offers to every doctor they like more than β . Thus, B prefers β to α , and so there is no instability.

Surprisingly, the correctness of the algorithm does not depend on which hospital makes its offer in which round. In fact, there is a stronger sense in which the order of offers doesn't matter—no matter which unassigned hospital makes an offer in each round, *the algorithm always computes the same matching!* Let's say that α is a *feasible* doctor for A if there is a stable matching that assigns doctor α to hospital A , and let $best(A)$ be the highest-ranked feasible doctor on A 's preference list.

Lemma 1. *The Boston Pool algorithm assigns $best(A)$ to A , for every hospital A .*

Proof: In the final matching, no hospital A is assigned a doctor they prefer to $best(A)$, because that would create an instability (by definition of 'feasible' and 'best'). A hospital A can only be assigned a doctor they like less than $best(A)$ if $best(A)$ rejects their offer.

So consider the *first* round where some hospital A is rejected by its best choice $\alpha = best(A)$. In that round, α got an offer from another hospital B that α ranks higher than A . Now, B must rank α at least as highly as its best choice $best(B)$, because this is the *first* round where a hospital is rejected by its best choice. Thus, B must rank α higher than *any* doctor that is feasible for B .

Now consider the stable matching where α is assigned to A . On the one hand, α prefers B to A . On the other hand, because this is a stable matching, B is assigned a doctor β that is feasible for B , which implies that B prefers α to β . But this means the matching is unstable, and we have a contradiction. \square

In other words, from the hospitals' point of view, the Boston Pool algorithm computes the best possible stable matching. It turns out that this is also the *worst* possible matching from the doctors' viewpoint! Let $worst(\alpha)$ be the lowest-ranked feasible hospital on doctor α 's preference list.

Lemma 2. *The Boston Pool algorithm assigns α to $worst(\alpha)$, for every doctor α .*

Proof: Suppose the Boston Pool algorithm assigns doctor α to hospital A ; the previous lemma implies that $\alpha = best(A)$. To prove the lemma, we need to show that $A = worst(\alpha)$.

Consider an arbitrary stable matching where A is *not* matched with α but with another doctor β . Then A must prefer α to β . Since this matching is stable, α must therefore prefer her current assignment to A . This argument works for *any* stable assignment, so α prefers *every* other feasible match to A ; in other words, $A = worst(\alpha)$. \square

In 1998, the National Residency Matching Program reversed its matching algorithm, so that potential residents offer to work for hospitals, and each hospital accepts its best offer. Thus, the new algorithm computes the best possible stable matching for the doctors, and the worst possible stable matching for the hospitals. As far as I know, the precise effect of this change on the *patients* is an open problem.

0.6 Why are we here, anyway?

This class is ultimately about learning two skills that are crucial for all computer scientists.

1. **Reasoning:** How to *think* about abstract computation.
2. **Communication:** How to *talk* about abstract computation.

The first 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? These are *not* easy questions. Anyone who says differently is selling something.

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 to clarify your own understanding. As Albert Einstein (or was it Richard Feynman?) apocryphally put it, “You do not really understand something unless you can explain it to your grandmother.”

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 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 *incredibly* useful, and it's impossible to develop good intuition and good communication skills without them, but they aren't the main point of the course either. At this point in your educational career, you should be able to pick up most of those skills on your own, once you know what you're trying to do.

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 activities (music, writing, juggling, acting, martial arts, sports, cooking, programming, teaching, etc.), the *only* way to master these skills is to make them your own, through practice, practice, and more practice. You can only develop good problem-solving skills by solving problems. You can only develop good communication skills by communicating. Good intuition is the product of experience, not its replacement. We *can't* teach you how to do well in this class. All we can do (and what we will do) is lay out some fundamental tools, show you how to use them, create opportunities for you to practice with them, and give you honest feedback, based on our own hard-won 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!



Boethius the algorist versus Pythagoras the abacist.
from *Margarita Philosophica* by Gregor Reisch (1503)

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. The traditional Devonian/Cornish drinking song “The Barley Mow” has the following pseudolyrics¹⁸, where $container[i]$ is the name of a container that holds 2^i ounces of beer. One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. (Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.)

BARLEYMOW(n):

“Here’s a health to the barley-mow, my brave boys,”
 “Here’s a health to the barley-mow!”

“We’ll drink it out of the jolly brown bowl,”
 “Here’s a health to the barley-mow!”

“Here’s a health to the barley-mow, my brave boys,”
 “Here’s a health to the barley-mow!”

for $i \leftarrow 1$ to n
 “We’ll drink it out of the container[i], boys,”
 “Here’s a health to the barley-mow!”

for $j \leftarrow i$ downto 1
 “The container[j],”
 “And the jolly brown bowl!”

“Here’s a health to the barley-mow!”
 “Here’s a health to the barley-mow, my brave boys,”
 “Here’s a health to the barley-mow!”

- (a) Suppose each container name $container[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)
- (b) If you want to sing this song for $n > 20$, you’ll have to make up your own container names. To avoid repetition, these names will get progressively longer as n increases¹⁹. Suppose $container[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)
- (c) Suppose each time you mention the name of a container, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $container[i]$. Assuming for purposes of this problem that you are at least 21 years old, exactly how many ounces of beer would you drink if you sang BARLEYMOW(n)? (Give an exact answer, not just an asymptotic bound.)
2. Describe and analyze the Boston Pool stable matching algorithm in more detail, so that the worst-case running time is $O(n^2)$, as claimed earlier in the notes.

¹⁸Pseudolyrics are to lyrics as pseudocode is to code.

¹⁹“We’ll drink it out of the hemisemidemiyottapint, boys!”

3. Consider a generalization of the stable matching problem, where some doctors do not rank all hospitals and some hospitals do not rank all doctors, and a doctor can be assigned to a hospital only if each appears in the other's preference list. In this case, there are three additional unstable situations:

- A hospital prefers an unmatched doctor to its assigned match.
- A doctor prefers an unmatched hospital to its assigned match.
- An unmatched doctor and an unmatched hospital each appear in the other's preference list.

Describe and analyze an efficient algorithm that computes a stable matching in this setting.

Note that a stable matching may leave some doctors and hospitals unmatched, even though their preference lists are non-empty. For example, if every doctor lists Harvard as their only acceptable hospital, and every hospital lists Doogie as their only acceptable intern, then only Doogie and Harvard will be matched.

4. Recall that the input to the Huntington-Hill apportionment algorithm `APPORTIONCONGRESS` is an array $P[1..n]$, where $P[i]$ is the population of the i th state, and an integer R , the total number of representatives to be allotted. The output is an array $r[1..n]$, where $r[i]$ is the number of representatives allotted to the i th state by the algorithm.

Let $P = \sum_{i=1}^n P[i]$ denote the total population of the country, and let $r_i^* = R \cdot P[i]/P$ denote the ideal number of representatives for the i th state.

- (a) Prove that $r[i] \geq \lfloor r_i^* \rfloor$ for all i .
- (b) Describe and analyze an algorithm that computes exactly the same congressional apportionment as `APPORTIONCONGRESS` in $O(n \log n)$ time. (Recall that the running time of `APPORTIONCONGRESS` depends on R , which could be arbitrarily larger than n .)
- (c) If a state's population is small relative to the other states, its ideal number r_i^* of representatives could be close to zero; thus, tiny states are over-represented by the Huntington-Hill apportionment process. Surprisingly, this can also be true of very large states. Let $\alpha = (1 + \sqrt{2})/2 \approx 1.20710678119$. Prove that for any $\varepsilon > 0$, there is an input to `APPORTIONCONGRESS` with $\max_i P[i] = P[1]$, such that $r[1] > (\alpha - \varepsilon)r_1^*$.
- ★(d) Can you improve the constant α in the previous question?

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

— Henry David Thoreau

*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)

Nothing is particularly hard if you divide it into small jobs.

— Henry Ford

1 Recursion

1.1 Simplify and delegate

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 Lecture 0 reduces the problem of apportioning Congress to the problem of maintaining a priority queue under the operations `INSERT` and `EXTRACTMAX`. Those data structure operations are black boxes; the apportionment algorithm does not depend on any specific implementation. Conversely, when we design a particular priority queue data structure, we typically neither know nor care how our data structure will be used. Whether or not the Census Bureau plans to use our code to apportion Congress is completely irrelevant to our design choices. As a general rule, when we design algorithms, we may not know—and we should not care—how the basic building blocks we use are implemented, or how your algorithm might be used as a basic building block to solve a bigger problem.

A particularly powerful kind of reduction is *recursion*, which can be defined loosely as follows:

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more *simpler instances of the same problem*.

If the self-reference is confusing, it's helpful 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 simpler subproblems.¹

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’ subproblems. Eventually, the recursive reductions must stop with an elementary *base case* that can be solved by some other method; otherwise, the recursive algorithm will never terminate. This finiteness condition is almost always satisfied trivially, but we should always be wary of ‘obvious’ recursive algorithms that actually recurse forever.²

¹I used to refer to ‘elves’ instead of the Recursion Fairy, referring to the traditional fairy tale about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves have finished everything overnight. Someone more entheogenically experienced than I might recognize them as Terence McKenna’s ‘self-adjusting machine elves’.

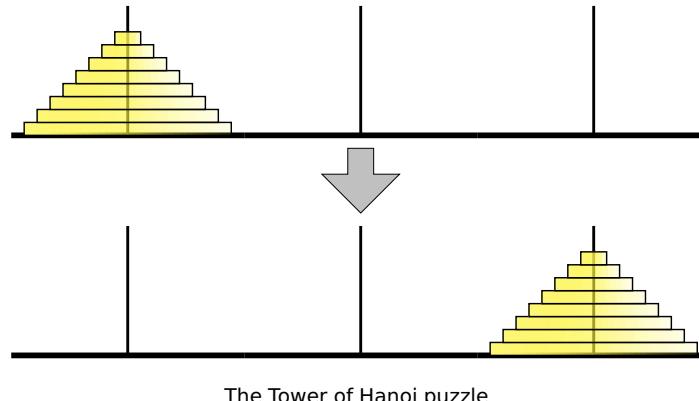
²All too often, ‘obvious’ is a synonym for ‘false’.

1.2 Tower of Hanoi

The Tower of Hanoi puzzle was first published by the mathematician François Édouard Anatole Lucas in 1883, under the pseudonym ‘N. Claus (de Siam)’ (an anagram of ‘Lucas d’Amiens’). The following year, 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.

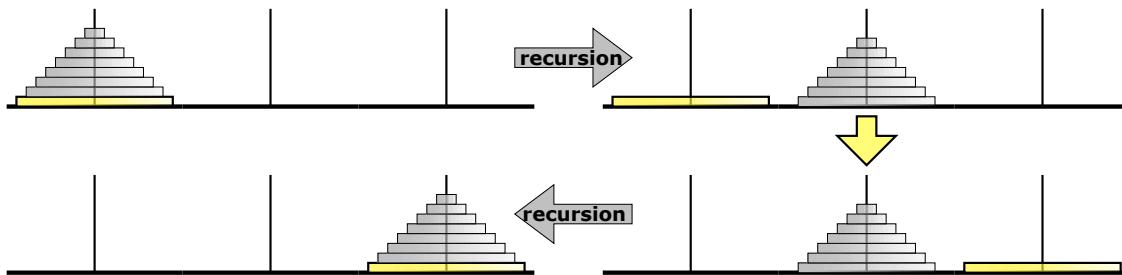
Of course, being good computer scientists, we read this story and immediately substitute n for the hardwired constant sixty-four. How can we move a tower of n disks from one needle to another, using a third needles as an occasional placeholder, never placing any disk on top of a smaller disk?



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 n th disk. And then after we move the n th 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…

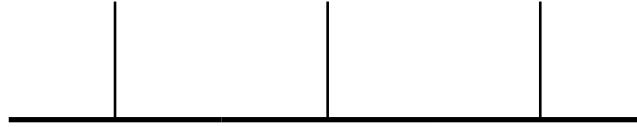
STOP!! That’s it! We’re done! We’ve successfully reduced the n -disk Tower of Hanoi problem to two instances of the $(n - 1)$ -disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy (or, to carry the original story further, to the junior monks at the temple).



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

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

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 spoon.

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, unfolding the recursive calls is merely distracting. 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 $T(n) = 2^n - 1$. 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 tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

1.3 MergeSort

Mergesort is one of the earliest algorithms proposed for sorting. According to Donald 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 —and we can delegate the second step to 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.

<pre> MERGESORT(A[1..n]): if (n > 1) m ← ⌊n/2⌋ MERGESORT(A[1..m]) MERGESORT(A[m+1..n]) MERGE(A[1..n], m) </pre>	<pre> MERGE(A[1..n], m): i ← 1; j ← m + 1 for k ← 1 to n if j > n B[k] ← A[i]; i ← i + 1 else if i > m B[k] ← A[j]; j ← j + 1 else if A[i] < A[j] B[k] ← A[i]; i ← i + 1 else B[k] ← A[j]; j ← j + 1 for k ← 1 to n A[k] ← B[k] </pre>
--	---

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.⁴ This completes the proof.

Now we can prove `MERGESORT` correct by another round of straightforward induction. The base cases $n \leq 1$ are trivial. Otherwise, by the inductive hypothesis, the two smaller subarrays $A[1..m]$ and $A[m+1..n]$ 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`:

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

Aside: Domain Transformations. Except for the floor and ceiling, this recurrence falls firmly into the “all levels equal” case of the recursion tree method, or its corollary, the Master Theorem. If we simply ignore the floor and ceiling, the method 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:

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

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

⁴“The inductive hypothesis” is just a technical nickname for our friend the Recursion Fairy.

versions of the recurrence for $T(n + \alpha)$:

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

For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 + 1$, which implies that $\alpha = 2$. The recursion tree method 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 realize this, we don't need to bother grinding through the details ever again!

1.4 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.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	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  $A[p]$ 
     $k \leftarrow \text{PARTITION}(A, p)$ 
    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

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

and we'd have $T(n) = O(n \log n)$ by the recursion tree method.

In fact, it is possible to locate the median element in an unsorted array in linear time. However, 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

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

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 $T(n) \leq T(1) + T(n-2) + O(n)$, 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. We will formalize this intuition about average cases in a later lecture.

1.5 The Pattern

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

1. **Divide** the problem into several *smaller independent* subproblems.
2. **Delegate** each subproblem to the Recursion Fairy to get a sub-solution.
3. **Combine** the sub-solutions 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 recursion trees (or, if you insist, the Master Theorem), perhaps after a simple domain transformation.

1.6 Median Selection

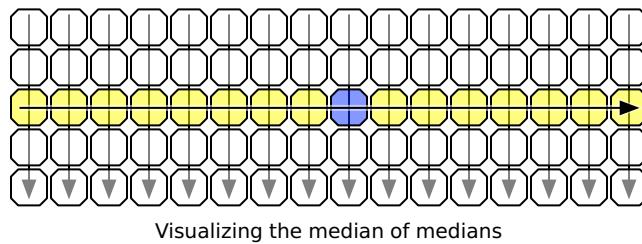
So, how do we find the median element of an array in linear time? The following algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the k th largest element in an array, using the following recursive divide-and-conquer strategy. The subroutine PARTITION is the same as the one used in QUICKSORT.

```
SELECT( $A[1..n], k$ ):
    if  $n \leq 25$ 
        use brute force
    else
         $m \leftarrow \lceil n/5 \rceil$ 
        for  $i \leftarrow 1$  to  $m$ 
             $B[i] \leftarrow \text{SELECT}(A[5i - 4..5i], 3)$       «Brute force!»
            mom  $\leftarrow \text{SELECT}(B[1..m], \lfloor m/2 \rfloor)$       «Recursion!»
         $r \leftarrow \text{PARTITION}(A[1..n], mom)$ 
        if  $k < r$ 
            return  $\text{SELECT}(A[1..r - 1], k)$       «Recursion!»
        else if  $k > r$ 
            return  $\text{SELECT}(A[r + 1..n], k - r)$       «Recursion!»
        else
            return mom
```

If the input array is too large to handle by brute force, we divide it into $\lceil n/5 \rceil$ blocks, each containing exactly 5 elements, except possibly the last. (If the last block isn't full, just throw in a few ∞ s.) We find the median of each block by brute force and collect those medians into a new array. Then we recursively compute the median of the new array (the median of medians — hence ‘mom’) and use it to partition the input array. Finally, either we get lucky and the median-of-medians is the k th largest element of A , or we recursively search one of the two subarrays.

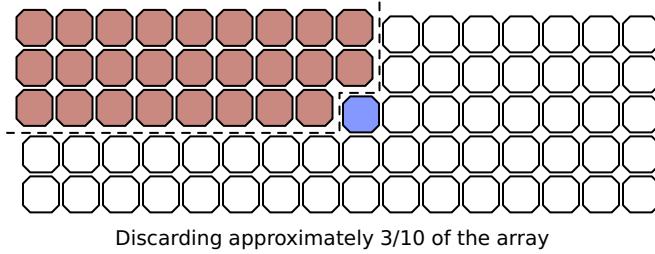
The key insight is that these two subarrays cannot be too large or too small. The median-of-medians is larger than $\lceil \lceil n/5 \rceil / 2 \rceil - 1 \approx n/10$ medians, and each of those medians is larger than two other elements in its block. In other words, the median-of-medians is larger than at least $3n/10$ elements in the input array. Symmetrically, mom is smaller than at least $3n/10$ input elements. Thus, in the worst case, the final recursive call searches an array of size $7n/10$.

We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that the *algorithm* doesn't actually do this!) In this arrangement, the median-of-medians is the element closest to the center of the grid.



The left half of the first three rows of the first three columns contains $3n/10$ elements, each of which is smaller than the median-of-medians. If the element we're looking for is larger than the median-of-medians,

our algorithm will throw away *everything* smaller than the median-of-median, including those $3n/10$ elements, before recursing. A symmetric argument applies when our target element is smaller than the median-of-medians.



We conclude that the worst-case running time of the algorithm obeys the following recurrence:

$$T(n) \leq O(n) + T(n/5) + T(7n/10).$$

The recursion tree method implies the solution $T(n) = O(n)$.

A finer analysis reveals that the hidden constants are quite large, even if we count only comparisons; this is not a practical algorithm for small inputs. (In particular, mergesort uses fewer comparisons in the worst case when $n < 4,000,000$.) Selecting the median of 5 elements requires 6 comparisons, so we need $6n/5$ comparisons to set up the recursive subproblem. We need another $n - 1$ comparisons to partition the array after the recursive call returns. So the actual recurrence is

$$T(n) \leq 11n/5 + T(n/5) + T(7n/10).$$

The recursion tree method implies the upper bound

$$T(n) \leq \frac{11n}{5} \sum_{i \geq 0} \left(\frac{9}{10}\right)^i = \frac{11n}{5} \cdot 10 = 22n.$$

1.7 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 get a more efficient algorithm by exploiting the following identity:

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

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
     $m \leftarrow \lceil n/2 \rceil$ 
     $a \leftarrow \lfloor x/10^m \rfloor$ ;  $b \leftarrow x \bmod 10^m$ 
     $d \leftarrow \lfloor y/10^m \rfloor$ ;  $c \leftarrow y \bmod 10^m$ 
     $e \leftarrow \text{MULTIPLY}(a, c, m)$ 
     $f \leftarrow \text{MULTIPLY}(b, d, m)$ 
     $g \leftarrow \text{MULTIPLY}(b, c, m)$ 
     $h \leftarrow \text{MULTIPLY}(a, d, m)$ 
    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

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

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

But there's a trick, first published by Anatoliï Karatsuba in 1962.⁵ 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 us replace the last three lines in the previous algorithm as follows:

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

⁵However, the same basic trick was used non-recursively by Gauss in the 1800s to multiply complex numbers using only three real multiplications.

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

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

After a domain transformation, we can plug this into a recursion tree to get the solution $T(n) = O(n^{\lg 3}) = O(n^{1.585})$, a significant improvement over our earlier quadratic-time algorithm.⁶

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.⁷ We'll talk about Fast Fourier transforms in detail in another lecture.

1.8 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 we really require is that a belong to a multiplicative group.⁸ Since we don't know what kind of things we're multiplying, 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 following simple recursive 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.

⁶Karatsuba 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.

⁷This fast algorithm for multiplying integers using FFTs was discovered by Arnold Schönhage and Volker Strassen in 1971. The $O(n \log n)$ running time requires the standard assumption that $O(\log n)$ -bit integer arithmetic can be performed in constant time; the number of bit operations is $O(n \log n \log \log n)$.

⁸A *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 an *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 $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

```

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

```

The total number of multiplications satisfies the recurrence $T(n) \leq T(\lfloor n/2 \rfloor) + 2$, with the base case $T(1) = 0$. After a domain transformation, recursion trees give us the solution $T(n) = O(\log n)$.

Incidentally, this algorithm is asymptotically optimal—any algorithm for computing a^n must perform at least $\Omega(\log n)$ 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 ($a \rightarrow a^2 \rightarrow a^3 \rightarrow a^5 \rightarrow a^{10} \rightarrow a^{15}$). Nobody knows of an efficient algorithm that always uses the minimum possible number of multiplications.

Exercises

1. (a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character &. Preorder and postorder traversals of the tree visit the nodes in the following order:
 - Preorder: **I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X**
 - Postorder: **H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I**
 Draw George's binary tree.

 (b) Describe and analyze a recursive algorithm for reconstructing a binary tree, given its preorder and postorder node sequences.

 (c) Describe and analyze a recursive algorithm for reconstructing a binary tree, given its preorder and *inorder* node sequences.

2. Consider a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed.
 - (a) Prove that any such chessboard can be tiled without gaps or overlaps by L-shaped pieces, each composed of 3 squares.
 - (b) Describe and analyze an algorithm to compute such a tiling, given the integer n and two n -bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time.

3. Prove that the recursive Tower of Hanoi algorithm is *exactly equivalent* to each of the following non-recursive algorithms; in other words, prove that all three algorithms move the same disks, to and from the same needles, in the same order. The needles are labeled 0, 1, and 2, and our problem is to move a stack of n disks from needle 0 to needle 2 (as shown on page 2).
 - (a) Follow these four rules:
 - Never move the same disk twice in a row.

- If n is even, always move the smallest disk forward ($\cdots \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow \cdots$).
 - If n is odd, always move the smallest disk backward ($\cdots \rightarrow 0 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow \cdots$).
 - When there is no move that satisfies the other rules, the puzzle is solved.
- (b) 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:

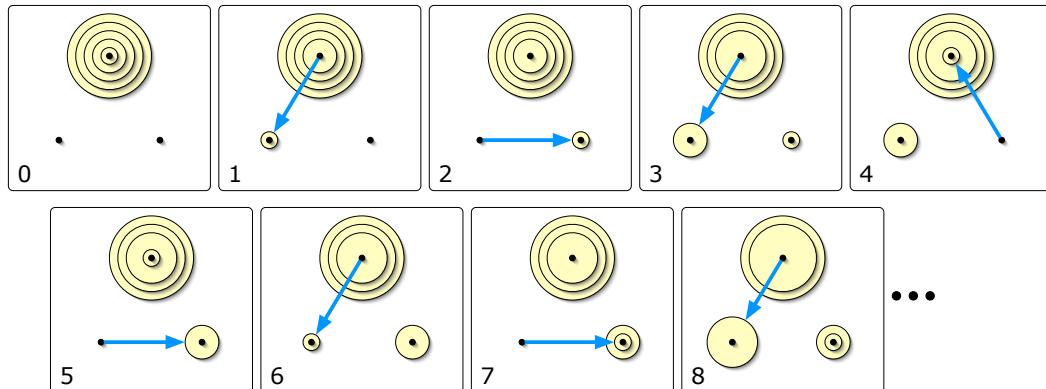
$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 6, 1, 2, 1, 3, 1, \dots$

Here’s the non-recursive algorithm in one line:

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

When this rule requires us to move disk $n + 1$, the algorithm ends.

4. Consider the following restricted variants of the Tower of Hanoi puzzle. In each problem, the needles are numbered 0, 1, and 2, as in problem 3, and your task is to move a stack of n disks from needle 1 to needle 2.
- Suppose you are forbidden to move any disk directly between needle 1 and needle 2; every move must involve needle 0. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?
 - Suppose you are only allowed to move disks from needle 0 to needle 2, from needle 2 to needle 1, or from needle 1 to needle 0. Equivalently, Suppose the needles are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

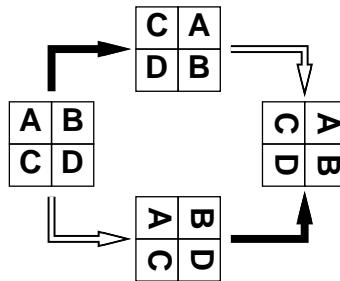


A top view of the first eight moves in a clockwise Towers of Hanoi solution

- ★(c) Finally, suppose your only restriction is that you may never move a disk directly from needle 1 to needle 2. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make? [Hint: This is considerably harder to analyze than the other two variants.]

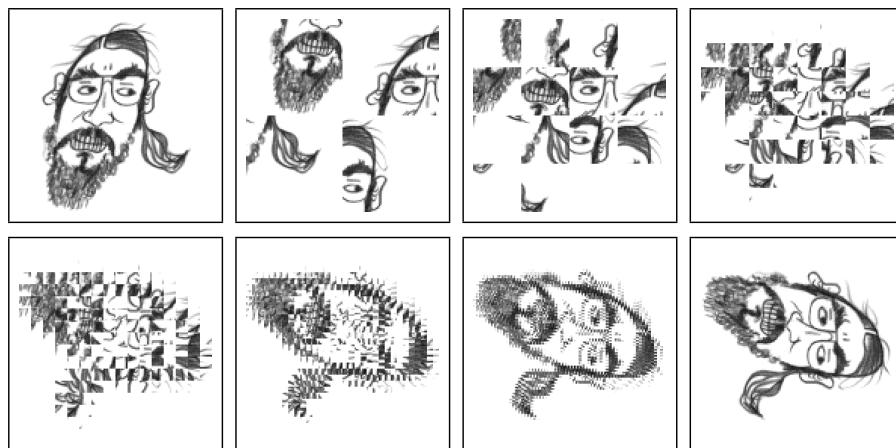
5. Most graphics hardware includes support for a low-level operation called *blit*, or **block transfer**, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.



Two algorithms for rotating a pixel map.

Black arrows indicate blitting the blocks into place; white arrows indicate recursively rotating the blocks.



The first rotation algorithm (blit then recurse) in action.

- (a) Prove that both versions of the algorithm are correct when n is a power of two.
 - (b) *Exactly* how many blits does the algorithm perform when n is a power of two?
 - (c) Describe how to modify the algorithm so that it works for arbitrary n , not just powers of two. How many blits does your modified algorithm perform?
 - (d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?
 - (e) What if a $k \times k$ blit takes only $O(k)$ time?
6. An *inversion* in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n -element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward).

Describe and analyze an algorithm to count the number of inversions in an n -element array in $O(n \log n)$ time. [Hint: Modify mergesort.]

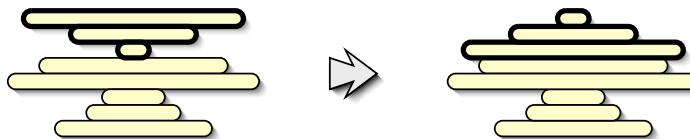
7. (a) Prove that the following algorithm actually sorts its input.

```

STOOGESORT( $A[0..n-1]$ ) :
  if  $n = 2$  and  $A[0] > A[1]$ 
    swap  $A[0] \leftrightarrow A[1]$ 
  else if  $n > 2$ 
     $m = \lceil 2n/3 \rceil$ 
    STOOGESORT( $A[0..m-1]$ )
    STOOGESORT( $A[n-m..n-1]$ )
    STOOGESORT( $A[0..m-1]$ )

```

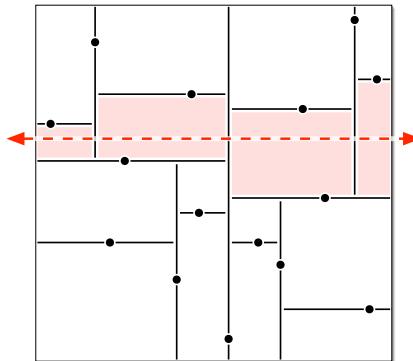
- (b) Would STOOGESORT still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.
- (c) State a recurrence (including the base case(s)) for the number of comparisons executed by STOOGESORT.
- (d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.]
- (e) Prove that the number of swaps executed by STOOGESORT is at most $\binom{n}{2}$.
8. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n , and flip them all over.



Flipping the top three pancakes.

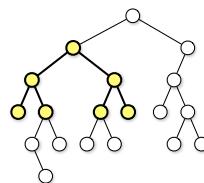
- (a) Describe an algorithm to sort an arbitrary stack of n pancakes using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?
- (b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?
9. You are a contestant on the hit game show “Beat Your Neighbors!” You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year’s supply of Rice-A-Roni™, to which you are hopelessly addicted.
- (a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?
- *(b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?
- *(c) Prove that your solution to part (b) is optimal up to a constant factor.

10. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k . Describe an algorithm to find the k th smallest element in the union of A and B in $\Theta(\log n)$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, your algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicate elements. [Hint: First solve the special case $k = n$.]
- (b) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k . Describe an algorithm to find the k th smallest element in A in $O(\log n)$ time.
- (c) Finally, suppose we are given a two dimensional array $A[1..m][1..n]$ in which every row $A[i][\cdot]$ is sorted, and an integer k . Describe an algorithm to find the k th smallest element in A as quickly as possible. How does the running time of your algorithm depend on m ? [Hint: Use the linear-time SELECT algorithm as a subroutine.]
11. (a) Describe and analyze an algorithm to sort an array $A[1..n]$ by calling a subroutine $\text{SQRTSORT}(k)$, which sorts the subarray $A[k+1..k+\sqrt{n}]$ in place, given an arbitrary integer k between 0 and $n - \sqrt{n}$ as input. (To simplify the problem, assume that \sqrt{n} is an integer.) Your algorithm is **only** allowed to inspect or modify the input array by calling SQRTSORT ; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SQRTSORT in the worst case?
- (b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SQRTSORT , prove that no algorithm can sort using $o(f(n))$ calls to SQRTSORT .
- (c) Now suppose SQRTSORT is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k} , so that repeated square roots are always integers.)
12. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the points as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point inside the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



A kd-tree for 15 points. The dashed line crosses the four shaded cells.

- (a) How many cells are there, as a function of n ? Prove your answer is correct.
- (b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n ? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k .
- (c) Suppose we have n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. [Hint: Use part (b).]
- (d) Describe an analyze an efficient algorithm that counts, given a kd-tree storing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. [Hint: Use part (c).]
13. You are at a political convention with n delegates, each one a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any two delegates belong to the *same* party or not by introducing them to each other—members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.
- (a) Suppose a majority (more than half) of the delegates are from the same political party. Describe an efficient algorithm that identifies a member (*any* member) of the majority party.
- (b) Now suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality political party as parsimoniously as possible. (Please.)
14. The median of a set of size n is its $\lceil n/2 \rceil$ th largest element, that is, the element that is as close as possible to the middle of the set in sorted order. In this lecture, we saw a fairly complicated algorithm to compute the median in $O(n)$ time.
- During your lifelong quest for a simpler linear-time median-finding algorithm, you meet and befriend the Near-Middle Fairy. Given any set X , the Near-Middle Fairy can find an element $m \in X$ that is *near* the middle of X in $O(1)$ time. Specifically, at least a third of the elements of X are smaller than m , and at least a third of the elements of X are larger than m .
- Describe and analyze a simple recursive algorithm to find the median of a set in $O(n)$ time if you are allowed to ask the Near-Middle Fairy for help.
15. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

16. Consider the following classical recursive algorithm for computing the factorial $n!$ of a non-negative integer n :

```
FACTORIAL( $n$ ):
  if  $n = 0$ 
    return 0
  else
    return  $n \cdot \text{FACTORYAL}(n - 1)$ 
```

- (a) How many multiplications does this algorithm perform?
- (b) How many bits are required to write $n!$ in binary? Express your answer in the form $\Theta(f(n))$, for some familiar function $f(n)$. [Hint: Use Stirling's approximation: $n! \approx \sqrt{2\pi n} \cdot (n/e)^n$.]
- (c) Your answer to (b) should convince you that the number of multiplications is *not* a good estimate of the actual running time of FACTORIAL. The grade-school multiplication algorithm takes $O(k \cdot l)$ time to multiply a k -digit number and an l -digit number. What is the running time of FACTORIAL if we use this multiplication algorithm as a subroutine?
- (d) The following algorithm also computes $n!$, but groups the multiplication differently:

```
FACTORYAL2( $n, m$ ):      «Compute  $n!/(n - m)!$ »
  if  $m = 0$ 
    return 1
  else if  $m = 1$ 
    return  $n$ 
  else
    return FACTORYAL2( $n, \lfloor m/2 \rfloor$ ) · FACTORYAL2( $n - \lfloor m/2 \rfloor, \lceil m/2 \rceil$ )
```

What is the running time of FACTORYAL2(n, n) if we use grade-school multiplication? [Hint: Ignore the floors and ceilings.]

- (e) Describe and analyze a variant of Karatsuba's algorithm that can multiply any k -digit number and any l -digit number, where $k \geq l$, in $O(k \cdot l^{\lg 3 - 1}) = O(k \cdot l^{0.585})$ time.
- (f) What are the running times of FACTORIAL(n) and FACTORYAL2(n, n) if we use the modified Karatsuba multiplication from part (e)?

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, eleven, thirteen, 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)

It needs evaluation

So let the games begin

A heinous crime, a show of force

A murder would be nice, of course

— “Bad Horse’s Letter”, Dr. Horrible’s Sing-Along Blog (2008)

*A Fast Fourier Transforms

A.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 :

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

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

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

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

$$p(x) = a_0 + x(a_1 + x(a_2 + \dots + x a_n)).$$

```

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— $O(n^{1+\epsilon})$ for your favorite value $\epsilon > 0$ —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.

A.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.

A.2.1 Roots

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

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

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 of $p + q$. We could convert the polynomials to the more familiar coefficient representation first—this takes $O(n^2)$

¹I'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 using $O(n \log n \log \log n \log \log \log n \log \log \log \log n \dots)$ bit operations. The algorithm uses an n -element Fast Fourier Transform, which requires several $O(\log n)$ -bit 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.

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

A.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 $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$, 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.

A.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	$O(n)$	$O(n)$	$O(n^2)$
roots + scale	$O(n)$	∞	$O(n)$
samples	$O(n^2)$	$O(n)$	$O(n)$

A.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.

²This is where numerical analysis comes from.

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, \dots, x_{n-1} of sample *positions*, and let V be the $n \times n$ matrix where $v_{ij} = x_i^{j-1}$ (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 $\vec{a} = (a_0, a_1, \dots, a_{n-1})$ and the vector of sample values $\vec{y} = (y_0, y_1, \dots, y_{n-1})$ are related by the matrix equation

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

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 implicitly hard-coding the sample positions 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.

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

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 written directly 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 add, multiply, or evaluate directly in either representation in $O(n^2)$ time. But wait!

³Actually, 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.

There's a degree of freedom we haven't exploited—*We get to choose the sample positions!* 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.

A.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 *n th 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 n th root of unity is a power of the *primitive* root

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

A typical n th 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 n th 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 n th root of unity is also a (kn) th root of unity.

If we sample a polynomial of degree $n - 1$ at the n th 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

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

We can obviously compute P^* in $O(n^2)$ time, but the structure of the n th 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 n th root of unity; specifically,

$$v_{jk} = \omega_n^{jk}$$

⁴In this lecture, i always represents the square root of -1 . 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 \mathbf{i} are just stupid.

for all integers j and k . Thus,

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

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 the following amazing fact implies that this is almost the same as multiplying by V itself:

Claim: $V^{-1} = \bar{V}/n$

Proof: Let $W = \bar{V}/n$. We just have to show that $M = VW$ is the identity matrix. We can compute a single entry in M 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 $\omega_n^{j-k} = \omega_n^0 = 1$, 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! □

In other words, if $W = V^{-1}$ then $w_{jk} = \overline{v_{jk}}/n = \overline{\omega_n^{jk}}/n = \omega_n^{-jk}/n$. 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.

A.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 1965⁵, assumes that n is a power of two; if necessary, we can just pad the coefficient vector with zeros.

⁵Actually, 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. So of course it's often called the Coley-Tukey algorithm. But the algorithm 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 himself 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.

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 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

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

In particular, if x is an n th 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 n th 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:

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

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.

<u>FFT($P[0..n - 1]$):</u>
if $n = 1$
return P
for $j \leftarrow 0$ to $n/2 - 1$
$U[j] \leftarrow P[2j]$
$V[j] \leftarrow P[2j + 1]$
$U^* \leftarrow \text{FFT}(U[0..n/2 - 1])$
$V^* \leftarrow \text{FFT}(V[0..n/2 - 1])$
$\omega_n \leftarrow \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right)$
$\omega \leftarrow 1$
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 $P^*[0..n - 1]$

<u>INVERSEFFT($P^*[0..n - 1]$):</u>
if $n = 1$
return P
for $j \leftarrow 0$ to $n/2 - 1$
$U^*[j] \leftarrow P^*[2j]$
$V^*[j] \leftarrow P^*[2j + 1]$
$U \leftarrow \text{INVERSEFFT}(U[0..n/2 - 1])$
$V \leftarrow \text{INVERSEFFT}(V[0..n/2 - 1])$
$\overline{\omega_n} \leftarrow \cos\left(\frac{2\pi}{n}\right) - i \sin\left(\frac{2\pi}{n}\right)$
$\omega \leftarrow 1$
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 $T(n) = \Theta(n \log n)$.

A.6 Fast Multiplication

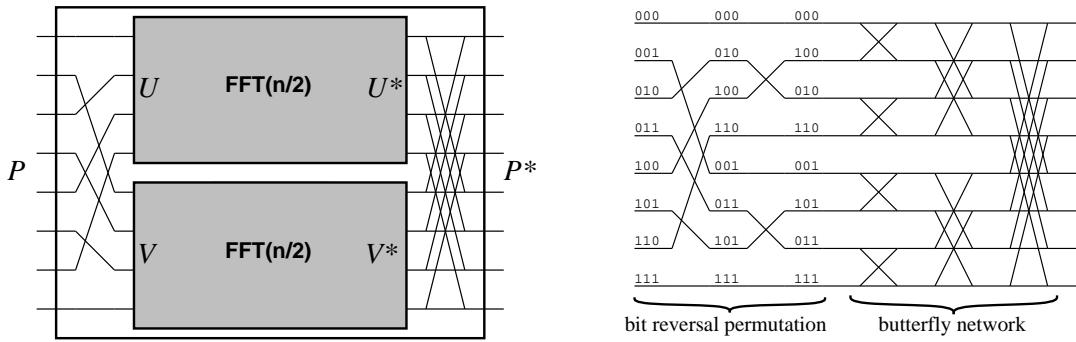
Given two polynomials p and q , each represented by an array of coefficients, we can multiply them in $\Theta(n \log n)$ arithmetic operations as follows. First, pad the coefficient vectors and with zeros until the

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( $R^*$ )
```

A.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:

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

We can evaluate $p(x)$ by recursively evaluating $p_{\text{even}}(x^2)$ and $p_{\text{odd}}(x^2)$ 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 one of the following conditions holds:

- X has one element.
- The set $X^2 = \{x^2 \mid x \in X\}$ has only $n/2$ elements and is delicious.

Clearly the size of any delicious set is a power of 2. 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 $\{0, 1, \dots, n - 1\}$ 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: $\{0, c, 2c, \dots, (n-1)c\}$ is delicious mod cn . We can avoid modular arithmetic entirely by using complex roots of unity—the set $\{e^{2\pi i(k/n)} \mid k = 0, 1, \dots, n - 1\}$ is delicious! The sequence of values $p(e^{2\pi i(k/n)})$ 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.

Exercises

1. For any two sets X and Y of integers, the Minkowski sum $X + Y$ is the set of all pairwise sums $\{x + y \mid x \in X, y \in Y\}$.
 - (a) Describe an analyze and algorithm to compute the number of elements in $X + Y$ in $O(n^2 \log n)$ time. [Hint: The answer is **not** always n^2 .]
 - (b) Describe and analyze an algorithm to compute the number of elements in $X + Y$ in $O(M \log M)$ time, where M is the largest absolute value of any element of $X \cup Y$. [Hint: What's this lecture about?]
2. (a) Describe an algorithm that determines whether a given set of n integers contains two elements whose sum is zero, in $O(n \log n)$ time.
 - (b) Describe an algorithm that determines whether a given set of n integers contains *three* elements whose sum is zero, in $O(n^2)$ time.
 - (c) Now suppose the input set X contains only integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three elements whose sum is zero, in $O(n \log n)$ time. [Hint: Hint.]

'Tis a lesson you should heed,
 Try, try again;
 If at first you don't succeed,
 Try, try again;
 Then your courage should appear,
 For, if you will persevere,
 You will conquer, never fear;
 Try, try again.

— Thomas H. Palmer, *The Teacher's Manual: Being an Exposition of an Efficient and Economical System of Education Suited to the Wants of a Free People* (1840)

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

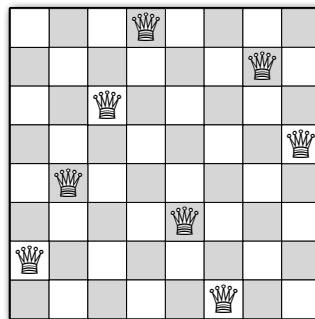
— Yogi Berra

2 Backtracking

In this lecture, I want to describe another recursive algorithm strategy called *backtracking*. A backtracking algorithm tries to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between two alternatives to the next component of the solution, it simply tries both options recursively.

2.1 *n*-Queens

The prototypical backtracking problem is the classical *n-Queens Problem*, first proposed by German chess enthusiast Max Bezzel in 1848 for the standard 8×8 board, and both solved and generalized to larger boards by Franz Nauck in 1850. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens can attack each other. For readers not familiar with the rules of chess, this means that no two queens are in the same row, column, or diagonal.



One solution to the 8 queens problem, represented by the array [4,7,3,8,2,5,1,6]

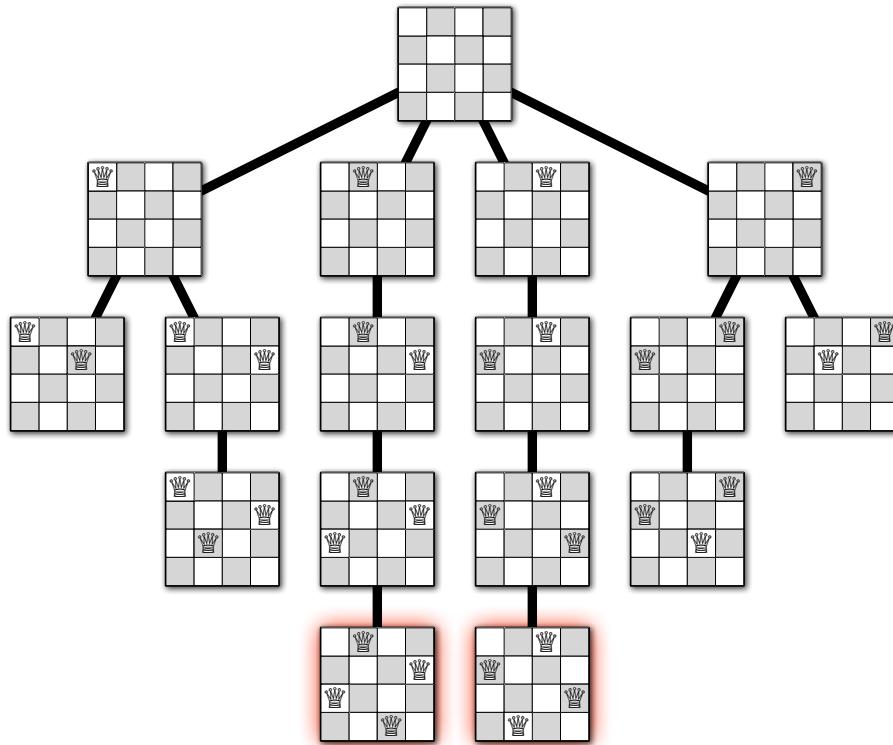
Obviously, in any solution to the *n*-Queens problem, there is exactly one queen in each column. So we will represent our possible solutions using an array $Q[1..n]$, where $Q[i]$ indicates the square in column i that contains a queen, or 0 if no queen has yet been placed in column i . To find a solution, we will put queens on the board row by row, starting at the top. A *partial* solution is an array $Q[1..n]$ whose first $r - 1$ entries are positive and whose last $n - r + 1$ entries are all zeros, for some integer r .

The following recursive algorithm recursively enumerates all complete *n*-queens solutions that are consistent with a given partial solution. The input parameter r is the first empty row. Thus, to compute all *n*-queens solutions with no restrictions, we would call `RECURSIVEQUEENS(Q[1..n], 1)`.

```

RECURSIVENQUEENS( $Q[1..n]$ ,  $r$ ):
  if  $r = n + 1$ 
    print  $Q$ 
  else
    for  $j \leftarrow 1$  to  $n$ 
      legal  $\leftarrow$  TRUE
      for  $i \leftarrow 1$  to  $r - 1$ 
        if ( $Q[i] = j$ ) or ( $Q[i] = r - j + i$ ) or ( $Q[i] = r + i - j$ )
          legal  $\leftarrow$  FALSE
      if legal
         $Q[r] \leftarrow j$ 
        RECURSIVENQUEENS( $Q[1..n]$ ,  $r + 1$ )
    
```

Like most recursive algorithms, the execution of a backtracking algorithm can be illustrated using a *recursion tree*. The root of the recursion tree corresponds to the original invocation of the algorithm; edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the n -Queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that cannot be extended, either because there is already a queen on every row, or because every position in the next empty row is in the same row, column, or diagonal as an existing queen.



The complete recursion tree for our algorithm for the 4-queens problem.

2.2 Subset Sum

Let's consider a more complicated problem, called **SUBSETSUM**: Given a set X of positive integers and *target* integer T , is there a subset of elements in 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 subsets $\{8, 7\}$ or $\{7, 5, 3\}$ or $\{6, 9\}$ or $\{5, 10\}$. 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, because the elements of the empty set add up to zero.¹ 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 is a subset of X that sums to T if and only if one of the following statements is true:

- There is a subset of X that *includes* x and whose sum is T .
- There is a subset of X that *excludes* x and whose sum is T .

In the first case, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$; in the second case, there must be a subset of $X \setminus \{x\}$ that sums to T . So we can solve $\text{SUBSETSUM}(X, T)$ by reducing it to two simpler instances: $\text{SUBSETSUM}(X \setminus \{x\}, T - x)$ and $\text{SUBSETSUM}(X \setminus \{x\}, T)$. Here's how the resulting recursive 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 ( $\text{SUBSETSUM}(X[2..n], T) \vee \text{SUBSETSUM}(X[2..n], T - X[1])$ )
```

Proving this algorithm correct is a straightforward exercise in induction. If $T = 0$, then the elements of the empty subset sum to T , so TRUE is the correct output. Otherwise, if T is negative or the set X is empty, then no subset of X sums to T , so FALSE is the correct output. 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.

Here is a similar recursive algorithm that actually *constructs* 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  $\emptyset$ 
    if  $T < 0$  or  $n = 0$ 
        return NONE
     $Y \leftarrow \text{CONSTRUCTSUBSET}(X[2..n], T)$ 
    if  $Y \neq \text{NONE}$ 
        return  $Y$ 
     $Y \leftarrow \text{CONSTRUCTSUBSET}(X[2..n], T - X[1])$ 
    if  $Y \neq \text{NONE}$ 
        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

¹There's no base case like the vacuous base case!

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 as 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 .²

2.3 Longest Increasing Subsequence

Now suppose we are given a sequence of integers, and we want to find the longest subsequence whose elements are in increasing order. More concretely, the input is an array $A[1..n]$ of integers, and we want to find the longest sequence of indices $1 \leq i_1 < i_2 < \dots < i_k \leq n$ 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 $A[1] \leq x$, the LIS of $A[1..n]$ with elements larger than x is
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.

²Indeed, because SUBSETSUM is NP-hard, there is almost certainly no way to solve it in subexponential time.

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.

$\text{LIS}(A[1..n]):$ return $\text{LISBIGGER}(-\infty, A[1..n])$	$\text{LISBIGGER}(prev, A[1..n]):$ if $n = 0$ return 0 else $max \leftarrow \text{LISBIGGER}(prev, A[2..n])$ if $A[1] > prev$ $L \leftarrow 1 + \text{LISBIGGER}(A[1], A[2..n])$ if $L > max$ $max \leftarrow L$ return max
---	--

The running time of this algorithm satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$, so as usual the annihilator method implies that $T(n) = O(2^n)$. 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.

The following alternative strategy 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 new strategy gives us the following algorithm:

$\text{FILTER}(A[1..n], x):$ $j \leftarrow 1$ for $i \leftarrow 1$ to n if $A[i] > x$ $B[j] \leftarrow A[i]; j \leftarrow j + 1$ return $B[1..j]$	$\text{LIS}(A[1..n]):$ if $n = 0$ return 0 else $max \leftarrow \text{LIS}(prev, A[2..n])$ $L \leftarrow 1 + \text{LIS}(A[1], \text{FILTER}(A[2..n], A[1]))$ 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; indeed, if the input sequence is sorted in increasing order, this assumption is correct.

2.4 Optimal Binary Search Trees

Our next example combines recursive backtracking with the divide-and-conquer strategy.

Hopefully you remember that the cost of a successful search in a binary search tree is proportional to the number of ancestors of the target node.³ 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

³An *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 .

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 $A[i]$, and let r be the index of the root node. Ignoring constant factors, the cost of searching for $A[i]$ is the number of nodes on the path from the root v_r to v_i . Thus, the total cost of performing all the binary searches is given by the following expression:

$$\text{Cost}(T, f[1..n]) = \sum_{i=1}^n f[i] \cdot \#\text{nodes between } v_r \text{ and } v_i$$

Every search path includes the root node v_r . If $i < r$, then all other nodes on the search path to v_i are in the left subtree; similarly, if $i > r$, all other nodes on the search path to v_i are in the right subtree. Thus, we can partition the cost function into three parts as follows:

$$\begin{aligned} \text{Cost}(T, f[1..n]) &= \sum_{i=1}^{r-1} f[i] \cdot \#\text{nodes between } \text{left}(v_r) \text{ and } v_i \\ &\quad + \sum_{i=1}^n f[i] \\ &\quad + \sum_{i=r+1}^n f[i] \cdot \#\text{nodes between } \text{right}(v_r) \text{ and } v_i \end{aligned}$$

Now the first and third summations look exactly like our original expression (*) for $\text{Cost}(T, f[1..n])$. Simple substitution gives us our recursive definition for Cost :

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

The base case for this recurrence is, as usual, $n = 0$; the cost of performing no searches in the empty tree 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 $\text{Cost}(T, f)$ immediately implies that the left subtree $\text{left}(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree $\text{right}(T_{\text{opt}})$ 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 $\text{OptCost}(f[1..n])$ denote the total cost of the optimal search tree for the given frequency counts. We immediately have the following recursive definition.

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

Again, the base case is $\text{OptCost}(f[1..0]) = 0$; the best way to organize no keys, which we will plan to search zero times, 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

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

The $\Theta(n)$ term comes from computing the total number of searches $\sum_{i=1}^n f[i]$.

Yeah, that's one ugly recurrence, but it's actually easier to solve than it looks. 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 $T(n) = \Theta(3^n)$ now follows from the annihilator method.

Let me emphasize that this recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with n nodes satisfies the recurrence

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

which has the closed-form 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 $N(n)$.

Exercises

1. (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is both a subsequence of A and a subsequence of B . Give a simple recursive definition for the function $\text{lcs}(A, B)$, which gives the length of the *longest common subsequence* of A and B .
- (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of A and B is another sequence that contains both A and B as subsequences. Give a simple recursive definition for the function $\text{scs}(A, B)$, which gives the length of the *shortest common supersequence* of A and B .
- (c) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i , and $X[i] > X[i+1]$ for all odd i . Give a simple recursive definition for the function $\text{los}(A)$, which gives the length of the longest oscillating subsequence of an arbitrary array A of integers.
- (d) Give a simple recursive definition for the function $\text{sos}(A)$, which gives the length of the shortest oscillating supersequence of an arbitrary array A of integers.

- (e) Call a sequence $X[1..n]$ *accelerating* if $2 \cdot X[i] < X[i - 1] + X[i + 1]$ for all i . Give a simple recursive definition for the function $lxs(A)$, which gives the length of the longest accelerating subsequence of an arbitrary array A of integers.

For more backtracking exercises, see the next two lecture notes!

*Wouldn't the sentence "I want to put a hyphen between the words Fish and And and And and Chips in my Fish-And-Chips sign." have been clearer if quotation marks had been placed before Fish, and between Fish and and, and and and Chips, as well as after Chips?*¹

— unknown, from the original FreeBSD ‘fortune’ file

*B Efficient Exponential-Time Algorithms

In another lecture note, we discuss the class of *NP-hard* problems. For every problem in this class, the fastest algorithm anyone knows has an exponential running time. Moreover, there is *very* strong evidence (but alas, no proof) that it is *impossible* to solve any NP-hard problem in less than exponential time—it’s not that we’re all stupid; the problems really are that hard! Unfortunately, an enormous number of problems that arise in practice are NP-hard; for some of these problems, even *approximating* the right answer is NP-hard.

Suppose we absolutely have to find the exact solution to some NP-hard problem. A polynomial-time algorithm is almost certainly out of the question; the best running time we can hope for is exponential. But *which* exponential? An algorithm that runs in $O(1.5^n)$ time, while still unusable for large problems, is still significantly better than an algorithm that runs in $O(2^n)$ time!

For most NP-hard problems, the only approach that is guaranteed to find an optimal solution is recursive backtracking. The most straightforward version of this approach is to recursively generate *all* possible solutions and check each one: all satisfying assignments, or all vertex colorings, or all subsets, or all permutations, or whatever. However, most NP-hard problems have some additional structure that allows us to prune away most of the branches of the recursion tree, thereby drastically reducing the running time.

B.1 3SAT

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 $\text{poly}(n)$ to represent some arbitrary polynomial in n ; in other words, $\text{poly}(n) = n^{O(1)}$. For example, the trivial algorithm for 3SAT runs in time $O(2^n \text{poly}(n))$.

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

¹If you ever decide to read this sentence out loud, be sure to pause briefly between ‘Fish and and’ and ‘and and and And’, ‘and and and And’ and ‘and And and and’, ‘and And and and’ and ‘and and and And’, ‘and and and And’ and ‘and And and and’, and ‘and And and and’ and ‘and and and Chips’!

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

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

whose solution is $O(3^n \text{poly}(n))$. 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  $\Phi = \emptyset$ 
    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

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

where $\text{poly}(n)$ 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

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

where $\lambda \approx 1.83928675521\dots$ 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 Φ 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  $\Phi = \emptyset$ 
    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

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

and the annihilator method implies that

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

where $\mu \approx 1.76929235424\dots$ 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 $O(1.473^n)$ time², but there is absolutely no reason to believe that this is the best possible.

B.2 Maximum Independent Set

Now suppose we are given an undirected graph G and are asked to find the size of the *largest independent set*, that is, the largest subset of the vertices of G with no edges between them. Once again, we have an obvious recursive algorithm: Try every subset of nodes, and return the largest subset with no edges. Expressed recursively, the algorithm might look like this.

²Tobias 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  $G = \emptyset$ 
    return 0
else
     $v \leftarrow$  any node in  $G$ 
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$ 
     $withoutv \leftarrow \text{MAXIMUMINDSETSIZE}(G \setminus \{v\})$ 
    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 $T(n) = 2T(n - 1) + \text{poly}(n) = O(2^n \text{poly}(n))$. Surprise, surprise.

This algorithm is mirroring a crude recursive upper bound for the number of *maximal* independent sets in a graph; an independent set is maximal if every vertex in G is either already in the set or a neighbor of a vertex in the set. 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 $O(3^{n/3} \text{poly}(n)) = O(1.44224957032^n)$ time.

```
MAXIMUMINDSETSIZE( $G$ ):
  if  $G = \emptyset$ 
    return 0
  else if  $G$  has a node  $v$  with degree 0
    return  $1 + \text{MAXIMUMINDSETSIZE}(G \setminus \{v\})$       « $n - 1$ »
  else if  $G$  has a node  $v$  with degree 1
     $w \leftarrow v$ 's neighbor
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$     « $n - 2$ »
     $withw \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(w))$     « $\leq n - 2$ »
    return  $\max\{withv, withw\}$ 
  else if  $G$  has a node  $v$  with degree greater than 2
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$     « $\leq n - 4$ »
     $withoutv \leftarrow \text{MAXIMUMINDSETSIZE}(G \setminus \{v\})$       « $\leq n - 1$ »
    return  $\max\{withv, withoutv\}$ 
  else «every node in  $G$  has degree 2»
     $v \leftarrow$  any node;  $u, w \leftarrow v$ 's neighbors
     $withu \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(u))$     « $\leq n - 3$ »
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$     « $\leq n - 3$ »
     $withw \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(w))$     « $\leq n - 3$ »
    return  $\max\{withu, withv, withw\}$ 
```

Exercises

- *1. Describe an algorithm to solve 3SAT in time $O(\phi^n \text{poly}(n))$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618034$.
[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]

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 Steelers’ head coach Bill Cowher, responding to David Romer’s dynamic-programming analysis of football strategy (2003)

3 Dynamic Programming

3.1 Fibonacci Numbers

The Fibonacci numbers F_n , named after Leonardo Fibonacci Pisano¹, 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 RECFCBO( $n - 1$ ) + RECFCBO( $n - 2$ )
```

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 RECFCBO, 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 $T(n) = 2F_{n+1} - 1$. 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 RECFCBO 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 RECFCBO(1) (which returns 1) exactly F_n times. A quick inductive argument implies that RECFCBO(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.

¹literally, “Leonardo, son of Bonacci, of Pisa”

3.2 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
       $F[n] \leftarrow \text{MEMFIBO}(n - 1) + \text{MEMFIBO}(n - 2)$ 
    return  $F[n]$ 
```

If we actually trace through the recursive calls made by `MEMFIBO`, we find that the array $F[]$ is filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. Once we see this pattern, we can replace the recursion with a simple for-loop that fills the array in 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 $F_{-1} = 1$ so that `ITERFIBO2(0)` returns the correct value 0.)

Even this algorithm can be improved further. 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 $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ 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 fact. So if we want the n th Fibonacci number, we just have to compute the n th power of the matrix $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$.

If we use repeated squaring, computing the n th power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$ 2×2 matrix multiplications, each of which reduces to 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?

3.3 Uh... wait a minute.

Well, not exactly. Fibonacci numbers grow exponentially fast. The n th 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. If we use fast Fourier transforms, multiplying two n -digit numbers takes $O(n \log n)$ time. Thus, 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?) Our matrix algorithm really is faster than our iterative algorithm, but not exponentially faster.

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.

3.4 The Pattern: Smart Recursion

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

1. **Formulate the problem recursively.** Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hard part.
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 intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

- (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input? For example, the argument to REC_{FIBO} is always an integer between 0 and n .
- (b) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.
- (c) **Analyze running time and space.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2} , we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.
- (d) **Identify dependencies between subproblems.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.
- (e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. *Be careful!*
- (f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence.

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* in some kind of array or table. Many algorithms students make the mistake of focusing on the table (because tables are easy and familiar) instead of the *much* more important (and difficult) task of finding a correct recurrence.

Dynamic programming is not about filling in tables; it's about smart recursion.

As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recursion is incorrect, nothing works.

3.5 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:

FOOD → MOD → MOND → MONED → 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.

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. Our 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.

So let's recursively define the edit distance between two strings $A[1..m]$ and $B[1..n]$, which we denote by $Edit(A[1..m], B[1..n])$. 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, the edit distance is equal to $Edit(A[1..m-1], B[1..n]) + 1$. The $+1$ is the cost of the final insertion, and the recursive expression gives the minimum cost for the other columns.
- **Deletion:** The last entry in the top row is empty. In this case, the edit distance is equal to $Edit(A[1..m], B[1..n-1]) + 1$. The $+1$ is the cost of the final deletion, and the recursive expression gives the minimum cost for the other columns.
- **Substitution:** Both rows have characters in the last column. If the characters are the same, the substitution is free, so the edit distance is equal to $Edit(A[1..m-1], B[1..n-1])$. If the characters are different, then the edit distance is equal to $Edit(A[1..m-1], B[1..n-1]) + 1$.

The edit distance between A and B is the smallest of these three possibilities:³

$$Edit(A[1..m], B[1..n]) = \min \left\{ \begin{array}{l} Edit(A[1..m-1], B[1..n]) + 1 \\ Edit(A[1..m], B[1..n-1]) + 1 \\ Edit(A[1..m-1], B[1..n-1]) + [A[m] \neq B[n]] \end{array} \right\}$$

This recurrence has two easy base cases. The only way to convert the empty string into a string of n characters is by performing n insertions. Similarly, the only way to convert a string of m characters into the empty string is with m deletions. Thus, if ϵ denotes the empty string, we have

$$Edit(A[1..m], \epsilon) = m, \quad Edit(\epsilon, B[1..n]) = n.$$

Both of these expressions imply the trivial base case $Edit(\epsilon, \epsilon) = 0$.

³Once 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.

Now notice that the arguments to our recursive subproblems are always *prefixes* of the original strings A and B . Thus, we can simplify our notation considerably by using the lengths of the prefixes, instead of the prefixes themselves, as the arguments to our recursive function. So let's write $Edit(i, j)$ as shorthand for $Edit(A[1..i], B[1..j])$. This function satisfies the following recurrence:

$$Edit(i, j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i - 1, j) + 1, \\ Edit(i, j - 1) + 1, \\ Edit(i - 1, j - 1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

The edit distance between the original strings A and B is just $Edit(m, n)$.

This recurrence translates directly into a recursive algorithm. Just out of curiosity, we can analyze the running time of this algorithm by solving the following recurrence:⁴

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

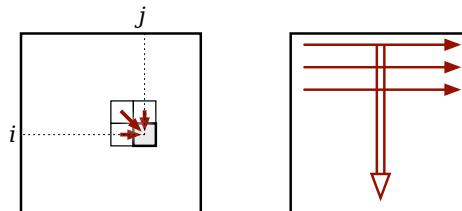
I don't know of a general closed-form solution for this mess, but we can derive an upper bound by defining a new function

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

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 dramatically reduce the running time of this algorithm down by memoization. Because each recursive subproblem can be identified by two indices i and j , we can store the intermediate values in a two-dimensional array $Edit[0..m, 0..n]$. Note that the index ranges start at zero to accommodate the base cases. Since there are $\Theta(mn)$ entries in the table, our memoized algorithm uses $\Theta(mn)$ space. Since each entry in the table can be computed in $\Theta(1)$ time once we know its predecessors, our memoized algorithm runs in $\Theta(mn)$ time.

It's not immediately clear what order the recursive algorithm fills the rest of the table; all we can tell from the recurrence is that each entry $Edit[i, j]$ is filled in *after* the neighboring entries directly above, directly to the left, and both above and to the left. But just this partial information is enough to give us an explicit evaluation order. If we fill in our table in the standard row-major order—row by row from top down, each row from left to right—then whenever we reach an entry in the table, the entries it depends on are already available.



Dependencies in the memoization table for edit distance, and a legal evaluation order

⁴You can skip this step. Since we're going to memoize this algorithm, the running time of the non-memoized algorithm is not particularly important. In particular, we won't ask you do this on homeworks or exams.

Putting everything together, we obtain the following dynamic programming algorithm:

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

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 post-processing 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
A	0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9								
L	1	0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8							
T	2	1	0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8						
R	3	2	1	1 → 2 → 3 → 4 → 5 → 6 → 7 → 8					
U	4	3	2	2	2 → 3 → 4 → 5 → 6 → 7 → 8				
I	5	4	3	3	3	3 → 4 → 5 → 6 → 7 → 8			
S	6	5	4	4	4	4 → 3 → 4 → 5 → 6 → 7 → 8			
T	7	6	5	5	5	5 → 4 → 4 → 5 → 6 → 7 → 8			
I	8	7	6	6	6	6 → 5 → 4 → 4 → 5 → 6 → 7 → 8			
C	9	8	7	7	7	7 → 6 → 5 → 4 → 4 → 5 → 6 → 7 → 8			
	10	9	8	8	8	8 → 7 → 6 → 5 → 4 → 4 → 5 → 6 → 7 → 8			

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

3.6 Warning: Greed is Stupid

If we're 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. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

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 don't 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.

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 almost never work!
Use dynamic programming instead!

What, never? No, never! What, *never*? Well... hardly ever.⁵

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 several greedy algorithms later in semester.

3.7 Dynamic Programming on Trees

So far, all of our dynamic programming example use a multidimensional array to store the results of recursive subproblems. However, as the next example shows, this is not always the most appropriate data structure to use.

A *independent set* in a graph is a subset of the vertices that have no edges between them. Finding the largest independent set in an arbitrary graph is extremely hard; in fact, this is one of the canonical NP-hard problems described in another lecture note. But from some special cases of graphs, we can find the largest independent set efficiently. In particular, when the input graph is a tree (a connected and acyclic graph) with n vertices, we can compute the largest independent set in $O(n)$ time.

In the recursion notes, we saw a recursive algorithm for computing the size of the largest independent set in an arbitrary graph:

```

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

⁵Greedy hardly ever ever works! 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*!

Here, $N(v)$ denotes the *neighborhood* of v : the set containing v and all of its neighbors. As we observed in the other lecture notes, this algorithm has a worst-case running time of $O(2^n \text{poly}(n))$, where n is the number of vertices in the input graph.

Now suppose we require that the input graph is a tree; we will call this tree T instead of G from now on. We need to make a slight change to the algorithm to make it truly recursive. The subgraphs $T \setminus \{v\}$ and $T \setminus N(v)$ are forests, which may have more than one component. But the largest independent set in a disconnected graph is just the union of the largest independent sets in its components, so we can separately consider each tree in these forests. Fortunately, this has the added benefit of making the recursive algorithm more efficient, especially if we can choose the node v such that the trees are all significantly smaller than T . Here is the modified algorithm:

```
MAXIMUMINDSETSIZE( $T$ ):
  if  $T = \emptyset$ 
    return 0
   $v \leftarrow$  any node in  $T$ 
   $withv \leftarrow 1$ 
  for each tree  $T'$  in  $T \setminus N(v)$ 
     $withv \leftarrow withv + \text{MAXIMUMINDSETSIZE}(T')$ 
   $withoutv \leftarrow 0$ 
  for each tree  $T'$  in  $T \setminus \{v\}$ 
     $withoutv \leftarrow withoutv + \text{MAXIMUMINDSETSIZE}(T')$ 
  return  $\max\{withv, withoutv\}.$ 
```

Now let's try to memoize this algorithm. Each recursive subproblem considers a subtree (that is, a connected subgraph) of the original tree T . Unfortunately, a single tree T can have exponentially many subtrees, so we seem to be doomed from the start!

Fortunately, there's a degree of freedom that we have not yet exploited: *We get to choose the vertex v .* We need a recipe—an algorithm!—for choosing v in each subproblem that limits the number of different subproblems the algorithm considers. To make this work, we impose some additional structure on the original input tree. Specifically, we declare one of the vertices of T to be the *root*, and we orient all the edges of T away from that root. Then we let v be the root of the input tree; this choice guarantees that each recursive subproblem considers a *rooted* subtree of T . Each vertex in T is the root of exactly one subtree, so now the number of distinct subproblems is exactly n . We can further simplify the algorithm by only passing a single node instead of the entire subtree:

```
MAXIMUMINDSETSIZE( $v$ ):
   $withv \leftarrow 1$ 
  for each grandchild  $x$  of  $v$ 
     $withv \leftarrow withv + \text{MAXIMUMINDSETSIZE}(x)$ 
   $withoutv \leftarrow 0$ 
  for each child  $w$  of  $v$ 
     $withoutv \leftarrow withoutv + \text{MAXIMUMINDSETSIZE}(w)$ 
  return  $\max\{withv, withoutv\}.$ 
```

What data structure should we use to store intermediate results? The most natural choice is the tree itself! Specifically, for each node v , we store the result of $\text{MAXIMUMINDSETSIZE}(v)$ in a new field $v.\text{MIS}$. (We could use an array, but then we'd have to add a new field to each node anyway, pointing to the corresponding array entry. Why bother?)

What's the running time of the algorithm? The non-recursive time associated with each node v is proportional to the number of children and grandchildren of v ; this number can be very different from

one vertex to the next. But we can turn the analysis around: Each vertex contributes a constant amount of time to its parent and its grandparent! Since each vertex has at most one parent and at most one grandparent, the total running time is $O(n)$.

What's a good order to consider the subproblems? The subproblem associated with any node v depends on the subproblems associated with the children and grandchildren of v . So we can visit the nodes in any order, provided that all children are visited before their parent. In particular, we can use a straightforward post-order traversal.

Here is the resulting dynamic programming algorithm. Yes, it's still recursive. I've swapped the evaluation of the with- v and without- v cases; we need to visit the kids first anyway, so why not consider the subproblem that depends directly on the kids first?

```
MAXIMUMINDSETSIZE( $v$ ):
     $withoutv \leftarrow 0$ 
    for each child  $w$  of  $v$ 
         $withoutv \leftarrow withoutv + \text{MAXIMUMINDSETSIZE}(w)$ 
     $withv \leftarrow 1$ 
    for each grandchild  $x$  of  $v$ 
         $withv \leftarrow withv + x.\text{MIS}$ 
     $v.\text{MIS} \leftarrow \max\{withv, withoutv\}$ 
    return  $v.\text{MIS}$ 
```

Another option is to store *two* values for each rooted subtree: the size of the largest independent set *that includes the root*, and the size of the largest independent set *that excludes the root*. This gives us an even simpler algorithm, with the same $O(n)$ running time.

```
MAXIMUMINDSETSIZE( $v$ ):
     $v.\text{MISno} \leftarrow 0$ 
     $v.\text{MISyes} \leftarrow 1$ 
    for each child  $w$  of  $v$ 
         $v.\text{MISno} \leftarrow v.\text{MISno} + \text{MAXIMUMINDSETSIZE}(w)$ 
         $v.\text{MISyes} \leftarrow v.\text{MISyes} + w.\text{MISno}$ 
    return  $\max\{v.\text{MISyes}, v.\text{MISno}\}$ 
```

3.8 Optimal Binary Search Trees

In an earlier lecture, 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:

$$\text{OptCost}(f[1..n]) = \min_{1 \leq r \leq n} \left\{ \text{OptCost}(f[1..r-1]) + \sum_{i=1}^n f[i] + \text{OptCost}(f[r+1..n]) \right\}$$

To put this recurrence in more standard form, fix the frequency array f , and let $\text{OptCost}(i, j)$ 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

$$OptCost(i, j) = \begin{cases} 0 & \text{if } j < i \\ F(i, j) + \min_{i \leq r \leq j} (OptCost(i, r - 1) + OptCost(r + 1, j)) & \text{otherwise} \end{cases}$$

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

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

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

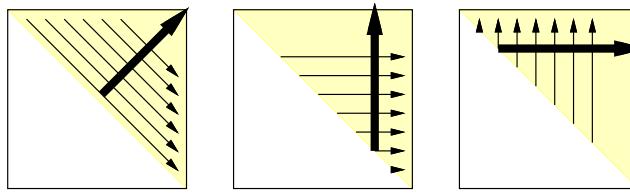
Each entry $OptCost[i, j]$ depends on all entries $OptCost[i, r - 1]$ and $OptCost[r + 1, j]$ with $i \leq k \leq j$. In other words, every entry in the table depends on all the entries directly to the left or directly below. In order to fill the table efficiently, we must choose an order that computes all those entries before $OptCost[i, j]$. 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 $OptCost[i, i - 1]$. The complete algorithm looks like this:

```
OPTIMALSEARCHTREE( $f[1..n]$ ):
INITF( $f[1..n]$ )
for  $i \leftarrow 1$  to  $n$ 
   $OptCost[i, i - 1] \leftarrow 0$ 
  for  $d \leftarrow 0$  to  $n - 1$ 
    for  $i \leftarrow 1$  to  $n - d$ 
      COMPUTEOPTCOST( $i, i + d$ )
return  $OptCost[1, n]$ 
```

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  $i \leftarrow n$  down to 1
     $OptCost[i, i - 1] \leftarrow 0$ 
    for  $j \leftarrow i$  to  $n$ 
      COMPUTE $OPTCOST(i, j)$ 
  return  $OptCost[1, n]$ 
```

```
OPTIMALSEARCHTREE3( $f[1..n]$ ):
  INITF( $f[1..n]$ )
  for  $j \leftarrow 0$  to  $n$ 
     $OptCost[j + 1, j] \leftarrow 0$ 
    for  $i \leftarrow j$  down to 1
      COMPUTE $OPTCOST(i, j)$ 
  return  $OptCost[1, n]$ 
```



Three different evaluation orders for the table $OptCost[i, j]$.

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

We could have predicted these space and time bounds directly from the original recurrence.

$$OptCost(i, j) = \begin{cases} 0 & \text{if } j = i - i \\ F(i, j) + \min_{i \leq r \leq j} (OptCost(i, r - 1) + OptCost(r + 1, j)) & \text{otherwise} \end{cases}$$

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.

3.9 More Examples

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

3.9.1 Subset Sum

Recall 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

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

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 $O(mn)$ 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 an algorithm that runs in $O(nT)$ 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( $X[1..n], T$ ):
   $S[n+1, 0] \leftarrow \text{TRUE}$ 
  for  $t \leftarrow 1$  to  $T$ 
     $S[n+1, t] \leftarrow \text{FALSE}$ 
  for  $i \leftarrow n$  downto 1
     $S[i, 0] = \text{TRUE}$ 
    for  $t \leftarrow 1$  to  $X[i] - 1$ 
       $S[i, t] \leftarrow S[i+1, t]$     «Avoid the case  $t < 0$ »
      for  $t \leftarrow X[i]$  to  $T$ 
         $S[i, t] \leftarrow S[i+1, t] \vee S[i+1, t-X[i]]$ 
  return  $S[1, T]$ 
```

This direct algorithm clearly always uses $O(nT)$ 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!

3.9.2 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:

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

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»
```

⁶Even though *SubsetSum* is NP-complete, this bound does not imply that P=NP, because T is not necessarily bounded by a polynomial function of the input size.

As predicted, this algorithm clearly uses $O(n^2)$ 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.

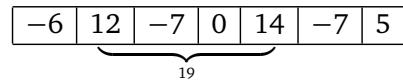
$$L'(i) = 1 + \max \{L'(j) \mid j > i \text{ and } A[j] > A[i]\}$$

Here, I'm assuming that $\max \emptyset = 0$, 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] = -∞           «Add a sentinel»
    for i ← n downto 0
        L'[i] ← 1
        for j ← i + 1 to n
            if A[j] > A[i] and 1 + L'[j] > L'[i]
                L'[i] ← 1 + L'[j]
    return L'[0] - 1      «Don't count the sentinel»
```

Exercises

- Suppose you are given an array $A[1..n]$ of integers. Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray $A[i..j]$. For example, if the array contains the numbers $(-6, 12, -7, 0, 14, -7, 5)$, then the largest sum of any contiguous subarray is $19 = 12 - 7 + 0 + 14$.



- This series of exercises asks you to develop efficient algorithms to find optimal subsequences of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings C, DAMN, YAIOAI, and DYNAMICPROGRAMMING are all subsequences of the sequence DYNAMICPROGRAMMING.
 - Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A common subsequence of A and B is another sequence that is a subsequence of both A and B . Describe an efficient algorithm to compute the length of the longest common subsequence of A and B .
 - Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A common supersequence of A and B is another sequence that contains both A and B as subsequences. Describe an efficient algorithm to compute the length of the shortest common supersequence of A and B .
 - Call a sequence $X[1..n]$ of numbers oscillating if $X[i] < X[i + 1]$ for all even i , and $X[i] > X[i + 1]$ for all odd i . Describe an efficient algorithm to compute the length of the longest oscillating subsequence of an arbitrary array A of integers.

- (d) Describe an efficient algorithm to compute the length of the shortest oscillating supersequence of an arbitrary array A of integers.
- (e) Call a sequence $X[1..n]$ of numbers *accelerating* if $2 \cdot X[i] < X[i - 1] + X[i + 1]$ for all i . Describe an efficient algorithm to compute the length of the longest accelerating subsequence of an arbitrary array A of integers.
- *(f) Recall that a sequence $X[1..n]$ of numbers is *increasing* if $X[i] < X[i + 1]$ for all i . Describe an efficient algorithm to compute the length of the *longest common increasing subsequence* of two given arrays of integers. For example, $\langle 1, 4, 5, 6, 7, 9 \rangle$ is the longest common increasing subsequence of the sequences $\langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3 \rangle$ and $\langle 1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5 \rangle$.
3. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.
- Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)
- (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.
- (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.
4. A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

- (a) Describe and analyze an algorithm to find the length of the longest *subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11.
- (b) Any string can be decomposed into a sequence of palindromes. For example, the string BUBBASEESABANANA ('Bubba sees a banana.') can be broken into palindromes in the following ways (and many others):

```

        BUB + BASESAB + ANANA
        B + U + BB + A + SEES + ABA + NAN + A
        B + U + BB + A + SEES + A + B + ANANA
        B + U + B + B + A + S + E + E + S + A + B + A + N + A + N + A
    
```

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm would return the integer 3.

5. Suppose you have a subroutine `QUALITY` that can compute the ‘quality’ of any given string $A[1..k]$ in time $O(k)$. For example, the quality of a string might be 1 if the string is a Québécois curse word, and 0 otherwise.

Given an arbitrary input string $T[1..n]$, we would like to break it into contiguous substrings, such that the total quality of all the substrings is as large as possible. For example, the string SAINTCIBIOREDESACRAMENTDECRISSÉ can be decomposed into the substrings SAINT + CIBIORE + DE + SACRAMENT + DE + CRISSE, of which three (or possibly four) are *sacres*.

Describe an algorithm that breaks a string into substrings of maximum total quality, using the `QUALITY` subroutine.

6. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..m, 1..n]$ whose entries are all 0 or 1. A *solid block* is a subarray of the form $M[i..i', j..j']$ in which every bit is equal to 1. Describe and analyze an efficient algorithm to find a solid block in M with maximum area.

7. Consider two horizontal lines ℓ_1 and ℓ_2 in the plane. Suppose we are given (the x -coordinates of) n distinct points a_1, a_2, \dots, a_n on ℓ_1 and n distinct points b_1, b_2, \dots, b_n on ℓ_2 . (The points a_i and b_j are *not* necessarily indexed in order from left to right, or in the same order.) Design an algorithm to compute the largest set S of non-intersecting line segments satisfying to the following restrictions:

- (a) Each segment in S connects some point a_i to the corresponding point b_i .
 - (b) No two segments in S intersect.
8. You are a bus driver with a soda fountain machine in the back and a bus full of very hyper students, who are drinking more soda as they ride along the highway. Your goal is to drop the students off as quickly as possible. More specifically, each minute that a student is on your bus, he drinks another ounce of soda. Your goal is to drop the students off quickly, so that in total they drink as little soda as possible.

You know how many students will get off of the bus at each exit. Your bus begins partway along the highway (probably not at either end), and moves at a constant speed of 37.4 miles per hour. You must drive the bus along the highway; however, you may drive forward to one exit then backward to an exit in the other direction, switching as often as you like. (You can stop the bus, drop off students, and turn around instantaneously.)

Describe an efficient algorithm to drop the students off so that they drink as little soda as possible. The input to the algorithm should be: the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (which you may assume is an exit).

9. In a previous life, you worked as a cashier in the lost Antarctic colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency

of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.⁷

- (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.
 - (b) Describe and analyze a recursive algorithm that computes, given an integer k , the minimum number of bills needed to make k Dream Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)
 - (c) Describe a dynamic programming algorithm that computes, given an integer k , the minimum number of bills needed to make k Dream Dollars. (This one needs to be fast.)
10. What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basket-weaving! The World Champions will be decided by a best-of- $2n - 1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requiring four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1 - p$ that Urbana will win.

Let $P(i, j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0, j) = 1$ for any j , because Champaign needs no more victories to win. Similarly, $P(i, 0) = 0$ for any i , as Champaign cannot possibly win if Urbana already has. $P(0, 0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q , so

$$P(i, j) = p \cdot P(i - 1, j) + q \cdot P(i, j - 1)$$

for any $i \geq 1$ and $j \geq 1$.

Describe and analyze an efficient algorithm that computes the probability that Champaign will win the series (that is, calculate $P(n, n)$), given the parameters n , p , and q as input.

11. *Vankin's Mile* is a solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

For example, given the grid below, the player can score $8 - 6 + 7 - 3 + 4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)

⁷For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see <http://www.dream-dollars.com>.

-1	7	-8	10	-5
-4	-9	8	-6	0
5	-2	-6	-6	7
-7	4	7 \Rightarrow -3	-3	-3
7	1	-6	4	-9

Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

12. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, ‘bananaanananas’ is a shuffle of ‘banana’ and ‘ananas’ in several different ways.

$$\text{banana}_\text{ananas} \quad \text{ban}_\text{ana}_\text{ana}_\text{nas} \quad \text{b}_\text{an}_\text{an}_\text{a}_\text{na}_\text{na}_\text{s}$$

The strings ‘prodgyrnamatammiiincg’ and ‘dyprongarmammicing’ are both shuffles of ‘dynamic’ and ‘programming’:

$$\text{pro}^d_\text{g} \text{y}^r \text{n}^m \text{a}^m \text{m}^i \text{i}^n \text{c}^g \quad \text{dy}^p_\text{p} \text{r}^n \text{o}^g \text{a}^r \text{m}^m \text{a}^m \text{m}^i \text{i}^c \text{n}^g \text{c}^i \text{i}^n \text{g}^g$$

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B .

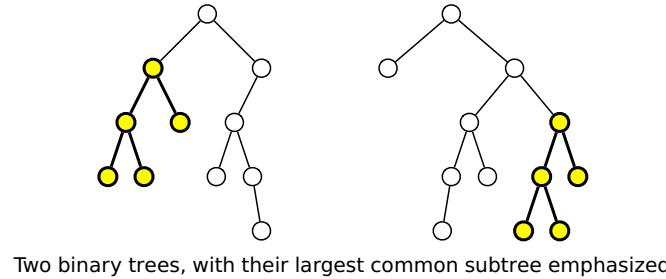
13. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the i th word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of white-space between any two words on the same line.

Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j , then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^j p_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

14. Let P be a set of points in the plane in *convex position*. Intuitively, if a rubber band were wrapped around the points, then every point would touch the rubber band. More formally, for any point p in P , there is a line that separates p from the other points in P . Moreover, suppose the points are indexed $P[1], P[2], \dots, P[n]$ in counterclockwise order around the ‘rubber band’, starting with the leftmost point $P[1]$.

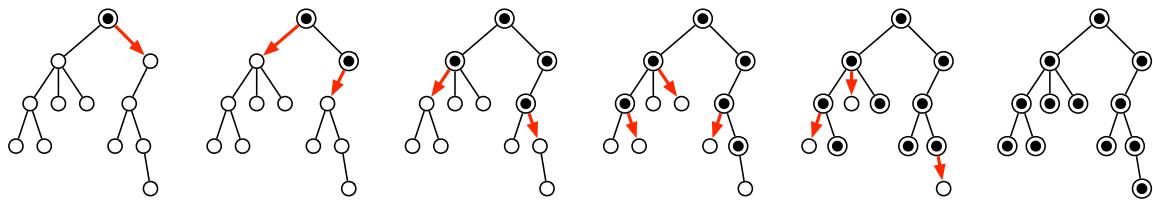
This problem asks you to solve a special case of the traveling salesman problem, where the salesman must visit every point in P , and the cost of moving from one point $p \in P$ to another point $q \in P$ is the Euclidean distance $|pq|$.

- (a) Describe a simple algorithm to compute the shortest *cyclic* tour of P .
- (b) A *simple* tour is one that never crosses itself. Prove that the shortest tour of P must be simple.
- (c) Describe and analyze an efficient algorithm to compute the shortest tour of P that starts at the leftmost point $P[1]$ and ends at the rightmost point $P[r]$.
15. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \text{poly}(n))$ time. Given an undirected n -vertex graph G with weighted edges, your algorithm should return the weight of the lightest cycle in G that visits every vertex exactly once, or ∞ if G has no such cycles. [Hint: The obvious recursive algorithm takes $O(n!)$ time.]
16. Recall that a *subtree* of a (rooted, ordered) binary tree T consists of a node and all its descendants. Design and analyze an efficient algorithm to compute the **largest common subtree** of two given binary trees T_1 and T_2 ; this is the largest subtree of T_1 that is isomorphic to a subtree in T_2 . The contents of the nodes are irrelevant; we are only interested in matching the underlying combinatorial structure.



Two binary trees, with their largest common subtree emphasized

17. Suppose we need to distribute a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. Design an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes.



A message being distributed through a tree in five rounds.

18. A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how ‘fun’ the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it’s her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the ‘fun’ ratings of the guests.

- *19. Scientists have branched out from the bizarre planet of Yggdrasil to study the vodes which have settled on Ygdrasil's moon, Xryltcon. All vodes on Xryltcon are descended from the first vode to arrive there, named George. Each vode has a color, either cyan, magenta, or yellow, but breeding patterns are *not* the same as on Yggdrasil; every vode, regardless of color, has either two children (with arbitrary colors) or no children.

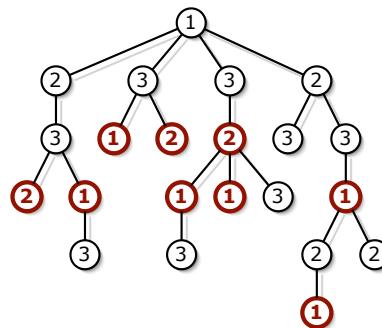
George and all his descendants are alive and well, and they are quite excited to meet the scientists who wish to study them. Unsurprisingly, these vodes have had some strange mutations in their isolation on Xryltcon. Each vode has a *weirdness* rating; weirder vodes are more interesting to the visiting scientists. (Some vodes even have negative weirdness ratings; they make other vodes more boring just by standing next to them.)

Also, Xryltconian society is strictly governed by a number of sacred cultural traditions.

- No cyan vode may be in the same room as its non-cyan children (if it has any).
- No magenta vode may be in the same room as its parent (if it has one).
- Each yellow vode must be attended at all times by its grandchildren (if it has any).
- George must be present at any gathering of more than fifty vodes.

The scientists have exactly one chance to study a group of vodes in a single room. You are given the family tree of all the vodes on Xryltcon, along with the weirdness value of each vode. Design and analyze an efficient algorithm to decide which vodes the scientists should invite to maximize the sum of the weirdness values of the vodes in the room. Be careful to respect all of the vodes' cultural taboos.

20. Oh, no! You have been appointed as the gift czar for Giggle, Inc.'s annual mandatory holiday party! The president of the company, who is certifiably insane, has declared that *every* Giggle employee must receive one of three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy. How do you decide what gifts everyone gets if you want to minimize the number of people that get fired?



A tree labeling with cost 9. Bold nodes have smaller labels than their parents.

This is *not* the optimal labeling for this tree.

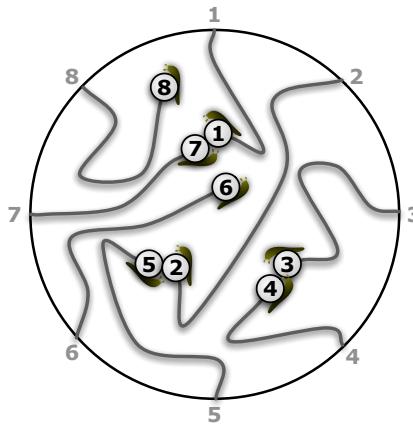
More formally, suppose you are given a rooted tree T , representing the company hierarchy. You want to label each node in T with an integer 1, 2, or 3, such that every node has a different

label from its parent.. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T . (Your algorithm does *not* have to compute the actual best labeling—just its cost.)

21. Every year, as part of its annual meeting, the Antarctic Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n . During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

For every pair of snails, the Antarctic SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.



The end of a typical Antarctic SLUG race. Snails 6 and 8 never find mates.

The organizers must pay $M[3, 4] + M[2, 5] + M[1, 7]$.

22. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{A, C, G, U\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule.

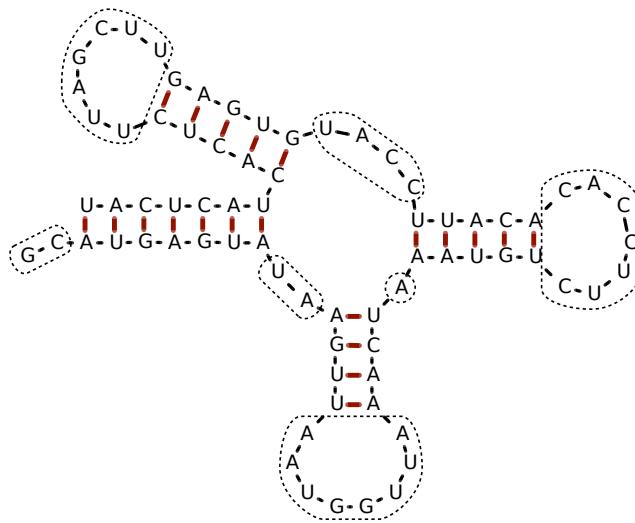
We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ **overlap** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

- Each base can be paired with at most one other base.
- Only A-U pairs and C-G pairs can bond.
- Pairs of the form $(i, i + 1)$ and $(i, i + 2)$ cannot bond.
- Overlapping base pairs cannot bond.

The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.

- Describe and analyze an algorithm that computes the maximum possible *number* of bonded base pairs in a secondary structure for a given RNA sequence.
- A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths.⁸ Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.



Example RNA secondary structure with 21 base pairs, indicated by heavy red lines. Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$

- *23. Let $\mathcal{A} = \{A_1, A_2, \dots, A_n\}$ be a finite set of strings over some fixed alphabet Σ . An *edit center* for \mathcal{A} is a string $C \in \Sigma^*$ such that the maximum edit distance from C to any string in \mathcal{A} is as small as possible. The *edit radius* of \mathcal{A} is the maximum edit distance from an edit center to a string in \mathcal{A} . A set of strings may have several edit centers, but its edit radius is unique.

$$\text{EditRadius}(\mathcal{A}) = \min_{C \in \Sigma^*} \max_{A \in \mathcal{A}} \text{Edit}(A, C) \quad \text{EditCenter}(\mathcal{A}) = \arg \min_{C \in \Sigma^*} \max_{A \in \mathcal{A}} \text{Edit}(A, C)$$

- Describe and analyze an efficient algorithm to compute the edit radius of three given strings.
- Describe and analyze an efficient algorithm to approximate the edit radius of an arbitrary set of strings within a factor of 2. (Computing the edit radius exactly is NP-hard unless the number of strings is fixed.)

⁸This score function has absolutely no connection to reality; I just made it up. Real RNA structure prediction requires *much* more complicated scoring functions.

- *24. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. An *digital subsequence* of D is an sequence of positive integers composed in the usual way from disjoint substrings of D . For example, 3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279 is an increasing digital subsequence of the first several digits of π :

3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the preceding example has length 12.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D . [Hint: Be careful about your computational assumptions. How long does it take to compare two k -digit numbers?]

*Ninety percent of science fiction is crud.
But then, ninety percent of everything is crud,
and it's the ten percent that isn't crud that is important.*

— [Theodore] Sturgeon's Law (1953)

*C Advanced Dynamic Programming Tricks

Dynamic programming is a powerful technique for efficiently solving recursive problems, but it's hardly the end of the story. In many cases, once we have a basic dynamic programming algorithm in place, we can make further improvements to bring down the running time or the space usage. We saw one example in the Fibonacci number algorithm. Buried inside the naïve iterative Fibonacci algorithm is a recursive problem—computing a power of a matrix—that can be solved more efficiently by dynamic programming techniques—in this case, repeated squaring.

C.1 Saving Space: Divide and Conquer

Just as we did for the Fibonacci recurrence, we can reduce the space complexity of our edit distance algorithm from $O(mn)$ to $O(m + n)$ by only storing the current and previous rows of the memoization table. This ‘sliding window’ technique provides an easy space improvement for most (but *not all*) dynamic programming algorithm.

Unfortunately, this technique seems to be useful only if we are interested in the *cost* of the optimal edit sequence, not if we want the optimal edit sequence itself. By throwing away most of the table, we apparently lose the ability to walk backward through the table to recover the optimal sequence.

However, if we throw away most of the rows in the table, it seems 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 edit sequence in $O(mn)$ time, using just $O(m + n)$ 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 $\text{Half}(i, j)$ 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 $\text{Half}(m, n)$ is the correct value of h . We can easily modify our earlier algorithm so that it computes $\text{Half}(m, n)$ at the same time as the edit distance $\text{Edit}(m, n)$, 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:

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

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.

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

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

Hirschberg's divide-and-conquer trick can be applied to almost any dynamic programming problem to obtain an algorithm to construct an optimal *structure* (in this case, the cheapest edit sequence) within the same space and time bounds as computing the *cost* of that optimal structure (in this case, edit distance). For this reason, we will almost always ask you for algorithms to compute the cost of some optimal structure, not the optimal structure itself.

C.2 Saving Time: Sparseness

In many applications of dynamic programming, we are faced with instances where almost every recursive subproblem will be resolved exactly the same way. We call such instances *sparse*. For example, we might want to compute the edit distance between two strings that have few characters in common, which means there are few “free” substitutions anywhere in the table. Most of the table has exactly the same structure. If we can reconstruct the entire table from just a few key entries, then why compute the entire table?

To better illustrate how to exploit sparseness, let's consider a simplification of the edit distance problem, in which substitutions are not allowed (or equivalently, where a substitution counts as two operations instead of one). Now our goal is to maximize the number of “free” substitutions, or equivalently, to find the *longest common subsequence* of the two input strings.

Let $LCS(i, j)$ denote the length of the longest common subsequence of two fixed strings $A[1..m]$ and $B[1..n]$. This function can be defined recursively as follows:

$$LCS(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i - 1, j - 1) & \text{if } A[i] = B[j] \\ \max \{LCS(i, j - 1), LCS(i - 1, j)\} & \text{otherwise} \end{cases}$$

This recursive definition directly translates into an $O(mn)$ -time dynamic programming algorithm.

Call an index pair (i, j) a *match point* if $A[i] = B[j]$. In some sense, match points are the only ‘interesting’ locations in the memoization table; given a list of the match points, we could easily reconstruct the entire table.

<< A L G O R I T H M S >>												
«	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	1	1	1	1	1	1	1	1	1
L	2	2	2	2	2	2	2	2	2	2	2	2
T	2	2	2	2	3	3	3	3	3	3	3	3
R	2	2	3	3	3	3	3	3	3	3	3	3
U	2	2	3	3	3	3	3	3	3	3	3	3
I	2	2	4	4	4	4	4	4	4	4	4	4
S	2	2	4	4	4	4	4	4	5	5	5	5
T	2	2	4	5	5	5	5	5	5	5	5	5
I	2	2	4	5	5	5	5	5	5	5	5	5
C	2	2	4	5	5	5	5	5	5	5	5	5
»	0	2	2	4	5	5	5	5	5	5	5	6

The *LCS* memoization table for ALGORITHMS and ALTRUISTIC; the brackets « and » are sentinel characters.

More importantly, we can compute the *LCS* function directly from the list of match points using the following recurrence:

$$LCS(i, j) = \begin{cases} 0 & \text{if } i = j = 0 \\ \max \{LCS(i', j') \mid A[i'] = B[j'] \text{ and } i' < i \text{ and } j' < j\} + 1 & \text{if } A[i] = B[j] \\ \max \{LCS(i', j') \mid A[i'] = B[j'] \text{ and } i' \leq i \text{ and } j' \leq j\} & \text{otherwise} \end{cases}$$

(Notice that the inequalities are strict in the second case, but not in the third.) To simplify boundary issues, we add unique sentinel characters $A[0] = B[0]$ and $A[m+1] = B[n+1]$ to both strings. This ensures that the sets on the right side of the recurrence equation are non-empty, and that we only have to consider match points to compute $LCS(m, n) = LCS(m+1, n+1) - 1$.

If there are K match points, we can actually compute them all in $O(m \log m + n \log n + K)$ time. Sort the characters in each input string, but remembering the original index of each character, and then essentially merge the two sorted arrays, as follows:

```

FINDMATCHES(A[1..m], B[1..n]):
    for i ← 1 to m: I[i] ← i
    for j ← 1 to n: J[j] ← j
    sort A and permute I to match
    sort B and permute J to match
    i ← 1; j ← 1
    while i < m and j < n
        if A[i] < B[j]
            i ← i + 1
        else if A[i] > B[j]
            j ← j + 1
        else
            ii ← i
            jj ← j
            while A[ii] = A[i]
                jj ← jj + 1
                while B[jj] = B[j]
                    report (I[ii], J[jj])
                    jj ← jj + 1
            ii ← ii + 1
            i ← ii; j ← jj

```

To efficiently evaluate our modified recurrence, we once again turn to dynamic programming. We consider the match points in lexicographic order—the order they would be encountered in a standard row-major traversal of the $m \times n$ table—so that when we need to evaluate $LCS(i, j)$, all match points (i', j') with $i' < i$ and $j' < j$ have already been evaluated.

```
SPARSELCS( $A[1..m], B[1..n]$ ):
   $Match[1..K] \leftarrow \text{FINDMATCHES}(A, B)$ 
   $Match[K + 1] \leftarrow (m + 1, n + 1)$     «Add end sentinel»
  Sort  $M$  lexicographically
  for  $k \leftarrow 1$  to  $K$ 
     $(i, j) \leftarrow Match[k]$ 
     $LCS[k] \leftarrow 1$                       «From start sentinel»
    for  $\ell \leftarrow 1$  to  $k - 1$ 
       $(i', j') \leftarrow Match[\ell]$ 
      if  $i' < i$  and  $j' < j$ 
         $LCS[k] \leftarrow \min\{LCS[k], 1 + LCS[\ell]\}$ 
  return  $LCS[K + 1] - 1$ 
```

The overall running time of this algorithm is $O(m \log m + n \log n + K^2)$. So as long as $K = o(\sqrt{mn})$, this algorithm is actually faster than naïve dynamic programming.

C.3 Saving Time: Monotonicity

Recall the optimal binary search tree problem from the previous lecture. Given an array $F[1..n]$ of access frequencies for n items, the problem is to compute the binary search tree that minimizes the cost of all accesses. A straightforward dynamic programming algorithm solves this problem in $O(n^3)$ time.

As for longest common subsequence problem, the algorithm can be improved by exploiting some structure in the memoization table. In this case, however, the relevant structure isn't in the table of costs, but rather in the table used to reconstruct the actual optimal tree. Let $OptRoot[i, j]$ denote the index of the root of the optimal search tree for the frequencies $F[i..j]$; this is always an integer between i and 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. More formally:

$$OptRoot[i, j - 1] \leq OptRoot[i, j] \leq OptRoot[i + 1, j] \text{ for all } i \text{ and } j.$$

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

FASTEROPTIMALSEARCHTREE($f[1..n]$): <pre> INITF($f[1..n]$) for $i \leftarrow 1$ downto n $OptCost[i, i - 1] \leftarrow 0$ $OptRoot[i, i - 1] \leftarrow i$ for $d \leftarrow 0$ to n for $i \leftarrow 1$ to n COMPUTECOSTANDROOT($i, i + d$) return $OptCost[1, n]$</pre>	COMPUTECOSTANDROOT(i, j): <pre> $OptCost[i, j] \leftarrow \infty$ for $r \leftarrow OptRoot[i, j - 1]$ to $OptRoot[i + 1, j]$ $tmp \leftarrow OptCost[i, r - 1] + OptCost[r + 1, j]$ if $OptCost[i, j] > tmp$ $OptCost[i, j] \leftarrow tmp$ $OptRoot[i, j] \leftarrow r$ $OptCost[i, j] \leftarrow OptCost[i, j] + F[i, j]$</pre>
---	---

It's not hard to see that the loop index r increases monotonically from 1 to n during each iteration of the *outermost* for loop of FASTEROPTIMALSEARCHTREE. Consequently, the total cost of all calls to COMPUTECOSTANDROOT is only $O(n^2)$.

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 Tucker¹ computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

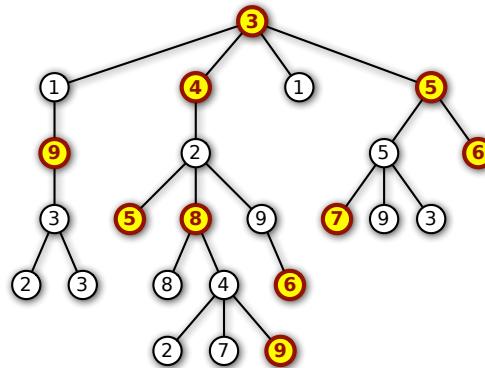
Exercises

1. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1..n]$ in $O(m \log m + n \log n + K^2)$ time, where K is the number of match points. [Hint: Use the FINDMATCHES algorithm on page 3 as a subroutine.]

2. (a) Describe an algorithm to compute the *longest increasing subsequence* of a string $X[1..n]$ in $O(n \log n)$ time.
(b) Using your solution to part (a) as a subroutine, describe an algorithm to compute the longest common subsequence of two strings $A[1..m]$ and $B[1..n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points.

3. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1..n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points. [Hint: Combine your answers for problems 1 and 2(b).]

4. Let T be an arbitrary rooted tree, where each vertex is labeled with a positive integer. A subset S of the nodes of T is *heap-ordered* if it satisfies two properties:
 - S contains a node that is an ancestor of every other node in S .
 - For any node v in S , the label of v is larger than the labels of any ancestor of v in S .



A heap-ordered subset of nodes in a tree.

- (a) Describe an algorithm to find the largest heap-ordered subset S of nodes in T that has the heap property in $O(n^2)$ time.

¹T. 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.

- (b) Modify your algorithm from part (a) so that it runs in $O(n \log n)$ time when T is either a linked list. [Hint: This special case is equivalent to a problem you've seen before.]
- *(c) Describe an algorithm to find the largest subset S of nodes in T that has the heap property, in $O(n \log n)$ time. [Hint: Find an algorithm to merge two sorted lists of lengths k and ℓ in $O(\log \binom{k+\ell}{k})$ time.]

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, “The Divine Afflatus”,
New York Evening Mail (November 16, 1917)

4 Greedy Algorithms

4.1 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 k th file is

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

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

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

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

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

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. $\mathbb{E}[\text{cost}(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ 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. \square

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

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

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. 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. $\Sigma\text{cost}(\pi)$ is minimized when $\frac{L[\pi(i)]}{F[\pi(i)]} \leq \frac{L[\pi(i+1)]}{F[\pi(i+1)]}$ for all i .

Proof: Suppose $L[\pi(i)]/F[\pi(i)] > L[\pi(i+1)]/F[\pi(i+1)]$ 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 total cost by $L[b]F[a] - L[a]F[a]$. But this change is an improvement, since

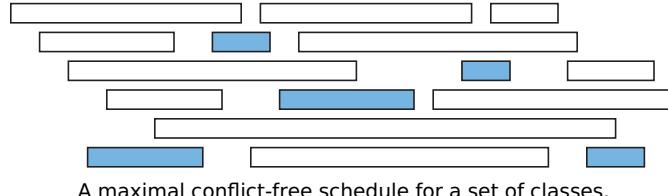
$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \implies L[b]F[a] - L[a]F[a] < 0.$$

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

4.2 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 $X \subseteq \{1, 2, \dots, n\}$ 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.



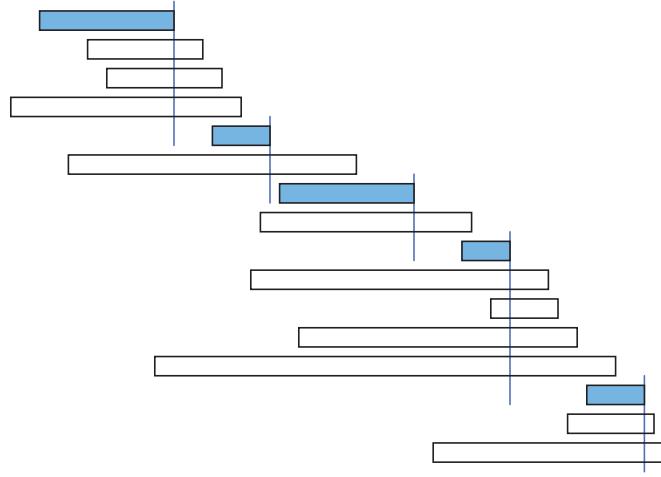
A maximal conflict-free schedule for a set of classes.

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 *later than* class 1 ends:

$$B_4 = \{i \mid 2 \leq i \leq n \text{ and } F[i] < S[1]\} \quad L_8 = \{i \mid 2 \leq i \leq n \text{ and } S[i] > F[1]\}$$

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 $\{2, 3, \dots, n\}$ 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  $X[1..count]$ 
```

¹But 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. \square

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 . \square

The proof might be easier to understand if we unroll the induction slightly.

Proof: Let $\langle g_1, g_2, \dots, g_k \rangle$ be the sequence of classes chosen by the greedy algorithm. Suppose we have a maximal conflict-free schedule of the form

$$\langle g_1, g_2, \dots, g_{j-1}, c_j, c_{j+1}, \dots, c_m \rangle,$$

where the classes c_i are different from the classes chosen by the greedy algorithm. By construction, the j th greedy choice g_j does not conflict with any earlier class g_1, g_2, \dots, 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}, \dots, c_m . Thus, the schedule

$$\langle g_1, g_2, \dots, g_{j-1}, g_j, c_{j+1}, \dots, c_m \rangle,$$

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 $\langle g_1, g_2, \dots, g_k, c_{k+1}, \dots, c_m \rangle$ 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. \square

4.3 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.

4.4 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 (· · ·) includes the code for E (·) 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

$$\sum_{i=1}^n f[i] \cdot \text{depth}(i).$$

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

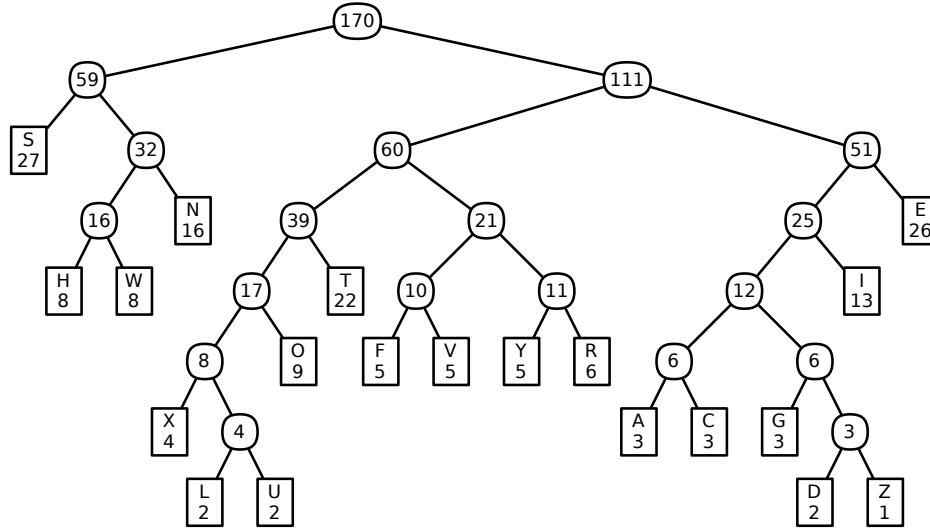
Huffman's algorithm picks out the two least frequent letters, breaking ties arbitrarily—in this case, say, Z and D—and merges them together into a single new character \square with frequency 3. This new character becomes an internal node in the code tree we are constructing, with Z and D as its children; 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	\square
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

²This looks almost exactly like the cost of a binary search tree, but the optimization problem is very different: code trees are **not** search trees!

³A. 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.

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:

1001 0100 1101 00 00 111 011 1001 111 011 110001 111 110001 10001 011 1001 110000 1101 ...
 T H I S S E N T E N C E C E C O N T A I ...

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, and suppose this tree has 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,

$$\text{cost}(T') = \text{cost}(T) - (f[a] - f[x])\Delta.$$

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. \square

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, $\text{depth}(i)$ 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.)

$$\begin{aligned} \text{cost}(T) &= \sum_{i=1}^n f[i] \cdot \text{depth}(i) \\ &= \sum_{i=3}^{n+1} f[i] \cdot \text{depth}(i) + f[1] \cdot \text{depth}(1) + f[2] \cdot \text{depth}(2) - f[n+1] \cdot \text{depth}(n+1) \\ &= \text{cost}(T') + f[1] \cdot \text{depth}(1) + f[2] \cdot \text{depth}(2) - f[n+1] \cdot \text{depth}(n+1) \\ &= \text{cost}(T') + (f[1] + f[2]) \cdot \text{depth}(T) - f[n+1] \cdot (\text{depth}(T) - 1) \\ &= \text{cost}(T') + f[1] + f[2] \end{aligned}$$

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. \square

To actually implement Huffman codes efficiently, we keep the characters in a min-heap, where the priority of each character is its frequency. We can 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  $2n - 1$ 
     $x \leftarrow \text{EXTRACTMIN}()$ 
     $y \leftarrow \text{EXTRACTMIN}()$ 
     $f[i] \leftarrow f[x] + f[y]$ 
     $L[i] \leftarrow x$ ;  $R[i] \leftarrow y$ 
     $P[x] \leftarrow i$ ;  $P[y] \leftarrow i$ 
    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  $L[v] = 0$ 
       $A[k] \leftarrow v$ 
       $k \leftarrow k + 1$ 
   $v \leftarrow 2n - 1$ 
```

Exercises

- Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The *size* of a tiling cover is just the number of intervals.

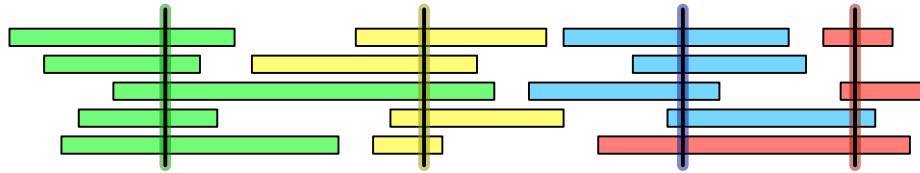
Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X . If you use a greedy algorithm, you must prove that it is correct.



A set of intervals. The seven shaded intervals form a tiling path.

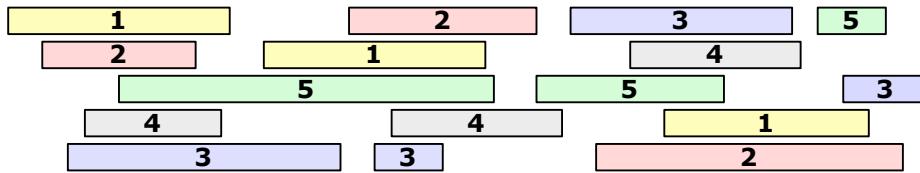
- Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P . Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X . Assume that your input consists of two arrays $X_L[1..n]$ and

$X_R[1..n]$, representing the left and right endpoints of the intervals in X . As usual, If you use a greedy algorithm, you must prove that it is correct.



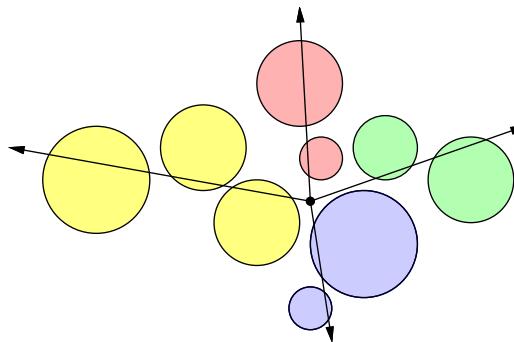
A set of intervals stabbed by four points (shown here as vertical segments)

- Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color X . Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, where $L[i]$ and $R[i]$ are the left and right endpoints of the i th interval. As usual, if you use a greedy algorithm, you must prove that it is correct.



A proper coloring of a set of intervals using five colors.

- Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic™ to pop all the balloons, without moving from your current location. The Zap-O-Matic™ shoots a high-powered laser beam, which pops all the balloons it hits. Since each shot requires enough energy to power a small country for a year, you want to fire as few shots as possible.



Nine balloons popped by 4 shots of the Zap-O-Matic™

The *minimum zap* problem can be stated more formally as follows. Given a set C of n circles in the plane, each specified by its radius and the (x, y) coordinates of its center, compute the minimum number of rays from the origin that intersect every circle in C . Your goal is to find an efficient algorithm for this problem.

- (a) Suppose it is possible to shoot a ray that does not intersect any balloons. Describe and analyze a greedy algorithm that solves the minimum zap problem in this special case. [Hint: See Exercise 2.]
- (b) Describe and analyze a greedy algorithm whose output is within 1 of optimal. That is, if m is the minimum number of rays required to hit every balloon, then your greedy algorithm must output either m or $m + 1$. (Of course, you must prove this fact.)
- (c) Describe an algorithm that solves the minimum zap problem in $O(n^2)$ time.
- *(d) Describe an algorithm that solves the minimum zap problem in $O(n \log n)$ time.

Assume you have a subroutine $\text{INTERSECTS}(r, c)$ that determines whether a ray r intersects a circle c in $O(1)$ time. It's not that hard to write this subroutine, but it's not the interesting part of the problem.

*The problem is that we attempt to solve the simplest questions cleverly, thereby rendering them unusually complex.
One should seek the simple solution.*

— Anton Pavlovich Chekhov (c. 1890)

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams

*D Matroids

D.1 Definitions

Many problems that can be correctly solved by greedy algorithms can be described in terms of an abstract combinatorial object called a *matroid*. Matroids were first described in 1935 by the mathematician Hassler Whitney as a combinatorial generalization of linear independence of vectors—‘matroid’ means ‘something sort of like a matrix’.

A matroid \mathcal{M} is a finite collection of finite sets that satisfies three axioms:

- **Non-emptiness:** The empty set is in \mathcal{M} . (Thus, \mathcal{M} is not itself empty.)
- **Heredity:** If a set X is an element of \mathcal{M} , then any subset of X is also in \mathcal{M} .
- **Exchange:** If X and Y are two sets in \mathcal{M} and $|X| > |Y|$, then there is an element $x \in X \setminus Y$ such that $Y \cup \{x\}$ is in \mathcal{M} .

The sets in \mathcal{M} are typically called *independent sets*; for example, we would say that any subset of an independent set is independent. The union of all sets in \mathcal{M} is called the *ground set*. An independent set is called a *basis* if it is not a proper subset of another independent set. The exchange property implies that every basis of a matroid has the same cardinality. The *rank* of a subset X of the ground set is the size of the largest independent subset of X . A subset of the ground set that is not in \mathcal{M} is called *dependent* (surprise, surprise). Finally, a dependent set is called a *circuit* if every proper subset is independent.

Most of this terminology is justified by Whitney’s original example:

Linear matroid: Let A be any $n \times m$ matrix. A subset $I \subseteq \{1, 2, \dots, n\}$ is independent if and only if the corresponding subset of columns of A is linearly independent.

The heredity property follows directly from the definition of linear independence; the exchange property is implied by an easy dimensionality argument. A basis in any linear matroid is also a basis (in the linear-algebra sense) of the vector space spanned by the columns of A . Similarly, the rank of a set of indices is precisely the rank (in the linear-algebra sense) of the corresponding set of column vectors.

Here are several other examples of matroids; some of these we will see again later. I will leave the proofs that these are actually matroids as exercises for the reader.

- **Uniform matroid $U_{k,n}$:** A subset $X \subseteq \{1, 2, \dots, n\}$ is independent if and only if $|X| \leq k$. Any subset of $\{1, 2, \dots, n\}$ of size k is a basis; any subset of size $k+1$ is a circuit.
- **Graphic/cycle matroid $\mathcal{M}(G)$:** Let $G = (V, E)$ be an arbitrary undirected graph. A subset of E is independent if it defines an acyclic subgraph of G . A basis in the graphic matroid is a spanning tree of G ; a circuit in this matroid is a cycle in G .

- **Cographic/cocycle matroid $\mathcal{M}^*(G)$:** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq E$ is independent if the complementary subgraph $(V, E \setminus I)$ of G is *connected*. A basis in this matroid is the complement of a spanning tree; a circuit in this matroid is a *cocycle*—a minimal set of edges that disconnects the graph.
- **Matching matroid:** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq V$ is independent if there is a matching in G that covers I .
- **Disjoint path matroid:** Let $G = (V, E)$ be an arbitrary *directed* graph, and let s be a fixed vertex of G . A subset $I \subseteq V$ is independent if and only if there are edge-disjoint paths from s to each vertex in I .

Now suppose each element of the ground set of a matroid \mathcal{M} is given an arbitrary non-negative weight. The **matroid optimization problem** is to compute a basis with maximum total weight. For example, if \mathcal{M} is the cycle matroid for a graph G , the matroid optimization problem asks us to find the maximum spanning tree of G . Similarly, if \mathcal{M} is the cocycle matroid for G , the matroid optimization problem seeks (the complement of) the *minimum* spanning tree.

The following natural greedy strategy computes a basis for any weighted matroid:

```

GREEDYBASIS( $\mathcal{M}, w$ ):
   $X[1..n] \leftarrow \bigcup \mathcal{M}$  (the ground set)
  sort  $X$  in decreasing order of weight  $w$ 
   $G \leftarrow \emptyset$ 
  for  $i \leftarrow 1$  to  $n$ 
    if  $G \cup \{X[i]\} \in \mathcal{M}$ 
      add  $X[i]$  to  $G$ 
  return  $G$ 
```

Suppose we can test in $F(n)$ whether a given subset of the ground set is independent. Then this algorithm runs in $O(n \log n + n \cdot F(n))$ time.

Theorem 1. *For any matroid \mathcal{M} and any weight function w , $\text{GREEDYBASIS}(\mathcal{M}, w)$ returns a maximum-weight basis of \mathcal{M} .*

Proof: Let $G = \{g_1, g_2, \dots, g_k\}$ be the independent set returned by $\text{GREEDYBASIS}(\mathcal{M}, w)$. If any other element could be added to G to get a larger independent set, the greedy algorithm would have added it. Thus, G is a basis.

For purposes of deriving a contradiction, suppose there is an independent set $H = \{h_1, h_2, \dots, h_\ell\}$ such that

$$\sum_{i=1}^k w(g_i) < \sum_{j=1}^\ell w(h_j).$$

Without loss of generality, we assume that H is a basis. The exchange property now implies that $k = \ell$.

Now suppose the elements of G and H are indexed in order of decreasing weight. Let i be the smallest index such that $w(g_i) < w(h_i)$, and consider the independent sets

$$G_{i-1} = \{g_1, g_2, \dots, g_{i-1}\} \quad \text{and} \quad H_i = \{h_1, h_2, \dots, h_{i-1}, h_i\}.$$

By the exchange property, there is some element $h_j \in H_i$ such that $G_{i-1} \cup \{h_j\}$ is an independent set. We have $w(h_j) \geq w(h_i) > w(g_i)$. Thus, the greedy algorithm considers *and rejects* the heavier element h_j before it considers the lighter element g_i . But this is impossible—the greedy algorithm accepts elements in decreasing order of weight. \square

We now immediately have a correct greedy optimization algorithm for *any* matroid. Returning to our examples:

- Linear matroid: Given a matrix A , compute a subset of vectors of maximum total weight that span the column space of A .
- Uniform matroid: Given a set of weighted objects, compute its k largest elements.
- Cycle matroid: Given a graph with weighted edges, compute its maximum spanning tree. In this setting, the greedy algorithm is better known as *Kruskal's algorithm*.
- Cocycle matroid: Given a graph with weighted edges, compute its minimum spanning tree.
- Matching matroid: Given a graph, determine whether it has a perfect matching.
- Disjoint path matroid: Given a directed graph with a special vertex s , find the largest set of edge-disjoint paths from s to other vertices.

The exchange condition for matroids turns out to be crucial for the success of this algorithm. A *subset system* is a finite collection \mathcal{S} of finite sets that satisfies the heredity condition—if $X \in \mathcal{S}$ and $Y \subseteq X$, then $Y \in \mathcal{S}$ —but not necessarily the exchange condition.

Theorem 2. *For any subset system \mathcal{S} that is **not** a matroid, there is a weight function w such that $\text{GREEDYBASIS}(\mathcal{S}, w)$ does **not** return a maximum-weight set in \mathcal{S} .*

Proof: Let X and Y be two sets in \mathcal{S} that violate the exchange property— $|X| > |Y|$, but for any element $x \in X \setminus Y$, the set $Y \cup \{x\}$ is not in \mathcal{S} . Let $m = |Y|$. We define a weight function as follows:

- Every element of Y has weight $m + 2$.
- Every element of $X \setminus Y$ has weight $m + 1$.
- Every other element of the ground set has weight zero.

With these weights, the greedy algorithm will consider and accept every element of Y , then consider and reject every element of X , and finally consider all the other elements. The algorithm returns a set with total weight $m(m + 2) = m^2 + 2m$. But the total weight of X is at least $(m + 1)^2 = m^2 + 2m + 1$. Thus, the output of the greedy algorithm is not the maximum-weight set in \mathcal{S} . \square

Recall the Applied Chaos scheduling problem considered in the previous lecture note. There is a natural subset system associated with this problem: A set of classes is independent if and only if not two classes overlap. (This is just the graph-theory notion of ‘independent set’!) This subset system is *not* a matroid, because there can be maximal independent sets of different sizes, which violates the exchange property. If we consider a *weighted* version of the class scheduling problem, say where each class is worth a different number of hours, Theorem 2 implies that the greedy algorithm will *not* always find the optimal schedule. (In fact, there’s an easy counterexample with only two classes!) However, Theorem 2 does *not* contradict the correctness of the greedy algorithm for the original *unweighted* problem, however; that problem uses a particularly lucky choice of weights (all equal).

D.2 Scheduling with Deadlines

Suppose you have n tasks to complete in n days; each task requires your attention for a full day. Each task comes with a *deadline*, the last day by which the job should be completed, and a *penalty* that you must pay if you do not complete each task by its assigned deadline. What order should you perform your tasks in to minimize the total penalty you must pay?

More formally, you are given an array $D[1..n]$ of deadlines and an array $P[1..n]$ of penalties. Each deadline $D[i]$ is an integer between 1 and n , and each penalty $P[i]$ is a non-negative real number. A *schedule* is a permutation of the integers $\{1, 2, \dots, n\}$. The scheduling problem asks you to find a schedule π that minimizes the following cost:

$$\text{cost}(\pi) := \sum_{i=1}^n P[i] \cdot [\pi(i) > D[i]].$$

This doesn't look anything like a matroid optimization problem. For one thing, matroid optimization problems ask us to find an optimal set; this problem asks us to find an optimal *permutation*. Surprisingly, however, this scheduling problem is actually a matroid optimization in disguise! For any schedule π , call tasks i such that $\pi(i) > D[i]$ *late*, and all other tasks *on time*. The following trivial observation is the key to revealing the underlying matroid structure.

The cost of a schedule is determined by the subset of tasks that are on time.

Call a subset X of the tasks *realistic* if there is a schedule π in which every task in X is on time. We can precisely characterize the realistic subsets as follows. Let $X(t)$ denote the subset of tasks in X whose deadline is on or before t :

$$X(t) := \{i \in X \mid D[i] \leq t\}.$$

In particular, $X(0) = \emptyset$ and $X(n) = X$.

Lemma 3. *Let $X \subseteq \{1, 2, \dots, n\}$ be an arbitrary subset of the n tasks. X is realistic if and only if $|X(t)| \leq t$ for every integer t .*

Proof: Let π be a schedule in which every task in X is on time. Let i_t be the t th task in X to be completed. On the one hand, we have $\pi(i_t) \geq t$, since otherwise, we could not have completed $t - 1$ other jobs in X before i_t . On the other hand, $\pi(i_t) \leq D[i]$, because i_t is on time. We conclude that $D[i_t] \geq t$, which immediately implies that $|X(t)| \leq t$.

Now suppose $|X(t)| \leq t$ for every integer t . If we perform the tasks in X in increasing order of deadline, then we complete all tasks in X with deadlines t or less by day t . In particular, for any $i \in X$, we perform task i on or before its deadline $D[i]$. Thus, X is realistic. \square

We can define a *canonical schedule* for any set X as follows: execute the tasks in X in increasing deadline order, and then execute the remaining tasks in any order. The previous proof implies that a set X is realistic if and only if every task in X is on time in the canonical schedule for X . Thus, our scheduling problem can be rephrased as follows:

Find a realistic subset X such that $\sum_{i \in X} P[i]$ is maximized.

So we're looking for optimal subsets after all.

Lemma 4. *The collection of realistic sets of jobs forms a matroid.*

Proof: The empty set is vacuously realistic, and any subset of a realistic set is clearly realistic. Thus, to prove the lemma, it suffices to show that the exchange property holds. Let X and Y be realistic sets of jobs with $|X| > |Y|$.

Let t^* be the largest integer such that $|X(t^*)| \leq |Y(t^*)|$. This integer must exist, because $|X(0)| = 0 \leq 0 = |Y(0)|$ and $|X(n)| = |X| > |Y| = |Y(n)|$. By definition of t^* , there are more tasks with deadline $t^* + 1$ in X than in Y . Thus, we can choose a task j in $X \setminus Y$ with deadline $t^* + 1$; let $Z = Y \cup \{j\}$.

Let t be an arbitrary integer. If $t \leq t^*$, then $|Z(t)| = |Y(t)| \leq t$, because Y is realistic. On the other hand, if $t > t^*$, then $|Z(t)| = |Y(t)| + 1 \leq |X(t)| < t$ by definition of t^* and because X is realistic. The previous lemma now implies that Z is realistic. This completes the proof of the exchange property. \square

This lemma implies that our scheduling problem is a matroid optimization problem, so the greedy algorithm finds the optimal schedule.

```

GREEDYSCHEDULE( $D[1..n], P[1..n]$ ):
    Sort  $P$  in increasing order, and permute  $D$  to match
     $j \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $n$ 
         $X[j+1] \leftarrow i$ 
        if  $X[1..j+1]$  is realistic
             $j \leftarrow j+1$ 
    return the canonical schedule for  $X[1..j]$ 

```

To turn this outline into a real algorithm, we need a procedure to test whether a given subset of jobs is realistic. Lemma 9 immediately suggests the following strategy to answer this question in $O(n)$ time.

```

REALISTIC?( $X[1..m], D[1..n]$ ):
    « $X$  is sorted by increasing deadline:  $i \leq j \implies D[X[i]] \leq D[X[j]]$ »
     $N \leftarrow 0$ 
     $j \leftarrow 0$ 
    for  $t \leftarrow 1$  to  $n$ 
        if  $D[X[j]] = t$ 
             $N \leftarrow N + 1$ ;  $j \leftarrow j + 1$ 
    «Now  $N = |X(t)|$ »
    if  $N > t$ 
        return FALSE
    return TRUE

```

If we use this subroutine, GREEDYSCHEDULE runs in $O(n^2)$ time. By using some appropriate data structures, the running time can be reduced to $O(n \log n)$; details are left as an exercise for the reader.

Exercises

1. Prove that for any graph G , the ‘graphic matroid’ $\mathcal{M}(G)$ is in fact a matroid. (This problem is really asking you to prove that Kruskal’s algorithm is correct!)
2. Prove that for any graph G , the ‘cographic matroid’ $\mathcal{M}^*(G)$ is in fact a matroid.
3. Prove that for any graph G , the ‘matching matroid’ of G is in fact a matroid. [Hint: What is the symmetric difference of two matchings?]

4. Prove that for any directed graph G and any vertex s of G , the resulting ‘disjoint path matroid’ of G is in fact a matroid. [Hint: This question is **much** easier if you’re already familiar with maximum flows.]
5. Let G be an undirected graph. A set of cycles $\{c_1, c_2, \dots, c_k\}$ in G is called *redundant* if every edge in G appears in an even number of c_i ’s. A set of cycles is *independent* if it contains no redundant subset. A maximal independent set of cycles is called a *cycle basis* for G .
 - (a) Let C be any cycle basis for G . Prove that for any cycle γ in G , there is a subset $A \subseteq C$ such that $A \cap \{\gamma\}$ is redundant. In other words, γ is the ‘exclusive or’ of the cycles in A .
 - (b) Prove that the set of independent cycle sets form a matroid.

*(c) Now suppose each edge of G has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a set of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis in G .
6. Describe a modification of GREEDYSCHEDULE that runs in $O(n \log n)$ time. [Hint: Store X in an appropriate data structure that supports the operations “Is $X \cup \{i\}$ realistic?” and “Add i to X ” in $O(\log n)$ time each.]

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 [John von] 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 with 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 to 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

$$T_{\text{average-case}}(n) = \underset{|X|=n}{\mathbb{E}} [T(X)] = \sum_{|X|=n} T(x) \cdot \Pr[X].$$

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:

$$T_{\text{worst-case expected}}(n) = \max_{|X|=n} \mathbb{E}[T(X)].$$

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 $n - 1$; its 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:

$$\mathbb{E}[T(n)] = \sum_{k=1}^{n-1} k \cdot \Pr[T(n) = k]$$

²The notation $\mathbb{E}[\cdot]$ for expectation has nothing to do with the shift operator \mathbf{E} used in the annihilator method for solving recurrences!

³This is what most people think 'random' means, but they're wrong.

⁴Note 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 k th 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 k th 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.

$$\begin{aligned} E[T(n)] &= \sum_{k=1}^{n-2} \frac{k}{n} + \frac{2(n-1)}{n} \\ &= \sum_{k=1}^{n-1} \frac{k}{n} + \frac{n-1}{n} \\ &= \frac{n(n-1)}{2n} + 1 - \frac{1}{n} \\ &= \frac{n+1}{2} - \frac{1}{n} \end{aligned}$$

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:

$$T(1) = 0, \quad E[T(n)] = 1 + \frac{n-1}{n} E[T(n-1)]$$

To get the solution, we define a new function $t(n) = n E[T(n)]$ and rewrite:

$$t(1) = 0, \quad t(n) = n + t(n-1)$$

This recurrence translates into a simple summation, which we can easily solve.

$$\begin{aligned} t(n) &= \sum_{k=2}^n k = \frac{n(n+1)}{2} - 1 \\ \implies E[T(n)] &= \frac{t(n)}{n} = \frac{n+1}{2} - \frac{1}{n} \end{aligned}$$

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

$$\begin{aligned} T(n) &= 2n - 1 + \max_{1 \leq k \leq n} \{T(k - 1) + T(n - k)\} \\ &= 2n - 1 + T(n - 1) \end{aligned}$$

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 matching 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 1995⁵, 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.

⁵János 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 $\bar{T}(n)$ in place of $E[T(n)]$ 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 k th smallest for any k , the expected number of tests performed by our algorithm is given by the following recurrence:

$$\begin{aligned}\bar{T}(n) &= 2n - 1 + E_k [\bar{T}(k - 1) + \bar{T}(n - k)] \\ &= 2n - 1 + \frac{1}{n} \sum_{k=1}^n (\bar{T}(k - 1) + \bar{T}(n - k))\end{aligned}$$

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:

$$\bar{T}(n) \leq 2n - 1 + \frac{1}{2} \left(\bar{T}\left(\frac{3n}{4}\right) + \bar{T}\left(\frac{n}{4}\right) \right) + \frac{1}{2} \cdot \bar{T}(n)$$

A little algebra simplifies this even further:

$$\bar{T}(n) \leq 4n - 2 + \bar{T}\left(\frac{3n}{4}\right) + \bar{T}\left(\frac{n}{4}\right)$$

We can solve this recurrence using the recursion tree method, giving us the unsurprising upper bound $\bar{T}(n) = O(n \log n)$. A similar argument gives us the matching lower bound $\bar{T}(n) = \Omega(n \log n)$.

Unfortunately, while this argument is convincing, it is *not* a formal proof, because it relies on the unproven assumption that $\bar{T}(n)$ is a *convex* function, which means that $\bar{T}(n+1) + \bar{T}(n-1) \geq 2\bar{T}(n)$ for all n . $\bar{T}(n)$ is actually convex, but we never proved it. Convexity follows from 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 i th smallest bolt, and N_j denote the j th 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:

$$\mathbb{E}[T(n)] = \mathbb{E} \left[\sum_{i=1}^n \sum_{j=1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=1}^n \mathbb{E}[X_{ij}].$$

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: $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$.

To analyze our algorithm, we only need to compute the expected value of each X_{ij} . By definition of expectation,

$$\mathbb{E}[X_{ij}] = 0 \cdot \Pr[X_{ij} = 0] + 1 \cdot \Pr[X_{ij} = 1] = \Pr[X_{ij} = 1],$$

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 $\{B_i, B_{i+1}, \dots, B_j\}$ is either B_i or B_j .

Since the set $\{B_i, B_{i+1}, \dots, B_j\}$ contains $j - i + 1$ bolts, each of which is equally likely to be chosen first, we immediately have

$$\mathbb{E}[X_{ij}] = \frac{2}{j - i + 1} \quad \text{for all } i < j.$$

Symmetric arguments give us $\mathbb{E}[X_{ij}] = \frac{2}{i-j+1}$ 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.

$$\begin{aligned} \mathbb{E}[T(n)] &= \sum_{i=1}^n \sum_{j=1}^n \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^n \mathbb{E}[X_{ii}] + 2 \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\ &= \boxed{n + 4 \sum_{i=1}^n \sum_{j=i+1}^n \frac{1}{j-i+1}} \end{aligned}$$

This is quite a bit simpler than the recurrence we got before. With just a few more lines of algebra, we can turn it into an exact, closed-form expression for the expected number of comparisons.

$$\begin{aligned} \mathbb{E}[T(n)] &= n + 4 \sum_{i=1}^n \sum_{j=2}^{n-i+1} \frac{1}{k} && [\text{substitute } k = j - i + 1] \\ &= n + 4 \sum_{k=2}^n \sum_{i=1}^{n-k+1} \frac{1}{k} && [\text{reorder summations}] \\ &= n + 4 \sum_{k=2}^n \frac{n-k+1}{k} \\ &= n + 4 \left((n-1) \sum_{k=2}^n \frac{1}{k} - \sum_{k=2}^n 1 \right) \\ &= n + 4((n+1)(H_n - 1) - (n-1)) \\ &= \boxed{4nH_n - 7n + 4H_n} \end{aligned}$$

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.

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

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 step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

$$\begin{aligned} n\bar{T}(n) &= 2n^2 - n + 2 \sum_{k=0}^{n-1} \bar{T}(k) \\ (n-1)\bar{T}(n-1) &= \underbrace{2(n-1)^2 - (n-1)}_{2n^2 - 5n + 3} + 2 \sum_{k=0}^{n-2} \bar{T}(k) \\ n\bar{T}(n) - (n-1)\bar{T}(n-1) &= 4n - 3 + 2T(n-1) \\ \bar{T}(n) &= 4 - \frac{3}{n} + \frac{n+1}{n}\bar{T}(n-1) \end{aligned}$$

To solve this limited-history recurrence, we define a new function $t(n) = \bar{T}(n)/(n+1)$. (I call this “Magic step #2”.) This gives us an even simpler recurrence for $t(n)$ in terms of $t(n-1)$:

$$\begin{aligned} t(n) &= \frac{\bar{T}(n)}{n+1} \\ &= \frac{1}{n+1} \left(4 - \frac{3}{n} + (n+1) \frac{T(n-1)}{n} \right) \\ &= \frac{4}{n+1} - \frac{3}{n(n+1)} + t(n-1) \\ &= \frac{7}{n+1} - \frac{3}{n} + t(n-1) \end{aligned}$$

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 $\bar{T}(n) = (n+1)t(n)$ and simplifying gives us the exact solution to the original recurrence.

$$\bar{T}(n) = 4(n+1)H_n - 7(n+1) + 7 = \boxed{4nH_n - 7n + 4H_n}$$

Surprise, surprise, we get exactly the same solution!

Exercises

Unless a problem specifically states otherwise, you can assume a function `RANDOM(k)` that returns, given any positive integer k , an integer chosen independently and uniformly at random from the set $\{1, 2, \dots, k\}$, in $O(1)$ time. For example, to perform a fair coin flip, one could call `RANDOM(2)`.

1. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. [Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]

2. Consider the following algorithm for finding the smallest element in an unsorted array:

```
RANDOMMIN( $A[1..n]$ ):
   $min \leftarrow \infty$ 
  for  $i \leftarrow 1$  to  $n$  in random order
    if  $A[i] < min$ 
       $min \leftarrow A[i]$  (*)
  return  $min$ 
```

- (a) In the worst case, how many times does RANDOMMIN execute line (*)?
(b) What is the probability that line (*) is executed during the n th iteration of the for loop?
(c) What is the *exact* expected number of executions of line (*)?
3. Let S be a set of n points in the plane. A point p in S is called *Pareto-optimal* if no other point in S is both above and to the right of p .
 - Describe and analyze a deterministic algorithm that computes the Pareto-optimal points in S in $O(n \log n)$ time.
 - Suppose each point in S is chosen independently and uniformly at random from the unit square $[0, 1] \times [0, 1]$. What is the *exact* expected number of Pareto-optimal points in S ?
4. Suppose we want to write an efficient function RANDOMPERMUTATION(n) that returns a permutation of the integers $\langle 1, \dots, n \rangle$ chosen uniformly at random.
 - Prove that the following algorithm is *not* correct. [Hint: Consider the case $n = 3$.]

```
RANDOMPERMUTATION( $n$ ):
  for  $i \leftarrow 1$  to  $n$ 
     $\pi[i] \leftarrow i$ 
  for  $i \leftarrow 1$  to  $n$ 
    swap  $\pi[i] \leftrightarrow \pi[RANDOM(n)]$ 
```

- (b) Consider the following implementation of RANDOMPERMUTATION.

```
RANDOMPERMUTATION( $n$ ):
  for  $i \leftarrow 1$  to  $n$ 
     $\pi[i] \leftarrow \text{NULL}$ 
  for  $i \leftarrow 1$  to  $n$ 
     $j \leftarrow \text{RANDOM}(n)$ 
    while ( $\pi[j] \neq \text{NULL}$ )
       $j \leftarrow \text{RANDOM}(n)$ 
     $\pi[j] \leftarrow i$ 
  return  $\pi$ 
```

Prove that this algorithm is correct. Analyze its expected runtime.

- (c) Consider the following partial implementation of RANDOMPERMUTATION.

```
RANDOMPERMUTATION( $n$ ):
  for  $i \leftarrow 1$  to  $n$ 
     $A[i] \leftarrow \text{RANDOM}(n)$ 
   $\pi \leftarrow \text{SOMEFUNCTION}(A)$ 
  return  $\pi$ 
```

Prove that if the subroutine SOMEFUNCTION is deterministic, then this algorithm cannot be correct. [Hint: There is a one-line proof.]

- *(d) Consider a correct implementation of RANDOMPERMUTATION(n) with the following property: whenever it calls RANDOM(k), the argument k is at most m . Prove that this algorithm *always* calls RANDOM at least $\Omega(\frac{n \log n}{\log m})$ times.
 - (e) Describe and analyze an implementation of RANDOMPERMUTATION that runs in expected worst-case time $O(n)$.
5. Consider the following randomized algorithm for generating biased random bits. The subroutine FAIRCOIN returns either 0 or 1 with equal probability; the random bits returned by FAIRCOIN are mutually independent.

```
ONEINTHREE:
  if FAIRCOIN = 0
    return 0
  else
    return  $1 - \text{ONEINTHREE}$ 
```

- (a) Prove that ONEINTHREE returns 1 with probability 1/3.
- (b) What is the *exact* expected number of times that this algorithm calls FAIRCOIN?
- (c) Now suppose you are *given* a subroutine ONEINTHREE that generates a random bit that is equal to 1 with probability 1/3. Describe a FAIRCOIN algorithm that returns either 0 or 1 with equal probability, using ONEINTHREE as a subroutine. **For this question, your only source of randomness is ONEINTHREE; in particular, you may not use the RANDOM function.**
- (d) What is the *exact* expected number of times that your FAIRCOIN algorithm calls ONEINTHREE?

6. Suppose n lights labeled $0, \dots, n - 1$ are placed clockwise around a circle. Initially, every light is off. Consider the following random process.

```
LIGHTTHECIRCLE( $n$ ):
   $k \leftarrow 0$ 
  turn on light 0
  while at least one light is off
    with probability  $1/2$ 
       $k \leftarrow (k + 1) \bmod n$ 
    else
       $k \leftarrow (k - 1) \bmod n$ 
    if light  $k$  is off, turn it on
```

- (a) Let $p(i, n)$ be the probability that light i is the last to be turned on by $\text{LIGHTTHECIRCLE}(n, 0)$. For example, $p(0, 2) = 0$ and $p(1, 2) = 1$. Find an exact closed-form expression for $p(i, n)$ in terms of n and i . Prove your answer is correct.
- (b) Give the tightest upper bound you can on the expected running time of this algorithm.
7. Consider a random walk on a path with vertices numbered $1, 2, \dots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n .
- (a) Prove that the probability that the walk ends by falling off the *right* end of the path is exactly $1/(n + 1)$.
- (b) Prove that if we start at vertex k , the probability that we fall off the *right* end of the path is exactly $k/(n + 1)$.
- (c) Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n .
- (d) Suppose we start at vertex $n/2$ instead. State and prove a tight Θ -bound on the expected length of the random walk in this case.
8. A *data stream* is an extremely long sequence of items that you can only read only once, in order. A good example of a data stream is the sequence of packets that pass through a router. Data stream algorithms must process each item in the stream quickly, using very little memory; there is simply too much data to store, and it arrives too quickly for any complex computations. Every data stream algorithm looks roughly like this:

```
DoSOMETHINGINTERESTING(stream  $S$ ):
  repeat
     $x \leftarrow$  next item in  $S$ 
     $\langle\langle$ do something fast with  $x$  $\rangle\rangle$ 
  until  $S$  ends
  return  $\langle\langle$ something $\rangle\rangle$ 
```

Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself).

9. The following randomized algorithm, sometimes called ‘one-armed quicksort’, selects the r th smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call $\text{RANDOMSELECT}(A, 1)$; to find the median element, you would call $\text{RANDOMSELECT}(A, \lfloor n/2 \rfloor)$. Recall from lecture that PARTITION splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element. The subroutine $\text{RANDOM}(n)$ returns an integer chosen uniformly at random between 1 and n , in $O(1)$ time.

```
RANDOMSELECT( $A[1..n], r$ ):
     $k \leftarrow \text{PARTITION}(A[1..n], \text{RANDOM}(n))$ 
    if  $r < k$ 
        return RANDOMSELECT( $A[1..k - 1], r$ )
    else if  $r > k$ 
        return RANDOMSELECT( $A[k + 1..n], r - k$ )
    else
        return  $A[k]$ 
```

- (a) State a recurrence for the expected running time of RANDOMSELECT , as a function of n and r .
 - (b) What is the *exact* probability that RANDOMSELECT compares the i th smallest and j th smallest elements in the input array? The correct answer is a simple function of i , j , and r . [Hint: Check your answer by trying a few small examples.]
 - (c) Show that for any n and r , the expected running time of RANDOMSELECT is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, find the *exact* expected number of comparisons, as a function of n and r .
 - (d) What is the expected number of times that RANDOMSELECT calls itself recursively?
10. Let $M[1..n][1..n]$ be an $n \times n$ matrix in which every row and every column is sorted. Such an array is called *totally monotone*. No two elements of M are equal.
- (a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M smaller than $M[i][j]$ and larger than $M[i'][j']$.
 - (b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements smaller than $M[i][j]$ and larger than $M[i'][j']$. Assume the requested range is always non-empty.
 - (c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.
11. Clock Solitaire is played with a standard deck of 52 cards, containing 13 ranks of cards in four different suits. To set up the game, deal the cards face down into 13 piles of four cards each, one in each of the ‘hour’ positions of a clock and one in the center. Each pile corresponds to a particular rank— A through Q in clockwise order for the hour positions, and K for the center. To start the game, turn over a card in the center pile. Then repeatedly turn over a card in the pile corresponding to the value of the previous card. The game ends when you try to turn over a card from a pile whose four cards are already face up. (This is always the center pile—why?) You win if and only if every card is face up when the game ends.

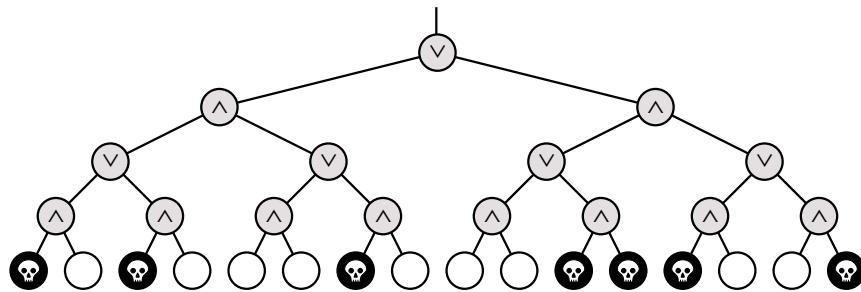
What is the *exact* probability that you win a game of Clock Solitaire, assuming that the cards are permuted uniformly at random before they are dealt into their piles?

12. Suppose we have a circular linked list of numbers, implemented as a pair of arrays, one storing the actual numbers and the other storing successor pointers. Specifically, let $X[1..n]$ be an array of n distinct real numbers, and let $N[1..n]$ be an array of indices with the following property: If $X[i]$ is the largest element of X , then $X[N[i]]$ is the smallest element of X ; otherwise, $X[N[i]]$ is the smallest element of X that is larger than $X[i]$. For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$N[i]$	6	8	9	5	2	3	1	7	4

Describe and analyze a randomized algorithm that determines whether a given number x appears in the array X in $O(\sqrt{n})$ expected time. **Your algorithm may not modify the arrays X and N .**

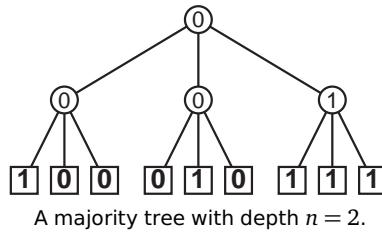
13. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- (a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*
- (b) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*
- *(c) Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm from part (b) at all!]*

14. A *majority tree* is a complete binary tree with depth n , where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled 1, 0, 0, 0, 1, 0, 1, 1, 1, the root has value 0.



- (a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. [Hint: Consider the special case $n = 1$. Recurse.]
- (b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. [Hint: Consider the special case $n = 1$. Recurse.]

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)

What is luck?

Luck is probability taken personally.

It is the excitement of bad math.

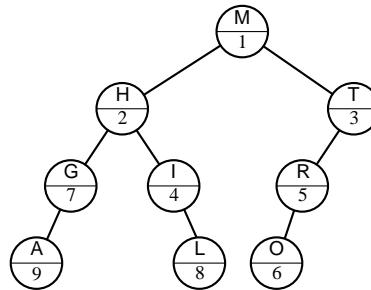
— Penn Jillette (2001), quoting Chip Denman (1998)

6 Randomized Binary Search Trees

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.



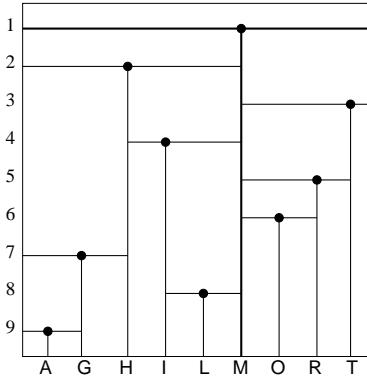
A treap. The top half of each node shows its search key and the bottom half shows its priority.

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 $\text{key}(u) < \text{key}(v)$ must be in the left subtree, and any node w with $\text{key}(w) > \text{key}(v)$ 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.

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.

¹Sometimes 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’.

A third description 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.



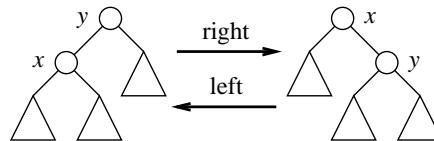
A geometric interpretation of the same treap.

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

6.2 Treap 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.



A right rotation at x and a left rotation at y are inverses.

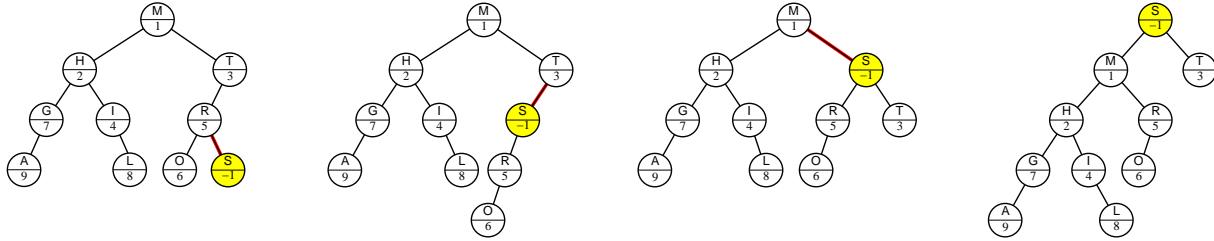
²J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

³E. M. McCreight. Priority search trees. *SIAM J. Comput.* 14(2):257–276, 1985.

⁴R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica* 16:464–497, 1996.

⁵C. 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.

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 $\text{key}(z)$.



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 than π . A simple way to do this is to insert a new node z with $\text{key}(z) = \pi$ and $\text{priority}(z) = -\infty$. 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.

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 $\text{key}(v) = k$. 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 $\Theta(n)$.

6.3 Random Priorities

A *randomized treap* 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 slightly messier. 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 k th smallest search key. To analyze the expected depth, we define an indicator variable

$$A_k^i = [x_i \text{ is a proper ancestor of } x_k].$$

(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.

$$\mathbb{E}[\text{depth}(x_k)] = \sum_{i=1}^n \Pr[A_k^i = 1]$$

(Just as in our analysis of matching nuts and bolts, we're using linearity of expectation and the fact that $\mathbb{E}[X] = \Pr[X = 1]$ 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 $\{x_i, x_{i+1}, \dots, x_k\}$ or the subset $\{x_k, x_{k+1}, \dots, x_i\}$, depending on whether $i < k$ or $i > k$. $X(i, k)$ and $X(k, i)$ always denote precisely 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 hand⁶, 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. \square

Since each node in $X(i, k)$ is equally likely to have smallest priority, we immediately have the probability we wanted:

$$\Pr[A_k^i = 1] = \frac{[i \neq k]}{|k - i| + 1} = \begin{cases} \frac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \frac{1}{i-k+1} & \text{if } i > k \end{cases}$$

⁶See Larry Niven and Jerry Pournelle, *The Gripping Hand*, Pocket Books, 1994.

To compute the expected depth of a node x_k , we just plug this probability into our formula and grind through the algebra.

$$\begin{aligned}
 E[\text{depth}(x_k)] &= \sum_{i=1}^n \Pr[A_k^i = 1] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^n \frac{1}{i-k+1} \\
 &= \sum_{j=2}^k \frac{1}{j} + \sum_{i=2}^{n-k+1} \frac{1}{j} \\
 &= H_k - 1 + H_{n-k+1} - 1 \\
 &< \ln k + \ln(n-k+1) - 2 \\
 &< 2 \ln n - 2.
 \end{aligned}$$

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).

6.4 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.

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 that this algorithm will run in $O(n \log n)$ expected time, since each key is inserted in $O(\log n)$ expected time.

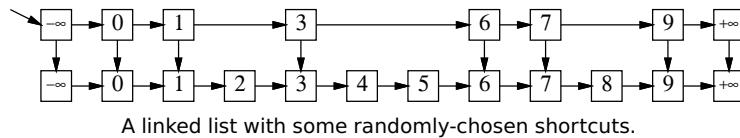
Why is this quicksort? Just like last time, all we've done is rearrange the order of the comparisons. Intuitively, the binary tree is just the recursion tree created by the normal version of quicksort. In the recursive formulation, we compare the initial pivot against everything else and then recurse. 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 i th largest and k th 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.

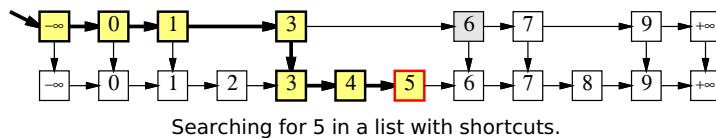
6.5 Skip Lists

Skip lists, which were first discovered by Bill Pugh in the late 1980's⁷, have many of the usual desirable properties of balanced binary search trees, but their structure is very different.

At a high level, a skip list is just a sorted linked list with some random 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.



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.

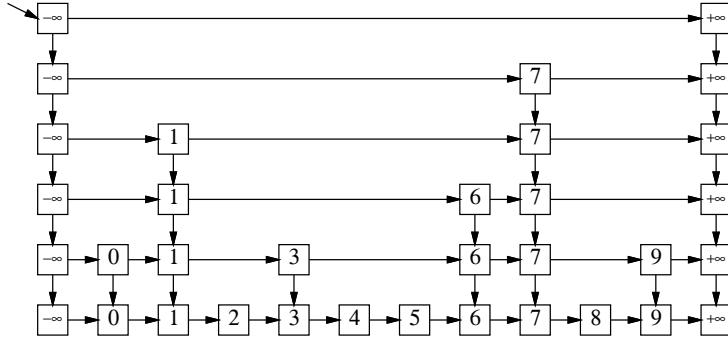


Since each node appears in the shortcut list with probability $1/2$, the expected number of nodes examined in the first phase is at most $n/2$. Only one of the nodes examined in the second phase has a duplicate. The probability that any node is followed by k nodes without duplicates is 2^{-k} , so the 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.

6.6 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 ($\text{key}(v)$), a pointer to its next lower copy ($\text{down}(v)$), and a pointer to the next node in its level ($\text{right}(v)$).

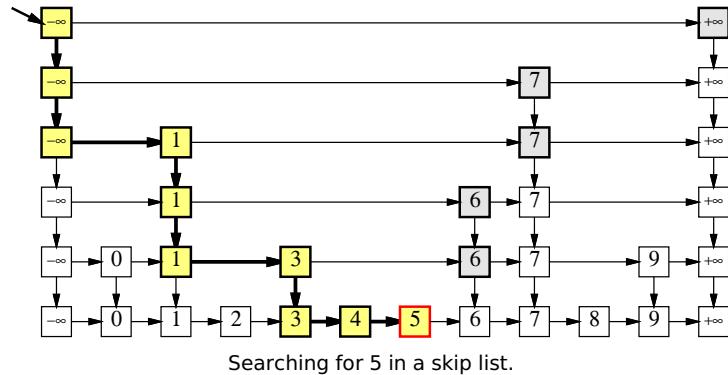
⁷William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33(6):668–676, 1990.



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 ( $v \neq \text{NULL}$  and  $\text{key}(v) \neq x$ )
        if  $\text{key}(\text{right}(v)) > x$ 
             $v \leftarrow \text{down}(v)$ 
        else
             $v \leftarrow \text{right}(v)$ 
    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 roughly 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.

6.7 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:

$$E[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2}(1 + E[L(x)])$$

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 level ℓ , it must have flipped ℓ heads in a row when it was inserted, so $\Pr[L(x) \geq \ell] = 2^{-\ell}$. The skip list has at least ℓ levels if and only if $L(x) \geq \ell$ for at least one of the n search keys.

$$\Pr[L \geq \ell] = \Pr[(L(1) \geq \ell) \vee (L(2) \geq \ell) \vee \dots \vee (L(n) \geq \ell)]$$

Using the *union bound* — $\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$ for any random events A and B — we can simplify this as follows:

$$\Pr[L \geq \ell] \leq \sum_{x=1}^n \Pr[L(x) \geq \ell] = n \cdot \Pr[L(x) \geq \ell] = \frac{n}{2^\ell}.$$

When $\ell \leq \lg n$, this bound is trivial. However, for any constant $c > 1$, we have a strong upper bound

$$\Pr[L \geq c \lg n] \leq \frac{1}{n^{c-1}}.$$

We conclude that **with high probability, a skip list has $O(\log n)$ levels**.

This high-probability bound indirectly implies a bound on the *expected* number of levels. Some simple algebra gives us the following alternate definition for expectation:

$$E[L] = \sum_{\ell \geq 0} \ell \cdot \Pr[L = \ell] = \sum_{\ell \geq 1} \Pr[L \geq \ell]$$

Clearly, if $\ell < \ell'$, then $\Pr[L(x) \geq \ell] > \Pr[L(x) \geq \ell']$. So we can derive an upper bound on the expected number of levels as follows:

$$\begin{aligned} E[L(x)] &= \sum_{\ell \geq 1} \Pr[L \geq \ell] \\ &= \sum_{\ell=1}^{\lg n} \Pr[L \geq \ell] + \sum_{\ell \geq \lg n+1} \Pr[L \geq \ell] \\ &\leq \sum_{\ell=1}^{\lg n} 1 + \sum_{\ell \geq \lg n+1} \frac{n}{2^\ell} \\ &= \lg n + \sum_{i \geq 1} \frac{1}{2^i} \quad [i = \ell - \lg n] \\ &= \lg n + 2 \end{aligned}$$

So in expectation, a skip list has *at most two* more levels than an ideal version where each level contains exactly half the nodes of the next level below.

6.8 Logarithmic Search Time

It's a little easier to analyze the cost of a search if we imagine running the algorithm backwards. `SKIPLISTFIND` 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 '(v)nowob → v' or '(v)righting → v'.⁸

```
SKIPLISTFIND(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, `SKIPLISTFIND` 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 worst-case search time is $O(\log n)$ with high probability (and therefore in expectation).

Exercises

1. Prove that the expected number of proper descendants of any node in a treap is *exactly* equal to the expected depth of that node.
2. What is the *exact* expected number of leaves in an n -node treap? [Hint: What is the probability that in an n -node treap, the node with k th smallest search key is a leaf?]
3. Consider an n -node treap T . As in the lecture notes, we identify nodes in T by the ranks of their search keys. Thus, 'node 5' means the node with the 5th smallest search key. Let i, j, k be integers such that $1 \leq i \leq j \leq k \leq n$.
 - (a) What is the *exact* probability that node j is a common ancestor of node i and node k ?
 - (b) What is the *exact* expected length of the unique path from node i to node k in T ?
4. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A **heater** is a priority search tree in which the *priorities* are given by

⁸Just like all the other problems in this lecture, this one is not designed to keep you from discovering secrets!

the user, and the *search keys* are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is a sort of dual treap.

The following problems consider an n -node heater T whose node priorities are the integers from 1 to n . We identify nodes in T by their priorities; thus, ‘node 5’ means the node in T with priority 5. The min-heap property implies that node 1 is the root of T . Finally, let i and j be integers with $1 \leq i < j \leq n$.

- (a) Prove that in a random permutation of the $(i + 1)$ -element set $\{1, 2, \dots, i, j\}$, elements i and j are adjacent with probability $2/(i + 1)$.
 - (b) Prove that node i is an ancestor of node j with probability $2/(i + 1)$. [Hint: Use part (a)!]
 - (c) What is the probability that node i is a descendant of node j ? [Hint: Don’t use part (a)!]
 - (d) What is the *exact* expected depth of node j ?
 - (e) Describe and analyze an algorithm to insert a new item into a heater. Express the expected running time of the algorithm in terms of the rank of the newly inserted item.
 - (f) Describe an algorithm to delete the minimum-priority item (the root) from an n -node heater. What is the expected running time of your algorithm?
5. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue containing the empty set.
- **FINDMIN(Q)**: Return the smallest element of Q (if any).
- **DELETEMIN(Q)**: Remove the smallest element in Q (if any).
- **INSERT(Q, x)**: Insert element x into Q , if it is not already there.
- **DECREASEKEY(Q, x, y)**: Replace an element $x \in Q$ with a smaller key y . (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x .
- **DELETE(Q, x)**: Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x .
- **MELD(Q_1, Q_2)**: Return a new priority queue containing all the elements of Q_1 and Q_2 ; this operation destroys Q_1 and Q_2 .

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```

MELD( $Q_1, Q_2$ ):
    if  $Q_1$  is empty return  $Q_2$ 
    if  $Q_2$  is empty return  $Q_1$ 
    if  $key(Q_1) > key(Q_2)$ 
        swap  $Q_1 \leftrightarrow Q_2$ 
    with probability 1/2
         $left(Q_1) \leftarrow \text{MELD}(left(Q_1), Q_2)$ 
    else
         $right(Q_1) \leftarrow \text{MELD}(right(Q_1), Q_2)$ 
    return  $Q_1$ 

```

- (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of $\text{MELD}(Q_1, Q_2)$ is $O(\log n)$, where $n = |Q_1| + |Q_2|$. [Hint: How long is a random root-to-leaf path in an n -node binary tree if each left/right choice is made with equal probability?]
- (b) Prove that $\text{MELD}(Q_1, Q_2)$ runs in $O(\log n)$ time with high probability.
- (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)
- *6. In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0, 1]$, but in practice, computers only have access to random *bits*. This problem asks you to analyze a modification of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1 .. \infty]$ of random bits, which is interpreted as the rational number

$$\text{priority}(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or ‘reveal’) new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LARGERPRIORITY( $v, w$ ):
  for  $i \leftarrow 1$  to  $\infty$ 
    if  $i > \ell_v$ 
       $\ell_v \leftarrow i$ ;  $\pi_v[i] \leftarrow \text{RANDOMBIT}$ 
    if  $i > \ell_w$ 
       $\ell_w \leftarrow i$ ;  $\pi_w[i] \leftarrow \text{RANDOMBIT}$ 
    if  $\pi_v[i] > \pi_w[i]$ 
      return  $v$ 
    else if  $\pi_v[i] < \pi_w[i]$ 
      return  $w$ 
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LARGERPRIORITY during these insertions.

- (a) Prove that $E[L] = \Theta(n)$.
- (b) Prove that $E[\ell_v] = \Theta(1)$ for any node v . [Hint: This is equivalent to part (a). Why?]
- (c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. [Hint: Why doesn’t this contradict part (b)?]
- *7. Any skip list \mathcal{L} can be transformed into a binary search tree $T(\mathcal{L})$ as follows. The root of $T(\mathcal{L})$ is the leftmost node on the highest non-empty level of \mathcal{L} ; the left and subtrees are constructed recursively from the nodes to the left and to the right of the root. Let’s call the resulting tree $T(\mathcal{L})$ a *skip list tree*.

- (a) Show that any search in $T(\mathcal{L})$ is no more expensive than the corresponding search in \mathcal{L} . (Searching in $T(\mathcal{L})$ could be *considerably* cheaper—why?)
- (b) Describe an algorithm to insert a new search key into a skip list tree in $O(\log n)$ expected time. Inserting key x into $T(\mathcal{L})$ should produce *exactly* the same tree as inserting x into \mathcal{L} and then transforming \mathcal{L} into a tree. [*Hint: You will need to maintain some additional information in the tree nodes.*]
- (c) Describe an algorithm to delete a search key from a skip list tree in $O(\log n)$ expected time. Again, deleting key x from $T(\mathcal{L})$ should produce *exactly* the same tree as deleting x from \mathcal{L} and then transforming \mathcal{L} into a tree.

*If you hold a cat by the tail
you learn things you cannot learn any other way.*
— Mark Twain

*E Tail Inequalities

The simple recursive structure of skip lists made 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 called *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.

E.1 Markov's Inequality

Perhaps the simplest tail inequality was named after the Russian mathematician Andrey Markov; however, in strict accordance with Stigler's Law of Eponymy, it first appeared in the works of Markov's probability teacher, Pafnuty Chebyshev.¹

Markov's Inequality. *Let X be a non-negative integer random variable. For any $t > 0$, we have $\Pr[X \geq t] \leq \mathbb{E}[X]/t$.*

Proof: The inequality follows from the definition of expectation by simple algebraic manipulation.

$$\begin{aligned}
 \mathbb{E}[X] &= \sum_{k=0}^{\infty} k \cdot \Pr[X = k] && [\text{definition of } \mathbb{E}[X]] \\
 &= \sum_{k=0}^{\infty} \Pr[X \geq k] && [\text{algebra}] \\
 &\geq \sum_{k=0}^{t-1} \Pr[X \geq k] && [\text{since } t < \infty] \\
 &\geq \sum_{k=0}^{t-1} \Pr[X \geq t] && [\text{since } k < t] \\
 &= t \cdot \Pr[X \geq t] && [\text{algebra}] \quad \square
 \end{aligned}$$

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.

¹The closely related tail bound traditionally called Chebyshev's inequality was actually discovered by the French statistician Irénée-Jules Bienaymé, a friend and colleague of Chebyshev's.

E.2 Sums of Indicator Variables

A set of random variables X_1, X_2, \dots, X_n are said to be *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, \dots, 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 the expectation of the product of the X_i 's is equal to the product of the expectations:

$$\mathbb{E} \left[\prod_{i=1}^n X_i \right] = \prod_{i=1}^n \mathbb{E}[X_i].$$

Moreover, if X_1, X_2, \dots, X_n are independent, then for any function f , the random variables $f(X_1), f(X_2), \dots, f(X_n)$ are also mutually independent.

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 $\mu = \mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = \sum_i p_i$.

Chernoff Bound (Upper Tail).
$$\boxed{\Pr[X > (1 + \delta)\mu] < \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu} \text{ for any } \delta > 0.$$

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:

$$\Pr[X > (1 + \delta)\mu] < \frac{\mathbb{E}[\exp(tX)]}{\exp(t(1 + \delta)\mu)}.$$

We can simplify the expectation on the right using the fact that the terms X_i are independent.

$$\mathbb{E}[\exp(tX)] = \mathbb{E} \left[\exp \left(t \sum_i X_i \right) \right] = \mathbb{E} \left[\prod_i \exp(tX_i) \right] = \prod_i \mathbb{E}[\exp(tX_i)]$$

We can bound the individual expectations $\mathbb{E}[e^{tX_i}]$ using The World's Most Useful Inequality:

$$\mathbb{E}[\exp(tX_i)] = p_i e^t + (1 - p_i) = 1 + (e^t - 1)p_i < \exp((e^t - 1)p_i)$$

This inequality gives us a simple upper bound for $\mathbb{E}[e^{tX}]$:

$$\mathbb{E}[\exp(tX)] < \prod_i \exp((e^t - 1)p_i) < \exp \left(\sum_i (e^t - 1)p_i \right) = \exp((e^t - 1)\mu)$$

Substituting this back into our original fraction from Markov's inequality, we obtain

$$\Pr[X > (1 + \delta)\mu] < \frac{\mathbb{E}[\exp(tX)]}{\exp(t(1 + \delta)\mu)} < \frac{\exp((e^t - 1)\mu)}{\exp(t(1 + \delta)\mu)} = (\exp(e^t - 1 - t(1 + \delta)))^\mu$$

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

$$\Pr[X > (1 + \delta)\mu] < (\exp(\delta - (1 + \delta)\ln(1 + \delta)))^\mu = \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu$$

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

$$\boxed{\Pr[X > (1 + \delta)\mu] < e^{-\mu\delta^2/3} \text{ for any } 0 < \delta < 1.}$$

A similar argument gives us an inequality bounding the probability that X is significantly *smaller* than its expected value:

Chernoff Bound (Lower Tail). $\boxed{\Pr[X < (1 - \delta)\mu] < \left(\frac{e^\delta}{(1 - \delta)^{1-\delta}}\right)^\mu < e^{-\mu\delta^2/2} \text{ for any } \delta > 0.}$

E.3 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 i th smallest key ('node i ') was a proper ancestor of the node with the k th 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

$$\mathbb{E}[\text{depth}(k)] = \sum_{i=1}^n \Pr[A_k^i = 1] = H_k + H_{n-k} - 2 < 2 \ln n.$$

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 that the relevant indicator variables are mutually independent.

Lemma 1. *For any index k , the $k-1$ random variables A_k^i with $i < k$ are mutually independent. Similarly, for any index k , the $n-k$ random variables A_k^i with $i > k$ are mutually independent.*

Proof: To simplify the notation, we explicitly consider only the case $k = 1$, although the argument generalizes easily to other values of k . Fix $n - 1$ arbitrary indicator values x_2, x_3, \dots, x_n . We prove the lemma by induction on n , with the vacuous base case $n = 1$. The definition of conditional probability gives us

$$\begin{aligned} \Pr \left[\bigwedge_{i=2}^n (A_1^i = x_i) \right] &= \Pr \left[\bigwedge_{i=2}^{n-1} (A_k^i = x_i) \wedge A_1^n = x_n \right] \\ &= \Pr \left[\bigwedge_{i=2}^{n-1} (A_k^i = x_i) \mid A_1^n = x_n \right] \cdot \Pr [A_1^n = x_n] \end{aligned}$$

Now recall that $A_1^n = 1$ if and only if node n has the smallest priority, and the other $n - 2$ indicator variables A_1^i depend only on the order of the priorities of nodes 1 through $n - 1$. There are exactly $(n - 1)!$ permutations of the n priorities in which the n th priority is smallest, and each of these permutations is equally likely. Thus,

$$\Pr \left[\bigwedge_{i=2}^{n-1} (A_k^i = x_i) \mid A_1^n = x_n \right] = \Pr \left[\bigwedge_{i=2}^{n-1} (A_k^i = x_i) \right]$$

The inductive hypothesis implies that the variables A_1^2, \dots, A_1^{n-1} are mutually independent, so

$$\Pr \left[\bigwedge_{i=2}^{n-1} (A_k^i = x_i) \right] = \prod_{i=2}^{n-1} \Pr [A_1^i = x_i].$$

We conclude that

$$\Pr \left[\bigwedge_{i=2}^n (A_1^i = x_i) \right] = \Pr [A_1^n = x_n] \cdot \prod_{i=2}^{n-1} \Pr [A_1^i = x_i] = \prod_{i=1}^{n-1} \Pr [A_1^i = x_i],$$

or in other words, that the indicator variables are mutually independent. \square

Theorem 2. *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 $8 \ln n$. There's nothing special about the constant 8 here; I'm being 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 $\text{depth}(k) = d_{<}(k) + d_{>}(k)$. If $\text{depth}(k) > 8 \ln n$, then either $d_{<}(k) > 4 \ln n$ or $d_{>}(k) > 4 \ln n$.

Chernoff's inequality, with $\mu = \mathbb{E}[d_{<}(k)] = H_k - 1 < \ln n$ and $\delta = 3$, bounds the probability that $d_{<}(k) > 4 \ln n$ as follows.

$$\Pr[d_{<}(k) > 4 \ln n] < \Pr[d_{<}(k) > 4\mu] < \left(\frac{e^3}{4^4} \right)^\mu < \left(\frac{e^3}{4^4} \right)^{\ln n} = n^{\ln(e^3/4^4)} = n^{3-4\ln 4} < \frac{1}{n^2}.$$

(The last step uses the fact that $4 \ln 4 \approx 5.54518 > 5$.) The same analysis implies that $\Pr[d_{>}(k) > 4 \ln n] < 1/n^2$. These inequalities imply the crude bound $\Pr[\text{depth}(k) > 4 \ln n] < 2/n^2$.

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:

$$\Pr \left[\max_k \text{depth}(k) > 8 \ln n \right] = \Pr \left[\bigwedge_{k=1}^n (\text{depth}(k) > 8 \ln n) \right] \leq \sum_{k=1}^n \Pr[\text{depth}(k) > 8 \ln n] < \frac{2}{n}.$$

This argument implies more generally that for any constant c , the depth of the treap is greater than $c \ln n$ with probability at most $2/n^{c \ln c - c}$. We can make the failure probability an arbitrarily small polynomial by choosing c appropriately. \square

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 $O(\log n)$.

Exercises

1. Prove that for any integer k such that $1 < k < n$, the $n - 1$ indicator variables A_k^i with $i \neq k$ are *not* mutually independent. [Hint: Consider the case $n = 3$.]
2. Recall from Exercise 1 in the previous note that the expected number of descendants of any node in a treap is $O(\log n)$. Why doesn't the Chernoff-bound argument for depth imply that, with high probability, *every* node in a treap has $O(\log n)$ descendants? The conclusion is clearly bogus—Every treap has a node with n descendants!—but what's the hole in the argument?
3. A *heater* is a sort of dual treap, in which the priorities of the nodes are given, but their search keys are generated independently and uniformly from the unit interval $[0, 1]$. You can assume all priorities and keys are distinct.
 - (a) Prove that for any r , the node with the r th smallest *priority* has expected depth $O(\log r)$.
 - (b) Prove that an n -node heater has depth $O(\log n)$ with high probability.
 - (c) Describe algorithms to perform the operations `INSERT` and `DELETEMIN` in a heater. What are the expected worst-case running times of your algorithms? In particular, can you express the expected running time of `INSERT` in terms of the priority rank of the newly inserted item?

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?
Hobbes: 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 finite¹ 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 any function of the form

$$h: \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\},$$

mapping each possible item in \mathcal{U} to a slot in the hash table. We say that an item x *hashes* to the slot $T[h(x)]$.

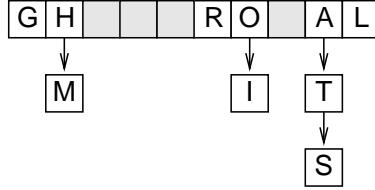
Of course, if $u = m$, then we can always just use the trivial hash function $h(x) = x$. In other words, use the item itself as the index into the table. This is called a *direct access table*, or more commonly, an *array*. In most applications, though, the universe of possible keys is orders of magnitude too large for this approach to be practical. Even when it is possible to allocate enough 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 they 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.

¹This 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^{32} .)



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: If $x \neq y$ then $\Pr[h(x) = h(y)] = 1/m$.

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 hash 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

$$C_{x,y} = [h(x) = h(y)].$$

(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:

$$x \neq y \implies E[C_{x,y}] = \Pr[C_{x,y} = 1] = \frac{1}{m}.$$

Now we just have to grind through the definitions.

$$E[\ell(x)] = \sum_{y \in T} E[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}$$

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 .

$$\alpha = \frac{n}{m}$$

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 of 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, $E[L] = \Theta(\log n / \log \log n)$ 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} .

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 $0 \leq a, b \leq m - 1$, define the function $h_{a,b} : \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\}$ as follows:

$$h_{a,b}(x, x') = (ax + bx') \bmod m.$$

Then the set

$$\mathcal{H} = \{h_{a,b} \mid 0 \leq a, b \leq m - 1\}$$

of all such functions is universal. To prove this, we need to show that for any pair of distinct items $(x, x') \neq (y, y')$, exactly m of the m^2 functions in \mathcal{H} cause a collision.

²No, this is not the name of Hannibal's brother. It's Latin for "Reader beware!"

Choose two items $(x, x') \neq (y, y')$, and assume without loss of generality³ that $x \neq y$. 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 $x - y \neq 0$, 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 an 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 $(x_0, x_1, \dots, x_{r-1})$ of integers between 0 and $m - 1$. (Think of x as an r -digit number written in base m .) Then for each vector $a = (a_0, a_1, \dots, a_{r-1})$, define the corresponding hash function h_a as follows:

$$h_a(x) = (a_0 x_0 + a_1 x_1 + \dots + a_{r-1} x_{r-1}) \bmod m.$$

Then the set of all m^r such functions is universal.

*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 version⁴ 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.*

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 $x' \neq y'$.

⁴The simple uniform hashing assumption requires only *pairwise* independence, but the following analysis requires *full* independence.

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, $E[X_j] = 1$ 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,

$$\begin{aligned}\Pr[X_j = k] &= \binom{n}{k} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \\ &\leq \frac{n^k}{k!} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \\ &< \frac{1}{k!}\end{aligned}$$

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:

$$\Pr[X_1 \geq k] < \frac{n}{k!}$$

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

$$\Pr\left[X_1 \geq \frac{(c+1) \ln n}{\ln \ln n}\right] < \frac{1}{n^c}.$$

This probability bound holds for every bin j . Thus, by the union bound, we conclude that

$$\begin{aligned}\Pr\left[\max_j X_j > \frac{(c+1) \ln n}{\ln \ln n}\right] &= \Pr\left[X_j > \frac{(c+1) \ln n}{\ln \ln n} \text{ for all } j\right] \\ &\leq \sum_{j=1}^n \Pr\left[X_j > \frac{(c+1) \ln n}{\ln \ln n}\right] \\ &< \frac{1}{n^{c-1}}.\end{aligned}$$
□

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 $E[C] = \binom{n}{2}/m$. 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 j th 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, $E[n_j^2] < 2$.*

Proof: Let X_{ij} be the indicator variable that equals 1 if item i hashes to slot j in the primary hash table.

$$\begin{aligned}
E[n_j^2] &= E \left[\left(\sum_{i=1}^n X_{ij} \right) \left(\sum_{k=1}^n X_{kj} \right) \right] = E \left[\sum_{i=1}^n \sum_{k=1}^n X_{ij} X_{kj} \right] \\
&= \sum_{i=1}^n \sum_{k=1}^n E[X_{ij} X_{kj}] \quad [\text{linearity of expectation}] \\
&= \sum_{i=1}^n E[X_{ij}^2] + 2 \sum_{i=1}^n \sum_{k=i+1}^n E[X_{ij} X_{kj}]
\end{aligned}$$

Because X_{ij} is an indicator variable, we have $X_{ij}^2 = X_{ij}$, which implies that $E[X_{ij}^2] = E[X_{ij}] = 1/n$ by the uniform hashing assumption. The uniform hashing assumption also implies that X_{ij} and X_{kj} are independent whenever $i \neq k$, so $E[X_{ij} X_{kj}] = E[X_{ij}] E[X_{kj}] = 1/n^2$. Thus,

$$E[n_j^2] = \sum_{i=1}^n \frac{1}{n} + 2 \sum_{i=1}^n \sum_{k=i+1}^n \frac{1}{n^2} = 1 + 2 \binom{n}{2} \frac{1}{n^2} = 2 - \frac{1}{n}. \quad \square$$

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 $\langle h_0, h_1, h_2, \dots, h_{m-1} \rangle$, such that for any item x , the *probe sequence* $\langle h_0(x), h_1(x), \dots, h_{m-1}(x) \rangle$ is a permutation of $\langle 0, 1, 2, \dots, m-1 \rangle$. 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  $T[h_i(x)] = \emptyset$ 
      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 :

$$E[T(m, n)] = 1 + \frac{n}{m} E[T(m - 1, n - 1)].$$

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 $E[T(m, n)] \leq m/(m - n)$:

$$\begin{aligned} E[T(m, n)] &= 1 + \frac{n}{m} E[T(m - 1, n - 1)] \\ &\leq 1 + \frac{n}{m} \cdot \frac{m - 1}{m - n} && [\text{induction hypothesis}] \\ &< 1 + \frac{n}{m} \cdot \frac{m}{m - n} && [m - 1 < m] \\ &= \frac{m}{m - n} && [\text{algebra}] \end{aligned}$$

Rewriting this in terms of the load factor $\alpha = n/m$, we get $E[T(m, n)] \leq 1/(1 - \alpha)$. 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.

Exercises

- Suppose we are using an *open-addressed* hash table of size m to store n items, where $n \leq m/2$. Assume that the strong uniform hashing assumption holds. For any i , let X_i denote the number of probes required for the i th insertion into the table, and let $X = \max_i X_i$ denote the length of the longest probe sequence.
 - Prove that $\Pr[X_i > k] \leq 1/2^k$ for all i and k .
 - Prove that $\Pr[X_i > 2\lg n] \leq 1/n^2$ for all i .
 - Prove that $\Pr[X > 2\lg n] \leq 1/n$.
 - Prove that $E[X] = O(\lg n)$.

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.

2. Your boss wants you to find a *perfect* hash function for mapping a known set of n items into a table of size m . A hash function is *perfect* if there are *no* collisions; each of the n items is mapped to a different slot in the hash table. Of course, this requires that $m \geq n$. (This is a different definition of perfect hashing than the one considered in the lecture notes.) After cursing your algorithms instructor for not teaching you about perfect hashing, you decide to try something simple: repeatedly pick random hash functions until you find one that happens to be perfect. A random hash function h satisfies two properties:

- $\Pr[h(x) = h(y)] = 1/m$ for any pair of items $x \neq y$.
- $\Pr[h(x) = i] = 1/m$ for any item x and any integer $1 \leq i \leq m$.

- (a) Suppose you pick a random hash function h . What is the *exact* expected number of collisions, as a function of n (the number of items) and m (the size of the table)? Don't worry about how to resolve collisions; just count them.
 - (b) What is the *exact* probability that a random hash function is perfect?
 - (c) What is the *exact* expected number of different random hash functions you have to test before you find a perfect hash function?
 - (d) What is the *exact* probability that none of the first N random hash functions you try is perfect?
 - (e) How many random hash functions do you have to test to find a perfect hash function *with high probability*?
- ★3. Recall that F_k denotes the k th Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for all $k \geq 2$. Suppose we are building a hash table of size $m = F_k$ using the hash function

$$h(x) = (F_{k-1} \cdot x) \bmod F_k$$

Prove that if the consecutive integers $0, 1, 2, \dots, F_k - 1$ are inserted in order into an initially empty table, each integer will be hashed into the largest contiguous empty interval in the table. In particular, show that there are no collisions.

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.

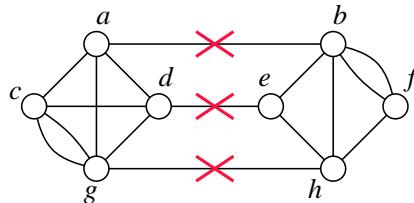
— William Shakespeare, *As You Like It*, Act V, Scene 4 (1600)

F Randomized Minimum Cut

E.1 Setting Up the Problem

This lecture considers a problem that arises in robust network design. Suppose we have a connected multigraph¹ 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.

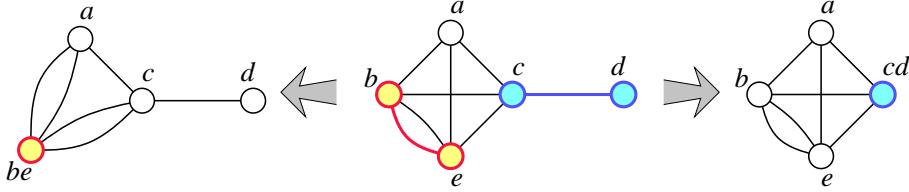
This problem has a long history. The classical deterministic algorithms for this problem rely on *network flow* techniques, which are discussed in another lecture. The fastest such algorithms (that we will discuss) run in $O(n^3)$ time and are quite complex and difficult to understand (unless you're already familiar with network flows). Here I'll describe a relatively simple randomized algorithm discovered by David Karger when he was a Ph.D. student.²

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 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

¹A 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.

²David R. Karger*. Random sampling in cut, flow, and network design problems. Proc. 25th STOC, 648–657, 1994.

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.

F.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  $i \leftarrow n$  down to 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.

$$P(n) \geq \prod_{i=3}^n \frac{i-2}{i} = \frac{\prod_{i=3}^n (i-2)}{\prod_{i=3}^n i} = \frac{\prod_{i=1}^{n-2} i}{\prod_{i=3}^n i} = \boxed{\frac{2}{n(n-1)}}$$

E.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  $N$ 
     $X \leftarrow \text{GUESSMINCUT}(G)$ 
    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:

$$\boxed{1 - x \leq e^{-x}}$$

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 $N = c \binom{n}{2} \ln n$ 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 $O(n^4 \log n)$ 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.)

F.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  $i \leftarrow n$  down to  $m$ 
    pick a random edge  $e$  in  $G$ 
     $G \leftarrow G/e$ 
return  $G$ 
```

BETTERGUESS(G):

```
if  $G$  has more than 8 vertices
     $X_1 \leftarrow \text{BETTERGUESS}(\text{CONTRACT}(G, n/\sqrt{2} + 1))$ 
     $X_2 \leftarrow \text{BETTERGUESS}(\text{CONTRACT}(G, n/\sqrt{2} + 1))$ 
    return  $\min\{X_1, X_2\}$ 
else
    use brute force
```

This might look like we're just doing the 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 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.

³David R. Karger* and Cliff Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. Proc. 25th STOC, 757–765, 1993.

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).

$$T(n) = O(n^2) + 2T\left(\frac{n}{\sqrt{2}} + 1\right) = O(n^2 \log n)$$

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 $O(n^2 \log^3 n)$ time!

*F.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 k th 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 $z(k) \leq 2k + 3$, since $z(0) = 3$. It follows that

$$p(k) \geq \bar{p}(k) > \frac{1}{2k+4} = \Omega(1/k).$$

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 $n \leq 8$. 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 $P(n) = p(k(n)) = \Omega(1/\log n)$, as promised.

Exercises

1. Suppose you had an algorithm to compute the minimum spanning tree of a graph in $O(m)$ time, where m is the number of edges in the input graph.⁴ Use this algorithm as a subroutine to improve the running time of GUESSMINCUT from $O(n^2)$ to $O(m)$.
2. Suppose you are given a graph G with weighted edges, and your goal is to find a cut whose total weight (not just number of edges) is smallest.
 - (a) Describe an algorithm to select a random edge of G , where the probability of choosing edge e is proportional to the weight of e .
 - (b) Prove that if you use the algorithm from part (a), instead of choosing edges uniformly at random, the probability that GUESSMINCUT returns a minimum-weight cut is still $\Omega(1/n^2)$.
 - (c) What is the running time of your modified GUESSMINCUT algorithm?
3. Prove that GUESSMINCUT returns the *second* smallest cut in its input graph with probability $\Omega(1/n^3)$. (The second smallest cut could be significantly larger than the minimum cut.)

⁴In fact, there is a randomized MST algorithm (due to Philip Klein, David Karger, and Robert Tarjan) whose expected running time is $O(m)$.

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 2. 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 2 whose sum is $n - 1$. If we add 2^0 to this set, we obtain a *multiset* of powers of two whose sum is n , which might contain two copies of 2^0 . Then as long as there are two copies of any 2^i in the multiset, we remove them both and insert 2^{i+1} in their place. The sum of the elements of the multiset is unchanged by this replacement, because $2^{i+1} = 2^i + 2^i$. Each iteration decreases the size of the multiset by 1, so the replacement process must eventually terminate. When it does terminate, we have a *set* of distinct powers of 2 whose sum is n . \square

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 1 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($B[0..\infty]$):

```
i ← 0
while B[i] = 1
    B[i] ← 0
    i ← i + 1
B[i] ← 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. The binary representation of any positive integer n is exactly $\lfloor \lg n \rfloor + 1$ bits long. Thus, if B represents an integer between 0 and n , INCREMENT takes $\Theta(\log n)$ time in the worst case.

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 INCREMENT, we get an upper bound of $O(n \log n)$ on the total running time. Although this bound is correct, we can do better. 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 Summation (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^i th time. If we start with an array full of 0's, 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 consider with randomized algorithms. 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* time for 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. For this reason, 'amortized cost' is a common synonym for amortized running time.

Most textbooks call this technique for bounding amortized costs the *aggregate* method, or *aggregate analysis*, but this is just a fancy name for adding up the costs of the individual operations and dividing by the number of operations.

The Summation Method. *Let $T(n)$ be the worst-case running time for a sequence of n operations. The amortized time for 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 distribute 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 incurred by the algorithm is never more than the total tax 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 or 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 it is 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.

The 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 its 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 i th operation (which changes D_{i-1} into D_i). Then the *amortized* cost of the i th operation, denoted a_i , is defined to be the actual cost plus the increase in potential:

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

So the *total* amortized cost of n operations is the actual total cost plus the total increase in potential:

$$\sum_{i=1}^n a_i = \sum_{i=1}^n (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^n c_i + \Phi_n - \Phi_0.$$

A potential function is *valid* if $\Phi_i - \Phi_0 \geq 0$ for all i . If the potential function is valid, then the total *actual* cost of any sequence of operations is always 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 i th 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 valid 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.

$$c_i = \#\text{bits changed from 0 to 1} + \#\text{bits changed from 1 to 0}$$

$$\Phi_i - \Phi_{i-1} = \#\text{bits changed from 0 to 1} - \#\text{bits changed from 1 to 0}$$

Thus, the amortized cost of the i th INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits changed from 0 to 1}$$

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 precisely the total unspent taxes paid using the taxation method, so not too surprisingly, we obtain precisely the same amortized cost. In general, however, there may be no way of interpreting the change in potential as ‘taxes’. Taxation and charging are useful when there is a convenient way to distribute costs to specific steps in the algorithm or components of the data structure; potential arguments allow us to argue more globally when a local distribution is difficult or impossible.

Different potential functions can 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 can both increment and decrement efficiently. A standard binary counter won’t work, even in an amortized sense; if we alternate between 2^k and $2^k - 1$, every operation costs $\Theta(k)$ time.

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):	DECREMENT(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 $P[i] \leftarrow 1$	$i \leftarrow 0$ 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.

$$\begin{array}{ccccccc}
 P = 10001 & P = 100\textcolor{red}{10} & P = 10011 & P = 100\textcolor{red}{00} & P = 10000 & P = 10000 & P = 10001 \\
 N = 01100 & \xrightarrow{++} N = 01100 & \xrightarrow{++} N = 01100 & \xrightarrow{++} N = 01000 & \xrightarrow{--} N = 01001 & \xrightarrow{--} N = 01010 & \xrightarrow{++} N = 01010 \\
 P - N = 5 & P - N = 6 & P - N = 7 & P - N = 8 & P - N = 7 & P - N = 6 & P - N = 7
 \end{array}$$

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

$$\begin{aligned}
 c_i &= \# \text{bits changed from 0 to 1} + \# \text{bits changed from 1 to 0} \\
 \Phi_i - \Phi_{i-1} &= \# \text{bits changed from 0 to 1} - \# \text{bits changed from 1 to 0} \\
 \implies a_i &= 2 \times \# \text{bits changed from 0 to 1}
 \end{aligned}$$

Since each operation changes *at most* one bit to 1, the i th operation has amortized cost $a_i \leq 2$.

*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 or 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 Ehrlich¹ 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
     $G[j] = 1 - G[j]$ 
     $F[j] \leftarrow F[j+1]$ 
     $F[j+1] \leftarrow j+1$ 

```

¹Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.* 20:500–513, 1973.

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 i th 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 also 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.

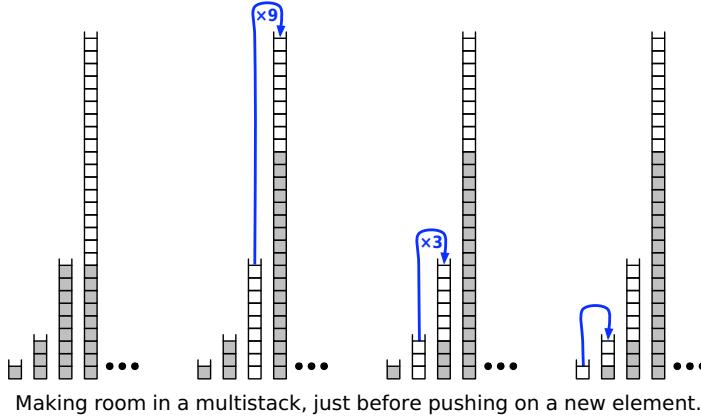
Exercises

1. Suppose we are maintaining a data structure under a series of operations. Let $f(n)$ denote the actual running time of the n th operation. For each of the following functions f , determine the resulting amortized cost of a single operation. (For practice, try *all* of the methods described in this note.)
 - (a) $f(n)$ is the largest power of 2 that divides n .
 - (b) $f(n) = n$ if n is a power of 2, and $f(n) = 1$ otherwise.
 - (c) $f(n) = n^2$ if n is a power of 2, and $f(n) = 1$ otherwise.
2. A *multistack* consists of an infinite series of stacks S_0, S_1, S_2, \dots , where the i th stack S_i can hold up to 3^i elements. The user always pushes and pops elements from the smallest stack S_0 . However, before any element can be pushed onto any full stack S_i , we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move

all its members to S_{i+2} .) Similarly, before any element can be popped from any empty stack S_i , we first pop 3^i elements from S_{i+1} and push them onto S_i to make room. (Thus, if S_{i+1} is already empty, we first recursively fill it by popping elements from S_{i+2} .) Moving a single element from one stack to another takes $O(1)$ time.

Here is pseudocode for the multistack operations `MSPUSH` and `MSPOP`. The internal stacks are managed with the subroutines `PUSH` and `POP`.

<u><code>MSPUSH(x)</code>:</u> $i \leftarrow 0$ while S_i is full $i \leftarrow i + 1$ while $i > 0$ $i \leftarrow i - 1$ for $j \leftarrow 1$ to 3^i $\quad \text{PUSH}(S_{i+1}, \text{POP}(S_i))$ $\text{PUSH}(S_0, x)$	<u><code>MSPOP(x)</code>:</u> $i \leftarrow 0$ while S_i is empty $i \leftarrow i + 1$ while $i > 0$ $i \leftarrow i - 1$ for $j \leftarrow 1$ to 3^i $\quad \text{PUSH}(S_i, \text{POP}(S_{i+1}))$ $\text{return POP}(S_0)$
--	---



- (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?
 - (b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack during its lifetime.
 - (c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where n is the maximum number of elements in the multistack during its lifetime.
3. Remember the difference between stacks and queues? Good.

- (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard subroutines `PUSH` and `POP`.
- (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:
 - **Push:** add a new item to the left end of the list;

- **Pop:** remove the item on the left end of the list;
- **Pull:** remove the item on the right end of the list.

Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any push, pop, or pull operation is $O(1)$. In particular, each element pushed onto the quack should be stored in exactly one of the three stacks. Again, you are *only* allowed to access the stacks through the standard functions `PUSH` and `POP`.

4. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. [Hint: Do not even look at the potential-function argument in CLRS; there is a much easier solution!]

5. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if and only if every building to the east of i is shorter than i .

Here is an algorithm that computes which buildings have a good view of Lake Michigan. What is the running time of this algorithm?

```

GOODVIEW( $A[1..n]$ ):
    initialize a stack  $S$ 
    for  $i \leftarrow 1$  to  $n$ 
        while ( $S$  not empty and  $A[i] > A[\text{TOP}(S)]$ )
             $\text{POP}(S)$ 
         $\text{PUSH}(S, i)$ 
    return  $S$ 
```

6. Design and analyze a simple data structure that maintains a list of integers and supports the following operations.

- `CREATE()` creates and returns a new list
- `PUSH(L, x)` appends x to the end of L
- `POP(L)` deletes the last entry of L and returns it
- `LOOKUP(L, k)` returns the k th entry of L

Your solution may use these primitive data structures: arrays, balanced binary search trees, heaps, queues, single or doubly linked lists, and stacks. If your algorithm uses *anything* fancier, you must give an explicit implementation. Your data structure must support all operations in amortized constant time. In addition, your data structure must support each `LOOKUP` in *worst-case* $O(1)$ time. At all times, the size of your data structure must be linear in the number of objects it stores.

- *7. 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 i th least significant fit indicates whether the sum includes the i th 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. A *doubly lazy binary counter* represents any number as a weighted sum of powers of two, where each weight is one of four values: -1 , 0 , 1 , or 2 . (For succinctness, I'll write \pm instead of -1 .) Every integer—positive, negative, or zero—has an infinite number of doubly lazy binary representations. For example, the number 13 can be represented as 1101 (the standard binary representation), or 2 ± 01 (because $2 \cdot 2^3 - 2^2 + 2^0 = 13$) or $10\pm 1\pm$ (because $2^4 - 2^2 + 2^1 - 2^0 = 13$) or $\pm 1200010\pm 1\pm$ (because $-2^{10} + 2^9 + 2 \cdot 2^8 + 2^4 - 2^2 + 2^1 - 2^0 = 13$).

To increment a doubly lazy binary counter, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the least significant bit, and then borrow the rightmost \pm (if any).

<u>LAZYINCREMENT($B[0..n]$):</u>
$B[0] \leftarrow B[0] + 1$
for $i \leftarrow 1$ to $n - 1$
if $B[i] = 2$
$B[i] \leftarrow 0$
$B[i + 1] \leftarrow B[i + 1] + 1$
return

<u>LAZYDECREMENT($B[0..n]$):</u>
$B[0] \leftarrow B[0] - 1$
for $i \leftarrow 1$ to $n - 1$
if $B[i] = -1$
$B[i] \leftarrow 1$
$B[i + 1] \leftarrow B[i + 1] - 1$
return

For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit $B[0]$ on the *right*, omitting all leading 0's

$0 \xrightarrow{++} 1 \xrightarrow{++} 10 \xrightarrow{++} 11 \xrightarrow{++} 20 \xrightarrow{++} 101 \xrightarrow{++} 110 \xrightarrow{++} 111 \xrightarrow{++} 120 \xrightarrow{++} 201 \xrightarrow{++} 210$
 $\xrightarrow{++} 1011 \xrightarrow{++} 1020 \xrightarrow{++} 1101 \xrightarrow{++} 1110 \xrightarrow{++} 1111 \xrightarrow{++} 1120 \xrightarrow{++} 1201 \xrightarrow{++} 1210 \xrightarrow{++} 2011 \xrightarrow{++} 2020$
 $\xrightarrow{--} 2011 \xrightarrow{--} 2010 \xrightarrow{--} 2001 \xrightarrow{--} 2000 \xrightarrow{--} 20\pm 1 \xrightarrow{--} 2\pm 10 \xrightarrow{--} 2\pm 01 \xrightarrow{--} 1100 \xrightarrow{--} 11\pm 1 \xrightarrow{--} 1010$
 $\xrightarrow{--} 1001 \xrightarrow{--} 1000 \xrightarrow{--} 10\pm 1 \xrightarrow{--} 1\pm 10 \xrightarrow{--} 1\pm 01 \xrightarrow{--} 100 \xrightarrow{--} 1\pm 1 \xrightarrow{--} 10 \xrightarrow{--} 1 \xrightarrow{--} 0$

Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0, the amortized time for each operation is $O(1)$. Do *not* assume, as in the example above, that all the increments come before all the decrements.

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.

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 independently² by Igal

¹Yeah, yeah. Skip lists aren't really binary search trees. Whatever you say, Mr. Picky.

²The 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.

Galperin* and Ron Rivest in 1993 [10]. The second is the *splay tree*, discovered by Danny Sleator and Bob Tarjan in 1981 [16, 14].

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 $O(\log n)$.

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).*

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.

³Alternately: 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.

⁴Well, we could use the Bentley-Saxe* logarithmic method [3], but that would raise the query time to $O(\log^2 n)$.

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:

$$I(v) = \max \{ 0, |size(left(v)) - size(right(v))| - 1 \}$$

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 $size(left(v))$ or $size(right(v))$, 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, $I(v) = \Omega(size(v))$.*

Before we prove this, let's first look at what it implies. If $I(v) = \Omega(size(v))$, then $\Omega(size(v))$ 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 $O(size(v))$ 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 $O(\log n)$.

Proof: Since we're about to rebuild the subtree at v , we must have $height(v) > \alpha \cdot \lg size(v)$. 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 $height(left(v)) \leq \alpha \cdot \lg size(left(v))$. Combining these two inequalities with the recursive definition of height, we get

$$\alpha \cdot \lg size(v) < height(v) \leq height(left(v)) + 1 \leq \alpha \cdot \lg size(left(v)) + 1.$$

After some algebra, this simplifies to $size(left(v)) > size(v)/2^{1/\alpha}$. Combining this with the identity $size(v) = size(left(v)) + size(right(v)) + 1$ and doing some more algebra gives us the inequality

$$size(right(v)) < (1 - 1/2^{1/\alpha})size(v) - 1.$$

Finally, we combine these two inequalities using the recursive definition of imbalance.

$$I(v) \geq size(left(v)) - size(right(v)) - 1 > (2/2^{1/\alpha} - 1)size(v)$$

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.

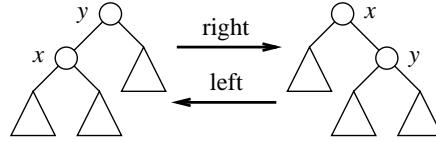


Figure 1. A right rotation at x and a left rotation at y are inverses.

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.

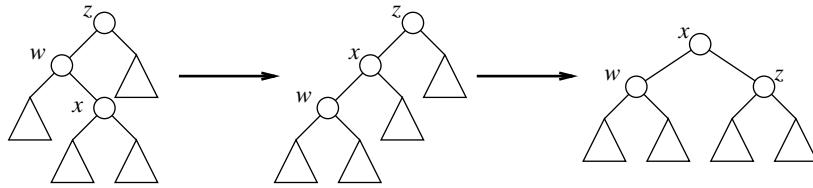


Figure 2. A zig-zag at x . The symmetric case is not shown.

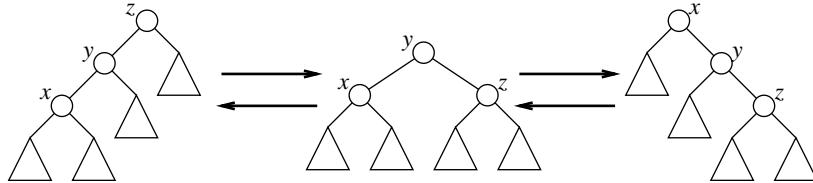


Figure 3. A right roller-coaster at x and a left roller-coaster at z .

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 $\text{depth}(v)$. (Obviously, this means the depth *before* splaying, since after splaying v is the root and thus has depth zero!)

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.

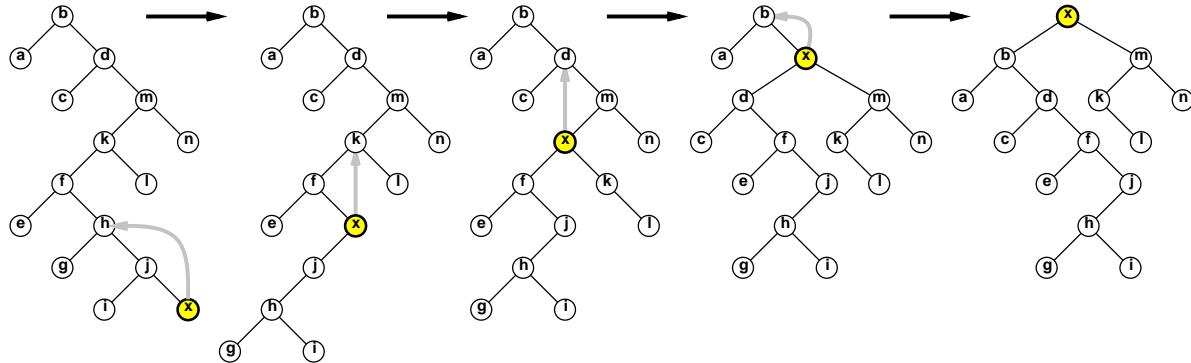


Figure 4. Splaying a node. Irrelevant subtrees are omitted for clarity.

- **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 (the inorder predecessor of x in the original tree), splay it, and finally join it to the right subtree.

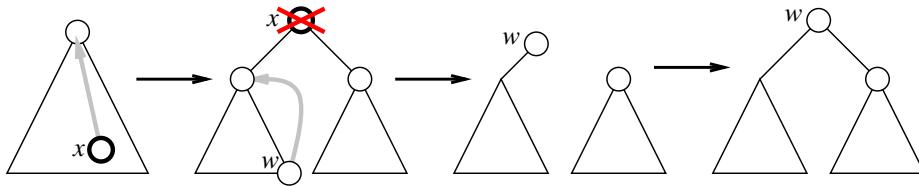


Figure 5. Deleting a node in a splay tree.

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 $\lfloor \lg \text{size}(v) \rfloor$, and the *potential* of a splay tree to be the sum of the ranks of its nodes:

$$\Phi = \sum_v \text{rank}(v) = \sum_v \lfloor \lg \text{size}(v) \rfloor$$

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, $\text{rank}(v)$ denotes the rank of v before a (single or double) rotation, and $\text{rank}'(v)$ 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 $1 + 3 \text{rank}'(v) - 3 \text{rank}(v)$, and the amortized cost of a double rotation at any node v is at most $3 \text{rank}'(v) - 3 \text{rank}(v)$.*

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 $1 + 3\text{rank}'(v) - 3\text{rank}(v)$, where $\text{rank}'(v)$ 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, $\text{rank}'(v) = \lfloor \lg n \rfloor$, 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:

$$s(v) := w(v) + s(\text{right}(v)) + s(\text{left}(v)).$$

If $w(v) = 1$ for every node v , then the size of a node is just the number of nodes in its subtree, as in the previous section. As before, we define the *rank* of any node v to be $r(v) = \lg s(v)$, and the *potential* of a splay tree to be the sum of the ranks of all its nodes:

$$\Phi = \sum_v r(v) = \sum_v \lg s(v)$$

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.

$$\begin{aligned} 1 + \Phi' - \Phi &= 1 + r'(x) + r'(y) - r(x) - r(y) && [\text{only } x \text{ and } y \text{ change rank}] \\ &\leq 1 + r'(x) - r(x) && [r'(y) \leq r(y)] \\ &\leq 1 + 3r'(x) - 3r(x) && [r'(x) \geq r(x)] \end{aligned}$$

Now consider a zig-zag, as shown in Figure 2. Only w , x , and z change rank.

$$\begin{aligned} 2 + \Phi' - \Phi &= 2 + r'(w) + r'(x) + r'(z) - r(w) - r(x) - r(z) && [\text{only } w, x, z \text{ change rank}] \\ &\leq 2 + r'(w) + r'(x) + r'(z) - 2r(x) && [r(x) \leq r(w) \text{ and } r'(x) = r(z)] \\ &= 2 + (r'(w) - r'(x)) + (r'(z) - r'(x)) + 2(r'(x) - r(x)) \\ &= 2 + \lg \frac{s'(w)}{s'(x)} + \lg \frac{s'(z)}{s'(x)} + 2(r'(x) - r(x)) \\ &\leq 2 + 2 \lg \frac{s'(x)/2}{s'(x)} + 2(r'(x) - r(x)) && [s'(w) + s'(z) \leq s'(x), \lg \text{ is concave}] \\ &= 2(r'(x) - r(x)) \\ &\leq 3(r'(x) - r(x)) && [r'(x) \geq r(x)] \end{aligned}$$

Finally, consider a roller-coaster, as shown in Figure 3. Only x , y , and z change rank.

$$\begin{aligned}
 2 + \Phi' - \Phi &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) && [\text{only } x, y, z \text{ change rank}] \\
 &\leq 2 + r'(x) + r'(z) - 2r(x) && [r'(y) \leq r(z) \text{ and } r(x) \geq r(y)] \\
 &= 2 + (r(x) - r'(x)) + (r'(z) - r'(x)) + 3(r'(x) - r(x)) \\
 &= 2 + \lg \frac{s(x)}{s'(x)} + \lg \frac{s'(z)}{s'(x)} + 3(r'(x) - r(x)) \\
 &\leq 2 + 2\lg \frac{s'(x)/2}{s'(x)} + 3(r'(x) - r(x)) && [s(x) + s'(z) \leq s'(x), \lg \text{ is concave}] \\
 &= 3(r'(x) - r(x))
 \end{aligned}$$

This completes the proof.⁵ □

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 $1 + 3r(\text{root}) - 3r(x)$. (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 $\text{dist}(x, z)$ denote the *rank distance* between x and z , that is, the number of nodes y such that $\text{key}(x) \leq \text{key}(y) \leq \text{key}(z)$ or $\text{key}(x) \geq \text{key}(y) \geq \text{key}(z)$. In particular, $\text{dist}(x, x) = 1$ for all x .

Static Finger Theorem. *For any fixed node f ('the finger'), the amortized cost of accessing x is $O(\lg \text{dist}(f, x))$.*

Proof: Set $w(x) = 1/\text{dist}(x, f)^2$ for each node x . Then $s(\text{root}) \leq \sum_{i=1}^{\infty} 2/i^2 = \pi^2/3 = O(1)$, and $r(x) \geq \lg w(x) = -2\lg \text{dist}(f, x)$. □

Here are a few more interesting properties of splay trees, which I'll state without proof.⁶ The proofs of these properties (especially the dynamic finger theorem) are considerably more complicated than the amortized analysis presented above.

Working Set Theorem [14]. *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 $D = n$.)*

⁵This proof is essentially taken verbatim from the original Sleator and Tarjan paper. Another proof technique, which may be more accessible, involves maintaining $\lfloor \lg s(v) \rfloor$ tokens on each node v and arguing about the changes in token distribution caused by each single or double rotation. But I haven't yet internalized this approach enough to include it here.

⁶This list and the following section are taken almost directly from Erik Demaine's lecture notes [5].

Scanning Theorem [17]. *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 $O(\lg \text{dist}(x, y))$.*

*9.8 Splay Tree Conjectures

Splay trees are conjectured to have many interesting properties in addition to the optimality properties that have been proved; I'll describe just a few of the more important ones.

The *Deque Conjecture* [17] 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 $O((m+n)\alpha(m+n))$ [15]. More recently, Pettie later improved this bound to $O(m\alpha^*(n))$ [13]. The Deque Conjecture states that the total cost is actually $O(m+n)$.

The *Traversal Conjecture* [14] 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* [12] 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, 12].

Finally, the most important conjecture about splay trees, and one of the most important open problems about data structures, is that they are *dynamically optimal* [14]. 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. However, three very similar $O(\log \log n)$ -competitive binary search trees have been discovered in the last few years: *Tango trees* [8], *multisplay trees* [18], and *chain-splay trees* [11]. A recently-published geometric formulation of dynamic binary search trees [9] also offers significant hope for future progress.

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. Bădiu* and E. D. Demaine. A simplified and dynamic unified structure. *Proc. 6th Latin American Symp. Theoretical Informatics*, 466–473, 2004. Lecture Notes Comput. Sci. 2976, Springer-Verlag.
- [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. Pătrașcu*. Dynamic optimality—almost. *Proc. 45th Ann. IEEE Symp. Foundations Comput. Sci.*, 484–490, 2004.
- [9] E. D. Demaine, D. Harmon*, J. Iacono, D. Kane*, and M. Pătrașcu*. The geometry of binary search trees. *Proc. 20th ACM-SIAM Symp. Discrete Algorithms.*, 496–505, 2009.
- [10] I. Galperin* and R. R. Rivest. Scapegoat trees. *Proc. 4th Ann. ACM-SIAM Symp. Discrete Algorithms*, 165–174, 1993.
- [11] G. Georgakopolous. Chain-splay trees, or, how to achieve and prove $\log \log N$ -competitiveness by splaying. *Inform. Proc. Lett.* 106:37–34, 2008.
- [12] J. Iacono*. Alternatives to splay trees with $O(\log n)$ worst-case access times. *Proc. 12th Ann. ACM-SIAM Symp. Discrete Algorithms*, 516–522, 2001.
- [13] S. Pettie. Splay trees, Davenport-Schinzel sequences, and the deque conjecture. *Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms*, 1115–1124, 2008.
- [14] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM* 32(3):652–686, 1985.
- [15] R. Sundar*. On the Deque conjecture for the splay algorithm. *Combinatorica* 12(1):95–124, 1992.
- [16] R. E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics 44. SIAM, 1983.
- [17] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica* 5(5):367–378, 1985.
- [18] C. C. Wang*, J. Derryberry*, and D. D. Sleator. $O(\log \log n)$ -competitive dynamic binary search trees. *Proc. 17th Ann. ACM-SIAM Symp. Discrete Algorithms*, 374–383, 2006.

*Starred authors were students at the time that the cited work was published.

Exercises

1. (a) An n -node binary tree is *perfectly balanced* if either $n \leq 1$, or its two subtrees are perfectly balanced binary trees, each with at most $\lfloor n/2 \rfloor$ nodes. Prove that $I(v) = 0$ for every node v of any perfectly balanced tree.
 (b) Prove that at most one subtree is rebalanced during a scapegoat tree insertion.
2. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.
 - (a) Describe and analyze an algorithm to purge an arbitrary n -node dirty binary search tree in $O(n)$ time. (Such an algorithm is necessary for scapegoat trees to achieve $O(\log n)$ amortized insertion cost.)
 - *(b) Modify your algorithm so that it uses only $O(\log n)$ space, in addition to the tree itself. Don't forget to include the recursion stack in your space bound.
 - ★(c) Modify your algorithm so that it uses only $O(1)$ additional space. In particular, your algorithm cannot call itself recursively at all.
3. Consider the following simpler alternative to splaying:

```
MOVEToRoot( $v$ ):
  while  $parent(v) \neq \text{NULL}$ 
    rotate at  $v$ 
```

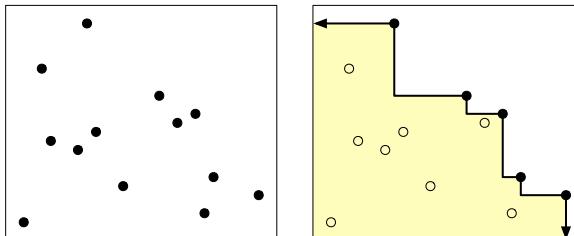
Prove that the amortized cost of `MoveToRoot` in an n -node binary tree can be $\Omega(n)$. That is, prove that for any integer k , there is a sequence of k `MoveToRoot` operations that require $\Omega(kn)$ time to execute.

4. Suppose we want to maintain a dynamic set of values, subject to the following operations:

- `INSERT(x)`: Add x to the set (if it isn't already there).
- `PRINT&DELETEBETWEEN(a, b)`: Print every element x in the range $a \leq x \leq b$, in increasing order, and delete those elements from the set.

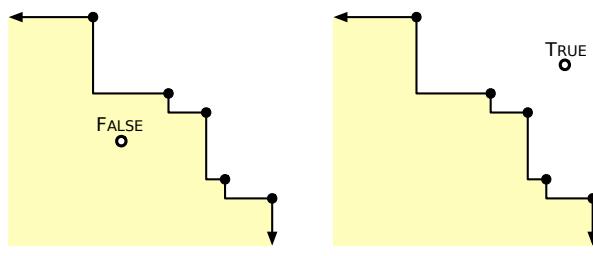
For example, if the current set is $\{1, 5, 3, 4, 8\}$, then `PRINT&DELETEBETWEEN(4, 6)` prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$. For any underlying set, `PRINT&DELETEBETWEEN(-\infty, \infty)` prints its contents in increasing order and deletes everything.

- (a) Describe and analyze a data structure that supports these operations, each with amortized cost $O(\log n)$, where n is the maximum number of elements in the set.
 - (b) What is the running time of your `INSERT` algorithm in the worst case?
 - (c) What is the running time of your `PRINT&DELETEBETWEEN` algorithm in the worst case?
5. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.



A set of points in the plane and its staircase (shaded).

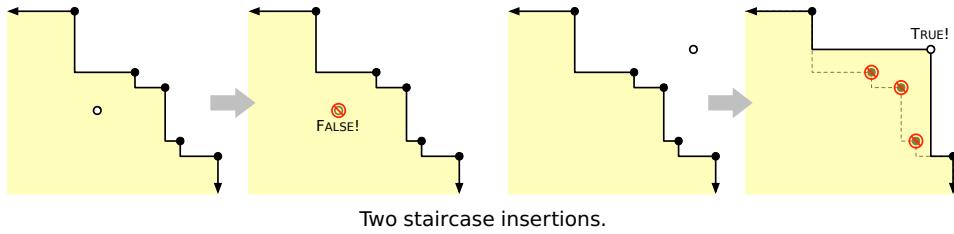
- (a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.
- (b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm `ABOVE?(x, y)` that returns `TRUE` if the point (x, y) is above the staircase, or `FALSE` otherwise. Your data structure should use $O(n)$ space, and your `ABOVE?` algorithm should run in $O(\log n)$ time.



Two staircase queries.

- (c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function `INSERT(x, y)` that adds the point

(x, y) to the underlying point set and returns TRUE or FALSE to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.



6. Say that a binary search tree is *augmented* if every node v also stores $\text{size}(v)$, the number of nodes in the subtree rooted at v .
 - (a) Show that a rotation in an augmented binary tree can be performed in constant time.
 - (b) Describe an algorithm `SCAPEGOATSELECT(k)` that selects the k th smallest item in an augmented scapegoat tree in $O(\log n)$ worst-case time. (The scapegoat trees presented in these notes are already augmented.)
 - (c) Describe an algorithm `SPLAYSELECT(k)` that selects the k th smallest item in an augmented splay tree in $O(\log n)$ amortized time.
 - (d) Describe an algorithm `TREAPSELECT(k)` that selects the k th smallest item in an augmented treap in $O(\log n)$ expected time.

7. Many applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Let T be an arbitrary binary tree. The secondary data structure at any node v stores exactly the same set of items as the subtree of T rooted at v . This secondary structure has size $O(\text{size}(v))$ and can be built in $O(\text{size}(v))$ time, where $\text{size}(v)$ denotes the number of descendants of v .

The primary and secondary data structures are typically defined by different attributes of the data being stored. For example, to store a set of points in the plane, we could define the primary tree T in terms of the x -coordinates of the points, and define the secondary data structures in terms of their y -coordinate.

Maintaining these secondary structures complicates algorithms for keeping the top-level search tree balanced. Specifically, performing a rotation at any node v in the primary tree now requires $O(\text{size}(v))$ time, because we have to rebuild one of the secondary structures (at the new child of v). When we insert a new item into T , we must also insert into one or more secondary data structures.

- (a) Overall, how much space does this data structure use in the worst case?
- (b) How much space does this structure use if the primary search tree is perfectly balanced?
- (c) Suppose the primary tree is a splay tree. Prove that the amortized cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. [Hint: This is easy!]
- (d) Now suppose the primary tree T is a scapegoat tree. How long does it take to rebuild the subtree of T rooted at some node v , as a function of $\text{size}(v)$?

- (e) Suppose the primary tree and all the secondary trees are scapegoat trees. What is the amortized cost of a single insertion?
- *(f) Finally, suppose the primary tree and every secondary tree is a treap. What is the worst-case expected time for a single insertion?
8. Let $X = \langle x_1, x_2, \dots, x_m \rangle$ be a sequence of m integers, each from the set $\{1, 2, \dots, n\}$. We can visualize this sequence as a set of integer points in the plane, by interpreting each element x_i as the point (x_i, i) . The resulting point set, which we can also call X , has exactly one point on each row of the $n \times m$ integer grid.
- (a) Let Y be an arbitrary set of integer points in the plane. Two points (x_1, y_1) and (x_2, y_2) in Y are *isolated* if (1) $x_1 \neq x_2$ and $y_1 \neq y_2$, and (2) there is no other point $(x, y) \in Y$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. If the set Y contains no isolated pairs of points, we call Y a *commune*.⁷
- Let X be an arbitrary set of points on the $n \times n$ integer grid with exactly one point per row. Show that there is a commune Y that contains X and consists of $O(n \log n)$ points.
- (b) Consider the following model of self-adjusting binary search trees. We interpret X as a sequence of accesses in a binary search tree. Let T_0 denote the initial tree. In the i th round, we traverse the path from the root to node x_i , and then *arbitrarily reconfigure* some subtree S_i of the current search tree T_{i-1} to obtain the next search tree T_i . The only restriction is that the subtree S_i must contain both x_i and the root of T_{i-1} . (For example, in a splay tree, S_i is the search path to x_i .) The *cost* of the i th access is the number of nodes in the subtree S_i .
- Prove that the minimum cost of executing an access sequence X in this model is at least the size of the smallest commune containing the corresponding point set X . [Hint: Lowest common ancestor.]
- *(c) Suppose X is a *random* permutation of the integers $1, 2, \dots, n$. Use the lower bound in part (b) to prove that the expected minimum cost of executing X is $\Omega(n \log n)$.
- ★(d) Describe a polynomial-time algorithm to compute (or even approximate up to constant factors) the smallest commune containing a given set X of integer points, with at most one point per row. Alternately, prove that the problem is NP-hard.

⁷Demaine et al. [9] refer to communes as *arborally satisfied sets*.

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, $\text{UNION}(A, \text{MAKESET}(x))$ adds a new element x to an existing set A . The sets A and B are specified by arbitrary elements, so $\text{UNION}(x, y)$ has exactly the same behavior as $\text{UNION}(\text{FIND}(x), \text{FIND}(y))$.

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.

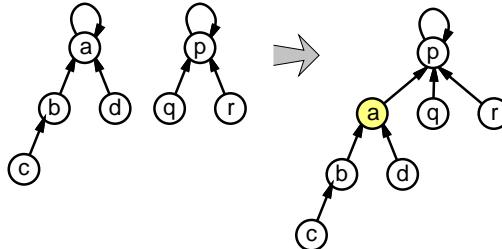
10.1 Reversed Trees

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( $x$ ):
   $\text{parent}(x) \leftarrow x$ 
```

```
FIND( $x$ ):
  while  $x \neq \text{parent}(x)$ 
     $x \leftarrow \text{parent}(x)$ 
  return  $x$ 
```

```
UNION( $x, y$ ):
   $\bar{x} \leftarrow \text{FIND}(x)$ 
   $\bar{y} \leftarrow \text{FIND}(y)$ 
   $\text{parent}(\bar{y}) \leftarrow \bar{x}$ 
```



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 $\text{FIND}(x)$ 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 $\Theta(n)$ 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 *shallow*er tree a child of the *deep*er one. This requires us to also maintain the depth of each tree, but this is quite easy.

```
MAKESET( $x$ ):
   $\text{parent}(x) \leftarrow x$ 
   $\text{depth}(x) \leftarrow 0$ 
```

```
FIND( $x$ ):
  while  $x \neq \text{parent}(x)$ 
     $x \leftarrow \text{parent}(x)$ 
  return  $x$ 
```

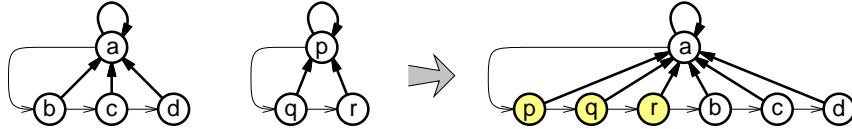
```
UNION( $x, y$ ):
   $\bar{x} \leftarrow \text{FIND}(x)$ 
   $\bar{y} \leftarrow \text{FIND}(y)$ 
  if  $\text{depth}(\bar{x}) > \text{depth}(\bar{y})$ 
     $\text{parent}(\bar{y}) \leftarrow \bar{x}$ 
  else
     $\text{parent}(\bar{x}) \leftarrow \bar{y}$ 
    if  $\text{depth}(\bar{x}) = \text{depth}(\bar{y})$ 
       $\text{depth}(\bar{y}) \leftarrow \text{depth}(\bar{y}) + 1$ 
```

With this new rule in place, it's not hard to prove by induction that for any set leader \bar{x} , the size of \bar{x} 's set is at least $2^{\text{depth}(\bar{x})}$, as follows. If $\text{depth}(\bar{x}) = 0$, then \bar{x} is the leader of a singleton set. For any $d > 0$, when $\text{depth}(\bar{x})$ becomes d for the first time, \bar{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 add elements to \bar{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 $\Theta(\log n)$ 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) $\leftarrow x$
next(x) $\leftarrow x$

FIND(x):
return leader(x)

UNION(x, y):
 $\bar{x} \leftarrow \text{FIND}(x)$
 $\bar{y} \leftarrow \text{FIND}(y)$
 $y \leftarrow \bar{y}$
leader(y) $\leftarrow \bar{x}$
while (next(y) $\neq \text{NULL}$)
 $y \leftarrow \text{next}(y)$
 leader(y) $\leftarrow \bar{x}$
next(y) $\leftarrow \text{next}(\bar{x})$
next(\bar{x}) $\leftarrow \bar{y}$

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 $\{1, 2, \dots, n\}$ from scratch.

WORSTCASESEQUENCE(n):
MAKESET(1)
for $i \leftarrow 2$ 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) $\leftarrow x$
next(x) $\leftarrow x$
size(x) $\leftarrow 1$

WEIGHTEDUNION(x, y)
 $\bar{x} \leftarrow \text{FIND}(x)$
 $\bar{y} \leftarrow \text{FIND}(y)$
if size(\bar{x}) $>$ size(\bar{y})
 UNION(\bar{x}, \bar{y})
 size(\bar{x}) $\leftarrow \text{size}(\bar{x}) + \text{size}(\bar{y})$
else
 UNION(\bar{y}, \bar{x})
 size(\bar{x}) $\leftarrow \text{size}(\bar{x}) + \text{size}(\bar{y})$

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.

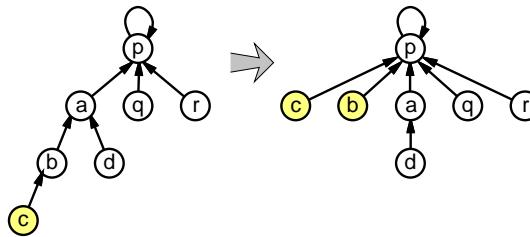
Since each WEIGHTEDUNION reduces the number of sets by one, there are $m - n$ sets at the end of the sequence, and at most n objects are *not* in singleton sets. Since each of the non-singleton objects had $O(\log n)$ leader changes, the total amount of work done in updating the leader pointers is $O(n \log n)$. \square

The aggregate method now implies that each WEIGHTEDUNION has *amortized cost* $O(\log n)$.

10.3 Path Compression

Using unthreaded trees, 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 $\text{Find}(c)$. Shaded nodes have a new parent.

```

FIND( $x$ )
  if  $x \neq \text{parent}(x)$ 
     $\text{parent}(x) \leftarrow \text{FIND}(\text{parent}(x))$ 
  return  $\text{parent}(x)$ 

```

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*. The following algorithm is usually called *union by rank*:

<pre> MAKESET(x): $\text{parent}(x) \leftarrow x$ $\text{rank}(x) \leftarrow 0$ </pre>	<pre> UNION(x, y) $\bar{x} \leftarrow \text{FIND}(x)$ $\bar{y} \leftarrow \text{FIND}(y)$ if $\text{rank}(\bar{x}) > \text{rank}(\bar{y})$ $\text{parent}(\bar{y}) \leftarrow \bar{x}$ else $\text{parent}(\bar{x}) \leftarrow \bar{y}$ if $\text{rank}(\bar{x}) = \text{rank}(\bar{y})$ $\text{rank}(\bar{y}) \leftarrow \text{rank}(\bar{y}) + 1$ </pre>
--	--

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:

$$\lg^* n = \begin{cases} 1 & \text{if } n \leq 2, \\ 1 + \lg^*(\lg n) & \text{otherwise.} \end{cases}$$

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 $\text{parent}(x)$ 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: $\text{size}(\bar{x}) \geq 2^{\text{rank}(\bar{x})}$. (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 \bar{x} changes from $r - 1$ to r , mark all the objects in \bar{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 proofs² relied on complicated charging schemes or potential-function arguments; 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.

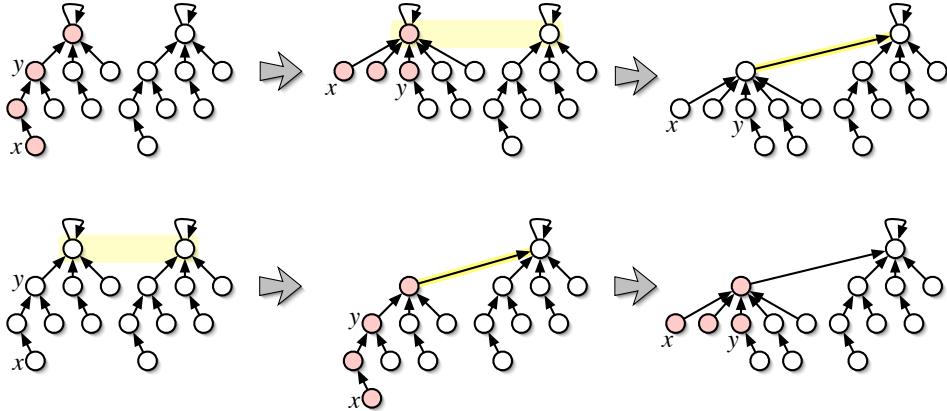
COMPRESS(x, y): $\langle\!\langle y \text{ must be an ancestor of } x \rangle\!\rangle$ if $x \neq y$ $\quad \text{COMPRESS}(\text{parent}(x), y)$ $\quad \text{parent}(x) \leftarrow \text{parent}(y)$	SHATTER(x): if $\text{parent}(x) \neq x$ $\quad \text{SHATTER}(\text{parent}(x))$ $\quad \text{parent}(x) \leftarrow x$
---	--

¹Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM J. Computing* 34(3):515–525, 2005.

²Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.* 22:215–225, 1975.

Clearly, the running time of $\text{FIND}(x)$ operation is dominated by the running time of $\text{COMPRESS}(x, y)$, 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 $T(m, n, r)$ 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 .

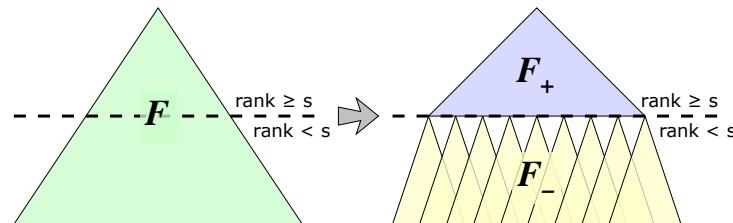
The following trivial upper bound will be the base case for our recursive argument.

Theorem 2. $T(m, n, r) \leq nr$

Proof: Each node can change parents at most r times, because each new parent has higher rank than the previous parent. \square

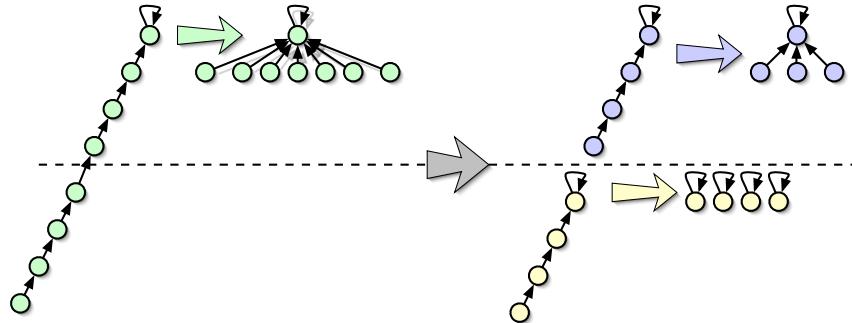
Fix a forest F of n nodes with maximum rank r , and a sequence C of m COMPRESS operations on F , and let $T(F, C)$ denote the total number of pointer assignments executed by this sequence.

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 $\text{parent}(x) \leftarrow y$ with $\text{parent}(x) \leftarrow x$.



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( $y$ ) ≤  $r$ 
    COMPRESS( $x, y, F_-$ )    «in  $C_-$ »
  else
     $z \leftarrow$  highest ancestor of  $x$  in  $F$  with rank at most  $r$ 
    COMPRESS( $\text{parent}_F(z), y, F_+$ )    «in  $C_+$ »
    SHATTER( $x, z, F_-$ )
     $\text{parent}(z) \leftarrow z$     (*)
  
```

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 $\text{parent}(x) \leftarrow y$, 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:

$$T(F, C) \leq T(F_+, C_+) + T(F_-, C_-) + m_+ + n$$

Since there are only $n/2^i$ nodes of any rank i , we have $n_+ \leq \sum_{i>s} n/2^i = n/2^s$. 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 $s = \lg r$, so that $T(F_+, C_+) \leq n$. We can now simplify our earlier recurrence to

$$T(F, C) \leq T(F_-, C_-) + m_+ + 2n,$$

or equivalently,

$$T(F, C) - m \leq T(F_-, C_-) - m_- + 2n.$$

Since this argument applies to *any* forest F and *any* sequence C , we have just proved that

$$T'(m, n, r) \leq T'(m, n, \lfloor \lg r \rfloor) + 2n,$$

where $T'(m, n, r) = T(m, n, r) - m$. The solution to this recurrence is $T'(n, m, r) \leq 2n \lg^* r$. Voilá!

Theorem 3. $T(m, n, r) \leq m + 2n \lg^* r$

*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

$$T(F_+, C_+) \leq m_+ + 2n \frac{\lg^* r}{2^{\lg^* r}} \leq m_+ + 2n.$$

This implies the recurrence

$$T(F, C) \leq T(F_-, C_-) + 2m_+ + 3n,$$

which in turn implies that

$$T''(m, n, r) \leq T''(m, n, \lg^* r) + 3n,$$

where $T''(m, n, r) = T(m, n, r) - 2m$. The solution to this equation is $T(m, n, r) \leq 2m + 3n \lg^{**} r$, where $\lg^{**} r$ is the *iterated iterated logarithm* of r :

$$\lg^{**} r = \begin{cases} 1 & \text{if } r \leq 2, \\ 1 + \lg^{**}(\lg^* r) & \text{otherwise.} \end{cases}$$

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

$$T(m, n, r) \leq cm + (c+1)n \lg^{*c} r$$

where

$$\lg^{*c} r = \begin{cases} \lg r & \text{if } c = 0, \\ 1 & \text{if } r \leq 2, \\ 1 + \lg^{*c}(\lg^{*c-1} r) & \text{otherwise.} \end{cases}$$

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 $\lg^{*c} r$ is smaller than a constant:

$$\alpha(r) = \min \left\{ c \geq 1 \mid \lg^{*c} n \leq 3 \right\}.$$

(The threshold value 3 is used here because $\lg^* 5 \geq 2$ for all c .) By setting $c = \alpha(r)$, we obtain our final upper bound.

Theorem 4. $T(m, n, r) \leq m\alpha(r) + 3n(\alpha(r) + 1)$

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 $O(\alpha(n))$ amortized time.

Even this upper bound is somewhat conservative if m is larger than n . A closer estimate is given by the function

$$\alpha(m, n) = \min \left\{ c \geq 1 \mid \log^{*c}(\lg n) \leq m/n \right\}.$$

It's not hard to prove that if $m = \Theta(n)$, then $\alpha(m, n) = \Theta(\alpha(n))$. On the other hand, if $m \geq n \lg^{****} n$, for any constant number of stars, then $\alpha(m, n) = O(1)$. 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

$$2 \uparrow^c n = \begin{cases} 2 & \text{if } n = 1 \\ 2n & \text{if } c = 0 \\ 2 \uparrow^{c-1} (2 \uparrow^c (n-1)) & \text{otherwise} \end{cases}$$

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 $2^{2^{\dots^{2}}}_n$; this function is also sometimes called *tetration*. John Conway named $2 \uparrow\uparrow\uparrow n$ the *wower* function: $2 \uparrow\uparrow\uparrow n = \underbrace{2 \uparrow\uparrow 2 \uparrow\uparrow \dots \uparrow\uparrow 2}_n$. And so on, *et cetera, ad infinitum*.

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 $n \leq 2^{256}$, we have $\log^* n \leq 5$, $\log^{**} n \leq 4$, and $\log^{*c} n \leq 3$ for any $c \geq 3$.

The function $\alpha(n)$ is usually called the *inverse Ackerman function*.⁵ Our earlier definition is equivalent to $\alpha(n) = \min\{c \geq 1 \mid 2 \uparrow^{c+2} 3 \geq n\}$; 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

³Robert E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Comput. Syst. Sci.* 19:110–127, 1979.

⁴Ackermann 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.

⁵Strictly speaking, the name ‘inverse Ackerman function’ is inaccurate. One good formal definition of the true inverse Ackerman function is $\tilde{\alpha}(n) = \min \{c \geq 1 \mid \lg^{*c} n \leq c\} = \min \{c \geq 1 \mid 2 \uparrow^{c+2} c \geq n\}$. However, it's not hard to prove that $\tilde{\alpha}(n) \leq \alpha(n) \leq \tilde{\alpha}(n) + 1$ 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!

$\alpha(n) \leq 3$ for all even *remotely imaginable* values of n . Nevertheless, the function $\alpha(n)$ is eventually larger than any constant, so it is *not* $O(1)$.

$2 \uparrow^c n$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
$2n$	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	$2^{2^{2^{\cdot^{\cdot^2}}}}_{65536}$	$2^{2^{2^{\cdot^{\cdot^2}}}}_{2^{2^{2^{\cdot^2}}}}_{65536}$
$2 \uparrow\uparrow\uparrow\uparrow n$	2	4	$2^{2^{2^{\cdot^2}}}_{65536}$	$2^{2^{2^{\cdot^2}}}_{2^{2^{\cdot^2}}}_{65536}$	$2^{2^{2^{\cdot^2}}}_{2^{2^{2^{\cdot^2}}}}_{65536}$ «Yeah, right.»
$2 \uparrow\uparrow\uparrow\uparrow\uparrow n$	2	4	$2^{2^{2^{\cdot^2}}}_{65536}$	$2^{2^{2^{\cdot^2}}}_{2^{2^{\cdot^2}}}_{65536}$	$2^{2^{2^{\cdot^2}}}_{2^{2^{2^{\cdot^2}}}}_{65536}$ «Very funny.»
					«Argh! My eyes!»

Small (!!?) values of Ackermann's functions.

10.7 To infinity... and beyond!

Of course, one can generalize the inverse Ackermann function to functions that grow arbitrarily more slowly, starting with the *iterated* inverse Ackermann function

$$\alpha^*(n) = \begin{cases} 1 & \text{if } n \leq 4 \\ 1 + \alpha^*(\alpha(n)) & \text{otherwise,} \end{cases}$$

then the *iterated* iterated inverse Ackermann function

$$\alpha^{**}(n) = \begin{cases} 1 & \text{if } n \leq 4 \\ 1 + \alpha^{**}(\alpha(n)) & \text{otherwise,} \end{cases}$$

and then the diagonalized inverse Ackermann function

$$\text{Head-asplode}(n) = \min\{c \geq 1 \mid \alpha^{*^c} n \leq 4\},$$

and so on forever. Fortunately(?), such functions appear extremely rarely in algorithm analysis. In fact, the only example (as far as Jeff knows) of a naturally-occurring super-constant sub-inverse-Ackermann function is a recent result of Seth Pettie⁶, who proved that if a splay tree is used as a double-ended queue — insertions and deletions of only smallest or largest elements — then the amortized cost of any operation is $O(\alpha^*(n))$.

Exercises

- Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader \bar{x} stores the number of elements of its set in the field $\text{weight}(\bar{x})$. Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

⁶Splay trees, Davenport-Schinzel sequences, and the deque conjecture. *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1115–1124, 2008.

$\text{MAKESET}(x):$ $\text{parent}(x) \leftarrow x$ $\text{weight}(x) \leftarrow 1$	$\text{UNION}(x, y)$ $\bar{x} \leftarrow \text{FIND}(x)$ $\bar{y} \leftarrow \text{FIND}(y)$ $\text{if } \text{weight}(\bar{x}) > \text{weight}(\bar{y})$ $\quad \text{parent}(\bar{y}) \leftarrow \bar{x}$ $\quad \text{weight}(\bar{x}) \leftarrow \text{weight}(\bar{x}) + \text{weight}(\bar{y})$ else $\quad \text{parent}(\bar{x}) \leftarrow \bar{y}$ $\quad \text{weight}(\bar{x}) \leftarrow \text{weight}(\bar{x}) + \text{weight}(\bar{y})$
$\text{FIND}(x):$ $\text{while } x \neq \text{parent}(x)$ $\quad x \leftarrow \text{parent}(x)$ $\text{return } x$	

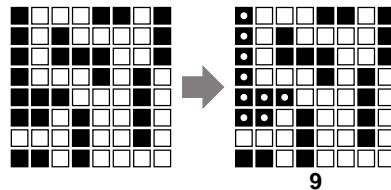
Prove that if we use union-by-weight, the *worst-case* running time of $\text{FIND}(x)$ is $O(\log n)$, where n is the cardinality of the set containing x .

2. Consider a union-find data structure that uses union by depth (or equivalently union by rank) *without* path compression. For all integers m and n such that $m \geq 2n$, prove that there is a sequence of n MakeSet operations, followed by m UNION and FIND operations, that require $\Omega(m \log n)$ time to execute.
3. Consider an arbitrary sequence of m MAKESET operations, followed by u UNION operations, followed by f FIND operations, and let $n = m + u + f$. Prove that if we use union by rank and FIND with path compression, all n operations are executed in $O(n)$ time.
4. Describe and analyze a data structure to support the following operations on an array $X[1..n]$ as quickly as possible. Initially, $X[i] = 0$ for all i .
 - Given an index i such that $X[i] = 0$, set $X[i]$ to 1.
 - Given an index i , return $X[i]$.
 - Given an index i , return the smallest index $j \geq i$ such that $X[j] = 0$, or report that no such index exists.

For full credit, the first two operations should run in *worst-case constant* time, and the amortized cost of the third operation should be as small as possible.

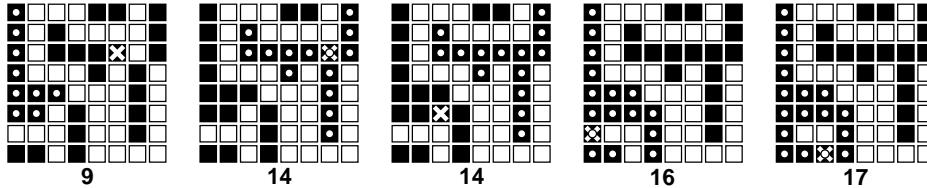
5. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

For example, given the bitmap below as input, your algorithm should return the number 9, because the largest connected black component (marked with white dots on the right) contains nine pixels.



- (b) Design and analyze an algorithm $\text{BLACKEN}(i, j)$ that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.



(c) What is the *worst-case* running time of your BLACKEN algorithm?

6. Consider the following game. I choose a positive integer n and keep it secret; your goal is to discover this integer. We play the game in rounds. In each round, you write a list of *at most* n integers on the blackboard. If you write more than n numbers in a single round, you lose. (Thus, in the first round, you must write only the number 1; do you see why?) If n is one of the numbers you wrote, you win the game; otherwise, I announce which of the numbers you wrote is smaller or larger than n , and we proceed to the next round. For example:

You	Me
1	It's bigger than 1.
4, 42	It's between 4 and 42.
8, 15, 16, 23, 30	It's between 8 and 15.
9, 10, 11, 12, 13, 14	It's 11; you win!

Describe a strategy that allows you to win in $O(\alpha(n))$ rounds!

Why are our days numbered and not, say, lettered?

— Woody Allen

G String Matching

G.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 integer¹, 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 ‘AMANAPLANACATALANPALANA’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  $s \leftarrow 1$  to  $n - m + 1$ 
    equal  $\leftarrow$  true
     $i \leftarrow 1$ 
    while equal and  $i \leq m$ 
        if  $T[s + i - 1] \neq P[i]$ 
            equal  $\leftarrow$  false
        else
             $i \leftarrow i + 1$ 
    if equal
        return  $s$ 
return ‘none’
```

¹Yes, *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 ‘`¶ = ¶[∞++]/¶;`’ in your Java code! Joy!

²Dan Hoey (or rather, his computer program) found the following 540-word palindrome in 1984:

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 zog, 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!

Peter Norvig generated even longer examples in 2002; see <http://norvig.com/palindrome.html>.

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 entirely 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.

G.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 :

$$p = \sum_{i=1}^m 10^{m-i} \cdot P[i] \quad t_s = \sum_{i=1}^m 10^{m-i} \cdot T[s+i-1]$$

For example, if $T = 31415926535897932384626433832795028841971$ 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*:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10 \cdot P[1]) \cdots))$$

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 $T[r+m]$:

$$t_{s+1} = 10(t_s - 10^{m-1} \cdot T[s]) + T[s+m]$$

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  $s$ 
     $t_{s+1} \leftarrow 10 \cdot (t_s - \sigma \cdot T[s]) + T[s+m]$ 
  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!

G.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 '

$$p \bmod q = P[m] + (\dots + (10 \cdot (P[2] + (10 \cdot P[1] \bmod q) \bmod q) \bmod q) \dots) \bmod q$$

and similarly, given $(t_s \bmod q)$, we can compute $(t_{s+1} \bmod q)$ in constant time.

$$t_{s+1} \bmod q = (10 \cdot (t_s - ((10^{m-1} \bmod q) \cdot T[s] \bmod q) \bmod q) \bmod q) + T[s+m] \bmod q$$

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 $\tilde{t}_s = t_s \bmod q$.)

```

KARP RABIN( $T[1..n], P[1..m]$ ):
    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  $s$ 
             $\tilde{t}_{s+1} \leftarrow (10 \cdot (\tilde{t}_s - (\sigma \cdot T[s] \bmod q) \bmod q) \bmod q) + T[s+m] \bmod q$ 
    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.

G.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 KARP RABIN 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. \square

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.

The probability of getting a false match is $O(1/m)$.

Recall that the running time of KARP RABIN is $O(n + mF)$, where F is the number of false matches. By using the *really* loose upper bound $E[F] \leq \Pr[F > 0] \cdot n$, we can conclude that the expected number of false matches is $O(n/m)$. Thus, the expected running time of the KARP RABIN algorithm is $O(n)$.

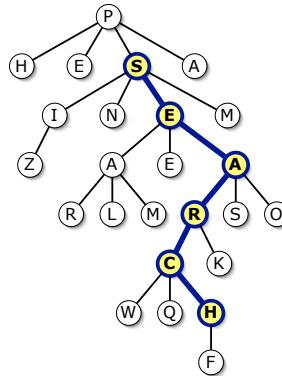
G.5 Random Prime Number?

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 the lecture notes on number-theoretic algorithms, or in the textbook *Randomized Algorithms* by Rajeev Motwani and Prabhakar Raghavan (Cambridge, 1995).

Exercises

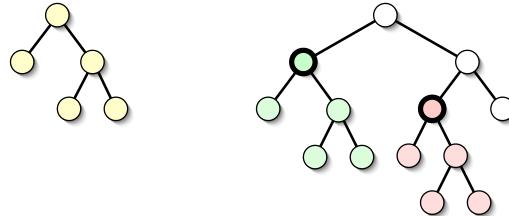
1. Describe and analyze a two-dimensional variant of KARP RABIN that searches for a given two-dimensional pattern $P[1..p][1..q]$ within a given two-dimensional ‘text’ $T[1..m][1..n]$. Your algorithm should report *all* index pairs (i, j) such that the subarray $T[i..i+p-1][j..j+q-1]$ is identical to the given pattern, in $O(pq + mn)$ expected time.

2. Describe and analyze a variant of KARP-RABIN 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 KARP-RABIN 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 feet 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)

H More String Matching

H.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	ZADABRA
	ABRACADABRA

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	ZADABRA
ABR	ACADABRA
ABRA	CABDABRA
ABRAC	ACADABRA
ABRACAD	ABDABRA
ABRACADA	BABA
ABRACADAB	RA
ABRACADABA	

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 $T[15] = P[4] = A$. 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.

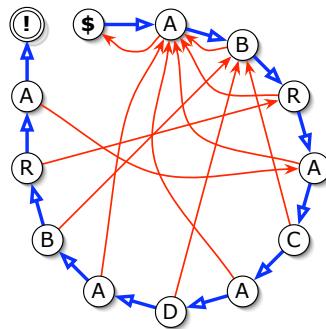
If you play with this idea long enough, you'll notice that the character comparisons should always advance through the text. **Once we've found a match for a text character, we never need to do another comparison with that text character again.** In other words, we should be able to optimize the brute-force 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.

H.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 $\$$ 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 $\$$ 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 $\$$, follow the success edge to the next node and increment i . (So there is no failure edge from the start node $\$$.)
- 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 $\text{fail}[1..n]$, so that for each j there is a failure edge from node j to node $\text{fail}[j]$. Following a failure edge back to an earlier state exactly corresponds, in our earlier formulation, to shifting the pattern forward. The failure function $\text{fail}[j]$ 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 KNUTHMORRIS-PRATT 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 $O(n)$.

H.3 Computing the Failure Function

We can now rephrase our second intuitive rule about how to choose a reasonable shift after a character mismatch $T[i] \neq P[j]$:

$P[1..fail[j] - 1]$ 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:

$P[1..fail[j] - 1]$ 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 $fail[j]$ for all $j > 1$.¹ By convention we set $fail[1] = 0$; 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
$fail[i]$	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!

¹Many 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 $\pi[j] = fail[j + 1] - 1$. The off-by-one difference between the two functions adds a few extra +1s 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.)

$j \leftarrow 0, i \leftarrow 1$ $fail[i] \leftarrow j$	\$ ^j A ⁱ B R A C A D A B R X ... 0
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$ $j \leftarrow fail[j]$	\$ A ^j B ⁱ R A C A D A B R X ... 0 1 \$ A B ⁱ R A C A D A B R X ...
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$ $j \leftarrow fail[j]$	\$ A ^j B R ⁱ A C A D A B R X ... 0 1 1 \$ A B R ⁱ A C A D A B R X ...
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$ $j \leftarrow fail[j]$	\$ A ^j B R A ⁱ C A D A B R X ... 0 1 1 1 \$ A B R A C A D A B R X ...
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$ $j \leftarrow fail[j]$	\$ A B ^j R A C ⁱ A D A B R X ... 0 1 1 1 2 \$ A B R A C ⁱ A D A B R X ... \$ A B R A C ⁱ A D A B R X ...
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$	\$ A ^j B R A C A ⁱ D A B R X ... 0 1 1 1 2 1 \$ A B R A C A D ⁱ A B R X ... \$ A B R A C A D ⁱ A B R X ...
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$ $j \leftarrow fail[j]$	\$ A ^j B R A C A D A ⁱ B R X ... 0 1 1 1 2 1 2 1 \$ A B R A C A D A B ⁱ R X ... \$ A B R A C A D A B 2 R X ...
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$ $j \leftarrow fail[j]$	\$ A B ^j R A C A D A B R ⁱ X ... 0 1 1 1 2 1 2 1 2 1 2 3 \$ A B R A C A D A B R ⁱ X ... \$ A B R A C A D A B R 3 X ...
$j \leftarrow j + 1, i \leftarrow i + 1$ $fail[i] \leftarrow j$ $j \leftarrow fail[j]$ $j \leftarrow fail[j]$	\$ A B R A ^j C A D A B R X ⁱ ... 0 1 1 1 2 1 2 1 2 1 2 3 4 \$ A B R A C A D A B R X ⁱ ... \$ 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 i th 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

$$fail^c[j] = fail[\overbrace{fail^{c-1}[j]}^c] = \overbrace{fail[\cdots [fail[j]] \cdots]}^c.$$

In particular, if $\text{fail}^{c-1}[j] = 0$, then $\text{fail}^c[j]$ is undefined. We can easily show by induction that every string of the form $P[1.. \text{fail}^c[j]-1]$ 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 $P[1.. \text{fail}^c[j]]$ — **that is**, the one with smallest c — such that $P[\text{fail}^c[j]] = P[i]$. This is exactly what the while loop in COMPUTEFAILURE computes; the $(c+1)$ th iteration compares $P[\text{fail}^c[j]] = P[\text{fail}^{c+1}[i]]$ against $P[i]$. COMPUTEFAILURE is actually a *dynamic programming* implementation of the following recursive definition of $\text{fail}[i]$:

$$\text{fail}[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max_{c \geq 1} \{ \text{fail}^c[i-1] + 1 \mid P[i-1] = P[\text{fail}^c[i-1]] \} & \text{otherwise.} \end{cases}$$

H.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 $P[\text{fail}[j]]$. With the current definition, however, it is possible that $P[j]$ and $P[\text{fail}[j]]$ are actually the same character, in which case the next character comparison will automatically fail. So why do the comparison at all?

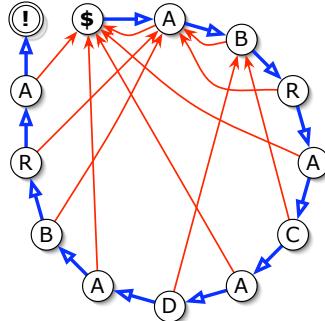
We can optimize the failure function by ‘short-circuiting’ these redundant comparisons with some simple post-processing:

```
OPTIMIZEFAILURE( $P[1..m]$ ,  $\text{fail}[1..m]$ ):
    for  $i \leftarrow 2$  to  $m$ 
        if  $P[i] = P[\text{fail}[i]]$ 
             $\text{fail}[i] \leftarrow \text{fail}[\text{fail}[i]]$ 
```

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  $P[i] = P[j]$ 
             $\text{fail}[i] \leftarrow \text{fail}[j]$ 
        else
             $\text{fail}[i] \leftarrow j$ 
        while  $j > 0$  and  $P[i] \neq P[j]$ 
             $j \leftarrow \text{fail}[j]$ 
         $j \leftarrow j + 1$ 
```

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
unoptimized $fail[i]$	0	1	1	2	3	4	1	2	3	4	5
optimized $fail[i]$	0	1	0	1	0	4	0	1	0	1	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'.

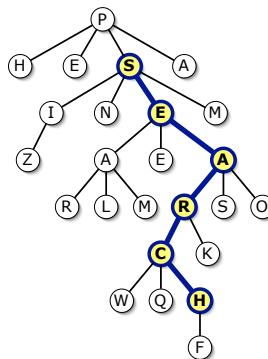
$P[i]$	A	B	B	A	B	B	A	B	A	B	B	A	B
unoptimized $fail[i]$	0	1	1	1	2	3	4	5	6	2	3	4	5
optimized $fail[i]$	0	1	1	0	1	1	0	1	6	1	1	0	1

Failure functions for 'ABBABBABABBAB'.

Exercises

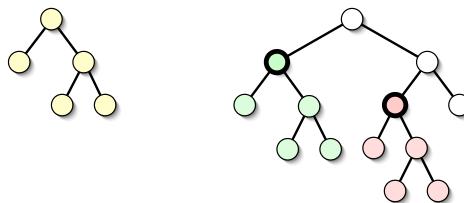
1. A *palindrome* is any string that is the same as its reversal, such as X, ABBA, or REDIVIDER. Describe and analyze an algorithm that computes the longest palindrome that is a (not necessarily proper) prefix of a given string $T[1..n]$. Your algorithm should run in $O(n)$ time.
2. Describe a modification 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 SCHABRAINCADBRANCH; 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 SCHABRUCADIBRANCH. 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=SP0C=S** appears in the texts WHUHOCUSPOCUSOT and ABRAHOCASPOCASCADABRA, but *not* in the text FRISHOCUSPOCESTIX. 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* $j \rightarrow \text{fail}[j] \rightarrow \text{fail}[\text{fail}[j]] \rightarrow \text{fail}[\text{fail}[\text{fail}[j]]] \rightarrow \dots \rightarrow 0$, 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 $O(\log m)$.

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)

*I study my Bible as I gather apples.
First I shake the whole 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

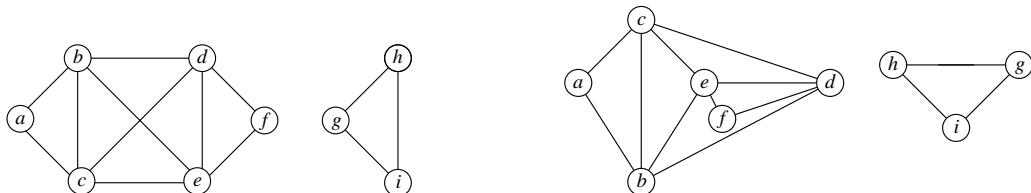
11 Basic Graph Properties

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 $0 \leq E \leq \binom{V}{2}$, and in a directed graph, $0 \leq E \leq V(V - 1)$.

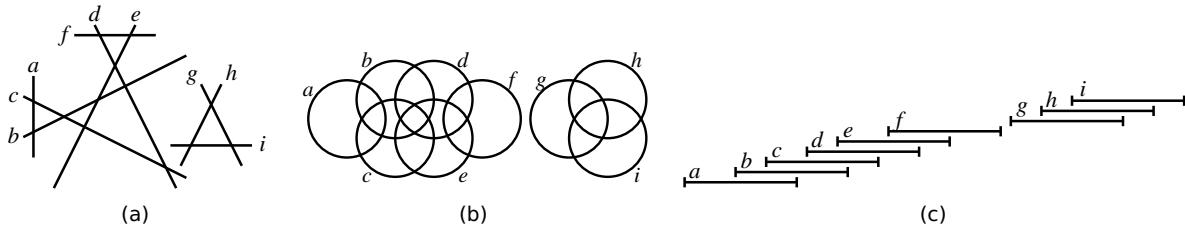
We usually visualize graphs by looking at an *embedding*. An embedding of a graph maps each vertex to a point in the plane and each edge to a curve or straight line segment between the two vertices. A graph is *planar* if it has an embedding where no two edges cross. The same graph can have many different 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.

¹The singular of 'vertices' is **vertex**. The singular of 'matrices' is **matrix**. Unless you're speaking Italian, there is no such thing as a vertex, a matrice, an indice, an appendice, a helice, an apice, a vortice, a radice, a simplice, 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.

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 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* of v and vice versa. The *degree* of a node is the number of neighbors. In directed graphs, we have two kinds of neighbors. If $u \rightarrow v$ 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

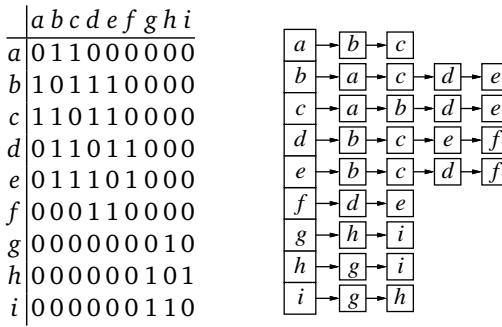
There are two common data structures used to explicitly represent graphs: *adjacency matrices*² and *adjacency lists*.

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:

$$A[i, j] = [(i, j) \in E].$$

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.



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 stored 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 when we locate the edge or when we fall off 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

²See footnote 1.

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 (\emptyset, s) in bag while the bag is not empty take (p, v) from the bag (*) if v is unmarked mark v $parent(v) \leftarrow p$ for each edge (v, w) (†) put (v, w) into the bag (**)

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 $(v, parent(v))$ with $parent(v) \neq \emptyset$ 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 $s \rightarrow \dots \rightarrow u \rightarrow v$ 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.

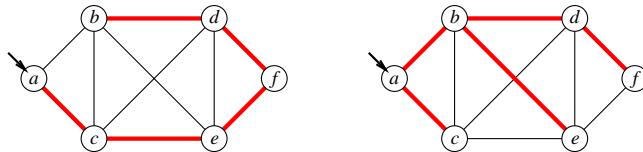
Call an edge $(v, \text{parent}(v))$ with $\text{parent}(v) \neq \emptyset$ a *parent edge*. For any node v , the path of parent edges $v \rightarrow \text{parent}(v) \rightarrow \text{parent}(\text{parent}(v)) \rightarrow \dots$ eventually leads back to s , so the set of parent edges form a connected 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. \square

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 (\dagger) 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 (\dagger) 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 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 breadth-first spanning tree depends on the order in which neighbor edges are pushed onto the queue, but in general, shortest path trees are short and bushy. We'll see shortest paths again in a future lecture.



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 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 instead of an adjacency list, finding all the neighbors of each vertex in line (†) takes $O(V)$ time. Thus, depth- and breadth-first search each 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 $\text{TRAVERSE}(s)$ 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
             $\text{TRAVERSE}(v)$ 
```

Exercises

1. Prove that the following definitions are all 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.
2. Prove that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words ‘tree’ or ‘leaf’, or any well-known properties of trees; your proof should follow entirely from the definitions.
3. Let G be a connected graph, and let T be a depth-first spanning tree of G rooted at some node v . Prove that if T is also a breadth-first spanning tree of G rooted at v , then $G = T$.
4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B , which pecks pigeon C , which pecks pigeon A .
 - (a) Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. Pretty please.
 - (b) Suppose you are given a directed graph representing the pecking relationships among a set of n pigeons. The graph contains one vertex per pigeon, and it contains an edge $i \rightarrow j$ if and only if pigeon i pecks pigeon j . Describe and analyze an algorithm to compute a pecking order for the pigeons, as guaranteed by part (a).

5. You are helping a group of ethnographers analyze some oral history data they have collected by interviewing members of a village to learn about the lives of people lived there over the last two hundred years. From the interviews, you have learned about a set of people, all now deceased, whom we will denote P_1, P_2, \dots, P_n . The ethnographers have collected several facts about the lifespans of these people. Specifically, for some pairs (P_i, P_j) , the ethnographers have learned one of the following facts:
- P_i died before P_j was born.
 - P_i and P_j were both alive at some moment.

Naturally, the ethnographers are not sure that their facts are correct; memories are not so good, and all this information was passed down by word of mouth. So they'd like you to determine whether the data they have collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they have learned simultaneously hold.

Describe and analyze an algorithm to answer the ethnographers' problem. Your algorithm should either output possible dates of birth and death that are consistent with all the stated facts, or it should report correctly that no such dates exist.

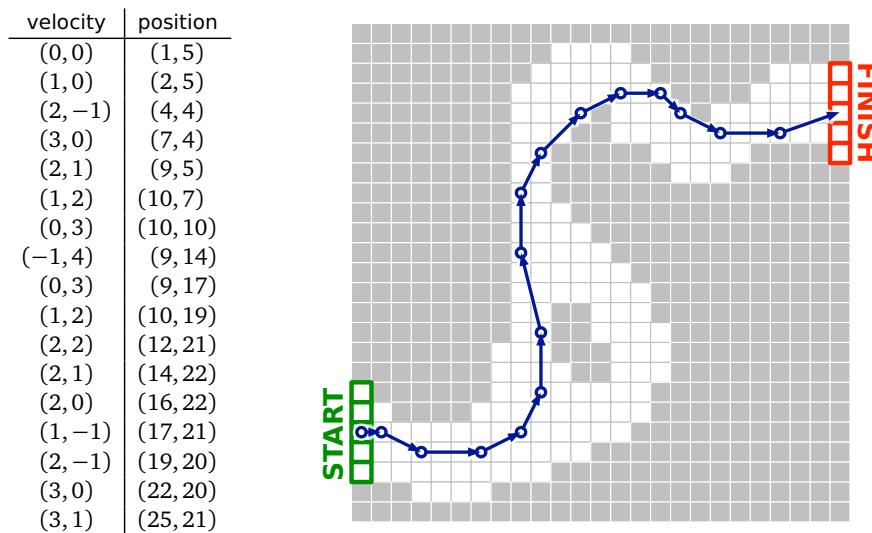
6. Let $G = (V, E)$ be a given directed graph.
- The *transitive closure* G^T is a directed graph with the same vertices as G , that contains any edge $u \rightarrow v$ if and only if there is a directed path from u to v in G . Describe an efficient algorithm to compute the transitive closure of G .
 - The *transitive reduction* G^{TR} is the smallest graph (meaning fewest edges) whose transitive closure is G^T . Describe an efficient algorithm to compute the transitive reduction of G .
7. A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R , such that every edge has one vertex in L and the other in R .
- Prove that every tree is a bipartite graph.
 - Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.
8. An *Euler tour* of a graph G is a closed walk through G that traverses every edge of G exactly once.
- Prove that a connected graph G has an Euler tour if and only if every vertex has even degree.
 - Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such graph exists.
9. The d -dimensional hypercube is the graph defined as follows. There are $2d$ vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.
- A Hamiltonian cycle in a graph G is a cycle of edges in G that visits every vertex of G exactly once. Prove that for all $d \geq 2$, the d -dimensional hypercube has a Hamiltonian cycle.
 - Which hypercubes have an Euler tour (a closed walk that traverses every edge exactly once)? [Hint: This is very easy.]

10. **Racetrack** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.³ The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

Each car has a *position* and a *velocity*, both with integer *x*- and *y*-coordinates. The initial position is a point on the starting line, chosen by the player; the initial velocity is always $(0, 0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position *on* the finish line.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting line' is the first column, and the 'finish line' is the last column.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. [Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]



A 16-step Racetrack run, on a 25×25 track. This is *not* the shortest run on this track.

- *11. Draughts/checkers is a game played on an $m \times m$ grid of squares, alternately colored light and dark. (The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size.) Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ('White') moves the white pieces; the other ('Black') moves the black pieces.

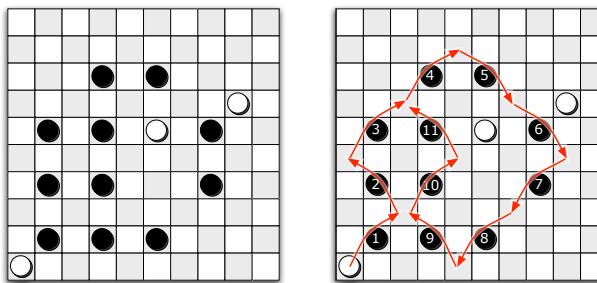
Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.⁴ Pieces can be moved in any of the four diagonal

³The actual game is a bit more complicated than the version described here. In particular, in the actual game, the boundaries of the track are a free-form curve, and (at least by default) the entire line segment between any two consecutive positions must lie inside the track. In the version Jeff played, if a car does run off the track, the car starts its next turn with zero velocity, at the legal grid point closest to where the car left the track.

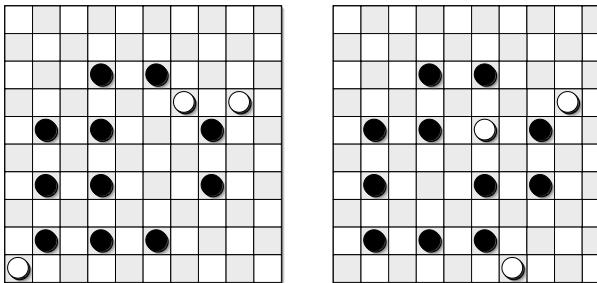
⁴Most other variants of draughts have 'flying kings', which behave *very* differently than what's described here.

directions, either one or two steps at a time. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her checkers. In a single jump, a piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. Multiple jumps are allowed in a single turn as long as they are made by the same piece. A player wins if her opponent has no pieces left on the board.

Describe an algorithm that correctly determines whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces.



White wins in one turn.



White cannot win in one turn from either of these positions.

[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does **not** always find a winning sequence of jumps even when one exists. See problem 8. Parity, parity, parity.]

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

A ship in port is safe, but that is not what ships are for.

— Rear Admiral Grace Murray Hopper

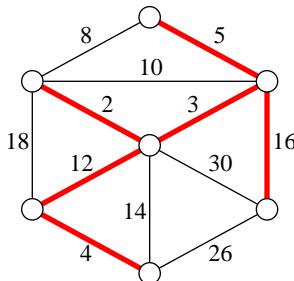
12 Minimum Spanning Trees

12.1 Introduction

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w: E \rightarrow \mathbb{R}$ 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 , that is, 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.

<code>SHORTEREDGE(i, j, k, l)</code> 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) $\langle\langle$ if $\max(i, j) < \max(k, l)$ $\rangle\rangle$ 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.

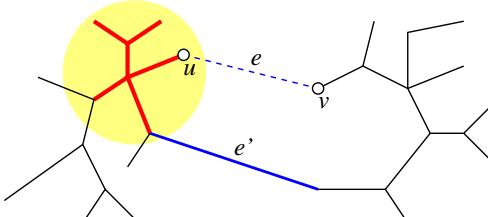
The intermediate spanning forest F induces two special types of edges. An edge is *useless* if it is not an edge of F , but both its endpoints are in the same component of F . For each component of F , we associate a *safe* edge—the minimum-weight edge with exactly one endpoint in that component. Different 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 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.

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.

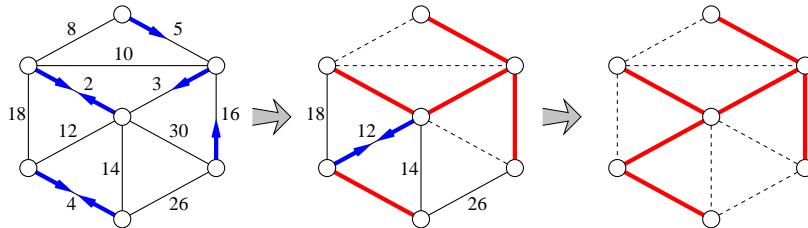
¹This lemma is actually a special case of two more general theorems. First, for any partition of the vertices of G into two disjoint subsets, the minimum-weight edge with one endpoint in each subset is in the minimum spanning tree. Second, the maximum-weight edge in any cycle in G is **not** in the minimum spanning tree.

12.3 Boruvka's Algorithm

The oldest and arguably simplest minimum spanning tree algorithm was discovered by Boruvka 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 is often called ‘Sollin’s algorithm’, especially in the parallel computing literature.

The Boruvka/Choquet/Florek/Łukaziewicz/Perkal/Stienhaus/Zubrzycki/Sollin algorithm can be summarized in one line:

BORUVKA: Add all the safe edges and recurse.



Boruvka'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 Boruvka 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 .

BORUVKA(V, E):
 $F = (V, \emptyset)$
while F has more than one component
choose leaders using DFS
FINDSAFEEDGES(V, E)
for each leader \bar{v}
add safe(\bar{v}) to F

FINDSAFEEDGES(V, E):
for each leader \bar{v}
safe(\bar{v}) $\leftarrow \infty$
for each edge $(u, v) \in E$
 $\bar{u} \leftarrow \text{leader}(u)$
 $\bar{v} \leftarrow \text{leader}(v)$
if $\bar{u} \neq \bar{v}$
if $w(u, v) < w(\text{safe}(\bar{u}))$
safe(\bar{u}) $\leftarrow (u, v)$
if $w(u, v) < w(\text{safe}(\bar{v}))$
safe(\bar{v}) $\leftarrow (u, v)$

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 BORUVKA 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 Boruvka’s algorithm is $O(E \log V)$.

Despite its relatively obscure origin, early algorithms researchers were aware of Boruvka’s algorithm, but dismissed it as being ‘too complicated’! As a result, despite its simplicity and efficiency, Boruvka’s algorithm is rarely mentioned in algorithms and data structures textbooks. On the other hand, more

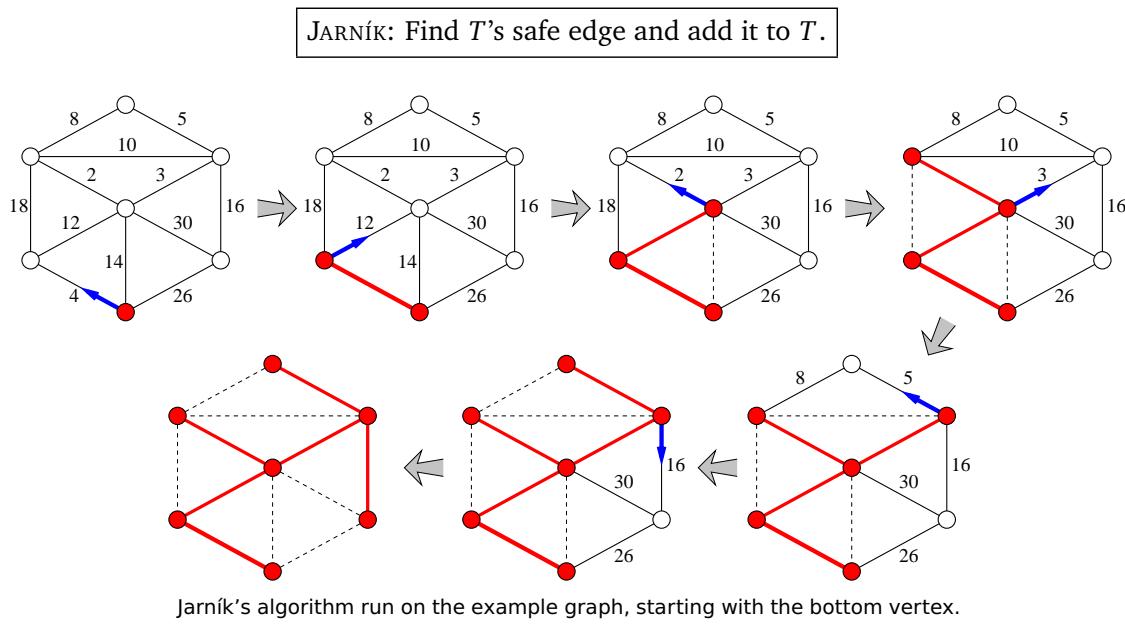
²Leonard Euler published the first graph theory result, his famous theorem about the bridges of Königsburg, in 1736. However, the first textbook on graph theory, written by Dénes König, was not published until 1936.

recent algorithms to compute minimum spanning trees are all generalizations of Boruvka's algorithm, not the other two classical algorithms described next.

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 Boruvka. 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 called 'Prim's algorithm', or sometimes even 'the Prim/Dijkstra algorithm', even though by 1958 Dijkstra already had another algorithm (inappropriately) named after him.

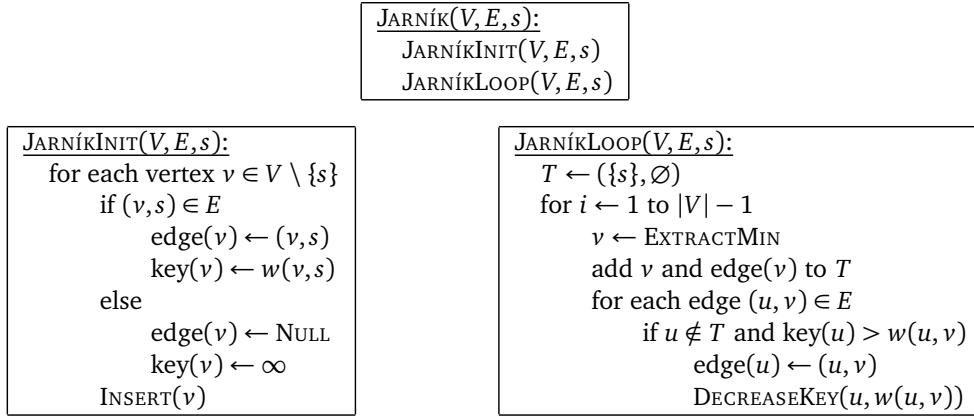
In Jarník's algorithm, the forest F contains only one nontrivial component T ; all the other components are isolated vertices. Initially, T consists of an arbitrary vertex of the graph. The algorithm repeats the following step until T spans the whole graph:



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 . 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 .

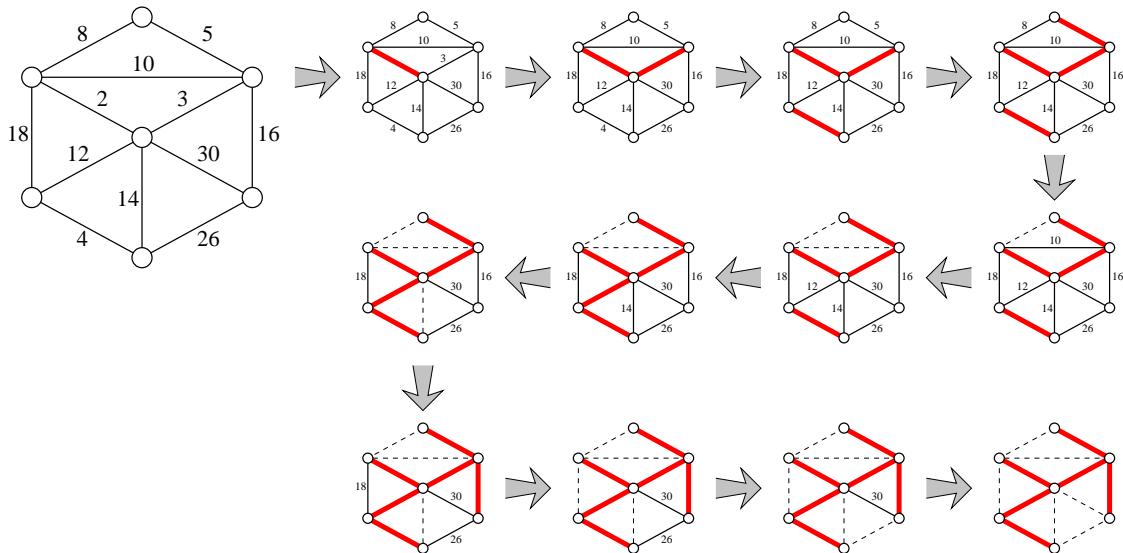


The running time of JARNÍK is dominated by the cost of the heap operations INSERT, EXTRACTMIN, and DECREASEKEY. INSERT and EXTRACTMIN are each called $O(V)$ times once for each vertex except s , and DECREASEKEY is called $O(E)$ times, at most twice for each edge. If we use a standard binary heap, each of these operations requires $O(\log V)$ time, so the overall running time of JARNÍK is $O((V + E)\log V) = O(E \log V)$. The running time can be improved to $O(E + V \log V)$ using a data structure called a *Fibonacci heap*, which supports INSERT and DECREASEKEY in constant *amortized* time; this is faster than Boruvka'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 Boruvka's original paper, claiming that Boruvka's algorithm was 'unnecessarily elaborate'.³ This algorithm was also rediscovered in 1957 by Loberman and Weinberger, but somehow avoided being renamed after them.

KRUSKAL: Scan all edges in increasing weight order; if an edge is safe, add it to F .



Kruskal's algorithm run on the example graph. Thick edges are in F . Dashed edges are useless.

³To be fair, Boruvka's original paper was unnecessarily elaborate, but in his followup paper, also published in 1927, simplified his algorithm to its current modern form. Kruskal was apparently unaware of Boruvka's second paper. Stupid Iron Curtain.

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 Borùvka'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 Borùvka'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.⁴ 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 weight
     $F \leftarrow \emptyset$ 
    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 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 $O(E \alpha(E, V))$.

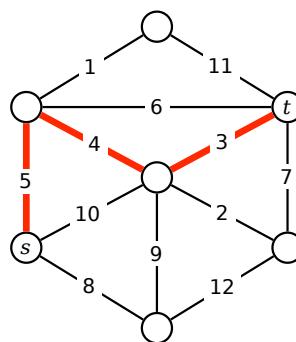
We need $O(E \log E) = O(E \log V)$ additional time just to sort the edges. Since this is bigger than the time for the UNION-FIND data structure, the overall running time of Kruskal's algorithm is $O(E \log V)$, exactly the same as Borùvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

Exercises

1. Most classical minimum-spanning-tree algorithms use the notions of 'safe' and 'useless' edges described in the lecture notes, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is *dangerous* if it is the longest edge in some cycle in G , and *useful* if it does not lie in any cycle in G .
 - (a) Prove that the minimum spanning tree of G contains every useful edge.
 - (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.
 - (c) Describe and analyze an efficient implementation of the "anti-Kruskal" MST algorithm: Examine the edges of G in decreasing order; if an edge is dangerous, remove it from G . [Hint: It won't be as fast as Kruskal's algorithm.]

⁴Live at the Assembly Hall! Only \$49.95 on Pay-Per-View!

2. Let $G = (V, E)$ be an arbitrary connected graph with weighted edges.
- Prove that for any partition of the vertices V into two subsets, the minimum-weight edge with one endpoint in each subset is in the minimum spanning tree of G .
 - Prove that the maximum-weight edge in any cycle of G is *not* in the minimum spanning tree of G .
 - Prove or disprove: The minimum spanning tree of G includes the minimum-weighted edge in *every* cycle in G .
3. Throughout this lecture note, we assumed that no two edges in the input graph have equal weights, which implies that the minimum spanning tree is unique. In fact, a weaker condition on the edge weights implies MST uniqueness.
- Describe an edge-weighted graph that has a unique minimum spanning tree, even though two edges have equal weights.
 - Prove that an edge-weighted graph G has a *unique* minimum spanning tree if and only if the following conditions hold:
 - For any partition of the vertices of G into two subsets, the minimum-weight edge with one endpoint in each subset is unique.
 - The maximum-weight edge in any cycle of G is unique.
 - Describe and analyze an algorithm to determine whether or not a graph has a unique minimum spanning tree.
4. (a) Describe and analyze an algorithm to compute the *maximum*-weight spanning tree of a given edge-weighted graph.
- (b) A *feedback edge set* of a graph G is a subset F of the edges such that every cycle in G contains at least one edge in F . In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of a given edge-weighted graph.
5. Consider a path between two vertices s and t in an undirected weighted graph G . The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t . (If there are no paths from s to t , the bottleneck distance between s and t is ∞ .)



The bottleneck distance between s and t is 5.

Describe and analyze an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

6. Suppose you are given a graph G with weighted edges and a minimum spanning tree T of G .
 - (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.
 - (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. [Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]
7. (a) Describe and analyze an algorithm to find the *second smallest spanning tree* of a given graph G , that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.
*(b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k , the k spanning trees of G with smallest weight.
8. We say that a graph $G = (V, E)$ is *dense* if $E = \Theta(V^2)$. Describe a modification of Jarník's minimum-spanning tree algorithm that runs in $O(V^2)$ time (independent of E) when the input graph is dense, using only simple data structures (and in particular, *without* using a Fibonacci heap).
9. Consider an algorithm that first performs k passes of Boruvka's algorithm, and then runs Jarník's algorithm (*with* a Fibonacci heap) on the resulting contracted graph.
 - (a) What is the running time of this hybrid algorithm, as a function of V , E , and k ?
 - (b) For which value of k is this running time minimized? What is the resulting running time?
10. Describe an algorithm to compute the minimum spanning tree of an n -vertex *planar* graph in $O(n)$ time. [Hint: Contracting an edge in a planar graph yields another planar graph. Any planar graph with n vertices has at most $3n - 6$ edges.]

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

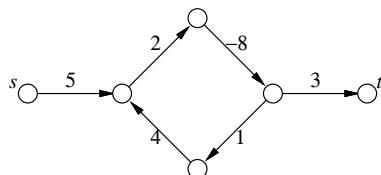
13.1 Introduction

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 we can always find a shorter path by going around the cycle one more time.



There is no shortest path from s to t .

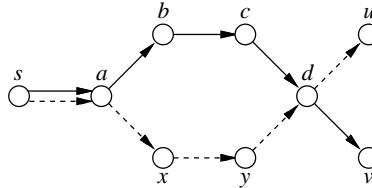
Every algorithm known for solving this problem actually solves (large portions of) the following more general *single source shortest path* or *SSSP* problem: find the shortest path from the source vertex s

¹West 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.

²In 1999 and 2000, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.

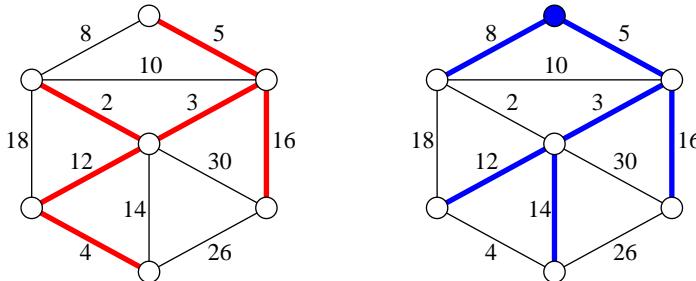
to every other vertex in the graph. In fact, the problem is usually solved by finding a *shortest path tree* rooted at s that contains all the desired shortest paths.

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 $s \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow v$ and $s \rightarrow a \rightarrow x \rightarrow y \rightarrow d \rightarrow u$ are both shortest paths,
then $s \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow u$ 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 described 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 negative-weight edge as a negative cycle of length 2.)

To emphasize the direction, I will consistently use the nonstandard notation $u \rightarrow v$ 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 .

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.

³Specifically, 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.

- $\text{pred}(v)$ 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 $\text{dist}(s) = 0$ and $\text{pred}(s) = \text{NULL}$. For every vertex $v \neq s$, we initially set $\text{dist}(v) = \infty$ and $\text{pred}(v) = \text{NULL}$ to indicate that we do not know of *any* path from s to v .

We call an edge $u \rightarrow v$ *tense* if $\text{dist}(u) + w(u \rightarrow v) < \text{dist}(v)$. If $u \rightarrow v$ is tense, then the tentative shortest path $s \rightsquigarrow v$ is incorrect, since the path $s \rightsquigarrow u \rightarrow v$ is shorter. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

```
RELAX( $u \rightarrow v$ ):
 $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \rightarrow v)$ 
 $\text{pred}(v) \leftarrow u$ 
```

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 $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of the predecessor chain ending at v :

$$s \rightarrow \dots \rightarrow \text{pred}(\text{pred}(v)) \rightarrow \text{pred}(v) \rightarrow v.$$

This is easy to prove by induction on the number of relaxation steps. (Hint, hint.)

2. If the algorithm halts, then $\text{dist}(v) \leq w(s \rightsquigarrow v)$ 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 $\text{dist}(v) = \infty$ by 1 or reduces the sum of the finite shortest path lengths by the difference between two edge weights.

Actually proving the first two claims above is not as straightforward as I make it sound; there are some unfortunate subtleties. It’s *much* easier to prove the following weaker claims, which also imply correctness:

- For every vertex v , $\text{dist}(v)$ is always greater than or equal to the shortest-path distance from s to v . (Induction on the number of relaxation steps.)
- If no edge is tense, then for every vertex v , $\text{dist}(v)$ is the shortest-path distance from s to v . (Induction on the number of edges in the true shortest path; see Bellman-Ford. If the edge weights are non-negative, induction on the rank of the shortest-path distance also works; see Dijkstra!)
- If no edge is tense, then for every vertex v , the path $s \rightarrow \dots \rightarrow \text{pred}(\text{pred}(v)) \rightarrow \text{pred}(v) \rightarrow v$ is a shortest path from s to v . (Induction on the number of edges in the path; why does $\text{pred}(v)$ have its current value?)
- The algorithm halts if there are no negative cycles reachable from s ; see above.

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 $u \rightarrow v$, we put v into the bag. Unlike our generic graph traversal algorithm, the same vertex might be visited many times.

```
INITSSSP( $s$ ):
   $dist(s) \leftarrow 0$ 
   $pred(s) \leftarrow \text{NULL}$ 
  for all vertices  $v \neq s$ 
     $dist(v) \leftarrow \infty$ 
     $pred(v) \leftarrow \text{NULL}$ 
```

```
GENERICSSSP( $s$ ):
  INITSSSP( $s$ )
  put  $s$  in the bag
  while the bag is not empty
    take  $u$  from the bag
    for all edges  $u \rightarrow v$ 
      if  $u \rightarrow v$  is tense
        RELAX( $u \rightarrow v$ )
      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.

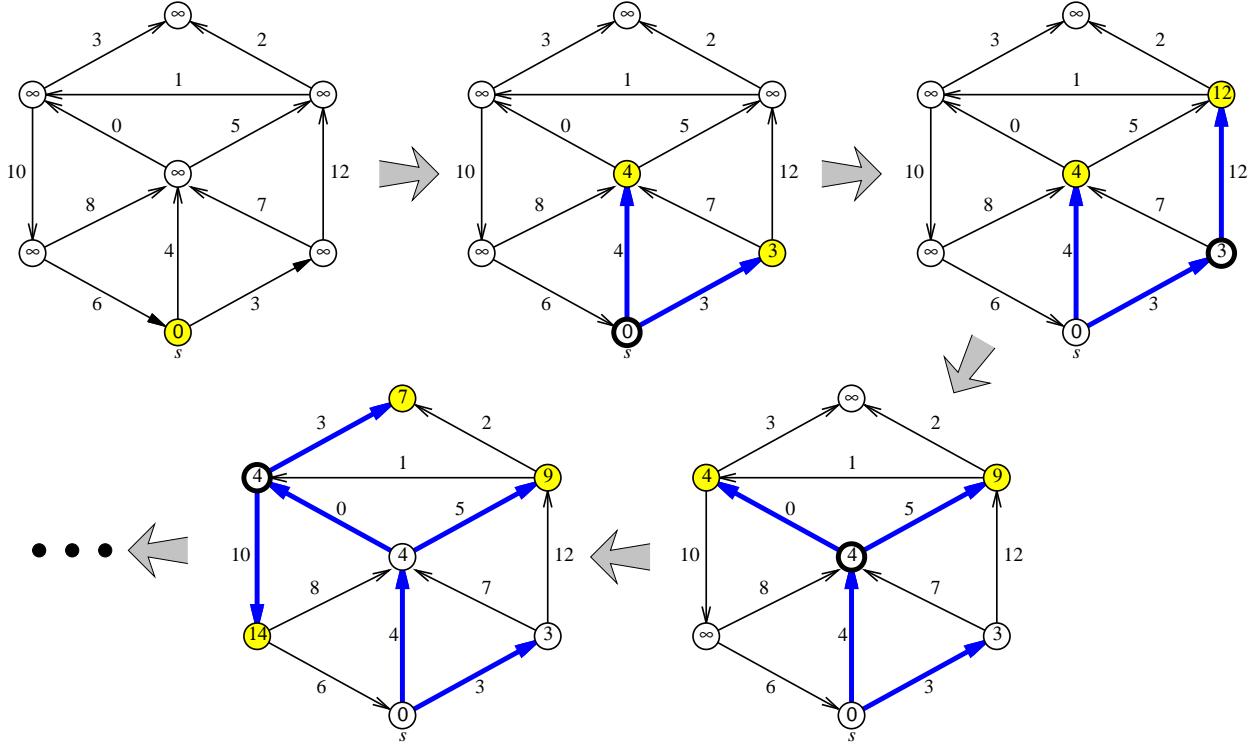
Dijkstra's algorithm, as it is universally known⁵, is particularly well-behaved if the graph has no negative-weight edges. In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s . It follows that each vertex 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 $O(E + V \log V)$; if we use a regular binary heap, the running time is $O(E \log V)$.

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 edges⁶, but the worst-case running time could be exponential. (Proving this unfortunate fact is a good homework problem.)

⁴in the first annual report on a research project performed for the Combat Development Department of the Army Electronic Proving Ground

⁵I 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".

⁶Many textbooks present a version of Dijkstra's algorithm that gives incorrect results for graphs with negative edges.



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.

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 $\text{GUESSESTIMATE}(v, t)$ 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 $\text{dist}(v) + \text{GUESSESTIMATE}(v, t)$.

The function GUESSESTIMATE is called *admissible* if $\text{GUESSESTIMATE}(v, t)$ never overestimates the actual shortest path distance from v to t . If GUESSESTIMATE 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 $\text{GUESSESTIMATE}(v, t)$ 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 many 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 Shimbler'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 Shimbler in 1955, then independently rediscovered by Moore in 1957, by Woodbury and Dantzig in 1957, and by Bellman in 1958. Since Bellman used the idea of relaxing edges, which was first proposed by Ford in 1956, this is usually called the 'Bellman-Ford' algorithm. Shimbler'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.)

Is describing this algorithm with a queue really helpful, or just confusing? After all, it's just a straightforward dynamic programming algorithm. And the structure of the algorithm is very close to the proof of correctness of the generic algorithm (induction on the number of edges in the shortest path).

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 i th 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 V th 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 Shimbels's algorithm is $O(VE)$.

Once we understand how the phases of Shimbels'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.

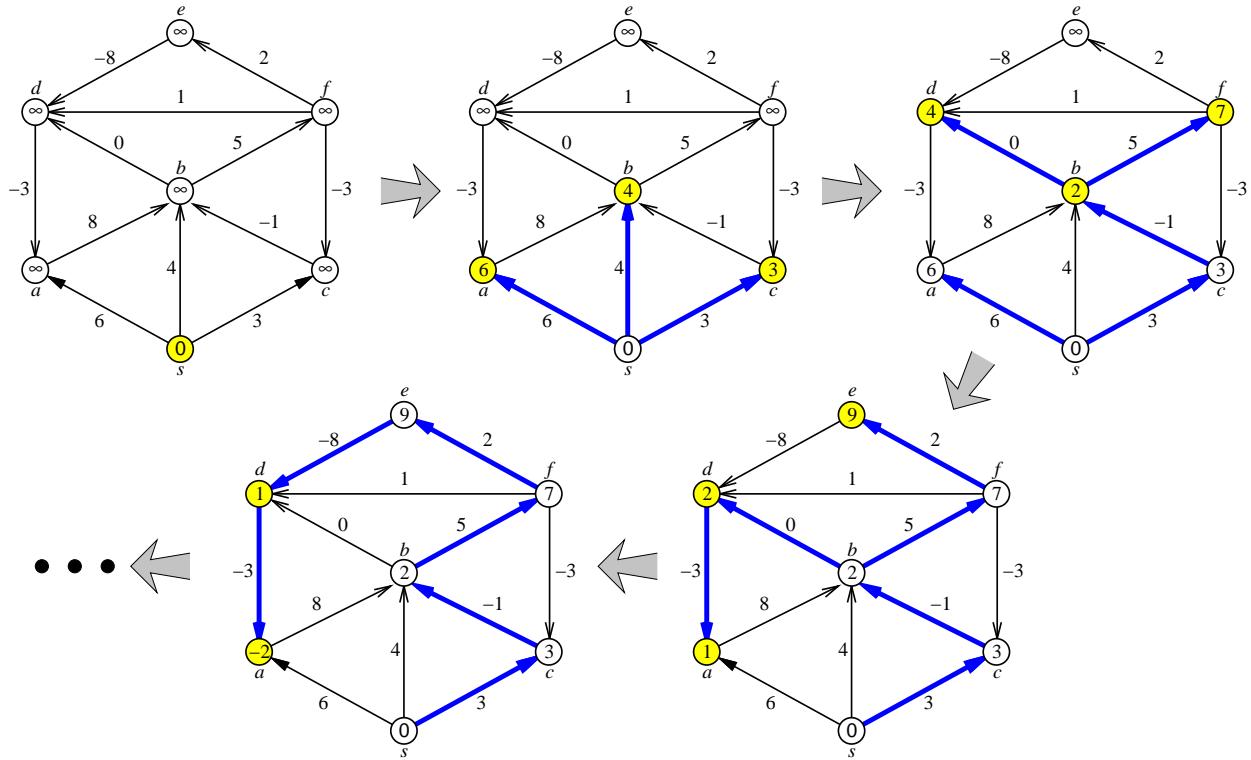
```

SHIMBELSSSP(s)
INITSSSP(s)
repeat  $V$  times:
    for every edge  $u \rightarrow v$ 
        if  $u \rightarrow v$  is tense
            RELAX( $u \rightarrow v$ )
    for every edge  $u \rightarrow v$ 
        if  $u \rightarrow v$  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 .

⁷In fact, this is closer to the description that Shimbels and Bellman used. Bob Tarjan recognized in the early 1980s that Shimbels's algorithm is equivalent to Dijkstra's algorithm with a queue instead of a heap.



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.

13.6 Greedy Relaxation?

Here's another algorithm that fits our generic framework, but which I've never seen analyzed.

Repeatedly relax the tensest edge.

Specifically, let's define the 'tension' of an edge $u \rightarrow v$ as follows:

$$\text{tension}(u \rightarrow v) = \max \{0, \text{dist}(v) - \text{dist}(u) - w(u \rightarrow v)\}$$

(This is defined even when $\text{dist}(v) = \infty$ or $\text{dist}(u) = \infty$, 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 $u \rightarrow v$, then $\text{dist}(v)$ decreases $\text{tension}(u \rightarrow v)$ and $\text{tension}(u \rightarrow v)$ 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 $u \rightarrow v$ 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

⁸I 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 number of relaxation steps in the worst case, so nobody's ever bothered to publish it.

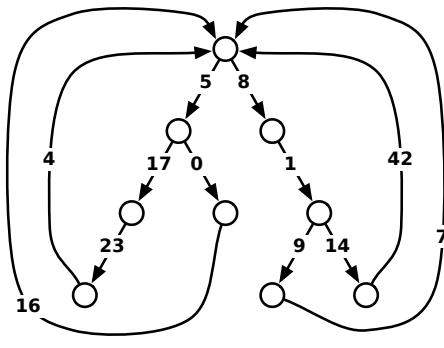
Exercises

1. This question asks you to fill in the remaining proof details for Ford's generic shortest-path algorithm: while at least one edge is tense, relax an arbitrary tense edge. You may assume that the input graph does not contain a negative cycle, and that all shortest paths in the input graph are unique.

- (a) Prove that the generic shortest-path algorithm halts.
- (b) Prove that after *every* call to RELAX, for every vertex v , either $dist(v) = \infty$ or $dist(v)$ is the total weight of some path from s to v .
- (c) Prove that when the generic shortest-path algorithm halts, then for every vertex v , either $dist(v) = \infty$, or $dist(v)$ is the total weight of the predecessor chain ending at v :

$$s \rightarrow \dots \rightarrow pred(pred(v)) \rightarrow pred(v) \rightarrow v.$$

- (d) Prove that when the generic shortest-path algorithm halts, then $dist(v) \leq w(s \rightsquigarrow v)$ for *every* path $s \rightsquigarrow v$.
- 2. Prove the following statement for every integer i and every vertex v : At the end of the i th phase of Shimbels algorithm, $dist(v)$ is less than or equal to the length of the shortest path $s \rightsquigarrow v$ consisting of i or fewer edges.
- *3. Prove that Ford's generic shortest-path algorithm (while the graph contains a tense edge, relax it) can take exponential time in the worst case when implemented with a stack instead of a priority queue (like Dijkstra) or a queue (like Shimbels algorithm). Specifically, for every positive integer n , construct a weighted directed n -vertex graph G_n , such that the stack-based shortest-path algorithm call RELAX $\Omega(2^n)$ times when G_n is the input graph. [Hint: Towers of Hanoi.]
- *4. Prove that Dijkstra's shortest-path algorithm can require exponential time in the worst case when edges are allowed to have negative weight. Specifically, for every positive integer n , construct a weighted directed n -vertex graph G_n , such that Dijkstra's algorithm calls RELAX $\Omega(2^n)$ times when G_n is the input graph. [Hint: This should be easy if you've already solved the previous problem.]
- 5. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.



- (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?
- (b) Describe and analyze a faster algorithm.
6. (a) Describe a modification of Ford's generic shortest-path algorithm that actually returns a negative cycle if any such cycle is reachable from s , or a shortest-path tree if there is no such cycle. You may assume that the unmodified algorithm halts in $O(2^V)$ steps if there is no negative cycle.
- (b) Describe a modification of Shimbel's shortest-path algorithm that actually returns a negative cycle if any such cycle is reachable from s , or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.
7. For any edge e in any graph G , let $G \setminus e$ denote the graph obtained by deleting e from G .
- (a) Suppose we are given a directed graph G in which the shortest path σ from vertex s to vertex t passes through *every* vertex of G . Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G , in $O(E \log V)$ time. Your algorithm should output a set of E shortest-path distances, one for each edge of the input graph. You may assume that all edge weights are non-negative. [Hint: If we delete an edge of the original shortest path, how do the old and new shortest paths overlap?]
- *(b) Let s and t be *arbitrary* vertices in an *arbitrary* directed graph G . Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G , in $O(E \log V)$ time. Again, you may assume that all edge weights are non-negative.
8. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G , and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H , so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert, in $O(E \log V)$ time.
9. Negative edges cause problems in shortest-path algorithms because of the possibility of negative cycles. But what if the input graph has no cycles?
- (a) Describe an efficient algorithm to compute the shortest path between two nodes s and t in a given *directed acyclic graph* with weighted edges. The edge weights could be positive, negative, or zero.
- (b) Describe an efficient algorithm to compute the *longest* path between two nodes s and t in a given directed acyclic graph with weighted edges.
10. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between bus lines at least once.
- Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per

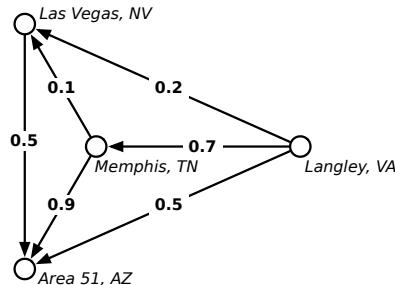
day. Your goal is to minimize your *arrival time*, not the time you actually spend traveling. Assume that the buses run exactly on schedule, that you have an accurate watch, and that you are too tired to walk between bus stops.

11. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

Suppose one of your customers wants to fly from city X to city Y . Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. [Hint: Modify the input data and apply Dijkstra's algorithm.]

12. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t .



For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.)

13. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, “Get out... while... you...”, thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

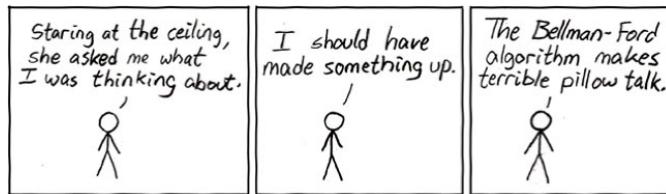
You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those

landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to leave the park immediately.

Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. *Be sure to account for both the vertex probabilities and the edge probabilities!*



— Randall Munroe, xkcd (<http://xkcd.com/c69.html>)

*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)”,
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 $dist(u, v) = w(u \rightarrow v)$ 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 $dist[s, \cdot]$, we invoke either Dijkstra’s or Shimbels algorithm starting at the source vertex s .

OBVIOUSAPSP(V, E, w):

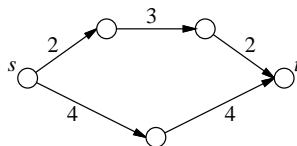
for every vertex s

$dist[s, \cdot] \leftarrow \text{SSSP}(V, E, w, s)$

The running time of this algorithm depends on which single-source shortest path algorithm we use. If we use Shimbels algorithm, the overall running time is $\Theta(V^2E) = O(V^4)$. If all the edge weights are non-negative, we can use Dijkstra's algorithm instead, which decreases the running time to $\Theta(VE + V^2 \log V) = O(V^3)$. 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:

$$w'(u \rightarrow v) = c(u) + w(u \rightarrow v) - c(v)$$

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 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 Shimbels algorithm (which doesn't care if the edge weights are negative), and then sets

$$c(v) = dist(s, v),$$

so the new weight of every edge is

$$w'(u \rightarrow v) = dist(s, u) + w(u \rightarrow v) - dist(s, v).$$

Why are all these new weights non-negative? Because otherwise, Shimbel's algorithm wouldn't be finished! Recall that an edge $u \rightarrow v$ is *tense* if $dist(s, u) + w(u \rightarrow v) < dist(s, v)$, 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  $v \in V$ 
         $w(s \rightarrow v) \leftarrow 0$ 
         $w(v \rightarrow s) \leftarrow \infty$ 
     $dist[s, \cdot] \leftarrow \text{SHIMBEL}(V, E, w, s)$ 
    abort if SHIMBEL found a negative cycle
    for every edge  $(u, v) \in E$ 
         $w'(u \rightarrow v) \leftarrow dist[s, u] + w(u \rightarrow v) - dist[s, v]$ 
    for every vertex  $u \in V$ 
         $dist[u, \cdot] \leftarrow \text{DIJKSTRA}(V, E, w', u)$ 
        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 $\Theta(VE + V^2 \log V)$.

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 $dist(u, v)$:

$$dist(u, v) = \begin{cases} 0 & \text{if } u = v \\ \min_x (dist(u, x) + w(x \rightarrow v)) & \text{otherwise} \end{cases}$$

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 $u \rightarrow v$. **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.

$$dist(u, v, k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min_x (dist(u, x, k - 1) + w(x \rightarrow v)) & \text{otherwise} \end{cases}$$

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 $\Theta(V^4)$ 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 $w(v \rightarrow v) = 0$ 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  $x \in V$ 
                         $dist[u, v, k] \leftarrow \min \{dist[u, v, k], dist[u, x, k - 1] + w(x \rightarrow v)\}$ 
```

The last four lines actually evaluate the recurrence.

In fact, this algorithm is almost exactly the same as running Shimbels algorithm once for every source vertex. The only difference is the innermost loop, which in Shimbels algorithm would read “for all edges $x \rightarrow v$ ”. This simple change improves the running time to $\Theta(V^2E)$, 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 $dist(u, v, k)$. Once again, to simplify things, let's assume $w(v \rightarrow v) = 0$.

$$dist(u, v, k) = \begin{cases} w(u \rightarrow v) & \text{if } k = 1 \\ \min_x (dist(u, x, k/2) + dist(x, v, k/2)) & \text{otherwise} \end{cases}$$

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 $\text{dist}(u, v, 2^{\lceil \lg V \rceil})$ 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 $\Theta(V^3 \log V)$ —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  $v \in V$ 
             $\text{dist}[u, v, 0] \leftarrow w(u \rightarrow v)$ 
    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  $v \in V$ 
                 $\text{dist}[u, v, i] \leftarrow \infty$ 
                for all vertices  $x \in V$ 
                    if  $\text{dist}[u, v, i] > \text{dist}[u, x, i - 1] + \text{dist}[x, v, i - 1]$ 
                         $\text{dist}[u, v, i] \leftarrow \text{dist}[u, x, i - 1] + \text{dist}[x, v, i - 1]$ 
```

14.7 Aside: ‘Funny’ Matrix Multiplication

There is a very close connection (first observed by Shimbrel, 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 V th 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 use 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 Naor,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

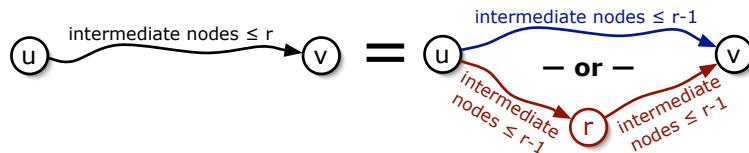
14.8 Floyd and Warshall's Algorithm

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 $\pi(u, v, r)$ 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 $dist(u, v, r)$:

$$dist(u, v, r) = \begin{cases} w(u \rightarrow v) & \text{if } r = 0 \\ \min \{ dist(u, v, r - 1), dist(u, r, r - 1) + dist(r, v, r - 1) \} & \text{otherwise} \end{cases}$$

We need to compute the shortest path distance from u to v with no restrictions, which is just $dist(u, v, V)$.

¹N. 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.

²R. 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.

Once again, we should immediately see that a dynamic programming algorithm that implements this recurrence will run in $\Theta(V^3)$ 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  $V$ 
             $dist[u, v, 0] \leftarrow w(u \rightarrow v)$ 
    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
                     $dist[u, v, r] \leftarrow dist[u, r, r - 1] + dist[r, v, r - 1]$ 
```

Exercises

1. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.
 - (a) How could we delete some node v from this graph, without changing the shortest-path distance between any other pair of nodes? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in H is equal to the shortest-path distance between the same two nodes in G , in $O(V^2)$ time.
 - (b) Now suppose we have already computed all shortest-path distances in G' . Describe an algorithm to compute the shortest-path distances from v to every other node, and from every other node to v , in the original graph G , in $O(V^2)$ time.
 - (c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time.
2. 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 Shimbels algorithm) and aborts; the dynamic programming algorithms just return incorrect results. However, all of these algorithms can be modified to return correct shortest-path distances, even in the presence of 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 , the algorithm should report that $dist[u, v] = -\infty$. If there is no directed path from u to v , the algorithm should return $dist[u, v] = \infty$. Otherwise, $dist[u, v]$ should equal the length of the shortest directed path from u to v .
 - (a) Describe how to modify Johnson's algorithm to return the correct shortest-path distances, even if the graph has negative cycles.
 - (b) Describe how to modify the Floyd-Warshall algorithm to return the correct shortest-path distances, even if the graph has negative cycles.
3. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets V_1, V_2, \dots, V_k ; that is,

every vertex of G belongs to exactly one subset V_i . For each i and j , let $\delta(i, j)$ denote the minimum shortest-path distance between vertices in V_i and vertices in V_j :

$$\delta(i, j) = \min\{dist(u, v) \mid u \in V_i \text{ and } v \in V_j\}.$$

Describe an algorithm to compute $\delta(i, j)$ for all i and j in time $O(V^2 + kE \log E)$.

4. Recall³ that a deterministic finite automaton (DFA) is formally defined as a tuple $M = (\Sigma, Q, q_0, F, \delta)$, where the finite set Σ is the input alphabet, the finite set Q is the set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of final (accepting) states, and $\delta : Q \times \Sigma \rightarrow Q$ is the transition function. Equivalently, a DFA is a directed (multi-)graph with labeled edges, such that each symbol in Σ is the label of exactly one edge leaving any vertex. There is a special ‘start’ vertex q_0 , and a subset of the vertices are marked as ‘accepting states’. Any string in Σ^* describes a unique walk starting at q_0 ; a string in Σ^* is *accepted* by M if this walk ends at a vertex in F .

Stephen Kleene⁴ proved that the language accepted by any DFA is identical to the language described by some regular expression. This problem asks you to develop a variant of the Floyd-Warshall all-pairs shortest path algorithm that computes a regular expression that is equivalent to the language accepted by a given DFA.

Suppose the input DFA M has n states, numbered from 1 to n , where (without loss of generality) the start state is state 1. Let $L(i, j, r)$ denote the set of all words that describe walks in M from state i to state j , where every intermediate state lies in the subset $\{1, 2, \dots, r\}$; thus, the language accepted by the DFA is exactly

$$\bigcup_{q \in F} L(1, q, n).$$

Let $R(i, j, r)$ be a regular expression that describes the language $L(i, j, r)$.

- (a) What is the regular expression $R(i, j, 0)$?
- (b) Write a recurrence for the regular expression $R(i, j, r)$ in terms of regular expressions of the form $R(i', j', r - 1)$.
- (c) Describe a polynomial-time algorithm to compute $R(i, j, n)$ for all states i and j . (Assume that you can concatenate two regular expressions in $O(1)$ time.)

- *5. Let $G = (V, E)$ be an undirected, unweighted, connected, n -vertex graph, represented by the adjacency matrix $A[1..n, 1..n]$. In this problem, we will derive Seidel’s sub-cubic algorithm to compute the $n \times n$ matrix $D[1..n, 1..n]$ of shortest-path distances using fast matrix multiplication. Assume that we have a subroutine `MATRIXMULTIPLY` that multiplies two $n \times n$ matrices in $\Theta(n^\omega)$ time, for some unknown constant $\omega \geq 2$.⁵

- (a) Let G^2 denote the graph with the same vertices as G , where two vertices are connected by a edge if and only if they are connected by a path of length at most 2 in G . Describe an algorithm to compute the adjacency matrix of G^2 using a single call to `MATRIXMULTIPLY` and $O(n^2)$ additional time.

³Automata theory is a prerequisite for the algorithms class at UIUC.

⁴Pronounced ‘clay knee’, not ‘clean’ or ‘clean-ee’ or ‘clay-nuh’ or ‘dimaggio’.

⁵The matrix multiplication algorithm you already know runs in $\Theta(n^3)$ time, but this is not the fastest algorithm known. The current record is $\omega \approx 2.376$, due to Don Coppersmith and Shmuel Winograd. Determining the smallest possible value of ω is a long-standing open problem; many people believe there is an undiscovered $O(n^2)$ -time algorithm for matrix multiplication.

- (b) Suppose we discover that G^2 is a complete graph. Describe an algorithm to compute the matrix D of shortest path distances in $O(n^2)$ additional time.
- (c) Let D^2 denote the (recursively computed) matrix of shortest-path distances in G^2 . Prove that the shortest-path distance from node i to node j is either $2 \cdot D^2[i, j]$ or $2 \cdot D^2[i, j] - 1$.
- (d) Suppose G^2 is not a complete graph. Let $X = D^2 \cdot A$, and let $\deg(i)$ denote the degree of vertex i in the original graph G . Prove that the shortest-path distance from node i to node j is $2 \cdot D^2[i, j]$ if and only if $X[i, j] \geq D^2[i, j] \cdot \deg(i)$.
- (e) Describe an algorithm to compute the matrix of shortest-path distances in G in $O(n^\omega \log n)$ time.

Col. Hogan: 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]

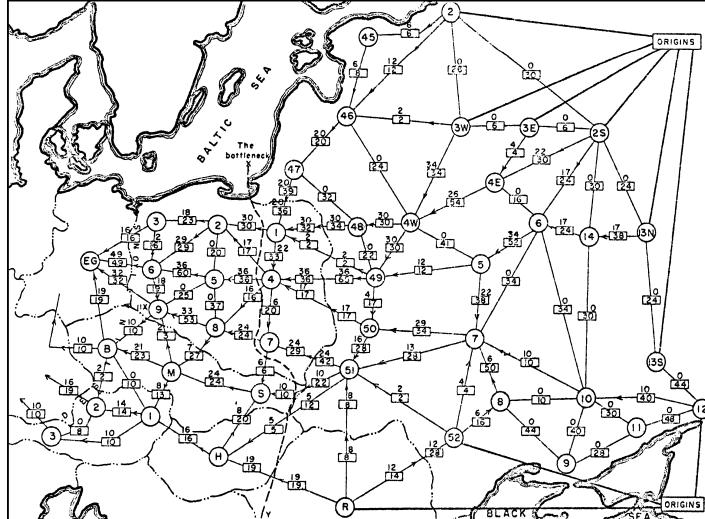
Col. Klink: If you knew which wire it was, why did you ask me?

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. For both problems, the input is a directed graph $G = (V, E)$, along with special vertices s and t 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 . Intuitively, the maximum flow problem asks for the largest amount of material that can be transported from one vertex to another; the minimum cut problem asks for the minimum damage needed to separate two vertices.

¹Both 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 a *flow* if the source and target are clear from context) is a function $f : E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the following *conservation constraint* at every vertex v except possibly s and t :

$$\sum_u f(u \rightarrow v) = \sum_w f(v \rightarrow w).$$

In English, the total flow into v is equal to the total flow out of v . To keep the notation simple, we assume here that $f(u \rightarrow v) = 0$ if there is no edge $u \rightarrow v$ in the graph.

The *value* of the flow f , denoted $|f|$, is defined as the excess flow out of the source vertex s :

$$|f| := \sum_w f(s \rightarrow w) - \sum_u f(u \rightarrow s)$$

It's not hard to show that the value $|f|$ is also equal to the excess flow *into* the target vertex t . First we observe that

$$\sum_v \left(\sum_w f(v \rightarrow w) - \sum_u f(u \rightarrow v) \right) = \sum_v \sum_w f(v \rightarrow w) - \sum_v \sum_u f(u \rightarrow v) = 0$$

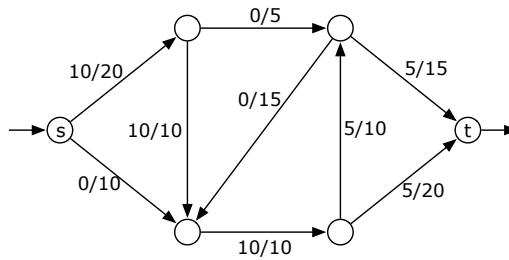
because both summations count the total flow across all edges. On the other hand, the conservation constraint implies that

$$\begin{aligned} & \sum_v \left(\sum_w f(v \rightarrow w) - \sum_u f(u \rightarrow v) \right) \\ &= \left(\sum_w f(t \rightarrow w) - \sum_u f(u \rightarrow s) \right) + \left(\sum_w f(t \rightarrow w) - \sum_u f(u \rightarrow t) \right) \\ &= |f| + \left(\sum_w f(t \rightarrow w) - \sum_u f(u \rightarrow t) \right). \end{aligned}$$

It follows that

$$|f| = \sum_u f(u \rightarrow t) - \sum_w f(t \rightarrow w).$$

Now suppose we have another function $c : E \rightarrow \mathbb{R}_{\geq 0}$ that assigns a non-negative *capacity* $c(e)$ to each edge e . We say that a flow f is *feasible* (with respect to c) if $f(e) \leq c(e)$ for every edge e . Most of the time we will consider only flows that are feasible with respect 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 a feasible (s, t) -flow in a given directed graph, with 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.

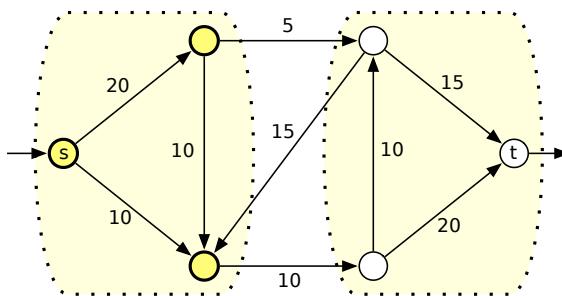
15.2 Cuts

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 $S \cap T = \emptyset$ —where $s \in S$ and $t \in T$.

If we have a capacity function $c: E \rightarrow \mathbb{R}_{\geq 0}$, the **capacity** of a cut is the sum of the capacities of the edges that start in S and end in T :

$$\|S, T\| = \sum_{v \in S} \sum_{w \in T} c(v \rightarrow w).$$

(Again, if $v \rightarrow w$ is not an edge in the graph, we assume $c(v \rightarrow w) = 0$.) 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 capacity is as large as possible.



An (s, t) -cut with capacity 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 feasible (s, t) -flow is at most the capacity of any (s, t) -cut**. Choose your favorite flow f and your favorite cut (S, T) , and then follow the bouncing inequalities:

$$\begin{aligned}
 |f| &= \sum_w f(s \rightarrow w) - \sum_u f(u \rightarrow s) && \text{by definition} \\
 &= \sum_{v \in S} \left(\sum_w f(v \rightarrow w) - \sum_u f(u \rightarrow v) \right) && \text{by the conservation constraint} \\
 &= \sum_{v \in S} \left(\sum_{w \in T} f(v \rightarrow w) - \sum_{u \in T} f(u \rightarrow v) \right) && \text{removing duplicate edges} \\
 &\leq \sum_{v \in S} \sum_{w \in T} f(v \rightarrow w) && \text{since } f(u \rightarrow v) \geq 0 \\
 &\leq \sum_{v \in S} \sum_{w \in T} c(v \rightarrow w) && \text{since } f(u \rightarrow v) \leq c(v \rightarrow w) \\
 &= \|S, T\| && \text{by definition}
 \end{aligned}$$

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 flow, and (S, T) must be a minimum cut.

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 capacity 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 $c: E \rightarrow \mathbb{R}_{\geq 0}$. The proof will be easier if we assume that the capacity function is **reduced**: For any vertices u and v , either $c(u \rightarrow v) = 0$ or $c(v \rightarrow u) = 0$, or equivalently, if an edge appears in G , then its reversal does not. This assumption is easy to enforce. Whenever an edge $u \rightarrow v$ and its reversal $v \rightarrow u$ are both in the graph, replace the edge $u \rightarrow v$ with a path $u \rightarrow x \rightarrow v$ of length two, where x is a new vertex and $c(u \rightarrow x) = c(x \rightarrow v) = c(u \rightarrow v)$. The modified graph has the same maximum flow value and minimum cut capacity as the original graph.

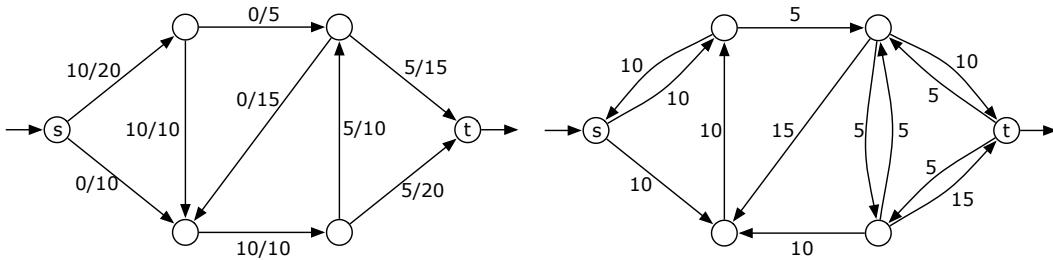


Enforcing the one-direction assumption.

Let f be a feasible flow. We define a new capacity function $c_f: V \times V \rightarrow \mathbb{R}$, called the **residual capacity**, as follows:

$$c_f(u \rightarrow v) = \begin{cases} c(u \rightarrow v) - f(u \rightarrow v) & \text{if } u \rightarrow v \in E \\ f(v \rightarrow u) & \text{if } v \rightarrow u \in E \\ 0 & \text{otherwise} \end{cases}$$

Since $f \geq 0$ and $f \leq c$, the residual capacities are always non-negative. It is possible to have $c_f(u \rightarrow v) > 0$ even if $u \rightarrow v$ 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 $c_f(u \rightarrow v) > 0$ and $c_f(v \rightarrow u) > 0$.

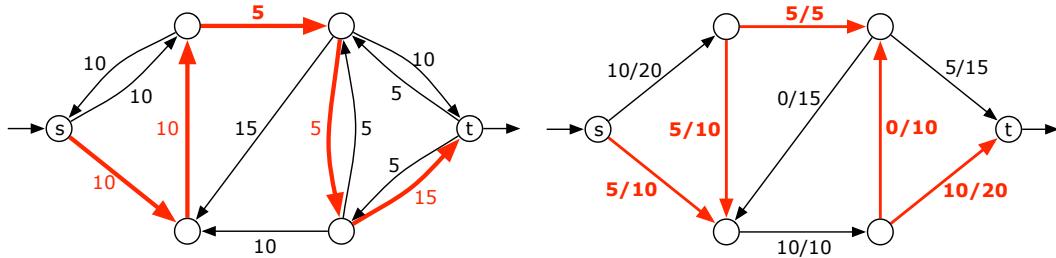


A flow f in a weighted graph G and the corresponding residual graph G_f .

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

$$c_f(u \rightarrow v) = (c(u \rightarrow v) - f(u \rightarrow v)) + f(v \rightarrow u) = 0,$$

which implies that $c(u \rightarrow v) - f(u \rightarrow v) = 0$ and $f(v \rightarrow u) = 0$. 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 $s = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_r = t$ in G_f . We refer to $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_r$ as an **augmenting path**. Let $F = \min_i c_f(v_i \rightarrow v_{i+1})$ denote the maximum amount of flow that we can push through the augmenting path in G_f . We define a new flow function $f': E \rightarrow \mathbb{R}$ as follows:

$$f'(u \rightarrow v) = \begin{cases} f(u \rightarrow v) + F & \text{if } u \rightarrow v \text{ is in the augmenting path} \\ f(u \rightarrow v) - F & \text{if } v \rightarrow u \text{ is in the augmenting path} \\ f(u \rightarrow v) & \text{otherwise} \end{cases}$$

To prove that the flow f' is feasible with respect to the original capacities c , we need to verify that $f' \geq 0$ and $f' \leq c$. Consider an edge $u \rightarrow v$ in G . If $u \rightarrow v$ is in the augmenting path, then $f'(u \rightarrow v) > f(u \rightarrow v) \geq 0$ and

$$\begin{aligned} f'(u \rightarrow v) &= f(u \rightarrow v) + F && \text{by definition of } f' \\ &\leq f(u \rightarrow v) + c_f(u \rightarrow v) && \text{by definition of } F \\ &= f(u \rightarrow v) + c(u \rightarrow v) - f(u \rightarrow v) && \text{by definition of } c_f \\ &= c(u \rightarrow v) && \text{Duh.} \end{aligned}$$

On the other hand, if the reversal $v \rightarrow u$ is in the augmenting path, then $f'(u \rightarrow v) < f(u \rightarrow v) \leq c(u \rightarrow v)$, which implies that

$$\begin{aligned} f'(u \rightarrow v) &= f(u \rightarrow v) - F && \text{by definition of } f' \\ &\geq f(u \rightarrow v) - c_f(v \rightarrow u) && \text{by definition of } F \\ &= f(u \rightarrow v) - f(u \rightarrow v) && \text{by definition of } c_f \\ &= 0 && \text{Duh.} \end{aligned}$$

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!

Exercises

- Let (S, T) and (S', T') be minimum (s, t) -cuts in some flow network G . Prove that $(S \cap S', T \cup T')$ and $(S \cup S', T \cap T')$ are also minimum (s, t) -cuts in G .
- Suppose (S, T) is a minimum (s, t) -cut in some flow network. Prove that (S, T) is also a minimum (x, y) -cut for all vertices $x \in S$ and $y \in T$.

3. Cuts are sometimes defined as subsets of the edges of the graph, instead of as partitions of its vertices. In this problem, you will prove that these two definitions are *almost* equivalent.

We say that a subset X of (directed) edges *separates* s and t if every directed path from s to t contains at least one (directed) edge in X . For any subset S of vertices, let δS denote the set of directed edges leaving S ; that is, $\delta S := \{u \rightarrow v \mid u \in S, v \notin S\}$.

- (a) Prove that if (S, T) is an (s, t) -cut, then δS separates s and t .
 - (b) Let X be an arbitrary subset of edges that separates s and t . Prove that there is an (s, t) -cut (S, T) such that $\delta S \subseteq X$.
 - (c) Let X be a *minimal* subset of edges that separates s and t . Prove that there is an (s, t) -cut (S, T) such that $\delta S = X$.
4. A flow f is *acyclic* if the subgraph of directed edges with positive flow contains no directed cycles.
- (a) Prove that for any flow f , there is an acyclic flow with the same value as f . (In particular, this implies that some maximum flow is acyclic.)
 - (b) A *path flow* assigns positive values only to the edges of one simple directed path from s to t . Prove that every acyclic flow can be written as the sum of a finite number of path flows.
 - (c) Describe a flow in a directed graph that *cannot* be written as the sum of path flows.
 - (d) A *cycle flow* assigns positive values only to the edges of one simple directed cycle. Prove that every flow can be written as the sum of a finite number of path flows and cycle flows.
 - (e) Prove that every flow with value 0 can be written as the sum of a finite number of cycle flows. (Zero-value flows are also called *circulations*.)
5. Suppose instead of capacities, we consider networks where each edge $u \rightarrow v$ has a non-negative *demand* $d(u \rightarrow v)$. Now an (s, t) -flow f is *feasible* if and only if $f(u \rightarrow v) \geq d(u \rightarrow v)$ for every edge $u \rightarrow v$. (Feasible flow values can now be arbitrarily large.) A natural problem in this setting is to find a feasible (s, t) -flow of *minimum* value.
- (a) Describe an efficient algorithm to compute a feasible (s, t) -flow, given the graph, the demand function, and the vertices s and t as input. [Hint: Find a flow that is non-zero everywhere, and then scale it up to make it feasible.]
 - (b) Suppose you have access to a subroutine `MAXFLOW` that computes *maximum* flows in networks with edge capacities. Describe an efficient algorithm to compute a *minimum* flow in a given network with edge demands; your algorithm should call `MAXFLOW` exactly once.
 - (c) State and prove an analogue of the max-flow min-cut theorem for this setting. (Do minimum flows correspond to maximum cuts?)

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)

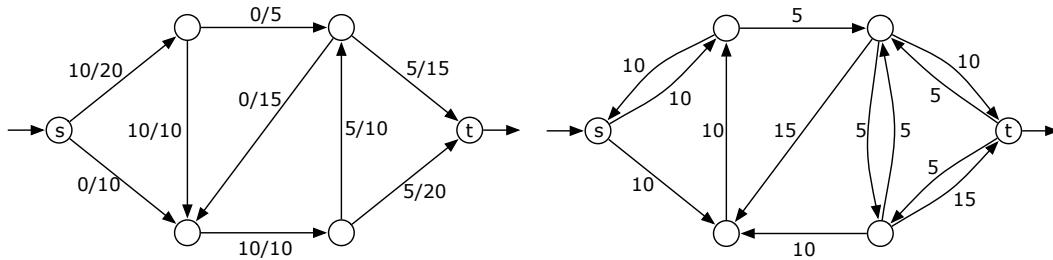
16 Max-Flow Algorithms

16.1 Recap

Fix a directed graph $G = (V, E)$ that does not contain both an edge $u \rightarrow v$ and its reversal $v \rightarrow u$, and fix a capacity function $c : E \rightarrow \mathbb{R}_{\geq 0}$. For any flow function $f : E \rightarrow \mathbb{R}_{\geq 0}$, the *residual capacity* is defined as

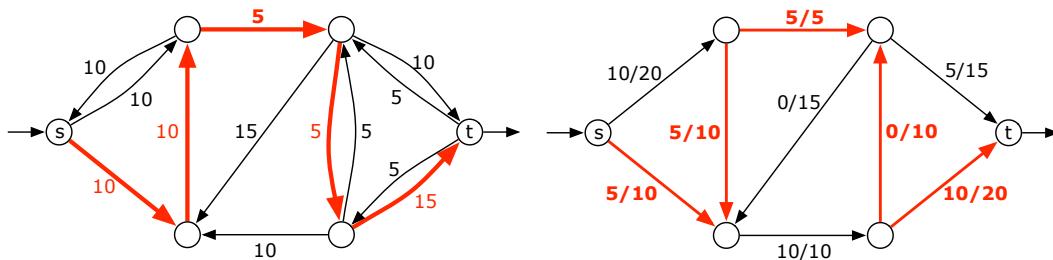
$$c_f(u \rightarrow v) = \begin{cases} c(u \rightarrow v) - f(u \rightarrow v) & \text{if } u \rightarrow v \in E \\ f(v \rightarrow u) & \text{if } v \rightarrow u \in E \\ 0 & \text{otherwise} \end{cases}$$

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 capacity 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 capacity 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' .

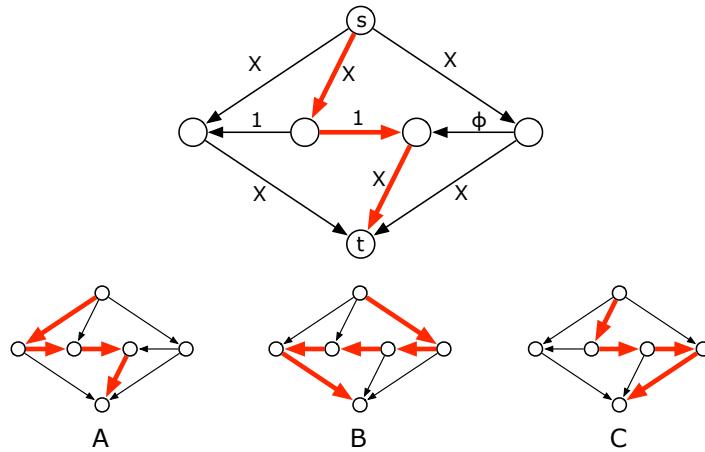
16.2 Ford-Fulkerson

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 $O(E|f^*|)$ 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 $\phi^{k-1}, 0, \phi^k$ 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 hardware¹, 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:

Choose the augmenting path with largest bottleneck value.

It's 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 in G 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 capacity of the cut (S, T) is at most $c(e) \cdot |E|$. On the other hand, the maxflow-mincut theorem implies that $\|S, T\| \geq |f|$. We conclude that $c(e) \geq |f|/|E|$.

The preceding argument implies that 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 $O(E^2 \log E \log|f^*|)$ time.

¹...or perhaps the laws of physics. Yeah, right. Whatever. Like reality actually matters in this class.

16.4 Dinitz/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 Dinitz (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 $\text{level}_i(v)$ 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. $\text{level}_{i+1}(v) \geq \text{level}_i(v)$ for all vertices v and integers i .

Proof: The claim is trivial for $v = s$, since $\text{level}_i(s) = 0$ for all i . Choose an arbitrary vertex $v \neq s$, and let $s \rightarrow \dots \rightarrow u \rightarrow v$ be a shortest path from s to v in G_{i+1} . (If there is no such path, then $\text{level}_{i+1}(v) = \infty$, and we’re done.) Because this is a shortest path, we have $\text{level}_{i+1}(v) = \text{level}_{i+1}(u) + 1$, and the inductive hypothesis implies that $\text{level}_{i+1}(u) \geq \text{level}_i(u)$.

We now have two cases to consider. If $u \rightarrow v$ is an edge in G_i , then $\text{level}_i(v) \leq \text{level}_i(u) + 1$, because the levels are defined by breadth-first traversal.

On the other hand, if $u \rightarrow v$ is not an edge in G_i , then $v \rightarrow u$ must be an edge in the i th augmenting path. Thus, $v \rightarrow u$ must lie on the shortest path from s to t in G_i , which implies that $\text{level}_i(v) = \text{level}_i(u) - 1 \leq \text{level}_i(u) + 1$.

In both cases, we have $\text{level}_{i+1}(v) = \text{level}_{i+1}(u) + 1 \geq \text{level}_i(u) + 1 \geq \text{level}_i(v)$. □

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 Dinitz/Edmonds-Karp algorithm, any edge $u \rightarrow v$ disappears from the residual graph G_f at most $V/2$ times.

Proof: Suppose $u \rightarrow v$ is in two residual graphs G_i and G_{j+1} , but not in any of the intermediate residual graphs G_{i+1}, \dots, G_j , for some $i < j$. Then $u \rightarrow v$ must be in the i th augmenting path, so $\text{level}_i(v) = \text{level}_i(u) + 1$, and $v \rightarrow u$ must be on the j th augmenting path, so $\text{level}_j(v) = \text{level}_j(u) - 1$. By the previous lemma, we have

$$\text{level}_j(u) = \text{level}_j(v) + 1 \geq \text{level}_i(v) + 1 = \text{level}_i(u) + 2.$$

²To 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; Dinitz also presumably discovered his algorithm a few years before its publication. (In Soviet Union, result publish you.) On the gripping hand, Dinitz’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 Dinitz’s algorithm.

In other words, the distance from s to u increased by at least 2 between the disappearance and reappearance of $u \rightarrow v$. Since every level is either less than V or infinite, the number of disappearances is at most $V/2$. \square

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, Dinitz' algorithm runs in $O(VE^2)$ time overall.

Exercises

1. A new assistant professor, teaching maximum flows for the first time, suggests the following greedy modification to the generic Ford-Fulkerson augmenting path algorithm. Instead of maintaining a residual graph, just reduce the capacity of edges along the augmenting path! In particular, whenever we saturate an edge, just remove it from the graph.

```
GREEDYFLOW( $G, c, s, t$ ):
    for every edge  $e$  in  $G$ 
         $f(e) \leftarrow 0$ 
    while there is a path from  $s$  to  $t$ 
         $\pi \leftarrow$  an arbitrary path from  $s$  to  $t$ 
         $F \leftarrow$  minimum capacity of any edge in  $\pi$ 
        for every edge  $e$  in  $\pi$ 
             $f(e) \leftarrow f(e) + F$ 
            if  $c(e) = F$ 
                remove  $e$  from  $G$ 
            else
                 $c(e) \leftarrow c(e) - F$ 
    return  $f$ 
```

- (a) Show that this algorithm does *not* always compute a maximum flow.
- (b) Prove that for any flow network, if the Greedy Path Fairy tells you precisely which path π to use at each iteration, then GREEDYFLOW does compute a maximum flow. (Sadly, the Greedy Path Fairy does not actually exist.)
2. Describe and analyze an algorithm to find the maximum-bottleneck path from s to t in a flow network G in $O(E \log V)$ time.
- ★3. Describe a directed graph with irrational edge capacities, such that the Edmonds-Karp 'fat pipe' heuristic does not halt.
4. Describe an efficient algorithm to check whether a given flow network contains a *unique* maximum flow.
5. For any flow network G and any vertices u and v , let $bottleneck_G(u, v)$ denote the maximum, over all paths π in G from u to v , of the minimum-capacity edge along π . Describe an algorithm

to construct a spanning tree T of G such that $bottleneck_T(u, v) = bottleneck_G(u, v)$. (Edges in T inherit their capacities from G .)

One way to think about this problem is to imagine the vertices of the graph as islands, and the edges as bridges. Each bridge has a maximum weight it can support. If a truck is carrying stuff from u to v , how much can the truck carry? We don't care what route the truck takes; the point is that the smallest-weight edge on the route will determine the load.

6. We can speed up the Edmonds-Karp ‘fat pipe’ heuristic, at least for integer capacities, by relaxing our requirements for the next augmenting path. Instead of finding the augmenting path with maximum bottleneck capacity, we find a path whose bottleneck capacity is at least half of maximum, using the following *capacity scaling* algorithm.

The algorithm maintains a bottleneck threshold Δ ; initially, Δ is the maximum capacity among all edges in the graph. In each *phase*, the algorithm augments along paths from s to t in which every edge has residual capacity at least Δ . When there is no such path, the phase ends, we set $\Delta \leftarrow \Delta/2$, and the next phase begins.

- (a) How many phases will the algorithm execute in the worst case, if the edge capacities are integers?
- (b) Let f be the flow at the end of a phase for a particular value of Δ . Let S be the nodes that are reachable from s in the residual graph G_f using only edges with residual capacity at least Δ , and let $T = V \setminus S$. Prove that the capacity (with respect to G 's original edge capacities) of the cut (S, T) is at most $|f| + E \cdot \Delta$.
- (c) Prove that in each phase of the scaling algorithm, there are at most $2E$ augmentations.
- (d) What is the overall running time of the scaling algorithm, assuming all the edge capacities are integers?

For a long time it puzzled me how something so expensive, so leading edge, could be so useless, and then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

— Bill Bryson, *Notes from a Big Country* (1999)

17 Applications of Maximum Flow

17.1 Edge-Disjoint Paths

One of the easiest applications of maximum flows is computing the maximum number of edge-disjoint paths between two specified vertices s and t in a directed graph G 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. Since any vertex of G lies on at most two saturated edges (one in and one out, or none at all), the subgraph S of saturated edges is the union of several edge-disjoint paths and cycles. Moreover, the number of paths is exactly 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.

Conversely, we can transform any collection of k 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 exactly k . It follows that the maxflow algorithm actually computes the largest possible set of edge-disjoint paths. The overall running time is $O(VE)$, just like for maximum bipartite matchings.

The same algorithm can also be used to find edge-disjoint paths in *undirected* graphs. We simply replace every undirected edge in G with a pair of directed edges, each with unit capacity, and compute a maximum flow from s to t in the resulting directed graph G' using the Ford-Fulkerson algorithm. For any edge uv in G , if our max flow saturates both directed edges $u \rightarrow v$ and $v \rightarrow u$ in G' , we can remove *both* edges from the flow without changing its value. Thus, without loss of generality, the maximum flow assigns a direction to every saturated edge, and we can extract the edge-disjoint paths by searching the graph of directed saturated edges.

17.2 Vertex Capacities and Vertex-Disjoint Paths

Suppose we have capacities on the vertices as well as the edges. Here, in addition to our other constraints, we require that for any vertex v other than s and t , the total flow into v (and therefore the total flow out of v) is at most some non-negative value $c(v)$. How can we compute a maximum flow with these new constraints?

One possibility is to modify our existing algorithms to take these vertex capacities into account. Given a flow f , we can define the *residual capacity* of a vertex v to be its original capacity minus the total flow into v :

$$c_f(v) = c(v) - \sum_u f(u \rightarrow v).$$

Since we cannot send any more flow into a vertex with residual capacity 0 we remove from the residual graph G_f every edge $u \rightarrow v$ that appears in G whose head vertex v is saturated. Otherwise, the augmenting-path algorithm is unchanged.

But an even simpler method is to transform the input into a traditional flow network, with only edge capacities. Specifically, we replace every vertex v with two vertices v_{in} and v_{out} , connected by

an edge $v_{\text{in}} \rightarrow v_{\text{out}}$ with capacity $c(v)$, and then replace every directed edge $u \rightarrow v$ with the edge $u_{\text{out}} \rightarrow v_{\text{in}}$ (keeping the same capacity). Finally, we compute the maximum flow from s_{out} to t_{in} in this modified flow network.

It is now easy to compute the maximum number of vertex-disjoint paths from s to t in any directed graph. Simply give every vertex capacity 1, and compute a maximum flow!

17.3 Maximum Matchings in Bipartite Graphs

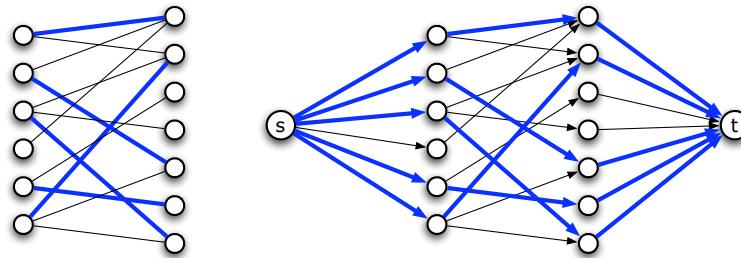
Another natural application of maximum flows is finding large *matchings* in bipartite graphs. A matching is a subgraph in which every vertex has degree at most one, or equivalently, a collection of edges such that no two share a vertex. The problem is to find the matching with the maximum number of edges 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 $U \cup W$, 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 uw in M , push one unit of flow along the path $s \rightarrow u \rightarrow w \rightarrow t$. 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 **$O(VE)$ time**.



A maximum matching in a bipartite graph G , and the corresponding maximum flow in G' .

17.4 Binary Assignment Problems

Maximum-cardinality matchings are a special case of a general family of so-called *assignment* problems.¹ An unweighted *binary* assignment problem involves two disjoint finite sets X and Y , which typically represent two different kinds of resources, such as web pages and servers, jobs and machines, rows and

¹Most authors refer to finding a maximum-weight matching in a bipartite graph as *the* assignment problem.

columns of a matrix, hospitals and interns, or customers and pints of ice cream. Our task is to choose the largest possible collection of pairs (x, y) as possible, where $x \in X$ and $y \in Y$, subject to several constraints of the following form:

- Each element $x \in X$ can appear in at most $c(x)$ pairs.
- Each element $y \in Y$ can appear in at most $c(y)$ pairs.
- Each pair $(x, y) \in X \times Y$ can appear in the output at most $c(x, y)$ times.

Each upper bound $c(x)$, $c(y)$, and $c(x, y)$ is either a (typically small) non-negative integer or ∞ . Intuitively, we create each pair in our output by *assigning* an element of X to an element of y .

The maximum-matching problem is a special case, where $c(z) = 1$ for all $z \in X \cup Y$, and each $c(x, y)$ is either 0 or 1, depending on whether the pair xy defines an edge in the underlying bipartite graph.

Here is a slightly more interesting example. A nearby school, famous for its onerous administrative hurdles, decides to organize a dance. Every pair of students (one boy, one girl) who wants to dance must register in advance. School regulations limit each boy-girl pair to at most three dances together, and limits each student to at most ten dances overall. How can we maximize the number of dances? This is a binary assignment problem for the set X of girls and the set Y of boys. For each girl x and boy y , we have $c(x) = 10$, $c(y) = 10$, and either $c(x, y) = 3$ (if x and y registered to dance) or $c(x, y) = 0$ (if they didn't).

This binary assignment problem can be reduced to a standard maximum flow problem as follows. We construct a flow network $G = (V, E)$ with vertices $X \cup Y \cup \{s, t\}$ and the following edges:

- an edge $s \rightarrow x$ with capacity $c(x)$ for each $x \in X$,
- an edge $y \rightarrow t$ with capacity $c(y)$ for each $y \in Y$.
- an edge $x \rightarrow y$ with capacity $c(x, y)$ for each $x \in X$ and $y \in Y$, and

Because all the edges have integer capacities, the Ford-Fulkerson algorithm constructs an integer maximum flow f^* . This flow can be decomposed into the sum of $|f^*|$ paths of the form $s \rightarrow x \rightarrow y \rightarrow t$ for some $x \in X$ and $y \in Y$. For each such path, we report the pair (x, y) . (Equivalently, the pair (x, y) appears in our output collection $f(x \rightarrow y)$ times.) It is easy to verify (hint, hint) that this collection of pairs satisfies all the necessary constraints. Conversely, any legal collection of r pairs can be transformed into a feasible integer flow with value r in G . Thus, the largest legal collection of pairs corresponds to a maximum flow in G . So our algorithm is correct.

17.5 Baseball Elimination

Every year millions of baseball fans eagerly watch their favorite team, hoping they will win a spot in the playoffs, and ultimately the World Series. Sadly, most teams are “mathematically eliminated” days or even weeks before the regular season ends. Often, it is easy to spot when a team is eliminated—they can’t win enough games to catch up to the current leader in their division. But sometimes the situation is more subtle.

For example, here are the actual standings from the American League East on August 30, 1996.

Team	Won-Lost	Left	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Lions	49–86	27	3	4	0	0	

Detroit is clearly behind, but some die-hard Lions fans may hold out hope that their team can still win. After all, if Detroit wins all 27 of their remaining games, they will end the season with 76 wins, more than any other team has now. So as long as every other team loses every game... but that's not possible, because some of those other teams still have to play each other. Here is one complete argument:²

By winning all of their remaining games, Detroit can finish the season with a record of 76 and 86. If the Yankees win just 2 more games, then they will finish the season with a 77 and 85 record which would put them ahead of Detroit. So, let's suppose the Tigers go undefeated for the rest of the season and the Yankees fail to win another game.

The problem with this scenario is that New York still has 8 games left with Boston. If the Red Sox win all of these games, they will end the season with at least 77 wins putting them ahead of the Tigers. Thus, the only way for Detroit to even have a chance of finishing in first place, is for New York to win exactly one of the 8 games with Boston and lose all their other games. Meanwhile, the Sox must loss all the games they play agains teams other than New York. This puts them in a 3-way tie for first place....

Now let's look at what happens to the Orioles and Blue Jays in our scenario. Baltimore has 2 games left with Boston and 3 with New York. So, if everything happens as described above, the Orioles will finish with at least 76 wins. So, Detroit can catch Baltimore only if the Orioles lose all their games to teams other than New York and Boston. In particular, this means that Baltimore must lose all 7 of its remaining games with Toronto. The Blue Jays also have 7 games left with the Yankees and we have already seen that for Detroit to finish in first place, Toronto must will all of these games. But if that happens, the Blue Jays will win at least 14 more games giving them at final record of 77 and 85 or better which means they will finish ahead of the Tigers. So, no matter what happens from this point in the season on, Detroit can not finish in first place in the American League East.

There has to be a better way to figure this out!

Here is a more abstract formulation of the problem. Our input consists of two arrays $W[1..n]$ and $G[1..n, 1..n]$, where $W[i]$ is the number of games team i has already won, and $G[i, j]$ is the number of upcoming games between teams i and j . We want to determine whether team n can end the season with the most wins (possibly tied with other teams).³

We model this question as an assignment problem: We want to **assign** a winner to each game, so that team n comes in first place. We have an assignment problem! Let $R[i] = \sum_j G[i, j]$ denote the number of remaining games for team i . We will assume that team n wins all $R[n]$ of its remaining games. Then team n can come in first place if and only if every other team i wins at most $W[n] + R[n] - W[i]$ of its $R[i]$ remaining games.

Since we want to **assign** winning teams to games, we start by building a bipartite graph, whose nodes represent the games and the teams. We have $\binom{n}{2}$ game nodes $g_{i,j}$, one for each pair $1 \leq i < j < n$, and $n - 1$ team nodes t_i , one for each $1 \leq i < n$. For each pair i, j , we add edges $g_{i,j} \rightarrow t_i$ and $g_{i,j} \rightarrow t_j$ with infinite capacity. We add a source vertex s and edges $s \rightarrow g_{i,j}$ with capacity $G[i, j]$ for each pair i, j . Finally, we add a target node t and edges $t_i \rightarrow t$ with capacity $W[n] - W[i] + R[n]$ for each team i .

Theorem: Team n can end the season in first place if and only if there is a feasible flow in this graph that saturates every edge leaving s .

Proof: Suppose it is possible for team n to end the season in first place. Then every team $i < n$ wins at most $W[n] + R[n] - W[i]$ of the remaining games. For each game between team i and team j that team i wins, add one unit of flow along the path $s \rightarrow g_{i,j} \rightarrow t_i \rightarrow t$. Because there are exactly $G[i, j]$ games between teams i and j , every edge leaving s is saturated. Because each team i wins at most $W[n] + R[n] - W[i]$ games, the resulting flow is feasible.

Conversely, Let f be a feasible flow that saturates every edge out of s . Suppose team i wins exactly $f(g_{i,j} \rightarrow t_i)$ games against team j , for all i and j . Then teams i and j play $f(g_{i,j} \rightarrow t_i) + f(g_{i,j} \rightarrow t_j) =$

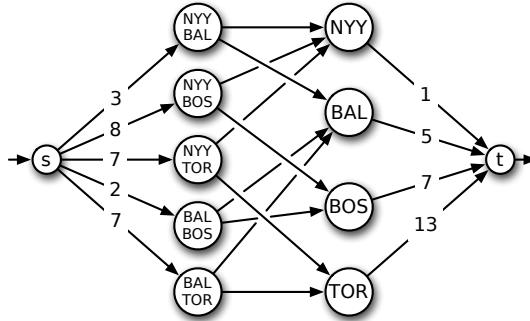
²Both the example and this argument are taken from <http://riot.ier.berkeley.edu/~baseball/detroit.html>.

³We assume here that no games end in a tie (always true for Major League Baseball), and that every game is actually played (not always true).

$f(s \rightarrow g_{i,j}) = G[i, j]$ games, so every upcoming game is played. Moreover, each team i wins a total of $\sum_j f(g_{i,j} \rightarrow t_i) = f(t_i \rightarrow t) \leq W[n] + R[n] - W[i]$ upcoming games, and therefore at most $W[n] + R[n]$ games overall. Thus, if team n win all their upcoming games, they end the season in first place. \square

So, to decide whether our favorite team can win, we construct the flow network, compute a maximum flow, and report whether than maximum flow saturates the edges leaving s . The flow network has $O(n^2)$ vertices and $O(n^2)$ edges, and it can be constructed in $O(n^2)$ time. Using Dinitz's algorithm, we can compute the maximum flow in $O(VE^2) = O(n^6)$ time.

The graph derived from the 1996 American League East standings is shown below. The total capacity of the edges leaving s is 27 (there are 27 remaining games), but the total capacity of the edges entering t is only 26. So the maximum flow has value at most 26, which means that Detroit is mathematically eliminated.



The flow graph for the 1996 American League East standings. Unlabeled edges have infinite capacity.

Exercises

- Given an undirected graph $G = (V, E)$, with three vertices u , v , and w , describe and analyze an algorithm to determine whether there is a path from u to w that passes through v .
- Let $G = (V, E)$ be a directed graph where for each vertex v , the in-degree and out-degree of v are equal. Let u and v be two vertices G , and suppose G contains k edge-disjoint paths from u to v . Under these conditions, must G also contain k edge-disjoint paths from v to u ? Give a proof or a counterexample with explanation.
- A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover all the vertices. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that no cycle cover exists. [Hint: Use bipartite matching!]
- Consider a directed graph $G = (V, E)$ with multiple source vertices $s_1, s_2, \dots, s_\sigma$ and multiple target vertices t_1, t_2, \dots, t_τ , where no vertex is both a source and a target. A *multiterminal flow* is a function $f : E \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the flow conservation constraint at every vertex that is neither a source nor a target. The value $|f|$ of a multiterminal flow is the total excess flow out of *all* the source vertices:

$$|f| := \sum_{i=1}^{\sigma} \left(\sum_w f(s_i \rightarrow w) - \sum_u f(u \rightarrow s_i) \right)$$

As usual, we are interested in finding flows with maximum value, subject to capacity constraints on the edges. (In particular, we don't care how much flow moves from any particular source to any particular target.)

- (a) Consider the following algorithm for computing multiterminal flows. The variables f and f' represent flow functions. The subroutine $\text{MAXFLOW}(G, s, t)$ solves the standard maximum flow problem with source s and target t .

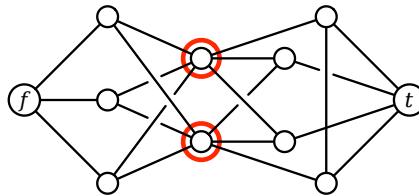
```
MAXMULTIFLOW( $G, s[1..σ], t[1..τ]$ ):
   $f \leftarrow 0$                                 «Initialize the flow»
  for  $i \leftarrow 1$  to  $σ$ 
    for  $j \leftarrow 1$  to  $τ$ 
       $f' \leftarrow \text{MAXFLOW}(G_f, s[i], t[j])$ 
       $f \leftarrow f + f'$                       «Update the flow»
  return  $f$ 
```

Prove that this algorithm correctly computes a maximum multiterminal flow in G .

- (b) Describe a more efficient algorithm to compute a maximum multiterminal flow in G .
5. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the number 2.



6. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A . For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

7. *Ad-hoc networks* are made up of low-powered wireless devices. In principle⁴, these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network.

These devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D .

We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k potential backup devices, all within distance D of x ; we call these k devices the **backup set** of x . Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

So suppose we are given the communication radius D , parameters b and k , and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j . Describe an algorithm that either computes a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

*8. A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe a polynomial-time algorithm to compute, given two rooted trees A and B , the largest common rooted subtree of A and B .

[Hint: Let $LCS(u, v)$ denote the largest common subtree whose root in A is u and whose root in B is v . Your algorithm should compute $LCS(u, v)$ for all vertices u and v using dynamic programming. This would be easy if every vertex had $O(1)$ children, and still straightforward if the children of each node were ordered from left to right and the common subtree had to respect that ordering. But for unordered trees with large degree, you need another trick to combine recursive subproblems efficiently. Don't waste your time trying to reduce the polynomial running time.]

⁴but not really in practice

"Who are you?" said Lunkwill, rising angrily from his seat. "What do you want?"
 "I am Majikthise!" announced the older one.
 "And I demand that I am Vroomfondel!" shouted the younger one.
 Majikthise turned on Vroomfondel. "It's alright," he explained angrily, "you don't need to demand that."
 "Alright!" bawled Vroomfondel banging on a nearby desk. "I am Vroomfondel, and that is not a demand, that is a solid fact! What we demand is solid facts!"
 "No we don't!" exclaimed Majikthise in irritation. "That is precisely what we don't demand!"
 Scarcely pausing for breath, Vroomfondel shouted, "We don't demand solid facts! What we demand is a total absence of solid facts. I demand that I may or may not be Vroomfondel!"

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1979)

*18 Extensions of Maximum Flow

18.1 Maximum Flows with Edge Demands

Now suppose each directed edge e in has both a capacity $c(e)$ and a *demand* $d(e) \leq c(e)$, and we want a flow f of maximum value that satisfies $d(e) \leq f(e) \leq c(e)$ at every edge e . We call a flow that satisfies these constraints a *feasible* flow. In our original setting, where $d(e) = 0$ for every edge e , the zero flow is feasible; however, in this more general setting, even determining whether a feasible flow exists is a nontrivial task.

Perhaps the easiest way to find a feasible flow (or determine that none exists) is to reduce the problem to a standard maximum flow problem, as follows. The input consists of a directed graph $G = (V, E)$, nodes s and t , demand function $d: E \rightarrow \mathbb{R}$, and capacity function $c: E \rightarrow \mathbb{R}$. Let D denote the sum of all edge demands in G :

$$D := \sum_{u \rightarrow v \in E} d(u \rightarrow v).$$

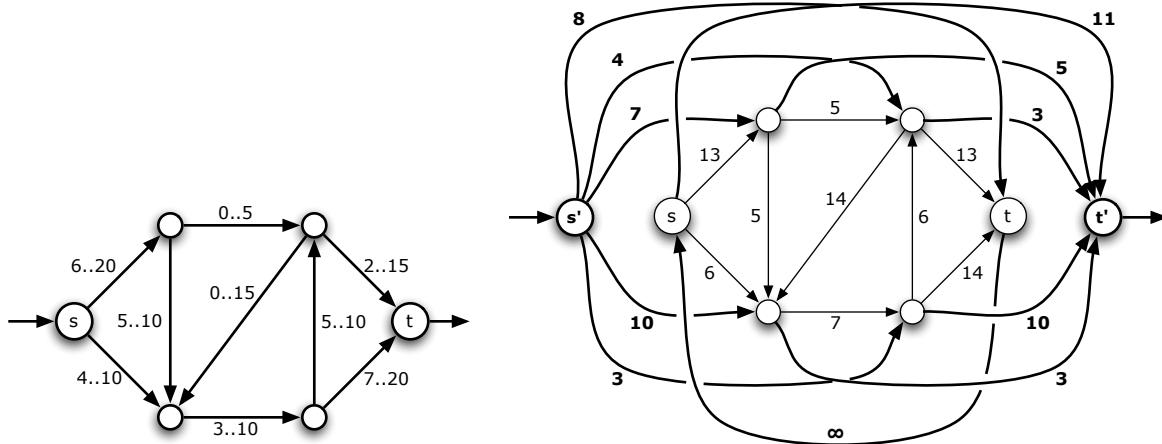
We construct a new graph $G' = (V', E')$ from G by adding new source and target vertices s' and t' , adding edges from s' to each vertex in V , adding edges from each vertex in V to t' , and finally adding an edge from t to s . We also define a new capacity function $c': E' \rightarrow \mathbb{R}$ as follows:

- For each vertex $v \in V$, we set $c'(s' \rightarrow v) = \sum_{u \in V} d(u \rightarrow v)$ and $c'(v \rightarrow t') = \sum_{w \in V} d(v \rightarrow w)$.
- For each edge $u \rightarrow v \in E$, we set $c'(u \rightarrow v) = c(u \rightarrow v) - d(u \rightarrow v)$.
- Finally, we set $c'(s \rightarrow t) = \infty$.

Intuitively, we construct G' by replacing any edge $u \rightarrow v$ in G with three edges: an edge $u \rightarrow v$ with capacity $c(u \rightarrow v) - d(u \rightarrow v)$, an edge $s' \rightarrow v$ with capacity $d(u \rightarrow v)$, and an edge $u \rightarrow t'$ with capacity $d(u \rightarrow v)$. If this construction produces multiple edges from s' to the same vertex v (or to t' from the same vertex v), we merge them into a single edge with the same total capacity.

In G' , the total capacity out of s' and the total capacity into t' are both equal to D . We call a flow with value exactly D a *saturating* flow, since it saturates all the edges leaving s' or entering t' . If G' has a saturating flow, it must be a maximum flow, so we can find it using any max-flow algorithm.

Lemma 1. G has a feasible (s, t) -flow if and only if G' has a saturating (s', t') -flow.



A flow network G with demands and capacities (written $d \dots c$), and the transformed network G' .

Proof: Let $f : E \rightarrow \mathbb{R}$ be a feasible (s, t) -flow in the original graph G . Consider the following function $f' : E' \rightarrow \mathbb{R}$:

$$\begin{aligned} f'(u \rightarrow v) &= f(u \rightarrow v) - d(u \rightarrow v) && \text{for all } u \rightarrow v \in E \\ f'(s' \rightarrow v) &= \sum_{u \in V} d(u \rightarrow v) && \text{for all } v \in V \\ f'(v \rightarrow t') &= \sum_{w \in V} d(v \rightarrow w) && \text{for all } v \in V \\ f'(t \rightarrow s) &= |f| \end{aligned}$$

We easily verify that f' is a saturating (s', t') -flow in G . The admissibility of f implies that $f(e) \geq d(e)$ for every edge $e \in E$, so $f'(e) \geq 0$ everywhere. Admissibility also implies $f(e) \leq c(e)$ for every edge $e \in E$, so $f'(e) \leq c'(e)$ everywhere. Tedious algebra implies that

$$\sum_{u \in V'} f'(u \rightarrow v) = \sum_{w \in V'} f(v \rightarrow w)$$

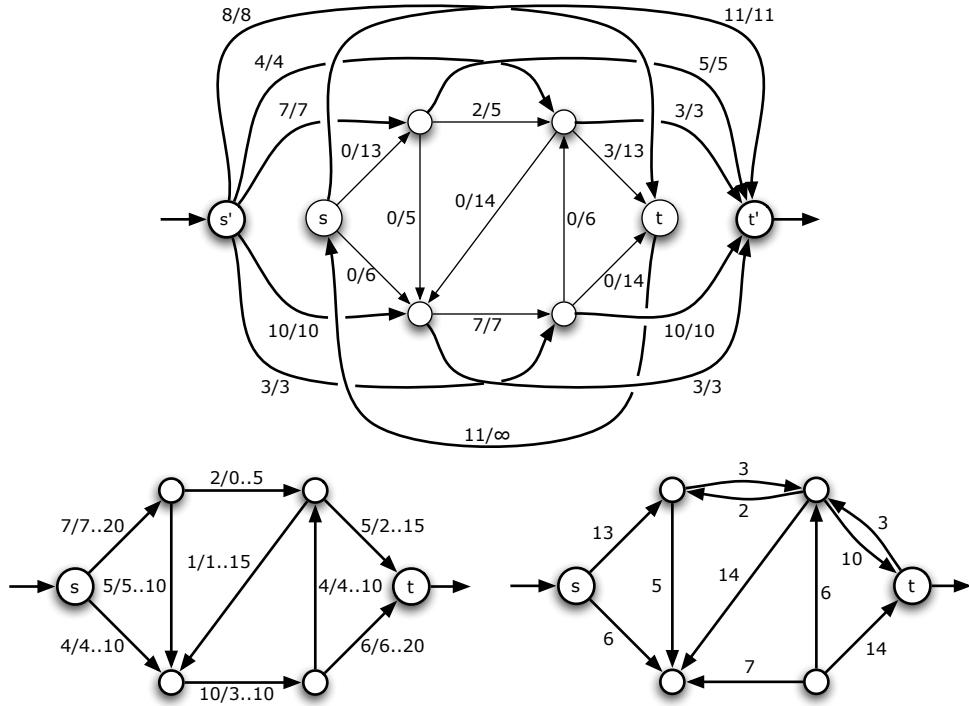
for every vertex $v \in V$ (including s and t). Thus, f' is a legal (s', t') -flow, and every edge out of s' or into t' is clearly saturated. Intuitively, f' diverts $d(u \rightarrow v)$ units of flow from u directly to the new target t' , and injects the same amount of flow into v directly from the new source s' .

The same tedious algebra implies that for any saturating (s', t') -flow $f' : E' \rightarrow \mathbb{R}$ for G' , the function $f = f'|_E + d$ is a feasible (s, t) -flow in G . \square

Thus, we can compute a feasible (s, t) -flow for G , if one exists, by searching for a maximum (s', t') -flow in G' and checking that it is saturating. Once we've found a feasible (s, t) -flow in G , we can transform it into a maximum flow using an augmenting-path algorithm, but with one small change. To ensure that *every* flow we consider is feasible, we must redefine the residual capacity of an edge as follows:

$$c_f(u \rightarrow v) = \begin{cases} c(u \rightarrow v) - f(u \rightarrow v) & \text{if } u \rightarrow v \in E, \\ f(v \rightarrow u) - d(v \rightarrow u) & \text{if } v \rightarrow u \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Otherwise, the algorithm is unchanged. If we use the Dinitz/Edmonds-Karp fat-pipe algorithm, we get an overall running time of $O(VE^2)$.



A saturating flow f' in G' , the corresponding feasible flow f in G , and the corresponding residual network G_f .

18.2 Node Supplies and Demands

Another useful variant to consider allows flow to be injected or extracted from the flow network at vertices other than s or t . Let $x: (V \setminus \{s, t\}) \rightarrow \mathbb{R}$ be an excess function describing how much flow is to be injected (or extracted if the value is negative) at each vertex. We now want a maximum ‘flow’ that satisfies the variant balance condition

$$\sum_{u \in V} f(u \rightarrow v) - \sum_{w \in V} f(v \rightarrow w) = x(v)$$

for every node v except s and t , or prove that no such flow exists. As above, call such a function f a *feasible flow*.

As for flows with edge demands, the only real difficulty in finding a maximum flow under these modified constraints is finding a feasible flow (if one exists). We can reduce this problem to a standard max-flow problem, just as we did for edge demands.

To simplify the transformation, let us assume without loss of generality that the total excess in the network is zero: $\sum_v x(v) = 0$. If the total excess is positive, we add an infinite capacity edge $t \rightarrow \tilde{t}$, where \tilde{t} is a new target node, and set $x(\tilde{t}) = -\sum_v x(v)$. Similarly, if the total excess is negative, we add an infinite capacity edge $\tilde{s} \rightarrow s$, where \tilde{s} is a new source node, and set $x(s) = -\sum_v x(v)$. In both cases, every feasible flow in the modified graph corresponds to a feasible flow in the original graph.

As before, we modify G to obtain a new graph G' by adding a new source s' , a new target t' , an infinite-capacity edge $t \rightarrow s$ from the old target to the old source, and several edges from s' and to t' . Specifically, for each vertex v , if $x(v) > 0$, we add a new edge $s' \rightarrow v$ with capacity $x(v)$, and if $x(v) < 0$, we add an edge $v \rightarrow t'$ with capacity $-x(v)$. As before, we call an (s', t') -flow in G' *saturating* if every edge leaving s' or entering t' is saturated; any saturating flow is a maximum flow. It is easy to check that saturating flows in G' are in direct correspondence with feasible flows in G ; we leave details as an exercise (hint, hint).

Similar reductions allow us to solve several other variants of the maximum flow problem using the same path-augmentation algorithms. For example, we could associate capacities and lower bounds with the vertices instead of (or in addition to) the edges. We could associate a *range* of excesses with every node, instead of a single excess value. We can associate a *cost* $c(e)$ with each edge, and then ask for the maximum-value flow f whose total cost $\sum_e c(e) \cdot f(e)$ is as small as possible. We could even apply all of these extensions at once: upper bounds, lower bounds, and cost functions for the flow through each edge, into each vertex, and out of each vertex.

18.3 Minimum-Cost Flows

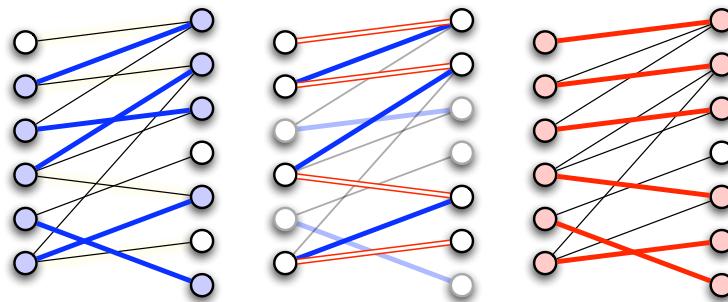
— To be written —

18.4 Maximum-Weight Matchings

Recall from the previous lecture that we can find a maximum-cardinality matching in any bipartite graph in $O(VE)$ time by reduction to the standard maximum flow problem.

Now suppose the input graph has weighted edges, and we want to find the matching with maximum total weight. Given a bipartite graph $G = (U \times W, E)$ and a non-negative weight function $w: E \rightarrow \mathbb{R}$, the goal is to compute a matching M whose total weight $w(M) = \sum_{uw \in M} w(uw)$ is as large as possible. Max-weight matchings can't be found directly using standard max-flow algorithms¹, but we can modify the algorithm for maximum-cardinality matchings described above.

It will be helpful to reinterpret the behavior of our earlier algorithm directly in terms of the original bipartite graph instead of the derived flow network. Our algorithm maintains a matching M , which is initially empty. We say that a vertex is *matched* if it is an endpoint of an edge in M . At each iteration, we find an *alternating path* π that starts and ends at unmatched vertices and alternates between edges in $E \setminus M$ and edges in M . Equivalently, let G_M be the directed graph obtained by orienting every edge in M from W to U , and every edge in $E \setminus M$ from U to W . An alternating path is just a directed path in G_M between two unmatched vertices. Any alternating path has odd length and has exactly one more edge in $E \setminus M$ than in M . The iteration ends by setting $M \leftarrow M \oplus \pi$, thereby increasing the number of edges in M by one. The max-flow/min-cut theorem implies that when there are no more alternating paths, M is a maximum matching.



A matching M with 5 edges, an alternating path π , and the augmented matching $M \oplus \pi$ with 6 edges.

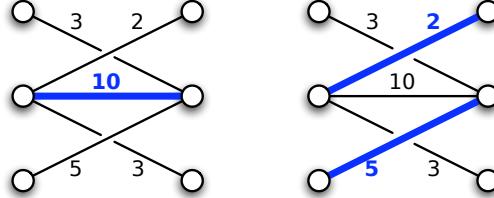
If the edges of G are weighted, we only need to make two changes to the algorithm. First, instead of looking for an arbitrary alternating path at each iteration, we look for the alternating path π such that

¹However, max-flow algorithms can be modified to compute maximum weighted flows, where every edge has both a capacity and a weight, and the goal is to maximize $\sum_{u \rightarrow v} w(u \rightarrow v)f(u \rightarrow v)$.

$M \oplus \pi$ has largest weight. Suppose we weight the edges in the residual graph G_M as follows:

$$\begin{aligned} w'(u \rightarrow w) &= -w(uw) && \text{for all } uw \notin M \\ w'(w \rightarrow u) &= w(uw) && \text{for all } uw \in M \end{aligned}$$

We now have $w(M \oplus \pi) = w(M) - w'(\pi)$. *Thus, the correct augmenting path π must be the directed path in G_M with minimum total residual weight $w'(\pi)$.* Second, because the matching with the maximum weight may not be the matching with the maximum cardinality, we return the heaviest matching considered in any iteration of the algorithm.



A maximum-weight matching is not necessarily a maximum-cardinality matching.

Before we determine the running time of the algorithm, we need to check that it actually finds the maximum-weight matching. After all, it's a greedy algorithm, and greedy algorithms don't work unless you prove them into submission! Let M_i denote the maximum-weight matching in G with exactly i edges. In particular, $M_0 = \emptyset$, and the global maximum-weight matching is equal to M_i for some i . (The figure above show M_1 and M_2 for the same graph.) Let G_i denote the directed residual graph for M_i , let w_i denote the residual weight function for M_i as defined above, and let π_i denote the directed path in G_i such that $w_i(\pi_i)$ is minimized. To simplify the proof, I will assume that there is a unique maximum-weight matching M_i of any particular size; this assumption can be enforced by applying a consistent tie-breaking rule. With this assumption in place, the correctness of our algorithm follows inductively from the following lemma.

Lemma 2. *If G contains a matching with $i + 1$ edges, then $M_{i+1} = M_i \oplus \pi_i$.*

Proof: I will prove the equivalent statement $M_{i+1} \oplus M_i = \pi_{i-1}$. To simplify notation, call an edge in $M_{i+1} \oplus M_i$ **red** if it is an edge in M_{i+1} , and **blue** if it is an edge in M_i .

The graph $M_{i+1} \oplus M_i$ has maximum degree 2, and therefore consists of pairwise disjoint paths and cycles, each of which alternates between red and blue edges. Since G is bipartite, every cycle must have even length. The number of edges in $M_{i+1} \oplus M_i$ is odd; specifically, $M_{i+1} \oplus M_i$ has $2i + 1 - 2k$ edges, where k is the number of edges that are in both matchings. Thus, $M_{i+1} \oplus M_i$ contains an odd number of paths of odd length, some number of paths of even length, and some number of cycles of even length.

Let γ be a cycle in $M_{i+1} \oplus M_i$. Because γ has an equal number of edges from each matching, $M_i \oplus \gamma$ is another matching with i edges. The total weight of this matching is exactly $w(M_i) - w_i(\gamma)$, which must be less than $w(M_i)$, so $w_i(\gamma)$ must be positive. On the other hand, $M_{i+1} \oplus \gamma$ is a matching with $i + 1$ edges whose total weight is $w(M_{i+1}) + w_i(\gamma) < w(M_{i+1})$, so $w_i(\gamma)$ must be negative! We conclude that no such cycle γ exists; $M_{i+1} \oplus M_i$ consists entirely of disjoint paths.

Exactly the same reasoning implies that no path in $M_{i+1} \oplus M_i$ has an even number of edges.

Finally, since the number of red edges in $M_{i+1} \oplus M_i$ is one more than the number of blue edges, the number of paths that start with a red edge is exactly one more than the number of paths that start with a blue edge. The same reasoning as above implies that $M_{i+1} \oplus M_i$ does not contain a blue-first path, because we can pair it up with a red-first path.

We conclude that $M_{i+1} \oplus M_i$ consists of a single alternating path π whose first edge is red. Since $w(M_{i+1}) = w(M_i) - w_i(\pi)$, the path π must be the one with minimum weight $w_i(\pi)$. \square

We can find the alternating path π_i using a single-source shortest path algorithm. Modify the residual graph G_i by adding zero-weight edges from a new source vertex s to every *unmatched* node in U , and from every *unmatched* node in W to a new target vertex t , exactly as in our unweighted matching algorithm. Then π_i is the shortest path from s to t in this modified graph. Since M_i is the maximum-weight matching with i vertices, G_i has no negative cycles, so this shortest path is well-defined. We can compute the shortest path in G_i in $O(VE)$ time using Shimbels's algorithm, so the overall running time of our algorithm is $O(V^2E)$.

The residual graph G_i has negative-weight edges, so we can't speed up the algorithm by replacing Shimbels's algorithm with Dijkstra's. However, we can use a variant of Johnson's all-pairs shortest path algorithm to improve the running time to $O(VE + V^2 \log V)$. Let $d_i(v)$ denote the distance from s to v in the residual graph G_i , using the distance function w_i . Let \tilde{w}_i denote the modified distance function $\tilde{w}_i(u \rightarrow v) = d_{i-1}(u) + w_i(u \rightarrow v) - d_{i-1}(v)$. As we argued in the discussion of Johnson's algorithm, shortest paths with respect to w_i are still shortest paths with respect to \tilde{w}_i . Moreover, $\tilde{w}_i(u \rightarrow v) > 0$ for every edge $u \rightarrow v$ in G_i :

- If $u \rightarrow v$ is an edge in G_{i-1} , then $w_i(u \rightarrow v) = w_{i-1}(u \rightarrow v)$ and $d_{i-1}(v) \leq d_{i-1}(u) + w_{i-1}(u \rightarrow v)$.
- If $u \rightarrow v$ is not in G_{i-1} , then $w_i(u \rightarrow v) = -w_{i-1}(v \rightarrow u)$ and $v \rightarrow u$ is an edge in the shortest path π_{i-1} , so $d_{i-1}(u) = d_{i-1}(v) + w_{i-1}(v \rightarrow u)$.

Let $\tilde{d}_i(v)$ denote the shortest path distance from s to v with respect to the distance function \tilde{w}_i . Because \tilde{w}_i is positive everywhere, we can quickly compute $\tilde{d}_i(v)$ for all v using Dijkstra's algorithm. This gives us both the shortest alternating path π_i and the distances $d_i(v) = \tilde{d}_i(v) + d_{i-1}(v)$ needed for the next iteration.

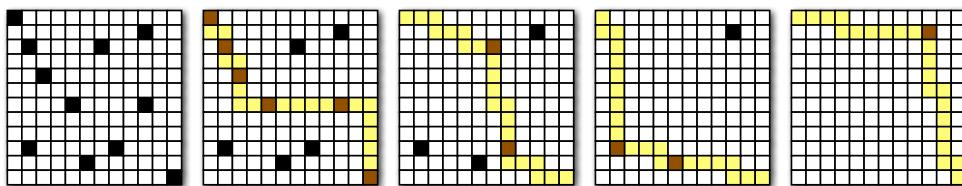
Exercises

1. Suppose we are given a directed graph $G = (V, E)$, two vertices s and t , and a capacity function $c: V \rightarrow \mathbb{R}^+$. A flow f is *feasible* if the total flow into every vertex v is at most $c(v)$:

$$\sum_u f(u \rightarrow v) \leq c(v) \quad \text{for every vertex } v.$$

Describe and analyze an efficient algorithm to compute a feasible flow of maximum value.

2. Suppose we are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{TRUE}$ if and only if cell (i, j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. Our goal is to cover the marked cells with as few monotone paths as possible.



Greedyly covering the marked cells in a grid with four monotone paths.

- (a) Describe an algorithm to find a monotone path that covers the largest number of marked cells.
- (b) There is a natural greedy heuristic to find a small cover by monotone paths: If there are any marked cells, find a monotone path π that covers the largest number of marked cells, unmark any cells covered by π those marked cells, and recurse. Show that this algorithm does *not* always compute an optimal solution.
- (c) Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.

*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)

*I 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. Kantorovich’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 his Soviet colleagues had 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:

$$\begin{aligned} & \text{maximize } \sum_{j=1}^d c_j x_j \\ & \text{subject to } \sum_{j=1}^d a_{ij} x_j \leq b_i \quad \text{for each } i = 1 \dots p \\ & \quad \sum_{j=1}^d a_{ij} x_j = b_i \quad \text{for each } i = p+1 \dots p+q \\ & \quad \sum_{j=1}^d a_{ij} x_j \geq b_i \quad \text{for each } i = p+q+1 \dots n \end{aligned}$$

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 $x \mapsto c \cdot x$ 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:

$$\begin{aligned} & \text{maximize} \sum_{j=1}^d c_j x_j \\ & \text{subject to} \sum_{j=1}^d a_{ij} x_j \leq b_i \quad \text{for each } i = 1..n \\ & \quad x_j \geq 0 \quad \text{for each } j = 1..d \end{aligned}$$

We can express this canonical form more compactly as follows. For two vectors $x = (x_1, x_2, \dots, x_d)$ and $y = (y_1, y_2, \dots, y_d)$, the expression $x \geq y$ means that $x_i \geq y_i$ for every index i .

$$\begin{aligned} & \max c \cdot x \\ & \text{s.t. } Ax \leq b \\ & \quad x \geq 0 \end{aligned}$$

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 $x_j^+ \geq 0$ and $x_j^- \geq 0$.
- Replace any equality constraint $\sum_j a_{ij} x_j = b_i$ with two inequality constraints $\sum_j a_{ij} x_j \geq b_i$ and $\sum_j a_{ij} x_j \leq b_i$.
- Replace any upper bound $\sum_j a_{ij} x_j \geq b_i$ with the equivalent lower bound $\sum_j -a_{ij} x_j \leq -b_i$.

This conversion potentially double the number of variables and the number of constraints; fortunately, it is rarely necessary in practice.

Another useful format for linear programming problems is *slack form*², in which every inequality is of the form $x_j \geq 0$:

$$\begin{aligned} & \max c \cdot x \\ & \text{s.t. } Ax = b \\ & \quad x \geq 0 \end{aligned}$$

It's fairly easy to convert any linear programming problem into slack form. Slack form is especially useful in executing the simplex algorithm (which we'll see in the next lecture).

I.1 The Geometry of Linear Programming

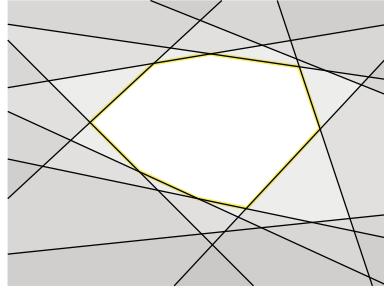
A point $x \in \mathbb{R}^d$ is *feasible* with respect to some linear programming problem if it satisfies all the linear constraints. The set of all feasible points is called the *feasible region* for that linear program. The feasible region has a particularly nice geometric structure that lends some useful intuition to the linear programming algorithms we'll see later.

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

¹Confusingly, some authors call this *standard form*.

²Confusingly, some authors call this *standard form*.

(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} .



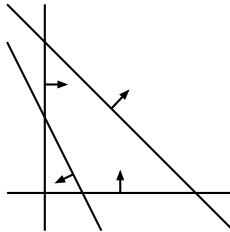
A two-dimensional polyhedron (white) defined by 10 linear inequalities.

By rotating \mathbb{R}^d (or choosing a coordinate frame) so that the objective function points downward, we can express *any* linear programming problem in the following geometric form:

Find the lowest point in a given polyhedron.

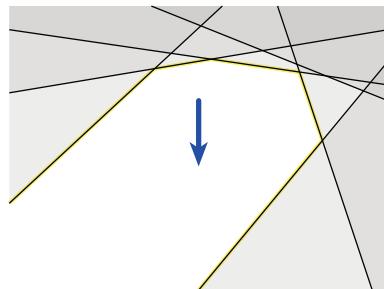
With this geometry in hand, we can easily picture two pathological cases where a given linear programming problem has no solution. The first possibility is that there are no feasible points; in this case the problem is called *infeasible*. For example, the following LP problem is infeasible:

$$\begin{aligned} &\text{maximize } x - y \\ &\text{subject to } 2x + y \leq 1 \\ &\quad x + y \geq 2 \\ &\quad x, y \geq 0 \end{aligned}$$



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.

I.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.

$$\begin{array}{ll} \text{maximize} & d_t \\ \text{subject to} & d_s = 0 \\ & d_v - d_u \leq \ell_{u \rightarrow v} \quad \text{for every edge } u \rightarrow v \end{array}$$

Here, $\ell_{u \rightarrow v}$ is the length of the edge $u \rightarrow v$. 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:

$$\begin{array}{ll} \text{maximize} & \sum_v d_v \\ \text{subject to} & d_s = 0 \\ & d_v - d_u \leq \ell_{u \rightarrow v} \quad \text{for every edge } u \rightarrow v \end{array}$$

There is another formulation of shortest paths as an LP minimization problem using an indicator variable $x_{u \rightarrow v}$ for each edge $u \rightarrow v$.

$$\begin{array}{ll} \text{minimize} & \sum_{u \rightarrow v} \ell_{u \rightarrow v} \cdot x_{u \rightarrow v} \\ \text{subject to} & \sum_u x_{u \rightarrow s} - \sum_w x_{s \rightarrow w} = 1 \\ & \sum_u x_{u \rightarrow t} - \sum_w x_{t \rightarrow w} = -1 \\ & \sum_u x_{u \rightarrow v} - \sum_w x_{v \rightarrow w} = 0 \quad \text{for every vertex } v \neq s, t \\ & x_{u \rightarrow v} \geq 0 \quad \text{for every edge } u \rightarrow v \end{array}$$

Intuitively, $x_{u \rightarrow v} = 1$ means $u \rightarrow v$ lies on the shortest path from s to t , and $x_{u \rightarrow v} = 0$ means $u \rightarrow v$ does not lie on this shortest path. 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.

However, there are other feasible solutions, possibly even *optimal* solutions, with non-integral values that do not represent paths. Nevertheless, there is always 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. (This fact is by no means obvious, but a proof is beyond the scope of these notes.) Moreover, in any optimal solution, even if not every x_e is an integer, the objective function gives the shortest path distance!

I.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:

$$\begin{aligned} \text{maximize} \quad & \sum_w f_{s \rightarrow w} - \sum_u f_{u \rightarrow s} \\ \text{subject to} \quad & \sum_w f_{v \rightarrow w} - \sum_u f_{u \rightarrow v} = 0 \quad \text{for every vertex } v \neq s, t \\ & f_{u \rightarrow v} \leq c_{u \rightarrow v} \quad \text{for every edge } u \rightarrow v \\ & f_{u \rightarrow v} \geq 0 \quad \text{for every edge } u \rightarrow v \end{aligned}$$

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 $X_{u \rightarrow v}$ for each edge $u \rightarrow v$, indicating whether $u \in S$ and $v \in T$, where (S, T) is some (s, t) -cut.³

$$\begin{aligned} \text{minimize} \quad & \sum_{u \rightarrow v} c_{u \rightarrow v} \cdot X_{u \rightarrow v} \\ \text{subject to} \quad & X_{u \rightarrow v} + S_v - S_u \geq 0 \quad \text{for every edge } u \rightarrow v \\ & X_{u \rightarrow v} \geq 0 \quad \text{for every edge } u \rightarrow v \\ & S_s = 1 \\ & S_t = 0 \end{aligned}$$

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!

I.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:

Primal (II)	\iff	Dual (II)
$\begin{aligned} \max \quad & c \cdot x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$		$\begin{aligned} \min \quad & y \cdot b \\ \text{s.t.} \quad & yA \geq c \\ & y \geq 0 \end{aligned}$

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

Primal (II)	\iff	Dual (II)
$\begin{aligned} \max \quad & c \cdot x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$		$\begin{aligned} \max \quad & -b^\top \cdot y^\top \\ \text{s.t.} \quad & -A^\top y^\top \leq -c \\ & y^\top \geq 0 \end{aligned}$

³These two linear programs are not quite *syntactic* duals; I’ve added two redundant variables S_s and S_t to the min-cut program to increase readability.

⁴For 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.

Written in this form, it should be immediately clear that duality is an *involution*: The dual of the dual linear program Π 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 Π has an optimal solution y^* such that $c \cdot x^* = y^* A x^* = y^* \cdot b$.*

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 Π , then $c \cdot x \leq y^* A x \leq y^* \cdot b$.*

Proof: Because x is feasible for Π , we have $Ax \leq b$. Since y is positive, we can multiply both sides of the inequality to obtain $yAx \leq y \cdot b$. Conversely, y is feasible for Π and x is positive, so $yAx \geq c \cdot x$. \square

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.

I.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:

$$\begin{aligned} & \text{maximize} && 4x_1 + x_2 + 3x_3 \\ & \text{subject to} && x_1 + 4x_2 \leq 2 \\ & && 3x_1 - x_2 + x_3 \leq 4 \\ & && x_1, x_2, x_3 \geq 0 \end{aligned}$$

Let σ^* denote the optimum objective value for this LP. The feasible solution $x = (1, 0, 0)$ gives us a lower bound $\sigma^* \geq 4$. A different feasible solution $x = (0, 0, 3)$ gives us a better lower bound $\sigma^* \geq 9$. 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 :

$$\begin{aligned} & \text{maximize} && 4x_1 + x_2 + 3x_3 \\ & \text{subject to} && y_1(x_1 + 4x_2) \leq 2y_1 \\ & && y_2(3x_1 - x_2 + x_3) \leq 4y_2 \\ & && x_1, x_2, x_3 \geq 0 \end{aligned}$$

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:

$$(y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \leq 2y_1 + 4y_2.$$

⁵For historical reasons, maximization LPs tend to be called ‘primal’ and minimization LPs tend to be called ‘dual’. This is a pointless religious tradition, nothing more. Duality is a relationship between LP problems, not a type of LP problem.

⁶This 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’.

Now suppose that each y_i is larger than the i th coefficient of the objective function:

$$y_1 + 3y_2 \geq 4, \quad 4y_1 - y_2 \geq 1, \quad y_2 \geq 3.$$

This assumption lets us derive an upper bound on the objective value of *any* feasible solution:

$$4x_1 + x_2 + 3x_3 \leq (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \leq 2y_1 + 4y_2. \quad (*)$$

In particular, by plugging in the optimal solution (x_1^*, x_2^*, x_3^*) for the original LP, we obtain the following upper bound on σ^* :

$$\sigma^* = 4x_1^* + x_2^* + 3x_3^* \leq 2y_1 + 4y_2.$$

Now it's natural to ask how tight we can make this upper bound. How small can we make the expression $2y_1 + 4y_2$ without violating any of the inequalities we used to prove the upper bound? This is just another linear programming problem.

$$\begin{aligned} & \text{minimize} && 2y_1 + 4y_2 \\ & \text{subject to} && y_1 + 3y_2 \geq 4 \\ & && 4y_1 - y_2 \geq 1 \\ & && y_2 \geq 3 \\ & && y_1, y_2 \geq 0 \end{aligned}$$

In fact, this is precisely the dual of our original linear program! Moreover, inequality $(*)$ is just an instantiation of the Weak Duality Theorem.

I.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 Π , such that $c \cdot x^* = y^* A x^* = y^* \cdot b$.*

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 $n \geq d$; 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 $y^* \cdot b = c \cdot x^*$:

$$y^* \cdot b = y_\bullet^* \cdot b_\bullet + y_\circ^* \cdot b_\circ = y_\bullet^* \cdot b_\bullet = cA_\bullet^{-1}b_\bullet = c \cdot x^*.$$

(The existence of the inverse matrix A_\bullet^{-1} follows from our non-degeneracy assumption.) Similarly, it's easy to verify that $y^* A \geq c$:

$$y^* A = y_\bullet^* A_\bullet + y_\circ^* A_\circ = y_\bullet^* A_\bullet = c.$$

Once we prove that y^*_\bullet is non-negative, and therefore feasible, the Weak Duality Theorem implies the result. We chose $y^*_\circ = 0$. As we will see below, the optimality of x^* implies the strict inequality $y^*_\bullet > 0$ —we had to use optimality 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 $a_i \cdot x^* = b_i$ in \mathbb{R}^d . The vector a_i is normal to this hyperplane and points *out* of the feasible region. The vectors a_1, \dots, 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:

$$c = y^*_\bullet A_\bullet = \sum_{i=1}^d y_i^* a_i.$$

In other words, y^*_\bullet lists the coefficients of the objective vector c in the coordinate frame a_1, \dots, a_d .

The point x^* lies on exactly d constraint hyperplanes; any $d - 1$ of these hyperplanes determine a line through x^* . For each $1 \leq i \leq d$, let ℓ_i denote the line that lies on all but the i th constraint plane, and let v_i denote a vector based at x^* that points into the halfspace $a_i x \leq b_i$ along the line ℓ_i . This vector lies along an edge of the feasible polytope. For all $j \neq i$, we have $a_j \cdot v_i = 0$. Thus, we can write

$$A_\bullet v_i = (0, \dots, 0, a_i \cdot v_i, 0, \dots, 0)^\top$$

where the scalar $a_i \cdot v_i$ appears in the i th coordinate. It follows that

$$c \cdot v_i = y^*_\bullet A_\bullet v_i = y_i^* (a_i \cdot v_i).$$

The optimality of x^* implies that $c \cdot v_i < 0$, and because v_i points into the feasible region while a_i points out, we have $a_i \cdot v_i < 0$. We conclude that $y_i^* > 0$. We're done! \square

Exercises

- (a) Describe precisely how to dualize a linear program written in slack form:

$\begin{aligned} \max \quad & c \cdot x \\ \text{s.t. } & Ax = b \\ & x \geq 0 \end{aligned}$

- (b) Describe precisely how to dualize a linear program written in general form:

$$\begin{aligned} & \text{maximize} \sum_{j=1}^d c_j x_j \\ & \text{subject to} \sum_{j=1}^d a_{ij} x_j \leq b_i \quad \text{for each } i = 1 \dots p \\ & \quad \sum_{j=1}^d a_{ij} x_j = b_i \quad \text{for each } i = p+1 \dots p+q \\ & \quad \sum_{j=1}^d a_{ij} x_j \geq b_i \quad \text{for each } i = p+q+1 \dots n \end{aligned}$$

In both cases, keep the number of dual variables as small as possible.

2. A matrix $A = (a_{ij})$ is *skew-symmetric* if and only if $a_{ji} = -a_{ij}$ for all indices $i \neq j$; in particular, every skew-symmetric matrix is square. A canonical linear program $\max\{c \cdot x \mid Ax \leq b; x \geq 0\}$ is *self-dual* if the matrix A is skew-symmetric and the objective vector c is *equal* to the constraint vector b .
- (a) Prove any self-dual linear program Π is equivalent to its dual program Π^* .
 - (b) Show that any linear program Π with d variables and n constraints can be transformed into a self-dual linear program with $n + d$ variables and $n + d$ constraints. The optimal solution to the self-dual program should include both the optimal solution for Π (in d of the variables) and the optimal solution for the dual program Π^* (in the other n variables).
3. (a) Model the maximum-cardinality bipartite matching problem as a linear programming problem. The input is a bipartite graph $G = (U \cup V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G . Your linear program should have one variable for each edge.
- (b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?
4. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.
- (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.
 - (b) Prove that finding the optimal feasible solution to an integer program is NP-hard.
- [Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]*
- *5. *Helly's theorem* states that for any collection of convex bodies in \mathbb{R}^d , if every $d + 1$ of them intersect, then there is a point lying in the intersection of all of them. Prove Helly's theorem for the special case where the convex bodies are halfspaces. Equivalently, show that if a system of linear inequalities $Ax \leq b$ does not have a solution, then we can select $d + 1$ of the inequalities such that the resulting subsystem also does not have a solution. *[Hint: Construct a dual LP from the system by choosing a 0 cost vector.]*

Simplicibus itaque verbis gaudet Mathematica Veritas, cum etiam per se simplex sit Veritatis oratio. [And thus Mathematical Truth prefers simple words, because the language of Truth is itself simple.]

— Tycho Brahe (quoting Seneca (quoting Euripides))
Epistolarum astronomicarum liber primus (1596)

*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.*

— Sankara, *Viveka-Chudamani* (c. 700), translator unknown

*J 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, no simplex variant is known to run in sub-exponential time in the worst case. 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.

J.1 Bases, Feasibility, and Local Optimality

Consider the canonical linear program $\max\{c \cdot x \mid Ax \leq b, x \geq 0\}$, 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: Every subset of d constraint hyperplanes intersect in a single point; at most d constraint hyperplanes pass through any point; and objective vector is linearly independent from any $d - 1$ constraint vectors.

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 subset of d hyperplanes (that is, 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. If there are no feasible bases, the linear program is **infeasible**.

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 II. For this reason, locally optimal bases are sometimes also called *dual feasible*. If there are no locally optimal bases, the linear program is *unbounded*.¹

Two bases are *neighbors* if they have $d - 1$ constraints in common. Equivalently, in geometric terms, two vertices are neighbors if they lie on a *line* determined by some $d - 1$ constraint hyperplanes. Every basis is a neighbor of exactly dn other bases; to change a basis into one of its neighbors, there are d choices for which constraint to remove and n choices for which constraint to add. The graph of vertices and edges on the boundary of the feasible polyhedron is a subgraph of the basis graph.

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; equivalently, 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.

J.2 The Primal Simplex Algorithm: Falling Marbles

From a geometric standpoint, Dantzig's simplex algorithm is very simple. The input is a set H of halfspaces; we want the lowest vertex in the intersection of these halfspaces.

```
SIMPLEX1( $H$ ):
  if  $\cap H = \emptyset$ 
    return INFEASIBLE
   $x \leftarrow$  any feasible vertex
  while  $x$  is not locally optimal
    ((pivot downward, maintaining feasibility))
    if every feasible neighbor of  $x$  is higher than  $x$ 
      return UNBOUNDED
    else
       $x \leftarrow$  any feasible neighbor of  $x$  that is lower than  $x$ 
  return  $x$ 
```

Let's ignore the first three lines for the moment. The algorithm maintains a feasible vertex x . At each so-called *pivot* operation, the algorithm moves to a *lower* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the feasible vertex x is locally optimal, and therefore the optimum vertex, or the feasible vertex x is not locally optimal but has no lower feasible neighbor, in which case the feasible region must be unbounded.

Notice that we have not specified *which* neighbor to choose at each pivot. Several different pivoting rules have been proposed, but for almost every known pivot rule, there is an input polyhedron that requires an exponential number of pivots under that rule. No pivoting rule is known that guarantees a polynomial number of pivots in the worst case.²

J.3 The Dual Simplex Algorithm: Rising Bubbles

We can also geometrically interpret the execution of the simplex algorithm on the dual linear program II. Again, the input is a set H of halfspaces, and we want the lowest vertex in the intersection of these

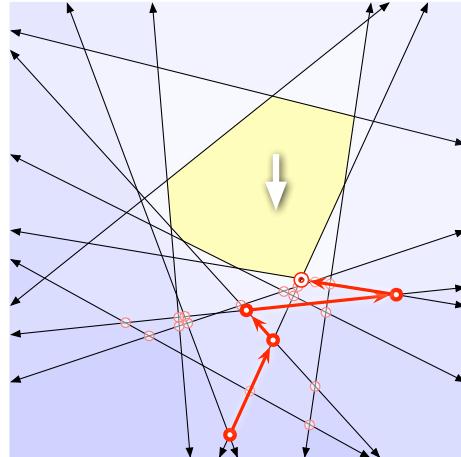
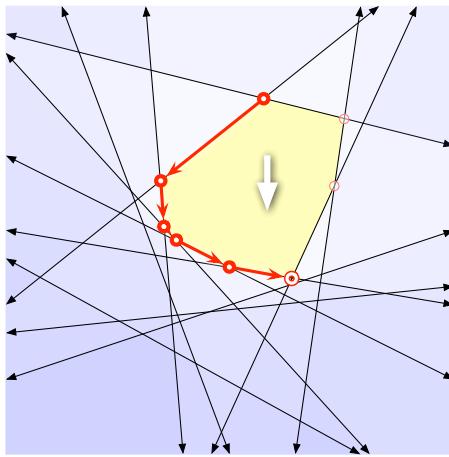
¹For non-degenerate linear programs, the feasible region is unbounded in the objective direction if and only if no basis is locally optimal. However, there are degenerate linear programs with no locally optimal basis that are infeasible.

²In 1957, Hirsch conjectured that for *any* linear programming instance with d variables and $n + d$ constraints, starting at any feasible basis, there is a sequence of *at most n* pivots that leads to the optimal basis. Hirsch's conjecture is still open 50 years later; no counterexamples have ever been found, but no proof is known except in a few special cases. Truly our ignorance is unbounded (or at least dual infeasible).

halfspaces. By the Strong Duality Theorem, this is the same as the *highest locally-optimal* vertex in the hyperplane arrangement.

```
SIMPLEX2( $H$ ):
    if there is no locally optimal vertex
        return UNBOUNDED
     $x \leftarrow$  any locally optimal vertex
    while  $x$  is not feasible
        ((pivot upward, maintaining local optimality))
        if every locally optimal neighbor of  $x$  is lower than  $x$ 
            return INFEASIBLE
        else
             $x \leftarrow$  any locally-optimal neighbor of  $x$  that is higher than  $x$ 
    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, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the locally optimal vertex x is feasible, and therefore the optimum vertex, or the locally optimal vertex x is not feasible but has no higher locally optimal neighbor, in which case the problem must be infeasible.



The primal simplex (falling marble) algorithm in action. The dual simplex (rising bubble) algorithm in action.

From the standpoint of linear algebra, there is absolutely no difference between running SIMPLEX1 on any linear program Π and running SIMPLEX2 on the dual linear program Π^* . The actual *code* is identical. The only difference between the two algorithms is how we interpret the linear algebra geometrically.

J.4 Computing the Initial Basis

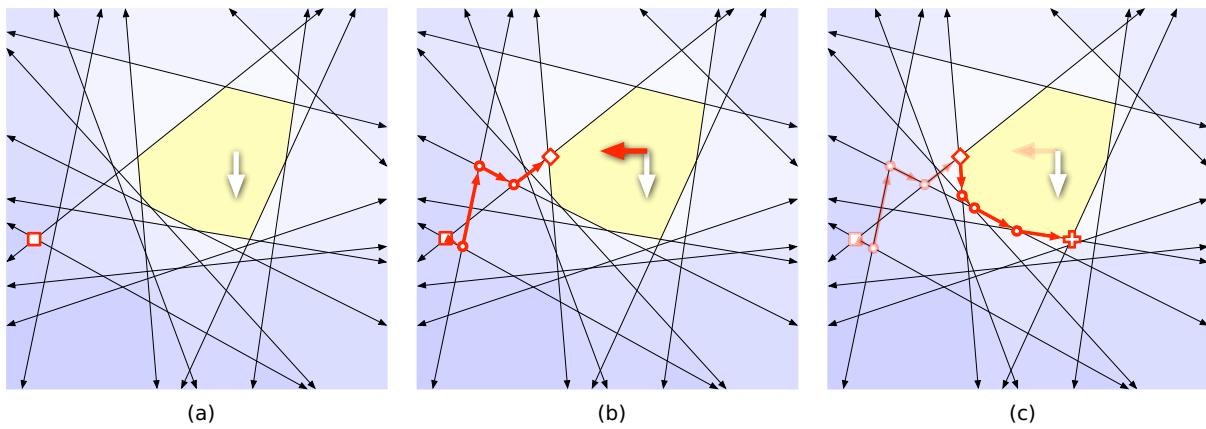
To complete our description of the simplex algorithm, we need to describe how to find the initial vertex x . Our algorithm relies on the following simple observations.

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 .

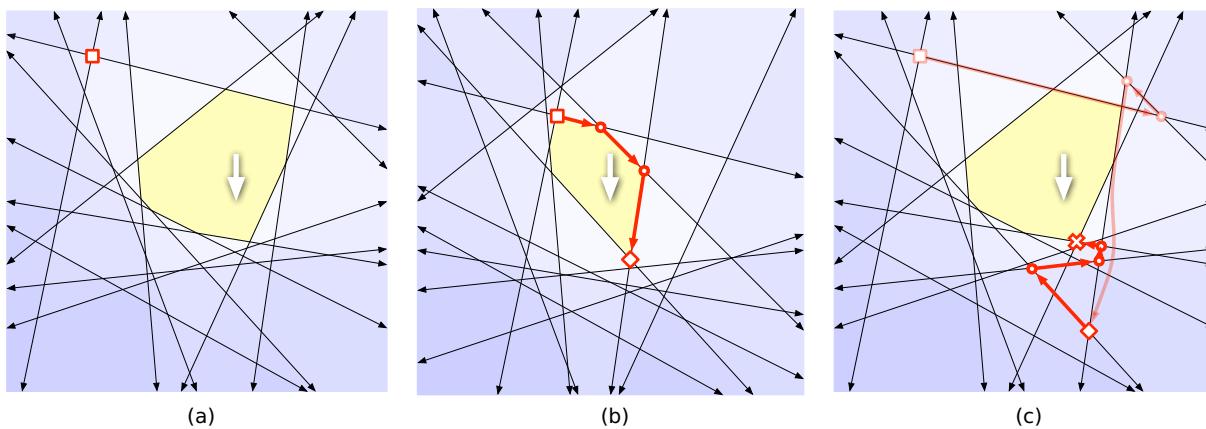
The second important observation is that *every* basis is locally optimal for *some* objective function. Specifically, it suffices to choose any vector that has a positive inner product with each of the normal vectors of the d chosen hyperplanes. Equivalently, we can make *any* basis feasible by translating 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 strictly 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!



(a) Choose any basis. (b) Rotate objective to make it locally optimal, and pivot 'upward' to find a feasible basis.
 (c) Pivot downward to the optimum basis for the original objective.

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 using the (primal) simplex algorithm. This vertex must be locally optimal, even after we restore the hyperplanes to their original locations!



(a) Choose any basis. (b) Translate constraints to make it feasible, and pivot downward to find a locally optimal basis.
 (c) Pivot upward to the optimum basis for the original constraints.

Here are more complete descriptions of the simplex algorithm with this initialization rule, in both primal and dual forms. As usual, the input is a set H of halfspaces, and the algorithms either return the lowest vertex in the intersection of these halfspaces or report that no such vertex exists.

```
SIMPLEX1( $H$ ):
 $x \leftarrow$  any vertex
 $\tilde{H} \leftarrow$  any rotation of  $H$  that makes  $x$  locally optimal
while  $x$  is not feasible
    if every locally optimal neighbor of  $x$  is lower (wrt  $\tilde{H}$ ) than  $x$ 
        return INFEASIBLE
    else
         $x \leftarrow$  any locally optimal neighbor of  $x$  that is higher (wrt  $\tilde{H}$ ) than  $x$ 
while  $x$  is not locally optimal
    if every feasible neighbor of  $x$  is higher than  $x$ 
        return UNBOUNDED
    else
         $x \leftarrow$  any feasible neighbor of  $x$  that is lower than  $x$ 
return  $x$ 
```

```
SIMPLEX2( $H$ ):
 $x \leftarrow$  any vertex
 $\tilde{H} \leftarrow$  any translation of  $H$  that makes  $x$  feasible
while  $x$  is not locally optimal
    if every feasible neighbor of  $x$  is higher (wrt  $\tilde{H}$ ) than  $x$ 
        return UNBOUNDED
    else
         $x \leftarrow$  any feasible neighbor of  $x$  that is lower (wrt  $\tilde{H}$ ) than  $x$ 
while  $x$  is not feasible
    if every locally optimal neighbor of  $x$  is lower than  $x$ 
        return INFEASIBLE
    else
         $x \leftarrow$  any locally-optimal neighbor of  $x$  that is higher than  $x$ 
return  $x$ 
```

J.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 will implicitly assume that the linear program is both feasible and bounded. (If necessary, we can guarantee boundedness by adding a single halfspace to H , and we can guarantee feasibility by adding a dimension.) A point x **violates** a constraint h if it is not contained in the corresponding halfspace.

```

SEIDELLP( $H, B$ ) :
  if  $|B| = d$ 
    return  $\bigcap B$ 
  if  $|H \cup B| = d$ 
    return  $\bigcap(H \cup B)$ 
   $h \leftarrow$  random element of  $H$ 
   $x \leftarrow$  SEIDELLP( $H \setminus h, B$ )      (*)
  if  $x$  violates  $h$ 
    return SEIDELLP( $H \setminus h, B \cup \partial h$ )
  else
    return  $x$ 

```

The point x recursively computed in line (*) is the optimal solution if and only if the random halfspace h is *not* one of the d halfspaces that define the optimal solution. In other words, the probability of calling $\text{SEIDELLP}(H, B \cup h)$ is exactly $(d - b)/n$. Thus, we have the following recurrence for the expected number of recursive calls for this algorithm:

$$T(n, b) = \begin{cases} 1 & \text{if } b = d \text{ or } n + b = d \\ T(n - 1, b) + \frac{d - b}{n} \cdot T(n - 1, b + 1) & \text{otherwise} \end{cases}$$

The recurrence is somewhat simpler if we write $\delta = d - b$:

$$T(n, \delta) = \begin{cases} 1 & \text{if } \delta = 0 \text{ or } n = \delta \\ T(n - 1, \delta) + \frac{\delta}{n} \cdot T(n - 1, \delta - 1) & \text{otherwise} \end{cases}$$

It's easy to prove by induction that $T(n, \delta) = O(\delta! n)$:

$$\begin{aligned} T(n, \delta) &= T(n - 1, \delta) + \frac{\delta}{n} \cdot T(n - 1, \delta - 1) \\ &\leq \delta!(n - 1) + \frac{\delta}{n}(\delta - 1)! \cdot (n - 1) && [\text{induction hypothesis}] \\ &= \delta!(n - 1) + \delta! \frac{n - 1}{n} \\ &\leq \delta! n \end{aligned}$$

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 $O(d! \cdot n)$ ***expected time***.

Exercises

1. Fix a non-degenerate linear program in canonical form with d variables and $n + d$ constraints.
 - (a) Prove that every *feasible* basis has exactly d *feasible* neighbors.
 - (b) Prove that every *locally optimal* basis has exactly n *locally optimal* neighbors.

2. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time. Your algorithm should return an optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. [Hint: Add one variable and one constraint.]
3. (a) Give an example of a non-empty polyhedron $Ax \leq b$ that is unbounded for every objective vector c .
(b) Give an example of an infeasible linear program whose dual is also infeasible.
In both cases, your linear program will be degenerate.
4. Describe and analyze an algorithm that solves the following problem in $O(n)$ time: Given n red points and n blue points in the plane, either find a line that separates every red point from every blue point, or prove that no such line exists.
5. The single-source shortest path problem can be formulated as a linear programming problem, with one variable d_v for each vertex $v \neq s$ in the input graph, as follows:

$$\begin{aligned} &\text{maximize} && \sum_v d_v \\ &\text{subject to} && d_v \leq \ell_{s \rightarrow v} \quad \text{for every edge } s \rightarrow v \\ & && d_v - d_u \leq \ell_{u \rightarrow v} \quad \text{for every edge } u \rightarrow v \text{ with } u \neq s \\ & && d_v \geq 0 \quad \text{for every vertex } v \neq s \end{aligned}$$

This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of distances. Assume that the edge weights $\ell_{u \rightarrow v}$ are all non-negative and that there is a unique shortest path between any two vertices in the graph.

- (a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?
(b) Show that in the optimal basis, every variable d_v is equal to the shortest-path distance from s to v .
(c) Describe the primal simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?
(d) Describe the dual simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?
(e) Is Dijkstra's algorithm an instance of the simplex method? Justify your answer.
(f) Is Shimbel's algorithm an instance of the simplex method? Justify your answer.
6. The maximum (s, t) -flow problem can be formulated as a linear programming problem, with one

variable $f_{u \rightarrow v}$ for each edge $u \rightarrow v$ in the input graph:

$$\begin{aligned} & \text{maximize} && \sum_w f_{s \rightarrow w} - \sum_u f_{u \rightarrow s} \\ & \text{subject to} && \sum_w f_{v \rightarrow w} - \sum_u f_{u \rightarrow v} = 0 \quad \text{for every vertex } v \neq s, t \\ & && f_{u \rightarrow v} \leq c_{u \rightarrow v} \quad \text{for every edge } u \rightarrow v \\ & && f_{u \rightarrow v} \geq 0 \quad \text{for every edge } u \rightarrow v \end{aligned}$$

This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of flows.

- (a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?
 - (b) Show that the optimal basis represents a maximum flow.
 - (c) Describe the primal simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?
 - (d) Describe the dual simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?
 - (e) Is the Ford-Fulkerson augmenting path algorithm an instance of the simplex method? Justify your answer. *[Hint: There is a one-line argument.]*
7. (a) Formulate the minimum spanning tree problem as an instance of linear programming. Try to minimize the number of variables and constraints.
- (b) In your MST linear program, what is a basis? What is a feasible basis? What is a locally optimal basis?
- (c) Describe the primal simplex algorithm for your MST linear program directly in terms of the input graph. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?
- (d) Describe the dual simplex algorithm for your MST linear program directly in terms of the input graph. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?
- (e) Which of the classical MST algorithms (Boruvka, Jarník, Kruskal, reverse greedy), if any, are instances of the simplex method? Justify your answer.

*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™ by Mayor Rudy Giuliani, 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.

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

¹What, never? No, NEVER. Anyone you see winning at Three Card Monte is a shill.

²Even 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.

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.

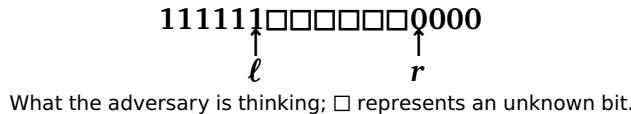
One absolutely crucial feature of this argument is that *the adversary makes absolutely no assumptions about the algorithm*. The adversary strategy can’t depend on some predetermined order of examining bits, and it doesn’t care about anything the algorithm might or might not do when it’s not looking 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 at the bits in every even position: $B[2], B[4], \dots, B[n - 1]$. 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...100...0. 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 $r = n + 1$.



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!)

<u>HIDE01(i):</u>
if $i \leq \ell$
$B[i] \leftarrow 1$
else if $i \geq r$
$B[i] \leftarrow 0$
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 $A[i, j] = \llbracket (i, j) \in E \rrbracket$. 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 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* if 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 topology³—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.

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 $X = \langle x_1, x_2, \dots, x_n \rangle$ of n distinct numbers, find indices i and j such that $x_i = \min X$ and $x_j = \max X$.

³Let Δ be a contractible simplicial complex whose automorphism group $\text{Aut}(\Delta)$ is vertex-transitive, and let Γ be a vertex-transitive subgroup of $\text{Aut}(\Delta)$. If there are normal subgroups $\Gamma_1 \triangleleft \Gamma_2 \triangleleft \Gamma$ 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.

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 $\langle x_1, x_2, \dots, x_n \rangle$ 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. \square

This lemma lets us rephrase the median-finding problem yet again.

Given a sequence $X = \langle x_1, x_2, \dots, x_n \rangle$ 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 $n - k + \lceil \lg \binom{n}{k-1} \rceil = \Omega((n - k) + k \log(n/k))$ comparisons are required to find the k th largest element in an n -element set. This bound is tight up to constant factors for all $k \leq n/2$; 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 $k \leq n/2$ 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 adversary 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 dormitory on campus. In order to determine how much insurance to buy, the university administration needs to determine the highest safe floor in the building. A floor is considered *safe* if a drunk student can fall from a window on that floor and land without breaking; if the egg breaks, the floor is considered *unsafe*. Any floor that is higher than an unsafe floor is also considered unsafe. The only way to determine whether a floor is safe is to drop an egg from a window on that floor.

You would like to find the lowest unsafe floor L by performing as few tests as possible; unfortunately, you have only a very limited supply of eggs.

 - (a) Prove that if you have only one egg, you can find the lowest unsafe floor with L tests. [Hint: Yes, this is trivial.]
 - (b) Prove that if you have only one egg, you must perform at least L tests in the worst case. In other words, prove that your algorithm from part (a) is optimal. [Hint: Use an adversary argument.]
 - (c) Describe an algorithm to find the lowest unsafe floor using two eggs and only $O(\sqrt{L})$ tests. [Hint: Ideally, each egg should be dropped the same number of times. How many floors can you test with n drops?]
 - (d) Prove that if you start with two eggs, you must perform at least $\Omega(\sqrt{L})$ tests in the worst case. In other words, prove that your algorithm from part (c) is optimal.
 - (e) Describe an algorithm to find the lowest unsafe floor using k eggs, using as few tests as possible, and prove your algorithm is optimal for all values of k .

The wonderful thing about standards is that there are so many of them to choose from.

— Real Admiral Grace Murray 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.

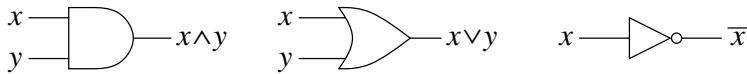
— Shimon Peres

21 NP-Hard Problems

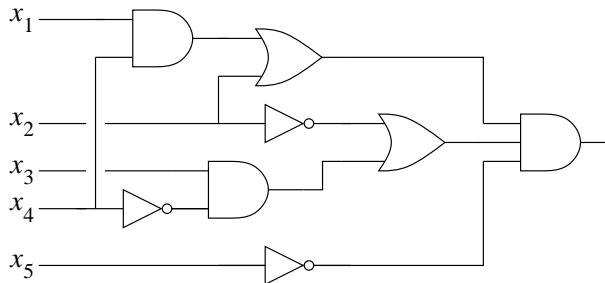
21.1 ‘Efficient’ Problems

A generally-accepted minimum requirement for an algorithm to be considered ‘efficient’ is that its running time is polynomial: $O(n^c)$ for some constant c , where n is the size of the input.¹ Researchers 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. There are several so-called *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, \dots, x_m . The output is a single boolean value. Given specific input values, we can calculate the output of the circuit in polynomial (actually, *linear*) time using depth-first-search, since we can compute the output of a k -input gate in $O(k)$ time.



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.

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!

¹This notion of efficiency was independently formalized by Alan Cobham (*The intrinsic computational difficulty of functions. Logic, Methodology, and Philosophy of Science (Proc. Int. Congress)*, 24–30, 1965), Jack Edmonds (*Paths, trees, and flowers. Canadian Journal of Mathematics* 17:449–467, 1965), and Michael Rabin (*Mathematical theory of automata. Proceedings of the 19th ACM Symposium in Applied Mathematics*, 153–175, 1966), although similar notions were considered more than a decade earlier by Kurt Gödel and John von Neumann.

21.2 P, NP, and co-NP

A *decision problem* is a problem whose output is a single boolean value: YES or No.² Let me define three classes of decision problems:

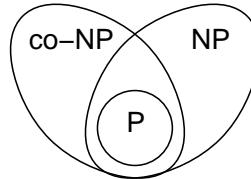
- **P** is the set of decision problems that can be solved in polynomial time.³ Intuitively, P is the set of problems that can be solved quickly.
- **NP** is the set of decision problems with the following property: If the answer is YES, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a YES answer quickly if we have the solution in front of us.
- **co-NP** is the 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.

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. It is widely believed that circuit satisfiability is *not* in P or in co-NP, but nobody actually knows.

Every decision problem in P is also in NP. If a problem is in P, we can verify YES answers in polynomial time recomputing 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 $P \neq NP$; the homeworks and exams in this class and others have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are obvious in retrospect. But nobody knows how to prove it.

A more subtle but still open question is whether NP and co-NP are different. Even if we can verify every YES answer quickly, there's no reason to think that we can also verify No answers quickly. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. It is generally believed that $NP \neq \text{co-NP}$, but nobody knows how to prove it.



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*. In other words:

$$\boxed{\Pi \text{ is NP-hard} \iff \text{If } \Pi \text{ can be solved in polynomial time, then } P=NP}$$

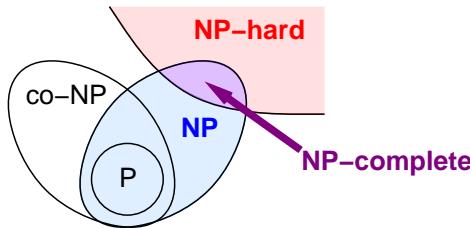
²Technically, I should be talking about *languages*, which are just sets of bit strings. The language associated with a decision problem is the set of bit strings for which the answer is YES. For example, for the problem is 'Is the input graph connected?', the corresponding language is the set of connected graphs, where each graph is represented as a bit string (for example, its adjacency matrix).

³More formally, P is the set of languages that can be recognized in polynomial time by a single-tape Turing machine. If you want to learn more about Turing machines, take CS 579.

Intuitively, 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.⁴

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.

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 (*that is*, all) of them seems incredibly unlikely.



More of what we *think* the world looks like.

It is not immediately clear that *any* decision problems are NP-hard or NP-complete. NP-hardness is already a lot to demand of a problem; insisting that the problem also have a nondeterministic polynomial-time algorithm seems almost completely unreasonable. The following remarkable theorem was first published by Steve Cook in 1971 and independently by Leonid Levin in 1973.⁵ I won’t even sketch the proof, since I’ve been (deliberately) vague about the definitions.

The Cook-Levin Theorem. *Circuit satisfiability is NP-complete.*

⁴More 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*.

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 a function $f : \Sigma^* \rightarrow \Sigma^*$ 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 primarily 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.

⁵Levin submitted his results, and discussed them in talks, several years before they were published. So naturally, in accordance with Stigler’s Law, this result is often called ‘Cook’s Theorem’. Cook won the Turing award for his proof; Levin did not.

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 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.

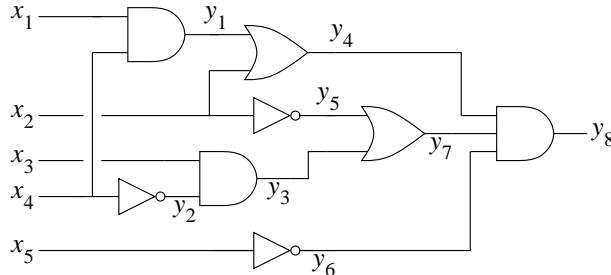
In other words, to prove that your problem is hard, you need to describe an algorithm to solve a *different* problem, which you already know is hard, using a mythical algorithm for *your* problem as a subroutine. The essential logic is a proof by contradiction. Your reduction shows implies that if your 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

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\bar{a} \Rightarrow d)) \vee (c \neq a \wedge b),$$

and the question is whether it is possible to assign boolean values to the variables a, b, c, \dots 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:

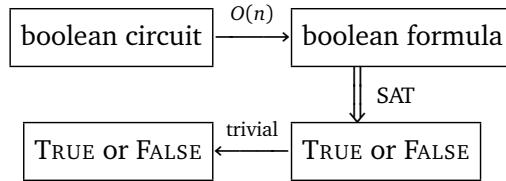


$$\begin{aligned} (y_1 = x_1 \wedge x_4) \wedge (y_2 = \bar{x}_4) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge \\ (y_5 = \bar{x}_2) \wedge (y_6 = \bar{x}_5) \wedge (y_7 = y_3 \vee y_5) \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8 \end{aligned}$$

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 to 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:



$$T_{\text{CSAT}}(n) \leq O(n) + T_{\text{SAT}}(O(n)) \implies T_{\text{SAT}}(n) \geq T_{\text{CSAT}}(\Omega(n)) - O(n)$$

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 *literal*s, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})}^{\text{clause}}$$

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:

$$\begin{aligned} a = b \wedge c &\longrightarrow (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\longrightarrow (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\longrightarrow (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

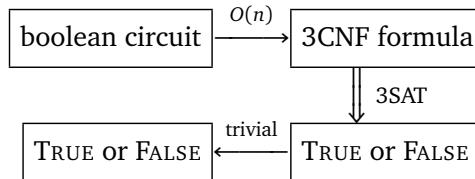
4. *Make sure every clause has exactly three literals.* Introduce new variables into each one- and two-literal clause, and expand it into two clauses as follows:

$$\begin{aligned} a &\longrightarrow (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \\ a \vee b &\longrightarrow (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \end{aligned}$$

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^{573}) of the circuit size, we would still have a valid reduction.

$$\begin{aligned}
 & (y_1 \vee \bar{x}_1 \vee \bar{x}_4) \wedge (\bar{y}_1 \vee x_1 \vee z_1) \wedge (\bar{y}_1 \vee x_1 \vee \bar{z}_1) \wedge (\bar{y}_1 \vee x_4 \vee z_2) \wedge (\bar{y}_1 \vee x_4 \vee \bar{z}_2) \\
 & \wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \bar{z}_3) \wedge (\bar{y}_2 \vee \bar{x}_4 \vee z_4) \wedge (\bar{y}_2 \vee \bar{x}_4 \vee \bar{z}_4) \\
 & \wedge (y_3 \vee \bar{x}_3 \vee \bar{y}_2) \wedge (\bar{y}_3 \vee x_3 \vee z_5) \wedge (\bar{y}_3 \vee x_3 \vee \bar{z}_5) \wedge (\bar{y}_3 \vee y_2 \vee z_6) \wedge (\bar{y}_3 \vee y_2 \vee \bar{z}_6) \\
 & \wedge (\bar{y}_4 \vee y_1 \vee x_2) \wedge (y_4 \vee \bar{x}_2 \vee z_7) \wedge (y_4 \vee \bar{x}_2 \vee \bar{z}_7) \wedge (y_4 \vee \bar{y}_1 \vee z_8) \wedge (y_4 \vee \bar{y}_1 \vee \bar{z}_8) \\
 & \wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \bar{z}_9) \wedge (\bar{y}_5 \vee \bar{x}_2 \vee z_{10}) \wedge (\bar{y}_5 \vee \bar{x}_2 \vee \bar{z}_{10}) \\
 & \wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \bar{z}_{11}) \wedge (\bar{y}_6 \vee \bar{x}_5 \vee z_{12}) \wedge (\bar{y}_6 \vee \bar{x}_5 \vee \bar{z}_{12}) \\
 & \wedge (\bar{y}_7 \vee y_3 \vee y_5) \wedge (y_7 \vee \bar{y}_3 \vee z_{13}) \wedge (y_7 \vee \bar{y}_3 \vee \bar{z}_{13}) \wedge (y_7 \vee \bar{y}_5 \vee z_{14}) \wedge (y_7 \vee \bar{y}_5 \vee \bar{z}_{14}) \\
 & \wedge (y_8 \vee \bar{y}_4 \vee \bar{y}_7) \wedge (\bar{y}_8 \vee y_4 \vee z_{15}) \wedge (\bar{y}_8 \vee y_4 \vee \bar{z}_{15}) \wedge (\bar{y}_8 \vee y_7 \vee z_{16}) \wedge (\bar{y}_8 \vee y_7 \vee \bar{z}_{16}) \\
 & \wedge (y_9 \vee \bar{y}_8 \vee \bar{y}_6) \wedge (\bar{y}_9 \vee y_8 \vee z_{17}) \wedge (\bar{y}_9 \vee y_8 \vee \bar{z}_{17}) \wedge (\bar{y}_9 \vee y_6 \vee z_{18}) \wedge (\bar{y}_9 \vee y_6 \vee \bar{z}_{18}) \\
 & \wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \bar{z}_{19} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \bar{z}_{20}) \wedge (y_9 \vee \bar{z}_{19} \vee \bar{z}_{20})
 \end{aligned}$$

This process transforms the circuit into an equivalent 3CNF formula; the output formula is satisfiable if and only if the input circuit is satisfiable. As with the more general SAT problem, the formula is only a constant factor larger than 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:



$$T_{\text{CSAT}}(n) \leq O(n) + T_{\text{3SAT}}(O(n)) \implies T_{\text{3SAT}}(n) \geq T_{\text{CSAT}}(\Omega(n)) - O(n)$$

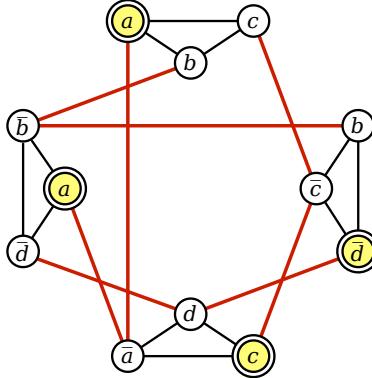
We conclude 3SAT is NP-hard. And because 3SAT is a special case of SAT, it is also in NP. Therefore, 3SAT is NP-complete.

21.6 Maximum Independent (from 3SAT)

For the next few problems we consider, the input is a simple, unweighted graph, and the problem asks for the size of the largest or smallest subgraph satisfying some structural property.

Let G be an arbitrary graph. An *independent set* in G is a subset of the vertices of G with no edges between them. The *maximum independent set* problem, or simply **MAXINDSET**, asks for the size of the largest independent set in a given graph.

I'll prove that **MAXINDSET** is NP-hard (but not NP-complete, since it isn't a decision problem) using a reduction from 3SAT. I'll describe a reduction from a 3CNF formula into a graph that has an independent set 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 the same clause, or (2) they correspond to a variable and its inverse. 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.

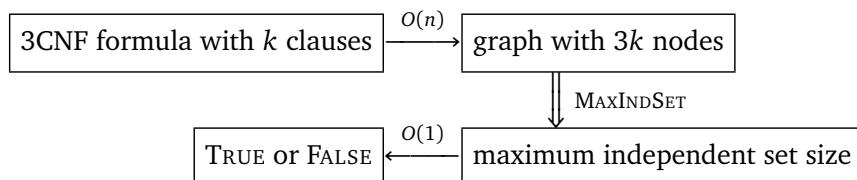


A graph derived from a 3CNF formula, and an independent set of size 4.
Black edges join literals from the same clause; red (heavier) edges join contradictory literals.

Now suppose the original formula had k clauses. Then I claim that the formula is satisfiable if and only if the graph has an independent set of size k .

1. **independent set \Rightarrow satisfying assignment:** If the graph has an independent set of k vertices, then each vertex must come from a different clause. To obtain a satisfying assignment, we assign the value TRUE to each literal in the independent set. Since contradictory literals are connected by edges, this assignment is consistent. There may be variables that have no literal in the independent set; we can set these to any value we like. The resulting assignment satisfies the original 3CNF formula.
2. **satisfying assignment \Rightarrow independent set:** If we have a satisfying assignment, then we can choose one literal in each clause that is TRUE. Those literals form an independent set in the graph.

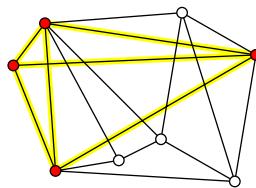
Thus, the reduction is correct. Since the reduction from 3CNF formula to graph takes polynomial time, we conclude that MAXINDSET is NP-hard. Here's a diagram of the reduction:



$$T_{\text{3SAT}}(n) \leq O(n) + T_{\text{MAXINDSET}}(O(n)) \implies T_{\text{MAXINDSET}}(n) \geq T_{\text{3SAT}}(\Omega(n)) - O(n)$$

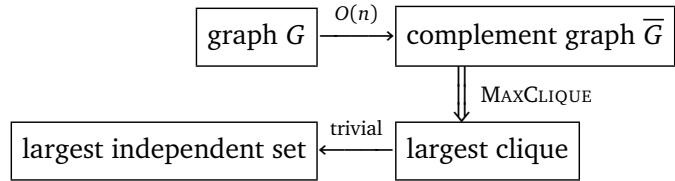
21.7 Clique (from Independent Set)

A *clique* is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. 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.

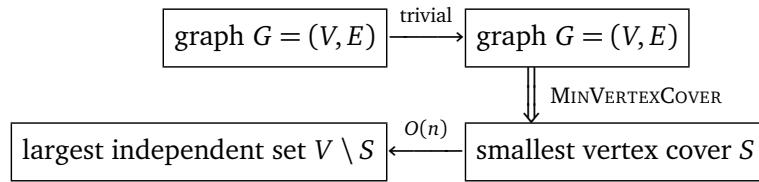
There is an easy proof that MAXCLIQUE is NP-hard, using a reduction from MAXINDSET. Any graph G has an *edge-complement* \bar{G} with the same vertices, but with exactly the opposite set of edges— (u, v) is an edge in \bar{G} if and only if it is *not* an edge in G . A set of vertices is independent in G if and only if the same vertices define a clique in \bar{G} . Thus, we can compute the largest independent in a graph simply by computing the largest clique in the complement of the graph.



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 MINVERTEXCOVER 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.



21.9 Graph Coloring (from 3SAT)

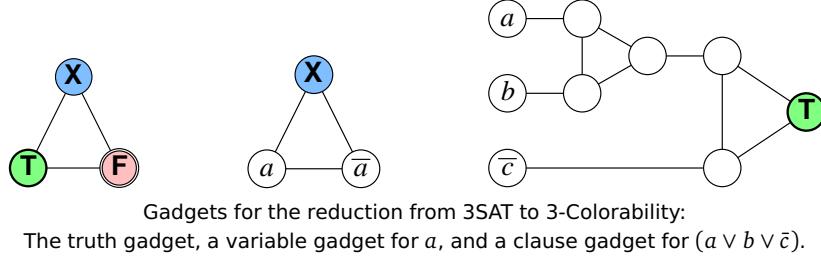
A k -coloring of a graph is a map $C: V \rightarrow \{1, 2, \dots, k\}$ that assigns one of k ‘colors’ to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring. To show that this problem 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.

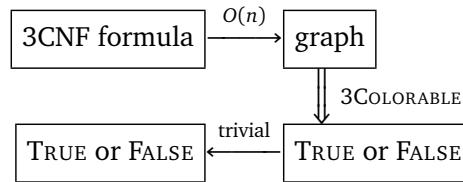
The truth gadget is just a triangle with three vertices T , F , and X , which intuitively stand for TRUE, FALSE, and OTHER. Since these vertices are all connected, they must have different colors in any 3-coloring. For the sake of convenience, we will *name* those colors TRUE, FALSE, and OTHER. Thus, 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 \bar{a} to node X in the truth gadget. Node a must be colored either TRUE or FALSE, and so node \bar{a} must be colored either FALSE or TRUE, respectively.

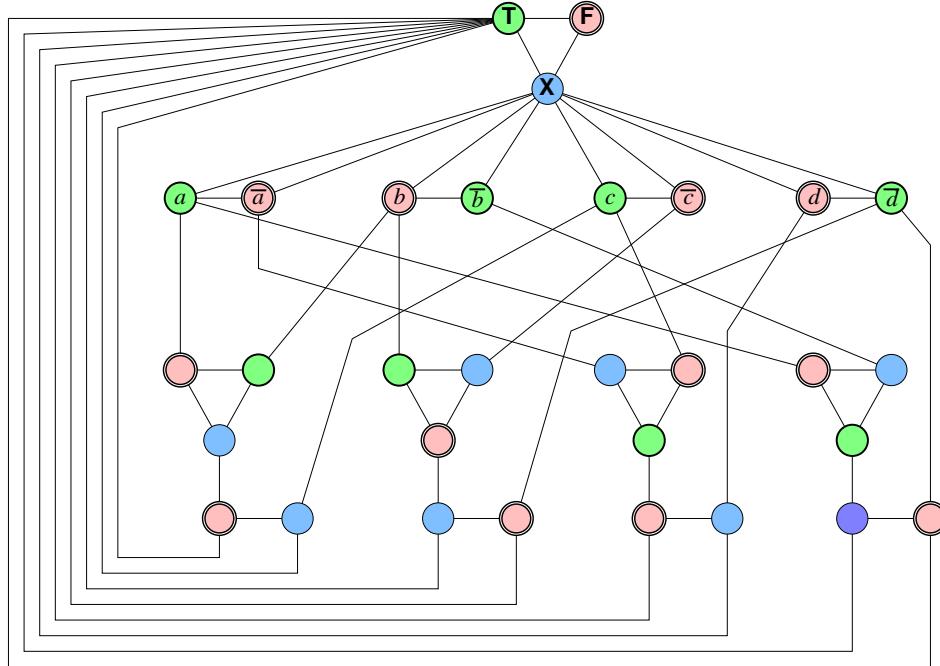
Finally, each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabeled nodes and ten edges; see the figure below. If all three literal nodes in the clause gadget are colored FALSE, then the rightmost vertex in the gadget cannot have one of the three colors. Since 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.



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.



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 $a = c = \text{TRUE}$, $b = d = \text{FALSE}$.



A 3-colorable graph derived from a satisfiable 3CNF formula.

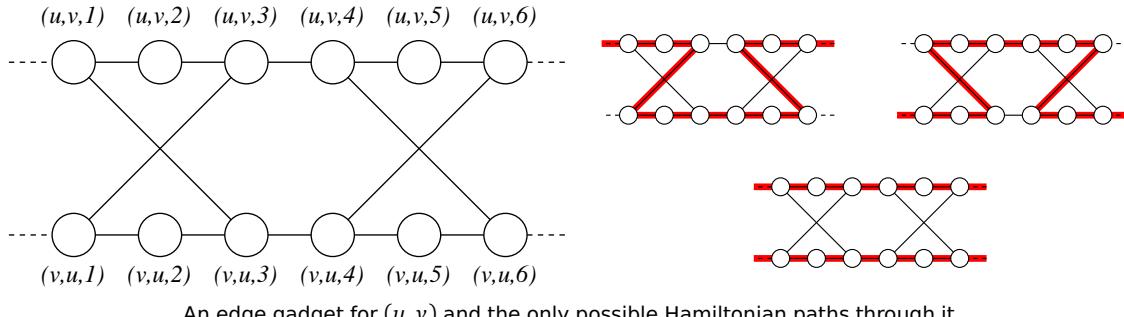
We can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge. Thus, 3COLORING is in NP, and therefore NP-complete. Moreover, since 3COLORING is a special case of the more general graph coloring problem—What is the minimum number of colors?—the more problem is also NP-hard, but *not* NP-complete, because it's not a decision problem.

21.10 Hamiltonian Cycle (from Vertex Cover)

A *Hamiltonian cycle* in 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.

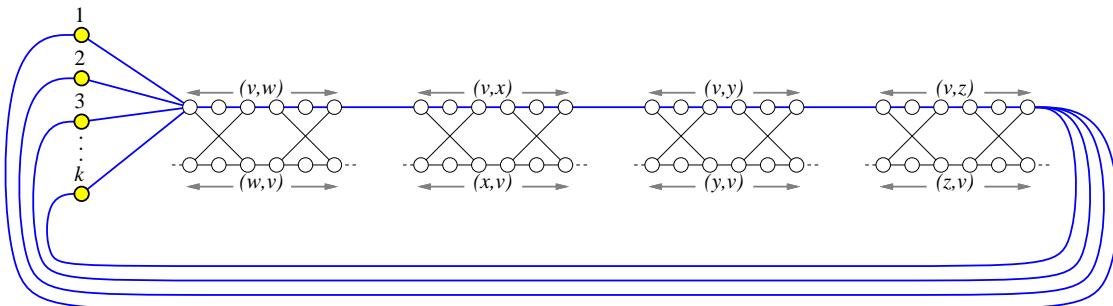
To prove this, we use a reduction from the vertex cover problem. Given a graph G and an integer k , we need to transform it into another graph G' , such that G' has a Hamiltonian cycle if and only if G has a vertex cover of size k . As usual, our transformation uses several gadgets.

- 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.



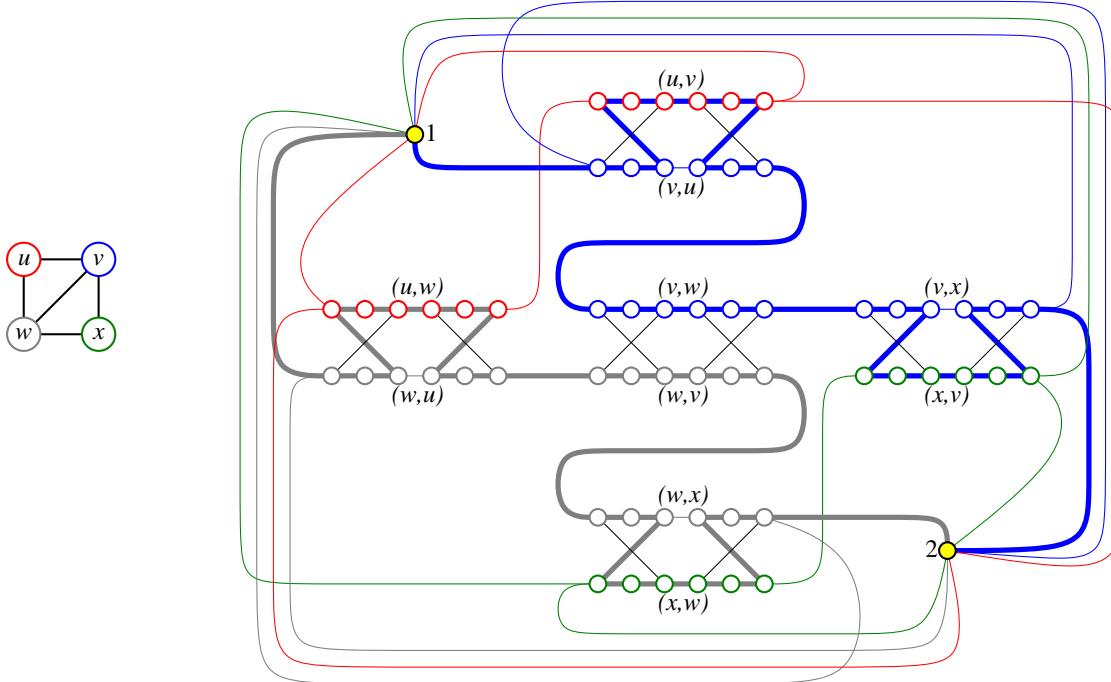
An edge gadget for (u, v) and the only possible Hamiltonian paths through it.

- 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, \dots, 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 $\{v_1, v_2, \dots, v_k\}$ 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 is 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 a 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 as an endpoint. Alternately, we can think of each integer in X as an $(m + 1)$ -digit number written in base 4. The m th digit is 1 if the integer represents a vertex, and 0 otherwise. For each $i < m$, the i th 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 i th digit is 1. Each edge number b_i contributes only one 1 to the i th digit of the sum, but the i th 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:

$$\begin{aligned} b_{uv} &:= 010000_4 = 256 \\ b_{uw} &:= 001000_4 = 64 \\ b_{vw} &:= 000100_4 = 16 \\ b_{vx} &:= 000010_4 = 4 \\ b_{wx} &:= 000001_4 = 1 \\ \\ a_u &:= 111000_4 = 1344 \\ a_v &:= 110110_4 = 1300 \\ a_w &:= 101101_4 = 1105 \\ a_x &:= 100011_4 = 1029 \end{aligned}$$

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*.⁶

- 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 easy⁷ 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 proved NP-hard by reduction from PLANARCIRCUITSAT.⁸
- EXACT3DIMENSIONALMATCHING or X3M: Given a set S and a collection of three-element subsets of S , called *triples*, is there a sub-collection 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 $A \cup B = S$, $A \cap B = \emptyset$, 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 $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$, find the smallest sub-collection 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 $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$, 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.
- 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.

⁶Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.

⁷or at least *nondeterministically* easy

⁸Surprisingly, PLANARNOTALLEQUAL3SAT is solvable in polynomial time!

- 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)⁹, Tetris (by reduction from 3PARTITION)¹⁰, and Sudoku (by reduction from 3SAT)¹¹. Most two-player games¹² 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.¹³

Exercises

1. 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?
2. (a) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.
 (b) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.
3. (a) Describe and analyze an 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.

⁹Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. <http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf>

¹⁰Ron 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.

¹¹Takayuki Yato and Takahiro Seta. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A(5):1052–1060, 2003. <http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf>.

¹²For 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>.

¹³PSPACE 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 $PSPACE=NP$.

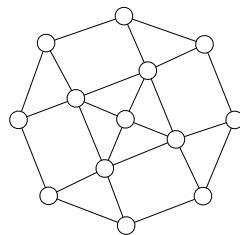
EXP (also called EXPTIME) is the set of decision problems that can be solved in exponential time: at most 2^{n^c} for some $c > 0$. Every problem in PSPACE (and therefore in NP (and therefore in P)) is also in EXP. It is generally believed that $PSPACE \subsetneq EXP$, but nobody can even prove that $NP \neq EXP$. 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!

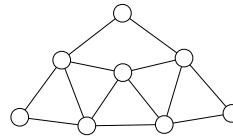
- (b) Why doesn't this algorithm imply that P=NP?
4. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (OR) or several *terms*, each of which is the conjunction (AND) of one or more literals. For example, the formula
- $$(\bar{a} \wedge b \wedge \bar{c}) \vee (b \wedge c) \vee (a \wedge \bar{b} \wedge \bar{c})$$
- is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.
- (a) Show that DNF-SAT is in P.
 - (b) What is the error in the following argument that P=NP?
- Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,
- $$(a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b}) \iff (a \wedge \bar{b}) \vee (b \wedge \bar{a}) \vee (\bar{c} \wedge \bar{a}) \vee (\bar{c} \wedge \bar{b})$$
- Now we can use the algorithms from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time! Since 3SAT is NP-hard, we must conclude that P=NP.
5. (a) 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.
- (b) 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.
- (c) 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.
6. (a) Prove that PLANARCIRCUITSAT is NP-complete.
- (b) Prove that NOTALLEQUAL3SAT is NP-complete.
- (c) Prove that the following variant of 3SAT is NP-complete: Given a formula ϕ in conjunctive normal form where each clause contains at most 3 literals and each variable appears in at most 3 clauses, is ϕ satisfiable?
7. The problem 12COLOR is defined as follows: Given a graph, can we color each vertex with one of twelve colors, so that every edge touches two different colors? Prove that 12COLOR is NP-hard.
8. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire that lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of “strongly favor”, “mostly neutral”, or “strongly oppose”. Each student may respond with “strongly favor” or “strongly oppose” to at most five questions. Because Jeff's students are very understanding, each student is happy if (but only if) he

or she prevails in just one of his or her strong policy preferences. Either describe a polynomial-time algorithm for setting course policy to maximize the number of happy students, or show that the problem is NP-hard.

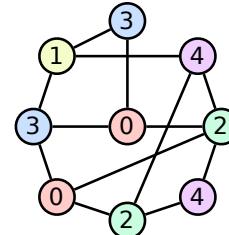
9. (a) Using the gadget on the right below, prove that deciding whether a given *planar* graph is 3-colorable is NP-complete. [Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]
- (b) Using part (a) and the middle gadget below, 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.]



(a) Gadget for planar 3-colorability.



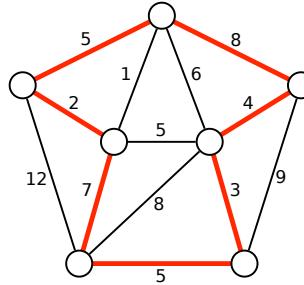
(b) Gadget for degree-4 planar 3-colorability.



(c) A careful 5-coloring.

10. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G an ‘color’ from the set $\{0, 1, 2, 3, 4\}$, such that for any edge uv , vertices u and v are assigned different ‘colors’. A 5-coloring is *careful* if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 ($\bmod 5$). Prove that deciding whether a given graph has a careful 5-coloring is NP-complete. [Hint: Reduce from the standard 5COLOR problem.]
11. 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?
12. The RECTANGLETILING problem asks, given a ‘large’ rectangle R and several ‘small’ rectangles r_1, r_2, \dots, r_n , whether the small rectangles can be placed inside the larger rectangle with no gaps or overlaps. Prove that RECTANGLETILING is NP-complete.
13. (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).]
14. For each problem below, either describe a polynomial-time algorithm or prove that the problem is NP-complete.

- (a) A *double-Eulerian circuit* in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G , does G have a double-Eulerian circuit?
- (b) A *double-Hamiltonian circuit* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G , does G have a double-Hamiltonian circuit?
15. A boolean formula in *exclusive-or conjunctive normal form* (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive-or* of several literals; that is, a clause is true if and only if it contains an odd number of true literals. The XCNF-SAT problem asks whether a given XCNF formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-hard.
16. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G . Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.



A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

17. *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 there is a sequence of pebbling moves that removes all but one pebble. Prove that PEBBLEDESTRUCTION is NP-complete.
18. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbletypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbletypeg (fixed during the Holy Roman Three-Way Mumbletypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of party-goers know each other and ask you to choose the teams, while he sharpens the knife.

Either describe and analyze a polynomial time algorithm to determine whether the party-goers can be split into three legal Three-Way Mumbletypeg teams, or prove that the problem is NP-hard.

19. (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.*]

Le mieux est l'ennemi du bien. [The best is the enemy of the good.]

— Voltaire, *La Béguerie* (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)

Now, distrust of corporations threatens our still-tentative economic recovery; it turns out greed is bad, after all.

— Paul Krugman, "Greed is Bad", *The New York Times*, June 4, 2002.

K Approximation Algorithms

K.1 Load Balancing

On the future smash hit reality-TV game show *Grunt Work*, scheduled to air Thursday nights at 3am (2am Central) on ESPN π , 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.*

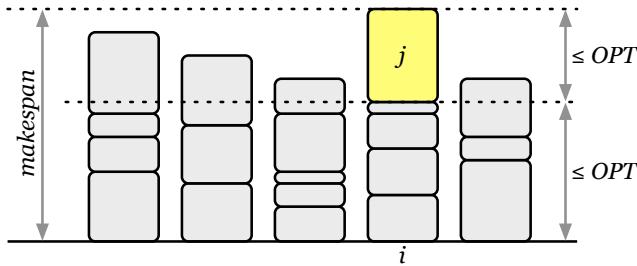
Proof: Fix an arbitrary input, and let OPT denote the makespan of its optimal assignment. The approximation bound follows from two trivial observations. First, the makespan of any assignment (and therefore of the optimal assignment) is at least the duration of the longest job. Second, the makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

$$OPT \geq \max_j T[j] \quad \text{and} \quad OPT \geq \frac{1}{m} \sum_{j=1}^n T[j]$$

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 $Total[i] - T[j] \leq OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $Total[i] - T[j] \leq Total[k]$ for all k . (Some values $Total[k]$ may have increased since job j was assigned, but that only helps us.) In particular, $Total[i] - T[j]$ is less than or equal to the average finishing time over all machines. Thus,

$$Total[i] - T[j] \leq \frac{1}{m} \sum_{i=1}^m Total[i] = \frac{1}{m} \sum_{j=1}^n T[j] \leq OPT$$

by our second trivial observation. We conclude that the makespan $Total[i]$ is at most $2 \cdot OPT$. \square



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. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

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 $\text{Total}[i] - T[j] \leq OPT$.

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 $\text{Total}[i]$ is at most $3 \cdot OPT/2$, as claimed. \square

K.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 **$\alpha(n)$ -approximation algorithm** if and only if

$$\frac{OPT(X)}{A(X)} \leq \alpha(n) \quad \text{and} \quad \frac{A(X)}{OPT(X)} \leq \alpha(n)$$

for all inputs X of size n . The function $\alpha(n)$ is called the **approximation factor** for algorithm A . For any given algorithm, 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.

Especially for problems where exact optimization is NP-hard, we have little hope of completely characterizing the optimal solution. The secret to proving that an algorithm satisfies some approximation ratio is to find a useful function of the input that provides both lower bounds on the cost of the optimal solution and upper bounds on the cost of the approximate solution. For example, if $OPT(X) \geq f(X)/2$ and $A(X) \leq 5f(X)$, for any function f , then A is a 10-approximation algorithm. Finding the right intermediate function can be a delicate balancing act.

K.3 Greedy Vertex Cover

Recall that the *vertex cover* 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 heuristic¹ for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

```

GREEDYVERTEXCOVER( $G$ ):
   $C \leftarrow \emptyset$ 
  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  $C$ 

```

¹A *heuristic* is an algorithm that doesn't work.

Obviously this algorithm doesn't compute the optimal vertex cover—that would imply P=NP!—but it does compute a reasonably close approximation.

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( $G$ ):
   $C \leftarrow \emptyset$ 
   $G_0 \leftarrow G$ 
   $i \leftarrow 0$ 
  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 $O(OPT \log n)$. \square

So far we've only proved an *upper bound* on the approximation factor of GREEDYVERTEXCOVER; perhaps a more careful analysis would imply that the approximation factor is only $O(\log \log n)$, or even $O(1)$. Alas, no such improvement is possible. For any integer n , a simple recursive construction gives us an n -vertex graph for which the greedy algorithm returns a vertex cover of size $\Omega(OPT \cdot \log n)$. Details are left as an exercise for the reader.

K.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 .² 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( $X, \mathcal{F}$ ):
     $\mathcal{C} \leftarrow \emptyset$ 
    while  $X \neq \emptyset$ 
         $S \leftarrow \arg \max_{S \in \mathcal{F}} |S \cap X|$ 
         $X \leftarrow X \setminus S$ 
         $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$ 
    return  $\mathcal{C}$ 
```

```
GREEDYHITTINGSET( $X, \mathcal{F}$ ):
     $H \leftarrow \emptyset$ 
    while  $\mathcal{F} \neq \emptyset$ 
         $x \leftarrow \arg \max_{x \in X} |\{S \in \mathcal{F} \mid x \in S\}|$ 
         $\mathcal{F} \leftarrow \mathcal{F} \setminus \{S \in \mathcal{F} \mid x \in S\}$ 
         $H \leftarrow H \cup \{x\}$ 
    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^* :

$$X^* = \{x^* \mid x \in X\}.$$

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 573 this semester. Then X^* has 26 elements, each containing the subset of CS 573 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^*) .

K.5 Vertex Cover, Again

The greedy approach doesn't always lead to the best approximation algorithms. Consider the following alternate heuristic for vertex cover:

²A matroid (see the lecture note on greedy algorithms) is a special type of set system.

```

DUMBVERTEXCOVER( $G$ ):
   $C \leftarrow \emptyset$ 
  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} .

K.6 Traveling Salesman: The Bad News

The *traveling salesman problem*³ 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 $P=NP$.*

³This is sometimes bowdlerized into the traveling salesperson problem. That's just silly. Who ever heard of a traveling salesperson sleeping with the farmer's child?

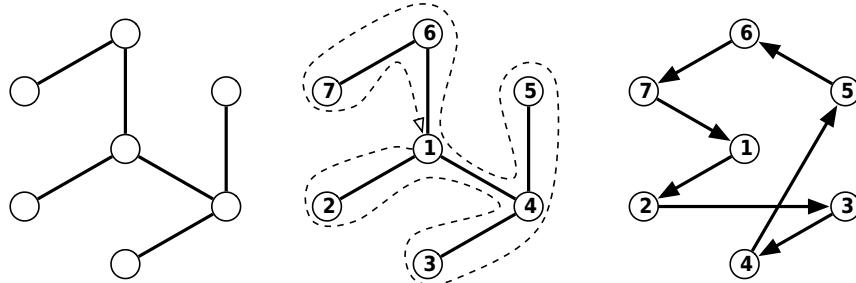
K.7 Traveling Salesman: The Good News

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*:

$$\ell(u, w) \leq \ell(u, v) + \ell(v, w) \quad \text{for any vertices } u, v, w.$$

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) 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.

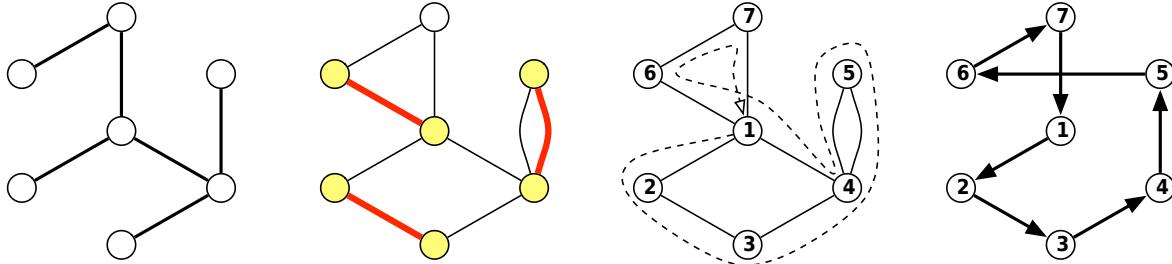
Let OPT denote the cost of the optimal TSP tour, let MST denote the total length of the minimum spanning tree, and let A be the length of the tour computed by our approximation algorithm. Consider the ‘tour’ obtained by walking through the minimum spanning tree in depth-first order. Since this tour traverses 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 this approximation factor 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 perfect 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. \square

K.8 k -center Clustering

The k -center clustering problem is defined as follows. We are given a set $P = \{p_1, p_2, \dots, p_n\}$ of n points in the plane⁴ and an integer k . Our goal to find a collection of k circles that collectively enclose all the input points, such that the radius of the largest circle is as large as possible. More formally, we want to compute a set $C = \{c_1, c_2, \dots, c_k\}$ of k center points, such that the following cost function is minimized:

$$\text{cost}(C) := \max_i \min_j |p_i c_j|.$$

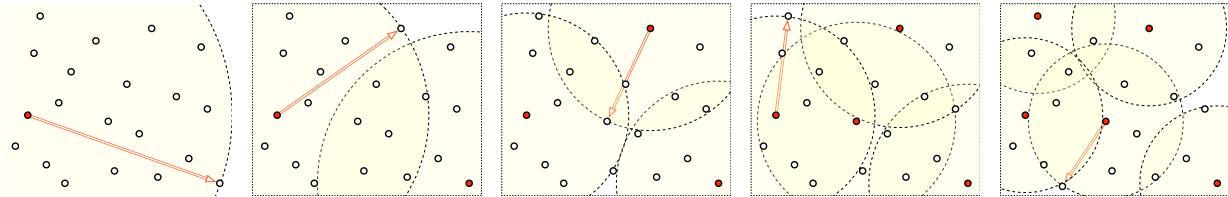
Here, $|p_i c_j|$ denotes the Euclidean distance between input point p_i and center point c_j . Intuitively, each input point is assigned to its closest center point; the points assigned to a given center c_j comprise a *cluster*. The distance from c_j to the farthest point in its cluster is the *radius* of that cluster; the cluster is contained in a circle of this radius centered at c_j . The k -center clustering cost $\text{cost}(C)$ is precisely the maximum cluster radius.

This problem turns out to be NP-hard, even to approximate within a factor of roughly 1.8. However, there is a natural greedy strategy, first analyzed in 1985 by Teofilo Gonzalez⁵, that is guaranteed to

⁴The k -center problem can be defined over *any* metric space, and the approximation analysis in this section holds in any metric space as well. The analysis in the next section, however, does require that the points come from the Euclidean plane.

⁵Teofilo F. Gonzalez. Clustering to minimize the maximum inter-cluster distance. *Theoretical Computer Science* 38:293–306, 1985.

produce a clustering whose cost is at most twice optimal. Choose the k center points one at a time, starting with an arbitrary input point as the first center. In each iteration, choose the input point that is farthest from any earlier center point to be the next center point.



The first five iterations of Gonzalez's k -center clustering algorithm.

In the pseudocode below, d_i denotes the current distance from point p_i to its nearest center, and r_j denotes the maximum of all d_i (or in other words, the cluster radius) after the first j centers have been chosen. The algorithm includes an extra iteration to compute the final clustering cost r_k .

```

GONZALEZKCENTER( $P, k$ ):
  for  $i \leftarrow 1$  to  $n$ 
     $d_i \leftarrow \infty$ 
   $c_1 \leftarrow p_1$ 
  for  $j \leftarrow 1$  to  $k$ 
     $r_j \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $n$ 
       $d_i \leftarrow \min\{d_i, |p_i c_j|\}$ 
      if  $r_j < d_i$ 
         $r_j \leftarrow d_i$ ;  $c_{j+1} \leftarrow p_i$ 
  return  $\{c_1, c_2, \dots, c_k\}$ 

```

GONZALEZKCENTER clearly runs in $O(nk)$ time. Using more advanced data structures, Tomas Feder and Daniel Greene⁶ described an algorithm to compute exactly the same clustering in only $O(n \log k)$ time.

Theorem 6. GONZALEZKCENTER computes a 2-approximation to the optimal k -center clustering.

Proof: Let r^* be the optimal k -center clustering radius for some point set P , and let r be the clustering radius computed by GONZALEZKCENTER. Suppose for purposes of proving a contradiction that $r > 2r^*$. Then among the first $k + 1$ center points selected by GONZALEZKCENTER, every pair has distance at least $r \geq 2r^*$. Thus, by the triangle inequality, any ball of radius r^* contains at most one of these $k + 1$ center points. But this implies that at least $k + 1$ balls of radius r^* are required to cover P , which contradicts the definition of r^* . \square

*K.9 Approximation Schemes

With just a little more work, we can compute an arbitrarily close approximation of the optimal k -clustering, using a so-called *approximation scheme*. An approximation scheme accepts both an instance of the problem and a value $\varepsilon > 0$ as input, and it computes a $(1 + \varepsilon)$ -approximation of the optimal

⁶Tomas Feder* and Daniel H. Greene. Optimal algorithms for approximate clustering. *Proc. 20th STOC*, 1988. Unlike Gonzalez's algorithm, Feder and Greene's faster algorithm does not work over arbitrary metric spaces; it requires that the input points come from some \mathbb{R}^d and that distances are measured in some L_p metric. The time analysis also assumes that the distance between any two points can be computed in $O(1)$ time.

output for that instance. As I mentioned earlier, computing even a 1.8-approximation is NP-hard, so we cannot expect our approximation scheme to run in polynomial time; nevertheless, at least for small values of k , the approximation scheme will be considerably more efficient than any exact algorithm.

Our approximation scheme works in three phases:

1. Compute a 2-approximate clustering of the input set P using GONZALEZKCENTER. Let r be the cost of this clustering.
2. Create a regular grid of squares of width $\delta = \varepsilon r / 2\sqrt{2}$. Let Q be a subset of P containing one point from each non-empty cell of this grid.
3. Compute an *optimal* set of k centers for Q . Return these k centers as the approximate k -center clustering for P .

The first phase requires $O(nk)$ time. By our earlier analysis, we have $r^* \leq r \leq 2r^*$, where r^* is the optimal k -center clustering cost for P .

The second phase can be implemented in $O(n)$ time using a hash table, or in $O(n \log n)$ time by standard sorting, by associating approximate coordinates $(\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor)$ to each point $(x, y) \in P$ and removing duplicates. The key observation is that the resulting point set Q is significantly smaller than P . We know P can be covered by k balls of radius r^* , each of which touches $O(r^*/\delta^2) = O(1/\varepsilon^2)$ grid cells. It follows that $|Q| = O(k/\varepsilon^2)$.

Let $T(n, k)$ be the running time of an *exact* k -center clustering algorithm, given n points as input. If this were a computational geometry class, we might see a “brute force” algorithm that runs in time $T(n, k) = O(n^{k+2})$; the fastest algorithm currently known⁷ runs in time $T(n, k) = n^{O(\sqrt{k})}$. If we use this algorithm, our third phase requires $(k/\varepsilon^2)^{O(\sqrt{k})}$ time.

It remains to show that the optimal clustering for Q implies a $(1 + \varepsilon)$ -approximation of the optimal clustering for P . Suppose the optimal clustering of Q consists of k balls B_1, B_2, \dots, B_k , each of radius \tilde{r} . Clearly $\tilde{r} \leq r^*$, since any set of k balls that cover P also cover any subset of P . Each point in $P \setminus Q$ shares a grid cell with some point in Q , and therefore is within distance $\delta\sqrt{2}$ of some point in Q . Thus, if we increase the radius of each ball B_i by $\delta\sqrt{2}$, the expanded balls must contain every point in P . We conclude that the optimal centers for Q gives us a k -center clustering for P of cost at most $r^* + \delta\sqrt{2} \leq r^* + \varepsilon r / 2 \leq r^* + \varepsilon r^* = (1 + \varepsilon)r^*$.

The total running time of the approximation scheme is $O(nk + (k/\varepsilon^2)^{O(\sqrt{k})})$. This is still exponential in the input size if k is large (say \sqrt{n} or $n/100$), but if k and ε are fixed constants, the running time is linear in the number of input points.

*K.10 An FPTAS for Subset Sum

An approximation scheme whose running time, for any fixed ε , is polynomial in n is called a *polynomial-time approximation scheme* or PTAS (usually pronounced “pee taz”). If in addition the running time depends only polynomially on ε , the algorithm is called a *fully polynomial-time approximation scheme* or FPTAS (usually pronounced “eff pee taz”). For example, an approximation scheme with running time $O(n^2/\varepsilon^2)$ is an FPTAS; an approximation scheme with running time $O(n^{1/\varepsilon^6})$ is a PTAS but not an FPTAS; and our approximation scheme for k -center clustering is not a PTAS.

The last problem we’ll consider is the SUBSETSUM problem: Given a set X containing n positive integers and a target integer t , determine whether X has a subset whose elements sum to t . The lecture notes on NP-completeness include a proof that SUBSETSUM is NP-hard. As stated, this problem doesn’t

⁷R. Z. Hwang, R. C. T. Lee, and R. C. Chan. The slab dividing approach to solve the Euclidean p -center problem. *Algorithmica* 9(1):1–22, 1993.

allow any sort of approximation—the answer is either TRUE or FALSE.⁸ So we will consider a related optimization problem instead: Given set X and integer t , find the subset of X whose sum is as large as possible but no larger than t .

We have already seen a dynamic programming algorithm to solve the decision version SUBSETSUM in time $O(nt)$; a similar algorithm solves the optimization version in the same time bound. Here is a different algorithm, whose running time does not depend on t :

```
SUBSETSUM( $X[1..n], t$ ):
   $S_0 \leftarrow \{0\}$ 
  for  $i \leftarrow 1$  to  $n$ 
     $S_i \leftarrow S_{i-1} \cup (S_{i-1} + X[i])$ 
    remove all elements of  $S_i$  bigger than  $t$ 
  return  $\max S_n$ 
```

Here $S_{i-1} + X[i]$ denotes the set $\{s + X[i] \mid s \in S_{i-1}\}$. If we store each S_i in a sorted array, the i th iteration of the for-loop requires time $O(|S_{i-1}|)$. Each set S_i contains all possible subset sums for the first i elements of X ; thus, S_i has at most 2^i elements. On the other hand, since every element of S_i is an integer between 0 and t , we also have $|S_i| \leq t + 1$. It follows that the total running time of this algorithm is $\sum_{i=1}^n O(|S_{i-1}|) = O(\min\{2^n, nt\})$.

Of course, this is only an estimate of worst-case behavior. If several subsets of X have the same sum, the sets S_i will have fewer elements, and the algorithm will be faster. The key idea for finding an approximate solution quickly is to ‘merge’ nearby elements of S_i —if two subset sums are nearly equal, ignore one of them. On the one hand, merging similar subset sums will introduce some error into the output, but hopefully not too much. On the other hand, by reducing the size of the set of sums we need to maintain, we will make the algorithm faster, hopefully significantly so.

Here is our approximation algorithm. We make only two changes to the exact algorithm: an initial sorting phase and an extra FILTERING step inside the main loop.

FILTER($Z[1..k], \delta$): <pre> SORT(Z) j ← 1 Y[j] ← Z[i] for i ← 2 to k if $Z[i] > (1 + \delta) \cdot Y[j]$ j ← j + 1 Y[j] ← Z[i] return Y[1..j]</pre>	APPROXSUBSETSUM($X[1..n], k, \varepsilon$): <pre> SORT(X) R₀ ← {0} for i ← 1 to n R_i ← R_{i-1} ∪ (R_{i-1} + X[i]) R_i ← FILTER(R_i, ε/2n) remove all elements of R_i bigger than t return max R_n</pre>
--	---

Theorem 7. APPROXSUBSETSUM returns a $(1 + \varepsilon)$ -approximation of the optimal subset sum, given any ε such that $0 < \varepsilon \leq 1$.

Proof: The theorem follows from the following claim, which we prove by induction:

For any element $s \in S_i$, there is an element $r \in R_i$ such that $r \leq s \leq r \cdot (1 + \varepsilon n/2)^i$.

The claim is trivial for $i = 0$. Let s be an arbitrary element of S_i , for some $i > 0$. There are two cases to consider: either $x \in S_{i-1}$, or $x \in S_{i-1} + x_i$.

⁸Do, or do not. There is no ‘try’. (Are old one thousand when years you, alphabetical also in order talk will you.)

- (1) Suppose $s \in S_{i-1}$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' \in R_i$, the claim obviously holds. On the other hand, if $r' \notin R_i$, there must be an element $r \in R_i$ such that $r < r' \leq r(1 + \varepsilon n/2)$, which implies that

$$r < r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1} \leq r \cdot (1 + \varepsilon n/2)^i,$$

so the claim holds.

- (2) Suppose $s \in S_{i-1} + x_i$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s - x_i \leq r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' + x_i \in R_i$, the claim obviously holds. On the other hand, if $r' + x_i \notin R_i$, there must be an element $r \in R_i$ such that $r < r' + x_i \leq r(1 + \varepsilon n/2)$, which implies that

$$\begin{aligned} r < r' + x_i &\leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1} + x_i \\ &\leq (r - x_i) \cdot (1 + \varepsilon n/2)^i + x_i \\ &\leq r \cdot (1 + \varepsilon n/2)^i - x_i \cdot ((1 + \varepsilon n/2)^i - 1) \\ &\leq r \cdot (1 + \varepsilon n/2)^i. \end{aligned}$$

so the claim holds.

Now let $s^* = \max S_n$ and $r^* = \max R_n$. Clearly $r^* \leq s^*$, since $R_n \subseteq S_n$. Our claim implies that there is some $r \in R_n$ such that $s^* \leq r \cdot (1 + \varepsilon/2n)^n$. But r cannot be bigger than r^* , so $s^* \leq r^* \cdot (1 + \varepsilon/2n)^n$. The inequalities $e^x \geq 1 + x$ for all x , and $e^x \leq 2x + 1$ for all $0 \leq x \leq 1$, imply that $(1 + \varepsilon/2n)^n \leq e^{\varepsilon/2} \leq 1 + \varepsilon$. \square

Theorem 8. APPROXSUBSETSUM runs in $O((n^3 \log n)/\varepsilon)$ time.

Proof: Assuming we keep each set R_i in a sorted array, we can merge the two sorted arrays R_{i-1} and $R_{i-1} + x_i$ in $O(|R_{i-1}|)$ time. FILTERIN R_i and removing elements larger than t also requires only $O(|R_{i-1}|)$ time. Thus, the overall running time of our algorithm is $O(\sum_i |R_i|)$; to express this in terms of n and ε , we need to prove an upper bound on the size of each set R_i .

Let $\delta = \varepsilon/2n$. Because we consider the elements of X in increasing order, every element of R_i is between 0 and $i \cdot x_i$. In particular, every element of $R_{i-1} + x_i$ is between x_i and $i \cdot x_i$. After FILTERING, at most one element $r \in R_i$ lies in the range $(1 + \delta)^k \leq r < (1 + \delta)^{k+1}$, for any k . Thus, at most $\lceil \log_{1+\delta} i \rceil$ elements of $R_{i-1} + x_i$ survive the call to FILTER. It follows that

$$\begin{aligned} |R_i| &= |R_{i-1}| + \left\lceil \frac{\log i}{\log(1 + \delta)} \right\rceil \\ &\leq |R_{i-1}| + \left\lceil \frac{\log n}{\log(1 + \delta)} \right\rceil & [i \leq n] \\ &\leq |R_{i-1}| + \left\lceil \frac{2 \ln n}{\delta} \right\rceil & [e^x \leq 1 + 2x \text{ for all } 0 \leq x \leq 1] \\ &\leq |R_{i-1}| + \left\lceil \frac{n \ln n}{\varepsilon} \right\rceil & [\delta = \varepsilon/2n] \end{aligned}$$

Unrolling this recurrence into a summation gives us the upper bound $|R_i| \leq i \cdot \lceil (n \ln n)/\varepsilon \rceil = O((n^2 \log n)/\varepsilon)$.

We conclude that the overall running time of APPROXSUBSETSUM is $O((n^3 \log n)/\varepsilon)$, as claimed. \square

Exercises

1. (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
 (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
 (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.
2. (a) Find the smallest graph (minimum number of edges) for which GREEDYVERTEXCOVER does not return the smallest vertex cover.
 (b) For any integer n , describe an n -vertex graph for which GREEDYVERTEXCOVER returns a vertex cover of size $OPT \cdot \Omega(\log n)$.
3. (a) Find the smallest graph (minimum number of edges) for which DUMBVERTEXCOVER does not return the smallest vertex cover.
 (b) Describe an infinite family of graphs for which DUMBVERTEXCOVER returns a vertex cover of size $2 \cdot OPT$.
4. Consider the following heuristic for constructing a vertex cover of a connected graph G : return the set of non-leaf nodes in any depth-first spanning tree of G .
 - (a) Prove that this heuristic returns a vertex cover of G .
 - (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G .
 - (c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.
5. Consider the following optimization version of the PARTITION problem. Given a set X of positive integers, our task is to partition X into disjoint subsets A and B such that $\max\{\sum A, \sum B\}$ is as small as possible. This problem is clearly NP-hard. Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```

PARTITION( $X[1..n]$ ):
  Sort  $X$  in increasing order
   $a \leftarrow 0$ ;  $b \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $n$ 
    if  $a < b$ 
       $a \leftarrow a + X[i]$ 
    else
       $b \leftarrow b + X[i]$ 
  return  $\max\{a, b\}$ 

```

6. The *chromatic number* $\chi(G)$ of a graph G is the minimum number of colors required to color the vertices of the graph, so that every edge has endpoints with different colors. Computing the chromatic number exactly is NP-hard.

Prove that the following problem is also NP-hard: Given an n -vertex graph G , return any integer between $\chi(G)$ and $\chi(G) + 573$. [Note: This does not contradict the possibility of a constant factor approximation algorithm.]

7. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.
 - (a) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G . Prove that it is NP-hard to approximate $zzz(G)$ within a factor of 10^{100} .
 - (b) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G . Suppose we assign each vertex in G a *random* color from the set {red, green, blue}. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.
8. Consider the following algorithm for coloring a graph G .

```

TREECOLOR( $G$ ):
   $T \leftarrow$  any spanning tree of  $G$ 
  Color the tree  $T$  with two colors
   $c \leftarrow 2$ 
  for each edge  $(u, v) \in G \setminus T$ 
     $T \leftarrow T \cup \{(u, v)\}$ 
    if  $color(u) = color(v)$    «(Try recoloring  $u$  with an existing color)»
      for  $i \leftarrow 1$  to  $c$ 
        if no neighbor of  $u$  in  $T$  has color  $i$ 
           $color(u) \leftarrow i$ 
    if  $color(u) = color(v)$    «(Try recoloring  $v$  with an existing color)»
      for  $i \leftarrow 1$  to  $c$ 
        if no neighbor of  $v$  in  $T$  has color  $i$ 
           $color(v) \leftarrow i$ 
    if  $color(u) = color(v)$    «(Give up and create a new color)»
       $c \leftarrow c + 1$ 
       $color(u) \leftarrow c$ 

```

- (a) Prove that this algorithm colors any bipartite graph with just two colors.
 - (b) Let $\Delta(G)$ denote the maximum degree of any vertex in G . Prove that this algorithm colors any graph G with at most $\Delta(G)$ colors. This trivially implies that TREECOLOR is a $\Delta(G)$ -approximation algorithm.
 - (c) Prove that TREECOLOR is *not* a constant-factor approximation algorithm.
9. The KNAPSACK problem can be defined as follows. We are given a finite set of elements X where each element $x \in X$ has a non-negative *size* and a non-negative *value*, along with an integer *capacity* c . Our task is to determine the maximum total value among all subsets of X whose total size is at most c . This problem is NP-hard. Specifically, the optimization version of SUBSETSUM is a special case, where each element's value is equal to its size.

Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
APPROXKNAPSACK( $X, c$ ):
    return max{GREEDYKNAPSACK( $X, c$ ), PICKBESTONE( $X, c$ )}
```

```
GREEDYKNAPSACK( $X, c$ ):
```

```
Sort  $X$  in decreasing order by the ratio  $value/size$ 
 $S \leftarrow 0; V \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$ 
    if  $S + size(x_i) > c$ 
        return  $V$ 
     $S \leftarrow S + size(x_i)$ 
     $V \leftarrow V + value(x_i)$ 
return  $V$ 
```

```
PICKBESTONE( $X, c$ ):
```

```
Sort  $X$  in increasing order by  $size$ 
 $V \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$ 
    if  $size(x_i) > c$ 
        return  $V$ 
    if  $value(x_i) > V$ 
         $V \leftarrow value(x_i)$ 
return  $V$ 
```

10. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is NP-Hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NEXTFIT( $W[1..n]$ ):
```

```
 $b \leftarrow 0$ 
 $Total[0] \leftarrow \infty$ 
for  $i \leftarrow 1$  to  $n$ 
    if  $Total[b] + W[i] > 1$ 
         $b \leftarrow b + 1$ 
         $Total[b] \leftarrow W[i]$ 
    else
         $Total[b] \leftarrow Total[b] + W[i]$ 
return  $b$ 
```

```
FIRSTFIT( $W[1..n]$ ):
```

```
 $b \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$ 
     $j \leftarrow 1; found \leftarrow \text{FALSE}$ 
    while  $j \leq b$  and  $found = \text{FALSE}$ 
        if  $Total[j] + W[i] \leq 1$ 
             $Total[j] \leftarrow Total[j] + W[i]$ 
             $found \leftarrow \text{TRUE}$ 
         $j \leftarrow j + 1$ 
    if  $found = \text{FALSE}$ 
         $b \leftarrow b + 1$ 
         $Total[b] = W[i]$ 
return  $b$ 
```

- (a) Prove that NEXTFIT uses at most twice the optimal number of bins.
- (b) Prove that FIRSTFIT uses at most twice the optimal number of bins.
- *(c) Prove that if the weight array W is initially sorted in decreasing order, then FIRSTFIT uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):
 - In the packing computed by FIRSTFIT, every item with weight more than $1/3$ is placed in one of the first OPT bins.
 - FIRSTFIT places at most $OPT - 1$ items outside the first OPT bins.

11. Given a graph G with edge weights and an integer k , suppose we wish to partition the vertices of G into k subsets S_1, S_2, \dots, S_k so that the sum of the weights of the edges that cross the partition (that is, have endpoints in different subsets) is as large as possible.

- (a) Describe an efficient $(1 - 1/k)$ -approximation algorithm for this problem.
- (b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.
12. The lecture notes describe a $(3/2)$ -approximation algorithm for the metric traveling salesman problem. Here, we consider computing minimum-cost Hamiltonian *paths*. Our input consists of a graph G whose edges have weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.
- (a) If our input includes zero endpoints, describe a $(3/2)$ -approximation to the problem of computing a minimum cost Hamiltonian path.
 - (b) If our input includes one endpoint u , describe a $(3/2)$ -approximation to the problem of computing a minimum cost Hamiltonian path that starts at u .
 - (c) If our input includes two endpoints u and v , describe a $(5/3)$ -approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v .
13. Suppose we are given a collection of n *jobs* to execute on a machine containing a row of m processors. When the i th job is executed, it occupies a *contiguous* set of $\text{prox}[i]$ processors for $\text{time}[i]$ seconds. A *schedule* for a set of jobs assigns each job an interval of processors and a starting time, so that no processor works on more than one job at any time. The *makespan* of a schedule is the time from the start to the finish of all jobs. Finally, the *parallel scheduling problem* asks us to compute the schedule with the smallest possible makespan.
- (a) Prove that the parallel scheduling problem is NP-hard.
 - (b) Give an algorithm that computes a 3-approximation of the minimum makespan of a set of jobs in $O(m \log m)$ time. That is, if the minimum makespan is M , your algorithm should compute a schedule with make-span at most $3M$. You can assume that n is a power of 2.
14. Consider the greedy algorithm for metric TSP: start at an arbitrary vertex u , and at each step, travel to the closest unvisited vertex.
- (a) Show that the greedy algorithm for metric TSP is an $O(\log n)$ -approximation, where n is the number of vertices. [Hint: Argue that the k th least expensive edge in the tour output by the greedy algorithm has weight at most $\text{OPT}/(n - k + 1)$; try $k = 1$ and $k = 2$ first.]
 - *(b) Show that the greedy algorithm for metric TSP is no better than an $O(\log n)$ -approximation. That is, describe an infinite family of weighted graphs such that the greedy algorithm returns a cycle whose weight is $\Omega(\log n)$ times the optimal TSP tour.

“...O Zarathustra, who you are and must become” behold you are the teacher of the eternal recurrence – that is your destiny! That you as the first must teach this doctrine – how could this great destiny not be your greatest danger and sickness too?

— Friedrich Nietzsche, *Also sprach Zarathustra* (1885)
[translated by Walter Kaufmann]

Solving Recurrences

1 Introduction

A *recurrence* is a recursive description of a function, usually of the form $F : \mathbb{N} \rightarrow \mathbb{R}$, or a description of such a function in terms of itself. Like all recursive structures, a recurrence consists of one or more *base cases* and one or more *recursive cases*. Each of these cases is an equation or inequality, with some function value $f(n)$ on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n . The recursive cases relate the function value $f(n)$ to function value $f(k)$ for one or more integers $k < n$; typically, each recursive case applies to an infinite number of possible values of n .

For example, the following recurrence (written in two different but standard ways) describes the identity function $f(n) = n$:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n - 1) + 1 & \text{otherwise} \end{cases} \quad \begin{aligned} f(0) &= 0 \\ f(n) &= f(n - 1) + 1 \text{ for all } n > 0 \end{aligned}$$

In both presentations, the first line is the only base case, and the second line is the only recursive case. The same function can satisfy *many* different recurrences; for example, both of the following recurrences also describe the identity function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & \text{otherwise} \end{cases} \quad f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot f(n/2) & \text{if } n \text{ is even and } n > 0 \\ f(n - 1) + 1 & \text{if } n \text{ is odd} \end{cases}$$

We say that a particular function *satisfies* a recurrence, or is the *solution* to a recurrence, if each of the statements in the recurrence is true. Most recurrences—at least, those that we will encounter in this class—have a solution; moreover, if every case of the recurrence is an equation, that solution is unique. Specifically, if we transform the recursive formula into a recursive *algorithm*, the solution to the recurrence is the function computed by that algorithm!

Recurrences arise naturally in the analysis of algorithms, especially recursive algorithms. In many cases, we can express the running time of an algorithm as a recurrence, where the recursive cases of the recurrence correspond exactly to the recursive cases of the algorithm. Recurrences are also useful tools for solving counting problems—How many objects of a particular kind exist?

By itself, a recurrence is not a satisfying description of the running time of an algorithm or a bound on the number of widgets. Instead, we need a *closed-form* solution to the recurrence; this is a *non-recursive* description of a function that satisfies the recurrence. For recurrence *equations*, we sometimes prefer an *exact* closed-form solution, but such a solution may not exist, or may be too complex to be useful. Thus, for most recurrences, especially those arising in algorithm analysis, we can be satisfied with an *asymptotic* solution of the form $\Theta(f(n))$, for some explicit (non-recursive) function $g(n)$.

For recursive *inequalities*, we prefer a *tight* solution; this is a function that would still satisfy the recurrence if all the inequalities were replaced with the corresponding equations. Again, exactly tight solutions may not exist, or may be too complex to be useful, so we may have to settle for a looser solution and/or an asymptotic solution of the form $O(g(n))$ or $\Omega(g(n))$.

2 The Ultimate Method: Guess and Confirm

Ultimately, there is only one fail-safe method to solve *any* recurrence:

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

Later sections of these notes describe techniques 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 will succeed, in which case you're done, or the proof will fail, in which case *the failure will help you refine your guess*. Where you get your initial guess is utterly irrelevant¹—from a classmate, from a textbook, on the web, from the answer to a different problem, scrawled on a bathroom wall in Siebel, included in a care package from your mom, dictated by the machine elves, whatever. If you can prove that the answer is correct, then it's correct!

2.1 Tower of Hanoi

The classical Tower of Hanoi problem gives us the recurrence $T(n) = 2T(n - 1) + 1$ with base case $T(0) = 0$. 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 $T(n)$, we discover that they are each one less than a power of two.

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

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

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

We were right! Hooray, we're done!

Another way we can guess the solution is by *unrolling* the recurrence, by substituting it into itself:

$$\begin{aligned} T(n) &= 2T(n - 1) + 1 \\ &= 2(2T(n - 2) + 1) + 1 \\ &= 4T(n - 2) + 3 \\ &= 4(2T(n - 3) + 1) + 3 \\ &= 8T(n - 2) + 7 \\ &= \dots \end{aligned}$$

¹... except of course during exams, where you aren't supposed to use *any* outside sources

It looks like unrolling the initial Hanoi recurrence k times, for any non-negative integer k , will give us the new recurrence $T(n) = 2^k T(n - k) + (2^k - 1)$. Let's prove this by induction:

$$\begin{aligned} T(n) &= 2T(n - 1) + 1 \quad \checkmark & [k = 0, \text{ by definition}] \\ T(n) &= 2^{k-1}T(n - (k - 1)) + (2^{k-1} - 1) & [\text{inductive hypothesis}] \\ &= 2^{k-1}(2T(n - k) + 1) + (2^{k-1} - 1) & [\text{initial recurrence for } T(n - (k - 1))] \\ &= 2^k T(n - k) + (2^k - 1) \quad \checkmark & [\text{algebra}] \end{aligned}$$

Our guess was correct! In particular, unrolling the recurrence n times give us the recurrence $T(n) = 2^n T(0) + (2^n - 1)$. Plugging in the base case $T(0) = 0$ give us the closed-form solution $T(n) = 2^n - 1$.

2.2 Fibonacci numbers

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

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\leq \alpha \cdot c^{n-1} + \alpha \cdot c^{n-2} & [\text{"induction hypothesis"}] \\ &\leq \alpha \cdot c^n \quad ??? \end{aligned}$$

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

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

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

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

What about a matching lower bound? Essentially the same inductive proof implies that $F_n \geq \beta \cdot \phi^n$ for some constant β , but the only value of β that works for *all* n is the trivial $\beta = 0$! We could try to find some lower-order term that makes the base case non-trivial, but an easier approach is to recall that asymptotic $\Omega()$ 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 easy 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 $\beta \leq 1/\phi^2$, which implies that $F_n \geq \phi^{n-2}$ for all $n \geq 1$.

Putting the upper and lower bounds together, we obtain the tight asymptotic bound $F_n = \Theta(\phi^n)$. It is possible to get a more exact solution by speculatively refining and conforming our current bounds, but it's not easy. Fortunately, if we really need it, we can get an exact solution using the *annihilator* method, which we'll see later in these notes.

2.3 Mergesort

Mergesort is a classical recursive divide-and-conquer algorithm for sorting an array. The algorithm splits the array in half, recursively sorts the two halves, and then merges the two sorted subarrays into the final sorted array.

```
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  $n$ 
     $A[k] \leftarrow B[k]$ 
```

Let $T(n)$ denote the worst-case running time of MERGESORT when the input array has size n . The MERGE subroutine clearly runs in $\Theta(n)$ time, so the function $T(n)$ satisfies the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{otherwise.} \end{cases}$$

For now, let's consider the special case where n is a power of 2; this assumption allows us to take the floors and ceilings out of the recurrence. (We'll see how to deal with the floors and ceilings later; the short version is that they don't matter.)

Because the recurrence itself is given only asymptotically—in terms of $\Theta()$ expressions—we can't hope for anything but an asymptotic solution. So we can safely simplify the recurrence further by removing the Θ 's; any asymptotic solution to the simplified recurrence will also satisfy the original recurrence. (This simplification is actually important for another reason; if we kept the asymptotic expressions, we might be tempted to simplify them inappropriately.)

Our simplified recurrence now looks like this:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

To guess at a solution, let's try unrolling the recurrence.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n = \dots \end{aligned}$$

It looks like $T(n)$ satisfies the recurrence $T(n) = 2^k T(n/2^k) + kn$ for any positive integer k . Let's verify this by induction.

$$\begin{aligned}
 T(n) &= 2T(n/2) + n = 2^1 T(n/2^1) + 1 \cdot n \quad \checkmark & [k=1, \text{ given recurrence}] \\
 T(n) &= 2^{k-1} T(n/2^{k-1}) + (k-1)n & [\text{inductive hypothesis}] \\
 &= 2^{k-1} (2T(n/2^k) + n/2^{k-1}) + (k-1)n & [\text{substitution}] \\
 &= 2^k T(n/2^k) + kn \quad \checkmark & [\text{algebra}]
 \end{aligned}$$

Our guess was right! The recurrence becomes trivial when $n/2^k = 1$, or equivalently, when $k = \log_2 n$:

$$T(n) = nT(1) + n \log_2 n = n \log_2 n + n.$$

Finally, we have to put back the Θ 's we stripped off; our final closed-form solution is $T(n) = \Theta(n \log n)$.

2.4 An uglier divide-and-conquer example

Consider the divide-and-conquer recurrence $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorem (which we'll see below), 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. (We could also attack this recurrence by unrolling, but let's see how far just guessing will take us.)

Let's start by trying to prove an upper bound $T(n) \leq a n \lg n$ for all sufficiently large n and some constant a to be determined later:

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

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

But before you break out the champagne, 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 $T(n) \geq b n \log n$ for sufficiently large n , we run into trouble.

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

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:

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

But an inductive proof of the upper 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.

Okay... what else is between n and $n \lg n$? How about $n \lg \lg n$?

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

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

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

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 $T(n) = \Theta(n \log \log n)$.

3 Divide and Conquer Recurrences (Recursion Trees)

Many divide and conquer algorithms 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. There is a simple and general technique for solving many recurrences in this and similar forms, using a **recursion tree**. The root of the recursion tree is a box containing the value $f(n)$; the root has a children, each of which is the root of a (recursively defined) recursion tree for the function $T(n/b)$.

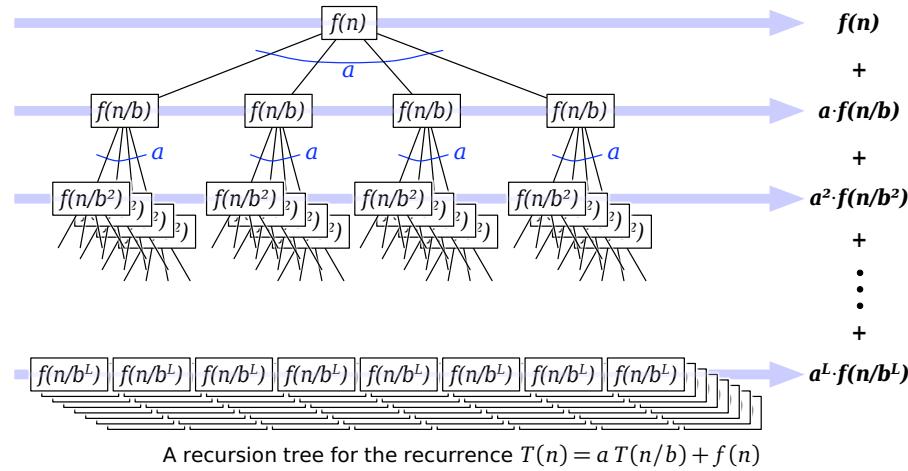
Equivalently, a recursion tree is a complete a -ary tree where each node at depth i contains the value $f(n/b^i)$. The recursion stops when we get to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the exact base case doesn't matter; we can safely assume that $T(1) = \Theta(1)$, or even that $T(n) = \Theta(1)$ for all $n \leq 10^{100}$. I'll also assume for simplicity that n is an integral power of b ; we'll see how to avoid this assumption later (but to summarize: it doesn't matter).

Now $T(n)$ is just the sum of all values stored in the recursion tree. For each i , the i th level of the tree contains a^i nodes, each with value $f(n/b^i)$. Thus,

$$T(n) = \sum_{i=0}^L a^i f(n/b^i) \quad (\Sigma)$$

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

For most divide-and-conquer recurrences, the level-by-level sum (Σ) is a *geometric series*—each term is a constant factor larger or smaller than the previous term. In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the $\Theta(\cdot)$ notation.



Here are several examples of the recursion-tree technique in action:

- **Mergesort (simplified): $T(n) = 2T(n/2) + n$**

There are 2^i nodes at level i , each with value $n/2^i$, so every term in the level-by-level sum (Σ) is the same:

$$T(n) = \sum_{i=0}^L n.$$

The recursion tree has $L = \log_2 n$ levels, so $T(n) = \Theta(n \log n)$.

- **Randomized selection: $T(n) = T(3n/4) + n$**

The recursion tree is a single path. The node at depth i has value $(3/4)^i n$, so the level-by-level sum (Σ) is a decreasing geometric series:

$$T(n) = \sum_{i=0}^L (3/4)^i n.$$

This geometric series is dominated by its initial term n , so $T(n) = \Theta(n)$. The recursion tree has $L = \log_{4/3} n$ levels, but so what?

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

There are 3^i nodes at depth i , each with value $n/2^i$, so the level-by-level sum (Σ) is an increasing geometric series:

$$T(n) = \sum_{i=0}^L (3/2)^i n.$$

This geometric series is dominated by its final term $(3/2)^L n$. Each leaf contributes 1 to this term; thus, the final term is equal to the number of leaves in the tree! The recursion tree has $L = \log_2 n$ levels, and therefore $3^{\log_2 n} = n^{\log_2 3}$ leaves, so $T(n) = \Theta(n^{\log_2 3})$.

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

The sum of all the nodes in the i th level is $n/(\lg n - i)$. This implies that the depth of the tree is at most $\lg n - 1$. The level sums are neither constant nor a geometric series, so we just have to evaluate the overall sum directly.

Recall (or if you're seeing this for the first time: Behold!) that the n th *harmonic number* H_n is the sum of the reciprocals of the first n positive integers:

$$H_n := \sum_{i=1}^n \frac{1}{i}$$

It's not hard to show that $H_n = \Theta(\log n)$; in fact, we have the stronger inequalities $\ln(n+1) \leq H_n \leq \ln n + 1$.

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

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

There are 4^i nodes at each level i , each with value $(n/2^i)\lg(n/2^i) = (n/2^i)(\lg n - i)$; again, the depth of the tree is at most $\lg n - 1$. We have the following summation:

$$T(n) = \sum_{i=0}^{\lg n - 1} n2^i(\lg n - i)$$

We can simplify this sum by substituting $j = \lg n - i$:

$$T(n) = \sum_{j=i}^{\lg n} n2^{\lg n - j} j = \sum_{j=i}^{\lg n} n^2 j / 2^j = \Theta(j^2)$$

The last step uses the fact that $\sum_{j=1}^{\infty} j/2^j = 2$. Although this is not quite a geometric series, it is still dominated by its largest term.

- **Ugly divide and conquer:** $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$

We solved this recurrence earlier by guessing the right answer and verifying, but we can use recursion trees to get the correct answer directly. The *degree* of the nodes in the recursion tree is

no longer constant, so we have to be a bit more careful, but the same basic technique still applies. It's not hard to see that the nodes in any level sum to n . 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 $T(n) = \Theta(n \lg \lg n)$.

- **Randomized quicksort:** $T(n) = T(3n/4) + T(n/4) + n$

This recurrence isn't in the standard form described earlier, but we can still solve it using recursion trees. Now nodes in the same level of the recursion tree have different values, and different leaves are at different levels. However, the nodes in any *complete* level (that is, above any of the leaves) sum to n . Moreover, every leaf in the recursion tree 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 $n \log_4 n \leq T(n) \leq n \log_{4/3} n$. Since these bounds differ by only a constant factor, we have $T(n) = \Theta(n \log n)$.

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

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 $T(n) = n + 9n/10 + 81n/100 + \dots$. 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 $T(n) = \Theta(n)$.

- **Randomized search trees:** $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$

This looks like a divide-and-conquer recurrence, but what does it mean to have a quarter of a child? The right approach is to imagine that each node in the recursion tree has a *weight* in addition to its value. Alternately, we get a standard recursion tree again if we add a second real parameter to the recurrence, defining $T(n) = T(n, 1)$, where

$$T(n, \alpha) = T(n/4, \alpha/4) + T(3n/4, 3\alpha/4) + \alpha.$$

In each complete level of the tree, the (weighted) node values sum to exactly 1. The leaves of the recursion tree are at different levels, but all between $\log_4 n$ and $\log_{4/3} n$. So we have upper and lower bounds $\log_4 n \leq T(n) \leq \log_{4/3} n$, which differ by only a constant factor, so $T(n) = \Theta(\log n)$.

- **Ham-sandwich trees:** $T(n) = T(n/2) + T(n/4) + 1$

Again, we have a lopsided recursion tree. If we only look at complete levels, we find that the level sums form an *ascending* geometric series $T(n) = 1 + 2 + 4 + \dots$, so the solution is dominated by the number of leaves. The recursion tree has $\log_4 n$ complete levels, so there are more than $2^{\log_4 n} = n^{\log_4 2} = \sqrt{n}$; on the other hand, every leaf has depth at most $\log_2 n$, so the total number of leaves is at most $2^{\log_2 n} = n$. Unfortunately, the crude bounds $\sqrt{n} \ll T(n) \ll n$ are the best we can derive using the techniques we know so far!

The following theorem completely describes the solution for any divide-and-conquer recurrence in the ‘standard form’ $T(n) = aT(n/b) + f(n)$, where a and b are constants and $f(n)$ is a polynomial. This theorem allows us to bypass recursion trees for ‘standard’ recurrences, but many people (including Jeff) find it harder to remember than the more general recursion-tree technique. Your mileage may vary.

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 sum is equal to $f(n)$. \square

4 Linear Recurrences (Annihilators)

Another common class of recurrences, called *linear* recurrences, arises in the context of recursive backtracking algorithms and counting problems. These recurrences express each function value $f(n)$ as a *linear* combination of a small number of nearby values $f(n-1), f(n-2), f(n-3), \dots$. The Fibonacci recurrence is a typical example:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

It turns out that the solution to *any* linear recurrence is a simple combination of polynomial and exponential functions in n . For example, we can verify by induction that the linear recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \text{ or } n = 2 \\ 3T(n-1) - 8T(n-2) + 4T(n-3) & \text{otherwise} \end{cases}$$

has the closed-form solution $T(n) = (n-3)2^n + 4$. First we check the base cases:

$$\begin{aligned} T(0) &= (0-3)2^0 + 4 = 1 & \checkmark \\ T(1) &= (1-3)2^1 + 4 = 0 & \checkmark \\ T(2) &= (2-3)2^2 + 4 = 0 & \checkmark \end{aligned}$$

And now the recursive case:

$$\begin{aligned} T(n) &= 3T(n-1) - 8T(n-2) + 4T(n-3) \\ &= 3((n-4)2^{n-1} + 4) - 8((n-5)2^{n-2} + 4) + 4((n-6)2^{n-3} + 4) \\ &= \left(\frac{3}{2} - \frac{8}{4} + \frac{4}{8}\right)n \cdot 2^n - \left(\frac{12}{2} - \frac{40}{4} + \frac{24}{8}\right)2^n + (2-8+4) \cdot 4 \\ &= (n-3) \cdot 2^n + 4 & \checkmark \end{aligned}$$

But how could we have possibly come up with that solution? In this section, I'll describe a general method for solving linear recurrences that's arguably easier than the induction proof!

4.1 Operators

Our technique for solving linear recurrences relies on the theory of *operators*. Operators are higher-order functions, which take one or more functions as input and produce different functions as output. For example, your first two semesters of calculus focus almost exclusively on the *differential* and *integral* operators $\frac{d}{dx}$ and $\int dx$. All the operators we will need are combinations of three elementary building blocks:

- **Sum:** $(f + g)(n) := f(n) + g(n)$
- **Scale:** $(\alpha \cdot f)(n) := \alpha \cdot (f(n))$
- **Shift:** $(Ef)(n) := f(n + 1)$

The shift and scale operators are *linear*, which means they can be distributed over sums; for example, for any functions f , g , and h , we have $E(f - 3(g - h)) = Ef + (-3)Eg + 3 Eh$.

We can combine these building blocks to obtain more complex *compound* operators. For example, the compound operator $E - 2$ is defined by setting $(E - 2)f := Ef + (-2)f$ for any function f . We can also apply the shift operator twice: $(E(Ef))(n) = f(n + 2)$; we write usually E^2f as a synonym for $E(Ef)$. More generally, for any positive integer k , the operator E^k shifts its argument k times: $E^k f(n) = f(n + k)$. Similarly, $(E - 2)^2$ is shorthand for the operator $(E - 2)(E - 2)$, which applies $(E - 2)$ twice.

For example, here are the results of applying different operators to the function $f(n) = 2^n$:

$$\begin{aligned} 2f(n) &= 2 \cdot 2^n = 2^{n+1} \\ 3f(n) &= 3 \cdot 2^n \\ Ef(n) &= 2^{n+1} \\ E^2f(n) &= 2^{n+2} \\ (E - 2)f(n) &= Ef(n) - 2f(n) = 2^{n+1} - 2^{n+1} = 0 \\ (E^2 - 1)f(n) &= E^2f(n) - f(n) = 2^{n+2} - 2^n = 3 \cdot 2^n \end{aligned}$$

These compound operators can be manipulated exactly as though they were polynomials over the ‘variable’ E . In particular, we can ‘factor’ compound operators into ‘products’ of simpler operators, and the order of the factors is unimportant. For example, the compound operators $E^2 - 3E + 2$ and $(E - 1)(E - 2)$ are equivalent:

$$\text{Let } g(n) := (E - 2)f(n) = f(n + 1) - 2f(n).$$

$$\begin{aligned} \text{Then } (E - 1)(E - 2)f(n) &= (E - 1)g(n) \\ &= g(n + 1) - g(n) \\ &= (f(n + 2) - 2f(n + 1)) - (f(n + 1) - 2f(n)) \\ &= f(n + 2) - 3f(n + 1) + 2f(n) \\ &= (E^2 - 3E + 2)f(n). \quad \checkmark \end{aligned}$$

It is an easy exercise to confirm that $E^2 - 3E + 2$ is also equivalent to the operator $(E - 2)(E - 1)$.

The following table summarizes everything we need to remember about operators.

Operator	Definition
addition	$(f + g)(n) := f(n) + g(n)$
subtraction	$(f - g)(n) := f(n) - g(n)$
multiplication	$(\alpha \cdot f)(n) := \alpha \cdot (f(n))$
shift	$Ef(n) := f(n+1)$
k -fold shift	$E^k f(n) := f(n+k)$
composition	$(X + Y)f := Xf + Yf$ $(X - Y)f := Xf - Yf$ $XYf := X(Yf) = Y(Xf)$
distribution	$X(f + g) = Xf + Xg$

4.2 Annihilators

An **annihilator** of a function f is any nontrivial operator that transforms f into the zero function. (We can trivially annihilate any function by multiplying it by zero, so as a technical matter, we do not consider the zero operator to be an annihilator.) Every compound operator we consider annihilates a specific class of functions; conversely, every function composed of polynomial and exponential functions has a unique (minimal) annihilator.

We have already seen that the operator $(E - 2)$ annihilates the function 2^n . It's not hard to see that the operator $(E - c)$ annihilates the function $\alpha \cdot c^n$, for any constants c and α . More generally, the operator $(E - c)$ annihilates the function a^n if and only if $c = a$:

$$(E - c)a^n = Ea^n - c \cdot a^n = a^{n+1} - c \cdot a^n = (a - c)a^n.$$

Thus, $(E - 2)$ is essentially the *only* annihilator of the function 2^n .

What about the function $2^n + 3^n$? The operator $(E - 2)$ annihilates the function 2^n , but leaves the function 3^n unchanged. Similarly, $(E - 3)$ annihilates 3^n while *negating* the function 2^n . But if we apply *both* operators, we annihilate both terms:

$$\begin{aligned} (E - 2)(2^n + 3^n) &= E(2^n + 3^n) - 2(2^n + 3^n) \\ &= (2^{n+1} + 3^{n+1}) - (2^{n+1} + 2 \cdot 3^n) = 3^n \\ \implies (E - 3)(E - 2)(2^n + 3^n) &= (E - 3)3^n = 0 \end{aligned}$$

In general, for any integers $a \neq b$, the operator $(E - a)(E - b) = (E - b)(E - a) = (E^2 - (a+b)E + ab)$ annihilates any function of the form $\alpha a^n + \beta b^n$, but nothing else.

What about the operator $(E - a)(E - a) = (E - a)^2$? It turns out that this operator annihilates all functions of the form $(\alpha n + \beta)a^n$:

$$\begin{aligned} (E - a)((\alpha n + \beta)a^n) &= (\alpha(n+1) + \beta)a^{n+1} - a(\alpha n + \beta)a^n \\ &= \alpha a^{n+1} \\ \implies (E - a)^2((\alpha n + \beta)a^n) &= (E - a)(\alpha a^{n+1}) = 0 \end{aligned}$$

More generally, the operator $(E - a)^d$ annihilates all functions of the form $p(n) \cdot a^n$, where $p(n)$ is a polynomial of degree at most $d - 1$. For example, $(E - 1)^3$ annihilates any polynomial of degree at most 2.

The following table summarizes everything we need to remember about annihilators.

Operator	Functions annihilated
$E - 1$	α
$E - a$	αa^n
$(E - a)(E - b)$	$\alpha a^n + \beta b^n$ [if $a \neq b$]
$(E - a_0)(E - a_1) \cdots (E - a_k)$	$\sum_{i=0}^k \alpha_i a_i^n$ [if a_i distinct]
$(E - 1)^2$	$\alpha n + \beta$
$(E - a)^2$	$(\alpha n + \beta)a^n$
$(E - a)^2(E - b)$	$(\alpha n + \beta)a^b + \gamma b^n$ [if $a \neq b$]
$(E - a)^d$	$(\sum_{i=0}^{d-1} \alpha_i n^i)a^n$
If X annihilates f , then X also annihilates Ef .	
If X annihilates both f and g , then X also annihilates $f \pm g$.	
If X annihilates f , then X also annihilates αf , for any constant α .	
If X annihilates f and Y annihilates g , then XY annihilates $f \pm g$.	

4.3 Annihilating Recurrences

Given a linear recurrence for a function, it's easy to extract an annihilator for that function. For many recurrences, we only need to rewrite the recurrence in operator notation. Once we have an annihilator, we can factor it into operators of the form $(E - c)$; the table on the previous page then gives us a generic solution with some unknown coefficients. If we are given explicit base cases, we can determine the coefficients by examining a few small cases; in general, this involves solving a small system of linear equations. If the base cases are not specified, the generic solution almost always gives us an asymptotic solution. Here is the technique step by step:

1. Write the recurrence in operator form
2. Extract an annihilator for the recurrence
3. Factor the annihilator (if necessary)
4. Extract the *generic solution* from the annihilator
5. Solve for coefficients using base cases (if known)

Here are several examples of the technique in action:

- $r(n) = 5r(n - 1)$, where $r(0) = 3$.

1. We can write the recurrence in operator form as follows:

$$r(n) = 5r(n - 1) \implies r(n + 1) - 5r(n) = 0 \implies (E - 5)r(n) = 0.$$

2. We immediately see that $(E - 5)$ annihilates the function $r(n)$.
3. The annihilator $(E - 5)$ is already factored.
4. Consulting the annihilator table on the previous page, we find the generic solution $r(n) = \alpha 5^n$ for some constant α .
5. The base case $r(0) = 3$ implies that $\alpha = 3$.

We conclude that $r(n) = 3 \cdot 5^n$. We can easily verify this closed-form solution by induction:

$$\begin{aligned} r(0) &= 3 \cdot 5^0 = 3 & \checkmark & & & & [\text{definition}] \\ r(n) &= 5r(n-1) & & & & & [\text{definition}] \\ &= 5 \cdot (3 \cdot 5^{n-1}) & & & & & [\text{induction hypothesis}] \\ &= 5^n \cdot 3 & \checkmark & & & & [\text{algebra}] \end{aligned}$$

- **Fibonacci numbers:** $F(n) = F(n - 1) + F(n - 2)$, where $F(0) = 0$ and $F(1) = 1$.

1. We can rewrite the recurrence as $(E^2 - E - 1)F(n) = 0$.
2. The operator $E^2 - E - 1$ clearly annihilates $F(n)$.
3. The quadratic formula implies that the annihilator $E^2 - E - 1$ factors into $(E - \phi)(E - \hat{\phi})$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618034$ is the golden ratio and $\hat{\phi} = (1 - \sqrt{5})/2 = 1 - \phi = -1/\phi$.
4. The annihilator implies that $F(n) = \alpha\phi^n + \hat{\alpha}\hat{\phi}^n$ for some unknown constants α and $\hat{\alpha}$.
5. The base cases give us two equations in two unknowns:

$$\begin{aligned} F(0) &= 0 = \alpha + \hat{\alpha} \\ F(1) &= 1 = \alpha\phi + \hat{\alpha}\hat{\phi} \end{aligned}$$

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

We conclude with the following exact closed form for the n th Fibonacci number:

$$F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

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.)

- **Towers of Hanoi:** $T(n) = 2T(n - 1) + 1$, where $T(0) = 0$. This is our first example of a *non-homogeneous* recurrence, which means the recurrence has one or more non-recursive terms.

1. We can rewrite the recurrence as $(E - 2)T(n) = 1$.
2. The operator $(E - 2)$ doesn't quite annihilate the function; it leaves a *residue* of 1. But we can annihilate the residue by applying the operator $(E - 1)$. Thus, the compound operator $(E - 1)(E - 2)$ annihilates the function.
3. The annihilator is already factored.
4. The annihilator table gives us the generic solution $T(n) = \alpha 2^n + \beta$ for some unknown constants α and β .
5. The base cases give us $T(0) = 0 = \alpha 2^0 + \beta$ and $T(1) = 1 = \alpha 2^1 + \beta$. Solving this system of equations, we find that $\alpha = 1$ and $\beta = -1$.

We conclude that $T(n) = 2^n - 1$.

For the remaining examples, I won't explicitly enumerate the steps in the solution.

- **Height-balanced trees:** $H(n) = H(n - 1) + H(n - 2) + 1$, where $H(-1) = 0$ and $H(0) = 1$. (Yes, we're starting at -1 instead of 0 . So what?)

We can rewrite the recurrence as $(E^2 - E - 1)H = 1$. The residue 1 is annihilated by $(E - 1)$, so the compound operator $(E - 1)(E^2 - E - 1)$ annihilates the recurrence. This operator factors into $(E - 1)(E - \phi)(E - \hat{\phi})$, where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$. Thus, we get the generic solution $H(n) = \alpha \cdot \phi^n + \beta + \gamma \cdot \hat{\phi}^n$, for some unknown constants α, β, γ that satisfy the following system of equations:

$$\begin{aligned} H(-1) = 0 &= \alpha\phi^{-1} + \beta + \gamma\hat{\phi}^{-1} = \alpha/\phi + \beta - \gamma/\hat{\phi} \\ H(0) = 1 &= \alpha\phi^0 + \beta + \gamma\hat{\phi}^0 = \alpha + \beta + \gamma \\ H(1) = 2 &= \alpha\phi^1 + \beta + \gamma\hat{\phi}^1 = \alpha\phi + \beta + \gamma\hat{\phi} \end{aligned}$$

Solving this system (using Cramer's rule or Gaussian elimination), we find that $\alpha = (\sqrt{5} + 2)/\sqrt{5}$, $\beta = -1$, and $\gamma = (\sqrt{5} - 2)/\sqrt{5}$. We conclude that

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

- $T(n) = 3T(n - 1) - 8T(n - 2) + 4T(n - 3)$, where $T(0) = 1$, $T(1) = 0$, and $T(2) = 0$. This was our original example of a linear recurrence.

We can rewrite the recurrence as $(E^3 - 3E^2 + 8E - 4)T = 0$, so we immediately have an annihilator $E^3 - 3E^2 + 8E - 4$. Using high-school algebra, we can factor the annihilator into $(E - 2)^2(E - 1)$, which implies the generic solution $T(n) = \alpha n 2^n + \beta 2^n + \gamma$. The constants α, β , and γ are determined by the base cases:

$$\begin{aligned} T(0) = 1 &= \alpha \cdot 0 \cdot 2^0 + \beta 2^0 + \gamma = \beta + \gamma \\ T(1) = 0 &= \alpha \cdot 1 \cdot 2^1 + \beta 2^1 + \gamma = 2\alpha + 2\beta + \gamma \\ T(2) = 0 &= \alpha \cdot 2 \cdot 2^2 + \beta 2^2 + \gamma = 8\alpha + 4\beta + \gamma \end{aligned}$$

Solving this system of equations, we find that $\alpha = 1$, $\beta = -3$, and $\gamma = 4$, so $T(n) = (n - 3)2^n + 4$.

- $T(n) = T(n - 1) + 2T(n - 2) + 2^n - n^2$

We can rewrite the recurrence as $(E^2 - E - 2)T(n) = E^2(2^n - n^2)$. Notice that we had to shift up the non-recursive parts of the recurrence when we expressed it in this form. The operator $(E - 2)(E - 1)^3$ annihilates the residue $2^n - n^2$, and therefore also annihilates the shifted residue $E^2(2^n + n^2)$. Thus, the operator $(E - 2)(E - 1)^3(E^2 - E - 2)$ annihilates the entire recurrence. We can factor the quadratic factor into $(E - 2)(E + 1)$, so the annihilator factors into $(E - 2)^2(E - 1)^3(E + 1)$. So the generic solution is $T(n) = \alpha n 2^n + \beta 2^n + \gamma n^2 + \delta n + \epsilon + \eta(-1)^n$. The coefficients $\alpha, \beta, \gamma, \delta, \epsilon, \eta$ satisfy a system of six equations determined by the first six function values $T(0)$ through $T(5)$. For almost² every set of base cases, we have $\alpha \neq 0$, which implies that $T(n) = \Theta(n2^n)$.

For a more detailed explanation of the annihilator method, see George Lueker, Some techniques for solving recurrences, *ACM Computing Surveys* 12(4):419-436, 1980.

²In fact, the only possible solutions with $\alpha = 0$ have the form $-2^{n-1} - n^2/2 - 5n/2 + \eta(-1)^n$ for some constant η .

5 Transformations

Sometimes we encounter recurrences that don't fit the structures required for recursion trees or annihilators. In many of those cases, we can transform the recurrence into a more familiar form, by defining a new function in terms of the one we want to solve. There are many different kinds of transformations, but these three are probably the most useful:

- Domain transformation: Define a new function $S(n) = T(f(n))$ with a simpler recurrence, for some simple function f .
- Range transformation: Define a new function $S(n) = f(T(n))$ with a simpler recurrence, for some simple function f .
- Difference transformation: Simplify the recurrence for $T(n)$ by considering the difference $T(n) - T(n - 1)$.

Here are some examples of these transformations in action.

- **Unsimplified Mergesort:** $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$

When n is a power of 2, we can simplify the mergesort recurrence to $T(n) = 2T(n/2) + \Theta(n)$, which has the solution $T(n) = \Theta(n \log n)$. Unfortunately, for other values of n , this simplified 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 2, we will *never* reach the base case $T(1) = 1$.

So we really need to solve the original recurrence. We have no hope of getting an *exact* solution, even if we ignore the $\Theta()$ in the recurrence; the floors and ceilings will eventually kill us. But we can derive a tight asymptotic solution using a domain transformation—we can rewrite the function $T(n)$ as a nested function $S(f(n))$, where $f(n)$ is a simple function and the function $S()$ has a simpler recurrence.

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

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

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

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

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 implies the matching lower bound $T(n) = \Omega(n \log n)$. So $T(n) = \Theta(n \log n)$ 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!

- **Ham-Sandwich Trees:** $T(n) = T(n/2) + T(n/4) + 1$

As we saw earlier, the recursion tree method only gives us the uselessly loose bounds $\sqrt{n} \ll T(n) \ll n$ for this recurrence, and the recurrence is in the wrong form for annihilators. The authors who discovered ham-sandwich trees (yes, this is a real data structure) solved this recurrence by guessing the solution and giving a complicated induction proof.

But a simple transformation allows us to solve the recurrence in just a few lines. We define a new function $t(k) = T(2^k)$, which satisfies the simpler linear recurrence $t(k) = t(k - 1) + t(k - 2) + 1$. This recurrence should immediately remind you of Fibonacci numbers. Sure enough, the annihilator method implies the solution $t(k) = \Theta(\phi^k)$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio. We conclude that

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

Many other divide-and-conquer recurrences can be similarly transformed into linear recurrences and then solved with annihilators. Consider once more the simplified mergesort recurrence $T(n) = 2T(n/2) + n$. The function $t(k) = T(2^k)$ satisfies the recurrence $t(k) = 2t(k - 1) + 2^k$. The annihilator method gives us the generic solution $t(k) = \Theta(k \cdot 2^k)$, which implies that $T(n) = t(\lg n) = \Theta(n \log n)$, just as we expected.

On the other hand, for some recurrences like $T(n) = T(n/3) + T(2n/3) + n$, the recursion tree method gives an easy solution, but there's no way to transform the recurrence into a form where we can apply the annihilator method directly.³

- **Random Binary Search Trees:** $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$

This looks like a divide-and-conquer recurrence, so we might be tempted to apply recursion trees, but what does it mean to have a quarter of a child? If we're not comfortable with weighted recursion trees, we can instead consider a new function $U(n) = n \cdot T(n)$, which satisfies the recurrence $U(n) = U(n/4) + U(3n/4) + n$. As we've already seen, recursion trees imply that $U(n) = \Theta(n \log n)$, which immediately implies that $T(n) = \Theta(\log n)$.

- **Randomized Quicksort:** $T(n) = \frac{2}{n} \sum_{k=0}^{n-1} T(k) + n$

This is our first example of a *full history* recurrence; each function value $T(n)$ is defined in terms of *all* previous function values $T(k)$ with $k < n$. Before we can apply any of our existing techniques, we need to convert this recurrence into an equivalent *limited history* form by shifting and subtracting away common terms. To make this step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

³However, we can still get a solution via functional transformations as follows. The function $t(k) = T((3/2)^k)$ satisfies the recurrence $t(n) = t(n - 1) + t(n - \lambda) + (3/2)^k$, where $\lambda = \log_{3/2} 3 = 2.709511\dots$. The *characteristic function* for this recurrence is $(r^\lambda - r^{\lambda-1} - 1)(r - 3/2)$, which has a double root at $r = 3/2$ and nowhere else. Thus, $t(k) = \Theta(k(3/2)^k)$, which implies that $T(n) = t(\log_{3/2} n) = \Theta(n \log n)$.

$$n \cdot T(n) = 2 \sum_{k=0}^{n-1} T(k) + n^2 \quad [\text{multiply both sides by } n]$$

$$(n-1) \cdot T(n-1) = 2 \sum_{k=0}^{n-2} T(k) + (n-1)^2 \quad [\text{shift}]$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1 \quad [\text{subtract}]$$

$$T(n) = \frac{n+1}{n} T(n-1) + 2 - \frac{1}{n} \quad [\text{simplify}]$$

We can solve this limited-history recurrence using another functional transformation. We define a new function $t(n) = T(n)/(n+1)$, which satisfies the simpler recurrence

$$t(n) = t(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)},$$

which we can easily unroll into a summation. If we only want an asymptotic solution, we can simplify the final recurrence to $t(n) = t(n-1) + \Theta(1/n)$, which unrolls into a very familiar summation:

$$t(n) = \sum_{i=1}^n \Theta(1/i) = \Theta(H_n) = \Theta(\log n).$$

Finally, substituting $T(n) = (n+1)t(n)$ gives us a solution to the original recurrence: $T(n) = \Theta(n \log n)$.

Exercises

1. For each of the following recurrences, first *guess* an exact closed-form solution, and then prove your guess is correct. You are free to use any method you want to make your guess—unrolling the recurrence, writing out the first several values, induction proof template, recursion trees, annihilators, transformations, ‘It looks like that other one’, whatever—but please describe your method. All functions are from the non-negative integers to the reals. If it simplifies your solutions, express them in terms of Fibonacci numbers F_n , harmonic numbers H_n , binomial coefficients $\binom{n}{k}$, factorials $n!$, and/or the floor and ceiling functions $\lfloor x \rfloor$ and $\lceil x \rceil$.

(a) $A(n) = A(n - 1) + 1$, where $A(0) = 0$.

(b) $B(n) = \begin{cases} 0 & \text{if } n < 5 \\ B(n - 5) + 2 & \text{otherwise} \end{cases}$

(c) $C(n) = C(n - 1) + 2n - 1$, where $C(0) = 0$.

(d) $D(n) = D(n - 1) + \binom{n}{2}$, where $D(0) = 0$.

(e) $E(n) = E(n - 1) + 2^n$, where $E(0) = 0$.

(f) $F(n) = 3 \cdot F(n - 1)$, where $F(0) = 1$.

(g) $G(n) = \frac{G(n-1)}{G(n-2)}$, where $G(0) = 1$ and $G(1) = 2$. [Hint: This is easier than it looks.]

(h) $H(n) = H(n - 1) + 1/n$, where $H(0) = 0$.

(i) $I(n) = I(n - 2) + 3/n$, where $I(0) = I(1) = 0$. [Hint: Consider even and odd n separately.]

(j) $J(n) = J(n - 1)^2$, where $J(0) = 2$.

(k) $K(n) = K(\lfloor n/2 \rfloor) + 1$, where $K(0) = 0$.

(l) $L(n) = L(n - 1) + L(n - 2)$, where $L(0) = 2$ and $L(1) = 1$.

[Hint: Write the solution in terms of Fibonacci numbers.]

(m) $M(n) = M(n - 1) \cdot M(n - 2)$, where $M(0) = 2$ and $M(1) = 1$.

[Hint: Write the solution in terms of Fibonacci numbers.]

(n) $N(n) = 1 + \sum_{k=1}^n (N(k - 1) + N(n - k))$, where $N(0) = 1$.

(p) $P(n) = \sum_{k=0}^{n-1} (k \cdot P(k - 1))$, where $P(0) = 1$.

(q) $Q(n) = \frac{1}{2 - Q(n-1)}$, where $Q(0) = 0$.

(r) $R(n) = \max_{1 \leq k \leq n} \{R(k - 1) + R(n - k) + n\}$

(s) $S(n) = \max_{1 \leq k \leq n} \{S(k - 1) + S(n - k) + 1\}$

(t) $T(n) = \min_{1 \leq k \leq n} \{T(k - 1) + T(n - k) + n\}$

(u) $U(n) = \min_{1 \leq k \leq n} \{U(k - 1) + U(n - k) + 1\}$

(v) $V(n) = \max_{n/3 \leq k \leq 2n/3} \{V(k - 1) + V(n - k) + n\}$

2. Use recursion trees to solve each of the following recurrences.

- (a) $A(n) = 2A(n/4) + \sqrt{n}$
- (b) $B(n) = 2B(n/4) + n$
- (c) $C(n) = 2C(n/4) + n^2$
- (d) $D(n) = 3D(n/3) + \sqrt{n}$
- (e) $E(n) = 3E(n/3) + n$
- (f) $F(n) = 3F(n/3) + n^2$
- (g) $G(n) = 4G(n/2) + \sqrt{n}$
- (h) $H(n) = 4H(n/2) + n$
- (i) $I(n) = 4I(n/2) + n^2$
- (j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$
- (k) $K(n) = K(n/2) + K(n/3) + K(n/6) + n^2$
- (l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$
- (m) $M(n) = \sqrt{2n} M(\sqrt{2n}) + \sqrt{n}$
- (n) $N(n) = \sqrt{2n} N(\sqrt{2n}) + n$
- (p) $P(n) = \sqrt{2n} P(\sqrt{2n}) + n^2$
- (q) $Q(n) = Q(n - 3) + 8^n$ — Don't use annihilators!
- (r) $R(n) = 2R(n - 2) + 4^n$ — Don't use annihilators!
- (s) $S(n) = 4S(n - 1) + 2^n$ — Don't use annihilators!

3. Make up a bunch of linear recurrences and then solve them using annihilators.

4. Solve the following recurrences, using any tricks at your disposal.

- (a) $T(n) = \sum_{i=1}^{\lg n} T(n/2^i) + n$ [Hint: Assume n is a power of 2.]
- (b) More to come...

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

L Fibonacci Heaps

L.1 Mergeable Heaps

A *mergeable heap* is a data structure that stores a collection of keys¹ 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.

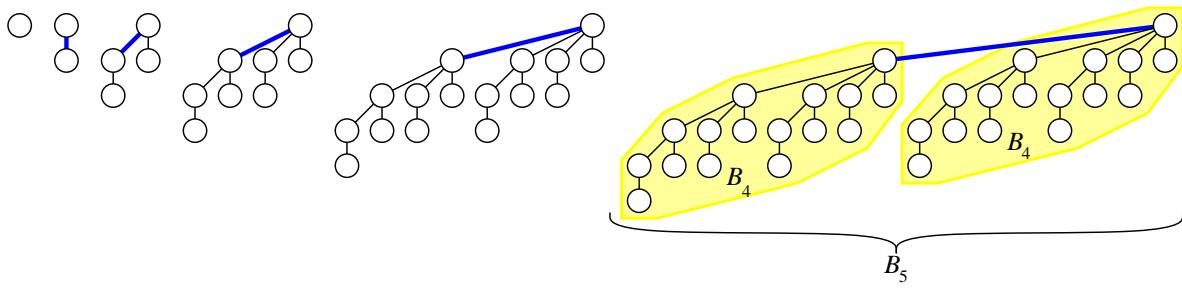
In this lecture, I’ll describe a data structure called a *Fibonacci heap* that supports `INSERTS`, `MERGES`, and `FINDMINS` in constant time, even in the worst case, and also handles `DELETEMIN` in $O(\log n)$ amortized time. That means that any sequence of n `INSERTS`, m `MERGES`, f `FINDMINS`, and d `DELETEMINS` takes $O(n + m + f + d \log n)$ time.

L.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.

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.

¹In the earlier lecture on treaps, I called these keys *priorities* to distinguish them from search keys.

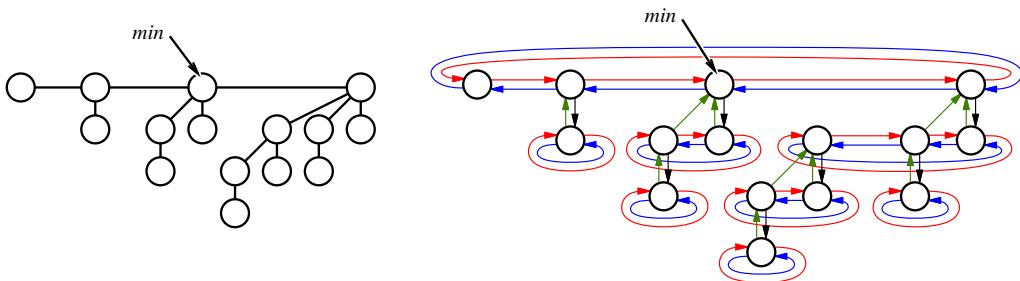


Binomial trees of order 0 through 5.

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, \dots, 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.

L.3 Operations on Fibonacci Heaps

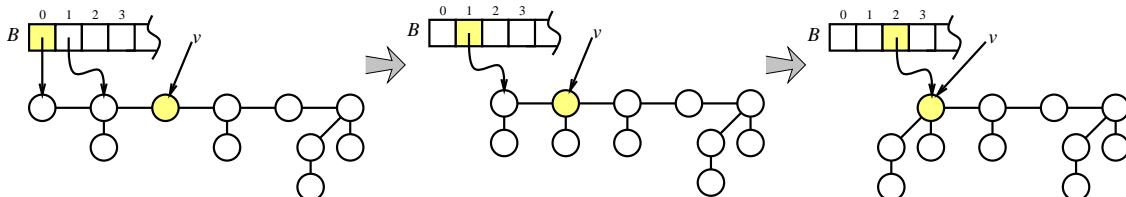
The `INSERT`, `MERGE`, and `FINDMIN` algorithms for Fibonacci heaps are exactly like the corresponding algorithms for linked lists. Since we maintain a pointer to the minimum key, `FINDMIN` is trivial. To insert a new key, we add a single node (which we should think of as a B_0) to the root list and (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 ← some node in the root list
  for  $i \leftarrow 0$  to  $\lfloor \lg n \rfloor$ 
     $B[i] \leftarrow \text{NULL}$ 
  for all nodes  $v$  in the root list
    parent( $v$ ) ← NULL (*) 
    if  $\text{key}(\text{newmin}) > \text{key}(v)$ 
      newmin ←  $v$ 
    MERGEDUPES( $v$ )
```

```
MERGEDUPES( $v$ ):
   $w \leftarrow B[\text{deg}(v)]$ 
  while  $w \neq \text{NULL}$ 
     $B[\text{deg}(v)] \leftarrow \text{NULL}$ 
    if  $\text{key}(v) \leq \text{key}(w)$ 
      swap  $v \leftrightarrow w$ 
    remove  $w$  from the root list (**)
    link  $w$  to  $v$ 
     $w \leftarrow B[\text{deg}(v)]$ 
     $B[\text{deg}(v)] \leftarrow v$ 
```



`MergeDups(v)`, ensuring that no earlier root has the same degree as v .

Notice 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 (*) is executed once for every node v on the root list, and line (**) 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`, 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 $\lfloor \lg n \rfloor$.

L.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 $r + \deg(\min)$, 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 $O(\log n)$, 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.

L.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 .

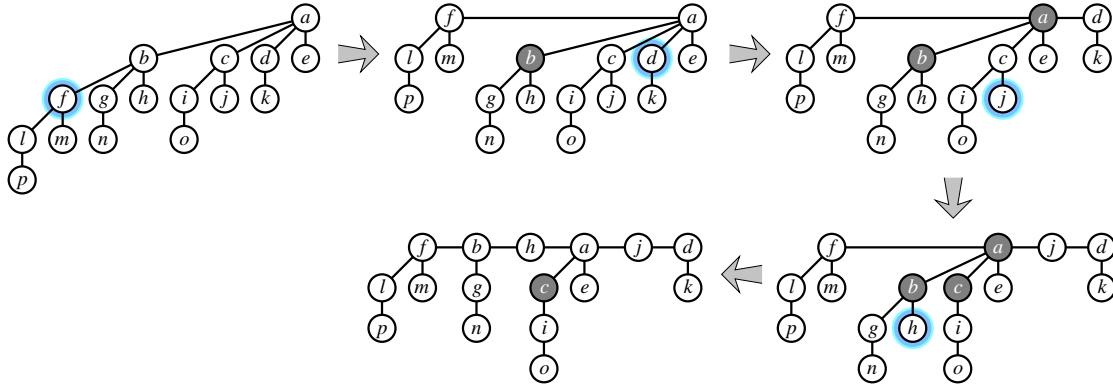
In order to enforce the second rule, we now *mark* certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well. Whenever we promote a marked node, we unmark 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.

<u>DECREASEKEY(v, k):</u>
$key(v) \leftarrow k$
update the pointer to the smallest key
PROMOTE(v)

<u>PROMOTE(v):</u>
unmark v
if $parent(v) \neq \text{NULL}$
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 $O(1 + \#consecutive marked ancestors of v)$. 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, DECREASEKEY runs in $\Theta(n)$ 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

$$t = 1 + \#consecutive marked ancestors of $v$$$

and if we set $\Phi = m$, the increase in potential is

$$m' - m \leq 1 - \#consecutive marked ancestors of v .$$

Since $t + \Delta\Phi \leq 2$, the amortized cost of DECREASEKEY is $O(1)$.

L.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 i th 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 i th 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.

L.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 our 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:

$$\Phi = r + 2m$$

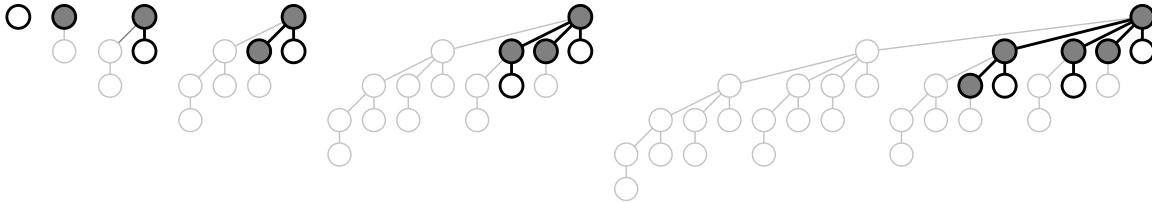
To see that this potential function gives us good amortized bounds for every Fibonacci heap operation, let's add two more columns to our table.

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!)

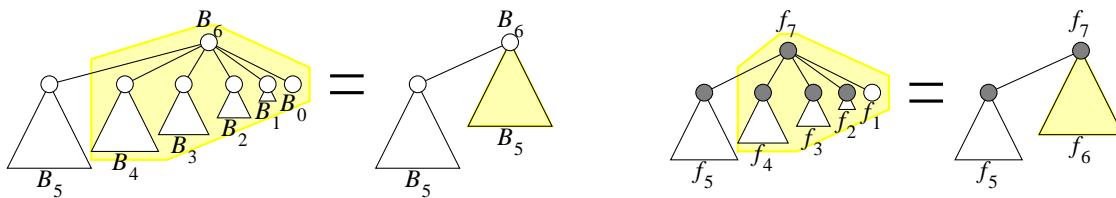
L.8 Fibonacci Trees

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, \dots, 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, \dots, 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 k th 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 $0 \leq d \leq \lfloor k/2 \rfloor - 1$.
- f_k has F_{k-2h-1} nodes with height h , for all $0 \leq h \leq \lfloor k/2 \rfloor - 1$, 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.

*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,
 Whoopie! 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.

— Kronecker

M Number Theoretic Algorithms

M.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:

$$q = \left\lfloor \frac{n}{d} \right\rfloor \quad \text{and} \quad r = n \bmod d = n - d \left\lfloor \frac{n}{d} \right\rfloor.$$

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.

Lemma 1. $\gcd(m, n)$ is the smallest positive integer of the form $am + bn$.

¹The 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!

²Do not use the notation (m, n) for greatest common divisor, unless you want Notation Kitty to visit you in the middle of the night and remove your eyes while you sleep. Nice kitty.

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. \square

M.2 Euclid's GCD Algorithm

We can compute the greatest common divisor of two given integers by recursively applying two simple observations:

$$\gcd(m, n) = \gcd(m, n - m) \quad \text{and} \quad \gcd(n, 0) = n$$

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  $n > 0$ 
     $m \leftarrow m \bmod n$  (*)  

    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 $m \geq F_{k+1} + F_k = F_{k+2}$. \square

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, $k \leq \log_\phi(\sqrt{5}(m + 1)) - 2 = O(\log m)$. \square

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 $O(\log^2 m)$. We can make the algorithm even faster by using a fast integer division algorithm (based on FFTs, for example).

M.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:

$$\mathbb{Z}_n = \{0, 1, 2, \dots, n - 1\}$$

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

³However, 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 | m
        return n
    else
        return n mod (m mod 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

$$(2, 3) + (1, 3) = (2 + 1 \bmod 3, 3 + 3 \bmod 5) = (0, 1) = 6 = 21 \bmod 15 = (8 + 13) \bmod 15.$$

$$(2, 3) \cdot (1, 3) = (2 \cdot 1 \bmod 3, 3 \cdot 3 \bmod 5) = (2, 4) = 14 = 104 \bmod 15 = (8 \cdot 13) \bmod 15.$$

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. \square

We can extend the Chinese remainder theorem inductively as follows:

⁴after 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.

⁵The author of *The Art of War*, who had the same name, lived more than 500 years earlier.

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 of \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.

M.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 $\text{Root}(n, i)$ 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  $(r + 2^\ell)^i \leq n$ 
         $r \leftarrow r + 2^\ell$ 
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 [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.

⁶This 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.

⁷This 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 $\langle 1, g, g^2, \dots \rangle$ 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 $g^x \equiv g^y \pmod{n}$. \square

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 an 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 $x = g^{\phi(p^e)/2}$. Because $x = n - 1$ is also a solution to the original equation, we must have $g^{\phi(p^e)/2} \equiv n - 1 \pmod{n}$. \square

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 .

M.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  $s$ 
         $a \leftarrow \text{RANDOM}(2, n - 2)$ 
        if  $\text{EUCLIDGCD}(a, n) \neq 1$ 
            return COMPOSITE!      ((obviously!))
         $x_0 \leftarrow a^u \pmod{n}$ 
        for  $j \leftarrow 1$  to  $t$ 
             $x_j \leftarrow x_{j-1}^2 \pmod{n}$ 
            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  $x_t \neq 1$           (( $x_t = a^{n-1} \pmod{n}$ ))
                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 $a^u \pmod{n}$ 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 $O(s \log^3 n)$. 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 $1 - 2^{-s}$.*

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:

$$F = \{a \in \mathbb{Z}_n^* \mid a^{n-1} \equiv 1 \pmod{n}\}.$$

⁸If 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.

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 ($\bmod n$) 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. \square

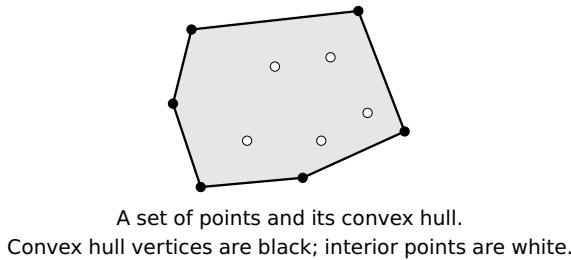
N Convex Hulls

N.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.



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 i th 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.

N.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 $a < c$

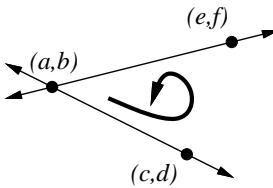
and $a < f$. Then the three points are in counterclockwise order if and only if the line $\overrightarrow{(a,b)(c,d)}$ is less than the slope of the line $\overrightarrow{(a,b)(e,f)}$:

$$\text{counterclockwise} \iff \frac{d-b}{c-a} < \frac{f-b}{e-a}$$

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 clockwise 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:

$$\boxed{\text{counterclockwise} \iff \begin{vmatrix} 1 & a & b \\ 1 & c & d \\ 1 & e & f \end{vmatrix} > 0}$$

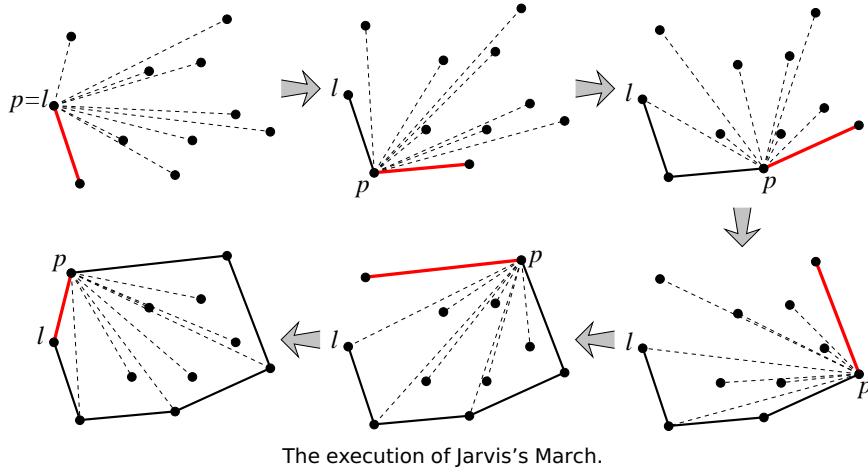
All three boxed expressions are algebraically identical.

This counterclockwise test plays *exactly* the same role in convex hull algorithms as comparisons play in sorting algorithms. Computing the convex hull of three points is analogous to sorting two numbers: either they're in the correct order or in the opposite order.

N.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 (that is, the point whose x -coordinate is smallest), because we know that the left most point must be a convex hull vertex. Finding ℓ clearly takes linear time.



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.

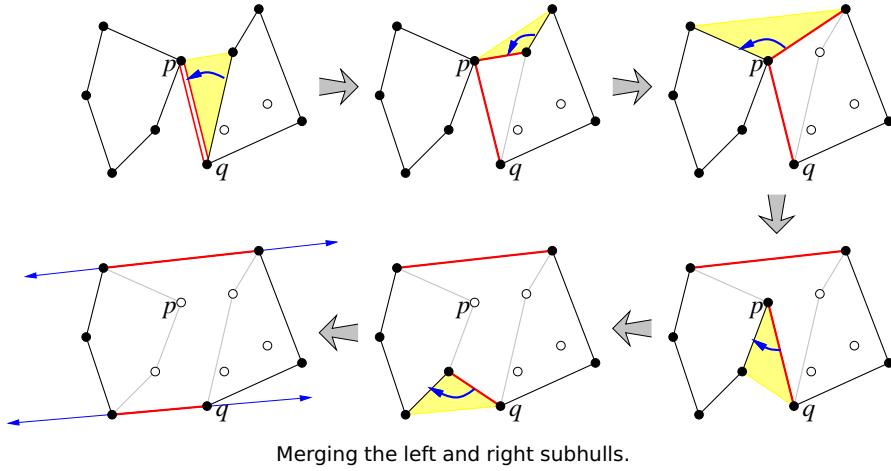
N.4 Divide and Conquer (Splitting)

The behavior of Jarvis's march 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.

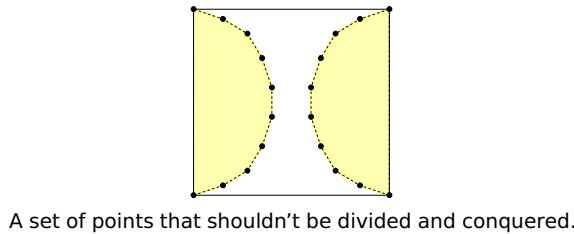
The merge step requires a little explanation. We start by connecting the two hulls with a line segment between the rightmost point of the hull of L with the leftmost point of the hull of R . Call these points p and q , respectively. (Yes, it's the same p .) Actually, let's add two copies of the segment \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.



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 $O(n \log n)$.

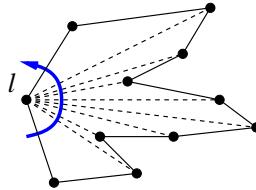
There are inputs where this algorithm is clearly wasteful (at least, clearly to us). If we're really unlucky, we'll spend a long time putting together the subhulls, only to throw almost everything away during the merge step. Thus, this divide-and-conquer algorithm is *not* output sensitive.



N.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.

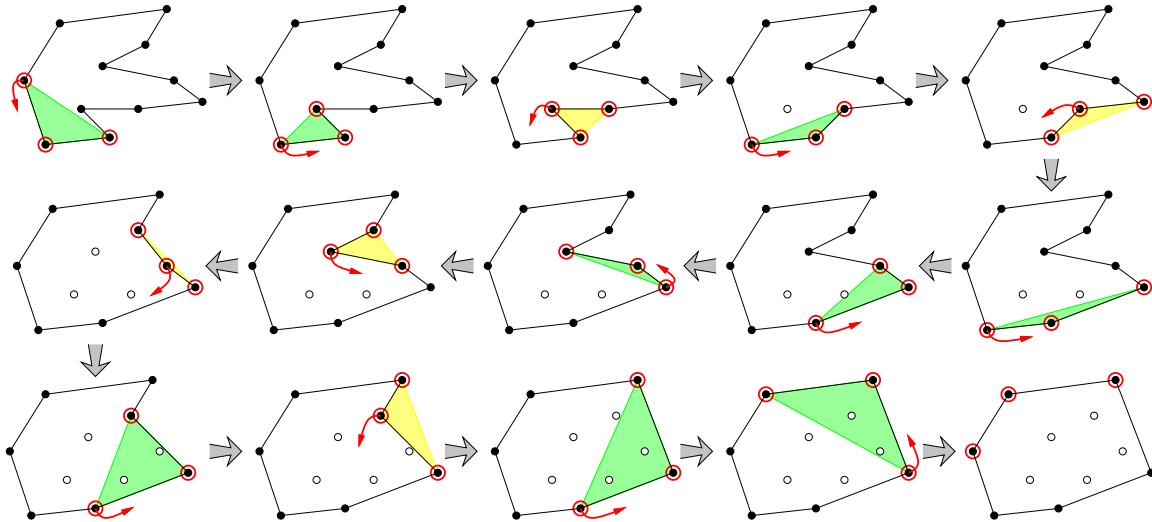
We start Graham's scan by finding the leftmost point ℓ , just as in Jarvis's march. Then we sort the points in counterclockwise order around ℓ . We can do this in $O(n \log n)$ time with any comparison-based sorting algorithm (quicksort, mergesort, heapsort, etc.). To compare two points p and q , we check whether the triple ℓ, p, q is oriented clockwise or counterclockwise. Once the points are sorted, we connect 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 .



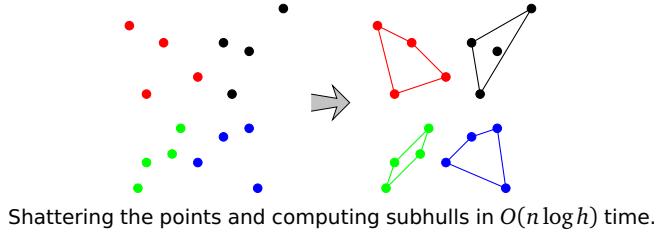
The ‘three-penny’ scanning phase of Graham’s scan.

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.

N.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 arbitrary¹ 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.

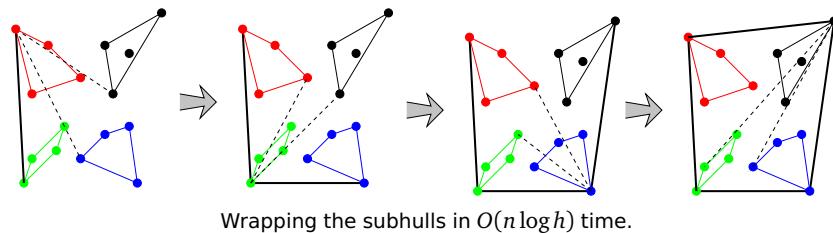


Shattering the points and computing subhulls in $O(n \log h)$ time.

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 that 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.

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 i th iteration, we have $h^* = 3^{2^i}$. In

¹In 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!

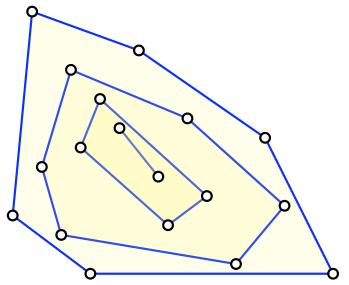
the final iteration, $h^* < h^2$, so the last iteration takes $O(n \log h^*) = O(n \log h^2) = O(n \log h)$ time. The total running time of Chan's algorithm is given by the sum

$$\sum_{i=1}^k O(n \log 3^{2^i}) = O(n) \cdot \sum_{i=1}^k 2^i$$

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

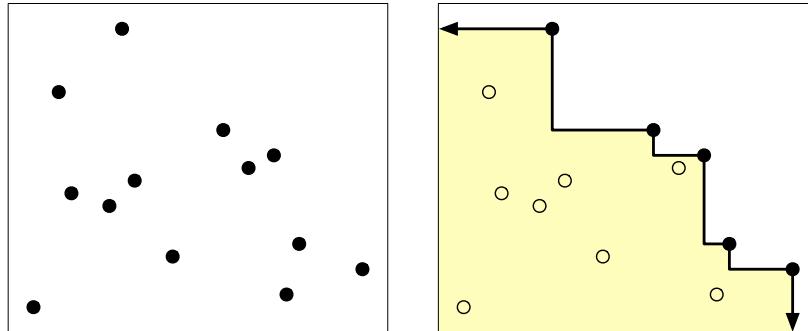
1. The *convex layers* of a point set X are defined by repeatedly computing the convex hull of X and removing its vertices from X , until X is empty.
 - (a) Describe an algorithm to compute the convex layers of a given set of n points in the plane in $O(n^2)$ time.
 - *(b) Describe an algorithm to compute 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 it 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)

O Line Segment Intersection

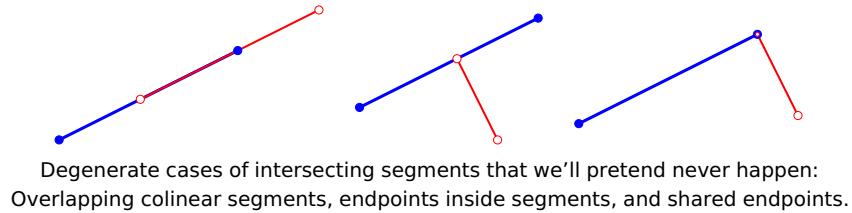
O.1 Introduction

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*.

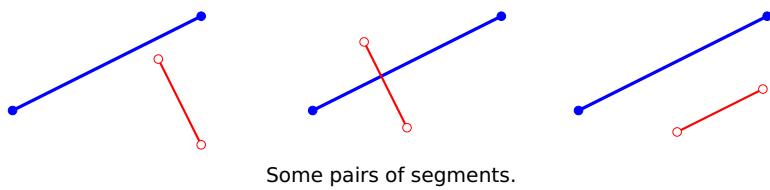
- No three endpoints lie on a common line.
- No two endpoints have the same x -coordinate. In particular, no segment is vertical, no segment is just a point, and no two segments share an endpoint.

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!



O.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.



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  $[\text{CCW}(a, c, d) \neq \text{CCW}(b, c, d)] \wedge [\text{CCW}(a, b, c) \neq \text{CCW}(a, b, d)]$ 
```

O.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.

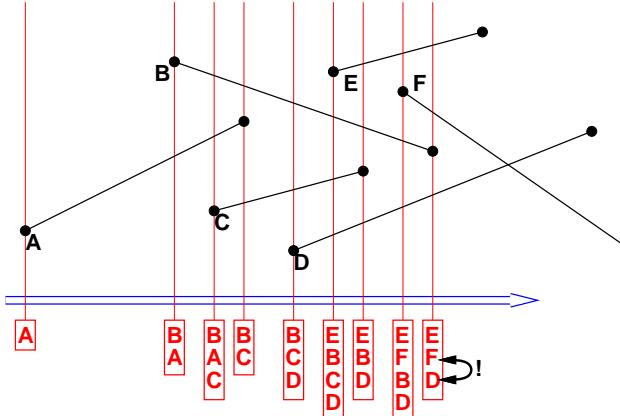
The sweep line algorithm begins by sorting the $2n$ segment endpoints from left to right by comparing their x -coordinates, in $O(n \log n)$ time. The algorithm then moves the sweep line from left to right, stopping at each endpoint.

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.
- 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.

¹Actually, 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.



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 $\text{left}(Y), \text{right}(Y), \text{left}(X)$ 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 intersect, 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 n th 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:

- $\text{label}[i]$ is the label of the i th leftmost endpoint. I’ll use indices into the input array S as the labels, so the i th vertex is an endpoint of $S[\text{label}[i]]$.
- $\text{isleft}[i]$ is TRUE if the i th leftmost endpoint is a left endpoint and FALSE if it’s a right endpoint.

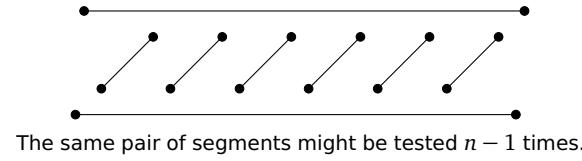
The functions `INSERT`, `DELETE`, `PREDECESSOR`, and `SUCCESSOR` modify or search through the sorted label sequence. Finally, `INTERSECT` tests whether two line segments intersect.

```

ANYINTERSECTIONS( $S[1..n]$ ):
    sort the endpoints of  $S$  from left to right
    create an empty label sequence
    for  $i \leftarrow 1$  to  $2n$ 
         $\ell \leftarrow \text{label}[i]$ 
        if  $\text{isleft}[i]$ 
            INSERT( $\ell$ )
            if INTERSECT( $S[\ell], S[\text{SUCCESSOR}(\ell)]$ )
                return TRUE
            if INTERSECT( $S[\ell], S[\text{PREDECESSOR}(\ell)]$ )
                return TRUE
        else
            if INTERSECT( $S[\text{SUCCESSOR}(\ell)], S[\text{PREDECESSOR}(\ell)]$ )
                return TRUE
            DELETE( $\text{label}[i]$ )
    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.

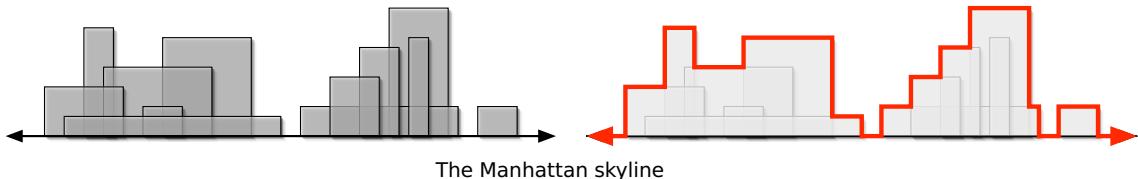


Exercises

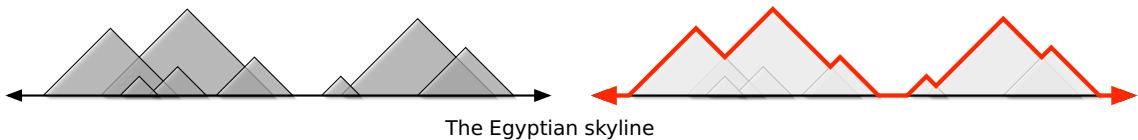
1. Let X be a set of n rectangles in the plane, each specified by a left and right x -coordinate and a top and bottom y -coordinate. Thus, the input consists of four arrays $L[1..n]$, $R[1..n]$, $T[1..n]$, and $B[1..n]$, where $L[i] < R[i]$ and $T[i] > B[i]$ for all i .
 - (a) Describe and analyze an 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.



- (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.



- (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 Montesquieu (1721)

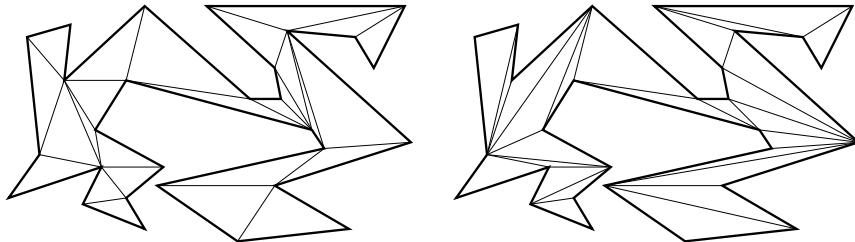
Down with Euclid! Death to triangles!

— Jean Dieudonné (1959)

P Polygon Triangulation

P.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.

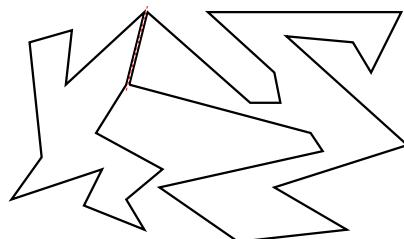


Two triangulations of the same polygon.

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.

P.2 Existence and Complexity

If you play around with a few examples, you quickly discover that every triangulation of an n -sided polygon 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 $n - k - 1$ triangles. Putting the two pieces back together, we have a total of $(k - 1) + (n - k - 1) = n - 2$ triangles.



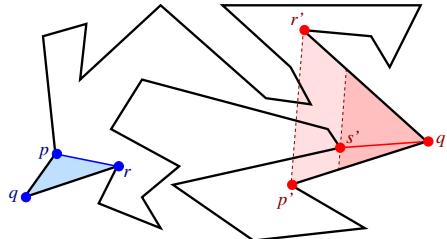
Breaking a polygon into two smaller polygons with a diagonal.

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 lowest one.) If \overline{pr} is a diagonal, we're done; in this case, we say that the triangle Δpqr is an *ear*.

If \overline{pr} is not a diagonal, then Δpqr must contain another vertex of the polygon. Out of all the vertices inside Δpqr , 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 s is the first vertex that the line hits. Then the line segment \overline{qs} is a diagonal. \square



The leftmost vertex q is the tip of an ear, so pr is a diagonal.

The rightmost vertex q' is not, since $\Delta p'q'r'$ contains three other vertices. In this case, $q's'$ is a diagonal.

P.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 Δpqr 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 Δpqs ; 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 points are in counterclockwise order, and to determine if a point is inside a triangle.

```

    <<Return twice the signed area of  $\Delta P[i]P[j]P[k]$ >>
    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  $\Delta P[p]P[q]P[r]$ ?>>
    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  $P[i].x < P[q].x$ 
             $q \leftarrow i$ 
     $p \leftarrow q - 1 \bmod n$ 
     $r \leftarrow q + 1 \bmod n$ 
     $ear \leftarrow \text{TRUE}$ 
     $s \leftarrow p$ 
    for  $i \leftarrow 1$  to  $n$ 
        if  $i \leq p$  and  $i \neq q$  and  $i \neq r$  and INSIDE( $i, p, q, r$ )
             $ear \leftarrow \text{FALSE}$ 
            if AREA( $i, r, p$ )  $>$  AREA( $s, r, p$ )
                 $s \leftarrow i$ 
        if  $ear = \text{TRUE}$ 
            return  $(p, r)$ 
        else
            return  $(q, s)$ 

```

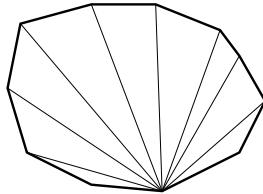
Once we have a diagonal, we can recursively triangulate the two pieces. The worst-case running time of this algorithm satisfies almost the same recurrence as quicksort:

$$T(n) \leq \max_{2 \leq k \leq n-2} T(k+1) + T(n-k+1) + O(n).$$

So we can now triangulate any polygon in $O(n^2)$ time.

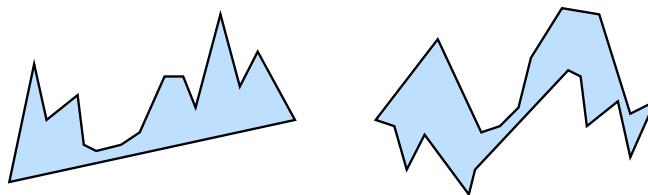
P.4 Faster Special Cases

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.



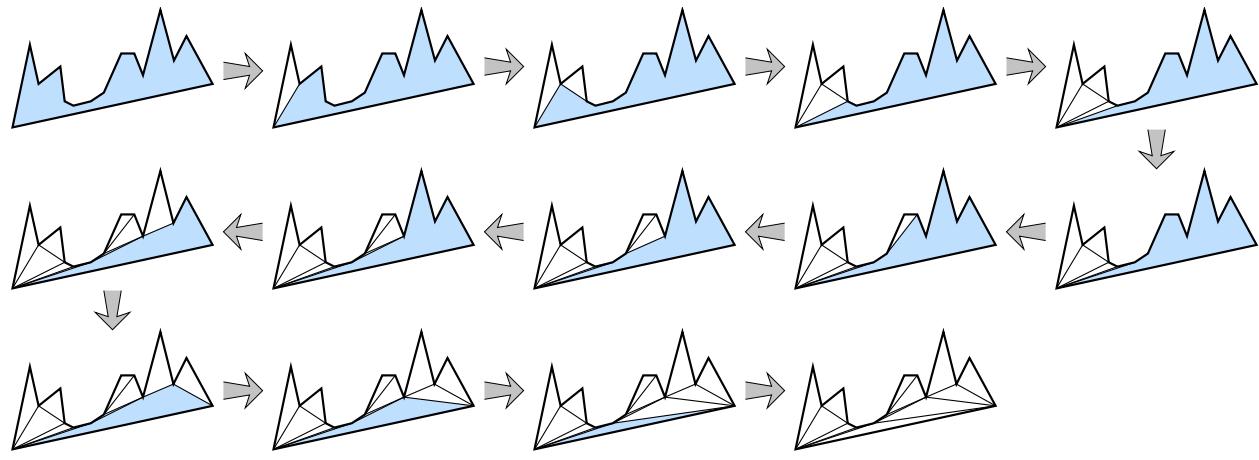
Triangulating a convex polygon is easy.

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.



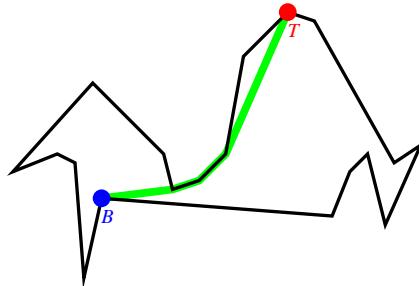
A monotone mountain and a monotone non-mountain.

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.



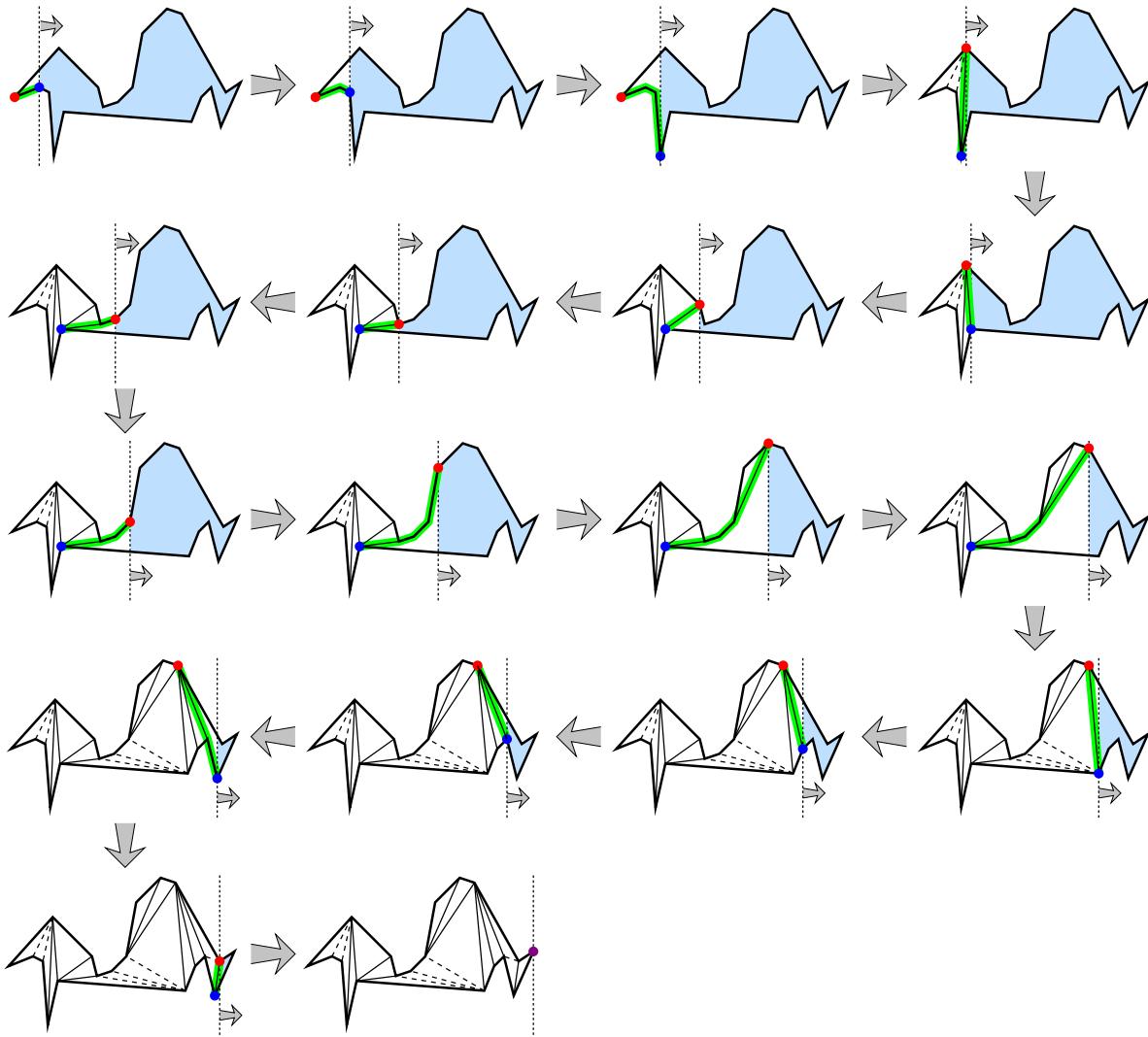
Triangulating a monotone mountain. (Some of the triangles are very thin.)

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 should 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 straightens 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.

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)

Q Reductions

Q.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 i th 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

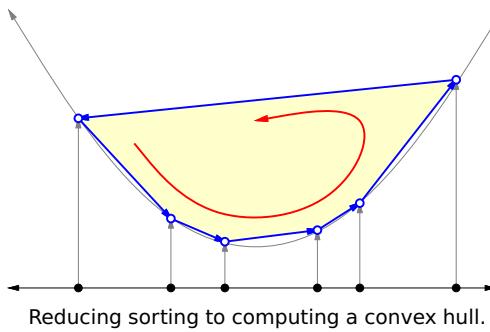
The rest of this lecture assumes some familiarity with computational geometry.

Q.2 Sorting to Convex Hulls

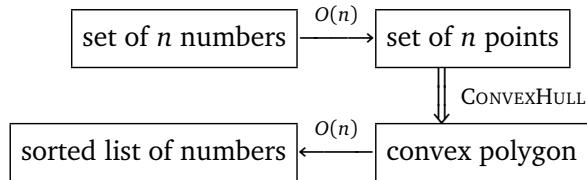
Here's a slightly less trivial example. Suppose we want to prove a lower bound for the problem of computing the convex hull of a set of n points in the plane. To do this, we demonstrate a reduction from sorting to convex hulls.

To sort a list of n numbers $\{a, b, c, \dots\}$, we first transform it into a set of n points $\{(a, a^2), (b, b^2), (c, c^2), \dots\}$. 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.

¹We 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 $\lceil \log_3(n!^2) \rceil = \Omega(n \log n)$.



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:



(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.

Q.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 binary² 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.

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!

²or ternary, if we allow colinear triples of points

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.

Q.4 Element Uniqueness (A Bad Example)

The *element uniqueness* problem asks, given a list of n numbers x_1, x_2, \dots, 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 hand³:

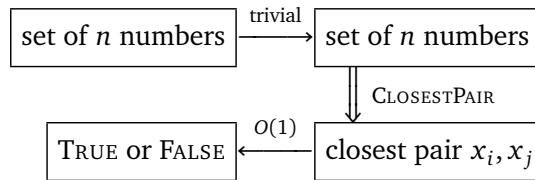
To prove that problem A is harder than problem B, reduce B to A.

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, \dots, 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.

Q.5 Closest Pair

The simplest version of the *closest pair* problem asks, given a list of n numbers x_1, x_2, \dots, 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.

Q.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.

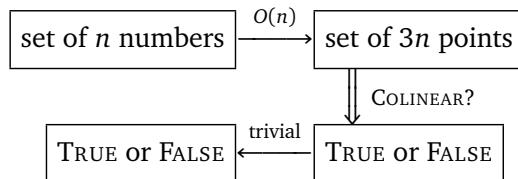
³right under all those rules about logarithms, geometric series, and recurrences

Here's an example. The problem 3SUM asks, given a sequence of n numbers x_1, \dots, 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, if 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:



$$T_{\text{3SUM}}(n) \leq T_{\text{colinear}}(3n) + O(n) \implies T_{\text{colinear}}(n) \geq T_{\text{3SUM}}(n/3) - O(n).$$

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 Overmars⁵ 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.

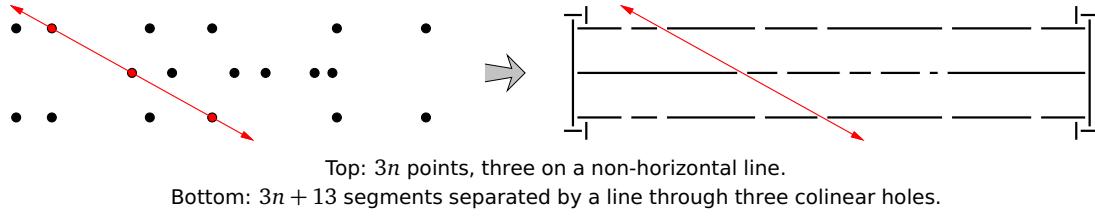
Q.7 ... to Segment Splitting ...

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?

⁴The $\Omega(n^2)$ 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.

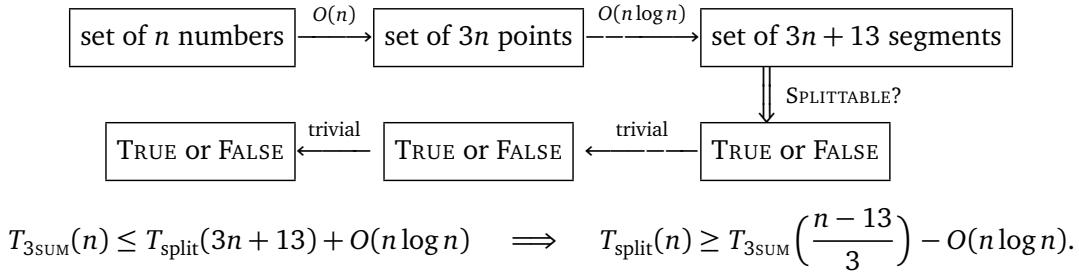
⁵A. Gajentaan and M. Overmars, On a class of $O(n^2)$ problems in computational geometry, *Comput. Geom. Theory Appl.* 5:165–185, 1995. <ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1993-15.ps.gz>

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.



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.

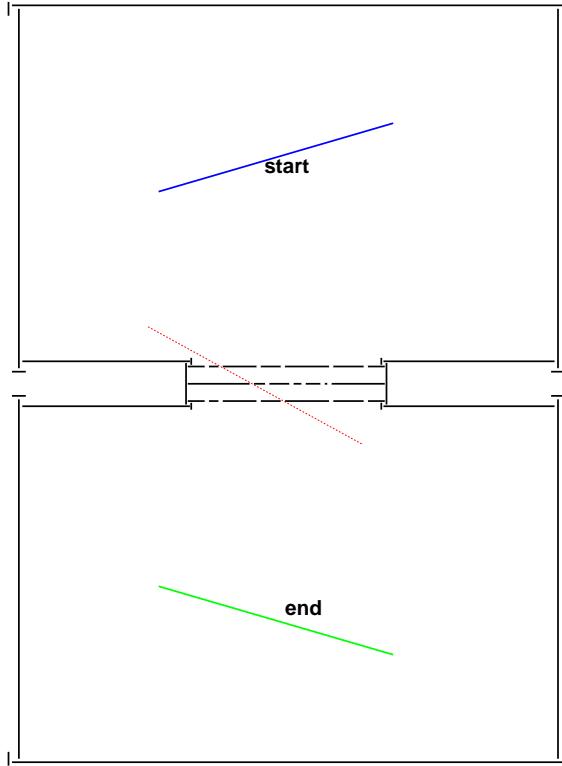
Thus, we have a valid reduction from 3SUM to segment splitting (by way of colinearity):



Q.8 ... to Motion Planning

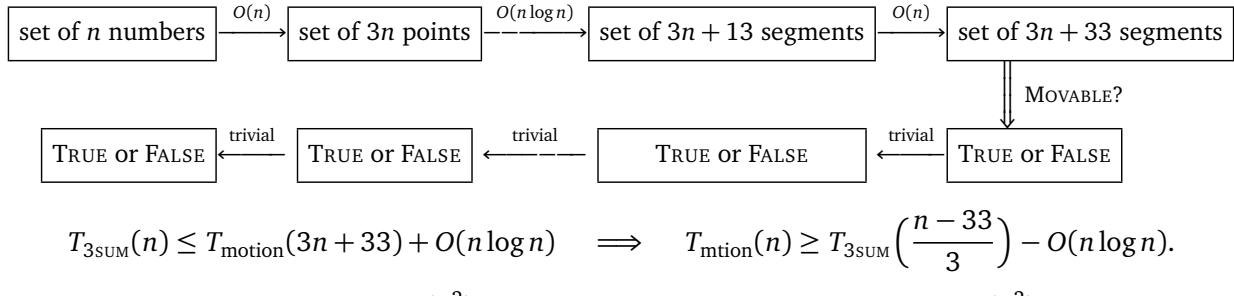
Finally, suppose we want to know whether a robot can move from one position and location to another. To make things simple, we'll assume that the robot is just a line segment, and the environment in which the robot moves is also made up of non-intersecting line segments. Given an initial position and 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):

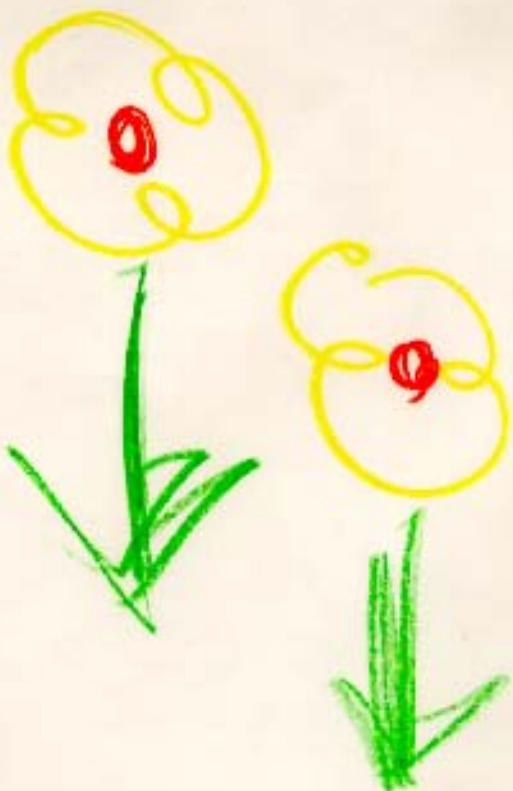


Thus, a sufficiently powerful $\Omega(n^2)$ lower bound for 3SUM would imply an $\Omega(n^2)$ lower bound for motion planning as well. The existing $\Omega(n^2)$ lower bound for 3SUM does *not* imply a lower bound for this problem — the model of computation in which the lower bound holds is too weak 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.

CS 373

Name: Johnny
alias: pikachu

HW1



1

Trees have no cycles but must be connected.
Tournaments are cliques with their edges directed.
Hamilton circuits, Eulerian paths,
These are a few of my favorite graphs.

Matchings and bicliques and blossoms and bases,
Kempe chains, hypercubes, forests, and faces,
AKS networks that split into halves,
These are a few of my favorite graphs.

Short paths of co-authors leading to Erdos,
Large neural networks that translate from Kurdish,
Finite projective planes - they make me laugh
These are a few of my favorite graphs.

Chorus: Propositions, corollaries,
Problems that are starred,
I simply remember my favorite graphs
And then they don't seem so hard.

If there's no K_5 or $K_{3,3}$ minor,
Old Kuratowski says it'll be 'plinor'.
Four's enough colors if there are no gaffes!
These are a few of my favorite graphs.

Quadrangles, thrackles, and triangulations,
Minor-closed families and sparsifications,
Voronoi diagrams found on giraffes,
These are a few of my favorite graphs.

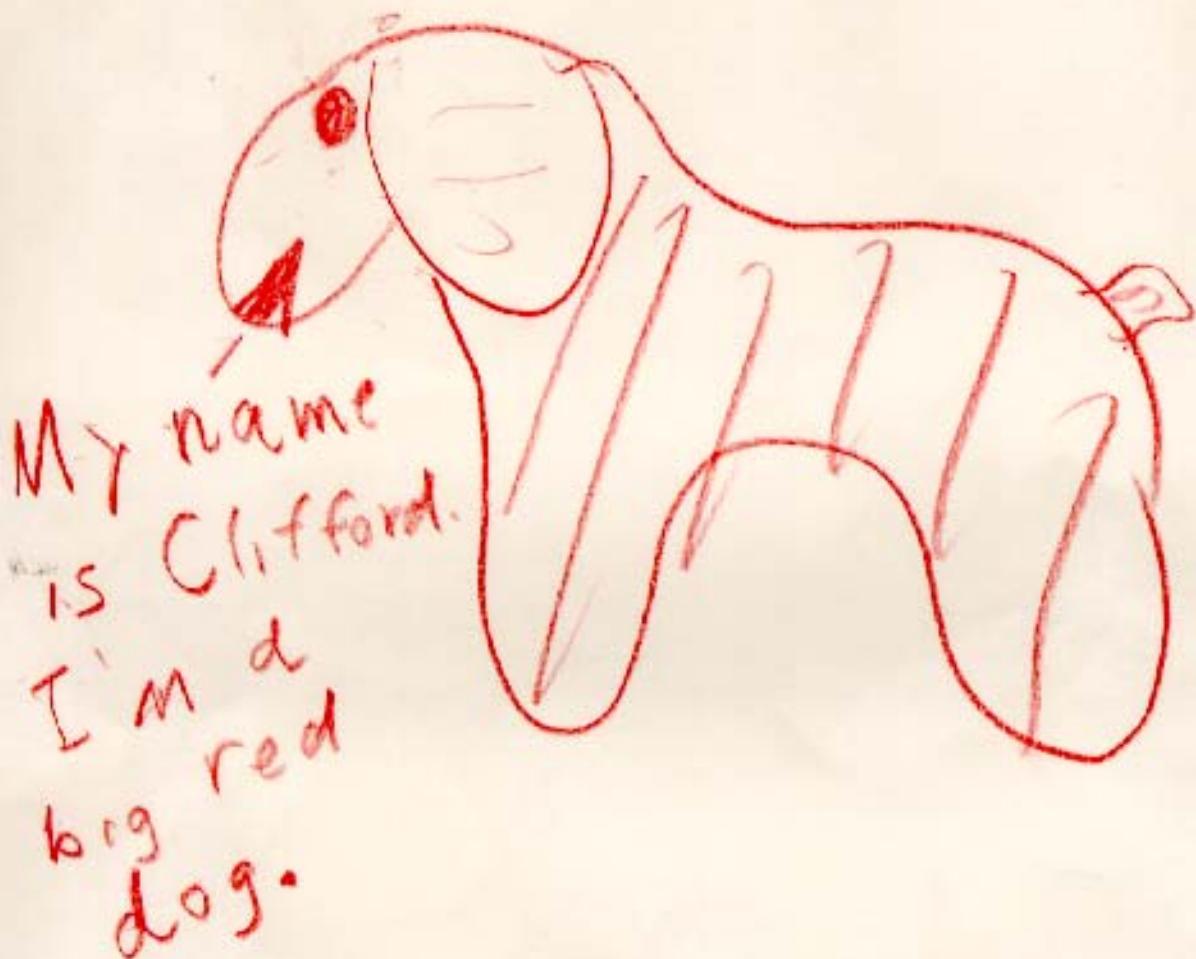
Chorus: Propositions, corollaries,
When they're just too deep,
I simply remember my favorite graphs
And then I go right to sleep.

Look! I
minimized
the zlop of
my favourite
graph song.



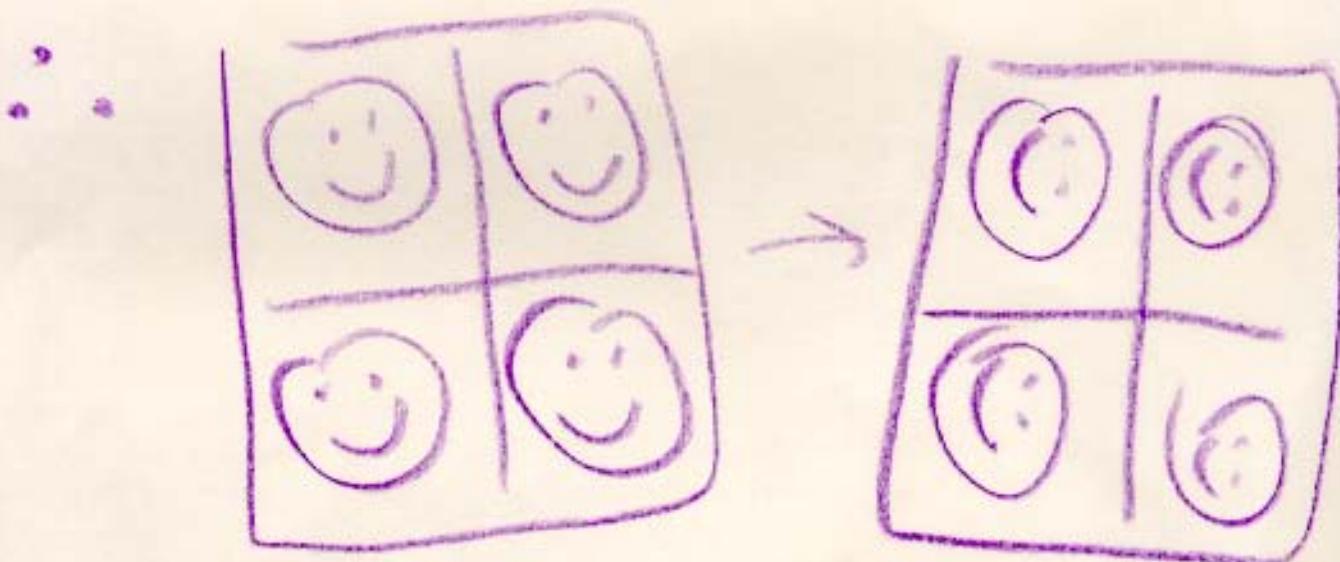
2.

Mamma always said:
2tupid is as stupid does.



3. This problem is the work of
Satan. It is very scan.
I was too afraid to do it.
Sorry.

4. I betcha that:



QED.

5. I am running this
algorithm with MGD. Will
report results...

much beer drinking
more analysis needed

in Ign is like a cheese log
if I'm hungry, call it
and have a slice
beer.

CS 373: Combinatorial Algorithms, Spring 1999

<http://www-courses.cs.uiuc.edu/cs373>

Homework 0 (due January 26, 1999 by the beginning of class)

Name:	
Net ID:	Alias:

Neatly print your name (first name first, with no comma), your network ID, and a short alias into the boxes above. Do not *sign* your name. Do not write your Social Security number. Staple this sheet of paper to the top of your homework. Grades will be listed on the course web site by alias, so your alias should not resemble your name (or your Net ID). If you do not give yourself an alias, you will be stuck with one we give you, no matter how much you hate it.

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

This homework tests your familiarity with the prerequisite material from CS 225 and CS 273 (and *their* prerequisites)—many of these problems appeared on homeworks and/or exams in those classes—primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.**

Undergrad/.75U Grad/1U Grad Problems

▶1. [173/273]

- (a) Prove that any positive integer can be written as the sum of distinct powers of 2. (For example: $42 = 2^5 + 2^3 + 2^1$, $25 = 2^4 + 2^3 + 2^0$, $17 = 2^4 + 2^0$.)
- (b) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. (For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.)
- (c) Prove that *any* integer can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. (For example: $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.)

▶2. [225/273] Sort the following functions from asymptotically smallest to largest, indicating ties if there are any:

n , $\lg n$, $\lg \lg^* n$, $\lg^* \lg n$, $\lg^* n$, $n \lg n$, $\lg(n \lg n)$, $n^{n/\lg n}$, $n^{\lg n}$, $(\lg n)^n$, $(\lg n)^{\lg n}$, $2^{\sqrt{\lg n \lg \lg n}}$, 2^n , $n^{\lg \lg n}$, $\sqrt[1000]{n}$, $(1 + \frac{1}{1000})^n$, $(1 - \frac{1}{1000})^n$, $\lg^{1000} n$, $\lg^{(1000)} n$, $\log_{1000} n$, $\lg^n 1000$, 1 .

[To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted as follows: $n \ll n^2 \equiv \binom{n}{2} \ll n^3$.]

3. [273/225] Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable (nontrivial) base cases. Extra credit will be given for more exact solutions.

- (a) $A(n) = A(n/2) + n$
- (b) $B(n) = 2B(n/2) + n$
- (c) $C(n) = 3C(n/2) + n$
- (d) $D(n) = \max_{n/3 < k < 2n/3} (D(k) + D(n - k) + n)$
- (e) $E(n) = \min_{0 < k < n} (E(k) + E(n - k) + 1)$
- (f) $F(n) = 4F(\lfloor n/2 \rfloor + 5) + n$
- (g) $G(n) = G(n - 1) + 1/n$
- *(h) $H(n) = H(n/2) + H(n/4) + H(n/6) + H(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.]
- *(i) $I(n) = 2I(n/2) + n/\lg n$
- *(j) $J(n) = \frac{J(n - 1)}{J(n - 2)}$

- 4. [273] Alice and Bob each have a fair n -sided die. Alice rolls her die once. Bob then repeatedly throws his die until the number he rolls is at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. (If you have to appeal to “intuition” or “common sense”, your answer is probably wrong.)

- 5. [225] George has a 26-node binary tree, with each node labeled by a unique letter of the alphabet. The preorder and postorder sequences of nodes are as follows:

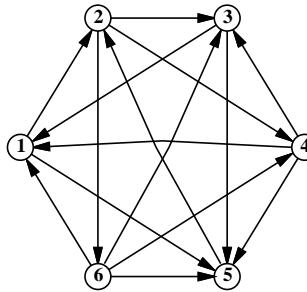
preorder: M N H C R S K W T G D X I Y A J P O E Z V B U L Q F
 postorder: C W T K S G R H D N A O E P J Y Z I B Q L F U V X M

Draw George’s binary tree.

Only 1U Grad Problems

- *1. [225/273] A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once.

Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

Practice Problems

1. [173/273] Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m .

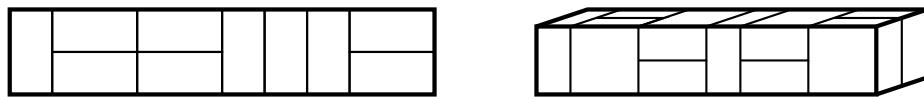
- (a) F_n is even if and only if n is divisible by 3.
- (b) $\sum_{i=0}^n F_i = F_{n+2} - 1$
- (c) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$
- ★(d) If n is an integer multiple of m , then F_n is an integer multiple of F_m .

2. [225/273]

- (a) Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor} / n = \Theta(n)$.
- (b) Is $2^{\lfloor \lg n \rfloor} = \Theta(2^{\lceil \lg n \rceil})$? Justify your answer.
- (c) Is $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta(2^{2^{\lceil \lg \lg n \rceil}})$? Justify your answer.

3. [273]

- (a) A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos?
- (b) A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

4. [273] Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ..., 52 of clubs. (They're big cards.) Penn shuffles the deck until each each of the $52!$ possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.

- (a) On average, how many cards does Penn give Teller?
- (b) On average, what is the smallest-numbered card that Penn gives Teller?
- *(c) On average, what is the largest-numbered card that Penn gives Teller?

Prove that your answers are correct. (If you have to appeal to “intuition” or “common sense”, your answers are probably wrong.) [Hint: Solve for an n -card deck, and then set n to 52.]

5. [273/225] Prove that for any nonnegative parameters a and b , the following algorithms terminate and produce identical output.

SLOWEUCLID(a, b) :

if $b > a$
 return SLOWEUCLID(b, a)
else if $b == 0$
 return a
else
 return SLOWEUCLID($a, b - a$)

FASTEUCLID(a, b) :

if $b == 0$
 return a
else
 return FASTEUCLID($b, a \bmod b$)

CS 373: Combinatorial Algorithms, Spring 1999

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 1 (due February 9, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
2. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
3. Justify the correctness of your algorithm, including termination if that is not obvious.
4. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

- ▶1. Consider the following sorting algorithm:

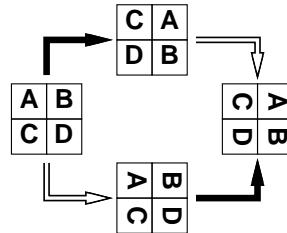
```

STUPIDSORT( $A[0..n - 1]$ ) :
  if  $n = 2$  and  $A[0] > A[1]$ 
    swap  $A[0] \leftrightarrow A[1]$ 
  else if  $n > 2$ 
     $m = \lceil 2n/3 \rceil$ 
    STUPIDSORT( $A[0..m - 1]$ )
    STUPIDSORT( $A[n - m..n - 1]$ )
    STUPIDSORT( $A[0..m - 1]$ )
  
```

- (a) Prove that STUPIDSORT actually sorts its input.
- (b) Would the algorithm still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.
- (c) State a recurrence (including the base case(s)) for the number of comparisons executed by STUPIDSORT.

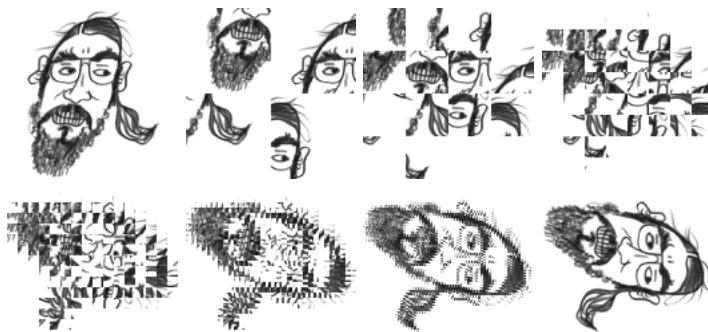
- (d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.] Does the algorithm deserve its name?
- *(e) Show that the number of *swaps* executed by STUPIDSORT is at most $\binom{n}{2}$.
- 2. Some graphics hardware includes support for an operation called *blit*, or **block transfer**, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixelmap 90° clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
Black arrows indicate blitting the blocks into place.
White arrows indicate recursively rotating the blocks.

The following sequence of pictures shows the first algorithm (blit then recurse) in action.



In the following questions, assume n is a power of two.

- Prove that both versions of the algorithm are correct.
- Exactly* how many blits does the algorithm perform?
- What is the algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?
- What if a $k \times k$ blit takes only $O(k)$ time?

►3. Dynamic Programming: The Company Party

A company is planning a party for its employees. The organizers of the party want it to be a fun party, and so have assigned a ‘fun’ rating to every employee. The employees are organized into a strict hierarchy, i.e. a tree rooted at the president. There is one restriction, though, on the guest list to the party: both an employee and their immediate supervisor (parent in the tree) cannot both attend the party (because that would be no fun at all). Give an algorithm that makes a guest list for the party that maximizes the sum of the ‘fun’ ratings of the guests.

►4. Dynamic Programming: Longest Increasing Subsequence (LIS)

Give an $O(n^2)$ algorithm to find the longest increasing subsequence of a sequence of numbers. Note: the elements of the subsequence need not be adjacent in the sequence. For example, the sequence (1, 5, 3, 2, 4) has an LIS (1, 3, 4).

►5. Nut/Bolt Median

You are given a set of n nuts and n bolts of different sizes. Each nut matches exactly one bolt (and vice versa, of course). The sizes of the nuts and bolts are so similar that you cannot compare two nuts or two bolts to see which is larger. You can, however, check whether a nut is too small, too large, or just right for a bolt (and vice versa, of course).

In this problem, your goal is to find the median bolt (*i.e.*, the $\lfloor n/2 \rfloor$ th largest) as quickly as possible.

- (a) Describe an efficient deterministic algorithm that finds the median bolt. How many nut/bolt comparisons does your algorithm perform in the worst case?
- (b) Describe an efficient *randomized* algorithm that finds the median bolt.
 - i. State a recurrence for the expected number of nut/bolt comparisons your algorithm performs.
 - ii. What is the probability that your algorithm compares the i th largest bolt with the j th largest nut?
 - iii. What is the expected number of nut-bolt comparisons made by your algorithm? [Hint: Use your answer to either of the previous two questions.]

Only 1U Grad Problems

►1. You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)

- (a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.
- *(b) Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Give an efficient algorithm that identifies a member of the plurality party.

- *(c) Suppose you don't know how many parties there are, but you do know that one party has a plurality, and at least p people in the plurality party are present. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)
- ★(d) Finally, suppose you don't know how many parties are represented at the convention, and you don't know how big the plurality is. Give an efficient algorithm to identify a member of the plurality party. How is the running time of your algorithm affected by the number of parties (k)? By the size of the plurality (p)?

Practice Problems

1. Second Smallest

Give an algorithm that finds the *second* smallest of n elements in at most $n + \lceil \lg n \rceil - 2$ comparisons. Hint: divide and conquer to find the smallest; where is the second smallest?

2. Linear in-place 0-1 sorting

Suppose that you have an array of records whose keys to be sorted consist only of 0's and 1's. Give a simple, linear-time $O(n)$ algorithm to sort the array in place (using storage of no more than constant size in addition to that of the array).

3. Dynamic Programming: Coin Changing

Consider the problem of making change for n cents using the least number of coins.

- (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
- (b) Suppose that the available coins have the values c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.
- (c) Give a set of 4 coin values for which the greedy algorithm does not yield an optimal solution, show why.
- (d) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of coin values.
- (e) Prove that, with only two coins a, b whose gcd is 1, the smallest value n for which change *can* be given for all values greater than or equal to n is $(a - 1)(b - 1)$.
- ★(f) For only three coins a, b, c whose gcd is 1, give an algorithm to determine the smallest value n for which change *can* be given for all values greater than n . (note: this problem is currently unsolved for $n > 4$.)

4. Dynamic Programming: Paragraph Justification

Consider the problem of printing a paragraph neatly on a printer (with fixed width font). The input text is a sequence of n words of lengths l_1, l_2, \dots, l_n . The line length is M (the maximum # of characters per line). We expect that the paragraph is left justified, that all first words on a line start at the leftmost position and that there is exactly one space between any two words on the same line. We want the uneven right ends of all the lines to be together as ‘neat’ as possible. Our criterion of neatness is that we wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of the lines. Note: if a printed line contains words i through j , then the number of spaces at the end of the line is $M - j + i - \sum_{k=i}^j l_k$.

- (a) Give a dynamic programming algorithm to do this.
- (b) Prove that if the neatness function is linear, a linear time greedy algorithm will give an optimum ‘neatness’.

5. Comparison of Amortized Analysis Methods

A sequence of n operations is performed on a data structure. The i th operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

- (a) Aggregate method
- (b) Accounting method
- *(c) Potential method

CS 373: Combinatorial Algorithms, Spring 1999

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 2 (due Thu. Feb. 18, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
2. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
3. Justify the correctness of your algorithm, including termination if that is not obvious.
4. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

▶ 1. Faster Longest Increasing Subsequence (LIS)

Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. Hint: In the dynamic programming solution, you don't really have to look back at all previous items.

▶ 2. $\text{SELECT}(A, k)$

Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

- (a) Show that a rotation in an augmented binary tree can be performed in constant time.
- (b) Describe an algorithm $\text{SCAPEGOATSELECT}(k)$ that selects the k th smallest item in an augmented scapegoat tree in $O(\log n)$ worst-case time. (The scapegoat trees presented in class were already augmented.)
- (c) Describe an algorithm $\text{SPLAYSELECT}(k)$ that selects the k th smallest item in an augmented splay tree in $O(\log n)$ amortized time.
- (d) Describe an algorithm $\text{TREAPSELECT}(k)$ that selects the k th smallest item in an augmented treap in $O(\log n)$ expected time.

►3. Scapegoat trees

- (a) Prove that only one tree gets rebalanced at any insertion.
- (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree ($I(v) = \max(0, |\hat{v}| - |\check{v}|)$, where \hat{v} is the child of greater height and \check{v} the child of lesser height, $|v|$ is the number of nodes in subtree v , and perfectly balanced means each subtree has as close to half the leaves as possible and is perfectly balanced itself).
- *(c) Show that you can rebuild a fully balanced binary tree in $O(n)$ time using only $O(1)$ additional memory.

►4. Memory Management

Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do not use the potential method (it makes it much more difficult).

Only 1U Grad Problems

▷1. Detecting overlap

- (a) You are given a list of ranges represented by min and max (e.g. [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$ -time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.
- (b) You are given a list of rectangles represented by min and max x - and y - coordinates. Give an $O(n \log n)$ -time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.

Practice Problems

1. Amortization

- (a) Modify the binary double-counter (see class notes Feb. 2) 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: Suppose p is the number of significant bits in P , and n is the number of significant bits in N . For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

- *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*2. Amortization

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 i th least significant fit indicates whether the sum includes the i th Fibonacci number F_i . For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

3. Rotations

- (a) Show that it is possible to transform any n -node binary search tree into any other n -node binary search tree using at most $2n - 2$ rotations.
- *(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

4. Fibonacci Heaps: SECONDMIN

We want to find the second smallest of a set efficiently.

- (a) Implement SECONDMIN by using a Fibonacci heap as a black box. Remember to justify its correctness and running time.
- *(b) Modify the Fibonacci Heap data structure to implement SECONDMIN in constant time.
5. Give an efficient implementation of the operation **Fib-Heap-Change-Key**(H, x, k), which changes the key of a node x in a Fibonacci heap H to the value k . The changes you make to Fibonacci heap data structure to support your implementation should not affect the amortized running time of any other Fibonacci heap operations. Analyze the amortized running time of your implementation for cases in which k is greater than, less than, or equal to $\text{key}[x]$.

CS 373: Combinatorial Algorithms, Spring 1999

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 2 (due Thu. Feb. 18, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
2. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
3. Justify the correctness of your algorithm, including termination if that is not obvious.
4. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

▶ 1. Faster Longest Increasing Subsequence (LIS)

Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. Hint: In the dynamic programming solution, you don't really have to look back at all previous items.

▶ 2. $\text{SELECT}(A, k)$

Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

- (a) Show that a rotation in an augmented binary tree can be performed in constant time.
- (b) Describe an algorithm $\text{SCAPEGOATSELECT}(k)$ that selects the k th smallest item in an augmented scapegoat tree in $O(\log n)$ worst-case time. (The scapegoat trees presented in class were already augmented.)
- (c) Describe an algorithm $\text{SPLAYSELECT}(k)$ that selects the k th smallest item in an augmented splay tree in $O(\log n)$ amortized time.
- (d) Describe an algorithm $\text{TREAPSELECT}(k)$ that selects the k th smallest item in an augmented treap in $O(\log n)$ expected time.

►3. Scapegoat trees

- (a) Prove that only one tree gets rebalanced at any insertion.
- (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree ($I(v) = \max(0, |\hat{v}| - |\check{v}|)$, where \hat{v} is the child of greater height and \check{v} the child of lesser height, $|v|$ is the number of nodes in subtree v , and perfectly balanced means each subtree has as close to half the leaves as possible and is perfectly balanced itself).
- *(c) Show that you can rebuild a fully balanced binary tree in $O(n)$ time using only $O(1)$ additional memory.

►4. Memory Management

Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do not use the potential method (it makes it much more difficult).

Only 1U Grad Problems

▷1. Detecting overlap

- (a) You are given a list of ranges represented by min and max (e.g. [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$ -time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.
- (b) You are given a list of rectangles represented by min and max x - and y - coordinates. Give an $O(n \log n)$ -time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.

Practice Problems

1. Amortization

- (a) Modify the binary double-counter (see class notes Feb. 2) 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: Suppose p is the number of significant bits in P , and n is the number of significant bits in N . For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

- *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*2. Amortization

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 i th least significant fit indicates whether the sum includes the i th Fibonacci number F_i . For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

3. Rotations

- (a) Show that it is possible to transform any n -node binary search tree into any other n -node binary search tree using at most $2n - 2$ rotations.
- *(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

4. Fibonacci Heaps: SECONDMIN

We want to find the second smallest of a set efficiently.

- (a) Implement SECONDMIN by using a Fibonacci heap as a black box. Remember to justify its correctness and running time.
- *(b) Modify the Fibonacci Heap data structure to implement SECONDMIN in constant time.
5. Give an efficient implementation of the operation **Fib-Heap-Change-Key**(H, x, k), which changes the key of a node x in a Fibonacci heap H to the value k . The changes you make to Fibonacci heap data structure to support your implementation should not affect the amortized running time of any other Fibonacci heap operations. Analyze the amortized running time of your implementation for cases in which k is greater than, less than, or equal to $\text{key}[x]$.

CS 373: Combinatorial Algorithms, Spring 1999

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 3 (due Thu. Mar. 11, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. (New!) If not already done, model the problem appropriately. Often the problem is stated in real world terms; give a more rigorous description of the problem. This will help you figure out what is assumed (what you know and what is arbitrary, what operations are and are not allowed), and find the tools needed to solve the problem.
2. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
3. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
4. Justify the correctness of your algorithm, including termination if that is not obvious.
5. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

▶ 1. Hashing

- (a) (2 pts) Consider an open-address hash table with uniform hashing and a load factor $\alpha = 1/2$. What is the expected number of probes in an unsuccessful search? Successful search?
- (b) (3 pts) Let the hash function for a table of size m be

$$h(x) = \lfloor Ax \rfloor \bmod m$$

where $A = \hat{\phi} = \frac{\sqrt{5}-1}{2}$. Show that this gives the best possible spread, i.e. if the x are hashed in order, $x+1$ will be hashed in the largest remaining contiguous interval.

►2. (5 pts) Euler Tour:

Given an **undirected** graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge *exactly* once, or says that it can't be done.

►3. (5 pts) Makefiles:

In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called ‘make’ that only recompiles those files that were changed, *and* any intermediate files in the compilation that depend on those changed. Design an algorithm to recompile only those necessary.

►4. (5 pts) Shortest Airplane Trip:

A person wants to fly from city A to city B in the shortest possible time. S/he turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose an optimal route. Hint: rather than modify Dijkstra’s algorithm, modify the data. The time is from departure to arrival at the destination, so it will include layover time (time waiting for a connecting flight).

►5. (9 pts, 3 each) Minimum Spanning Tree changes Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).

- Give an algorithm to update the MST when an edge is added to G .
- Give an algorithm to update the MST when an edge is deleted from G .
- Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G .

Only 1U Grad Problems

▷1. Nesting Envelopes

You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B . Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

Practice Problems

1. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} 1 & (i, j) \in E, \\ 0 & (i, j) \notin E. \end{cases}$$

- (a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose). Justify.
- (b) Describe what all the entries of the matrix product B^TB represent. Justify.
- ★(c) Let $C = BB^T - 2A$. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees. (A is the adjacency matrix of G , with zeroes on the diagonal).

2. $o(V^2)$ Adjacency Matrix Algorithms

- (a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.
- (b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V - 2$ (the body) connected to the other $V - 3$ vertices (the feet). Some of the feet may be connected to other feet.

Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

- (c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

3. Shortest Cycle:

Given an **undirected** graph $G = (V, E)$, and a weight function $f : E \rightarrow \mathbf{R}$ on the **edges**, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G .

4. Longest Simple Path:

Let graph $G = (V, E)$, $|V| = n$. A *simple path* of G , is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G . Hint: It can be done in $O(n^c 2^n)$ time, for some constant c .

5. Minimum Spanning Tree:

Suppose all edge weights in a graph G are equal. Give an algorithm to compute an MST.

6. Transitive reduction:

Give an algorithm to construct a *transitive reduction* of a directed graph G , i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR} .

7. (a) What is $5^{2^{29}5^0 + 23^4 + 17^3 + 11^2 + 5^1} \mod 6$?

- (b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Spring 1999

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 4 (due Thu. Apr. 1, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. (New!) If not already done, model the problem appropriately. Often the problem is stated in real world terms; give a more rigorous description of the problem. This will help you figure out what is assumed (what you know and what is arbitrary, what operations are and are not allowed), and find the tools needed to solve the problem.
2. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
3. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
4. Justify the correctness of your algorithm, including termination if that is not obvious.
5. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

▶ 1. (5 pts total) Collinearity

Give an $O(n^2 \log n)$ algorithm to determine whether any three points of a set of n points are collinear. Assume two dimensions and exact arithmetic.

▶ 2. (4 pts, 2 each) Convex Hull Recurrence

Consider the following generic recurrence for convex hull algorithms that divide and conquer:

$$T(n, h) = T(n_1, h_1) + T(n_2, h_2) + O(n)$$

where $n \geq n_1 + n_2$, $h = h_1 + h_2$ and $n \geq h$. This means that the time to compute the convex hull is a function of both n , the number of input points, and h , the number of convex hull vertices. The splitting and merging parts of the divide-and-conquer algorithm take $O(n)$ time. When n is a constant, $T(n, h)$ is $O(1)$, but when h is a constant, $T(n, h)$ is $O(n)$. Prove that for both of the following restrictions, the solution to the recurrence is $O(n \log h)$:

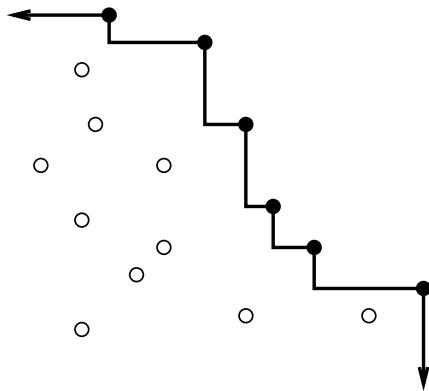
- (a) $h_1, h_2 < \frac{3}{4}h$
- (b) $n_1, n_2 < \frac{3}{4}n$

►3. (5 pts) Circle Intersection

Give an $O(n \log n)$ algorithm to test whether any two circles in a set of size n intersect.

►4. (5 pts total) Staircases

You are given a set of points in the first quadrant. A *left-up* point of this set is defined to be a point that has no points both greater than it in both coordinates. The left-up subset of a set of points then forms a *staircase* (see figure).



- (a) (3 pts) Give an $O(n \log n)$ algorithm to find the staircase of a set of points.

- (b) (2 pts) Assume that points are chosen uniformly at random within a rectangle. What is the average number of points in a staircase? Justify. Hint: you will be able to give an exact answer rather than just asymptotics. You have seen the same analysis before.

Only 1U Grad Problems

►1. (6 pts, 2 each) Ghostbusters and Ghosts

A group of n ghostbusters is battling n ghosts. Each ghostbuster can shoot a single energy beam at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits a ghost. The ghostbusters must all fire at the same time and no two energy beams may cross. The positions of the ghosts and ghostbusters is fixed in the plane (assume that no three points are collinear).

- (a) Prove that for any configuration ghosts and ghostbusters there exists such a non-crossing matching.
- (b) Show that there exists a line passing through one ghostbuster and one ghost such that the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Give an efficient algorithm to find such a line.
- (c) Give an efficient divide and conquer algorithm to pair ghostbusters and ghosts so that no two streams cross.

Practice Problems

1. Basic Computation (assume two dimensions and *exact* arithmetic)
 - (a) Intersection: Extend the basic algorithm to determine if two line segments intersect by taking care of *all* degenerate cases.
 - (b) Simplicity: Give an $O(n \log n)$ algorithm to determine whether an n -vertex polygon is simple.
 - (c) Area: Give an algorithm to compute the area of a simple n -polygon (not necessarily convex) in $O(n)$ time.
 - (d) Inside: Give an algorithm to determine whether a point is within a simple n -polygon (not necessarily convex) in $O(n)$ time.

2. External Diagonals and Mouths
 - (a) A pair of polygon vertices defines an *external diagonal* if the line segment between them is completely outside the polygon. Show that every nonconvex polygon has at least one external diagonal.
 - (b) Three consecutive polygon vertices p, q, r form a *mouth* if p and r define an external diagonal. Show that every nonconvex polygon has at least one mouth.

3. On-Line Convex Hull

We are given the set of points one point at a time. After receiving each point, we must compute the convex hull of all those points so far. Give an algorithm to solve this problem in $O(n^2)$ (We could obviously use Graham's scan n times for an $O(n^2 \log n)$ algorithm). Hint: How do you maintain the convex hull?

4. Another On-Line Convex Hull Algorithm
 - (a) Given an n -polygon and a point outside the polygon, give an algorithm to find a tangent.
 - *(b) Suppose you have found both tangents. Give an algorithm to remove the points from the polygon that are within the angle formed by the tangents (as segments!) and the opposite side of the polygon.
 - (c) Use the above to give an algorithm to compute the convex hull on-line in $O(n \log n)$

- ★5. Order of the size of the convex hull

The convex hull on $n \geq 3$ points can have anywhere from 3 to n points. The average case depends on the distribution.

 - (a) Prove that if a set of points is chosen randomly within a given rectangle, then the average size of the convex hull is $O(\log n)$.
 - (b) Prove that if a set of points is chosen randomly within a given circle, then the average size of the convex hull is $O(\sqrt{n})$.

CS 373: Combinatorial Algorithms, Spring 1999

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 5 (due Thu. Apr. 22, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: You will be held accountable for the appropriate responses for answers (e.g. give models, proofs, analyses, etc)

Undergrad/.75U Grad/1U Grad Problems

- ▶1. (5 pts) Show how to find the occurrences of pattern P in text T by computing the prefix function of the string PT (the concatenation of P and T).

- ▶2. (10 pts total) Fibonacci strings and KMP matching
Fibonacci strings are defined as follows:

$$F_1 = \text{"b"}, \quad F_2 = \text{"a"}, \quad \text{and } F_n = F_{n-1}F_{n-2}, (n > 2)$$

where the recursive rule uses concatenation of strings, so F_2 is "ab", F_3 is "aba". Note that the length of F_n is the n th Fibonacci number.

- (a) (2 pts) Prove that in any Fibonacci string there are no two b's adjacent and no three a's.
- (b) (2 pts) Give the unoptimized and optimized 'prefix' (fail) function for F_7 .
- (c) (3 pts) Prove that, in searching for a Fibonacci string of length m using unoptimized KMP, it may shift up to $\lceil \log_\phi m \rceil$ times, where $\phi = (1 + \sqrt{5})/2$, is the golden ratio. (Hint: Another way of saying the above is that we are given the string F_n and we may have to shift n times. Find an example text T that gives this number of shifts).
- (d) (3 pts) What happens here when you use the optimized prefix function? Explain.

- ▶3. (5 pts) Prove that finding the second smallest of n elements takes $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. Prove for both upper and lower bounds. Hint: find the (first) smallest using an elimination tournament.

- ▶4. (4 pts, 2 each) Lower Bounds on Adjacency Matrix Representations of Graphs
 - (a) Prove that the time to determine if an undirected graph has a cycle is $\Omega(V^2)$.

- (b) Prove that the time to determine if there is a path between two nodes in an undirected graph is $\Omega(V^2)$.

Only 1U Grad Problems

- ▷ 1. (5 pts) Prove that $\lceil 3n/2 \rceil - 2$ comparisons are necessary in the worst case to find both the minimum and maximum of n numbers. Hint: Consider how many are potentially either the min or max.

Practice Problems

1. String matching with wild-cards

Suppose you have an alphabet for patterns that includes a ‘gap’ or wild-card character; any length string of any characters can match this additional character. For example if ‘*’ is the wild-card, then the pattern ‘foo*bar*nad’ can be found in ‘foofoowangbarnad’. Modify the computation of the prefix function to correctly match strings using KMP.

2. Prove that there is no comparison sort whose running time is linear for at least $1/2$ of the $n!$ inputs of length n . What about at least $1/n$? What about at least $1/2^n$?

3. Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.

4. Find asymptotic upper and lower bounds to $\lg(n!)$ without Stirling’s approximation (Hint: use integration).

5. Given a sequence of n elements of n/k blocks (k elements per block) all elements in a block are less than those to the right in sequence, show that you cannot have the whole sequence sorted in better than $\Omega(n \lg k)$. Note that the entire sequence would be sorted if each of the n/k blocks were individually sorted in place. Also note that combining the lower bounds for each block is not adequate (that only gives an upper bound).

6. Some elementary reductions

- (a) Prove that if you can decide whether a graph G has a clique of size k (or less) then you can decide whether a graph G' has an independent set of size k (or more).

- (b) Prove that if you can decide whether one graph G_1 is a subgraph of another graph G_2 then you can decide whether a graph G has a clique of size k (or less).

7. There is no Proof but We are pretty Sure

Justify (prove) the following logical rules of inference:

- (a) Classical - If $a \rightarrow b$ and a hold, then b holds.

- (b) Fuzzy - Prove: If $a \rightarrow b$ holds, and a holds with probability p , then b holds with probability less than p . Assume all probabilities are independent.

- (c) Give formulas for computing the probabilities of the fuzzy logical operators ‘and’, ‘or’, ‘not’, and ‘implies’, and fill out truth tables with the values T (true, $p = 1$), L (likely, $p = 0.9$), M (maybe, $p = 0.5$), N (not likely, $p = 0.1$), and F (false, $p = 0$).

- (d) If you have a poly time (algorithmic) reduction from problem B to problem A (i.e. you can solve B using A with a poly time conversion), and it is very unlikely that A has better than lower bound $\Omega(2^n)$ algorithm, what can you say about problem A . Hint: a solution to A implies a solution to B .

CS 373: Combinatorial Algorithms, Spring 1999

Midterm 1 (February 23, 1999)

Name:

Net ID:

Alias:

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your $8\frac{1}{2}'' \times 11''$ cheat sheet, please leave it at the front of the classroom.

- **Don't panic!**

- Print your name, netid, and alias in the boxes above, and print your name at the top of every page.
 - **Answer four of the five questions on the exam.** Each question is worth 10 points. If you answer more than four questions, the one with the lowest score will be ignored. **1-unit graduate students must answer question #5.**
 - Please write your answers on the front of the exam pages. Use the backs of the pages as scratch paper. Let us know if you need more paper.
 - Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
 - Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
 - Write *something* down for every problem. Don't panic and erase large chunks of work. Even if you think it's nonsense, it might be worth partial credit.
- **Don't panic!**
-

#	Score	Grader
1		
2		
3		
4		
5		

1. Multiple Choice

Every question below has one of the following answers.

- (a) $\Theta(1)$ (b) $\Theta(\log n)$ (c) $\Theta(n)$ (d) $\Theta(n \log n)$ (e) $\Theta(n^2)$

For each question, write the letter that corresponds to your answer. You do not need to justify your answer. Each correct answer earns you 1 point, but each incorrect answer *costs* you $\frac{1}{2}$ point. (You cannot score below zero.)

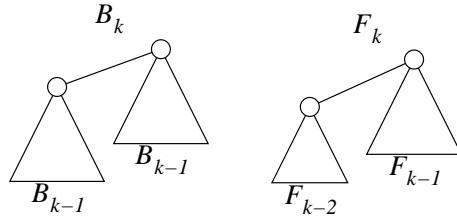
- What is $\sum_{i=1}^n i$?
- What is $\sum_{i=1}^n \frac{1}{i}$?
- What is the solution of the recurrence $T(n) = T(\sqrt{n}) + n$?
- What is the solution of the recurrence $T(n) = T(n - 1) + \lg n$?
- What is the solution of the recurrence $T(n) = 2T\left(\lceil \frac{n+27}{2} \rceil\right) + 5n - 7\sqrt{\lg n} + \frac{1999}{n}$?
- The amortized time for inserting one item into an n -node splay tree is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty splay tree?
- The expected time for inserting one item into an n -node randomized treap is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty treap?
- What is the worst-case running time of randomized quicksort?
- How many bits are there in the binary representation of the n th Fibonacci number?
- What is the worst-case cost of merging two arbitrary splay trees with n items total into a single splay tree with n items.
- Suppose you correctly identify three of the answers to this question as obviously wrong. If you pick one of the two remaining answers at random, what is your expected score for this problem?

2. (a) [5 pt] Recall that a binomial tree of order k , denoted B_k , is defined recursively as follows. B_0 is a single node. For any $k > 0$, B_k consists of two copies of B_{k-1} linked together.

Prove that the degree of any node in a binomial tree is equal to its height.

- (b) [5 pt] Recall that a Fibonacci tree of order k , denoted F_k , is defined recursively as follows. F_1 and F_2 are both single nodes. For any $k > 2$, F_k consists of an F_{k-2} linked to an F_{k-1} .

Prove that for any node v in a Fibonacci tree, $\text{height}(v) = \lceil \text{degree}(v)/2 \rceil$.



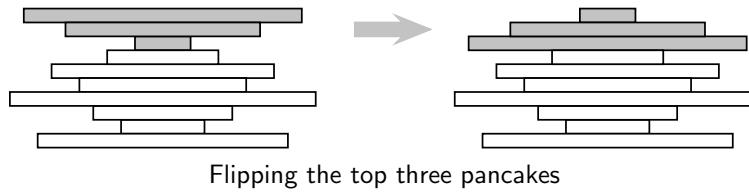
Recursive definitions of binomial trees and Fibonacci trees.

3. Consider the following randomized algorithm for computing the smallest element in an array.

```
RANDOMMIN( $A[1..n]$ ):  
     $min \leftarrow \infty$   
    for  $i \leftarrow 1$  to  $n$  in random order  
        if  $A[i] < min$   
             $min \leftarrow A[i]$  (*)  
    return  $min$ 
```

- (a) [1 pt] In the worst case, how many times does RANDOMMIN execute line (*)?
- (b) [3 pt] What is the probability that line (*) is executed during the n th iteration of the for loop?
- (c) [6 pt] What is the exact expected number of executions of line (*)? (A correct $\Theta()$ bound is worth 4 points.)

4. Suppose we have a stack of n pancakes of different sizes. We want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation we can perform is a *flip* — insert a spatula under the top k pancakes, for some k between 1 and n , and flip them all over.



- (a) [3 pt] Describe an algorithm to sort an arbitrary stack of n pancakes.
- (b) [3 pt] Prove that your algorithm is correct.
- (c) [2 pt] Exactly how many flips does your algorithm perform in the worst case? (A correct $\Theta()$ bound is worth one point.)
- (d) [2 pt] Suppose one side of each pancake is burned. Exactly how many flips do you need to sort the pancakes *and* have the burned side of every pancake on the bottom? (A correct $\Theta()$ bound is worth one point.)

5. You are given an array $A[1..n]$ of integers. Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray $A[i..j]$.

For example, if the array contains the numbers $(-6, 12, -7, 0, 14, -7, 5)$, then the largest sum is $19 = 12 - 7 + 0 + 14$.

-6	12	-7	0	14	-7	5
19						

To get full credit, your algorithm must run in $\Theta(n)$ time — there are at least three different ways to do this. An algorithm that runs in $\Theta(n^2)$ time is worth 7 points.

CS 373: Combinatorial Algorithms, Spring 1999

Midterm 2 (April 6, 1999)

Name:

Net ID:

Alias:

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your $8\frac{1}{2}'' \times 11''$ cheat sheet, please leave it at the front of the classroom.

-
- Don't panic!
 - Print your name, netid, and alias in the boxes above, and print your name at the top of every page.
 - Answer four of the five questions on the exam. Each question is worth 10 points. If you answer more than four questions, the one with the lowest score will be ignored. **1-unit graduate students must answer question #5.**
 - Please write your answers on the front of the exam pages. You can use the backs of the pages as scratch paper. Let us know if you need more paper.
 - Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
 - Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
 - Write *something* down for every problem. Don't panic and erase large chunks of work. Even if you think it's nonsense, it might be worth partial credit.
 - Don't panic!
-

#	Score	Grader
1		
2		
3		
4		
5		

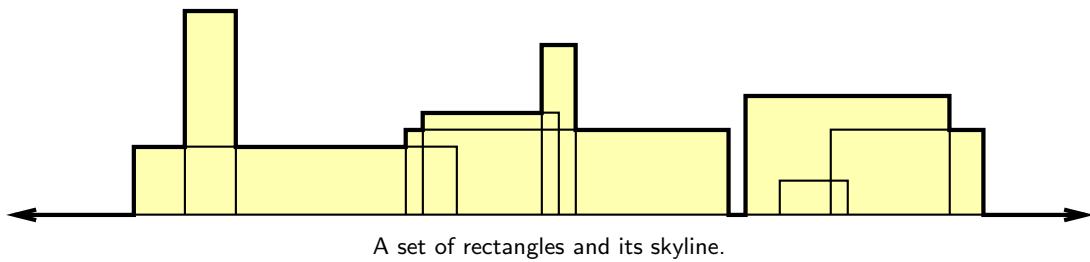
1. Bipartite Graphs

A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R , such that every edge has one vertex in L and the other in R .

- (a) Prove that every tree is a bipartite graph.
- (b) Describe and analyze an efficient algorithm that determines whether a given connected, undirected graph is bipartite.

2. Manhattan Skyline

The purpose of the following problem is to compute the outline of a projection of rectangular buildings. You are given the height, width, and left x -coordinate of n rectangles. The bottom of each rectangle is on the x -axis. Describe and analyze an efficient algorithm to compute the vertices of the “skyline”.



3. Least Cost Vertex Weighted Path

Suppose you want to drive from Champaign to Los Angeles via a network of roads connecting cities. You don't care how long it takes, how many cities you visit, or how much gas you use. All you care about is how much money you spend on food. Each city has a possibly different, but fixed, value for food.

More formally, you are given a directed graph $G = (V, E)$ with nonnegative weights on the vertices $w: V \rightarrow \mathbb{R}^+$, a source vertex $s \in V$, and a target vertex $t \in V$. Describe and analyze an efficient algorithm to find a minimum-weight path from s to t . [Hint: Modify the graph.]

4. Union-Find with Alternate Rule

In the UNION-FIND data structure described in CLR and in class, each set is represented by a rooted tree. The UNION algorithm, given two sets, decides which set is to be the parent of the other by comparing their ranks, where the rank of a set is an upper bound on the height of its tree.

Instead of rank, we propose using the *weight* of the set, which is just the number of nodes in the set. Here's the modified UNION algorithm:

```
UNION(A, B):
    if weight(A) > weight(B)
        parent(B) ← A
        weight(A) ← weight(A) + weight(B)
    else
        parent(A) ← B
        weight(B) ← weight(A) + weight(B)
```

Prove that if we use this method, then after any sequence of n MAKESETS, UNIONS, and FINDS (with path compression), the *height* of the tree representing any set is $O(\log n)$.

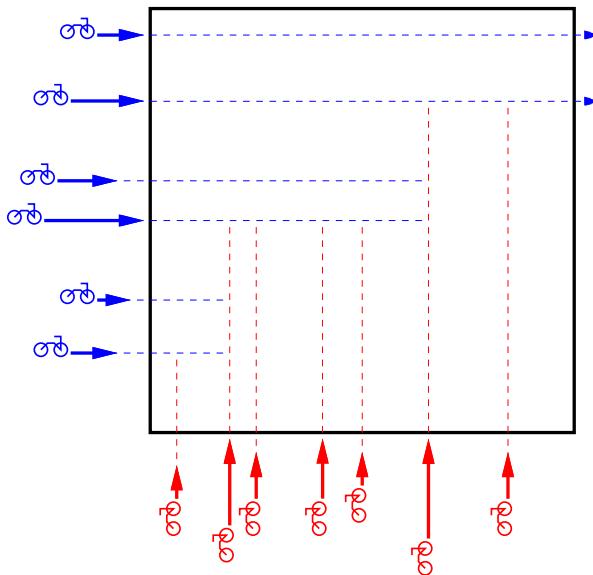
[Hint: First prove it without path compression, and then argue that path compression doesn't matter (for this problem).]

5. Motorcycle Collision

One gang, Hell's Ordinates, start west of the arena facing directly east; the other, The Vicious Abscissas of Death, start south of the arena facing due north. All the motorcycles start moving simultaneously at a prearranged signal. Each motorcycle moves at a fixed speed—no speeding up, slowing down, or turning is allowed. Each motorcycle leaves an oil trail behind it. If another motorcycle crosses that trail, it falls over and stops leaving a trail.

More formally, we are given two sets H and V , each containing n motorcycles. Each motorcycle is represented by three numbers (s, x, y) : its speed and the x - and y -coordinates of its initial location. Bikes in H move horizontally; bikes in V move vertically.

Assume that the bikes are infinitely small points, that the bike trails are infinitely thin lie segments, that a bike crashes stops *exactly* when it hits a oil trail, and that no two bikes collide with each other.



Two sets of motorcycles and the oil trails they leave behind.

- (a) Solve the case $n = 1$. Given only two motorcycles moving perpendicular to each other, determine which one of them falls over and where in $O(1)$ time.
- (b) Describe an efficient algorithm to find the set of all points where motorcycles fall over.

5. Motorcycle Collision (continued)

Incidentally, the movie *Tron* is being shown during Roger Ebert's Forgotten Film Festival at the Virginia Theater in Champaign on April 25. Get your tickets now!

CS 373: Combinatorial Algorithms, Spring 1999

Final Exam (May 7, 1999)

Name:

Net ID:

Alias:

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your two $8\frac{1}{2}'' \times 11''$ cheat sheets, please leave it at the front of the classroom.

-
- Print your name, netid, and alias in the boxes above, and print your name at the top of every page.
 - **Answer six of the seven questions on the exam.** Each question is worth 10 points. If you answer every question, the one with the lowest score will be ignored. **1-unit graduate students must answer question #7.**
 - Please write your answers on the front of the exam pages. Use the backs of the pages as scratch paper. Let us know if you need more paper.
 - Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
 - Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
 - Write *something* down for every problem. Don't panic and erase large chunks of work. Even if you think it's nonsense, it might be worth partial credit.
-

#	Score	Grader
1		
2		
3		
4		
5		
6		
7		

1. Short Answer

sorting	induction	Master theorem	divide and conquer
randomized algorithm	amortization	brute force	hashing
binary search	depth-first search	splay tree	Fibonacci heap
convex hull	sweep line	minimum spanning tree	shortest paths
shortest path	adversary argument	NP-hard	reduction
string matching	evasive graph property	dynamic programming	H_n

Choose from the list above the best method for solving each of the following problems. We do *not* want complete solutions, just a short description of the proper solution technique! Each item is worth 1 point.

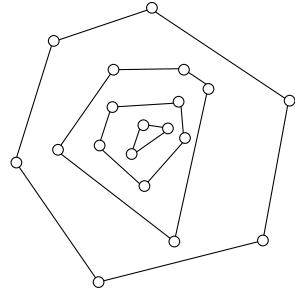
- (a) Given a Champaign phone book, find your own phone number.
- (b) Given a collection of n rectangles in the plane, determine whether any two intersect in $O(n \log n)$ time.
- (c) Given an undirected graph G and an integer k , determine if G has a complete subgraph with k edges.
- (d) Given an undirected graph G , determine if G has a triangle — a complete subgraph with three vertices.
- (e) Prove that any n -vertex graph with minimum degree at least $n/2$ has a Hamiltonian cycle.
- (f) Given a graph G and three distinguished vertices u , v , and w , determine whether G contains a path from u to v that passes through w .
- (g) Given a graph G and two distinguished vertices u and v , determine whether G contains a path from u to v that passes through at most 17 edges.
- (h) Solve the recurrence $T(n) = 5T(n/17) + O(n^{4/3})$.
- (i) Solve the recurrence $T(n) = 1/n + T(n - 1)$, where $T(0) = 0$.
- (j) Given an array of n integers, find the integer that appears most frequently in the array.

- (a) _____ (f) _____
- (b) _____ (g) _____
- (c) _____ (h) _____
- (d) _____ (i) _____
- (e) _____ (j) _____

2. Convex Layers

Given a set Q of points in the plane, define the *convex layers* of Q inductively as follows: The first convex layer of Q is just the convex hull of Q . For all $i > 1$, the i th convex layer is the convex hull of Q after the vertices of the first $i - 1$ layers have been removed.

Give an $O(n^2)$ -time algorithm to find all convex layers of a given set of n points. [Partial credit for a correct slower algorithm; extra credit for a correct faster algorithm.]



A set of points with four convex layers.

3. Suppose you are given an array of n numbers, sorted in increasing order.

- (a) [3 pts] Describe an $O(n)$ -time algorithm for the following problem:

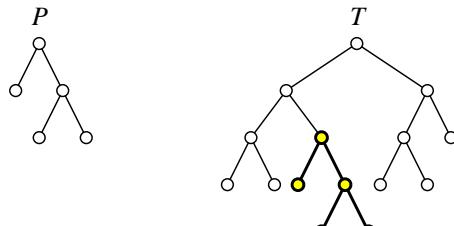
Find two numbers from the list that add up to zero, or report that there is no such pair.
In other words, find two numbers a and b such that $a + b = 0$.

- (b) [7 pts] Describe an $O(n^2)$ -time algorithm for the following problem:

Find *three* numbers from the list that add up to zero, or report that there is no such triple.
In other words, find three numbers a , b , and c , such that $a + b + c = 0$. [Hint: Use something similar to part (a) as a subroutine.]

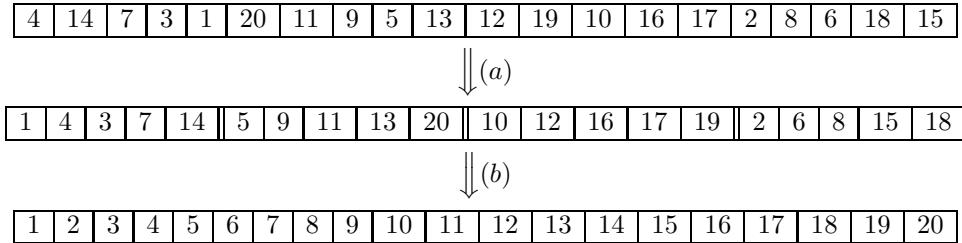
4. Pattern Matching

- (a) [4 pts] A *cyclic rotation* of a string is obtained by chopping off a prefix and gluing it at the end of the string. For example, ALGORITHM is a cyclic shift of RITHMALGO. Describe and analyze an algorithm that determines whether one string $P[1..m]$ is a cyclic rotation of another string $T[1..n]$.
- (b) [6 pts] Describe and analyze an algorithm that decides, given any two binary trees P and T , whether P equals a subtree of T . [Hint: First transform both trees into strings.]



5. Two-stage Sorting

- (a) [1 pt] Suppose we are given an array $A[1..n]$ of distinct integers. Describe an algorithm that splits A into n/k subarrays, each with k elements, such that the elements of each subarray $A[(i-1)k+1..ik]$ are sorted. Your algorithm should run in $O(n \log k)$ time.
- (b) [2 pts] Given an array $A[1..n]$ that is already split into n/k sorted subarrays as in part (a), describe an algorithm that sorts the entire array in $O(n \log(n/k))$ time.
- (c) [3 pts] Prove that your algorithm from part (a) is optimal.
- (d) [4 pts] Prove that your algorithm from part (b) is optimal.



6. SAT Reduction

Suppose you are have a black box that magically solves SAT (the formula satisfiability problem) in constant time. That is, given a boolean formula of variables and logical operators (\wedge, \vee, \neg), the black box tells you, in constant time, whether or not the formula can be satisfied. Using this black box, design and analyze a **polynomial-time** algorithm that computes an assignment to the variables that satisfies the formula.

7. Knapsack

You're hiking through the woods when you come upon a treasure chest filled with objects. Each object has a different size, and each object has a price tag on it, giving its value. There is no correlation between an object's size and its value. You want to take back as valuable a subset of the objects as possible (in one trip), but also making sure that you will be able to carry it in your knapsack which has a limited size.

In other words, you have an integer capacity K and a target value V , and you want to decide whether there is a subset of the objects whose total size is *at most* K and whose total value is *at least* V .

- (a) [5 pts] Show that this problem is NP-hard. [Hint: Restate the problem more formally, then reduce from the NP-hard problem PARTITION: Given a set S of nonnegative integers, is there a partition of S into disjoint subsets A and B (where $A \cup B = S$) whose sums are equal, *i.e.*, $\sum_{a \in A} a = \sum_{b \in B} b$.]
- (b) [5 pts] Describe and analyze a dynamic programming algorithm to solve the knapsack problem in $O(nK)$ time. Prove your algorithm is correct.

CS 373: Combinatorial Algorithms, Fall 2000

Homework 0, due August 31, 2000 at the beginning of class

Name:	
Net ID:	Alias:

Neatly print your name (first name first, with no comma), your network ID, and a short alias into the boxes above. **Do not sign your name. Do not write your Social Security number.** Staple this sheet of paper to the top of your homework.

Grades will be listed on the course web site by alias give us, so your alias should not resemble your name or your Net ID. If you don't give yourself an alias, we'll give you one that you won't like.

Before you do anything else, read the Homework Instructions and FAQ on the CS 373 course web page (<http://www-courses.cs.uiuc.edu/~cs373/hw/faq.html>), and then check the box below. This web page gives instructions on how to write and submit homeworks—staple your solutions together in order, write your name and netID on every page, don't turn in source code, analyze everything, use good English and good logic, and so forth.

I have read the CS 373 Homework Instructions and FAQ.

This homework tests your familiarity with the prerequisite material from CS 173, CS 225, and CS 273—many of these problems have appeared on homeworks or exams in those classes—primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Parberry and Chapters 1–6 of CLR should be sufficient review, but you may want to consult other texts as well.

Required Problems

- Sort the following 25 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any:

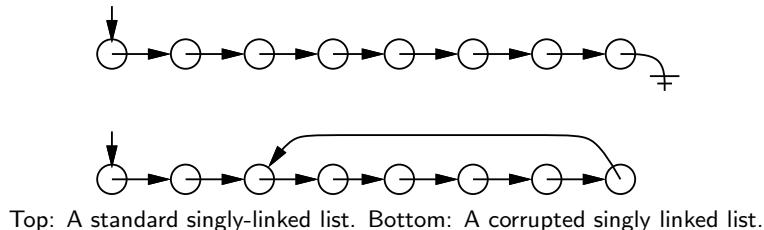
$$\begin{array}{ccccc} 1 & n & n^2 & \lg n & \lg(n \lg n) \\ \lg^* n & \lg^* 2^n & 2^{\lg^* n} & \lg \lg^* n & \lg^* \lg n \\ n^{\lg n} & (\lg n)^n & (\lg n)^{\lg n} & n^{1/\lg n} & n^{\lg \lg n} \\ \log_{1000} n & \lg^{1000} n & \lg^{(1000)} n & \left(1 + \frac{1}{n}\right)^n & n^{1/1000} \end{array}$$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

2. (a) Prove that any positive integer can be written as the sum of distinct powers of 2. For example: $42 = 2^5 + 2^3 + 2^1$, $25 = 2^4 + 2^3 + 2^0$, $17 = 2^4 + 2^0$. [Hint: “Write the number in binary” is *not* a proof; it just restates the problem.]
- (b) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.
- (c) Prove that *any* integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example: $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.
3. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. If no base cases are given, assume something reasonable but nontrivial. Extra credit will be given for more exact solutions.
- (a) $A(n) = 3A(n/2) + n$
 - (b) $B(n) = \max_{n/3 < k < 2n/3} (B(k) + B(n - k) + n)$
 - (c) $C(n) = 4C(\lfloor n/2 \rfloor + 5) + n^2$
 - *(d) $D(n) = 2D(n/2) + n/\lg n$
 - *(e) $E(n) = \frac{E(n-1)}{E(n-2)}$, where $E(1) = 1$ and $E(2) = 2$.
4. Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ..., 52 of clubs. (They’re big cards.) Penn shuffles the deck until each each of the $52!$ possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.
- (a) On average, how many cards does Penn give Teller?
 - (b) On average, what is the smallest-numbered card that Penn gives Teller?
 - *(c) On average, what is the largest-numbered card that Penn gives Teller?

[Hint: Solve for an n -card deck, and then set $n = 52$.] Prove that your answers are correct. If you have to appeal to “intuition” or “common sense”, your answers are probably wrong!

5. Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node's pointer is `NULL`. Unfortunately, your list might have been corrupted by a bug in somebody else's code¹, so that the last node has a pointer back to some other node in the list instead.

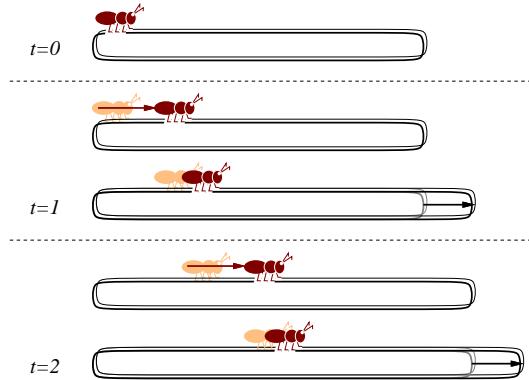


Top: A standard singly-linked list. Bottom: A corrupted singly linked list.

Describe an algorithm² that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

An ant is walking along a rubber band, starting at the left end. Once every second, the ant walks one inch to the right, and then you make the rubber band one inch longer by pulling on the right end. The rubber band stretches uniformly, so stretching the rubber band also pulls the ant to the right. The initial length of the rubber band is n inches, so after t seconds, the rubber band is $n + t$ inches long.



Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

- (a) How far has the ant moved after t seconds, as a function of n and t ? Set up a recurrence and (for full credit) give an *exact* closed-form solution. [Hint: What fraction of the rubber band's length has the ant walked?]
- *(b) How long does it take the ant to get to the right end of the rubber band? For full credit, give an answer of the form $f(n) + \Theta(1)$ for some explicit function $f(n)$.

¹After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

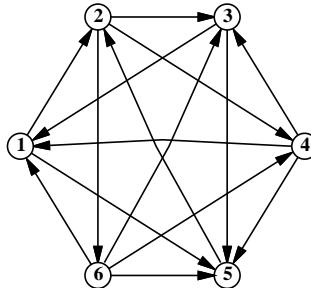
²Since you've read the Homework Instructions, you know what the phrase "describe an algorithm" means. Right?

Practice Problems

These remaining practice problems are entirely for your benefit. Don't turn in solutions—we'll just throw them out—but feel free to ask us about these questions during office hours and review sessions. Think of these as potential exam questions (hint, hint).

1. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m .
 - (a) F_n is even if and only if n is divisible by 3.
 - (b) $\sum_{i=0}^n F_i = F_{n+2} - 1$
 - (c) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$
 - ★(d) If n is an integer multiple of m , then F_n is an integer multiple of F_m .

2. A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once. Prove that every tournament contains at least one Hamiltonian path.



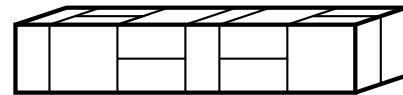
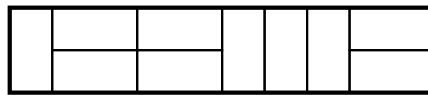
A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

3. (a) Prove the following identity by induction:

$$\binom{2n}{n} = \sum_{k=0}^n \binom{n}{k} \binom{n}{n-k}.$$

- (b) Give a non-inductive combinatorial proof of the same identity, by showing that the two sides of the equation count exactly the same thing in two different ways. There is a correct one-sentence proof.

4. (a) Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor} / n = \Theta(n)$.
 (b) Is $2^{\lfloor \lg n \rfloor} = \Theta(2^{\lceil \lg n \rceil})$? Justify your answer.
 (c) Is $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta(2^{2^{\lceil \lg \lg n \rceil}})$? Justify your answer.
 (d) Prove that if $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$. Justify your answer.
 (e) Prove that $f(n) = O(g(n))$ does *not* imply that $\log(f(n)) = O(\log(g(n)))$?.
 *(f) Prove that $\log^k n = o(n^{1/k})$ for any positive integer k .
5. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. If no base cases are given, assume something reasonable (but nontrivial). Extra credit will be given for more exact solutions.
- (a) $A(n) = A(n/2) + n$
 (b) $B(n) = 2B(n/2) + n$
 (c) $C(n) = \min_{0 < k < n} (C(k) + C(n - k) + 1)$, where $C(1) = 1$.
 (d) $D(n) = D(n - 1) + 1/n$
 *(e) $E(n) = 8E(n - 1) - 15E(n - 2) + 1$
 *(f) $F(n) = (n - 1)(F(n - 1) + F(n - 2))$, where $F(0) = F(1) = 1$
 ★(g) $G(n) = G(n/2) + G(n/4) + G(n/6) + G(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.]
6. (a) A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos? Set up a recurrence relation and give an *exact* closed-form solution.
 (b) A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

7. Professor George O'Jungle has a favorite 26-node binary tree, whose nodes are labeled by letters of the alphabet. The preorder and postorder sequences of nodes are as follows:

preorder: M N H C R S K W T G D X I Y A J P O E Z V B U L Q F
 postorder: C W T K S G R H D N A O E P J Y Z I B Q L F U V X M

Draw Professor O'Jungle's binary tree, and give the inorder sequence of nodes.

8. Alice and Bob each have a fair n -sided die. Alice rolls her die once. Bob then repeatedly throws his die until he rolls a number at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. If you have to appeal to “intuition” or “common sense”, your answer is probably wrong!

9. Prove that for any nonnegative parameters a and b , the following algorithms terminate and produce identical output.

<u>SLOWEUCLID(a, b) :</u>
if $b > a$
return SLOWEUCLID(b, a)
else if $b = 0$
return a
else
return SLOWEUCLID($b, a - b$)

<u>FASTEUCLID(a, b) :</u>
if $b = 0$
return a
else
return FASTEUCLID($b, a \bmod b$)

CS 373: Combinatorial Algorithms, Fall 2000

Homework 1 (due September 12, 2000 at midnight)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the i th word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j , then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^j p_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

2. Consider the following sorting algorithm:

```

STUPIDSORT( $A[0..n-1]$ ) :
  if  $n = 2$  and  $A[0] > A[1]$ 
    swap  $A[0] \leftrightarrow A[1]$ 
  else if  $n > 2$ 
     $m \leftarrow \lceil 2n/3 \rceil$ 
    STUPIDSORT( $A[0..m-1]$ )
    STUPIDSORT( $A[n-m..n-1]$ )
    STUPIDSORT( $A[0..m-1]$ )

```

- (a) Prove that STUPIDSORT actually sorts its input.
 - (b) Would the algorithm still sort correctly if we replaced the line $m \leftarrow \lceil 2n/3 \rceil$ with $m \leftarrow \lfloor 2n/3 \rfloor$? Justify your answer.
 - (c) State a recurrence (including the base case(s)) for the number of comparisons executed by STUPIDSORT.
 - (d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.] Does the algorithm deserve its name?
 - *(e) Show that the number of swaps executed by STUPIDSORT is at most $\binom{n}{2}$.
3. The following randomized algorithm selects the r th smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call RANDOMSELECT($A, 1$); to find the median element, you would call RANDOMSELECT($A, \lfloor n/2 \rfloor$). Recall from lecture that PARTITION splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element.

```

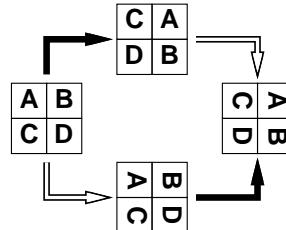
RANDOMSELECT( $A[1..n], r$ ) :
   $p \leftarrow \text{RANDOM}(1, p)$ 
   $k \leftarrow \text{PARTITION}(A[1..n], p)$ 
  if  $r < k$ 
    return RANDOMSELECT( $A[1..k-1], r$ )
  else if  $r > k$ 
    return RANDOMSELECT( $A[k+1..n], r-k$ )
  else
    return  $A[k]$ 

```

- (a) State a recurrence for the expected running time of RANDOMSELECT, as a function of n and r .
- (b) What is the *exact* probability that RANDOMSELECT compares the i th smallest and j th smallest elements in the input array? The correct answer is a simple function of i , j , and r . [Hint: Check your answer by trying a few small examples.]
- *(c) What is the expected running time of RANDOMSELECT, as a function of n and r ? You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, give the *exact* expected number of comparisons.
- (d) What is the expected number of times that RANDOMSELECT calls itself recursively?

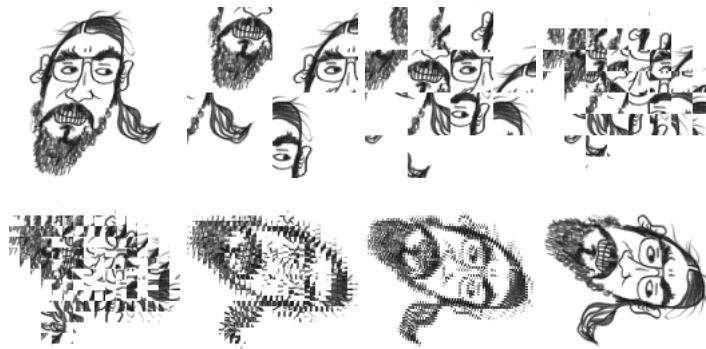
4. Some graphics hardware includes support for an operation called *blit*, or **block transfer**, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixelmap 90° clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
Black arrows indicate blitting the blocks into place.
White arrows indicate recursively rotating the blocks.

The following sequence of pictures shows the first algorithm (blit then recurse) in action.



In the following questions, assume n is a power of two.

- Prove that both versions of the algorithm are correct. [Hint: If you exploit all the available symmetries, your proof will only be a half of a page long.]
- Exactly* how many blits does the algorithm perform?
- What is the algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?
- What if a $k \times k$ blit takes only $O(k)$ time?

5. The traditional Devonian/Cornish drinking song “The Barley Mow” has the following pseudolyrics¹, where $\text{container}[i]$ is the name of a container that holds 2^i ounces of beer.²

```
BARLEYMOW( $n$ ):
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

    "We'll drink it out of the jolly brown bowl,"
    "Here's a health to the barley-mow!"
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

for  $i \leftarrow 1$  to  $n$ 
    "We'll drink it out of the container[ $i$ ], boys,"
    "Here's a health to the barley-mow!"
for  $j \leftarrow i$  downto 1
    "The container[j],"
    "And the jolly brown bowl!"
    "Here's a health to the barley-mow!"
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"
```

- (a) Suppose each container name $\text{container}[i]$ is a single word, and you can sing four words a second. How long would it take you to sing $\text{BARLEYMOW}(n)$? (Give a tight asymptotic bound.)
- (b) If you want to sing this song for $n > 20$, you’ll have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases³. Suppose $\text{container}[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing $\text{BARLEYMOW}(n)$? (Give a tight asymptotic bound.)
- (c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $\text{container}[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang $\text{BARLEYMOW}(n)$? (Give an *exact* answer, not just an asymptotic bound.)

¹Pseudolyrics are to lyrics as pseudocode is to code.

²One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

³“We'll drink it out of the hemisemidemiyottapint, boys!”

6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how ‘fun’ the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it’s her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the ‘fun’ ratings of the guests.

Practice Problems

1. Give an $O(n^2)$ algorithm to find the longest increasing subsequence of a sequence of numbers. The elements of the subsequence need not be adjacent in the sequence. For example, the sequence $\langle 1, 5, 3, 2, 4 \rangle$ has longest increasing subsequence $\langle 1, 3, 4 \rangle$.

2. You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)
 - (a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.
 - (b) Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)

3. Give an algorithm that finds the *second* smallest of n elements in at most $n + \lceil \lg n \rceil - 2$ comparisons. [Hint: divide and conquer to find the smallest; where is the second smallest?]

4. Suppose that you have an array of records whose keys to be sorted consist only of 0’s and 1’s. Give a simple, linear-time $O(n)$ algorithm to sort the array in place (using storage of no more than constant size in addition to that of the array).

5. Consider the problem of making change for n cents using the least number of coins.
- Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
 - Suppose that the available coins have the values c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the obvious greedy algorithm always yields an optimal solution.
 - Give a set of 4 coin values for which the greedy algorithm does not yield an optimal solution.
 - Describe a dynamic programming algorithm that yields an optimal solution for an arbitrary set of coin values.
 - Suppose we have only two types of coins whose values a and b are relatively prime. Prove that any value of greater than $(a - 1)(b - 1)$ can be made with these two coins.
- ★(f) For only three coins a, b, c whose greatest common divisor is 1, give an algorithm to determine the smallest value n such that change *can* be given for all values greater than n . [Note: this problem is currently unsolved for more than four coins!]
6. Suppose you have a subroutine that can find the median of a set of n items (i.e., the $\lfloor n/2 \rfloor$ smallest) in $O(n)$ time. Give an algorithm to find the k th biggest element (for arbitrary k) in $O(n)$ time.
7. You're walking along the beach and you stub your toe on something in the sand. You dig around it and find that it is a treasure chest full of gold bricks of different (integral) weight. Your knapsack can only carry up to weight n before it breaks apart. You want to put as much in it as possible without going over, but you *cannot* break the gold bricks up.
- Suppose that the gold bricks have the weights $1, 2, 4, 8, \dots, 2^k$, $k \geq 1$. Describe and prove correct a greedy algorithm that fills the knapsack as much as possible without going over.
 - Give a set of 3 weight values for which the greedy algorithm does *not* yield an optimal solution and show why.
 - Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of gold brick values.

CS 373: Combinatorial Algorithms, Fall 2000

Homework 2 (due September 28, 2000 at midnight)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Faster Longest Increasing Subsequence (15 pts)

Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. [Hint: In the dynamic programming solution, you don't really have to look back at all previous items. There was a practice problem on HW 1 that asked for an $O(n^2)$ algorithm for this. If you are having difficulty, look at the HW 1 solutions.]

2. SELECT(A, k) (10 pts)

Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

- Show that a rotation in an augmented binary tree can be performed in constant time.
- Describe an algorithm `SCAPEGOATSELECT(k)` that selects the k th smallest item in an augmented scapegoat tree in $O(\log n)$ worst-case time.
- Describe an algorithm `SPLAYSELECT(k)` that selects the k th smallest item in an augmented splay tree in $O(\log n)$ amortized time.

- (d) Describe an algorithm $\text{TREAPSELECT}(k)$ that selects the k th smallest item in an augmented treap in $O(\log n)$ *expected* time.

[Hint: The answers for (b), (c), and (d) are almost exactly the same!]

3. Scapegoat trees (15 pts)

- (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.
- (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v . A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)
- *(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory. For 5 extra credit points, use only $O(1)$ additional memory.

4. Memory Management (10 pts)

Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method—it makes the problem much too hard!

5. Fibonacci Heaps: SECONDMIN (10 pts)

- (a) Implement SECONDMIN by using a Fibonacci heap as a black box. Remember to justify its correctness and running time.
- *(b) Modify the Fibonacci Heap data structure to implement the SECONDMIN operation in constant time, without degrading the performance of any other Fibonacci heap operation.

Practice Problems

1. Amortization

- (a) Modify the binary double-counter (see class notes Sept 12) 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: Suppose p is the number of significant bits in P , and n is the number of significant bits in N . For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

- *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*2. Amortization

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 i th least significant fit indicates whether the sum includes the i th Fibonacci number F_i . For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is not the same representation as on Homework 0!]]

3. Rotations

- (a) Show that it is possible to transform any n -node binary search tree into any other n -node binary search tree using at most $2n - 2$ rotations.
- *(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.
4. Give an efficient implementation of the operation CHANGEKEY(x, k), which changes the key of a node x in a Fibonacci heap to the value k . The changes you make to Fibonacci heap data structure to support your implementation should not affect the amortized running time of any other Fibonacci heap operations. Analyze the amortized running time of your implementation for cases in which k is greater than, less than, or equal to $\text{key}[x]$.

5. Detecting overlap

- (a) You are given a list of ranges represented by min and max (e.g., [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$ -time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

- (b) You are given a list of rectangles represented by min and max x - and y -coordinates. Give an $O(n \log n)$ -time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). [Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.]

6. Comparison of Amortized Analysis Methods

A sequence of n operations is performed on a data structure. The i th operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

- (a) Aggregate method
- (b) Accounting method
- *(c) Potential method

CS 373: Combinatorial Algorithms, Fall 2000

Homework 3 (due October 17, 2000 at midnight)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

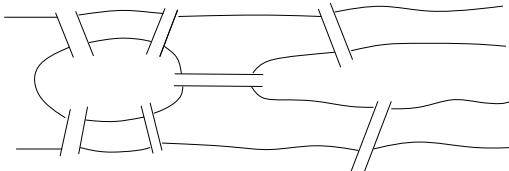
1. Suppose you have to design a dictionary that holds 2048 items.
 - (a) How many probes are used for an unsuccessful search if the dictionary is implemented as a sorted array? Assume the use of Binary Search.
 - (b) How large a hashtable do you need if your goal is to have 2 as the expected number of probes for an unsuccessful search?
 - (c) How much more space is needed by the hashtable compared to the sorted array? Assume that each pointer in a linked list takes 1 word of storage.
2. In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called ‘make’ that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of

the other files which are listed. Thus a source file must be recompiled if a file on which it depends is changed.

Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design an algorithm to recompile only those necessary. Don't worry about the details of parsing a Makefile.

3. A person wants to fly from city A to city B in the shortest possible time. She turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose a route with the minimum total travel time—initial takeoff to final landing, including layovers. [*Hint: Modify the data and call a shortest-path algorithm.*]

4. During the eighteenth century the city of Königsberg in East Prussia was divided into four sections by the Pregel river. Seven bridges connected these regions, as shown below. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to their starting point.



- (a) Show how the residents of the city could accomplish such a walk or prove no such walk exists.
- (b) Given any undirected graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge exactly once, or says that it can't be done.

5. Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).
 - (a) Give an algorithm to update the MST when an edge is added to G .
 - (b) Give an algorithm to update the MST when an edge is deleted from G .
 - (c) Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G .

6. [*This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.*]

You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B . Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

Practice Problems

- ★1. Let the hash function for a table of size m be

$$h(x) = \lfloor Ax \rfloor \bmod m$$

where $A = \hat{\phi} = \frac{\sqrt{5}-1}{2}$. Show that this gives the best possible spread, i.e. if the x are hashed in order, $x+1$ will be hashed in the largest remaining contiguous interval.

2. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = [(i, j) \in E] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

(a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose).

(b) Describe what all the entries of the matrix product B^TB represent.

★(c) Let $C = BB^T - 2A$, where A is the adjacency matrix of G , with zeroes on the diagonal.

Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees.

3. (a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.

(b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V - 2$ (the body) connected to the other $V - 3$ vertices (the feet). Some of the feet may be connected to other feet.

Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

(c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

4. Given an *undirected* graph $G = (V, E)$, and a weight function $f : E \rightarrow \mathbb{R}$ on the edges, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G .

5. Let $G = (V, E)$ be a graph with n vertices. A *simple path* of G , is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G . Hint: It can be done in $O(n^c 2^n)$ time, for some constant c .

6. Suppose all edge weights in a graph G are equal. Give an algorithm to compute a minimum spanning tree of G .

7. Give an algorithm to construct a *transitive reduction* of a directed graph G , i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR} .

8. (a) What is $5^{29^5^0 + 23^4^1 + 17^3^2 + 11^2^3 + 5^1^4} \pmod{6}$?
- (b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Fall 2000

Homework 4 (due October 26, 2000 at midnight)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grad. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

- (10 points) A certain algorithms professor once claimed that the height of an n -node Fibonacci heap is of height $O(\log n)$. Disprove his claim by showing that for a positive integer n , a sequence of Fibonacci heap operations that creates a Fibonacci heap consisting of just one tree that is a (downward) linear chain of n nodes.
- (20 points) *Fibonacci strings* are defined as follows:

$$\begin{aligned}F_1 &= b \\F_2 &= a \\F_n &= F_{n-1}F_{n-2} \quad \text{for all } n > 2\end{aligned}$$

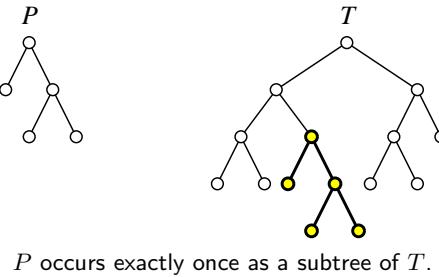
where the recursive rule uses concatenation of strings, so $F_3 = ab$, $F_4 = aba$, and so on. Note that the length of F_n is the n th Fibonacci number.

- Prove that in any Fibonacci string there are no two b's adjacent and no three a's.

- (b) Give the unoptimized and optimized failure function for F_7 .
- (c) Prove that, in searching for the Fibonacci string F_k , the unoptimized KMP algorithm may shift $\lceil k/2 \rceil$ times on the same text character. In other words, prove that there is a chain of failure links $j \rightarrow \text{fail}[j] \rightarrow \text{fail}[\text{fail}[j]] \rightarrow \dots$ of length $\lceil k/2 \rceil$, and find an example text T that would cause KMP to traverse this entire chain on the same position in the text.
- (d) What happens here when you use the optimized prefix function? Explain.
3. (10 points) Show how to extend the Rabin-Karp fingerprinting method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. The pattern may be shifted horizontally and vertically, but it may not be rotated.

4. (10 points)

- (a) A *cyclic rotation* of a string is obtained by chopping off a prefix and gluing it at the end of the string. For example, ALGORITHM is a cyclic shift of RITHMALGO. Describe and analyze an algorithm that determines whether one string $P[1..m]$ is a cyclic rotation of another string $T[1..n]$.
- (b) Describe and analyze an algorithm that decides, given any two binary trees P and T , whether P equals a subtree of T . We want an algorithm that compares the *shapes* of the trees. There is no data stored in the nodes, just pointers to the left and right children. [Hint: First transform both trees into strings.]



5. (10 points) [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Refer to the notes for lecture 11 for this problem. The GENERICSSSP algorithm described in class can be implemented using a stack for the ‘bag’. Prove that the resulting algorithm can be forced to perform in $\Omega(2^n)$ relaxation steps. To do this, you need to describe, for any positive integer n , a specific weighted directed n -vertex graph that forces this exponential behavior. The easiest way to describe such a family of graphs is using an *algorithm*!

Practice Problems

1. String matching with wild-cards

Suppose you have an alphabet for patterns that includes a ‘gap’ or wild-card character; any length string of any characters can match this additional character. For example if ‘*’ is the wild-card, then the pattern foo*bar*nad can be found in foofoowangbarnad. Modify the computation of the prefix function to correctly match strings using KMP.

2. Prove that there is no comparison sort whose running time is linear for at least $1/2$ of the $n!$ inputs of length n . What about at least $1/n$? What about at least $1/2^n$?
3. Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.
4. Find asymptotic upper and lower bounds to $\lg(n!)$ without Stirling’s approximation (Hint: use integration).
5. Given a sequence of n elements of n/k blocks (k elements per block) all elements in a block are less than those to the right in sequence, show that you cannot have the whole sequence sorted in better than $\Omega(n \lg k)$. Note that the entire sequence would be sorted if each of the n/k blocks were individually sorted in place. Also note that combining the lower bounds for each block is not adequate (that only gives an upper bound).
6. Show how to find the occurrences of pattern P in text T by computing the prefix function of the string PT (the concatenation of P and T).
7. Lower Bounds on Adjacency Matrix Representations of Graphs
 - (a) Prove that the time to determine if an undirected graph has a cycle is $\Omega(V^2)$.
 - (b) Prove that the time to determine if there is a path between two nodes in an undirected graph is $\Omega(V^2)$.

CS 373: Combinatorial Algorithms, Fall 2000

Homework 1 (due November 16, 2000 at midnight)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

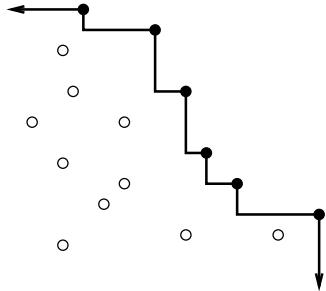
Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Give an $O(n^2 \log n)$ algorithm to determine whether any three points of a set of n points are collinear. Assume two dimensions and *exact* arithmetic.

2. We are given an array of n bits, and we want to determine if it contains two consecutive 1 bits. Obviously, we can check every bit, but is this always necessary?
 - (a) (4 pts) Show that when $n \bmod 3 = 0$ or 2, we must examine every bit in the array. That is, give an adversary strategy that forces any algorithm to examine every bit when $n = 2, 3, 5, 6, 8, 9, \dots$.
 - (b) (4 pts) Show that when $n = 3k + 1$, we only have to examine $n - 1$ bits. That is, describe an algorithm that finds two consecutive 1s or correctly reports that there are none after examining at most $n - 1$ bits, when $n = 1, 4, 7, 10, \dots$.
 - (c) (2 pts) How many n -bit strings are there with two consecutive ones? For which n is this number even or odd?

3. You are given a set of points in the plane. A point is *maximal* if there is no other point both above and to the right. The subset of maximal points of points then forms a *staircase*.

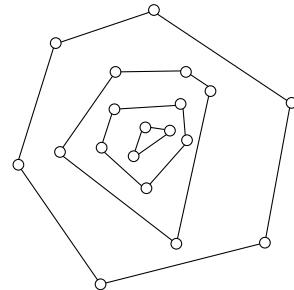


The staircase of a set of points. Maximal points are black.

- (a) (0 pts) Prove that maximal points are *not* necessarily on the convex hull.
- (b) (6 pts) Give an $O(n \log n)$ algorithm to find the maximal points.
- (c) (4 pts) Assume that points are chosen uniformly at random within a rectangle. What is the average number of maximal points? Justify. Hint: you will be able to give an exact answer rather than just asymptotics. You have seen the same analysis before.

4. Given a set Q of points in the plane, define the *convex layers* of Q inductively as follows: The first convex layer of Q is just the convex hull of Q . For all $i > 1$, the i th convex layer is the convex hull of Q after the vertices of the first $i - 1$ layers have been removed.

Give an $O(n^2)$ -time algorithm to find all convex layers of a given set of n points.



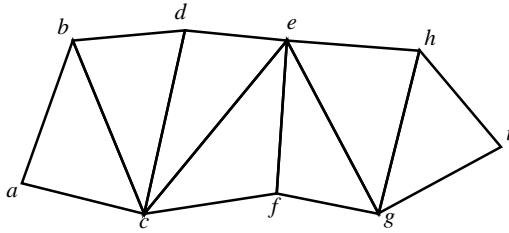
A set of points with four convex layers.

5. Prove that finding the second smallest of n elements takes $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. Prove for both upper and lower bounds. Hint: find the (first) smallest using an elimination tournament.

6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Almost all computer graphics systems, at some level, represent objects as collections of triangles. In order to minimize storage space and rendering time, many systems allow objects to be stored as a set of *triangle strips*. A triangle strip is a sequence of vertices $\langle v_1, v_2, \dots, v_k \rangle$, where each contiguous triple of vertices v_i, v_{i+1}, v_{i+2} represents a triangle. As the rendering system reads the sequence of vertices and draws the triangles, it keeps the two most recent vertices in a cache.

Some systems allow triangle strips to contain *swaps*: special flags indicating that the order of the two cached vertices should be reversed. For example, the triangle strip $\langle a, b, c, d, \text{swap}, e, f, \text{swap}, g, h, i \rangle$ represents the sequence of triangles $(a, b, c), (b, c, d), (d, c, e), (c, e, f), (f, e, g)$.



Two triangle strips are *disjoint* if they share no triangles (although they may share vertices). The *length* of a triangle strip is the length of its vertex sequence, including swaps; for example, the example strip above has length 11. A *pure* triangle strip is one with no swaps. The adjacency graph of a triangle strip is a simple path. If the strip is pure, this path alternates between left and right turns.

Suppose you are given a set S of interior-disjoint triangles whose adjacency graph is a tree. (In other words, S is a triangulation of a simple polygon.) Describe a linear-time algorithm to decompose S into a set of disjoint triangle strips of minimum total length.

Practice Problems

1. Consider the following generic recurrence for convex hull algorithms that divide and conquer:

$$T(n, h) = T(n_1, h_1) + T(n_2, h_2) + O(n)$$

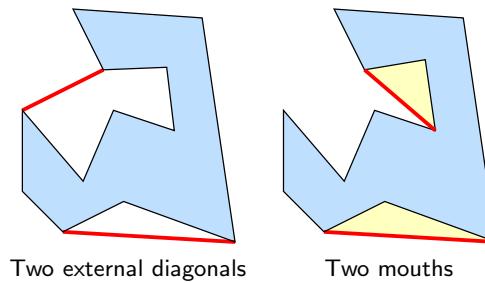
where $n \geq n_1 + n_2$, $h = h_1 + h_2$ and $n \geq h$. This means that the time to compute the convex hull is a function of both n , the number of input points, and h , the number of convex hull vertices. The splitting and merging parts of the divide-and-conquer algorithm take $O(n)$ time. When n is a constant, $T(n, h) = O(1)$, but when h is a constant, $T(n, h) = O(n)$. Prove that for both of the following restrictions, the solution to the recurrence is $O(n \log h)$:

- (a) $h_1, h_2 < \frac{3}{4}h$
- (b) $n_1, n_2 < \frac{3}{4}n$

2. Circle Intersection

Give an $O(n \log n)$ algorithm to test whether any two circles in a set of size n intersect.

3. Basic polygon computations (assume *exact* arithmetic)
- Intersection: Extend the basic algorithm to determine if two line segments intersect by taking care of *all* degenerate cases.
 - Simplicity: Give an $O(n \log n)$ algorithm to determine whether an n -vertex polygon is simple.
 - Area: Give an algorithm to compute the area of a simple n -polygon (not necessarily convex) in $O(n)$ time.
 - Inside: Give an algorithm to determine whether a point is within a simple n -polygon (not necessarily convex) in $O(n)$ time.
4. We are given the set of points one point at a time. After receiving each point, we must compute the convex hull of all those points so far. Give an algorithm to solve this problem in $O(n^2)$ total time. (We could obviously use Graham's scan n times for an $O(n^2 \log n)$ -time algorithm). Hint: How do you maintain the convex hull?
5. *(a) Given an n -polygon and a point outside the polygon, give an algorithm to find a tangent.
(b) Suppose you have found both tangents. Give an algorithm to remove the points from the polygon that are within the angle formed by the tangents (as segments!) and the opposite side of the polygon.
(c) Use the above to give an algorithm to compute the convex hull on-line in $O(n \log n)$
6. (a) A pair of polygon vertices defines an *external diagonal* if the line segment between them is completely outside the polygon. Show that every nonconvex polygon has at least one external diagonal.
(b) Three consecutive polygon vertices p, q, r form a *mouth* if p and r define an external diagonal. Show that every nonconvex polygon has at least one mouth.



7. A group of n ghostbusters is battling n ghosts. Each ghostbuster can shoot a single energy beam at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The ghostbusters all fire at the same time and no two energy beams may cross. The positions of the ghosts and ghostbusters are fixed points in the plane.
- Prove that for any configuration of ghosts and ghostbusters, there is such a non-crossing matching. (Assume that no three points are collinear.)

- (b) Show that there is a line passing through one ghostbuster and one ghost such that the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Give an efficient algorithm to find such a line.
- (c) Give an efficient divide and conquer algorithm to pair ghostbusters and ghosts so that no two streams cross.

CS 373: Combinatorial Algorithms, Fall 2000

Homework 6 (due December 7, 2000 at midnight)

Name:		
Net ID:	Alias:	<input type="checkbox"/> U <input checked="" type="checkbox"/> <input type="checkbox"/> 1

Name:		
Net ID:	Alias:	<input type="checkbox"/> U <input checked="" type="checkbox"/> <input type="checkbox"/> 1

Name:		
Net ID:	Alias:	<input type="checkbox"/> U <input checked="" type="checkbox"/> <input type="checkbox"/> 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

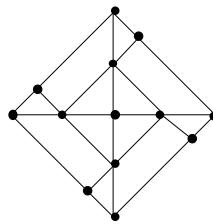
1. (a) Prove that $P \subseteq \text{co-NP}$.
(b) Show that if $NP \neq \text{co-NP}$, then *no* NP-complete problem is a member of co-NP.
2. 2SAT is a special case of the formula satisfiability problem, where the input formula is in conjunctive normal form and every clause has at most two literals. Prove that 2SAT is in P.
3. Describe an algorithm that solves the following problem, called 3SUM, as quickly as possible: Given a set of n numbers, does it contain three elements whose sum is zero? For example, your algorithm should answer TRUE for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5+7+(-2)=0$, and FALSE for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.

4. (a) Show that the problem of deciding whether one undirected graph is a subgraph of another is NP-complete.
(b) Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than k is NP-complete.
 5. (a) Consider the following problem: Given a set of axis-aligned rectangles in the plane, decide whether any point in the plane is covered by k or more rectangles. Now also consider the CLIQUE problem. Describe and analyze a reduction of one problem to the other.
(b) Finding the largest clique in an arbitrary graph is NP-hard. What does this fact imply about the complexity of finding a point that lies inside the largest number of rectangles?
6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

PARTITION is the problem of deciding, given a set $S = \{s_1, s_2, \dots, s_n\}$ of numbers, whether there is a subset T containing half the 'weight' of S , i.e., such that $\sum T = \frac{1}{2} \sum S$. SUBSETSUM is the problem of deciding, given a set $S = \{s_1, s_2, \dots, s_n\}$ of numbers and a target sum t , whether there is a subset $T \subseteq S$ such that $\sum T = t$. Give two reductions between these two problems, one in each direction.

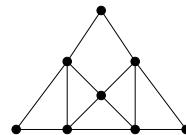
Practice Problems

1. What is the *exact* worst case number of comparisons needed to find the median of 5 numbers?
For 6 numbers?
2. The EXACTCOVERBYTHREES problem is defined as follows: given a finite set X and a collection C of 3-element subsets of X , does C contain an *exact cover* for X , that is, a subcollection $C' \subseteq C$ where every element of X occurs in exactly one member of C' ? Given that EXACTCOVERBYTHREES is NP-complete, show that the similar problem EXACTCOVERBYFOURS is also NP-complete.
3. Using 3COLOR and the ‘gadget’ below, prove that the problem of deciding whether a planar graph can be 3-colored 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.]



Crossing gadget for PLANAR3COLOR.

4. Using the previous result, and the ‘gadget’ below, prove that the problem of deciding whether a planar graph with no vertex of degree greater than four can be 3-colored is NP-complete. [Hint: Show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.]



Degree gadget for DEGREE4PLANAR3COLOR

5. Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.
6. (a) Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian. Give a polynomial time algorithm for finding a hamiltonian cycle in an undirected bipartite graph or establishing that it does not exist.
(b) Show that the hamiltonian-path problem can be solved in polynomial time on directed acyclic graphs.
(c) Explain why the results in previous questions do not contradict the fact that both HAMILTONIANCYCLE and HAMILTONIANPATH are NP-complete problems.
7. Consider the following pairs of problems:

- (a) MIN SPANNING TREE and MAX SPANNING TREE
- (b) SHORTEST PATH and LONGEST PATH
- (c) TRAVELING SALESMAN and VACATION TOUR (the longest tour is sought).
- (d) MIN CUT and MAX CUT (between s and t)
- (e) EDGE COVER and VERTEX COVER
- (f) TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

(All of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph.) Which of these pairs are polytime equivalent and which are not?

*8. Consider the problem of deciding whether one graph is isomorphic to another.

- (a) Give a brute force algorithm to decide this.
- (b) Give a dynamic programming algorithm to decide this.
- (c) Give an efficient probabilistic algorithm to decide this.
- ★(d) Either prove that this problem is NP-complete, give a poly time algorithm for it, or prove that neither case occurs.

*9. Prove that PRIMALITY (Given n , is n prime?) is in $\text{NP} \cap \text{co-NP}$. Showing that PRIMALITY is in co-NP is easy. (What's a certificate for showing that a number is composite?) For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that this tree of primitive roots can be checked to be correct and used to show that n is prime, and that this check takes polynomial time.

10. How much wood would a woodchuck chuck if a woodchuck could chuck wood?

CS 373: Combinatorial Algorithms, Fall 2000

Midterm 1 — October 3, 2000

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your $8\frac{1}{2}'' \times 11''$ cheat sheet, please leave it at the front of the classroom.

-
- Print your name, netid, and alias in the boxes above. Circle U if you are an undergrad, $\frac{3}{4}$ if you are a 3/4-unit grad student, or 1 if you are a 1-unit grad student. Print your name at the top of every page (in case the staple falls out!).
 - **Answer four of the five questions on the exam.** Each question is worth 10 points. If you answer more than four questions, the one with the lowest score will be ignored. **1-unit graduate students must answer question 5.**
 - Please write your final answers on the front of the exam pages. Use the backs of the pages as scratch paper. Let us know if you need more paper.
 - Unless we specifically say otherwise, proofs are not required. However, they may help us give you partial credit.
 - Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
 - Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
 - Write something down for every problem. Don't panic and erase large chunks of work. Even if you think it's absolute nonsense, it might be worth partial credit.
 - Relax. Breathe. Kick some ass.

#	Score	Grader
1		
2		
3		
4		
5		
Total		

1. Multiple Choice

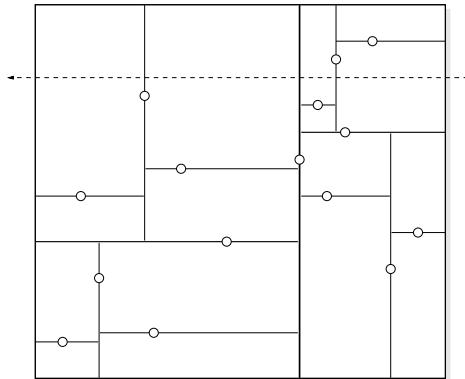
Every question below has one of the following answers.

- (a) $\Theta(1)$ (b) $\Theta(\log n)$ (c) $\Theta(n)$ (d) $\Theta(n \log n)$ (e) $\Theta(n^2)$

For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point, but each incorrect answer *costs* you $\frac{1}{2}$ point. You cannot score below zero.

- What is $\sum_{i=1}^n \log i$?
- What is $\sum_{i=1}^n \frac{n}{i}$?
- How many digits do you need to write 2^n in decimal?
- What is the solution of the recurrence $T(n) = 25T(n/5) + n$?
- What is the solution of the recurrence $T(n) = T(n - 1) + \frac{1}{2^n}$?
- What is the solution of the recurrence $T(n) = 3T\left(\lceil \frac{n+51}{3} \rceil\right) + 17n - \sqrt[3]{\lg \lg n} - 2^{2^{\log^* n}} + \pi$?
- What is the worst-case running time of randomized quicksort?
- The expected time for inserting one item into an n -node randomized treap is $O(\log n)$.
What is the worst-case time for a sequence of n insertions into an initially empty treap?
- The amortized time for inserting one item into an n -node scapegoat tree is $O(\log n)$.
What is the worst-case time for a sequence of n insertions into an initially empty scapegoat tree?
- In the worst case, how many nodes can be in the root list of a Fibonacci heap storing n keys, immediately after a DECREASEKEY operation?
- Every morning, an Amtrak train leaves Chicago for Champaign, 200 miles away. The train can accelerate or decelerate at 10 miles per hour per second, and it has a maximum speed of 60 miles an hour. Every 50 miles, the train must stop for five minutes while a school bus crosses the tracks. Every hour, the conductor stops the train for a union-mandated 10-minute coffee break. How long does it take the train to reach Champaign?

2. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the rectangle as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line passes through some point inside the box (*not* on the boundary) and partitions the rest of the interior points as evenly as possible. If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.

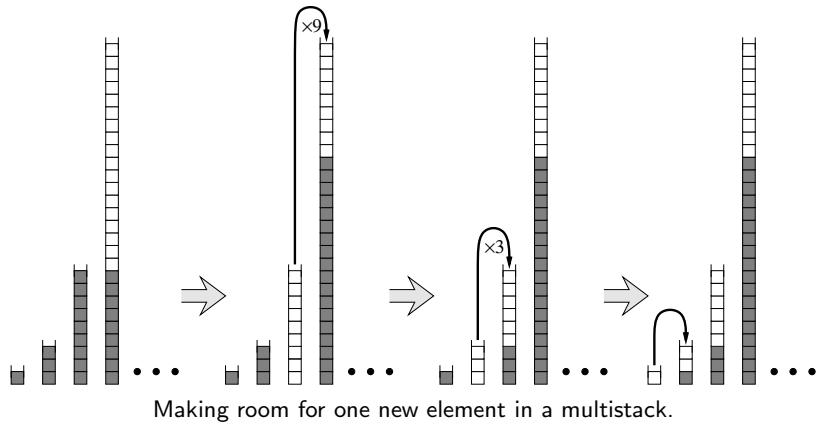


A kd-tree for 15 points. The dashed line crosses four cells.

- (a) [2 points] How many cells are there, as a function of n ? Prove your answer is correct.
- (b) [8 points] In the worst case, exactly how many cells can a horizontal line cross, as a function of n ? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k .
[For full credit, you must give an exact answer. A tight asymptotic bound (with proof) is worth 5 points. A correct recurrence is worth 3 points.]
- (c) [5 points extra credit] In the worst case, how many cells can a diagonal line cross?

Incidentally, ‘kd-tree’ originally meant ‘ k -dimensional tree’—for example, the specific data structure described here used to be called a ‘2d-tree’—but current usage ignores this etymology. The phrase ‘ d -dimensional kd-tree’ is now considered perfectly standard, even though it’s just as redundant as ‘ATM machine’, ‘PIN number’, ‘HIV virus’, ‘PDF format’, ‘Mt. Fujiyama’, ‘Sahara Desert’, ‘The La Brea Tar Pits’, or ‘and etc.’ On the other hand, ‘BASIC code’ is *not* redundant; ‘Beginner’s All-Purpose Instruction Code’ is a backronym. Hey, aren’t you supposed to be taking a test?

3. A *multistack* consists of an infinite series of stacks S_0, S_1, S_2, \dots , where the i th stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i , we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2} , and so on. Moving a single element from one stack to the next takes $O(1)$ time.



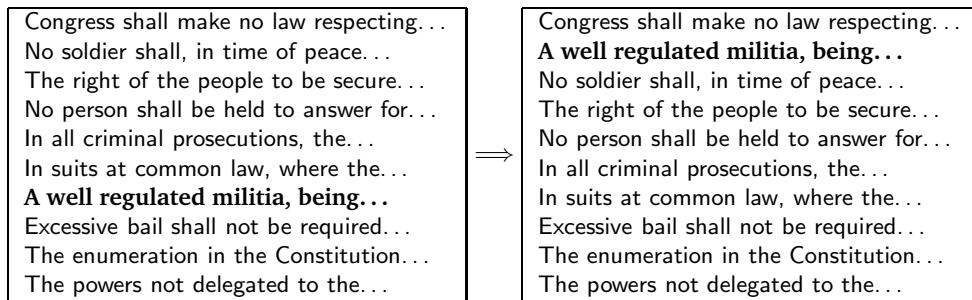
- (a) [1 point] In the worst case, how long does it take to push one more element onto a multistack containing n elements?
- (b) [9 points] Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.

4. After graduating with a computer science degree, you find yourself working for a software company that publishes a word processor. The program stores a document containing n characters, grouped into p paragraphs. Your manager asks you to implement a ‘Sort Paragraphs’ command that rearranges the paragraphs into alphabetical order.

Design and analyze an efficient paragraph-sorting algorithm, using the following pair of routines as black boxes.

- COMPAREPARAGRAPHS(i, j) compares the i th and j th paragraphs, and returns i or j depending on which paragraph should come first in the final sorted output. (Don’t worry about ties.) This function runs in $O(1)$ time, since almost any two paragraphs can be compared by looking at just their first few characters!
- MOVEPARAGRAPH(i, j) ‘cuts’ out the i th paragraph and ‘pastes’ it back in as the j th paragraph. This function runs in $O(n_i)$ time, where n_i is the number of characters in the i th paragraph. (So in particular, $n_1 + n_2 + \dots + n_p = n$.)

Here is an example of MOVEPARAGRAPH(7, 2):



[Hint: For full credit, your algorithm should run in $o(n \log n)$ time when $p = o(n)$.]

5. [1-unit grad students must answer this question.]

Describe and analyze an algorithm to randomly shuffle an array of n items, so that each of the $n!$ possible permutations is equally likely. Assume that you have a function RANDOM(i, j) that returns a random integer from the set $\{i, i + 1, \dots, j\}$ in constant time.

[Hint: As a sanity check, you might want to confirm that for $n = 3$, all six permutations have probability $1/6$. For full credit, your algorithm must run in $\Theta(n)$ time. A correct algorithm that runs in $\Theta(n \log n)$ time is worth 7 points.]

From: "Josh Pepper" <jwpepper@uiuc.edu>
To: "Chris Neihengen" <neihenge@uiuc.edu>
Subject: FW: proof
Date: Fri, 29 Sep 2000 09:34:56 -0500

thought you might like this.

Problem: To prove that computer science 373 is indeed the work of Satan.

Proof: First, let us assume that everything in "Helping Yourself with Numerology", by Helyn Hitchcock, is true.

Second, let us apply divide and conquer to this problem. There are main parts:

1. The name of the course: "Combinatorial Algorithms"
2. The most important individual in the course, the "Recursion Fairy"
3. The number of this course: 373.

We examine these sequentially.

The name of the course. "Combinatorial Algorithms" can actually be expressed as a single integer - 23 - since it has 23 letters.

The most important individual, the Recursion Fairy, can also be expressed as a single integer - 14 - since it has 14 letters. In other words:

COMBINATORIAL ALGORITHMS = 23
RECUSION FAIRY = 14

As a side note, a much shorter proof has already been published showing that the Recursion Fairy is Lucifer, and that any class involving the Fairy is from Lucifer, however, that proofs numerical significance is slight.

Now we can move on to an analysis of the number of course, which holds great meaning. The first assumption we make is that the number of the course, 373, is not actually a base 10 number. We can prove this inductively by making a reasonable guess for the actual base, then finding a new way to express the nature of the course, and if the answer

confirms what we assumed, then we're right. That's the way induction works.

What is a reasonable guess for the base of the course? The answer is trivial, since the basest of all beings is the Recursion Fairy, the base is 14. So a true base 10 representation of 373 (base 14) is 689. So we see:

373 (base 14) = 689 (base 10)

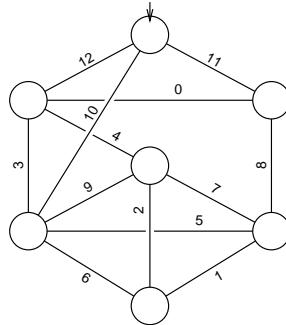
Now since the nature of the course has absolutely nothing to do with combinatorial algorithms (instead having much to do with the work of the devil), we can subtract from the above result everything having to do with combinatorial algorithms just by subtracting 23. Here we see that:

689 - 23 = 666

QED.

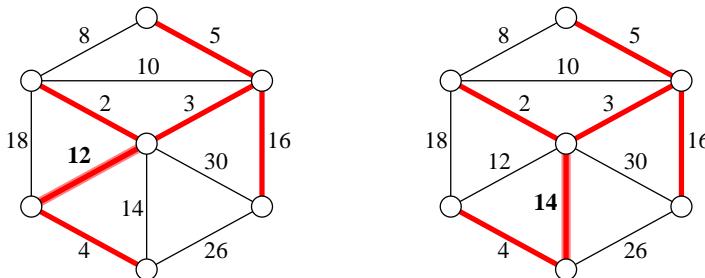
1. Using any method you like, compute the following subgraphs for the weighted graph below. Each subproblem is worth 3 points. Each incorrect edge costs you 1 point, but you cannot get a negative score for any subproblem.

- (a) a depth-first search tree, starting at the top vertex;
- (b) a breadth-first search tree, starting at the top vertex;
- (c) a shortest path tree, starting at the top vertex;
- (d) the minimum spanning tree.



2. Suppose you are given a weighted undirected graph G (represented as an adjacency list) and its minimum spanning tree T (which you already know how to compute). Describe and analyze an algorithm to find the *second-minimum spanning tree* of G , i.e., the spanning tree of G with smallest total weight except for T .

The minimum spanning tree and the second-minimum spanning tree differ by exactly one edge. But *which* edge is different, and *how* is it different? That's what your algorithm has to figure out!

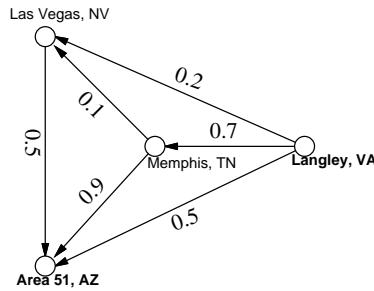


The minimum spanning tree and the second-minimum spanning tree of a graph.

3. (a) [4 pts] Prove that a connected acyclic graph with V vertices has exactly $V - 1$ edges. (“It’s a tree!” is not a proof.)
- (b) [4 pts] Describe and analyze an algorithm that determines whether a given graph is a tree, where the graph is represented by an adjacency list.
- (c) [2 pts] What is the running time of your algorithm from part (b) if the graph is represented by an adjacency matrix?

4. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t .



With the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted!¹

5. [1-unit grad students must answer this question.]

Many string matching applications allow the following *wild card* characters in the pattern.

- The wild card ? represents an arbitrary single character. For example, the pattern s?r?ng matches the strings string, sprung, and sarong.
- The wild card * represents an arbitrary string of zero or more characters. For example, the pattern te*st* matches the strings test, tensest, and technostructuralism.

Both wild cards can occur in a single pattern. For example, the pattern f*a?? matches the strings face, football, and flippyfloppydingdongdang. On the other hand, neither wild card can occur in the text.

Describe how to modify the Knuth-Morris-Pratt algorithm to support patterns with these wild cards, and analyze the modified algorithm. Your algorithm should find the first substring in the text that matches the pattern. An algorithm that supports only one of the two wild cards is worth 5 points.

¹That's how they got Elvis, you know.

1. True, False, or Maybe

Indicate whether each of the following statements is always true, sometimes true, always false, or unknown. Some of these questions are deliberately tricky, so read them carefully. Each correct choice is worth +1, and each incorrect choice is worth -1. **Guessing will hurt you!**

- (a) Suppose SMARTALGORITHM runs in $\Theta(n^2)$ time and DUMBALGORITHM runs in $\Theta(2^n)$ time for all inputs of size n . (Thus, for each algorithm, the best-case and worst-case running times are the same.) SMARTALGORITHM is faster than DUMBALGORITHM.

True False Sometimes Nobody Knows

- (b) QUICKSORT runs in $O(n^6)$ time.

True False Sometimes Nobody Knows

- (c) $\lfloor \log_2 n \rfloor \geq \lceil \log_2 n \rceil$

True False Sometimes Nobody Knows

- (d) The recurrence $F(n) = n + 2\sqrt{n} \cdot F(\sqrt{n})$ has the solution $F(n) = \Theta(n \log n)$.

True False Sometimes Nobody Knows

- (e) A Fibonacci heap with n nodes has depth $\Omega(\log n)$.

True False Sometimes Nobody Knows

- (f) Suppose a graph G is represented by an adjacency matrix. It is possible to determine whether G is an independent set without looking at every entry of the adjacency matrix.

True False Sometimes Nobody Knows

- (g) $NP \neq co\text{-}NP$

True False Sometimes Nobody Knows

- (h) Finding the smallest clique in a graph is NP-hard.

True False Sometimes Nobody Knows

- (i) A polynomial-time reduction from X to 3SAT proves that X is NP-hard.

True False Sometimes Nobody Knows

- (j) The correct answer for exactly three of these questions is “False”.

True False

2. Convex Hull

Suppose you are given the convex hull of a set of n points, and one additional point (x, y) . The convex hull is represented by an array of vertices in counterclockwise order, starting from the leftmost vertex. Describe how to test in $O(\log n)$ time whether or not the additional point (x, y) is inside the convex hull.

3. Finding the Largest Block

In your new job, you are working with screen images. These are represented using two dimensional arrays where each element is a 1 or a 0, indicating whether that position of the screen is illuminated. Design and analyze an efficient algorithm to find the largest rectangular block of ones in such an array. For example, the largest rectangular block of ones in the array shown below is in rows 2–4 and columns 2–3. [Hint: Use dynamic programming.]

1	0	1	0	0
1	1	1	0	1
0	1	1	1	1
1	1	1	0	0

4. The Hogwarts Sorting Hat

Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line.

After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took CS 373 many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use?

More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Design and analyze an algorithm that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.

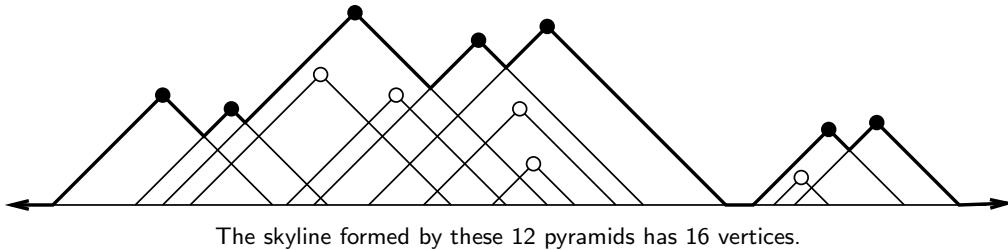
The diagram shows two horizontal arrays of 20 items each. The top array represents the initial state, and the bottom array represents the final state after sorting. Each item is a pair of letters (e.g., G, H, R, S) followed by a name. In the initial array, the items are mixed. In the final array, the items are grouped into four distinct clusters: G (Harry, Ekta, Bill, Chad, Nick, Mel, Dana, Fox, Ann, Jim, Dawn, Bart, Lisa, Tina, John, Bob, Liz, Mary, Kim, Jeff), H (Dawn, Bart, Lisa, Tina, John, Bob, Liz, Mary, Kim, Jeff), R (Harry, Ekta, Bill, Chad, Nick, Mel, Dana, Fox, Ann, Jim, Dawn, Bart, Lisa, Tina, John, Bob, Liz, Mary, Kim, Jeff), and S (Nick, Kim, Fox, Dana, Mel).

G	H	R	R	G	G	R	G	H	H	R	R	S	R	R	H	G	S	H	G	G
Harry	Ann	Bob	Tina	Chad	Bill	Lisa	Ekta	Bart	Jim	John	Jeff	Liz	Mary	Dawn	Nick	Kim	Fox	Dana	Mel	

G	G	G	G	G	G	G	H	H	H	H	H	R	R	R	R	R	R	R	S	S
Harry	Ekta	Bill	Chad	Nick	Mel	Dana	Fox	Ann	Jim	Dawn	Bart	Lisa	Tina	John	Bob	Liz	Mary	Kim	Jeff	

5. The Egyptian Skyline

Suppose you are given a set of n pyramids in the plane. Each pyramid is a isosceles triangle with two 45° edges and a horizontal edge on the x -axis. Each pyramid is represented by the x - and y -coordinates of its topmost point. Your task is to compute the “skyline” formed by these pyramids (the dark line shown below).



- (a) Describe and analyze an algorithm that determines which pyramids are visible on the skyline. These are the pyramids with black points in the figure above; the pyramids with white points are not visible. [Hint: You've seen this problem before.]
- (b) Once you know which pyramids are visible, how would you compute the *shape* of the skyline? Describe and analyze an algorithm to compute the left-to-right sequence of skyline vertices, including the vertices between the pyramids and on the ground.

6. DNF-SAT

A boolean formula is in *disjunctive normal form* (DNF) if it consists of clauses of conjunctions (ANDs) joined together by disjunctions (ORs). For example, the formula

$$(\bar{a} \wedge b \wedge \bar{c}) \vee (b \wedge c) \vee (a \wedge \bar{b} \wedge \bar{c})$$

is in disjunctive normal form. DNF-SAT is the problem that asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

- (a) Show that DNF-SAT is in P.
- (b) What is wrong with the following argument that P=NP?

Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,

$$(a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b}) \iff (a \wedge \bar{b}) \vee (b \wedge \bar{a}) \vee (\bar{c} \wedge \bar{a}) \vee (\bar{c} \wedge \bar{b})$$

Now we can use the answer to part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time! Since 3SAT is NP-hard, we must conclude that P=NP.

7. Magic 3-Coloring [1-unit graduate students must answer this question.]

The recursion fairy's distant cousin, the reduction genie, shows up one day with a magical gift for you—a box that determines in constant time whether or not a graph is 3-colorable. (A graph is 3-colorable if you can color each of the vertices red, green, or blue, so that every edge has different colors.) The magic box does not tell you *how* to color the graph, just whether or not it can be done. Devise and analyze an algorithm to 3-color any graph **in polynomial time** using this magic box.

CS 373: Combinatorial Algorithms, Spring 2001

Homework 0, due January 23, 2001 at the beginning of class

Name:	
Net ID:	Alias:

Neatly print your name (first name first, with no comma), your network ID, and a short alias into the boxes above. **Do not sign your name. Do not write your Social Security number.** Staple this sheet of paper to the top of your homework.

Grades will be listed on the course web site by alias give us, so your alias should not resemble your name or your Net ID. If you don't give yourself an alias, we'll give you one that you won't like.

This homework tests your familiarity with the prerequisite material from CS 173, CS 225, and CS 273—many of these problems have appeared on homeworks or exams in those classes—primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Parberry and Chapters 1–6 of CLR should be sufficient review, but you may want to consult other texts as well.

Before you do anything else, read the Homework Instructions and FAQ on the CS 373 course web page (<http://www-courses.cs.uiuc.edu/~cs373/hw/faq.html>), and then check the box below. This web page gives instructions on how to write and submit homeworks—staple your solutions together in order, write your name and netID on every page, don't turn in source code, analyze everything, use good English and good logic, and so forth.

I have read the CS 373 Homework Instructions and FAQ.

Required Problems

1. (a) Prove that any positive integer can be written as the sum of distinct powers of 2. For example: $42 = 2^5 + 2^3 + 2^1$, $25 = 2^4 + 2^3 + 2^0$, $17 = 2^4 + 2^0$. [Hint: ‘Write the number in binary’ is not a proof; it just restates the problem.]
- (b) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.
- (c) Prove that *any* integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example: $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.

2. Sort the following 20 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice.

$$\begin{array}{ccccc}
 1 & n & n^2 & \lg n & \lg^* n \\
 2^{2^{\lg \lg n + 1}} & \lg^* 2^n & 2^{\lg^* n} & \lfloor \lg(n!) \rfloor & \lfloor \lg n \rfloor! \\
 n^{\lg n} & (\lg n)^n & (\lg n)^{\lg n} & n^{1/\lg n} & n^{\lg \lg n} \\
 \log_{1000} n & \lg^{1000} n & \lg^{(1000)} n & \left(1 + \frac{1}{1000}\right)^n & n^{1/1000}
 \end{array}$$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

3. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. Extra credit will be given for more exact solutions.

- (a) $A(n) = 5A(n/3) + n \log n$
- (b) $B(n) = \min_{0 < k < n} (B(k) + B(n - k) + 1)$.
- (c) $C(n) = 4C(\lfloor n/2 \rfloor + 5) + n^2$
- (d) $D(n) = D(n - 1) + 1/n$
- *(e) $E(n) = n + 2\sqrt{n} \cdot E(\sqrt{n})$

4. This problem asks you to simplify some recursively defined boolean formulas as much as possible. In each case, prove that your answer is correct. Each proof can be just a few sentences long, but it must be a *proof*.

- (a) Suppose $\alpha_0 = p$, $\alpha_1 = q$, and $\alpha_n = (\alpha_{n-2} \wedge \alpha_{n-1})$ for all $n \geq 2$. Simplify α_n as much as possible. [Hint: What is α_5 ?]
- (b) Suppose $\beta_0 = p$, $\beta_1 = q$, and $\beta_n = (\beta_{n-2} \Leftrightarrow \beta_{n-1})$ for all $n \geq 2$. Simplify β_n as much as possible. [Hint: What is β_5 ?]
- (c) Suppose $\gamma_0 = p$, $\gamma_1 = q$, and $\gamma_n = (\gamma_{n-2} \Rightarrow \gamma_{n-1})$ for all $n \geq 2$. Simplify γ_n as much as possible. [Hint: What is γ_5 ?]
- (d) Suppose $\delta_0 = p$, $\delta_1 = q$, and $\delta_n = (\delta_{n-2} \bowtie \delta_{n-1})$ for all $n \geq 2$, where \bowtie is some boolean function with two arguments. Find a boolean function \bowtie such that $\delta_n = \delta_m$ if and only if $n - m$ is a multiple of 4. [Hint: There is only one such function.]

5. Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line.

After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took CS 373 many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use?

More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Describe an algorithm¹ that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.

G Harry	H Ann	R Bob	R Tina	G Chad	G Bill	R Lisa	G Ekta	H Bart	H Jim	R John	S Jeff	R Liz	R Mary	H Dawn	G Nick	S Kim	H Fox	G Dana	G Mel
↓																			
G Harry	G Ekta	G Bill	G Chad	G Nick	G Mel	G Dana	H Fox	H Ann	H Jim	H Dawn	H Bart	R Lisa	R Tina	R John	R Bob	R Liz	R Mary	S Kim	S Jeff

6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, . . . , 52 of clubs. (They’re big cards.) Penn shuffles the deck until each each of the 52! possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.

- (a) On average, how many cards does Penn give Teller?
- (b) On average, what is the smallest-numbered card that Penn gives Teller?
- *(c) On average, what is the largest-numbered card that Penn gives Teller?

[Hint: Solve for an n -card deck and then set $n = 52$.] In each case, give *exact* answers and prove that they are correct. If you have to appeal to “intuition” or “common sense”, your answers are probably wrong!

¹Since you’ve read the Homework Instructions, you know what the phrase ‘describe an algorithm’ means. Right?

Practice Problems

The remaining problems are entirely for your benefit; similar questions will appear in every homework. Don't turn in solutions—we'll just throw them out—but feel free to ask us about practice questions during office hours and review sessions. Think of them as potential exam questions (hint, hint). We'll post solutions to *some* of the practice problems after the homeworks are due.

1. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m .

(a) F_n is even if and only if n is divisible by 3.

(b) $\sum_{i=0}^n F_i = F_{n+2} - 1$

(c) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$

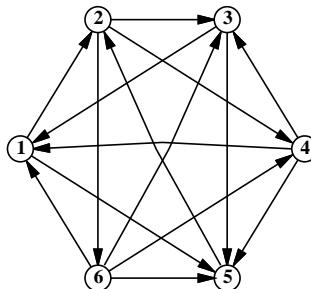
- ★(d) If n is an integer multiple of m , then F_n is an integer multiple of F_m .

2. (a) Prove the following identity by induction:

$$\binom{2n}{n} = \sum_{k=0}^n \binom{n}{k} \binom{n}{n-k}.$$

- (b) Give a non-inductive combinatorial proof of the same identity, by showing that the two sides of the equation count exactly the same thing in two different ways. There is a correct one-sentence proof.

3. A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once. Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

4. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. Extra credit will be given for more exact solutions.

- (a) $A(n) = A(n/2) + n$
- (b) $B(n) = 2B(n/2) + n$
- ★(c) $C(n) = n + \frac{1}{2}(C(n-1) + C(3n/4))$
- (d) $D(n) = \max_{n/3 < k < 2n/3} (D(k) + D(n-k) + n)$
- *(e) $E(n) = 2E(n/2) + n/\lg n$
- *(f) $F(n) = \frac{F(n-1)}{F(n-2)}$, where $F(1) = 1$ and $F(2) = 2$.
- *(g) $G(n) = G(n/2) + G(n/4) + G(n/6) + G(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.]
- *(h) $H(n) = n + \sqrt{n} \cdot H(\sqrt{n})$
- *(i) $I(n) = (n-1)(I(n-1) + I(n-2))$, where $I(0) = I(1) = 1$
- *(j) $J(n) = 8J(n-1) - 15J(n-2) + 1$

5. (a) Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor} = \Theta(n^2)$.

- (b) Prove or disprove: $2^{\lfloor \lg n \rfloor} = \Theta(2^{\lceil \lg n \rceil})$.
- (c) Prove or disprove: $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta(2^{2^{\lceil \lg \lg n \rceil}})$.
- (d) Prove or disprove: If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n)))$.
- (e) Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.
- *(f) Prove that $\log^k n = o(n^{1/k})$ for any positive integer k .

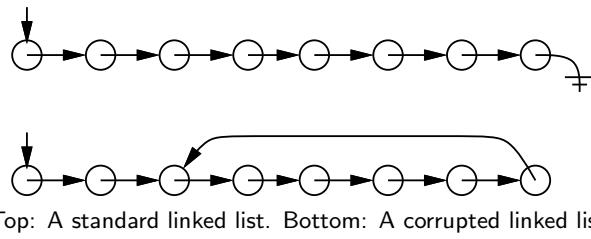
6. Evaluate the following summations; simplify your answers as much as possible. Significant partial credit will be given for answers in the form $\Theta(f(n))$ for some recognizable function $f(n)$.

$$(a) \sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{i}$$

$$*(b) \sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{j}$$

$$(c) \sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{k}$$

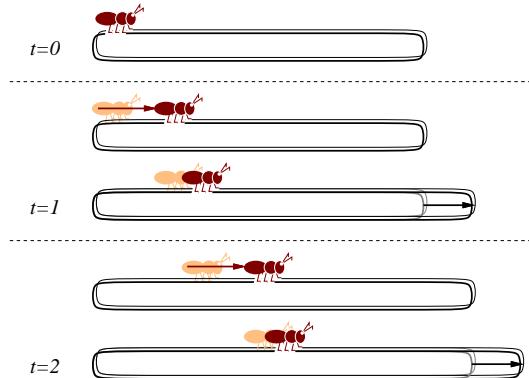
7. Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node's pointer is `NULL`. Unfortunately, your list might have been corrupted by a bug in somebody else's code², so that the last node has a pointer back to some other node in the list instead.



Top: A standard linked list. Bottom: A corrupted linked list.

Describe an algorithm that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

- *8. An ant is walking along a rubber band, starting at the left end. Once every second, the ant walks one inch to the right, and then you make the rubber band one inch longer by pulling on the right end. The rubber band stretches uniformly, so stretching the rubber band also pulls the ant to the right. The initial length of the rubber band is n inches, so after t seconds, the rubber band is $n + t$ inches long.

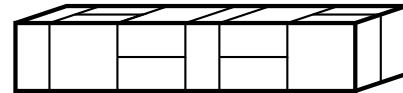


Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

- (a) How far has the ant moved after t seconds, as a function of n and t ? Set up a recurrence and (for full credit) give an *exact* closed-form solution. [Hint: What *fraction* of the rubber band's length has the ant walked?]
- (b) How long does it take the ant to get to the right end of the rubber band? For full credit, give an answer of the form $f(n) + \Theta(1)$ for some explicit function $f(n)$.
9. (a) A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos? Set up a recurrence relation and give an *exact* closed-form solution.

²After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

- (b) A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

10. Professor George O'Jungle has a favorite 26-node binary tree, whose nodes are labeled by letters of the alphabet. The preorder and postorder sequences of nodes are as follows:

preorder: M N H C R S K W T G D X I Y A J P O E Z V B U L Q F
 postorder: C W T K S G R H D N A O E P J Y Z I B Q L F U V X M

Draw Professor O'Jungle's binary tree, and give the inorder sequence of nodes.

11. Alice and Bob each have a fair n -sided die. Alice rolls her die once. Bob then repeatedly throws his die until he rolls a number at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. If you have to appeal to ‘intuition’ or ‘common sense’, your answer is probably wrong!

12. Prove that for any nonnegative parameters a and b , the following algorithms terminate and produce identical output.

SLOWEUCLID(a, b) :

```

if  $b > a$ 
    return SLOWEUCLID( $b, a$ )
else if  $b = 0$ 
    return  $a$ 
else
    return SLOWEUCLID( $b, a - b$ )
  
```

FASTEUCLID(a, b) :

```

if  $b = 0$ 
    return  $a$ 
else
    return FASTEUCLID( $b, a \bmod b$ )
  
```

CS 373: Combinatorial Algorithms, Spring 2001

Homework 1 (due Thursday, February 1, 2001 at 11:59:59 p.m.)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \dots < c[n]$. (In the U.S., for example, $n = 6$ and the values are 1, 5, 10, 25, 50 and 100 cents.) Your beloved and benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.
 - (a) In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and recursively give change for the remainder. El Generalissimo does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does *not* always give the smallest possible of coins.
 - (b) Describe and analyze a dynamic programming algorithm to determine, given a target amount A and a sorted array $c[1..n]$ of coin values, the smallest number of coins needed to make A cents in change. You can assume that $c[1] = 1$, so that it is possible to make change for any amount A .

2. Consider the following sorting algorithm:

```
STUPIDSORT( $A[0..n-1]$ ) :
  if  $n = 2$  and  $A[0] > A[1]$ 
    swap  $A[0] \leftrightarrow A[1]$ 
  else if  $n > 2$ 
     $m \leftarrow \lceil 2n/3 \rceil$ 
    STUPIDSORT( $A[0..m-1]$ )
    STUPIDSORT( $A[n-m..n-1]$ )
    STUPIDSORT( $A[0..m-1]$ )
```

- (a) Prove that STUPIDSORT actually sorts its input.
- (b) Would the algorithm still sort correctly if we replaced the line $m \leftarrow \lceil 2n/3 \rceil$ with $m \leftarrow \lfloor 2n/3 \rfloor$? Justify your answer.
- (c) State a recurrence (including the base case(s)) for the number of comparisons executed by STUPIDSORT.
- (d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.] Does the algorithm deserve its name?
- *(e) Show that the number of *swaps* executed by STUPIDSORT is at most $\binom{n}{2}$.

3. The following randomized algorithm selects the r th smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call RANDOMSELECT($A, 1$); to find the median element, you would call RANDOMSELECT($A, \lfloor n/2 \rfloor$). Recall from lecture that PARTITION splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element.

```
RANDOMSELECT( $A[1..n], r$ ) :
   $p \leftarrow \text{RANDOM}(1, n)$ 
   $k \leftarrow \text{PARTITION}(A[1..n], p)$ 
  if  $r < k$ 
    return RANDOMSELECT( $A[1..k-1], r$ )
  else if  $r > k$ 
    return RANDOMSELECT( $A[k+1..n], r-k$ )
  else
    return  $A[k]$ 
```

- (a) State a recurrence for the expected running time of RANDOMSELECT, as a function of n and r .
- (b) What is the *exact* probability that RANDOMSELECT compares the i th smallest and j th smallest elements in the input array? The correct answer is a simple function of i , j , and r . [Hint: Check your answer by trying a few small examples.]
- (c) Show that for any n and r , the expected running time of RANDOMSELECT is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, find the exact expected number of comparisons, as a function of n and r .
- (d) What is the expected number of times that RANDOMSELECT calls itself recursively?

4. What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basketweaving! The World Champions will be decided by a best-of- $2n - 1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requiring four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1 - p$ that Urbana will win.

Let $P(i, j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0, j) = 1$, $1 \leq j \leq n$, because Champaign needs no more victories to win. $P(i, 0) = 0$, $1 \leq i \leq n$, as Champaign cannot possibly win if Urbana already has. $P(0, 0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q , so

$$P(i, j) = p \cdot P(i - 1, j) + q \cdot P(i, j - 1)$$

for any $i \geq 1$ and $j \geq 1$.

Create and analyze an $O(n^2)$ -time dynamic programming algorithm that takes the parameters n , p and q and returns the probability that Champaign will win the series (that is, calculate $P(n, n)$).

5. The traditional Devonian/Cornish drinking song “The Barley Mow” has the following pseudolyrics¹, where $\text{container}[i]$ is the name of a container that holds 2^i ounces of beer.²

```
BARLEYMOW( $n$ ):
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

    "We'll drink it out of the jolly brown bowl,"
    "Here's a health to the barley-mow!"
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

for  $i \leftarrow 1$  to  $n$ 
    "We'll drink it out of the container[ $i$ ], boys,"
    "Here's a health to the barley-mow!"
    for  $j \leftarrow i$  downto 1
        "The container[j],"
        "And the jolly brown bowl!"
        "Here's a health to the barley-mow!"
        "Here's a health to the barley-mow, my brave boys,"
        "Here's a health to the barley-mow!"
```

- (a) Suppose each container name $\text{container}[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)
- (b) If you want to sing this song for $n > 20$, you’ll have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases³. Suppose $\text{container}[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)
- (c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $\text{container}[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BARLEYMOW(n)? (Give an *exact* answer, not just an asymptotic bound.)

¹Pseudolyrics are to lyrics as pseudocode is to code.

²One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

³“We'll drink it out of the hemisemidemiyottapint, boys!”

6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the i th word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j , then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^j p_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

Practice Problems

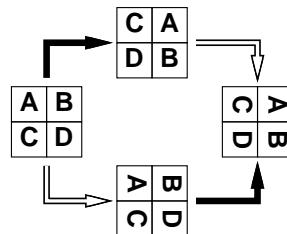
1. Give an $O(n^2)$ algorithm to find the longest increasing subsequence of a sequence of numbers. The elements of the subsequence need not be adjacent in the sequence. For example, the sequence $\langle 1, 5, 3, 2, 4 \rangle$ has longest increasing subsequence $\langle 1, 3, 4 \rangle$.

2. You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)
 - (a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.
 - (b) Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)

3. Give an algorithm that finds the *second* smallest of n elements in at most $n + \lceil \lg n \rceil - 2$ comparisons. [Hint: divide and conquer to find the smallest; where is the second smallest?]

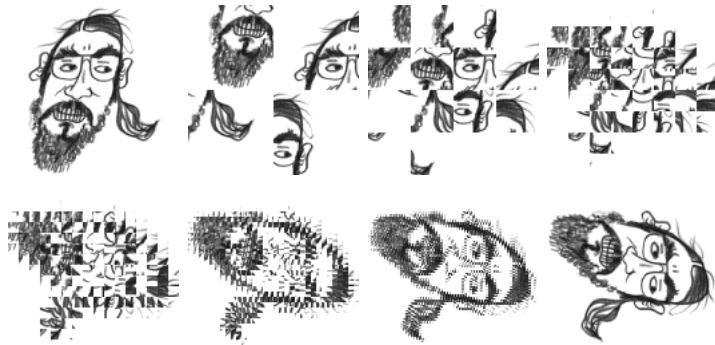
4. Some graphics hardware includes support for an operation called *blit*, or **block transfer**, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixelmap 90° clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
 Black arrows indicate blitting the blocks into place.
 White arrows indicate recursively rotating the blocks.

The following sequence of pictures shows the first algorithm (blit then recurse) in action.



In the following questions, assume n is a power of two.

- (a) Prove that both versions of the algorithm are correct. [Hint: If you exploit all the available symmetries, your proof will only be a half of a page long.]
 - (b) *Exactly* how many blits does the algorithm perform?
 - (c) What is the algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?
 - (d) What if a $k \times k$ blit takes only $O(k)$ time?
-
5. A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how ‘fun’ the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it’s her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the ‘fun’ ratings of the guests.

 6. Suppose you have a subroutine that can find the median of a set of n items (*i.e.*, the $\lfloor n/2 \rfloor$ smallest) in $O(n)$ time. Give an algorithm to find the k th biggest element (for arbitrary k) in $O(n)$ time.

 7. You’re walking along the beach and you stub your toe on something in the sand. You dig around it and find that it is a treasure chest full of gold bricks of different (integral) weight. Your knapsack can only carry up to weight n before it breaks apart. You want to put as much in it as possible without going over, but you *cannot* break the gold bricks up.
 - (a) Suppose that the gold bricks have the weights $1, 2, 4, 8, \dots, 2^k$, $k \geq 1$. Describe and prove correct a greedy algorithm that fills the knapsack as much as possible without going over.
 - (b) Give a set of 3 weight values for which the greedy algorithm does *not* yield an optimal solution and show why.
 - (c) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of gold brick values.

CS 373: Combinatorial Algorithms, Spring 2001

Homework 2 (due Thu. Feb. 15, 2001 at 11:59 PM)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

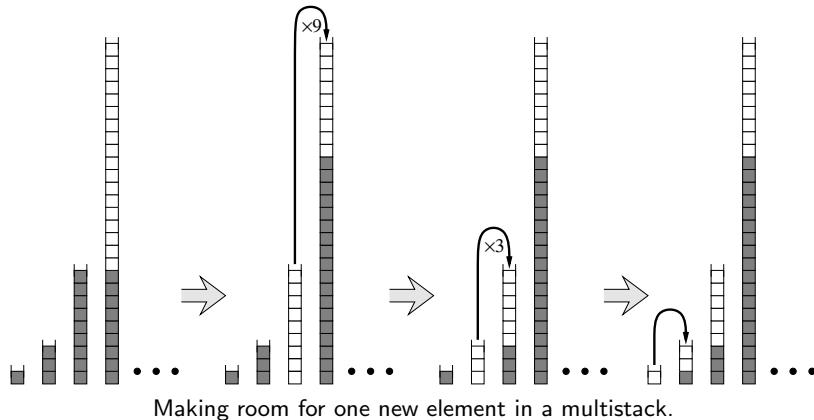
1. Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k . Describe an algorithm to find the k th smallest element in the union of A and B . (For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, our algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicates. For full credit, your algorithm should run in $\Theta(\log n)$ time. [Hint: First try to solve the special case $k = n$.]
2. Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.
 - (a) Show that a rotation in an augmented binary tree can be performed in constant time.
 - (b) Describe an algorithm SCAPEGOATSELECT(k) that selects the k th smallest item in an augmented scapegoat tree in $O(\log n)$ worst-case time.
 - (c) Describe an algorithm SPLAISELECT(k) that selects the k th smallest item in an augmented splay tree in $O(\log n)$ amortized time.

- (d) Describe an algorithm TREAPSELECT(k) that selects the k th smallest item in an augmented treap in $O(\log n)$ *expected* time.
3. (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.
- (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v . A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)
- *(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory.
4. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (it makes it much more difficult).

5. A *multistack* consists of an infinite series of stacks S_0, S_1, S_2, \dots , where the i th stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i , we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2} , and so on. Moving a single element from one stack to the next takes $O(1)$ time.



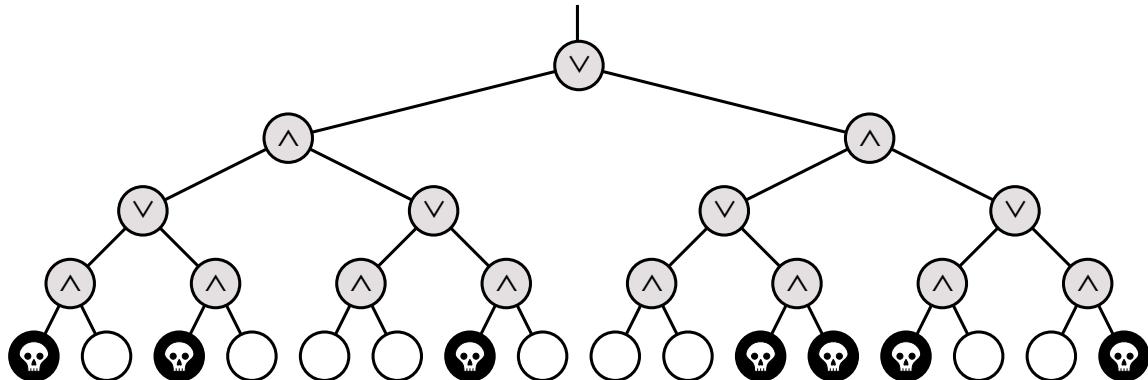
- (a) [1 point] In the worst case, how long does it take to push one more element onto a multistack containing n elements?
- (b) [9 points] Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.

6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.

You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- (a) (2 pts) Describe and analyze a deterministic algorithm to determine whether or not you can win. [Hint: This is easy!]
- (b) (8 pts) Unfortunately, Death won't let you even look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $\Theta(3^n)$ expected time. [Hint: Consider the case $n = 1$.]



Practice Problems

1. (a) Show that it is possible to transform any n -node binary search tree into any other n -node binary search tree using at most $2n - 2$ rotations.

*(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

2. Faster Longest Increasing Subsequence(LIS)

Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. [Hint: In the dynamic programming solution, you don't really have to look back at all previous items. There was a practice problem on HW 1 that asked for an $O(n^2)$ algorithm for this. If you are having difficulty, look at the solution provided in the HW 1 solutions.]

3. Amortization

- (a) Modify the binary double-counter (see class notes Sept 12) 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: Suppose p is the number of significant bits in P , and n is the number of significant bits in N . For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

- *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

- *4. Amortization

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 i th least significant fit indicates whether the sum includes the i th Fibonacci number F_i . For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

5. Detecting overlap

- (a) You are given a list of ranges represented by min and max (*e.g.*, [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$ -time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

- (b) You are given a list of rectangles represented by min and max x - and y -coordinates. Give an $O(n \log n)$ -time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). [Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.]

6. Comparison of Amortized Analysis Methods

A sequence of n operations is performed on a data structure. The i th operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

- (a) Aggregate method
- (b) Accounting method
- *(c) Potential method

CS 373: Combinatorial Algorithms, Spring 2001

Homework 3 (due Thursday, March 8, 2001 at 11:59.99 p.m.)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Hashing:

A hash table of size m is used to store n items with $n \leq m/2$. Open addressing is used for collision resolution.

- Assuming uniform hashing, show that for $i = 1, 2, \dots, n$, the probability that the i^{th} insertion requires strictly more than k probes is at most 2^{-k} .
- Show that for $i = 1, 2, \dots, n$, the probability that the i^{th} insertion requires more than $2 \lg n$ probes is at most $1/n^2$.

Let the random variable X_i denote the number of probes required by the i^{th} insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

- Show that $\Pr\{X > 2 \lg n\} \leq 1/n$.
- Show that the expected length of the longest probe sequence is $E[X] = O(\lg n)$.

2. Reliable Network:

Suppose you are given a graph of a computer network $G = (V, E)$ and a function $r(u, v)$ that gives a reliability value for every edge $(u, v) \in E$ such that $0 \leq r(u, v) \leq 1$. The reliability value gives the probability that the network connection corresponding to that edge will *not* fail. Describe and analyze an algorithm to find the most reliable path from a given source vertex s to a given target vertex t .

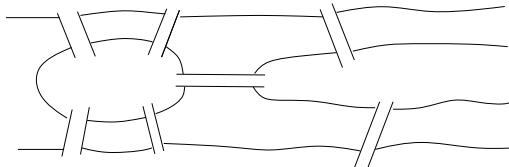
3. Aerophobia:

After graduating you find a job with Aerophobes-R'-Us, the leading traveling agency for aerophobic people. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying so the trip should be as short as possible.

In other words, a person wants to fly from city A to city B in the shortest possible time. S/he turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose an optimal route to minimize the total time in transit. Hint: rather than modify Dijkstra's algorithm, modify the data. The total transit time is from departure to arrival at the destination, so it will include layover time (time waiting for a connecting flight).

4. The Seven Bridges of Königsberg:

During the eighteenth century the city of Königsberg in East Prussia was divided into four sections by the Pregel river. Seven bridges connected these regions, as shown below. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to their starting point.



- (a) Show how the residents of the city could accomplish such a walk or prove no such walk exists.
 - (b) Given any undirected graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge exactly once, or says that it can't be done.
5. Minimum Spanning Tree changes:

Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).

- (a) Give an algorithm to update the MST when an edge is added to G .
- (b) Give an algorithm to update the MST when an edge is deleted from G .
- (c) Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G .

6. Nesting Envelopes

[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.] You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B . Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

Practice Problems

1. Makefiles:

In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called ‘make’ that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files which are listed. Thus a source file must be recompiled if a file on which it depends is changed.

Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design an algorithm to recompile only those necessary. DO NOT worry about the details of parsing a Makefile.

★2. Let the hash function for a table of size m be

$$h(x) = \lfloor Ax \rfloor \bmod m$$

where $A = \hat{\phi} = \frac{\sqrt{5}-1}{2}$. Show that this gives the best possible spread, i.e. if the x are hashed in order, $x + 1$ will be hashed in the largest remaining contiguous interval.

3. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} 1 & (i, j) \in E, \\ 0 & (i, j) \notin E. \end{cases}$$

(a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose). Justify.

(b) Describe what all the entries of the matrix product B^TB represent. Justify.

★(c) Let $C = BB^T - 2A$. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees. (A is the adjacency matrix of G , with zeroes on the diagonal).

4. $o(V^2)$ Adjacency Matrix Algorithms

(a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.

- (b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V - 2$ (the body) connected to the other $V - 3$ vertices (the feet). Some of the feet may be connected to other feet.

Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

- (c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

5. Shortest Cycle:

Given an **undirected** graph $G = (V, E)$, and a weight function $f : E \rightarrow \mathbf{R}$ on the **edges**, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G .

6. Longest Simple Path:

Let graph $G = (V, E)$, $|V| = n$. A *simple path* of G , is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G . Hint: It can be done in $O(n^c 2^n)$ time, for some constant c .

7. Minimum Spanning Tree:

Suppose all edge weights in a graph G are equal. Give an algorithm to compute an MST.

8. Transitive reduction:

Give an algorithm to construct a *transitive reduction* of a directed graph G , i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR} .

9. (a) What is $5^{2^{29}5^0 + 23^41 + 17^32 + 11^23 + 5^14} \pmod{6}$?

- (b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Spring 2001

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 4 (due Thu. March 29, 2001 at 11:59:59 pm)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

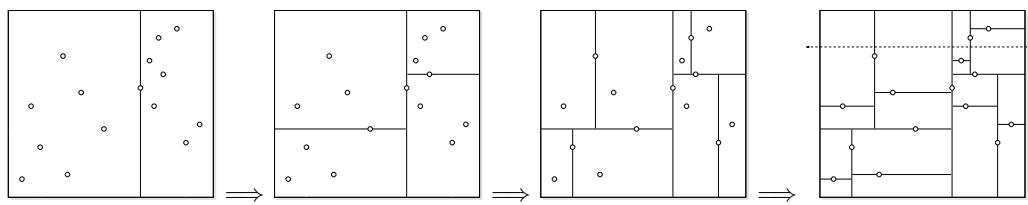
Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

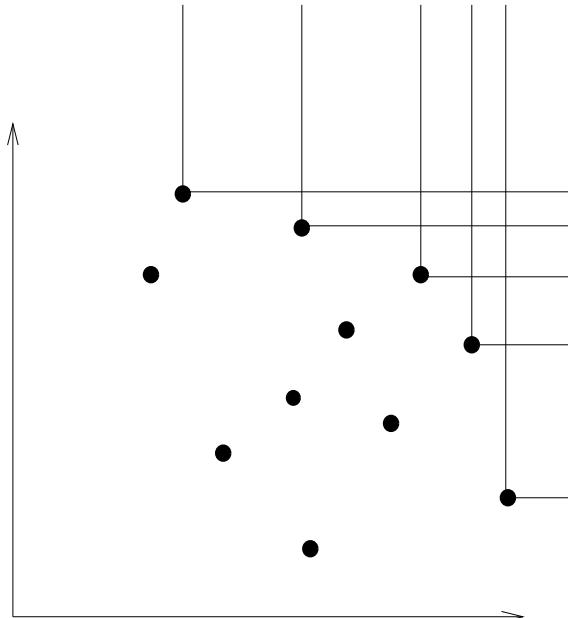
Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

- Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the rectangle as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point inside the box (*not* on the boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



Successive divisions of a kd-tree for 15 points. The dashed line crosses four cells.



An example staircase as in problem 3.

- (a) How many cells are there, as a function of n ? Prove your answer is correct.
- (b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n ? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k .
- (c) Suppose we have n points stored in a kd-tree. Describe an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) in $O(\sqrt{n})$ time.
- *(d) [Optional: 5 pts extra credit] Find an algorithm that counts the number of points that lie inside a rectangle R and show that it takes $O(\sqrt{n})$ time. You may assume that the sides of the rectangle are parallel to the sides of the box.

2. Circle Intersection [*This problem is worth 20 points*]

Describe an algorithm to decide, given n circles in the plane, whether any two of them intersect, in $O(n \log n)$ time. Each circle is specified by three numbers: its radius and the x - and y -coordinates of its center.

We only care about intersections between circle boundaries; concentric circles do not intersect. What general position assumptions does your algorithm require? [Hint: Modify an algorithm for detecting line segment intersections, but describe your modifications very carefully! There are at least two very different solutions.]

3. Staircases

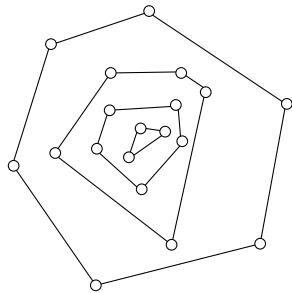
You are given a set of points in the first quadrant. A *left-up* point of this set is defined to be a point that has no points both greater than it in both coordinates. The left-up subset of a set of points then forms a *staircase* (see figure).

- (a) Prove that left-up points do not necessarily lie on the convex hull.
- (b) Give an $O(n \log n)$ algorithm to find the staircase of a set of points.

- (c) Assume that points are chosen uniformly at random within a rectangle. What is the average number of points in a staircase? Justify. Hint: you will be able to give an exact answer rather than just asymptotics. You have seen the same analysis before.
4. Convex Layers

Given a set Q of points in the plane, define the *convex layers* of Q inductively as follows: The first convex layer of Q is just the convex hull of Q . For all $i > 1$, the i th convex layer is the convex hull of Q after the vertices of the first $i - 1$ layers have been removed.

Give an $O(n^2)$ -time algorithm to find all convex layers of a given set of n points.



A set of points with four convex layers.

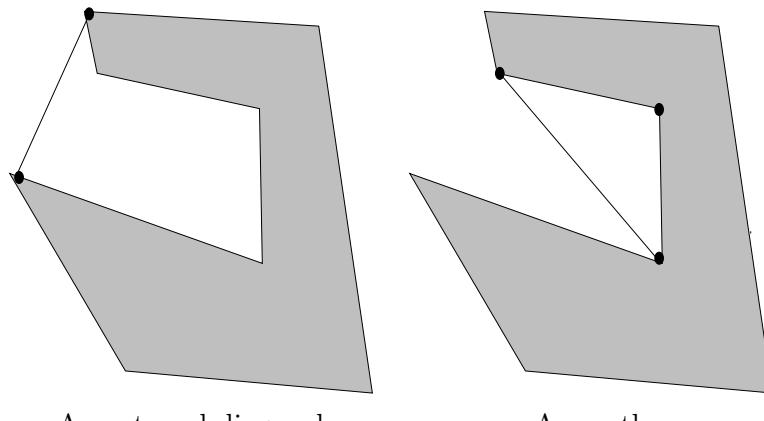
5. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.] Solve the travelling salesman problem for points in convex position (ie, the vertices of a convex polygon). Finding the shortest cycle that visits every point is easy – it's just the convex hull. Finding the shortest path that visits every point is a little harder, because the path can cross through the interior.

- (a) Show that the optimal path cannot be one that crosses itself.
 (b) Describe an $O(n^2)$ time dynamic programming algorithm to solve the problem.

Practice Problems

1. Basic Computation (assume two dimensions and *exact* arithmetic)
 - (a) Intersection: Extend the basic algorithm to determine if two line segments intersect by taking care of *all* degenerate cases.
 - (b) Simplicity: Give an $O(n \log n)$ algorithm to determine whether an n -vertex polygon is simple.
 - (c) Area: Give an algorithm to compute the area of a simple n -polygon (not necessarily convex) in $O(n)$ time.
 - (d) Inside: Give an algorithm to determine whether a point is within a simple n -polygon (not necessarily convex) in $O(n)$ time.

2. External Diagonals and Mouths
 - (a) A pair of polygon vertices defines an *external diagonal* if the line segment between them is completely outside the polygon. Show that every nonconvex polygon has at least one external diagonal.
 - (b) Three consecutive polygon vertices p, q, r form a *mouth* if p and r define an external diagonal. Show that every nonconvex polygon has at least one mouth.



3. On-Line Convex Hull

We are given the set of points one point at a time. After receiving each point, we must compute the convex hull of all those points so far. Give an algorithm to solve this problem in $O(n^2)$ (We could obviously use Graham's scan n times for an $O(n^2 \log n)$ algorithm). Hint: How do you maintain the convex hull?

4. Another On-Line Convex Hull Algorithm

- (a) Given an n -polygon and a point outside the polygon, give an algorithm to find a tangent.
- *(b) Suppose you have found both tangents. Give an algorithm to remove the points from the polygon that are within the angle formed by the tangents (as segments!) and the opposite side of the polygon.

- (c) Use the above to give an algorithm to compute the convex hull on-line in $O(n \log n)$
5. Order of the size of the convex hull
The convex hull on $n \geq 3$ points can have anywhere from 3 to n points. The average case depends on the distribution.
- (a) Prove that if a set of points is chosen randomly within a given rectangle then the average size of the convex hull is $O(\log n)$.
 - ★(b) Prove that if a set of points is chosen randomly within a given circle then the average size of the convex hull is $O(n^{1/3})$.
6. Ghostbusters and Ghosts
A group of n ghostbusters is battling n ghosts. Each ghostbuster can shoot a single energy beam at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits a ghost. The ghostbusters must all fire at the same time and no two energy beams may cross (it would be bad). The positions of the ghosts and ghostbusters is fixed in the plane (assume that no three points are collinear).
- (a) Prove that for any configuration of ghosts and ghostbusters there exists such a non-crossing matching.
 - (b) Show that there exists a line passing through one ghostbuster and one ghost such that the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Give an efficient algorithm to find such a line.
 - (c) Give an efficient divide and conquer algorithm to pair ghostbusters and ghosts so that no two streams cross.

CS 373: Combinatorial Algorithms, Spring 2001

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 5 (due Tue. Apr. 17, 2001 at 11:59 pm)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, $\frac{3}{4}$ -unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Prove that finding the second smallest of n elements takes EXACTLY $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. Prove for both upper and lower bounds. Hint: find the (first) smallest using an elimination tournament.
2. *Fibonacci strings* are defined as follows:

$$F_1 = "b", \quad F_2 = "a", \quad \text{and } F_n = F_{n-1}F_{n-2}, (n > 2)$$

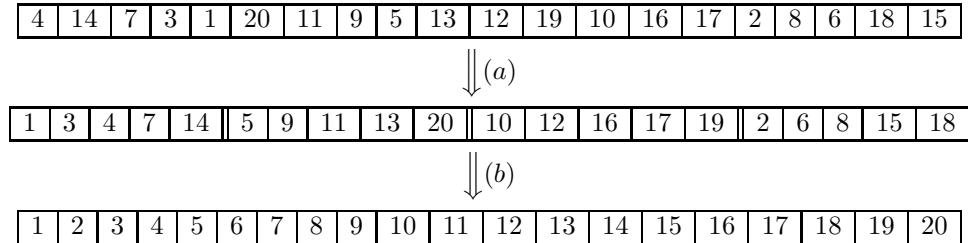
where the recursive rule uses concatenation of strings, so F_3 is “ab”, F_4 is “aba”. Note that the length of F_n is the n th Fibonacci number.

- (a) Prove that in any Fibonacci string there are no two b’s adjacent and no three a’s.
- (b) Give the unoptimized and optimized ‘prefix’ (fail) function for F_7 .
- (c) Prove that, in searching for the Fibonacci string F_k , the unoptimized KMP algorithm can shift $\lceil k/2 \rceil$ times in a row trying to match the last character of the pattern. In other words, prove that there is a chain of failure links $m \rightarrow \text{fail}[m] \rightarrow \text{fail}[\text{fail}[m]] \rightarrow \dots$ of length $\lceil k/2 \rceil$, and find an example text T that would cause KMP to traverse this entire chain at a single text position.

- (d) Prove that the unoptimized KMP algorithm can shift $k - 2$ times in a row at the same text position when searching for F_k . Again, you need to find an example text T that would cause KMP to traverse this entire chain on the same text character.
- (e) How do the failure chains in parts (c) and (d) change if we use the optimized failure function instead?

3. Two-stage sorting

- (a) Suppose we are given an array $A[1..n]$ of distinct integers. Describe an algorithm that splits A into n/k subarrays, each with k elements, such that the elements of each subarray $A[(i-1)k+1..ik]$ are sorted. Your algorithm should run in $O(n \log k)$ time.
- (b) Given an array $A[1..n]$ that is already split into n/k sorted subarrays as in part (a), describe an algorithm that sorts the entire array in $O(n \log(n/k))$ time.
- (c) Prove that your algorithm from part (a) is optimal.
- (d) Prove that your algorithm from part (b) is optimal.

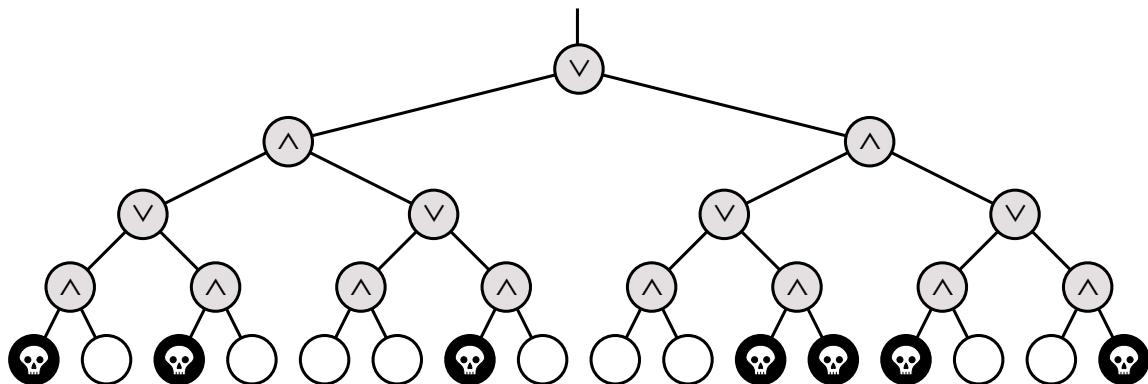


4. Show how to extend the Rabin-Karp fingerprinting method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted horizontally and vertically, but it may not be rotated.)

5. Death knocks on your door once more on a warm spring day. He remembers that you are an algorithms student and that you soundly defeated him last time and are now living out your immortality. Death is in a bit of a quandry. He has been losing a lot and doesn't know why. He wants you to prove a lower bound on your deterministic algorithm so that he can reap more souls. If you have forgotten, the game goes like this: It is a complete binary tree with 4^n leaves, each colored black or white. There is a token at the root of the tree. To play the game, you and Death took turns moving the token from its current node to one of its children. The game ends after $2n$ moves, when the token lands on a leaf. If the final leaf is black, the player dies; if it's white, you will live forever. You move first, so Death gets the last turn.

You decided whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should've challenged Death to a game of Twister instead.

Prove that *any* deterministic algorithm must examine *every* leaf of the tree in the worst case. Since there are 4^n leaves, this implies that any deterministic algorithm must take $\Omega(4^n)$ time in the worst case. Use an adversary argument, or in other words, assume Death cheats.



6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Lower Bounds on Adjacency Matrix Representations of Graphs

- Prove that the time to determine if an undirected graph has a cycle is $\Omega(V^2)$.
- Prove that the time to determine if there is a path between two nodes in an undirected graph is $\Omega(V^2)$.

Practice Problems

- String matching with wild-cards

Suppose you have an alphabet for patterns that includes a ‘gap’ or wild-card character; any length string of any characters can match this additional character. For example if ‘*’ is the wild-card, then the pattern ‘foo*bar*nad’ can be found in ‘foofoowangbarnad’. Modify the computation of the prefix function to correctly match strings using KMP.

2. Prove that there is no comparison sort whose running time is linear for at least $1/2$ of the $n!$ inputs of length n . What about at least $1/n$? What about at least $1/2^n$?
3. Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.
4. Find asymptotic upper and lower bounds to $\lg(n!)$ without Stirling's approximation (Hint: use integration).
5. Given a sequence of n elements of n/k blocks (k elements per block) all elements in a block are less than those to the right in sequence, show that you cannot have the whole sequence sorted in better than $\Omega(n \lg k)$. Note that the entire sequence would be sorted if each of the n/k blocks were individually sorted in place. Also note that combining the lower bounds for each block is not adequate (that only gives an upper bound).
6. Show how to find the occurrences of pattern P in text T by computing the prefix function of the string PT (the concatenation of P and T).

CS 373: Combinatorial Algorithms, Spring 2001

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 6 (due Tue. May 1, 2001 at 11:59.99 p.m.)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Note: You will be held accountable for the appropriate responses for answers (e.g. give models, proofs, analyses, etc). For NP-complete problems you should prove everything rigorously, i.e. for showing that it is in NP, give a description of a certificate and a poly time algorithm to verify it, and for showing NP-hardness, you must show that your reduction is polytime (by similarly proving something about the algorithm that does the transformation) and proving both directions of the ‘if and only if’ (a solution of one is a solution of the other) of the many-one reduction.

Required Problems

1. Complexity
 - (a) Prove that $P \subseteq \text{co-NP}$.
 - (b) Show that if $NP \neq \text{co-NP}$, then *every* NP-complete problem is *not* a member of co-NP.
2. 2-CNF-SAT
Prove that deciding satisfiability when all clauses have at most 2 literals is in P.
3. Graph Problems

(a) SUBGRAPH-ISOMORPHISM

Show that the problem of deciding whether one graph is a subgraph of another is NP-complete.

(b) LONGEST-PATH

Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than k is NP-complete.

4. PARTITION, SUBSET-SUM

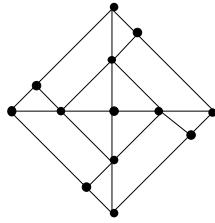
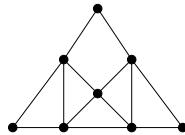
PARTITION is the problem of deciding, given a set of numbers, whether there exists a subset whose sum equals the sum of the complement, i.e. given $S = s_1, s_2 \dots, s_n$, does there exist a subset S' such that $\sum_{s \in S'} s = \sum_{t \in S - S'} t$. SUBSET-SUM is the problem of deciding, given a set of numbers and a target sum, whether there exists a subset whose sum equals the target, i.e. given $S = s_1, s_2 \dots, s_n$ and k , does there exist a subset S' such that $\sum_{s \in S'} s = k$. Give two reduction, one in both directions.

5. BIN-PACKING Consider the bin-packing problem: given a finite set U of n items and the positive integer size $s(u)$ of each item $u \in U$, can U be partitioned into k disjoint sets U_1, \dots, U_k such that the sum of the sizes of the items in each set does not exceed B ? Show that the bin-packing problem is NP-Complete. [Hint: Use the result from the previous problem.]

6. 3SUM

[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Describe an algorithm that solves the following problem as quickly as possible: Given a set of n numbers, does it contain three elements whose sum is zero? For example, your algorithm should answer TRUE for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5 + 7 + (-2) = 0$, and FALSE for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.

**Figure 1.** Gadget for PLANAR-3-COLOR.**Figure 2.** Gadget for DEGREE-4-PLANAR-3-COLOR.

Practice Problems

1. Consider finding the median of 5 numbers by using only comparisons. What is the exact worst case number of comparisons needed to find the median. Justify (exhibit a set that cannot be done in one less comparisons). Do the same for 6 numbers.

2. EXACT-COVER-BY-4-SETS

The EXACT-COVER-BY-3-SETS problem is defined as the following: given a finite set X with $|X| = 3q$ and a collection C of 3-element subsets of X , does C contain an *exact cover* for X , that is, a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C' ?

Given that EXACT-COVER-BY-3-SETS is NP-complete, show that EXACT-COVER-BY-4-SETS is also NP-complete.

3. PLANAR-3-COLOR

Using 3-COLOR, and the ‘gadget’ in figure 3, prove that the problem of deciding whether a planar graph can be 3-colored 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.

4. DEGREE-4-PLANAR-3-COLOR

Using the previous result, and the ‘gadget’ in figure 4, prove that the problem of deciding whether a planar graph with no vertex of degree greater than four can be 3-colored is NP-complete. Hint: show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.

5. Poly time subroutines can lead to exponential algorithms

Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

6. (a) Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian. Give a polynomial time algorithm for finding a **hamiltonian cycle** in an undirected bipartite graph or establishing that it does not exist.
(b) Show that the **hamiltonian-path** problem can be solved in polynomial time on directed acyclic graphs by giving an efficient algorithm for the problem.
(c) Explain why the results in previous questions do not contradict the facts that both HAM-CYCLE and HAM-PATH are NP-complete problems.
7. Consider the following pairs of problems:
 - (a) MIN SPANNING TREE and MAX SPANNING TREE
 - (b) SHORTEST PATH and LONGEST PATH
 - (c) TRAVELING SALESMAN PROBLEM and VACATION TOUR PROBLEM (the longest tour is sought).
 - (d) MIN CUT and MAX CUT (between s and t)
 - (e) EDGE COVER and VERTEX COVER
 - (f) TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

(all of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph).

Which of these pairs are polytime equivalent and which are not? Why?
- ★8. GRAPH-ISOMORPHISM
Consider the problem of deciding whether one graph is isomorphic to another.
 - (a) Give a brute force algorithm to decide this.
 - (b) Give a dynamic programming algorithm to decide this.
 - (c) Give an efficient probabilistic algorithm to decide this.
 - (d) Either prove that this problem is NP-complete, give a poly time algorithm for it, or prove that neither case occurs.
9. Prove that PRIMALITY (Given n , is n prime?) is in $\text{NP} \cap \text{co-NP}$. Hint: co-NP is easy (what's a certificate for showing that a number is composite?). For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that knowing this tree of primitive roots can be checked to be correct and used to show that n is prime, and that this check takes poly time.
10. How much wood would a woodchuck chuck if a woodchuck could chuck wood?

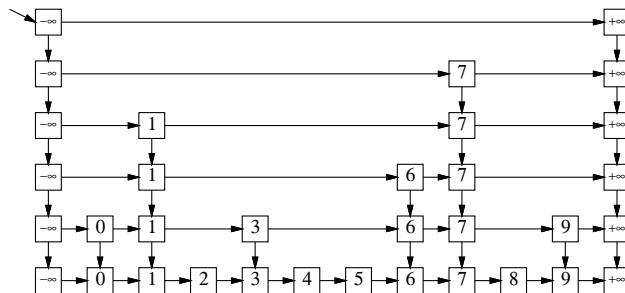
Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

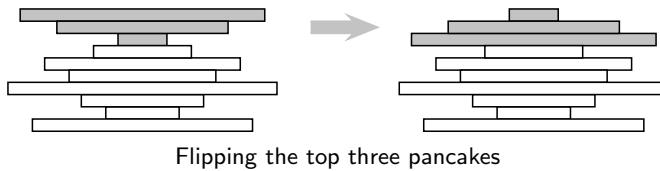
(a) $\Theta(1)$ (b) $\Theta(\log n)$ (c) $\Theta(n)$ (d) $\Theta(n \log n)$ (e) $\Theta(n^2)$

For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point, but each incorrect answer costs you $\frac{1}{2}$ point. You cannot score below zero.

- (a) What is $\sum_{i=1}^n H_i$?
 - (b) What is $\sum_{i=1}^{\lg n} 2^i$?
 - (c) How many digits do you need to write $n!$ in decimal?
 - (d) What is the solution of the recurrence $T(n) = 16T(n/4) + n$?
 - (e) What is the solution of the recurrence $T(n) = T(n - 2) + \lg n$?
 - (f) What is the solution of the recurrence $T(n) = 4T\left(\lceil \frac{n+51}{4} \rceil - \sqrt{n}\right) + 17n - 2^{8 \log^*(n^2)} + 6$?
 - (g) What is the worst-case running time of randomized quicksort?
 - (h) The expected time for inserting one item into a treap is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty treap?
 - (i) The amortized time for inserting one item into an n -node splay tree is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty splay tree?
 - (j) In the worst case, how long does it take to solve the traveling salesman problem for 10,000,000,000,000,000 cities?
2. What is the *exact* expected number of nodes in a skip list storing n keys, *not* counting the sentinel nodes at the beginning and end of each level? Justify your answer. A correct $\Theta()$ bound (with justification) is worth 5 points.



3. Suppose we have a stack of n pancakes of all different sizes. We want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation we can perform is a *flip* — insert a spatula under the top k pancakes, for some k between 1 and n , turn them all over, and put them back on top of the stack.



Flipping the top three pancakes

- (a) (3 pts) Describe an algorithm to sort an arbitrary stack of n pancakes using flips.
 - (b) (3 pts) Prove that your algorithm is correct.
 - (c) (2 pts) Exactly how many flips does your sorting algorithm perform in the worst case? A correct $\Theta()$ bound is worth one point.
 - (d) (2 pts) Suppose one side of each pancake is burned. Exactly how many flips do you need to sort the pancakes, so that the burned side of every pancake is on the bottom? A correct $\Theta()$ bound is worth one point.
4. Suppose we want to maintain a set of values in a data structure subject to the following operations:

- $\text{INSERT}(x)$: Add x to the set (if it isn't already there).
- $\text{DELETERANGE}(a, b)$: Delete every element x in the range $a \leq x \leq b$. For example, if the set was $\{1, 5, 3, 4, 8\}$, then $\text{DELETERANGE}(4, 6)$ would change the set to $\{1, 3, 8\}$.

Describe and analyze a data structure that supports these operations, such that the amortized cost of either operation is $O(\log n)$. [Hint: Use a data structure you saw in class. If you use the same INSERT algorithm, just say so—you don't need to describe it again in your answer.]

5. [1-unit grad students must answer this question.]

A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, ‘bananaananas’ is a shuffle of ‘banana’ and ‘ananas’ in several different ways.

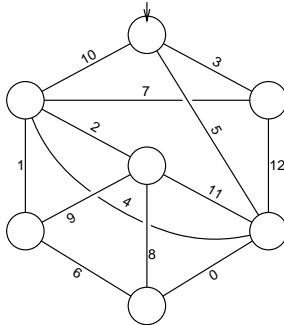
banana ananas ban ana ana nas b an an a na na s

The strings ‘prodgyrnammamiiincg’ and ‘dyprongarmammicing’ are both shuffles of ‘dynamic’ and ‘programming’:

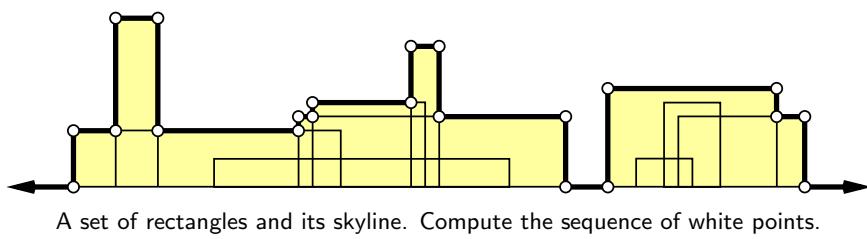
pro^dg y r nam ammi i n c g dy proⁿ g a r m amm ic ing

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B . For full credit, your algorithm should run in $\Theta(mn)$ time.

1. Using any method you like, compute the following subgraphs for the weighted graph below. Each subproblem is worth 3 points. Each incorrect edge costs you 1 point, but you cannot get a negative score for any subproblem.
- a depth-first search tree, starting at the top vertex;
 - a breadth-first search tree, starting at the top vertex;
 - a shortest path tree, starting at the top vertex;
 - the **maximum** spanning tree.



2. (a) [4 pts] Prove that a connected acyclic undirected graph with V vertices has exactly $V - 1$ edges. ("It's a tree!" is not a proof.)
- (b) [4 pts] Describe and analyze an algorithm that determines whether a given undirected graph is a tree, where the graph is represented by an adjacency list.
- (c) [2 pts] What is the running time of your algorithm from part (b) if the graph is represented by an adjacency matrix?
3. Suppose we want to sketch the Manhattan skyline (minus the interesting bits like the Empire State and Chrysler buildings). You are given a set of n rectangles, each rectangle represented by its left and right x -coordinates and its height. The bottom of each rectangle is on the x -axis. Describe and analyze an efficient algorithm to compute the vertices of the skyline.



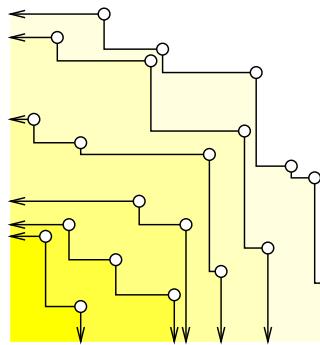
4. Suppose we model a computer network as a weighted undirected graph, where each vertex represents a computer and each edge represents a *direct* network connection between two computers. The weight of each edge represents the *bandwidth* of that connection—the number of bytes that can flow from one computer to the other in one second.¹ We want to implement a point-to-point network protocol that uses a single dedicated path to communicate between any pair of computers. Naturally, when two computers need to communicate, we should use the path with the highest bandwidth. The bandwidth of a *path* is the *minimum* bandwidth of its edges.

Describe an algorithm to compute the maximum bandwidth path between *every* pair of computers in the network. Assume that the graph is represented as an adjacency list.

5. [1-unit grad students must answer this question.]

Let P be a set of points in the plane. Recall that the *staircase* of P contains all the points in P that have no other point in P both above and to the right. We can define the *staircase layers* of P recursively as follows. The first staircase layer is just the staircase; for all $i > 1$, the i th staircase layer is the staircase of P after the first $i - 1$ staircase layers have been deleted.

Describe and analyze an algorithm to compute the staircase layers of P in $O(n^2)$ time.² Your algorithm should label each point with an integer describing which staircase layer it belongs to. You can assume that no two points have the same x - or y -coordinates.



A set of points and its six staircase layers.

¹Notice the bandwidth is symmetric; there are no cable modems or wireless phones. Don't worry about systems-level stuff like network load and latency. After all, this is a theory class!

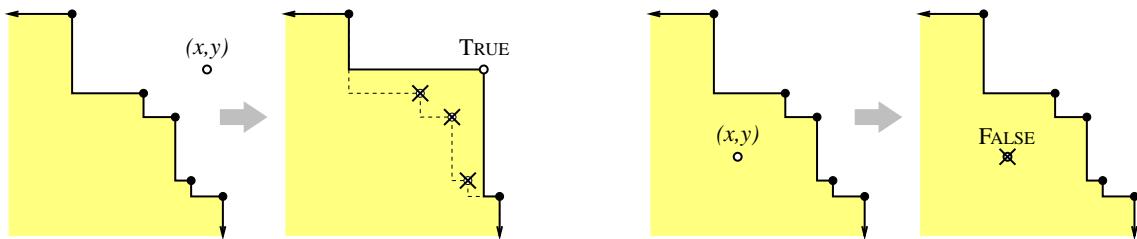
²This is *not* the fastest possible running time for this problem.

You must turn in this question sheet with your answers.**1. Déjà vu**

Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, and $17 = F_7 + F_4 + F_2$. You must give a complete, self-contained proof, not just a reference to the posted homework solutions.

2. L'esprit d'escalier

Recall that the *staircase* of a set of points consists of the points with no other point both above and to the right. Describe a method to maintain the staircase as new points are added to the set. Specifically, describe and analyze a data structure that stores the staircase of a set of points, and an algorithm $\text{INSERT}(x, y)$ that adds the point (x, y) to the set and returns TRUE or FALSE to indicate whether the staircase has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.

**3. Engage le jeu que je le gagne**

A palindrome is a text string that is exactly the same as its reversal, such as DEED, RACECAR, or SAIPPUAKAUPPIAS.¹

- Describe and analyze an algorithm to find the longest *prefix* of a given string that is also a palindrome. For example, the longest palindrome prefix of ILLINOISURBANACHAMPAIGN is ILLI, and the longest palindrome prefix of HYAKUGOJYUUICHI² is the single letter S. For full credit, your algorithm should run in $O(n)$ time.
- Describe and analyze an algorithm to find the length of the longest *subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of ILLINOISURBANACHAMPAIGN is NIAACAAIN (or NIAAHAIN), and the longest palindrome subsequence of HYAKUGOJYUUICHI is HUUUH³ (or HUGUH or HUYUH or...). You do not need to compute the actual subsequence; just its length. For full credit, your algorithm should run in $O(n^2)$ time.

¹Finnish for ‘soap dealer’.

²Japanese for ‘one hundred fifty-one’.

³English for ‘What the heck are you talking about?’

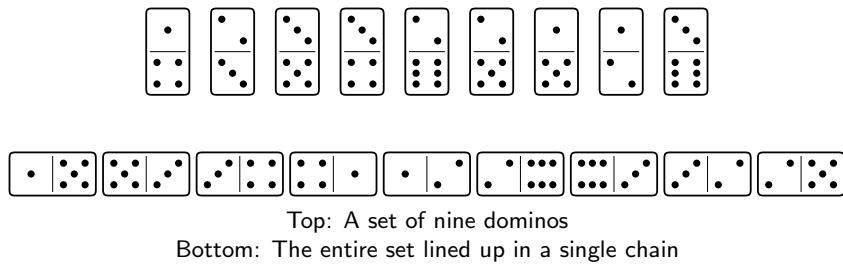
4. Toute votre base sont appartientent à nous

Prove that *exactly* $2n - 1$ comparisons are required in the worst case to merge two sorted arrays, each with n distinct elements. Describe and analyze an algorithm to prove the upper bound, and use an adversary argument to prove the lower bound. *You must give a complete, self-contained solution, not just a reference to the posted homework solutions.*⁴

5. Plus ça change, plus ça même chose

A domino is a 2×1 rectangle divided into two squares, with a certain number of pips (dots) in each square. In most domino games, the players lay down dominos at either end of a single chain. Adjacent dominos in the chain must have matching numbers. (See the figure below.)

Describe and analyze an efficient algorithm, or prove that it is NP-hard, to determine whether a given set of n dominos can be lined up in a single chain. For example, for the set of dominos shown below, the correct output is TRUE.



6. Ceci n'est pas une pipe

Consider the following pair of problems:

- **BOXDEPTH:** Given a set of n axis-aligned rectangles in the plane and an integer k , decide whether any point in the plane is covered by k or more rectangles.
 - **MAXCLIQUE:** Given a graph with n vertices and an integer k , decide whether the graph contains a clique with k or more vertices.
- Describe and analyze a reduction of one of these problems to the other.
 - MAXCLIQUE is NP-hard. What does your answer to part (a) imply about the complexity of BOXDEPTH?

7. C'est magique! [1-unit graduate students must answer this question.]

The recursion fairy's cousin, the reduction genie, shows up one day with a magical gift for you—a box that determines in constant time the size of the largest clique in any given graph. (Recall that a clique is a subgraph where every pair of vertices is joined by an edge.) The magic box does not tell you *where* the largest clique is, only its size. Describe and analyze an algorithm to actually find the largest clique in a given graph **in polynomial time**, using this magic box.

⁴The posted solution for this Homework 5 practice problem was incorrect. So don't use it!

CS 373: Combinatorial Algorithms, Fall 2002

Homework 0, due September 5, 2002 at the beginning of class

Name:		
Net ID:	Alias:	U G

Neatly print your name (first name first, with no comma), your network ID, and an alias of your choice into the boxes above. Circle U if you are an undergraduate, and G if you are a graduate student. **Do not sign your name. Do not write your Social Security number.** Staple this sheet of paper to the top of your homework.

Grades will be listed on the course web site by alias give us, so your alias should not resemble your name or your Net ID. If you don't give yourself an alias, we'll give you one that you won't like.

Before you do anything else, please read the Homework Instructions and FAQ on the CS 373 course web page (<http://www-courses.cs.uiuc.edu/~cs373/hwx/faq.html>) and then check the box below. There are 300 students in CS 373 this semester; we are quite serious about giving zeros to homeworks that don't follow the instructions.

I have read the CS 373 Homework Instructions and FAQ.

Every CS 373 homework has the same basic structure. There are six required problems, some with several subproblems. Each problem is worth 10 points. Only graduate students are required to answer problem 6; undergraduates can turn in a solution for extra credit. There are several practice problems at the end. Stars indicate problems we think are hard.

This homework tests your familiarity with the prerequisite material from CS 173, CS 225, and CS 273, primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Rosen (the 173/273 textbook), CLRS (especially Chapters 1–7, 10, 12, and A–C), and the lecture notes on recurrences should be sufficient review, but you may want to consult other texts as well.

Required Problems

1. Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. Please *don't* turn in proofs, but you should do them anyway to make sure you're right (and for practice).

$$\begin{array}{ccccc}
 1 & n & n^2 & \lg n & n \lg n \\
 n^{\lg n} & (\lg n)^n & (\lg n)^{\lg n} & n^{\lg \lg n} & n^{1/\lg n} \\
 \log_{1000} n & \lg^{1000} n & \lg^{(1000)} n & \lg(n^{1000}) & \left(1 + \frac{1}{1000}\right)^n
 \end{array}$$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

2. Solve these recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Please *don't* turn in proofs, but you should do them anyway just for practice. Assume reasonable but nontrivial base cases, and state them if they affect your solution. Extra credit will be given for more exact solutions. [Hint: Most of these are very easy.]

$$\begin{array}{ll}
 A(n) = 2A(n/2) + n & F(n) = 9F(\lfloor n/3 \rfloor + 9) + n^2 \\
 B(n) = 3B(n/2) + n & G(n) = 3G(n-1)/5G(n-2) \\
 C(n) = 2C(n/3) + n & H(n) = 2H(\sqrt{n}) + 1 \\
 D(n) = 2D(n-1) + 1 & I(n) = \min_{1 \leq k \leq n/2} (I(k) + I(n-k) + k) \\
 E(n) = \max_{1 \leq k \leq n/2} (E(k) + E(n-k) + n) & *J(n) = \max_{1 \leq k \leq n/2} (J(k) + J(n-k) + k)
 \end{array}$$

3. Recall that a binary tree is *full* if every node has either two children (an internal node) or no children (a leaf). Give at least *four different* proofs of the following fact:

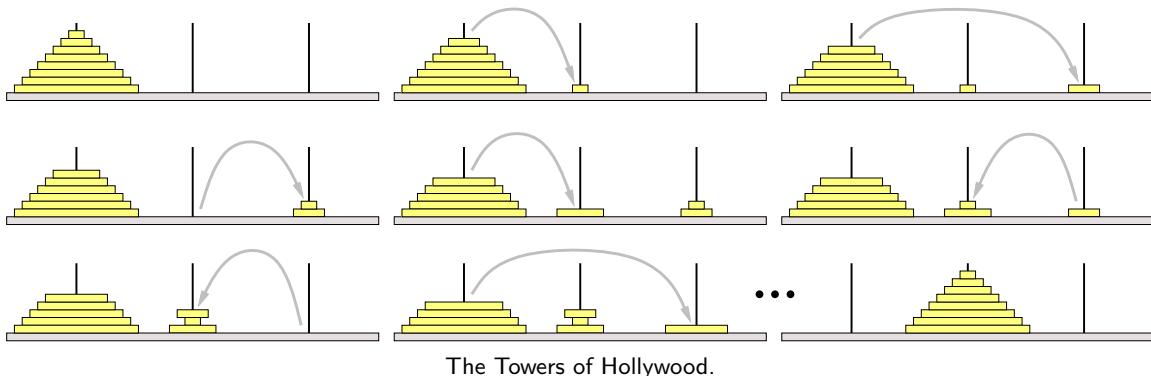
In any full binary tree, the number of leaves is exactly one more than the number of internal nodes.

For full credit, each proof must be self-contained, the proof must be substantially different from each other, and at least one proof must *not* use induction. For each n , your n th correct proof is worth n points, so you need four proofs to get full credit. Each correct proof beyond the fourth earns you extra credit. [Hint: I know of at least six different proofs.]

4. Most of you are probably familiar with the story behind the Tower of Hanoi puzzle:¹

At the great temple of Benares, there is a brass plate on which three vertical diamond shafts are fixed. On the shafts are mounted n golden disks of decreasing size.² At the time of creation, the god Brahma placed all of the disks on one pin, in order of size with the largest at the bottom. The Hindu priests unceasingly transfer the disks from peg to peg, one at a time, never placing a larger disk on a smaller one. When all of the disks have been transferred to the last pin, the universe will end.

Recently the temple at Benares was relocated to southern California, where the monks are considerably more laid back about their job. At the “Towers of Hollywood”, the golden disks were replaced with painted plywood, and the diamond shafts were replaced with Plexiglas. More importantly, the restriction on the order of the disks was relaxed. While the disks are being moved, the *bottom* disk on any pin must be the *largest* disk on that pin, but disks further up in the stack can be in any order. However, after all the disks have been moved, they must be in sorted order again.



Describe an algorithm³ that moves a stack of n disks from one pin to the another using the smallest possible number of moves. For full credit, your algorithm should be non-recursive, but a recursive algorithm is worth significant partial credit. *Exactly* how many moves does your algorithm perform? [Hint: The Hollywood monks can bring about the end of the universe quite a bit faster than the original monks at Benares could.]

The problem of computing the minimum number of moves was posed in the most recent issue of the *American Mathematical Monthly* (August/September 2002). No solution has been published yet.

¹The puzzle and the accompanying story were both invented by the French mathematician Eduoard Lucas in 1883. See <http://www.cs.wm.edu/~pkstoc/toh.html>

²In the original legend, $n = 64$. In the 1883 wooden puzzle, $n = 8$.

³Since you've read the Homework Instructions, you know exactly what this phrase means.

5. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

Rosencrantz: H H T

Guildenstern: H T H H

Rosencrantz: T

Guildenstern: (no flips)

Rosencrantz: H H H T

Guildenstern: T H H T H H T H T H H H

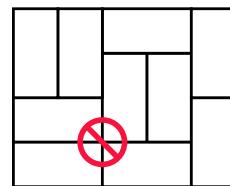
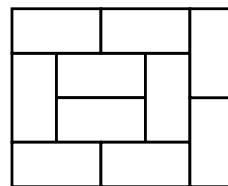
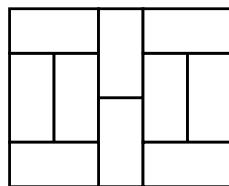
- (a) What is the expected number of flips in one of Rosencrantz's turns?
- (b) Suppose Rosencrantz flips k heads in a row on his turn. What is the expected number of flips in Guildenstern's next turn?
- (c) What is the expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

Prove your answers are correct. If you have to appeal to “intuition” or “common sense”, your answer is almost certainly wrong! You must give exact answers for full credit, but asymptotic bounds are worth significant partial credit.

6. [This problem is required only for graduate students (including I2CS students); undergrads can submit a solution for extra credit.]

Tatami are rectangular mats used to tile floors in traditional Japanese houses. Exact dimensions of tatami mats vary from one region of Japan to the next, but they are always twice as long in one dimension than in the other. (In Tokyo, the standard size is 180cm×90cm.)

- (a) How many different ways are there to tile a $2 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution. [Hint: The answer involves a familiar recursive sequence.]
- (b) According to tradition, tatami mats are always arranged so that four corners never meet. Thus, the first two arrangements below are traditional, but not the third.



Two traditional tatami arrangements and one non-traditional arrangement.

How many different *traditional* ways are there to tile a $3 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution.

- *(c) [5 points extra credit] How many different *traditional* ways are there to tile an $n \times n$ square with 1×2 tatami mats? Prove your answer is correct.

Practice Problems

These problems are only for your benefit; other problems can be found in previous semesters' homeworks on the course web site. You are *strongly* encouraged to do some of these problems as additional practice. Think of them as potential exam questions (hint, hint). Feel free to ask about any of these questions on the course newsgroup, during office hours, or during review sessions.

1. Removing any edge from a binary tree with n nodes partitions it into two smaller binary trees. If both trees have at least $\lceil(n - 1)/3\rceil$ nodes, we say that the partition is *balanced*.
 - (a) Prove that every binary tree with more than one vertex has a balanced partition. [Hint: I know of at least two different proofs.]
 - (b) If each smaller tree has more than $\lfloor n/3 \rfloor$ nodes, we say that the partition is *strictly balanced*. Show that for every n , there is an n -node binary tree with *no* strictly balanced partition.
2. Describe an algorithm COUNTTOTENTOTHE(n) that prints the integers from 1 to 10^n .

Assume you have a subroutine PRINTDIGIT(d) that prints any integer d between 0 and 9, and another subroutine PRINTSPACE that prints a space character. Both subroutines run in $O(1)$ time. You may want to write (and analyze) a separate subroutine PRINTINTEGER to print an arbitrary integer.

Since integer variables cannot store arbitrarily large values in most programming languages, your algorithm must not store any value larger than $\max\{10, n\}$ in any single integer variable. Thus, the following algorithm is **not correct**:

```
BOGUSCOUNTTOTENTOTHE( $n$ ):
  for  $i \leftarrow 1$  to POWER( $10, n$ )
    PRINTINTEGER( $i$ )
```

(So what exactly *can* you pass to PRINTINTEGER?)

What is the running time of your algorithm (as a function of n)? How many digits and spaces does it print? How much space does it use?

3. I'm sure you remember the following simple rules for taking derivatives:

- Simple cases: $\frac{d}{dx}\alpha = 0$ for any constant α , and $\frac{d}{dx}x = 1$
- Linearity: $\frac{d}{dx}(f(x) + g(x)) = f'(x) + g'(x)$
- The product rule: $\frac{d}{dx}(f(x) \cdot g(x)) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$
- The chain rule: $\frac{d}{dx}(f(g(x))) = f'(g(x)) \cdot g'(x)$

Using *only* these rules and induction, prove that $\frac{d}{dx}x^c = cx^{c-1}$ for any integer $c \neq -1$. Do not use limits, integrals, or any other concepts from calculus, except for the simple identities listed above. [Hint: Don't forget about negative values of c !]

4. This problem asks you to calculate the total resistance between two points in a series-parallel resistor network. Don't worry if you haven't taken a circuits class; everything you need to know can be summed up in two sentences and a picture.

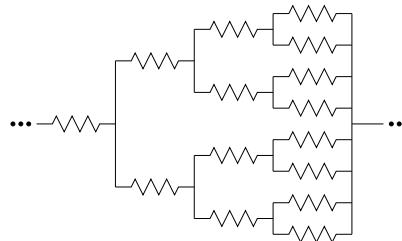
- The total resistance of two resistors in *series* is the sum of their individual resistances.
- The total resistance of two resistors in *parallel* is the reciprocal of the sum of the reciprocals of their individual resistances.

$$\dots - \begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} \begin{smallmatrix} x \\ y \end{smallmatrix} - \begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} - \dots = \dots - \begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} \begin{smallmatrix} x+y \\ \end{smallmatrix} - \dots \quad \dots - \boxed{\begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} \begin{smallmatrix} x \\ y \end{smallmatrix}} - \dots = \dots - \begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} \frac{I}{I/x + I/y} - \dots$$

Equivalence laws for series-parallel resistor networks.

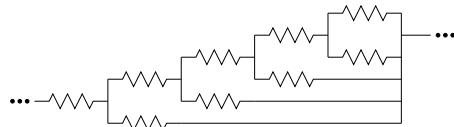
What is the *exact* total resistance⁴ of the following resistor networks as a function of n ? Prove your answers are correct. [Hint: Use induction. Duh.]

- (a) A complete binary tree with depth n , with a 1Ω resistor at every node, and a common wire joining all the leaves. Resistance is measured between the root and the leaves.



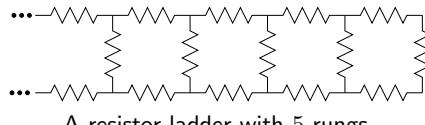
A balanced binary resistor tree with depth 3.

- (b) A totally unbalanced full binary tree with depth n (every internal node has two children, one of which is a leaf) with a 1Ω resistor at every node, and a common wire joining all the leaves. Resistance is measured between the root and the leaves.



A totally unbalanced binary resistor tree with depth 4.

- *(c) A ladder with n rungs, with a 1Ω resistor on every edge. Resistance is measured between the bottom of the legs.



A resistor ladder with 5 rungs.

⁴The ISO standard unit of resistance is the Ohm, written with the symbol Ω . Don't confuse this with the asymptotic notation $\Omega(f(n))$!

CS 373: Combinatorial Algorithms, Fall 2002

Homework 1, due September 17, 2002 at 23:59:59

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Undergrads Grads	

This homework is to be submitted in groups of up to three people. Graduate and undergraduate students are *not* allowed to work in the same group. Please indicate above whether you are undergraduate or graduate students. Only *one* submission per group will be accepted.

Required Problems

1. The traditional Devonian/Cornish drinking song “The Barley Mow” has the following pseudolyrics, where $\text{container}[i]$ is the name of a container¹ that holds 2^i ounces of beer.

<p><u>BARLEYMOW(n):</u></p> <p><i>“Here’s a health to the barley-mow, my brave boys,”</i> <i>“Here’s a health to the barley-mow!”</i></p> <p><i>“We’ll drink it out of the jolly brown bowl,”</i> <i>“Here’s a health to the barley-mow!”</i></p> <p><i>“Here’s a health to the barley-mow, my brave boys,”</i> <i>“Here’s a health to the barley-mow!”</i></p> <p>for $i \leftarrow 1$ to n</p> <p style="padding-left: 2em;"><i>“We’ll drink it out of the container[i], boys,”</i> <i>“Here’s a health to the barley-mow!”</i></p> <p>for $j \leftarrow i$ downto 1</p> <p style="padding-left: 2em;"><i>“The container[j],”</i> <i>“And the jolly brown bowl!”</i></p> <p><i>“Here’s a health to the barley-mow, my brave boys,”</i> <i>“Here’s a health to the barley-mow!”</i></p>

- (a) Suppose each container name $\text{container}[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)

¹One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 anklers, and the last three units are just silly.

- (b) Suppose $\text{container}[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)
- (c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $\text{container}[i]$. Assuming for purposes of this problem that you are over 21, *exactly* how many ounces of beer would you drink if you sang BARLEYMOW(n)? (Give an *exact* answer, not just an asymptotic bound.)
2. Suppose you have a set S of n numbers. Given two elements you *cannot* determine which is larger. However, you are given an oracle that will tell you the median of a set of three elements.
- Give a linear time algorithm to find the pair of the largest and smallest numbers in S .
 - Give an algorithm to sort S in $O(n \lg n)$ time.
3. Given a black and white pixel image $A[1 \dots m][1 \dots n]$, our task is to represent A with a search tree T . Given a query (x, y) , a simple search on T should return the color of pixel $A[x][y]$. The algorithm to construct T will be as follows.

```

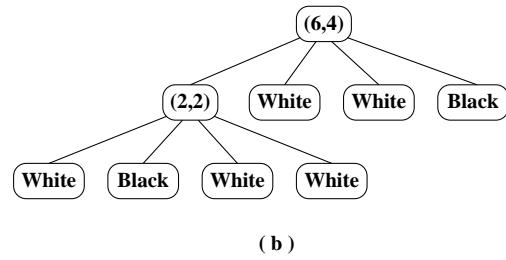
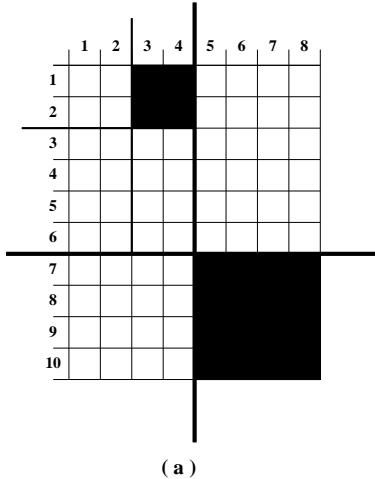
CONSTRUCTSEARCHTREE( $A[1 \dots m][1 \dots n]$ ):
  //Base Case
  if  $A$  contains only one color
    return a leaf node labeled with that color

  //Recurse on Subtrees
   $(i, j) \leftarrow \text{CHOOSECUT}(A[1 \dots m][1 \dots n])$ 
   $T_1 \leftarrow \text{CONSTRUCTSEARCHTREE}(A[1 \dots i][1 \dots j])$ 
   $T_2 \leftarrow \text{CONSTRUCTSEARCHTREE}(A[1 \dots i][j + 1 \dots n])$ 
   $T_3 \leftarrow \text{CONSTRUCTSEARCHTREE}(A[i + 1 \dots m][1 \dots j])$ 
   $T_4 \leftarrow \text{CONSTRUCTSEARCHTREE}(A[i + 1 \dots m][j + 1 \dots n])$ 

  //Construct the Root
   $T.\text{cut} \leftarrow (i, j)$ 
   $T.\text{children} \leftarrow T_1, T_2, T_3, T_4$ 
  return  $T$ 

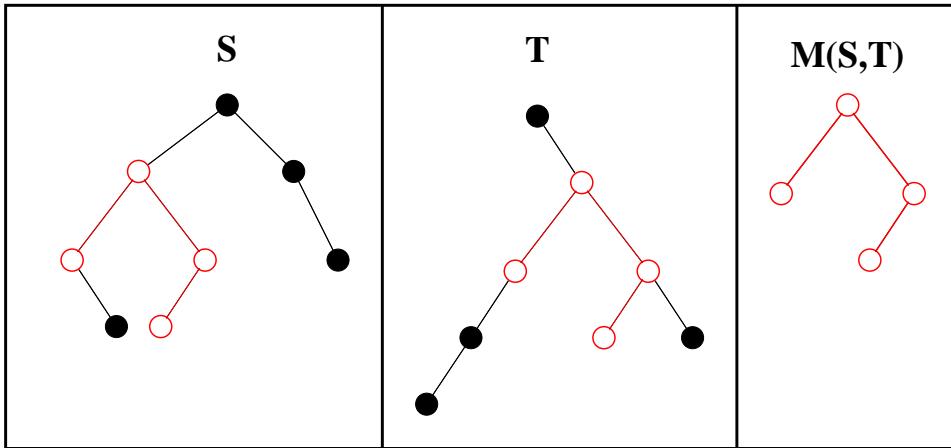
```

That is, this algorithm divides a multicolor image into quadrants and recursively constructs the search tree for each quadrant. Upon a query (x, y) of T (assuming $1 \leq x \leq m$ and $1 \leq y \leq n$), the appropriate subtree is searched. When the correct leaf node is reached, the pixel color is returned. Here's a toy example.

(a) An image A and the chosen cuts. (b) The corresponding search tree.

Your job in this problem is to give an algorithm for CHOOSECUT. The sequence of chosen cuts must result in an optimal search tree T . That is, the expected search depth of a uniformly chosen pixel must be minimized. You may use any external data structures (*i.e.* a global table) that you find necessary. You may also preprocess in order to initialize these structures before the initial call to CONSTRUCTSEARCHTREE($A[1 \dots m][1 \dots n]$).

4. Let A be a set of n positive integers, all of which are no greater than some constant $M > 0$. Give an $O(n^2M)$ time algorithm to determine whether or not it is possible to split A into two subsets such that the sum of the numbers in each subset are equal.
5. Let S and T be two binary trees. A *matching* of S and T is a tree M which is isomorphic to some subtree in each of S and T . Here's an illustration.

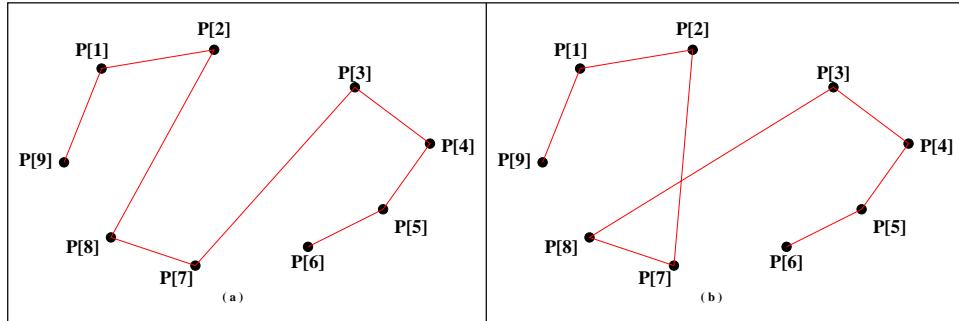
A matching $M(S, T)$ of binary trees S and T .

A *maximal* matching is a matching which contains at least as many vertices as any other matching. Give an algorithm to compute a maximal matching given the roots of two binary trees. Your algorithm should return the size of the match as well as the two roots of the matched subtrees of S and T .

6. [This problem is required only for graduate students (including I2CS students); undergrads can submit a solution for extra credit.]

Let $P[1, \dots, n]$ be a set of n convex points in the plane. Intuitively, if a rubber band were stretched to surround P then each point would touch the rubber band. Furthermore, suppose that the points are labeled such that $P[1], \dots, P[n]$ is a simple path along the convex hull (*i.e.* $P[i]$ is adjacent to $P[i + 1]$ along the rubber band).

- (a) Give a simple algorithm to compute a shortest *cyclic* tour of P .
- (b) A *monotonic* tour of P is a tour that never crosses itself. Here's an illustration.



(a) A monotonic tour of P . (b) A non-monotonic tour of P .

Prove that any shortest tour of P must be monotonic.

- (c) Given an algorithm to compute a shortest tour of P starting at point $P[1]$ and finishing on point $P[\lfloor \frac{n}{2} \rfloor]$.

Practice Problems

These remaining practice problems are entirely for your benefit. Don't turn in solutions—we'll just throw them out—but feel free to ask us about these questions during office hours and review sessions. Think of these as potential exam questions (hint, hint).

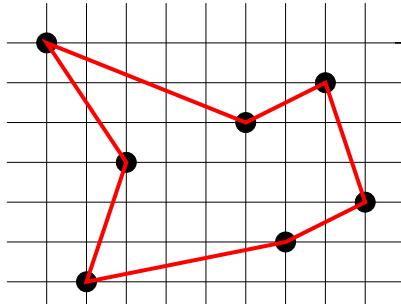
1. Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:
 - (a) the square immediately above,
 - (b) the square that is one up and one left (but only if the checker is not already in the leftmost column),
 - (c) the square that is one up and one right (but only if the checker is not already in the rightmost column).

Each time you move from square x to square y , you receive $p(x, y)$ dollars. You are given $p(x, y)$ for all pairs (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

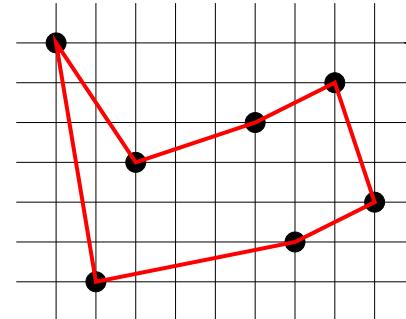
Give an algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. Your algorithm is free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination in order to maximize the number of dollars gathered along the way. What is the running time of your algorithm?

2. (CLRS 15-1) The *euclidean traveling-salesman problem* is the problem of determining the shortest closed tour that connects a given set of n points in the plane. Figure (a) below shows the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time.

J.L. Bentley has suggested that we simplify the problem by restricting our attention to *bitonic tours* (Figure (b) below). That is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. In this case, a polynomial-time algorithm is possible.



(a)



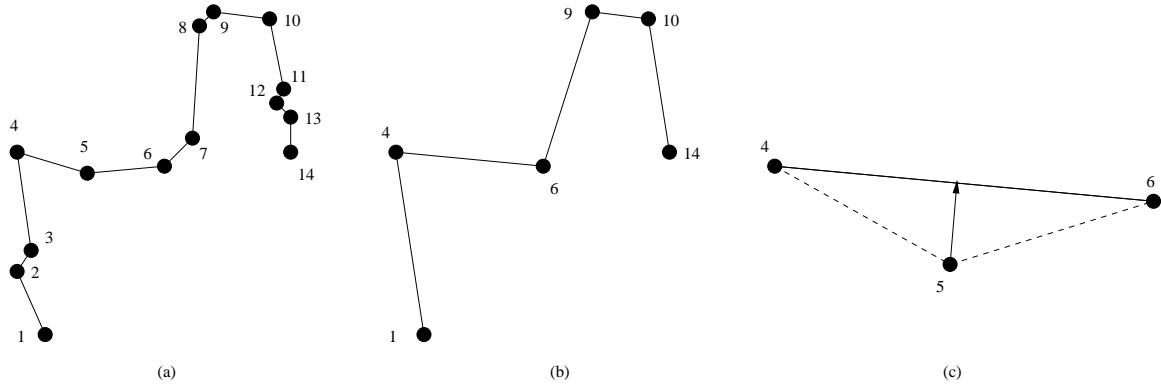
(b)

Seven points in the plane, shown on a unit grid. (a) The shortest closed tour, with length approximately 24.89. This tour is not bitonic. (b) The shortest bitonic tour for the same set of points. Its length is approximately 25.58.

Describe an $O(n^2)$ -time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x -coordinate. [Hint: Scan left to right, maintaining optimal possibilities for the two parts of the tour.]

3. You are given a polygonal line γ made out of n vertices in the plane. Namely, you are given a list of n points in the plane p_1, \dots, p_n , where $p_i = (x_i, y_i)$. You need to display this polygonal line on the screen, however, you realize that you might be able to draw a polygonal line with considerably less vertices that looks identical on the screen (because of the limited resolution of the screen). It is crucial for you to minimize the number of vertices of the polygonal line. (Because, for example, your display is a remote Java applet running on the user computer, and for each vertex of the polygon you decide to draw, you need to send the coordinates of the points through the network which takes a *long long long time*. So the fewer vertices you send, the snappier your applet would be.)

So, given such a polygonal line γ , and a parameter k , you would like to select k vertices of γ that yield the “best” polygonal line that looks like γ .



(a) The original polygonal line with 14 vertices. (b) A new polygonal line with 6 vertices. (c) The distance between p_5 on the original polygonal line and the simplification segment p_4p_6 . The error of p_5 is $\text{error}(p_5) = \text{dist}(p_5, p_4p_6)$.

Namely, you need to build a new polygonal line γ' and minimize the difference between the two polygonal-lines. The polygonal line γ' is built by selecting k vertices $\{p_{i_1}, p_{i_2}, \dots, p_{i_k}\}$ from γ . It is required that $i_1 = 1$, $i_k = n$, and $i_j < i_{j+1}$ for $j = 1, 2, \dots, k - 1$.

We define the error between γ and γ' by how far from γ' are the vertices of γ . More formally, The difference between the two polygonal lines is

$$\text{error}(\gamma, \gamma') = \sum_{j=1}^{k-1} \sum_{m=i_j+1}^{i_{j+1}-1} \text{dist}(p_m, p_{i_j}p_{i_{j+1}}).$$

Namely, for every vertex not in the simplification, its associated error, is the distance to the corresponding simplification segment (see (c) in above figure). The overall error is the sum over all vertices.

You can assume that you are provided with a subroutine that can calculate $\text{dist}(u, vw)$ in constant time, where $\text{dist}(u, vw)$ is the distance between the point u and the segment vw .

Give an $O(n^3)$ time algorithm to find the γ' that minimizes $\text{error}(\gamma, \gamma')$.

CS 373: Combinatorial Algorithms, Fall 2002

Homework 2 (due Thursday, September 26, 2002 at 11:59:59 p.m.)

Name:		
Net ID:	Alias:	U G

Name:		
Net ID:	Alias:	U G

Name:		
Net ID:	Alias:	U G

Homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since graduate students are required to solve problems that are worth extra credit for other students, **Grad students may not be on the same team as undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate or 1-unit grad student by circling U or G, respectively. Staple this sheet to the top of your homework. **NOTE: You must use different sheet(s) of paper for each problem assigned.**

Required Problems

1. For each of the following problems, the input is a set of n nuts and n bolts. For each bolt, there is exactly one nut of the same size. Direct comparisons between nuts or between bolts are not allowed, but you can compare a nut and a bolt in constant time.
 - (a) Describe and analyze a deterministic algorithm to find the largest bolt. *Exactly* how many comparisons does your algorithm perform in the worst case? [Hint: This is very easy.]
 - (b) Describe and analyze a randomized algorithm to find the largest bolt. What is the *exact* expected number of comparisons performed by your algorithm?
 - (c) Describe and analyze an algorithm to find the largest and smallest bolts. Your algorithm can be either deterministic or randomized. What is the *exact* worst-case expected number of comparisons performed by your algorithm? [Hint: Running part (a) twice is definitely not the most efficient algorithm.]

In each case, to receive **full** credit, you need to describe the most efficient algorithm possible.

2. Consider the following algorithm:

```

SLOWSHUFFLE( $A[1..n]$ ) :
  for  $i \leftarrow 1$  to  $n$ 
     $B[i] \leftarrow \text{Null}$ 
  for  $i \leftarrow 1$  to  $n$ 
    index  $\leftarrow \text{Random}(1,n)$ 
    while  $B[\text{index}] \neq \text{Null}$ 
      index  $\leftarrow \text{Random}(1,n)$ 
     $B[\text{index}] \leftarrow A[i]$ 
  for  $i \leftarrow 1$  to  $n$ 
     $A[i] \leftarrow B[i]$ 

```

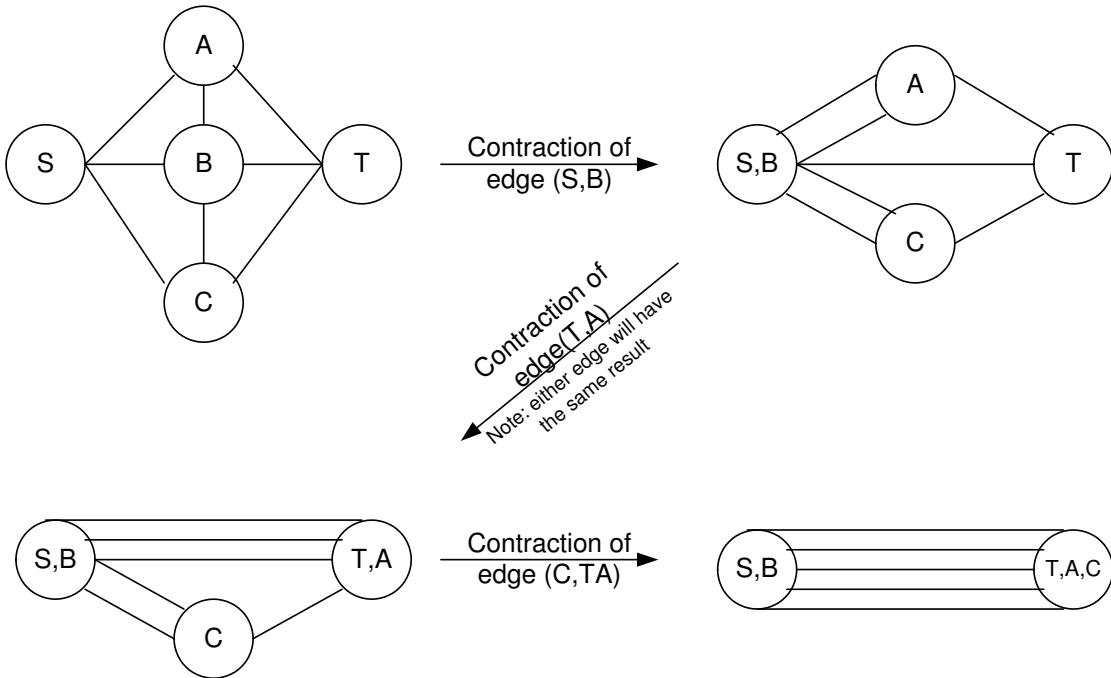
Suppose that $\text{Random}(i,j)$ will return a random number between i and j inclusive in constant time. SLOWSHUFFLE will shuffle the input array into a random order such that every permutation is equally likely.

- (a) What is the expected running time of the above algorithm. Justify your answer and give a tight asymptotic bound.
- (b) Describe an algorithm that randomly shuffles an n -element array, so that every permutation is *equally* likely, in $O(n)$ time.

3. Suppose we are given an undirected graph $G = (V, E)$ together with two distinguished vertices s and t . An **s-t min-cut** is a set of edges that once removed from the graph, will disconnect s from t . We want to find such a set with the minimum cardinality (The smallest number of edges). In other words, we want to find the smallest set of edges that will separate s and t

To do this we repeat the following step $|V| - 2$ times: Uniformly at random, pick an edge from the set E which contains all edges in the graph excluding those that directly connects vertices s and t . Merge the two vertices that are connected by this randomly selected edge. If as a result there are several edges between some pair of vertices, retain them all. Edges that are between the two merged vertices are removed so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single vertex as the *contraction* of that edge. Notice with each contraction the number of vertices of G decreases by one.

As this algorithm proceeds, the vertex s may get merged with a new vertex as the result of an edge being contracted. We call this vertex the s -vertex. Similarly, we have a t -vertex. During the contraction algorithm, we ensure that we never contract an edge between the s -vertex and the t -vertex.



- (a) Give an example of a graph in which the probability that this algorithm finds an s - t min-cut is exponentially small($O(1/a^n)$). Justify your answers.
(Hint: Think multigraphs)
- (b) Give an example of a graph such that there are $O(2^n)$ number of s - t min-cuts. Justify your answers.
4. Describe a modification of treaps that supports the following operations, each in $O(\log n)$ expected time:
- **INSERT(x):** Insert a new element x into the data structure.
 - **DELETE(x):** Delete an element x from the data structure.
 - **COMPUTERANK(x):** Return the number of elements in the data structure less than or equal to x .
 - **FINDBYRANK(r):** Return the k th smallest element in the data structure.

Describe and analyze the algorithms that implement each of these operations. *[Hint: Don't reinvent the wheel!]*

5. A *meldable priority queue* stores a set of keys from some totally ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue storing the empty set.
- **FINDMIN(Q)**: Return the smallest element stored in Q (if any).
- **DELETEMIN(Q)**: Delete the smallest element stored in Q (if any).
- **INSERT(Q, x)**: Insert element x into Q .
- **MELD(Q_1, Q_2)**: Return a new priority queue containing all the elements stored in Q_1 and Q_2 . The component priority queues are destroyed.
- **DECREASEKEY(Q, x, y)**: Replace an element x of Q with a smaller key y . (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q storing x .
- **DELETE(Q, x)**: Delete an element $x \in Q$. The input is a pointer directly to the node in Q storing x .

A simple way to implement this data structure is to use a heap-ordered binary tree, where each node stores an element, a pointer to its left child, a pointer to its right child, and a pointer to its parent. **MELD(Q_1, Q_2)** can be implemented with the following randomized algorithm.

- If either one of the queues is empty, return the other one.
 - If the root of Q_1 is smaller than the root of Q_2 , then recursively MELD Q_2 with either **right(Q_1)** or **left(Q_1)**, each with probability $1/2$.
 - Similarly, if the root of Q_2 is smaller than the root of Q_1 , then recursively MELD Q_1 with a randomly chosen child of Q_2 .
- (a) Prove that for *any* heap-ordered trees Q_1 and Q_2 , the expected running time of **MELD(Q_1, Q_2)** is $O(\log n)$, where $n = |Q_1| + |Q_2|$. [Hint: How long is a random path in an n -node binary tree, if each left/right choice is made with equal probability?] For extra credit, prove that the running time is $O(\log n)$ with high probability.
- (b) Show that each of the operations **DELETEMIN**, **INSERT**, **DECREASEKEY**, and **DELETE** can be implemented with one call to **MELD** and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ with high probability.)
6. [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

The following randomized algorithm selects the r th smallest element in an unsorted array $A[1,..,n]$. For example, to find the smallest element, you would call **RANDOMSELECT($A, 1$)**; to find the median element, you would call **RANDOMSELECT($A, \lfloor n/2 \rfloor$)**. Recall from lecture that **PARTITION** splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element.

```
RANDOMSELECT( $A[1..n], r$ ) :  
     $p \leftarrow \text{RANDOM}(1, n)$   
     $k \leftarrow \text{PARTITION}(A[1..n], p)$   
    if  $r < k$   
        return RANDOMSELECT( $A[1..k - 1], r$ )  
    else if  $r > k$   
        return RANDOMSELECT( $A[k + 1..n], r - k$ )  
    else  
        return  $A[k]$ 
```

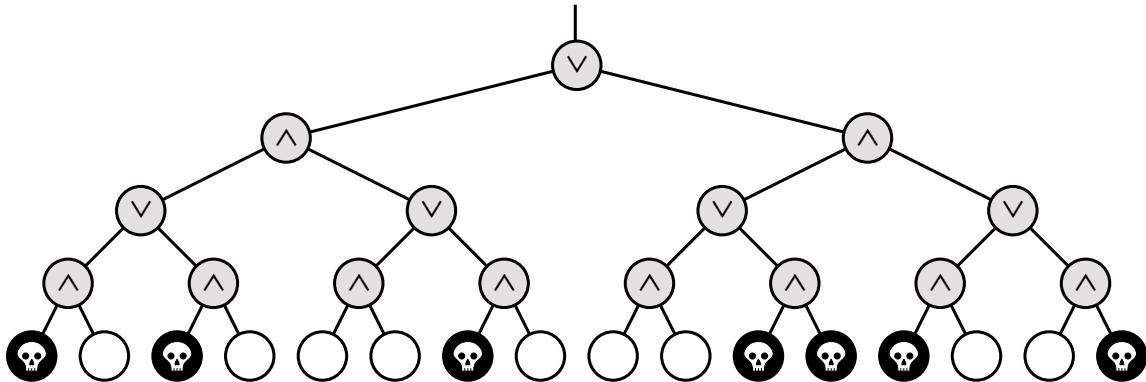
- (a) State a recurrence for the expected running time of RANDOMSELECT, as a function of both n and r .
- (b) What is the *exact* probability that RANDOMSELECT compares the i th smallest and j th smallest elements in the input array? The correct answer is a simple function of i , j , and r . [Hint: Check your answer by trying a few small examples.]
- (c) Show that for any n and r , the expected running time of RANDOMSELECT is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, find the *exact* expected number of comparisons, as a function of n and r .
- (d) What is the expected number of times that RANDOMSELECT calls itself recursively?

Practice Problems

1. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.

You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- (a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*
- (b) Unfortunately, Death won't let you even look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $\Theta(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*



2. What is the *exact* number of nodes in a skip list storing n keys, *not* counting the sentinel nodes at the beginning and end of each level? Justify your answer.
3. Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k . Describe an algorithm to find the k th smallest element in the union of A and B . (For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, our algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicates. Your algorithm should be able to run in $\Theta(\log n)$ time. *[Hint: First try to solve the special case $k = n$.]*

CS 373: Combinatorial Algorithms, Fall 2002
Homework 3, due October 17, 2002 at 23:59:59

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Undergrads	
Grads	

This homework is to be submitted in groups of up to three people. Graduate and undergraduate students are *not* allowed to work in the same group. Please indicate above whether you are undergraduate or graduate students. Only *one* submission per group will be accepted.

Required Problems

1.
 - (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.
 - (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v . A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)
 - *(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory.
 2. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:
 - After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
 - After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (it makes it much more difficult).

3. A stack is a FILO/LIFO data structure that represents a stack of objects; access is only allowed at the top of the stack. In particular, a stack implements two operations:

 - $\text{PUSH}(x)$: adds x to the top of the stack.

- **POP:** removes the top element and returns it.

A queue is a FIFO/LIFO data structure that represents a row of objects; elements are added to the front and removed from the back. In particular, a queue implements two operations:

- **ENQUEUE(x):** adds x to the front of the queue.
- **DEQUEUE:** removes the element at the back of the queue and returns it.

Using two stacks and no more than $O(1)$ additional space, show how to simulate a queue for which the operations ENQUEUE and DEQUEUE run in constant amortized time. You should treat each stack as a black box (i.e., you may call PUSH and POP, but you do not have access to the underlying stack implementation). Note that each PUSH and POP performed by a stack takes $O(1)$ time.

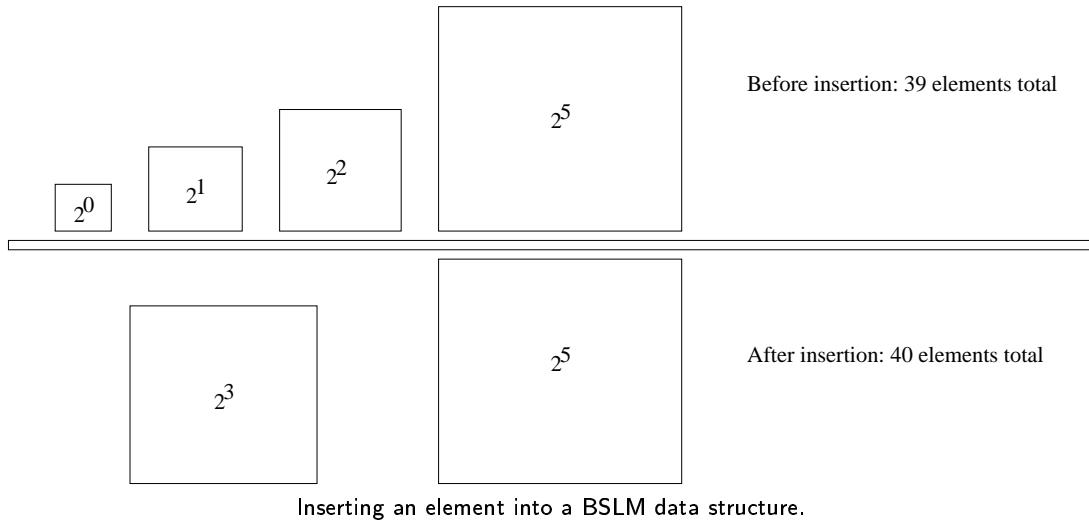
4. A data structure is *insertion-disabled* if there is no way to add elements to it. For the purposes of this problem, further assume that an insertion-disabled data structure implements the following operations with the given running times:

- **INITIALIZE(S):** Return an insertion-disabled data structure that contains the elements of S . Running time: $O(n \log n)$.
- **SEARCH(D, x):** Return TRUE if x is in D ; return FALSE if not. Running time: $O(\log n)$.
- **RETURNALL(D, x):** Return an unordered set of all elements in D . Running time: $O(n)$.
- **DELETE(D, x):** Remove x from D if x is in D . Running time: $O(\log n)$.

Using an approach known as the Bentley-Saxe Logarithmic Method (BSLM), it is possible to represent a dynamic (i.e., supports insertions) data structure with a collection of insertion-disabled data structures, where each insertion-disabled data structure stores a number of elements that is a distinct power of two. For example, to store $39 = 2^0 + 2^1 + 2^2 + 2^5$ elements in a BSLM data structure, we use four insertion-disabled data structures with 2^0 , 2^1 , 2^2 , and 2^5 elements.

To find an element in a BSLM data structure, we search the collection of insertion-disabled data structures until we find (or don't find) the element.

To insert an element into a BSLM data structure, we think about adding a 2^0 -size insertion-disabled data structure. However, an insertion-disabled data structure with 2^0 elements may already exist. In this case, we can combine two 2^0 -size structures into a single 2^1 -size structure. However, there may already be a 2^1 -size structure, so we will need to repeat this process. In general, we do the following: Find the smallest i such that for all nonnegative $k < i$, there is a 2^k -sized structure in our collection. Create a 2^i -sized structure that contains the element to be inserted and all elements from 2^k -sized data structures for all $k < i$. Destroy all 2^k -sized data structures for $k < i$.



We delete elements from the BSLM data structure lazily. To delete an element, we first search the collection of insertion-disabled data structures for it. Then we call `DELETE` to remove the element from its insertion-disabled data structure. This means that a 2^i -sized insertion-disabled data structure might store less than 2^i elements. That's okay; we just say that it stores 2^i elements and say that 2^i is its *pretend size*. We keep track of a single variable, called *Waste*, which is initially 0 and is incremented by 1 on each deletion. If *Waste* exceeds three-quarters of the total pretend size of all insertion-disabled data structures in our collection (i.e., the total number of elements stored), we rebuild our collection of insertion-disabled data structures. In particular, we create a 2^m -sized insertion-disabled data structure, where 2^m is the smallest power that is greater than or equal to the total number of elements stored. All elements are stored in this 2^m -sized insertion-disabled data structure, and all other insertion-disabled data structures in our collection are destroyed. *Waste* is reset to $2^m - n$, where n is the total number of elements stored in the BSLM data structure.

Your job is to prove the running times of the following three BSLM operations:

- `SEARCHBSLM(D, x)`: Search for x in the collection of insertion-disabled data structures that represent the BSLM data structure D . Running time: $O(\log^2 n)$ worst-case.
 - `INSERTBSLM(D, x)`: Insert x into the collection of insertion-disabled data structures that represent the BSLM data structure D , modifying the collection as necessary. Running time: $O(\log^2 n)$ amortized.
 - `DELETEBSLM(D, x)`: Delete x from the collection of insertion-disabled data structures that represent the BSLM data structure D , rebuilding when there is a lot of wasted space. Running time: $O(\log^2 n)$ amortized.
5. Except as noted, the following sub-problems refer to a Union-Find data structure that uses both path compression and union by rank.
- Prove that in a set of n elements, a sequence of n consecutive `FIND` operations takes $O(n)$ total time.
 - Show that any sequence of m `MAKESET`, `FIND`, and `UNION` operations takes only $O(m)$ time if all of the `UNION` operations occur before any of the `FIND` operations.

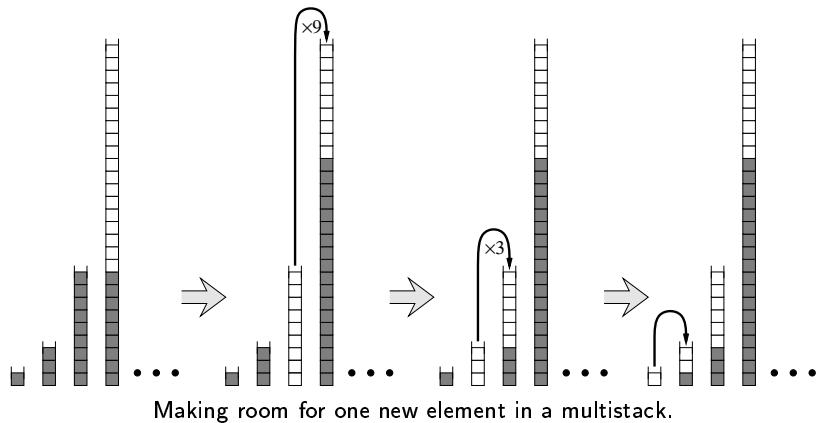
- (c) Now consider part b with a Union-Find data structure that uses path compression but does *not* use union by rank. Is $O(m)$ time still correct? Prove your answer.
6. [This problem is required only for graduate students (including I2CS students); undergrads can submit a solution for extra credit.]

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 i th least significant fit indicates whether the sum includes the i th Fibonacci number F_i . For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string.]

Practice Problems

These remaining practice problems are entirely for your benefit. Don't turn in solutions—we'll just throw them out—but feel free to ask us about these questions during office hours and review sessions. Think of these as potential exam questions (hint, hint).

1. A *multistack* consists of an infinite series of stacks S_0, S_1, S_2, \dots , where the i th stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i , we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2} , and so on. Moving a single element from one stack to the next takes $O(1)$ time.



- (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?
(b) Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.
2. A hash table of size m is used to store n items with $n \leq m/2$. Open addressing is used for collision resolution.
 - Assuming uniform hashing, show that for $i = 1, 2, \dots, n$, the probability that the i^{th} insertion requires strictly more than k probes is at most 2^{-k} .
 - Show that for $i = 1, 2, \dots, n$, the probability that the i^{th} insertion requires more than $2 \lg n$ probes is at most $1/n^2$.

Let the random variable X_i denote the number of probes required by the i^{th} insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

- Show that $\Pr\{X > 2 \lg n\} \leq 1/n$.
- Show that the expected length of the longest probe sequence is $E[X] = O(\lg n)$.

3. A sequence of n operations is performed on a data structure. The i th operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

- (a) Aggregate method
- (b) Accounting method
- *(c) Potential method

CS 373: Combinatorial Algorithms, Fall 2002

Homework 4, due Thursday, October 31, 2002 at 23:59.99

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Undergrads	Grads

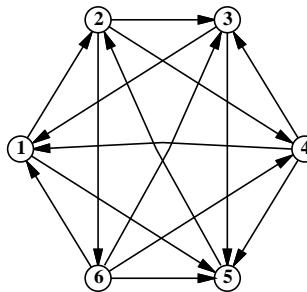
This homework is to be submitted in groups of up to three people. Graduate and undergraduate students are *not* allowed to work in the same group. Please indicate above whether you are undergraduate or graduate students. Only *one* submission per group will be accepted.

Required Problems

1. Tournament:

A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once.

Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

2. Acrophobia:

Consider a graph $G = (V, E)$ whose nodes are cities, and whose edges are roads connecting the cities. For each edge, the weight is assigned by h_e , the maximum altitude encountered when traversing the specified road. Between two cities s and t , we are interested in those paths whose maximum altitude is as low as possible. We will call a subgraph, G' , of G an *acrophobic friendly subgraph*, if for any two nodes s and t the path of minimum altitude is always

included in the subgraph. For simplicity, assume that the maximum altitude encountered on each road is unique.

- (a) Prove that every graph of n nodes has an acrophobic friendly subgraph that has only $n - 1$ edges.
- (b) Construct an algorithm to find an acrophobic friendly subgraph given a graph $G = (V, E)$.
3. Refer to the lecture notes on single-source shortest paths. The **GENERICSSSP** algorithm described in class can be implemented using a stack for the 'bag'. Prove that the resulting algorithm, given a graph with n nodes as input, could perform $\Omega(2^n)$ relaxation steps before stopping. You need to describe, for any positive integer n , a specific weighted directed n -vertex graph that forces this exponential behavior. The easiest way to describe such a family of graphs is using an *algorithm*!

4. Neighbors:

Two spanning trees T and T' are defined as *neighbors* if T' can be obtained from T by swapping a single edge. More formally, there are two edges e and f such that T' is obtained from T by adding edge e and deleting edge f .

- (a) Let T denote the minimum cost spanning tree and suppose that we want to find the second cheapest tree T' among all trees. Assuming unique costs for all edges, prove that T and T' are neighbors.
- (b) Given a graph $G = (V, E)$, construct an algorithm to find the second cheapest tree, T' .
- (c) Consider a graph, H , whose vertices are the spanning trees of the graph G . Two vertices are connected by an edge if and only if they are neighbors as previously defined. Prove that for any graph G this new graph H is connected.

5. Network Throughput:

Suppose you are given a graph of a (tremendously simplified) computer network $G = (V, E)$ such that a weight, b_e , is assigned to each edge representing the communication bandwidth of the specified channel in Kb/s and each node is assigned a value, l_v , representing the server latency measured in seconds/packet. Given a fixed packet size, and assuming all edge bandwidth values are a multiple of the packet size, your job is to build a system to decide which paths to route traffic between specified servers.

More formally, a person wants to route traffic from server s to server t along the path of maximum throughput. Give an algorithm that will allow a network design engineer to choose an optimal path by which to route data traffic.

6. All-Pairs-Shortest-Path:

[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Given an undirected, unweighted, connected graph $G = (V, E)$, we wish to solve the distance version of the all-pairs-shortest-path problem. The algorithm APD takes the $n \times n$ 0-1 adjacency matrix A and returns an $n \times n$ matrix D such that d_{ij} represents the shortest path between vertices i and j .

```

APD( $A$ )
 $Z \leftarrow A \cdot A$ 
let  $B$  be an  $n \times n$  matrix, where  $b_{ij} = 1$  iff  $i \neq j$  and ( $a_{ij} = 1$  or  $z_{ij} > 0$ )
if  $b_{ij} = 1$  for all  $i \neq j$ 
    return  $D \leftarrow 2B - A$ 
 $T \leftarrow APD(B)$ 
 $X \leftarrow T \cdot A$ 
foreach  $x_{ij}$ 
    if  $x_{ij} \geq t_{ij} \cdot \text{degree}(j)$ 
         $d_{ij} \leftarrow 2t_{ij}$ 
    else
         $d_{ij} \leftarrow 2t_{ij} - 1$ 
return  $D$ 

```

- (a) In the APD algorithm above, what do the matrices Z , B , T , and X represent? Justify your answers.
- (b) Prove that the APD algorithm correctly computes the matrix of shortest path distances. In other words, prove that in the output matrix D , each entry d_{ij} represents the shortest path distance between node i and node j .
- (c) Suppose we can multiply two $n \times n$ matrices in $M(n)$ time, where $M(n) = \Omega(n^2)$.¹ Prove that APD runs in $O(M(n) \log n)$ time.

¹The matrix multiplication algorithm you already know runs in $O(n^3)$ time, but this is not the fastest known. The current record is $M(n) = O(n^{2.376})$, due to Don Coppersmith and Shmuel Winograd. Determining the smallest possible value of $M(n)$ is a long-standing open problem.

Practice Problems

1. Makefiles:

In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called ‘make’ that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files which are listed. Thus a source file must be recompiled if a file on which it depends is changed.

Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design an algorithm to recompile only those necessary. DO NOT worry about the details of parsing a Makefile.

2. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} 1 & \text{if vertex } v_i \text{ is an endpoint of edge } e_j \\ 0 & \text{otherwise} \end{cases}$$

- (a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose). Justify.
- (b) Describe what all the entries of the matrix product B^TB represent. Justify.
- ★(c) Let $C = BB^T - 2A$. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees. (A is the adjacency matrix of G , with zeroes on the diagonal).

3. Reliable Network:

Suppose you are given a graph of a computer network $G = (V, E)$ and a function $r(u, v)$ that gives a reliability value for every edge $(u, v) \in E$ such that $0 \leq r(u, v) \leq 1$. The reliability value gives the probability that the network connection corresponding to that edge will *not* fail. Describe and analyze an algorithm to find the most reliable path from a given source vertex s to a given target vertex t .

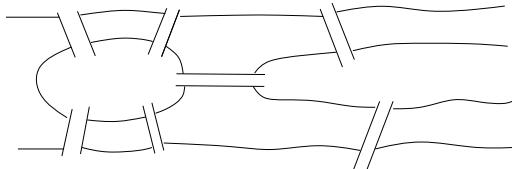
4. Aerophobia:

After graduating you find a job with Aerophobes-R'-Us, the leading traveling agency for aerophobic people. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying so the trip should be as short as possible.

In other words, a person wants to fly from city A to city B in the shortest possible time. S/he turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose an optimal route to minimize the total time in transit. Hint: rather than modify Dijkstra’s algorithm, modify the data. The total transit time is from departure to arrival at the destination, so it will include layover time (time waiting for a connecting flight).

5. The Seven Bridges of Königsberg:

During the eighteenth century the city of Königsberg in East Prussia was divided into four sections by the Pregel river. Seven bridges connected these regions, as shown below. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to their starting point.



- (a) Show how the residents of the city could accomplish such a walk or prove no such walk exists.
- (b) Given any undirected graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge exactly once, or says that it can't be done.
- 6. Given an undirected graph $G = (V, E)$ with costs $c_e \geq 0$ on the edges $e \in E$ give an $O(|E|)$ time algorithm that tests if there is a minimum cost spanning tree that contains the edge e .

7. Combining Boruvka and Prim:

Give an algorithm that finds the MST of a graph G in $O(m \log \log n)$ time by combining Boruvka's and Prim's algorithm.

8. Minimum Spanning Tree changes:

Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).

- (a) Give an algorithm to update the MST when an edge is added to G .
- (b) Give an algorithm to update the MST when an edge is deleted from G .
- (c) Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G .

9. Nesting Envelopes

You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B . Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

10. $o(V^2)$ Adjacency Matrix Algorithms

- (a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.

- (b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V - 2$ (the body) connected to the other $V - 3$ vertices (the feet). Some of the feet may be connected to other feet.

Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

- (c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

11. Shortest Cycle:

Given an **undirected** graph $G = (V, E)$, and a weight function $f : E \rightarrow \mathbf{R}$ on the **edges**, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G .

12. Longest Simple Path:

Let graph $G = (V, E)$, $|V| = n$. A *simple path* of G , is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G . Hint: It can be done in $O(n^c 2^n)$ time, for some constant c .

13. Minimum Spanning Tree:

Suppose all edge weights in a graph G are equal. Give an algorithm to compute an MST.

14. Transitive reduction:

Give an algorithm to construct a *transitive reduction* of a directed graph G , i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR} .

15. (a) What is $5^{2^{29}5^0 + 23^41 + 17^32 + 11^23 + 5^14} \pmod{6}$?

- (b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Fall 2002

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 5 (due Thur. Nov. 21, 2002 at 11:59 pm)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, $\frac{3}{4}$ -unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. (10 points) Given two arrays, $A[1..n]$ and $B[1..m]$ we want to determine whether there is an $i \geq 0$ such that $B[1] = A[i + 1], B[2] = A[i + 2], \dots, B[m] = A[i + m]$. In other words, we want to determine if B is a substring of A . Show how to solve this problem in $O(n \log n)$ time with high probability.
2. (5 points) Let $a, b, c \in \mathbb{Z}^+$.
 - (a) Prove that $\gcd(a, b) \cdot \text{lcm}(a, b) = ab$.
 - (b) Prove $\text{lcm}(a, b, c) = \text{lcm}(\text{lcm}(a, b), c)$.
 - (c) Prove $\gcd(a, b, c)\text{lcm}(ab, ac, bc) = abc$.
3. (5 points) Describe an efficient algorithm to compute multiplicative inverses modulo a prime p . Does your algorithm work if the modulus is composite?
4. (10 points) Describe an efficient algorithm to compute $F_n \bmod m$, given integers n and m as input.

5. (10 points) Let n have the prime factorization $p_1^{k_1} p_2^{k_2} \cdots p_t^{k_t}$, where the primes p_i are distinct and have exponents $k_i > 0$. Prove that

$$\phi(n) = \prod_{i=1}^t p_i^{k_i-1} (p_i - 1).$$

Conclude that $\phi(n)$ can be computed in polynomial time given the prime factorization of n .

6. (10 points) Suppose we want to compute the Fast Fourier Transform of an integer vector $P[0..n-1]$. We could choose an integer m larger than any coefficient $P[i]$, and then perform all arithmetic modulo m (or more formally, in the ring \mathbb{Z}_m). In order to make the FFT algorithm work, we need to find an integer that functions as a "primitive n th root of unity modulo m ".

For this problem, let's assume that $m = 2^{n/2} + 1$, where as usual n is a power of two.

- (a) Prove that $2^n \equiv 1 \pmod{m}$.
 - (b) Prove that $\sum_{k=0}^{n-1} 2^k \equiv 0 \pmod{m}$. These two conditions imply that 2 is a primitive n th root of unity in \mathbb{Z}_m .
 - (c) Given (a), (b), and (c), *briefly* argue that the "FFT modulo m " of P is well-defined and can be computed in $O(n \log n)$ arithmetic operations.
 - (d) Prove that n has a multiplicative inverse in \mathbb{Z}_m . [Hint: n is a power of 2, and m is odd.] We need this property to implement the inverse FFT modulo m .
 - (e) What is the FFT of the sequence $[3, 1, 3, 3, 7, 3, 7, 3]$ modulo 17?
7. (10 points) [*This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.*]
- (a) Prove that for any integer $n > 1$, if the n -th Fibonacci number F_n is prime then either n is prime or $n = 4$.
 - (b) Prove that if a divides b , then F_a divides F_b .
 - (c) Prove that $\gcd(F_a, F_b) = F_{\gcd(a,b)}$. This immediately implies parts (a) and (b), so if you solve this part, you don't have to solve the other two.

Practice Problems

1. Let $a, b, n \in \mathbb{Z} \setminus \{0\}$. Assume $\gcd(a, b) | n$. Prove the entire set of solutions to the equation

$$n = ax + by$$

is given by:

$$\Gamma = \left\{ x_0 + \frac{tb}{\gcd(a, b)}, y_0 - \frac{ta}{\gcd(a, b)} : t \in \mathbb{Z} \right\}.$$

2. Show that in the RSA cryptosystem the decryption exponent d can be chosen such that $de \equiv 1 \pmod{\text{lcm}(p-1, q-1)}$.

3. Let (n, e) be a public RSA key. For a plaintext $m \in \{0, 1, \dots, n-1\}$, let $c = m^e \pmod{n}$ be the corresponding ciphertext. Prove that there is a positive integer k such that

$$m^{e^k} \equiv m \pmod{n}.$$

For such an integer k , prove that

$$c^{e^{k-1}} \equiv m \pmod{n}.$$

Is this dangerous for RSA?

4. Prove that if Alice's RSA public exponent e is 3 and an adversary obtains Alice's secret exponent d , then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n .

CS 373: Combinatorial Algorithms, Fall 2002

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 6 (Do not hand in!)

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

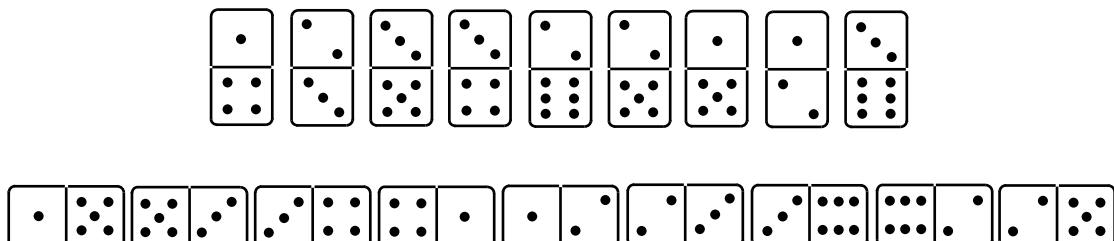
Name:		
Net ID:	Alias:	U $\frac{3}{4}$ 1

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $\frac{3}{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

- (10 points) Prove that SAT is still a NP-complete problem even under the following constraints: each variable must show up once as a positive literal and once or twice as a negative literal in the whole expression. For instance, $(A \vee \bar{B}) \wedge (\bar{A} \vee C \vee D) \wedge (\bar{A} \vee B \vee \bar{C} \vee \bar{D})$ satisfies the constraints, while $(A \vee \bar{B}) \wedge (\bar{A} \vee C \vee D) \wedge (A \vee B \vee \bar{C} \vee \bar{D})$ does not, because positive literal A appears twice.
- (10 points) A domino is 2×1 rectangle divided into two squares, with a certain number of pips(dots) in each square. In most domino games, the players lay down dominos at either end of a single chain. Adjacent dominos in the chain must have matching numbers. (See the figure below.)

Describe and analyze an efficient algorithm, or prove that it is NP-complete, to determine whether a given set of n dominos can be lined up in a single chain. For example, for the sets of dominos shown below, the correct output is TRUE.



Top: A set of nine dominos
Bottom: The entire set lined up in a single chain

3. (10 points) Prove that the following 2 problems are NP-complete. Given an undirected Graph $G = (V, E)$, a subset of vertices $V' \subseteq V$, and a positive integer k :

- (a) determine whether there is a spanning tree T of G whose leaves are the same as V' .
- (b) determine whether there is a spanning tree T of G whose degree of vertices are all less than k .

4. (10 points) An optimized version of Knapsack problem is defined as follows. Given a finite set of elements U where each element of the set $u \in U$ has its own size $s(u) > 0$ and the value $v(u) > 0$, maximize $A(U') = \sum_{u \in U'} v(u)$ under the condition $\sum_{u \in U'} s(u) \leq B$ and $U' \subseteq U$. This problem is NP-hard. Consider the following polynomial time approximation algorithm. Determine the worst case approximation ratio $R(U) = \max_U Opt(U)/Approx(U)$ and prove it.

```
APPROXIMATION ALGORITHM:
A1 ← Greedy()
A2 ← SingleElement()
return max(A1, A2)
```

```
GREEDY:
Put all the elements  $u \in U$  into an array  $A[i]$ 
Sort  $A[i]$  by  $v(u)/s(u)$  in a decreasing order
 $S \leftarrow 0$ 
 $V \leftarrow 0$ 
for  $i \leftarrow 0$  to NumOfElements
    if  $(S + s(u[i]) > B)$ 
        break
     $S \leftarrow S + s(u[i])$ 
     $V \leftarrow V + v(u[i])$ 
return  $V$ 
```

```
SINGLE ELEMENT:
Put all the elements  $u \in U$  into an array  $A[i]$ 
 $V \leftarrow 0$ 
for  $i \leftarrow 0$  to NumOfElements
    if  $(s(u[i]) \leq B \& V < v(u[i]))$ 
         $V \leftarrow v(u[i])$ 
return  $V$ 
```

5. (10 points) The recursion fairy's distant cousin, the reduction genie, shows up one day with a magical gift for you: a box that determines in constant time whether or not a graph is 3-colorable.(A graph is 3-colorable if you can color each of the vertices red, green, or blue, so that every edge has do different colors.) The magic box does not tell you how to color the graph, just wheter or not it can be done. Devise and analyze an algorithm to 3-color any graph in **polynomial time** using the magic box.

6. (10 points) The following is an NP-hard version of PARTITION problem.

```
PARTITION(NP-HARD):
Given a set of  $n$  positive integers  $S = \{a_i | i = 0 \dots n - 1\}$ ,
minimize  $\max \left( \sum_{a_i \in T} a_i, \sum_{a_i \in S-T} a_i \right)$ 
where  $T$  is a subset of  $S$ .
```

A polynomial time approximation algorithm is given in what follows. Determine the worst case approximation ratio $\min_S Approx(S)/Opt(S)$ and prove it.

APPROXIMATION ALGORITHM:

```

Sort S in an increasing order
s1 ← 0
s2 ← 0
for i ← 0 to n
    if s1 ≤ s2
        s1 ← s1 + ai
    else
        s2 ← s2 + ai
result ← max(s1, s2)

```

Practice Problems

1. Construct a linear time algorithm for 2 SAT problem.

2. Assume that $P \neq NP$. Prove that there is no polynomial time approximation algorithm for an optimized version of Knapsack problem, which outputs $A(I)$ s.t. $|Opt(I) - A(I)| \leq K$ for any instance I , where K is a constant.

3. Your friend Toidi is planning to hold a party for the coming Christmas. He wants to take a picture of all the participants including himself, but he is quite **shy** and thus cannot take a picture of a person whom he does not know very well. Since he has only **shy** friends, every participant coming to the party is also **shy**. After a long struggle of thought he came up with a seemingly good idea:
 - At the beginning, he has a camera.
 - A person, holding a camera, is able to take a picture of another participant whom the person knows very well, and pass a camera to that participant.
 - Since he does not want to waste films, everyone has to be taken a picture exactly once.

Although there can be some people whom he does not know very well, he knows completely who knows whom well. Therefore, in theory, given a list of all the participants, he can determine if it is possible to take all the pictures using this idea. Since it takes only linear time to take all the pictures if he is brave enough (say “Say cheese!” N times, where N is the number of people), as a student taking CS373, you are highly expected to give him an advice:

- show him an efficient algorithm to determine if it is possible to take pictures of all the participants using his idea, given a list of people coming to the party.
- or prove that his idea is essentially facing a NP-complete problem, make him give up his idea, and give him an efficient algorithm to practice saying “Say cheese!”:

e.g., for $i \leftarrow 0$ to N
 Make him say “Say cheese!” 2^i times oops, it takes exponential time...

4. Show, given a set of numbers, that you can decide whether it has a subset of size 3 that adds to zero in polynomial time.

-
5. Given a CNF-normalized form that has at most one negative literal in each clause, construct an efficient algorithm to solve the satisfiability problem for these clauses. For instance,

$$\begin{aligned}(A \vee B \vee \bar{C}) \wedge (B \vee \bar{A}), \\ (A \vee \bar{B} \vee C) \wedge (B \vee \bar{A} \vee D) \wedge (A \vee D), \\ (\bar{A} \vee B) \wedge (B \vee \bar{A} \vee C) \wedge (C \vee D)\end{aligned}$$

satisfy the condition, while

$$\begin{aligned}(\bar{A} \vee B \vee \bar{C}) \wedge (B \vee \bar{A}), \\ (A \vee \bar{B} \vee C) \wedge (B \vee \bar{A} \vee \bar{D}) \wedge (A \vee D), \\ (\bar{A} \vee B) \wedge (B \vee \bar{A} \vee C) \wedge (\bar{C} \vee \bar{D})\end{aligned}$$

do not.

6. The ExactCoverByThrees problem is defined as follows: given a finite set X and a collection C of 3-element subsets of X , does C contain an exact cover for X , that is, a sub-collection $C' \subseteq C$ where every element of X occurs in exactly one member of C' ? Given that ExactCoverByThrees is NP-complete, show that the similar problem ExactCoverByFours is also NP-complete.

7. The *LongestSimpleCycle* problem is the problem of finding a simple cycle of maximum length in a graph. Convert this to a formal definition of a decision problem and show that it is NP-complete.

Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$ X: I don't know.

For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point. Each X earns you $\frac{1}{4}$ point. **Each incorrect answer costs you $\frac{1}{2}$ point.** Your total score will be rounded **down** to an integer. Negative scores will be rounded up to zero.

- (a) What is $\sum_{i=1}^n \frac{i}{n}$?
- (b) What is $\sum_{i=1}^n \frac{n}{i}$?
- (c) How many bits do you need to write 10^n in binary?
- (d) What is the solution of the recurrence $T(n) = 9T(n/3) + n$?
- (e) What is the solution of the recurrence $T(n) = T(n - 2) + \frac{3}{n}$?
- (f) What is the solution of the recurrence $T(n) = 5T\left(\lceil \frac{n-17}{25} \rceil - \lg \lg n\right) + \pi n + 2\sqrt{\log^* n} - 6$?
- (g) What is the worst-case running time of randomized quicksort?
- (h) The expected time for inserting one item into a randomized treap is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty treap?
- (i) Suppose STUPIDALGORITHM produces the correct answer to some problem with probability $1/n$. How many times do we have to run STUPIDALGORITHM to get the correct answer with high probability?
- (j) Suppose you correctly identify three of the possible answers to this question as obviously wrong. If you choose one of the three remaining answers at random, each with equal probability, what is your expected score for this question?

2. Consider the following algorithm for finding the smallest element in an unsorted array:

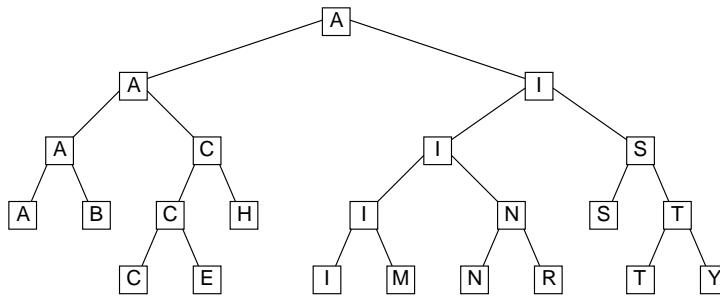
```
RANDOMMIN( $A[1..n]$ ):
     $min \leftarrow \infty$ 
    for  $i \leftarrow 1$  to  $n$  in random order
        if  $A[i] < min$ 
             $min \leftarrow A[i]$  (*)
```

return min

- (a) [1 point] In the worst case, how many times does RANDOMMIN execute line (*)?
- (b) [3 points] What is the probability that line (*) is executed during the n th iteration of the for loop?
- (c) [6 points] What is the exact expected number of executions of line (*)? (A correct $\Theta()$ bound is worth 4 points.)

3. Algorithms and data structures were developed millions of years ago by the Martians, but not quite in the same way as the recent development here on Earth. Intelligent life evolved independently on Mars' two moons, Phobos and Deimos.¹ When the two races finally met on the surface of Mars, after thousands of Phobos-orbits² of separate philosophical, cultural, religious, and scientific development, their disagreements over the proper structure of binary search trees led to a bloody (or more accurately, ichorous) war, ultimately leading to the destruction of all Martian life.

A *Phobian* binary search tree is a full binary tree that stores a set X of search keys. The root of the tree stores the *smallest* element in X . If X has more than one element, then the left subtree stores all the elements less than some pivot value p , and the right subtree stores everything else. Both subtrees are *nonempty* Phobian binary search trees. The actual pivot value p is *never* stored in the tree.



A Phobian binary search tree for the set $\{M, A, R, T, I, N, B, Y, S, C, H, E\}$.

- (a) [2 points] Describe and analyze an algorithm $\text{FIND}(x, T)$ that returns TRUE if x is stored in the Phobian binary search tree T , and FALSE otherwise.
 - (b) [2 points] Show how to perform a rotation in a Phobian binary search tree in $O(1)$ time.
 - (c) [6 points] A *Deimoid* binary search tree is almost exactly the same as its Phobian counterpart, except that the *largest* element is stored at the root, and both subtrees are Deimoid binary search trees. Describe and analyze an algorithm to transform an n -node Phobian binary search tree into a Deimoid binary search tree in $O(n)$ time, using as little additional space as possible.
4. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are five local minima in the following array:

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9
---	---	---	---	----------	---	---	---	---	---	----------	----------	---	---	----------	---

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. [Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]

¹Greek for “fear” and “panic”, respectively. Doesn’t that make you feel better?

²1000 Phobos orbits ≈ 1 Earth year

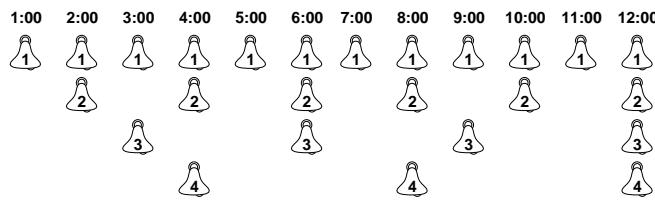
5. [Graduate students must answer this question.]

A *common supersequence* of two strings A and B is a string of minimum total length that includes both the characters of A in order and the characters of B in order. Design and analyze an algorithm to compute the length of the *shortest* common supersequence of two strings $A[1..m]$ and $B[1..n]$. For example, if the input strings are ANTHROHOPOBIOLOGICAL and PRETERDIPLOMATICALLY, your algorithm should output 31, since a shortest common supersequence of those two strings is PREANTHEROHODPOBIOPLOMATGICALY. You do not need to compute an actual supersequence, just its length. For full credit, your algorithm must run in $\Theta(nm)$ time.

Write your answers in the separate answer booklet.
This is a 90-minute exam. The clock started when you got the questions.

1. Professor Quasimodo has built a device that automatically rings the bells in the tower of the Cathédrale de Notre Dame de Paris so he can finally visit his true love Esmerelda. Every hour exactly on the hour (when the minute hand is pointing at the 12), the device rings at least one of the n bells in the tower. Specifically, the i th bell is rung once every i hours.

For example, suppose $n = 4$. If Quasimodo starts his device just after midnight, then his device rings the bells according to the following twelve-hour schedule:



What is the *amortized* number of bells rung per hour, as a function of n ? For full credit, give an exact closed-form solution; a correct $\Theta()$ bound is worth 5 points.

2. Let G be a directed graph, where every edge $u \rightarrow v$ has a weight $w(u \rightarrow v)$. To compute the shortest paths from a start vertex s to every other node in the graph, the generic single-source shortest path algorithm calls INITSSSP once and then repeatedly calls RELAX until there are no more tense edges.

INITSSSP(s):

```

dist( $s$ )  $\leftarrow 0$ 
pred( $s$ )  $\leftarrow \text{NULL}$ 
for all vertices  $v \neq s$ 
    dist( $v$ )  $\leftarrow \infty$ 
    pred( $v$ )  $\leftarrow \text{NULL}$ 

```

RELAX($u \rightarrow v$):

```

if dist( $v$ )  $>$  dist( $u$ ) +  $w(u \rightarrow v)$ 
    dist( $v$ )  $\leftarrow$  dist( $u$ ) +  $w(u \rightarrow v)$ 
    pred( $v$ )  $\leftarrow u$ 

```

Suppose the input graph has no negative cycles. Let v be an arbitrary vertex in the input graph. **Prove** that after every call to RELAX, if $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of some path from s to v .

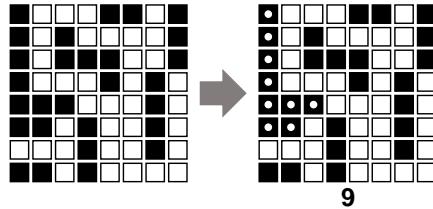
3. Suppose we want to maintain a dynamic set of values, subject to the following operations:

- **INSERT(x):** Add x to the set (if it isn't already there).
- **PRINT&DELETERANGE(a, b):** Print and delete every element x in the range $a \leq x \leq b$.
For example, if the current set is $\{1, 5, 3, 4, 8\}$, then PRINT&DELETERANGE(4, 6) prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$.

Describe and analyze a data structure that supports these operations, each with amortized cost $O(\log n)$.

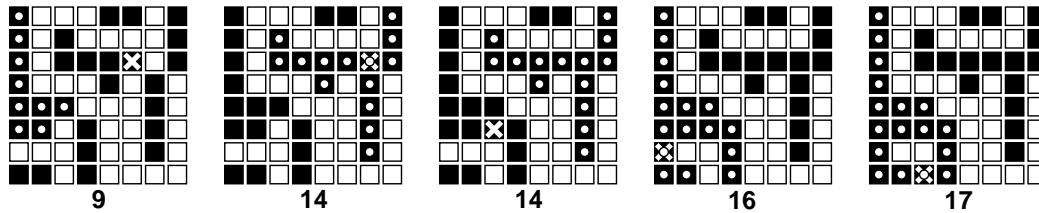
4. (a) [4 pts] Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

For example, given the bitmap below as input, your algorithm should return the number 9, because the largest connected black component (marked with white dots on the right) contains nine pixels.



- (b) [4 pts] Design and analyze an algorithm $\text{BLACKEN}(i, j)$ that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.



- (c) [2 pts] What is the *worst-case* running time of your BLACKEN algorithm?

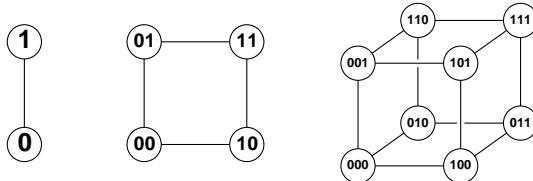
5. [Graduate students must answer this question.]

After a grueling 373 midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between bus lines at least once.

Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per day. Your goal is to minimize your *arrival time*, not the time you actually spend travelling. Assume that the buses run exactly on schedule, that you have an accurate watch, and that you are too tired to walk between bus stops.

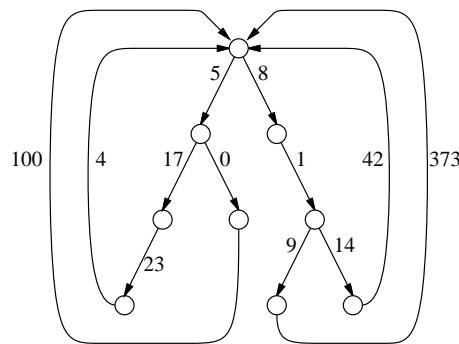
Write your answers in the separate answer booklet.
This is a 180-minute exam. The clock started when you got the questions.

1. The d -dimensional hypercube is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.



The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

- (a) [8 pts] Recall that a Hamiltonian cycle passes through every vertex in a graph exactly once. **Prove** that for all $d \geq 2$, the d -dimensional hypercube has a Hamiltonian cycle.
- (b) [2 pts] Which hypercubes have an Eulerian circuit (a closed walk that visits every edge exactly once)? [Hint: This is very easy.]
2. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight. The number of nodes in the graph is n .



- (a) How long would it take Dijkstra's algorithm to compute the shortest path between two vertices u and v in a looped tree?
- (b) Describe and analyze a faster algorithm.
3. Prove that $(x + y)^p \equiv x^p + y^p \pmod{p}$ for any prime number p .

4. A *palindrome* is a string that reads the same forwards and backwards, like X, 373, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. Any string can be written as a sequence of palindromes. For example, the string **bubbasesabananana** ('Bubba sees a banana.') can be decomposed in several ways; for example:

$$\begin{aligned} &\text{bub} + \text{baseesab} + \text{anana} \\ &\text{b} + \text{u} + \text{bb} + \text{a} + \text{sees} + \text{aba} + \text{nan} + \text{a} \\ &\text{b} + \text{u} + \text{bb} + \text{a} + \text{sees} + \text{a} + \text{b} + \text{anana} \\ &\text{b} + \text{u} + \text{b} + \text{b} + \text{a} + \text{s} + \text{e} + \text{e} + \text{s} + \text{a} + \text{b} + \text{a} + \text{n} + \text{a} + \text{n} + \text{a} \end{aligned}$$

Describe an efficient algorithm to find the *minimum* number of palindromes that make up a given input string. For example, given the input string **bubbasesabananana**, your algorithm would return the number 3.

5. Your boss wants you to find a *perfect* hash function for mapping a known set of n items into a table of size m . A hash function is perfect if there are *no* collisions; each of the n items is mapped to a different slot in the hash table. Of course, this requires that $m \geq n$.

After cursing your 373 instructor for not teaching you about perfect hashing, you decide to try something simple: repeatedly pick *random* hash functions until you find one that happens to be perfect. A random hash function h satisfies two properties:

- $\Pr[h(x) = h(y)] = \frac{1}{m}$ for any pair of items $x \neq y$.
- $\Pr[h(x) = i] = \frac{1}{m}$ for any item x and any integer $1 \leq i \leq m$.

- (a) [2 pts] Suppose you pick a random hash function h . What is the *exact* expected number of collisions, as a function of n (the number of items) and m (the size of the table)? Don't worry about how to *resolve* collisions; just count them.
- (b) [2 pts] What is the *exact* probability that a random hash function is perfect?
- (c) [2 pts] What is the *exact* expected number of different random hash functions you have to test before you find a perfect hash function?
- (d) [2 pts] What is the *exact* probability that none of the first N random hash functions you try is perfect?
- (e) [2 pts] How many random hash functions do you have to test to find a perfect hash function *with high probability*?

To get full credit for parts (a)–(d), give *exact* closed-form solutions; correct $\Theta(\cdot)$ bounds are worth significant partial credit. Part (e) requires only a $\Theta(\cdot)$ bound; an exact answer is worth extra credit.

6. Your friend Toidi is planning to hold a Christmas party. He wants to take a picture of all the participants, including himself, but he is quite shy and thus cannot take a picture of a person whom he does not know very well. Since he has only shy friends¹, everyone at the party is also shy. After thinking hard for a long time, he came up with a seemingly good idea:

- Toidi brings a disposable camera to the party.
- Anyone holding the camera can take a picture of someone they know very well, and then pass the camera to that person.
- In order not to waste any film, every person must have their picture taken exactly once.

Although there can be some people Toidi does not know very well, he knows completely who knows whom well. Thus, *in principle*, given a list of all the participants, he can determine whether it is possible to take all the pictures using this idea. But how quickly?

Either describe an efficient algorithm to solve Toidi's problem, or show that the problem is NP-complete.

7. The recursion fairy's cousin, the reduction genie, shows up one day with a magical gift for you: a box that can solve the NP-complete PARTITION problem in constant time! Given a set of positive integers as input, the magic box can tell you in constant time it can be split into two subsets whose total weights are equal.

For example, given the set $\{1, 4, 5, 7, 9\}$ as input, the magic box cheerily yells "YES!", because that set can be split into $\{1, 5, 7\}$ and $\{4, 9\}$, which both add up to 13. Given the set $\{1, 4, 5, 7, 8\}$, however, the magic box mutters a sad "Sorry, no."

The magic box does not tell you *how* to partition the set, only whether or not it can be done. Describe an algorithm to actually split a set of numbers into two subsets whose sums are equal, **in polynomial time**, using this magic box.²

¹Except you, of course. Unfortunately, you can't go to the party because you're taking a final exam. Sorry!

²Your solution to problem 4 in homework 1 does *not* solve this problem in polynomial time.

CS 373U: Combinatorial Algorithms, Spring 2004

Homework 0

Due January 28, 2004 at noon

Name:	
Net ID:	Alias:



I understand the Homework Instructions and FAQ.

-
- Neatly print your full name, your NetID, and an alias of your choice in the boxes above. Grades will be listed on the course web site by alias; for privacy reasons, your alias should not resemble your name or NetID. By providing an alias, you agree to let us list your grades; if you do not provide an alias, your grades will not be listed. *Never give us your Social Security number!*
 - Before you do anything else, read the Homework Instructions and FAQ on the course web page, and then check the box above. This web page gives instructions on how to write and submit homeworks—staple your solutions together in order, start each numbered problem on a new sheet of paper, write your name and NetID one every page, don't turn in source code, analyze and prove everything, use good English and good logic, and so on. See especially the policies regarding the magic phrases “I don’t know” and “and so on”. If you have *any* questions, post them to the course newsgroup or ask in lecture.
 - This homework tests your familiarity with prerequisite material—basic data structures, big-Oh notation, recurrences, discrete probability, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Chapters 1–10 of CLRS should be sufficient review, but you may also want consult your discrete mathematics and data structures textbooks.
 - Every homework will have five required problems and one extra-credit problem. Each numbered problem is worth 10 points.
-

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. Sort the functions in each box from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice. Don't merge the lists together.

To simplify your answers, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

(a)
$$\boxed{2^{\sqrt{\lg n}} \quad 2^{\lg \sqrt{n}} \quad \sqrt{2^{\lg n}} \quad \sqrt{2^{\lg n}} \quad \lg 2^{\sqrt{n}} \quad \lg \sqrt{2^n} \quad \lg \sqrt{2^n} \quad \sqrt{\lg 2^n}}$$

*(b)
$$\boxed{\lg(\sqrt{n}!) \quad \lg(\sqrt{n}!) \quad \sqrt{\lg(n!)} \quad (\lg \sqrt{n})! \quad (\sqrt{\lg n})! \quad \sqrt{(\lg n)!}}$$

[Hint: Use Stirling's approximation for factorials: $n! \approx n^{n+1/2}/e^n$]

2. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Proofs are *not* required; just give us the list of answers. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice. Assume reasonable but nontrivial base cases. If your solution requires specific base cases, state them! Extra credit will be awarded for more exact solutions.

(a) $A(n) = 9A(n/3) + n^2$

(b) $B(n) = 2B(n/2) + n/\lg n$

(c) $C(n) = \frac{2C(n-1)}{C(n-2)}$ [Hint: This is easy!]

(d) $D(n) = D(n-1) + 1/n$

(e) $E(n) = E(n/2) + D(n)$

(f) $F(n) = 2F(\lfloor (n+3)/4 \rfloor - \sqrt{5n \lg n} + 6) + 7\sqrt{n+8} - \lg^9 \lg \lg n + 10^{\lg^* n} - 11/n^{12}$

(g) $G(n) = 3G(n-1) - 3G(n-2) + G(n-3)$

*(h) $H(n) = 4H(n/2) - 4H(n/4) + 1$ [Hint: Careful!]

(i) $I(n) = I(n/3) + I(n/4) + I(n/6) + I(n/8) + I(n/12) + I(n/24) + n$

★(j) $J(n) = \sqrt{n} \cdot J(2\sqrt{n}) + n$

[Hint: First solve the secondary recurrence $j(n) = 1 + j(2\sqrt{n})$.]

3. Scientists have recently discovered a planet, tentatively named “Ygdrasil”, which is inhabited by a bizarre species called “nertices” (singular “nertex”). All nertices trace their ancestry back to a particular nertex named Rudy. Rudy is still quite alive, as is every one of his many descendants. Nertices reproduce asexually, like bees; each nertex has exactly one parent (except Rudy). There are three different types of nertices—red, green, and blue. The color of each nertex is correlated exactly with the number and color of its children, as follows:

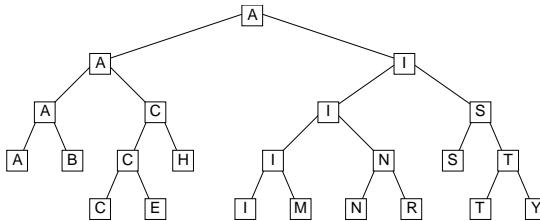
- Each red nertex has two children, exactly one of which is green.
- Each green nertex has exactly one child, which is not green.
- Blue nertices have no children.

In each of the following problems, let R , G , and B respectively denote the number of red, green, and blue nertices on Ygdrasil.

- (a) Prove that $B = R + 1$.
- (b) Prove that either $G = R$ or $G = B$.
- (c) Prove that $G = B$ if and only if Rudy is green.

4. Algorithms and data structures were developed millions of years ago by the Martians, but not quite in the same way as the recent development here on Earth. Intelligent life evolved independently on Mars’ two moons, Phobos and Deimos.¹ When the two races finally met on the surface of Mars, after thousands of years of separate philosophical, cultural, religious, and scientific development, their disagreements over the proper structure of binary search trees led to a bloody (or more accurately, ichorous) war, ultimately leading to the destruction of all Martian life.

A *Phobian* binary search tree is a full binary tree that stores a set X of search keys. The root of the tree stores the *smallest* element in X . If X has more than one element, then the left subtree stores all the elements less than some pivot value p , and the right subtree stores everything else. Both subtrees are *nonempty* Phobian binary search trees. The actual pivot value p is *never* stored in the tree.



A Phobian binary search tree for the set $\{M, A, R, T, I, N, B, Y, S, E, C, H\}$.

- (a) Describe and analyze an algorithm $\text{FIND}(x, T)$ that returns TRUE if x is stored in the Phobian binary search tree T , and FALSE otherwise.
- (b) A *Deimoid* binary search tree is almost exactly the same as its Phobian counterpart, except that the *largest* element is stored at the root, and both subtrees are Deimoid binary search trees. Describe and analyze an algorithm to transform an n -node Phobian binary search tree into a Deimoid binary search tree in $O(n)$ time, using as little additional space as possible.

¹Greek for “fear” and “panic”, respectively. Doesn’t that make you feel better?

5. Penn and Teller agree to play the following game. Penn shuffles a standard deck² of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs ($3\clubsuit$), at which point the remaining undrawn cards instantly burst into flames.

The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.³ To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

- (a) What is the expected number of cards that Teller draws?
- (b) What is the expected *maximum* value among the cards Teller gives to Penn?
- (c) What is the expected *minimum* value among the cards Teller gives to Penn?
- (d) What is the expected number of cards that Teller gives to Penn?

Full credit will be given only for *exact* answers (with correct proofs, of course).

*6. [Extra credit]⁴

Lazy binary is a variant of standard binary notation for representing natural numbers where we allow each “bit” to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

- The lazy binary representation of zero is 0.
- Given the lazy binary representation of any non-negative integer n , we can construct the lazy binary representation of $n + 1$ as follows:
 - (a) increment the rightmost digit;
 - (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

Here are the first several natural numbers in lazy binary notation:

0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, 102101, 102110, ...

- (a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.
- (b) Prove that for any natural number N , the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

²In a standard deck of 52 cards, each card has a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ and a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$, and every possible suit-value pair appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

³Specifically, he hurls them from the opposite side of the stage directly into the back of Penn’s right hand.

⁴The “I don’t know” rule does not apply to extra credit problems. There is no such thing as “partial extra credit”.

CS 373U: Combinatorial Algorithms, Spring 2004

Homework 1

Due Monday, February 9, 2004 at noon

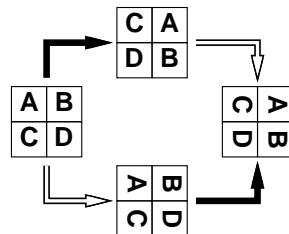
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

-
- For this and all following homeworks, groups of up to three people can turn in a single solution. Please write *all* your names and NetIDs on *every* page you turn in.
-

#	1	2	3	4	5	6*	Total
Score							
Grader							

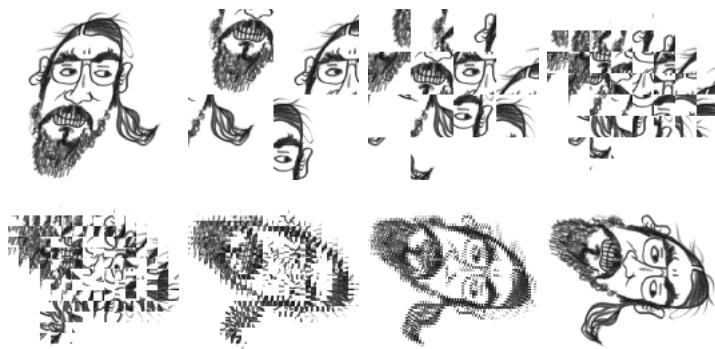
1. Some graphics hardware includes support for an operation called *blit*, or **block transfer**, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixelmap 90° clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
Black arrows indicate blitting the blocks into place.
White arrows indicate recursively rotating the blocks.

The following sequence of pictures shows the first algorithm (blit then recurse) in action.



In the following questions, assume n is a power of two.

- Prove that both versions of the algorithm are correct. [Hint: If you exploit all the available symmetries, your proof will only be a half of a page long.]
- Exactly how many blits does the algorithm perform?
- What is the algorithm's running time if each $k \times k$ blit takes $O(k^2)$ time?
- What if each $k \times k$ blit takes only $O(k)$ time?

2. The traditional Devonian/Cornish drinking song “The Barley Mow” has the following pseudolyrics¹, where $\text{container}[i]$ is the name of a container that holds 2^i ounces of beer.²

BARLEYMOW(n):

“Here’s a health to the barley-mow, my brave boys,”

“Here’s a health to the barley-mow!”

“We’ll drink it out of the jolly brown bowl,”

“Here’s a health to the barley-mow!”

“Here’s a health to the barley-mow, my brave boys,”

“Here’s a health to the barley-mow!”

for $i \leftarrow 1$ to n

 “*We’ll drink it out of the container[i], boys,*”

 “*Here’s a health to the barley-mow!*”

 for $j \leftarrow i$ downto 1

 “*The container[j],*”

 “*And the jolly brown bowl!*”

 “*Here’s a health to the barley-mow!*”

 “*Here’s a health to the barley-mow, my brave boys,*”

 “*Here’s a health to the barley-mow!*”

- (a) Suppose each container name $\text{container}[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)
- (b) If you want to sing this song for $n > 20$, you’ll have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases³. Suppose $\text{container}[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)
- (c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $\text{container}[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BARLEYMOW(n)? (Give an *exact* answer, not just an asymptotic bound.)

¹Pseudolyrics are to lyrics as pseudocode is to code.

²One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 anklers, and the last three units are just silly.

³“We’ll drink it out of the hemisemidemiyottapint, boys!”

3. In each of the problems below, you are given a ‘magic box’ that can solve one problem quickly, and you are asked to construct an algorithm that uses the magic box to solve a different problem.
- 3-Coloring:** A graph is *3-colorable* if it is possible to color each vertex red, green, or blue, so that for every edge, its two vertices have two different colors. Suppose you have a magic box that can tell you whether a given graph is 3-colorable in constant time. Describe an algorithm that constructs a 3-coloring of a given graph (if one exists) as quickly as possible.
 - 3SUM:** The 3SUM problem asks, given a set of integers, whether any three elements sum to zero. Suppose you have a magic box that can solve the 3SUM problem in constant time. Describe an algorithm that actually finds, given a set of integers, three elements that sum to zero (if they exist) as quickly as possible.
 - Traveling Salesman:** A *Hamiltonian cycle* in a graph is a cycle that visits every vertex exactly once. Given a complete graph where every edge has a weight, the *traveling salesman* cycle is the Hamiltonian cycle with minimum total weight; that is, the sum of the weight of the edges is smaller than for any other Hamiltonian cycle. Suppose you have a magic box that can tell you the weight of the traveling salesman cycle of a weighted graph in constant time. Describe an algorithm that actually constructs the traveling salesman cycle of a given weighted graph as quickly as possible.
4. (a) Describe and analyze an algorithm to sort an array $A[1..n]$ by calling a subroutine $\text{SQRTSORT}(k)$, which sorts the subarray $A[k+1..k + \lceil \sqrt{n} \rceil]$ in place, given an arbitrary integer k between 0 and $n - \lceil \sqrt{n} \rceil$ as input. Your algorithm is *only* allowed to inspect or modify the input array by calling SQRTSORT ; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SQRTSORT in the worst case?
- (b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SQRTSORT , prove that no algorithm can sort using $o(f(n))$ calls to SQRTSORT .
- (c) Now suppose SQRTSORT is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k} , so that repeated square roots are always integers.)

5. In a previous incarnation, you worked as a cashier in the lost Antarctic colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource on Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.⁴
- (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.
 - (b) Describe and analyze a recursive algorithm that computes, given an integer k , the minimum number of bills needed to make k Dream Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)
 - (c) Describe a dynamic programming algorithm that computes, given an integer k , the minimum number of bills needed to make k Dream Dollars. (This one needs to be fast.)

- *6. [Extra Credit] A popular puzzle called “Lights Out!”, made by Tiger Electronics, has the following description. The game consists of a 5×5 array of lighted buttons. By pushing any button, you toggle (on to off, off to on) that light and its four (or fewer) immediate neighbors. The goal of the game is to have every light off at the same time.

We generalize this puzzle to a graph problem. We are given an arbitrary graph with a lighted button at every vertex. Pushing the button at a vertex toggles its light and the lights at all of its neighbors in the graph. A *light configuration* is just a description of which lights are on and which are off. We say that a light configuration is *solvable* if it is possible to get from that configuration to the everything-off configuration by pushing buttons. Some (but clearly not all) light configurations are unsolvable.

- (a) Suppose the graph is just a cycle of length n . Give a simple and complete characterization of the solvable light configurations in this case. (What we're really looking for here is a *fast* algorithm to decide whether a given configuration is solvable or not.) [Hint: For which cycle lengths is **every** configuration solvable?]
- *(b) Characterize the set of solvable light configurations when the graph is an arbitrary tree.
- *(c) A *grid graph* is a graph whose vertices are a regular $h \times w$ grid of integer points, with edges between immediate vertical or horizontal neighbors. Characterize the set of solvable light configurations for an arbitrary grid graph. (For example, the original Lights Out puzzle can be modeled as a 5×5 grid graph.)

⁴For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see <http://www.dream-dollars.com>.

CS 373U: Combinatorial Algorithms, Spring 2004

Homework 2

Due Friday, February 20, 2004 at noon
(so you have the whole weekend to study for the midterm)

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

-
- Starting with this homework, we are changing the way we want you to submit solutions. For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
 - Unless specifically stated otherwise, you can use the fact that the following problems are NP-hard to prove that other problems are NP-hard: Circuit-SAT, 3SAT, Vertex Cover, Maximum Clique, Maximum Independent Set, Hamiltonian Path, Hamiltonian Cycle, k -Colorability for any $k \geq 3$, Traveling Salesman Path, Travelling Salesman Cycle, Subset Sum, Partition, 3Partition, Hitting Set, Minimum Steiner Tree, Minesweeper, Tetris, or any other NP-hard problem described in the lecture notes.
 - This homework is a little harder than the last one. You might want to start early.
-

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. In lecture on February 5, Jeff presented the following algorithm to compute the length of the longest increasing subsequence of an n -element array $A[1..n]$ in $O(n^2)$ time.

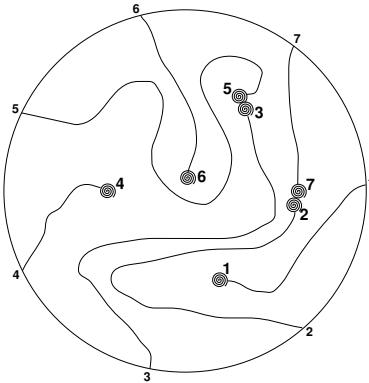
```

LENGTHOFLIS( $A[1..n]$ ):
   $A[n+1] = \infty$ 
  for  $i \leftarrow 1$  to  $n+1$ 
     $L[i] \leftarrow 1$ 
    for  $j \leftarrow 1$  to  $i-1$ 
      if  $A[j] < A[i]$  and  $1 + L[j] < L[i]$ 
         $L[i] \leftarrow 1 + L[j]$ 
  return  $L[n+1] - 1$ 

```

Describe another algorithm for this problem that runs in $O(n \log n)$ time. [Hint: Use a data structure to replace the inner loop with something faster.]

2. Every year, as part of its annual meeting, the Antarctic Snail Lovers of Union Glacier hold a Round Table Mating Race. A large number of high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n . The snails wander around the table, each snail leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail (even their own). When any two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if n is even and the race goes on forever.



The end of an Antarctic SLUG race. Snails 1, 4, and 6 never find a mate.
The organizers must pay $M[3, 5] + M[2, 7]$.

For every pair of snails, the Antarctic SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the $n \times n$ array M as input.

3. Describe and analyze a *polynomial-time* algorithm to determine whether a boolean formula in conjunctive normal form, with exactly *two* literals in each clause, is satisfiable.
4. This problem asks you to prove that four different variants of the minimum spanning tree problem are NP-hard. In each case, the input is a connected undirected graph G with weighted edges. Each problem considers a certain subset of the possible spanning trees of G , and asks you to compute the spanning tree with minimum total weight in that subset.
- Prove that finding the minimum-weight *depth first search* tree is NP-hard. (To remind yourself what depth first search is, and why it computes a spanning tree, see Jeff's introductory notes on graphs or Chapter 22 in CLRS.)
 - Suppose a subset S of the nodes in the input graph are marked. Prove that it is NP-hard to compute the minimum spanning tree whose leaves are all in S . [Hint: First consider the case $|S| = 2$.]
 - Prove that for any integer $\ell \geq 2$, it is NP-hard to compute the minimum spanning tree with exactly ℓ leaves. [Hint: First consider the case $\ell = 2$.]
 - Prove that for any integer $d \geq 2$, it is NP-hard to compute the minimum spanning tree with maximum degree d . [Hint: First consider the case $d = 2$. By now this should start to look familiar.]

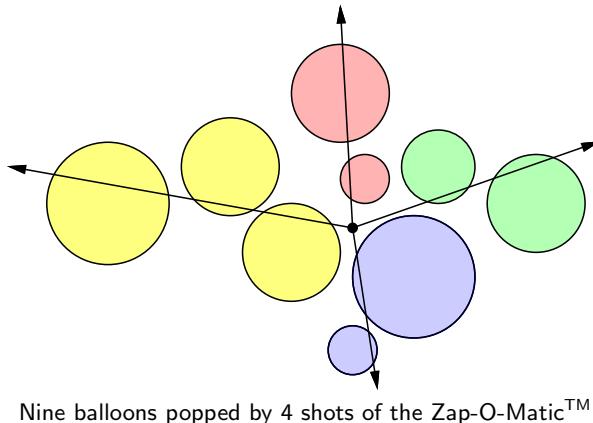
You're welcome to use reductions among these four problems. For example, even if you can't solve part (d), if you can prove that (d) implies (b), you will get full credit for (b). Just don't argue circularly.

5. Consider a machine with a row of n processors numbered 1 through n . A *job* is some computational task that occupies a contiguous set of processors for some amount of time. Each processor can work on only one job at a time. Each job is represented by a pair $J_i = (n_i, t_i)$, where n_i is the number of processors required and t_i is the amount of processing time required to perform the job. A *schedule* for a set of jobs $\{J_1, \dots, J_m\}$ assigns each job J_i to some set of n_i contiguous processors for an interval of t_i seconds, so that no processor works on more than one job at any time. The *make-span* of a schedule is the time from the start to the finish of all jobs.

The *parallel scheduling problem* asks, given a set of jobs as input, to compute a schedule for those jobs with the smallest possible make-span.

- Prove that the parallel scheduling problem is NP-hard.
- Give an algorithm that computes a 3-approximation of the minimum make-span of a set of jobs in $O(m \log m)$ time. That is, if the minimum make-span is M , your algorithm should compute a schedule with make-span at most $3M$. You can assume that n is a power of 2.

- *6. **[Extra credit]** Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic™ to pop all the balloons, without moving from your current location. The Zap-O-Matic™ shoots a high-powered laser beam, which pops all the balloons it hits. Since each shot requires enough energy to power a small country for a year, you want to fire as few shots as possible.



The *minimum zap* problem can be stated more formally as follows. Given a set C of n circles in the plane, each specified by its radius and the (x, y) coordinates of its center, compute the minimum number of rays from the origin that intersect every circle in C . Your goal is to find an efficient algorithm for this problem.

- (a) Describe and analyze a greedy algorithm whose output is within 1 of optimal. That is, if m is the minimum number of rays required to hit every circle in the input, then your greedy algorithm must output either m or $m + 1$. (Of course, you must prove this fact.)
- (b) Describe an algorithm that solves the minimum zap problem in $O(n^2)$ time.
- *(c) Describe an algorithm that solves the minimum zap problem in $O(n \log n)$ time.

Assume you have a subroutine $\text{INTERSECTS}(r, c)$ that determines, in $O(1)$ time, whether a ray r intersects a circle c . It's not that hard to write this subroutine, but it's not the interesting part of the problem.

CS 373U: Combinatorial Algorithms, Spring 2004

Homework 3

Due Friday, March 12, 2004 at noon

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

-
- For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
 - This homework is challenging. You might want to start early.
-

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. Let S be a set of n points in the plane. A point p in S is called *Pareto-optimal* if no other point in S is both above and to the right of p .

- (a) Describe and analyze a deterministic algorithm that computes the Pareto-optimal points in S in $O(n \log n)$ time.
- (b) Suppose each point in S is chosen independently and uniformly at random from the unit square $[0, 1] \times [0, 1]$. What is the *exact* expected number of Pareto-optimal points in S ?

2. Suppose we have an oracle $\text{RANDOM}(k)$ that returns an integer chosen independently and uniformly at random from the set $\{1, \dots, k\}$, where k is the input parameter; RANDOM is our only source of random bits. We wish to write an efficient function $\text{RANDOMPERMUTATION}(n)$ that returns a permutation of the integers $\langle 1, \dots, n \rangle$ chosen uniformly at random.

- (a) Consider the following implementation of RANDOMPERMUTATION .

```
RANDOMPERMUTATION( $n$ ):
    for  $i = 1$  to  $n$ 
         $\pi[i] \leftarrow \text{NULL}$ 
    for  $i = 1$  to  $n$ 
         $j \leftarrow \text{RANDOM}(n)$ 
        while ( $\pi[j] \neq \text{NULL}$ )
             $j \leftarrow \text{RANDOM}(n)$ 
         $\pi[j] \leftarrow i$ 
    return  $\pi$ 
```

Prove that this algorithm is correct. Analyze its expected runtime.

- (b) Consider the following partial implementation of RANDOMPERMUTATION .

```
RANDOMPERMUTATION( $n$ ):
    for  $i = 1$  to  $n$ 
         $A[i] \leftarrow \text{RANDOM}(n)$ 
     $\pi \leftarrow \text{SOMEFUNCTION}(A)$ 
    return  $\pi$ 
```

Prove that if the subroutine SOMEFUNCTION is deterministic, then this algorithm cannot be correct. [Hint: There is a one-line proof.]

- (c) Consider a correct implementation of $\text{RANDOMPERMUTATION}(n)$ with the following property: whenever it calls $\text{RANDOM}(k)$, the argument k is at most m . Prove that this algorithm **always** calls RANDOM at least $\Omega(\frac{n \log n}{\log m})$ times.
- (d) Describe and analyze an implementation of RANDOMPERMUTATION that runs in expected worst-case time $O(n)$.

3. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue containing the empty set.
- **FINDMIN(Q)**: Return the smallest element of Q (if any).
- **DELETEMIN(Q)**: Remove the smallest element in Q (if any).
- **INSERT(Q, x)**: Insert element x into Q , if it is not already there.
- **DECREASEKEY(Q, x, y)**: Replace an element $x \in Q$ with a smaller key y . (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x .
- **DELETE(Q, x)**: Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x .
- **MELD(Q_1, Q_2)**: Return a new priority queue containing all the elements of Q_1 and Q_2 ; this operation destroys Q_1 and Q_2 .

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

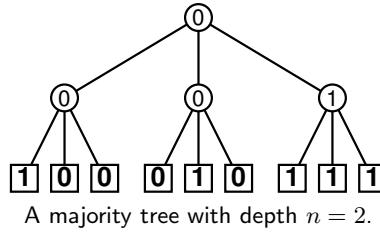
```

MELD( $Q_1, Q_2$ ):
    if  $Q_1$  is empty return  $Q_2$ 
    if  $Q_2$  is empty return  $Q_1$ 
    if  $key(Q_1) > key(Q_2)$ 
        swap  $Q_1 \leftrightarrow Q_2$ 
    with probability  $1/2$ 
         $left(Q_1) \leftarrow \text{MELD}(left(Q_1), Q_2)$ 
    else
         $right(Q_1) \leftarrow \text{MELD}(right(Q_1), Q_2)$ 
    return  $Q_1$ 

```

- (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of $\text{MELD}(Q_1, Q_2)$ is $O(\log n)$, where $n = |Q_1| + |Q_2|$. [Hint: How long is a random root-to-leaf path in an n -node binary tree if each left/right choice is made with equal probability?]
- (b) [Extra credit] Prove that $\text{MELD}(Q_1, Q_2)$ runs in $O(\log n)$ time with high probability.
- (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

4. A *majority tree* is a complete binary tree with depth n , where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled 1, 0, 0, 0, 1, 0, 1, 1, 1, the root has value 0.



- (a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. [Hint: Consider the special case $n = 1$. Recurse.]
- (b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. [Hint: Consider the special case $n = 1$. Recurse.]
5. Suppose n lights labeled $0, \dots, n - 1$ are placed clockwise around a circle. Initially, each light is set to the off position. Consider the following random process.

```

LIGHTTHECIRCLE( $n$ ):
 $k \leftarrow 0$ 
turn on light 0
while at least one light is off
  with probability  $1/2$ 
     $k \leftarrow (k + 1) \bmod n$ 
  else
     $k \leftarrow (k - 1) \bmod n$ 
  if light  $k$  is off, turn it on
  
```

Let $p(i, n)$ be the probability that light i is the last to be turned on by $\text{LIGHTTHECIRCLE}(n, 0)$. For example, $p(0, 2) = 0$ and $p(1, 2) = 1$. Find an exact closed-form expression for $p(i, n)$ in terms of n and i . Prove your answer is correct.

6. **[Extra Credit]** Let G be a *bipartite* graph on n vertices. Each vertex v has an associated set $C(v)$ of $\lg 2n$ colors with which v is compatible. We wish to find a coloring of the vertices in G so that every vertex v is assigned a color from its set $C(v)$ and no edge has the same color at both ends. Describe and analyze a randomized algorithm that computes such a coloring in expected worst-case time $O(n \log^2 n)$. [Hint: For any events A and B , $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$.]

CS 373U: Combinatorial Algorithms, Spring 2004

Homework 4

Due Friday, April 2, 2004 at noon

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

-
- For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
 - As with previous homeworks, we strongly encourage you to begin early.
-

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (like CLRS does); there is a much easier solution.

2. Remember the difference between stacks and queues? Good.

- (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard subroutines PUSH and POP.
- (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:
 - **Push:** add a new item to the left end of the list;
 - **Pop:** remove the item on the left end of the list;
 - **Pull:** remove the item on the right end of the list.

Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any push, pop, or pull operation is $O(1)$. Again, you are *only* allowed to access the stacks through the standard functions PUSH and POP.

3. Some applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Maintaining these secondary structures usually complicates algorithms for keeping the top-level search tree balanced.

Suppose we have a binary search tree T where every node v stores a secondary structure of size $O(|v|)$, where $|v|$ denotes the number of descendants of v in T . Performing a rotation at a node v in T now requires $O(|v|)$ time, because we have to rebuild one of the secondary structures.

- (a) [1 pt] Overall, how much space does this data structure use in the worst case?
- (b) [1 pt] How much space does this structure use if the top-level search tree T is balanced?
- (c) [2 pt] Suppose T is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. [*Hint: This is easy!*]
- (d) [3 pts] Now suppose T is a scapegoat tree, and that rebuilding the subtree rooted at v requires $\Theta(|v| \log |v|)$ time (because we also have to rebuild all the secondary structures). What is the *amortized* cost of inserting a new element into T ?
- (e) [3 pts] Finally, suppose T is a treap. What's the worst-case *expected* time for inserting a new element into T ?

4. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

Describe an algorithm to purge an arbitrary n -node dirty binary search tree in $O(n)$ time, using only $O(\log n)$ additional memory. For 5 points extra credit, reduce the additional memory requirement to $O(1)$ *without repeating an old CS373 homework solution*.¹

5. This problem considers a variant of the lazy binary notation introduced in the extra credit problem from Homework 0. In a *doubly lazy binary number*, each bit can take one of *four* values: -1 , 0 , 1 , or 2 . The only legal representation for zero is 0 . To increment, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the least significant bit, and then borrow the rightmost -1 (if any).

LAZYINCREMENT($B[0..n]$):

```

 $B[0] \leftarrow B[0] + 1$ 
for  $i \leftarrow 1$  to  $n - 1$ 
  if  $B[i] = 2$ 
     $B[i] \leftarrow 0$ 
     $B[i + 1] \leftarrow B[i + 1] + 1$ 
return
  
```

LAZYDECREMENT($B[0..n]$):

```

 $B[0] \leftarrow B[0] - 1$ 
for  $i \leftarrow 1$  to  $n - 1$ 
  if  $B[i] = -1$ 
     $B[i] \leftarrow 1$ 
     $B[i + 1] \leftarrow B[i + 1] - 1$ 
return
  
```

For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit (*i.e.*, $B[0]$) on the right. For succinctness, we write \pm instead of -1 and omit any leading 0 's.

```

0  $\xrightarrow{++}$  1  $\xrightarrow{++}$  10  $\xrightarrow{++}$  11  $\xrightarrow{++}$  20  $\xrightarrow{++}$  101  $\xrightarrow{++}$  110  $\xrightarrow{++}$  111  $\xrightarrow{++}$  120  $\xrightarrow{++}$  201  $\xrightarrow{++}$  210
 $\xrightarrow{++}$  1011  $\xrightarrow{++}$  1020  $\xrightarrow{++}$  1101  $\xrightarrow{++}$  1110  $\xrightarrow{++}$  1111  $\xrightarrow{++}$  1120  $\xrightarrow{++}$  1201  $\xrightarrow{++}$  1210  $\xrightarrow{++}$  2011  $\xrightarrow{++}$  2020
 $\xrightarrow{--}$  2011  $\xrightarrow{--}$  2010  $\xrightarrow{--}$  2001  $\xrightarrow{--}$  2000  $\xrightarrow{--}$  2011  $\xrightarrow{--}$  2110  $\xrightarrow{--}$  2101  $\xrightarrow{--}$  1100  $\xrightarrow{--}$  1111  $\xrightarrow{--}$  1010
 $\xrightarrow{--}$  1001  $\xrightarrow{--}$  1000  $\xrightarrow{--}$  1011  $\xrightarrow{--}$  1110  $\xrightarrow{--}$  1101  $\xrightarrow{--}$  1010  $\xrightarrow{--}$  100  $\xrightarrow{--}$  111  $\xrightarrow{--}$  10  $\xrightarrow{--}$  1  $\xrightarrow{--}$  0
  
```

Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0 , the amortized time for each operation is $O(1)$. Do *not* assume, as in the example above, that all the increments come before all the decrements.

¹That was for a slightly different problem anyway.

6. [Extra credit] My wife is teaching a class² where students work on homeworks in groups of exactly three people, subject to the following rule: *No two students may work together on more than one homework.* At the beginning of the semester, it was easy to find homework groups, but as the course progresses, it is becoming harder and harder to find a legal grouping. Finally, in despair, she decides to ask a computer scientist to write a program to find the groups for her.

- (a) We can formalize this homework-group-assignment problem as follows. The input is a graph, where the vertices are the n students, and two students are joined by an edge if they have not yet worked together. Every node in this graph has the same degree; specifically, if there have been k homeworks so far, each student is connected to exactly $n - 1 - 2k$ other students. The goal is to find $n/3$ disjoint triangles in the graph, or conclude that no such triangles exist. Prove (or disprove!) that this problem is NP-hard.
- (b) Suppose my wife had planned ahead and assigned groups for every homework at the beginning of the semester. How many homeworks can she assign, as a function of n , without violating the no-one-works-together-twice rule? Prove the best upper and lower bounds you can. To prove the upper bound, describe an algorithm that actually assigns the groups for each homework.

²Math 302: Non-Euclidean Geometry. Problem 1 from last week's homework assignment: "Invert Mr. Happy."

CS 373U: Combinatorial Algorithms, Spring 2004

Homework 5

Due Wednesday, April 28, 2004 at noon

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

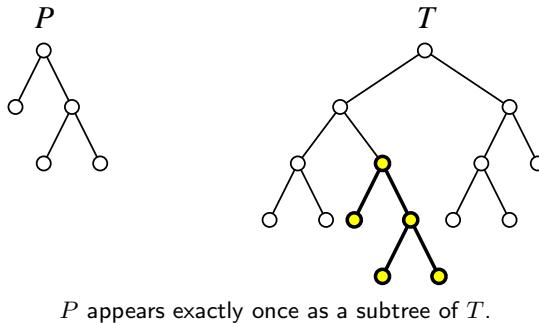
-
- For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
 - As with previous homeworks, we strongly encourage you to begin early.
 - This will be the last graded homework.
-

#	1	2	3	4	5	Total
Score						
Grader						

1. (a) Prove that every graph with the same number of vertices and edges has a cycle.
(b) Prove that every graph with exactly two fewer edges than vertices is disconnected.

Both proofs should be entirely self-contained. In particular, they should not use the word “tree” or any properties of trees that you saw in CS 225 or CS 273.

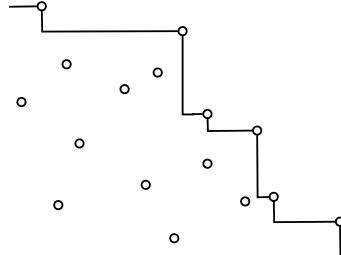
2. A *palindrome* is a string of characters that is exactly the same as its reversal, like X, FOOF, RADAR, or AMANAPLANACATACANALPANAMA.
 - (a) Describe and analyze an algorithm to compute the longest *prefix* of a given string that is a palindrome. For example, the longest palindrome prefix of RADARDETECTAR is RADAR, and the longest palindrome prefix of ALGORITHMSHOMEWORK is the single letter A.
 - (b) Describe and analyze an algorithm to compute a longest *subsequence* of a given string that is a palindrome. For example, the longest palindrome subsequence of RADARDETECTAR is RAETEAR (or RADADAR or RADRDAR or RATETAR or RATCTAR), and the longest palindrome subsequence of ALGORITHMSHOMEWORK is OMOMO (or RMHMR or OHSHO or...).
3. Describe and analyze an algorithm that decides, given two binary trees P and T , whether T is a subtree of P . There is no actual *data* stored in the nodes—these are not binary *search* trees or binary *heaps*. You are only trying to match the *shape* of the trees.



4. Describe and analyze an algorithm that computes the *second* smallest spanning tree of a given connected, undirected, edge-weighted graph.
5. Show that if the input graph is allowed to have negative edges (but no negative cycles), Dijkstra's algorithm¹ runs in exponential time in the worst case. Specifically, describe how to construct, for every integer n , a weighted directed graph G_n without negative cycles that forces Dijkstra's algorithm to perform $\Omega(2^n)$ relaxation steps. Give your description in the form of an algorithm! [Hint: Towers of Hanoi.]

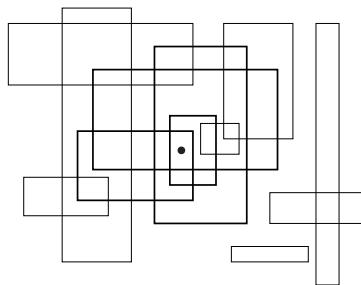
¹This refers to the version of Dijkstra's algorithm described in Jeff's lecture notes. The version in CLRS is always fast, but sometimes gives incorrect results for graphs with negative edges.

1. Let P be a set of n points in the plane. Recall that a point $p \in P$ is *Pareto-optimal* if no other point is both above and to the right of p . Intuitively, the sorted sequence of Pareto-optimal points describes a *staircase* with all the points in P below and to the left. Your task is to describe some algorithms that compute this staircase.



The staircase of a set of points

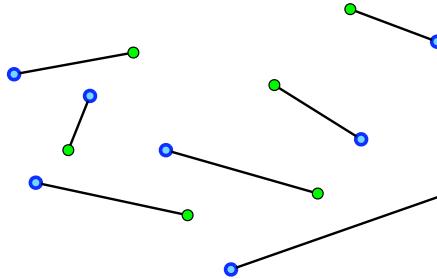
- (a) Describe an algorithm to compute the staircase of P in $O(nh)$ time, where h is the number of Pareto-optimal points.
 - (b) Describe a divide-and-conquer algorithm to compute the staircase of P in $O(n \log n)$ time. [Hint: I know of at least two different ways to do this.]
 - *(c) Describe an algorithm to compute the staircase of P in $O(n \log h)$ time, where h is the number of Pareto-optimal points. [Hint: I know of at least two different ways to do this.]
 - (d) Finally, suppose the points in P are already given in sorted order from left to right. Describe an algorithm to compute the staircase of P in $O(n)$ time. [Hint: I know of at least two different ways to do this.]
2. Let R be a set of n rectangles in the plane.
- (a) Describe and analyze a plane sweep algorithm to decide, in $O(n \log n)$ time, whether any two rectangles in R intersect.
 - *(b) The *depth* of a point is the number of rectangles in R that contain that point. The *maximum depth* of R is the maximum, over all points p in the plane, of the depth of p . Describe a plane sweep algorithm to compute the maximum depth of R in $O(n \log n)$ time.



A point with depth 4 in a set of rectangles.

- (c) Describe and analyze a polynomial-time reduction from the maximum depth problem in part (b) to MAXCLIQUE: Given a graph G , how large is the largest clique in G ?
- (d) MAXCLIQUE is NP-hard. So does your reduction imply that P=NP? Why or why not?

3. Let G be a set of n green points, called “Ghosts”, and let B be a set of n blue points, called “ghostBusters”, so that no three points lie on a common line. Each Ghostbuster has a gun that shoots a stream of particles in a straight line until it hits a ghost. The Ghostbusters want to kill all of the ghosts at once, by having each Ghostbuster shoot a different ghost. It is **very important** that the streams do not cross.



A non-crossing matching between 7 ghosts and 7 Ghostbusters

- (a) Prove that the Ghostbusters can succeed. More formally, prove that there is a collection of n non-intersecting line segments, each joining one point in G to one point in B . [Hint: Think about the set of joining segments with minimum total length.]
- (b) Describe and analyze an algorithm to find a line ℓ that passes through one ghost and one Ghostbuster, so that same number of ghosts as Ghostbusters are above ℓ .
- *(c) Describe and analyze an algorithm to find a line ℓ such that exactly half the ghosts and exactly half the Ghostbusters are above ℓ . (Assume n is even.)
- (d) Using your algorithm for part (b) or part (c) as a subroutine, describe and analyze an algorithm to find the line segments described in part (a). (Assume n is a power of two if necessary.)

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"?

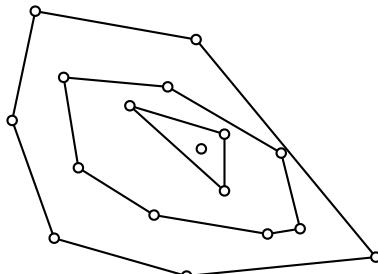
Spengler: Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light.

Stantz: Total protonic reversal!

Venkman: That's bad. Okay. Alright, important safety tip, thanks Egon.

— Dr. Egon Spengler (Harold Ramis), Dr. Peter Venkman (Bill Murray),
and Dr. Raymond Stanz (Dan Aykroyd), *Ghostbusters*, 1984

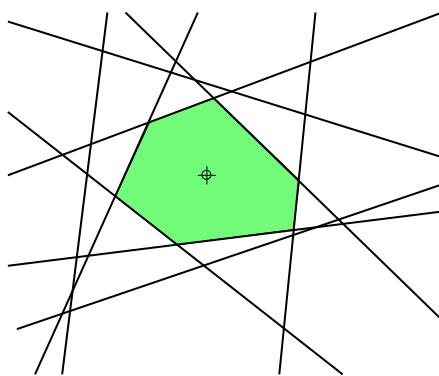
4. The *convex layers* of a point set P consist of a series of nested convex polygons. The convex layers of the empty set are empty. Otherwise, the first layer is just the convex hull of P , and the remaining layers are the convex layers of the points that are not on the convex hull of P .



The convex layers of a set of points.

Describe and analyze an efficient algorithm to compute the convex layers of a given n -point set. For full credit, your algorithm should run in $O(n^2)$ time.

5. Suppose we are given a set of n lines in the plane, where none of the lines passes through the origin $(0, 0)$ and at most two lines intersect at any point. These lines divide the plane into several convex polygonal regions, or *cells*. Describe and analyze an efficient algorithm to compute the cell containing the origin. The output should be a doubly-linked list of the cell's vertices. [Hint: There are literally dozens of solutions. One solution is to reduce this problem to the convex hull problem. Every other solution looks like a convex hull algorithm.]



The cell containing the origin in an arrangement of lines.

Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$ X: I don't know.

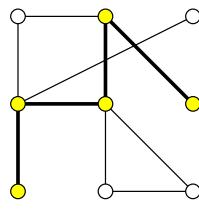
For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point. Each X earns you $\frac{1}{4}$ point. **Each incorrect answer costs you $\frac{1}{2}$ point.** Your total score will be rounded **down** to an integer. Negative scores will be rounded up to zero.

- (a) What is $\sum_{i=1}^n \lg i$?
- (b) What is $\sum_{i=1}^{\lg n} i 2^i$?
- (c) How many decimal digits are required to write the n th Fibonacci number?
- (d) What is the solution of the recurrence $T(n) = 4T(n/8) + n \log n$?
- (e) What is the solution of the recurrence $T(n) = T(n - 3) + \frac{5}{n}$?
- (f) What is the solution of the recurrence $T(n) = 5T\left(\lceil \frac{n+13}{3} \rceil + \lfloor \sqrt{n} \rfloor\right) + (10n - 7)^2 - \frac{\lg^3 n}{\lg \lg n}$?
- (g) How long does it take to construct a Huffman code, given an array of n character frequencies as input?
- (h) How long does it take to sort an array of size n using quicksort?
- (i) Given an unsorted array $A[1..n]$, how long does it take to construct a binary search tree for the elements of A ?
- (j) A train leaves Chicago at 8:00pm and travels south at 75 miles per hour. Another train leaves New Orleans at 1:00pm and travels north at 60 miles per hour. The conductors of both trains are playing a game of chess over the phone. After each player moves, the other player must move before his train has traveled five miles. How many moves do the two players make before their trains pass each other (somewhere near Memphis)?

2. Describe and analyze efficient algorithms to solve the following problems:

- (a) Given a set of n integers, does it contain a pair of elements a, b such that $a + b = 0$?
- (b) Given a set of n integers, does it contain three elements a, b, c such that $a + b = c$?

3. A *tonian path* in a graph G is a simple path in G that visits more than half of the vertices of G . (Intuitively, a tonian path is “most of a Hamiltonian path”.) Prove that it is NP-hard to determine whether or not a given graph contains a tonian path.



A tonian path in a 9-vertex graph.

4. *Vankin's Mile* is a solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

For example, given the grid below, the player can score $8 - 6 + 7 - 3 + 4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)

-1	7	-8	10	-5
-4	-9	8	-6	0
5	-2	-6	-6	7
-7	4	7 \Rightarrow -3	-3	-3
7	1	-6	4	-9

Describe and analyze an algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

5. Suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k . Describe an algorithm to find the k th smallest element in the union of A and B in $\Theta(\log(m+n))$ time. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \quad B[1..5] = [2, 5, 8, 17, 19] \quad k = 6$$

your algorithm should return 8. You can assume that the arrays contain no duplicates. [Hint: What can you learn from comparing one element of A to one element of B ?]

Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$ X: I don't know.

For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point. Each X earns you $\frac{1}{4}$ point. **Each incorrect answer costs you $\frac{1}{2}$ point.** Your total score will be rounded **down** to an integer. Negative scores will be rounded up to zero.

- What is $\sum_{i=1}^n \frac{n}{i}$?
- What is $\sum_{i=1}^{\lg n} 4^i$?
- How many bits are required to write $n!$ in binary?
- What is the solution of the recurrence $T(n) = 4T(n/2) + n \log n$?
- What is the solution of the recurrence $T(n) = T(n - 3) + \frac{5}{n}$?
- What is the solution of the recurrence $T(n) = 9T\left(\lceil \frac{n+13}{3} \rceil\right) + 10n - 7\sqrt{n} - \frac{\lg^3 n}{\lg \lg n}$?
- How long does it search for a value in an n -node binary search tree?
- Given a sorted array $A[1..n]$, how long does it take to construct a binary search tree for the elements of A ?
- How long does it take to construct a Huffman code, given an array of n character frequencies as input?
- A train leaves Chicago at 8:00pm and travels south at 75 miles per hour. Another train leaves New Orleans at 1:00pm and travels north at 60 miles per hour. The conductors of both trains are playing a game of chess over the phone. After each player moves, the other player must move before his train has traveled five miles. How many moves do the two players make before their trains pass each other (somewhere near Memphis)?

2. Describe and analyze an algorithm to find the length of the longest substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples:

- Given the input string ALGORITHM, your algorithm should return 0.
- Given the input string RECURSION, your algorithm should return 1, for the substring R.
- Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
- Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM.

For full credit, your algorithm should run in $O(n^2)$ time.

3. The *median* of a set of size n is its $\lceil n/2 \rceil$ th largest element, that is, the element that is as close as possible to the middle of the set in sorted order. It's quite easy to find the median of a set in $O(n \log n)$ time—just sort the set and look in the middle—but you (correctly!) think that you can do better.

During your lifelong quest for a faster median-finding algorithm, you meet and befriend the Near-Middle Fairy. Given any set X , the Near-Middle Fairy can find an element $m \in X$ that is *near* the middle of X in $O(1)$ time. Specifically, at least a third of the elements of X are smaller than m , and at least a third of the elements of X are larger than m .

Describe and analyze an algorithm to find the median of a set in $O(n)$ time if you are allowed to ask the Near-Middle Fairy for help. [Hint: You may need the PARTITION subroutine from QUICKSORT.]

4. SUBSETSUM and PARTITION are two closely related NP-hard problems.

- SUBSETSUM: Given a set X of integers and an integer k , does X have a subset whose elements sum up to k ?
- PARTITION: Given a set X of integers and an integer k , can X be partitioned into two subsets whose sums are equal?
 - (a) Describe and analyze a polynomial-time reduction from SUBSETSUM to PARTITION.
 - (b) Describe and analyze a polynomial-time reduction from PARTITION to SUBSETSUM.

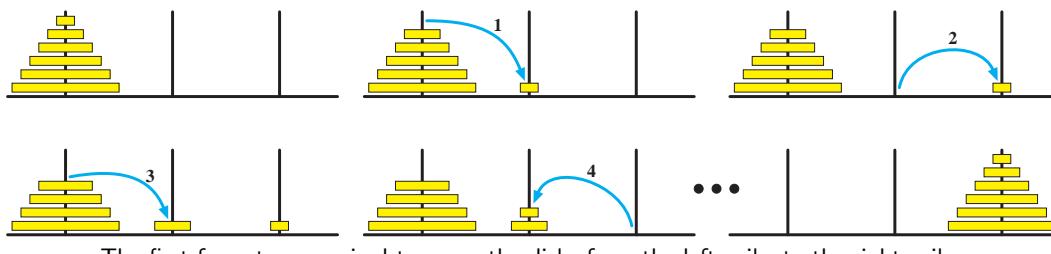
5. Describe and analyze efficient algorithms to solve the following problems:

- (a) Given a set of n integers, does it contain a pair of elements a, b such that $a + b = 0$?
- (b) Given a set of n integers, does it contain three elements a, b, c such that $a + b + c = 0$?

Write your answers in the separate answer booklet.

1. In the well-known *Tower of Hanoi* problem, we have three spikes, one of which has a tower of n disks of different sizes, stacked with smaller disks on top of larger ones. In a single step, we are allowed to take the top disk on any spike and move it to the top of another spike. We are never allowed to place a larger disk on top of a smaller one. Our goal is to move all the disks from one spike to another.

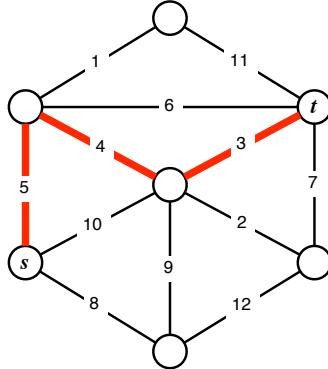
Hmmm.... You've probably known how to solve this problem since CS 125, so make it more interesting, let's add another constraint: The three spikes are arranged in a row, and we are also forbidden to move a disk directly from the left spike to the right spike or vice versa. In other words, we *must* move a disk either to or from the middle spike at *every* step.



The first four steps required to move the disks from the left spike to the right spike.

- (a) [4 pts] Describe an algorithm that moves the stack of n disks from the left needle to the right needle in as few steps as possible.
- (b) [6 pts] Exactly how many steps does your algorithm take to move all n disks? A correct Θ -bound is worth 3 points. [Hint: Set up and solve a recurrence.]
-
2. Consider a random walk on a path with vertices numbered $1, 2, \dots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n . In Midterm 2, you were asked to prove that if we start at vertex 1, the probability that the walk ends by falling off the *left* end of the path is exactly $n/(n+1)$.
- (a) [6 pts] Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n . [Hint: Set up and solve a recurrence. Use the result from Midterm 2.]
- (b) [4 pts] Suppose we start at vertex $n/2$ instead. State a tight Θ -bound on the expected length of the random walk in this case. **No proof is required.** [Hint: Set up and solve a recurrence. Use part (a), even if you can't prove it.]
-
3. Prove that any connected acyclic graph with n vertices has exactly $n - 1$ edges. Do not use the word "tree" or any well-known properties of trees; your proof should follow entirely from the definitions.

4. Consider a path between two vertices s and t in an undirected weighted graph G . The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t . (If there are no paths from s to t , the bottleneck distance between s and t is ∞ .)



The bottleneck distance between s and t is 5.

Describe and analyze an algorithm to compute the bottleneck distance between every pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

5. The 5COLOR asks, given a graph G , whether the vertices of a graph G can be colored with five colors so that no edge has two endpoints with the same color. You already know from class that this problem is NP-complete.

Now consider the related problem 5COLOR ± 1 : Given a graph G , can we color each vertex with an integer from the set $\{0, 1, 2, 3, 4\}$, so that for every edge, the colors of the two endpoints differ by exactly 1 modulo 5? (For example, a vertex with color 4 can only be adjacent to vertices colored 0 or 3.) We would like to show that 5COLOR ± 1 is NP-complete as well.

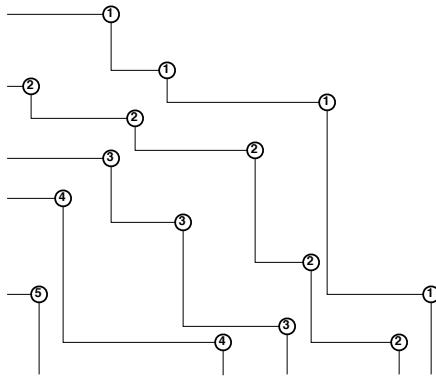
- (a) [2 pts] Show that 5COLOR ± 1 is in NP.
- (b) [1 pt] To prove that 5COLOR ± 1 is NP-hard (and therefore NP-complete), we must describe a polynomial time algorithm for *one* of the following problems. Which one?
 - Given an arbitrary graph G , compute a graph H such that 5COLOR(G) is true if and only if 5COLOR ± 1 (H) is true.
 - Given an arbitrary graph G , compute a graph H such that 5COLOR ± 1 (G) is true if and only if 5COLOR(H) is true.

- (c) [1 pt] Explain briefly why the following argument is not correct.

For any graph G , if 5COLOR ± 1 (G) is true, then 5COLOR(G) is true (using the same coloring). Therefore if we could solve 5COLOR ± 1 quickly, we could also solve 5COLOR quickly. In other words, 5COLOR ± 1 is at least as hard as 5COLOR. We know that 5COLOR is NP-hard, so 5COLOR ± 1 must also be NP-hard!

- (d) [6 pts] Prove that 5COLOR ± 1 is NP-hard. [Hint: Look at some small examples. Replace the edges of G with a simple gadget, so that the resulting graph H has the desired property from part (b).]

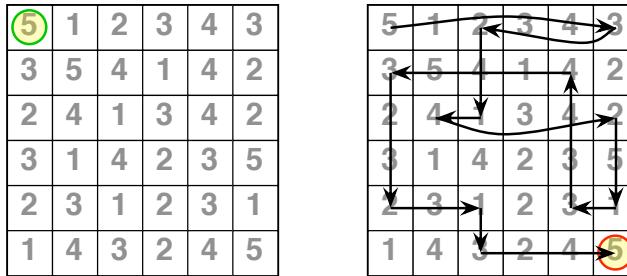
6. Let P be a set of points in the plane. Recall that a point $p \in P$ is *Pareto-optimal* if no other points in P are both above and to the right of p . Intuitively, the sequence of Pareto-optimal points forms a *staircase* with all the other points in P below and to the left. The *staircase layers* of P are defined recursively as follows. The empty set has no staircase layers. Otherwise, the first staircase layer contains all the Pareto-optimal points in P , and the remaining layers are the staircase layers of P minus the first layer.



A set of points with 5 staircase layers

Describe and analyze an algorithm to compute the number of staircase layers of a point set P as quickly as possible. For example, given the points illustrated above, your algorithm would return the number 5.

7. Consider the following puzzle played on an $n \times n$ square grid, where each square is labeled with a positive integer. A token is placed on one of the squares. At each turn, you may move the token left, right, up, or down; the distance you move the token must be equal to the number on the current square. For example, if the token is on a square labeled "3", you are allowed more the token three squares down, three square left, three squares up, or three squares right. You are never allowed to move the token off the board.



A sequence of legal moves from the top left corner to the bottom right corner.

- (a) [4 pts] Describe and analyze an algorithm to determine, given an $n \times n$ array of labels and two squares s and t , whether there is a sequence of legal moves that takes the token from s to t .
- (b) [6 pts] Suppose you are only given the $n \times n$ array of labels. Describe how to preprocess these values, so that afterwards, given any two squares s and t , you can determine in $O(1)$ time whether there is a sequence of legal moves from s to t .

Answer four of these seven problems; the lowest three scores will be dropped.

1. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are five local minima in the following array:

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. [Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]

2. Consider a random walk on a path with vertices numbered $1, 2, \dots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n . In Midterm 2, you were asked to prove that if we start at vertex 1, the probability that the walk ends by falling off the *left* end of the path is exactly $n/(n+1)$.
- (a) [6 pts] **Prove** that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n . [Hint: Set up and solve a recurrence. Use the result from Midterm 2.]
 - (b) [4 pts] Suppose we start at vertex $n/2$ instead. State and **prove** a tight Θ -bound on the expected length of the random walk in this case. [Hint: Set up and solve a recurrence. Use part (a), even if you can't prove it.]
3. **Prove** that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words “tree” or “leaf”, or any well-known properties of trees; your proof should follow entirely from the definitions.
4. Consider the following sketch of a “reverse greedy” algorithm. The input is a connected undirected graph G with weighted edges, represented by an adjacency list.

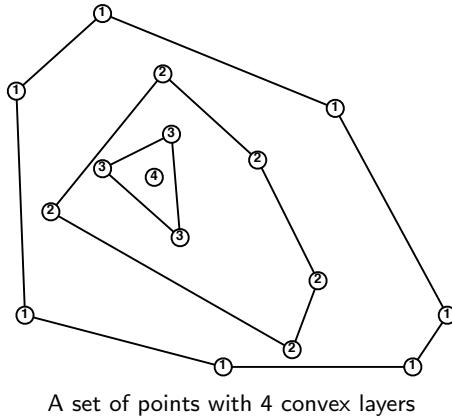
$\text{REVERSEGREEDYMST}(G):$ sort the edges E of G by weight for $i \leftarrow 1$ to $ E $ $e \leftarrow$ i th heaviest edge in E if $G \setminus e$ is connected \quad remove e from G

- (a) [4 pts] What is the worst-case running time of this algorithm? (Answering this question will require fleshing out a few details.)
- (b) [6 pts] **Prove** that the algorithm transforms G into its minimum spanning tree.

5. SUBSETSUM and PARTITION are two closely related NP-hard problems.

- SUBSETSUM: Given a set X of integers and an integer k , does X have a subset whose elements sum up to k ?
- PARTITION: Given a set X of integers, can X be partitioned into two subsets whose sums are equal?
 - [2 pts] Prove that PARTITION and SUBSETSUM are both in NP.
 - [1 pt] Suppose we knew that SUBSETSUM is NP-hard, and we wanted to prove that PARTITION is NP-hard. Which of the following arguments should we use?
 - Given a set X and an integer k , compute a set Y such that $\text{PARTITION}(Y)$ is true if and only if $\text{SUBSETSUM}(X, k)$ is true.
 - Given a set X , construct a set Y and an integer k such that $\text{PARTITION}(X)$ is true if and only if $\text{SUBSETSUM}(Y, k)$ is true.
 - [3 pts] Describe and analyze a polynomial-time reduction from PARTITION to SUBSETSUM. (See part (b).)
 - [4 pts] Describe and analyze a polynomial-time reduction from SUBSETSUM to PARTITION. (See part (b).)

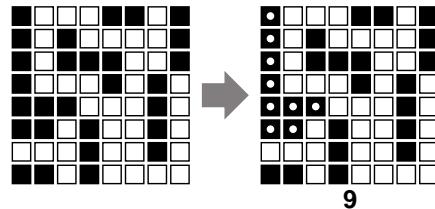
6. Let P be a set of points in the plane. The *convex layers* of P are defined recursively as follows. If P is empty, it has no convex layers. Otherwise, the first convex layer is the convex hull of P , and the remaining convex layers are the convex layers of P minus its convex hull.



Describe and analyze an algorithm to compute the number of convex layers of a point set P as quickly as possible. For example, given the points illustrated above, your algorithm would return the number 4.

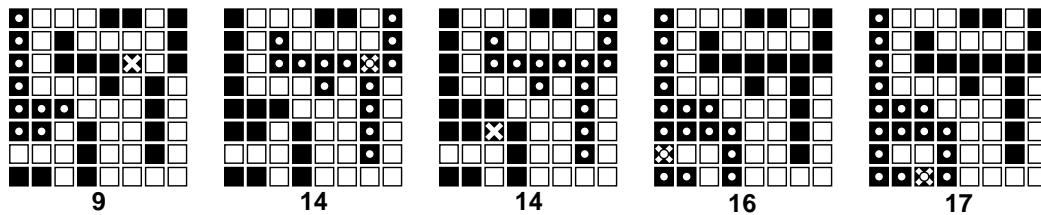
7. (a) [4 pts] Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

For example, given the bitmap below as input, your algorithm should return the number 9, because the largest connected black component (marked with white dots on the right) contains nine pixels.



- (b) [4 pts] Design and analyze an algorithm $\text{BLACKEN}(i, j)$ that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.



- (c) [2 pts] What is the *worst-case* running time of your BLACKEN algorithm?

Write your answers in the separate answer booklet.

1. A *data stream* is an extremely long sequence of items that you can only read only once, in order. A good example of a data stream is the sequence of packets that pass through a router. Data stream algorithms must process each item in the stream quickly, using very little memory; there is simply too much data to store, and it arrives too quickly for any complex computations. Every data stream algorithm looks roughly like this:

```
DOsomethingINTERESTING(stream  $S$ ):
repeat
     $x \leftarrow$  next item in  $S$ 
     $\langle\!\langle$  do something fast with  $x$   $\rangle\!\rangle$ 
until  $S$  ends
return  $\langle\!\langle$  something  $\rangle\!\rangle$ 
```

Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself). Assume you have a subroutine $\text{RANDOM}(n)$ that returns a random integer between 1 and n , each with equal probability, given any integer n as input.

2. Consider a random walk on a path with vertices numbered $1, 2, \dots, n$ from left to right. We start at vertex 1. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n .

Prove that the probability that the walk ends by falling off the *left* end of the path is exactly $n/(n+1)$. [Hint: Set up a recurrence and verify that $n/(n+1)$ satisfies it.]

3. Consider the following algorithms for maintaining a family of disjoint sets. The **UNION** algorithm uses a heuristic called *union by size*.

MAKESET(x):
$\text{parent}(x) \leftarrow x$
$\text{size}(x) \leftarrow 1$

FIND(x):
while $x \neq \text{parent}(x)$
$x \leftarrow \text{parent}(x)$
return x

UNION(x, y):
$\bar{x} \leftarrow \text{FIND}(x)$
$\bar{y} \leftarrow \text{FIND}(y)$
if $\text{size}(\bar{x}) < \text{size}(\bar{y})$
$\text{parent}(\bar{x}) \leftarrow \bar{y}$
$\text{size}(\bar{x}) \leftarrow \text{size}(\bar{x}) + \text{size}(\bar{y})$
else
$\text{parent}(\bar{y}) \leftarrow \bar{x}$
$\text{size}(\bar{y}) \leftarrow \text{size}(\bar{x}) + \text{size}(\bar{y})$

Prove that if we use union by size, $\text{FIND}(x)$ runs in $O(\log n)$ time *in the worst case*, where n is the size of the set containing element x .

4. Recall the SUBSETSUM problem: Given a set X of integers and an integer k , does X have a subset whose elements sum to k ?

- (a) [7 pts] Describe and analyze an algorithm that solves SUBSETSUM in time $O(nk)$.
- (b) [3 pts] SUBSETSUM is NP-hard. Does part (a) imply that P=NP? Justify your answer.

5. Suppose we want to maintain a set X of numbers under the following operations:

- $\text{INSERT}(x)$: Add x to the set X .
- $\text{PRINTANDDELETEBETWEEN}(a, z)$: Print every element $x \in X$ such that $a \leq x \leq z$, in order from smallest to largest, and then delete those elements from X .

For example, $\text{PRINTANDDELETEBETWEEN}(-\infty, \infty)$ prints all the elements of X in sorted order and then deletes everything.

- (a) [6 pts] Describe and analyze a data structure that supports these two operations, each in $O(\log n)$ *amortized* time, where n is the maximum number of elements in X .
- (b) [2 pts] What is the running time of your INSERT algorithm in the worst case?
- (c) [2 pts] What is the running time of your $\text{PRINTANDDELETEBETWEEN}$ algorithm in the worst case?

CS 473G: Combinatorial Algorithms, Fall 2005

Homework 0

Due Thursday, September 1, 2005, at the beginning of class (12:30pm CDT)

Name:	
Net ID:	Alias:



I understand the Homework Instructions and FAQ.

-
- Neatly print your full name, your NetID, and an alias of your choice in the boxes above. Grades will be listed on the course web site by alias. Please write the same alias on every homework and exam! For privacy reasons, your alias should not resemble your name or NetID. By providing an alias, you agree to let us list your grades; if you do not provide an alias, your grades will not be listed. ***Never give us your Social Security number!***
 - Read the “Homework Instructions and FAQ” on the course web page, and then check the box above. This page describes what we expect in your homework solutions—start each numbered problem on a new sheet of paper, write your name and NetID on every page, don’t turn in source code, analyze and prove everything, use good English and good logic, and so on—as well as policies on grading standards, regrading, and plagiarism. **See especially the course policies regarding the magic phrases “I don’t know” and “and so on”.** If you have *any* questions, post them to the course newsgroup or ask during lecture.
 - Don’t forget to submit this cover sheet with the rest of your homework solutions.
 - This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, discrete probability, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Chapters 1–10 of CLRS should be sufficient review, but you may also want consult your discrete mathematics and data structures textbooks.
 - Every homework will have five required problems. Most homeworks will also include one extra-credit problem and several practice (no-credit) problems. Each numbered problem is worth 10 points.
-

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice. Assume reasonable but nontrivial base cases. **If your solution requires specific base cases, state them!**

(a) $A(n) = 2A(n/4) + \sqrt{n}$

(b) $B(n) = \max_{n/3 < k < 2n/3} (B(k) + B(n - k) + n)$

(c) $C(n) = 3C(n/3) + n/\lg n$

(d) $D(n) = 3D(n - 1) - 3D(n - 2) + D(n - 3)$

(e) $E(n) = \frac{E(n - 1)}{3E(n - 2)}$ [Hint: This is easy!]

(f) $F(n) = F(n - 2) + 2/n$

(g) $G(n) = 2G(\lceil(n + 3)/4\rceil - 5n/\sqrt{\lg n} + 6\lg\lg n) + 7\sqrt[8]{n - 9} - \lg^{10} n/\lg\lg n + 11^{\lg^* n} - 12$

*(h) $H(n) = 4H(n/2) - 4H(n/4) + 1$ [Hint: Careful!]

(i) $I(n) = I(n/2) + I(n/4) + I(n/8) + I(n/12) + I(n/24) + n$

★(j) $J(n) = 2\sqrt{n} \cdot J(\sqrt{n}) + n$
[Hint: First solve the secondary recurrence $j(n) = 1 + j(\sqrt{n})$.]

2. Penn and Teller agree to play the following game. Penn shuffles a standard deck¹ of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs ($3\clubsuit$), at which point the remaining undrawn cards instantly burst into flames and the game is over.

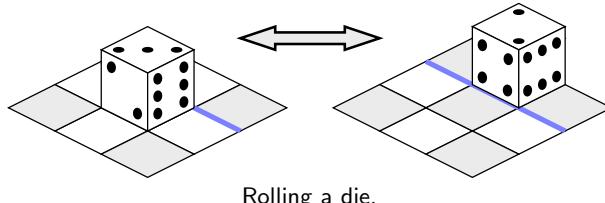
The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the previous card he gave to Penn, he gives the new card to Penn. To make the rules unambiguous, they agree on the numerical values $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

- (a) What is the expected number of cards that Teller draws?
- (b) What is the expected *maximum* value among the cards Teller gives to Penn?
- (c) What is the expected *minimum* value among the cards Teller gives to Penn?
- (d) What is the expected number of cards that Teller gives to Penn?

Full credit will be given only for *exact* answers (with correct proofs, of course).

¹In a standard deck of 52 cards, each card has a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ and a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$, and every possible suit-value pair appears in the deck exactly once. Penn and Teller normally use exploding razor-sharp ninja throwing cards for this trick.

3. A *rolling die maze* is a puzzle involving a standard six-sided die² and a grid of squares. You should imagine the grid lying on top of a table; the die always rests on and exactly covers one square. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.



Rolling a die.

Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number of pips on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

For example, here are two rolling die mazes. Black squares are blocked. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is solvable.

- (a) Suppose the input is a two-dimensional array $L[1..n][1..n]$, where each entry $L[i][j]$ stores the label of the square in the i th row and j th column, where 0 means the square is free and -1 means the square is blocked. Describe and analyze a polynomial-time algorithm to determine whether the given rolling die maze is solvable.
- *(b) Now suppose the maze is specified *implicitly* by a list of labeled and blocked squares. Specifically, suppose the input consists of an integer M , specifying the height and width of the maze, and an array $S[1..n]$, where each entry $S[i]$ is a triple (x, y, L) indicating that square (x, y) has label L . As in the explicit encoding, label -1 indicates that the square is blocked; free squares are not listed in S at all. Describe and analyze an efficient algorithm to determine whether the given rolling die maze is solvable. For full credit, the running time of your algorithm should be polynomial in the input size n .

[Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can only be solved if the initial position is chosen correctly.]

²A standard die is a cube, where each side is labeled with a different number of dots, called *pips*, between 1 and 6. The labeling is chosen so that any pair of opposite sides has a total of 7 pips.

4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons will always choose the same pecking order, even after years of separation, no matter what other pigeons are around. (Like most things, revenge is a foreign concept to pigeons.) Surprisingly, the overall pecking order in a set of pigeons can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A. Prove that any set of pigeons can be arranged in a row so that every pigeon pecks the pigeon immediately to its right.

5. Scientists have recently discovered a planet, tentatively named “Ygdrasil”, which is inhabited by a bizarre species called “vodes”. All vodes trace their ancestry back to a particular vode named Rudy. Rudy is still quite alive, as is every one of his many descendants. Vodes reproduce asexually, like bees; each vode has exactly one parent (except Rudy, who has no parent). There are three different colors of vodes—cyan, magenta, and yellow. The color of each vode is correlated exactly with the number and colors of its children, as follows:

- Each cyan vode has two children, exactly one of which is yellow.
- Each yellow vode has exactly one child, which is not yellow.
- Magenta vodes have no children.

In each of the following problems, let C , M , and Y respectively denote the number of cyan, magenta, and yellow vodes on Ygdrasil.

- (a) Prove that $M = C + 1$.
- (b) Prove that either $Y = C$ or $Y = M$.
- (c) Prove that $Y = M$ if and only if Rudy is yellow.

[Hint: Be very careful to **prove** that you have considered **all** possibilities.]

***6. [Extra credit]³**

Lazy binary is a variant of standard binary notation for representing natural numbers where we allow each “bit” to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

- The lazy binary representation of zero is 0.
- Given the lazy binary representation of any non-negative integer n , we can construct the lazy binary representation of $n + 1$ as follows:
 - (a) increment the rightmost digit;
 - (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

Here are the first several natural numbers in lazy binary notation:

0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, 102101, 102110, ...

- (a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.
- (b) Prove that for any natural number N , the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

³The “I don’t know” rule does not apply to extra credit problems. There is no such thing as “partial extra credit”.

Practice Problems

The remaining problems are for practice only. Please do not submit solutions. On the other hand, feel free to discuss these problems in office hours or on the course newsgroup.

- Sort the functions in each box from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice.

1	$\lg n$	$\lg^2 n$	\sqrt{n}	n	n^2	$2^{\sqrt{n}}$	$\sqrt{2}^n$
$2^{\sqrt{\lg n}}$	$2^{\lg \sqrt{n}}$	$\sqrt{2^{\lg n}}$	$\sqrt{2}^{\lg n}$	$\lg 2^{\sqrt{n}}$	$\lg \sqrt{2}^n$	$\lg \sqrt{2^n}$	$\sqrt{\lg 2^n}$
$\lg n^{\sqrt{2}}$	$\lg \sqrt{n}^2$	$\lg \sqrt{n^2}$	$\sqrt{\lg n^2}$	$\lg^2 \sqrt{n}$	$\lg^{\sqrt{2}} n$	$\sqrt{\lg^2 n}$	$\sqrt{\lg n^2}$

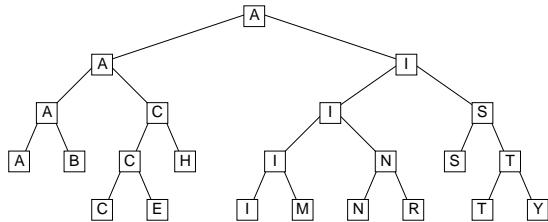
To simplify your answers, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

- Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m .
 - F_n is even if and only if n is divisible by 3.
 - $\sum_{i=0}^n F_i = F_{n+2} - 1$
 - $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$
 - If n is an integer multiple of m , then F_n is an integer multiple of F_m .
- Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ..., 52 of clubs. (They're big cards.) Penn shuffles the deck until each each of the $52!$ possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.
 - On average, how many cards does Penn give Teller?
 - On average, what is the smallest-numbered card that Penn gives Teller?
 - On average, what is the largest-numbered card that Penn gives Teller?

Prove that your answers are correct. (If you have to appeal to “intuition” or “common sense”, your answers are probably wrong.) [Hint: Solve for an n -card deck, and then set n to 52.]

4. Algorithms and data structures were developed millions of years ago by the Martians, but not quite in the same way as the recent development here on Earth. Intelligent life evolved independently on Mars' two moons, Phobos and Deimos.⁴ When the two races finally met on the surface of Mars, after thousands of years of separate philosophical, cultural, religious, and scientific development, their disagreements over the proper structure of binary search trees led to a bloody (or more accurately, ichorous) war, ultimately leading to the destruction of all Martian life.

A *Phobian* binary search tree is a full binary tree that stores a set X of search keys. The root of the tree stores the *smallest* element in X . If X has more than one element, then the left subtree stores all the elements less than some pivot value p , and the right subtree stores everything else. Both subtrees are *nonempty* Phobian binary search trees. The actual pivot value p is *never* stored in the tree.



A Phobian binary search tree for the set {M, A, R, T, I, N, B, Y, S, E, C, H}.

- (a) Describe and analyze an algorithm $\text{FIND}(x, T)$ that returns TRUE if x is stored in the Phobian binary search tree T , and FALSE otherwise.
 - (b) A *Deimoid* binary search tree is almost exactly the same as its Phobian counterpart, except that the *largest* element is stored at the root, and both subtrees are Deimoid binary search trees. Describe and analyze an algorithm to transform an n -node Phobian binary search tree into a Deimoid binary search tree in $O(n)$ time, using as little additional space as possible.
5. Tatami are rectangular mats used to tile floors in traditional Japanese houses. Exact dimensions of tatami mats vary from one region of Japan to the next, but they are always twice as long in one dimension than in the other. (In Tokyo, the standard size is 180cm×90cm.)
- (a) How many different ways are there to tile a $2 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution. [Hint: The answer involves a *familiar recursive sequence*.]
 - (b) According to tradition, tatami mats are always arranged so that four corners never meet. How many different *traditional* ways are there to tile a $3 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution.
 - *(c) How many different *traditional* ways are there to tile an $n \times n$ square with 1×2 tatami mats? Prove your answer is correct.

⁴Greek for “fear” and “panic”, respectively. Doesn’t that make you feel better?

CS 473G: Combinatorial Algorithms, Fall 2005

Homework 1

Due Tuesday, September 13, 2005, by midnight (11:59:59pm CDT)

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade.

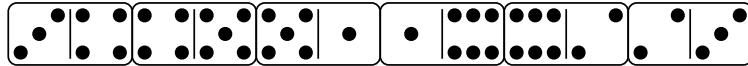
Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your answer to problem 1.

There are two steps required to prove NP-completeness: (1) Prove that the problem is in NP, by describing a polynomial-time verification algorithm. (2) Prove that the problem is NP-hard, by describing a polynomial-time reduction from some other NP-hard problem. Showing that the reduction is correct requires proving an if-and-only-if statement; don't forget to prove both the "if" part and the "only if" part.

Required Problems

1. Some NP-Complete problems
 - (a) Show that the problem of deciding whether one graph is a subgraph of another is NP-complete.
 - (b) Given a boolean circuit that embeds in the plane so that no 2 wires cross, PLANARCIRCUITSAT is the problem of determining if there is a boolean assignment to the inputs that makes the circuit output true. Prove that PLANARCIRCUITSAT is NP-Complete.
 - (c) Given a set S with $3n$ numbers, 3PARTITION is the problem of determining if S can be partitioned into n disjoint subsets, each with 3 elements, so that every subset sums to the same value. Given a set S and a collection of three element subsets of S , X3M (or *exact 3-dimensional matching*) is the problem of determining whether there is a subcollection of n disjoint triples that exactly cover S .
Describe a polynomial-time reduction from 3PARTITION to X3M.

- (d) A *domino* is a 1×2 rectangle divided into two squares, each of which is labeled with an integer.¹ In a *legal arrangement* of dominoes, the dominoes are lined up end-to-end so that the numbers on adjacent ends match.



A legal arrangement of dominos, where every integer between 1 and 6 appears twice

Prove that the following problem is NP-complete: Given an arbitrary collection D of dominoes, is there a legal arrangement of a subset of D in which every integer between 1 and n appears exactly twice?

2. Prove that the following problems are all polynomial-time equivalent, that is, if *any* of these problems can be solved in polynomial time, then *all* of them can.

- CLIQUE: Given a graph G and an integer k , does there exist a clique of size k in G ?
- FINDCLIQUE: Given a graph G and an integer k , find a clique of size k in G if one exists.
- MAXCLIQUE: Given a graph G , find the size of the largest clique in the graph.
- FINDMAXCLIQUE: Given a graph G , find a clique of maximum size in G .

3. Consider the following problem: Given a set of n points in the plane, find a set of line segments connecting the points which form a closed loop and do not intersect each other.

Describe a linear time reduction from the problem of sorting n numbers to the problem described above.

4. In graph coloring, the vertices of a graph are assigned colors so that no adjacent vertices receive the same color. We saw in class that determining if a graph is 3-colorable is NP-Complete.

Suppose you are handed a magic black box that, given a graph as input, tells you *in constant time* whether or not the graph is 3-colorable. Using this black box, give a *polynomial-time* algorithm to 3-color a graph.

5. Suppose that Cook had proved that graph coloring was NP-complete first, instead of CIRCUITSAT. Using only the fact that graph coloring is NP-complete, show that CIRCUITSAT is NP-complete.

¹These integers are usually represented by pips, exactly like dice. On a standard domino, the number of pips on each side is between 0 and 6; we will allow arbitrary integer labels. A standard set of dominoes has one domino for each possible unordered pair of labels; we do not require that every possible label pair is in our set.

Practice Problems

1. Given an initial configuration consisting of an undirected graph $G = (V, E)$ and a function $p : V \rightarrow \mathbb{N}$ indicating an initial number of pebbles on each vertex, PEBBLE-DESTRUCTION asks if there is a sequence of pebbling moves starting with the initial configuration and ending with a single pebble on only one vertex of V . Here, a pebbling move consists of removing two pebbles from a vertex v and adding one pebble to a neighbor of v . Prove that PEBBLE-DESTRUCTION is NP-complete.
2. Consider finding the median of 5 numbers by using only comparisons. What is the *exact* worst case number of comparisons needed to find the median? To prove your answer is correct, you must exhibit both an algorithm that uses that many comparisons and a proof that there is no faster algorithm. Do the same for 6 numbers.
3. PARTITION is the problem of deciding, given a set S of numbers, whether it can be partitioned into two subsets whose sums are equal. (A *partition* of S is a collection of disjoint subsets whose union is S .) SUBSETSUM is the problem of deciding, given a set S of numbers and a target sum t , whether any subset of S sum to t .
 - (a) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.
 - (b) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.
4. Recall from class that the problem of deciding whether a graph can be colored with three colors, so that no edge joins nodes of the same color, is NP-complete.
 - (a) Using the gadget in Figure 1(a), prove that deciding whether a *planar* graph can be 3-colored 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.]

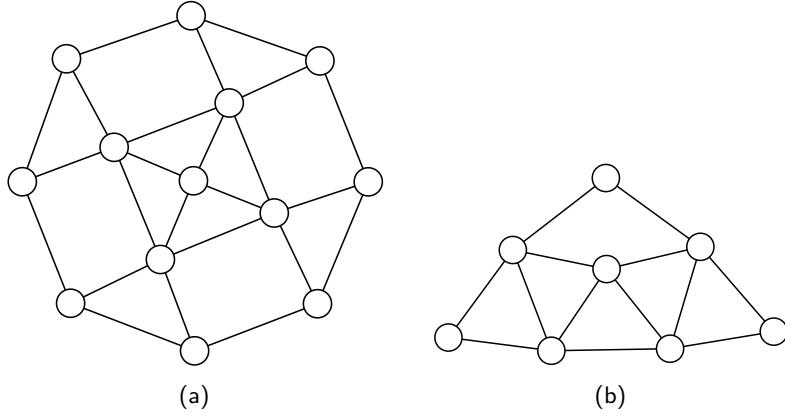


Figure 1. (a) Gadget for planar 3-colorability. (b) Gadget for degree-4 planar 3-colorability.

- (b) Using the previous result and the gadget in figure 1(b), prove that deciding whether a planar graph with maximum degree 4 can be 3-colored is NP-complete. [Hint: Show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.]

5. (a) Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian. Describe a polynomial-time algorithm to find a hamiltonian cycle in an undirected bipartite graph, or establish that no such cycle exists.
(b) Describe a polynomial time algorithm to find a hamiltonian *path* in a directed acyclic graph, or establish that no such path exists.
(c) Why don't these results imply that P=NP?
6. Consider the following pairs of problems:
 - (a) MIN SPANNING TREE and MAX SPANNING TREE
 - (b) SHORTEST PATH and LONGEST PATH
 - (c) TRAVELING SALESMAN PROBLEM and VACATION TOUR PROBLEM (the longest tour is sought).
 - (d) MIN CUT and MAX CUT (between s and t)
 - (e) EDGE COVER and VERTEX COVER
 - (f) TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

(all of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph).
Which of these pairs are polytime equivalent and which are not? Why?
7. Prove that PRIMALITY (Given n , is n prime?) is in $\text{NP} \cap \text{co-NP}$. [Hint: co-NP is easy—What's a certificate for showing that a number is composite? For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that this tree of primitive roots can be verified and used to show that n is prime in polynomial time.]
8. How much wood would a woodchuck chuck if a woodchuck could chuck wood?

CS 473G: Combinatorial Algorithms, Fall 2005

Homework 2

Due Thursday, September 22, 2005, by midnight (11:59:59pm CDT)

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your homework.

Required Problems

1. (a) Suppose Lois has an algorithm to compute the shortest common supersequence of two arrays of integers in $O(n)$ time. Describe an $O(n \log n)$ -time algorithm to compute the longest common subsequence of two arrays of integers, using Lois's algorithm as a subroutine.
(b) Describe an $O(n \log n)$ -time algorithm to compute the longest increasing subsequence of an array of integers, using Lois's algorithm as a subroutine.
(c) Now suppose Lisa has an algorithm that can compute the longest increasing subsequence of an array of integers in $O(n)$ time. Describe an $O(n \log n)$ -time algorithm to compute the longest common subsequence of two arrays $A[1..n]$ and $B[1..n]$ of integers, **where $A[i] \neq A[j]$ for all $i \neq j$** , using Lisa's algorithm as a subroutine.¹

¹For extra credit, remove the assumption that the elements of A are distinct. This is probably impossible.

2. In a previous incarnation, you worked as a cashier in the lost 19th-century Antarctic colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource on Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.²

- (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.
- (b) Describe and analyze an efficient algorithm that computes, given an integer n , the minimum number of bills needed to make n Dream Dollars.

3. Scientists have branched out from the bizarre planet of Yggdrasil to study the vodes which have settled on Ygdrasil's moon, Xryltcon. All vodes on Xryltcon are descended from the first vode to arrive there, named George. Each vode has a color, either cyan, magenta, or yellow, but breeding patterns are *not* the same as on Yggdrasil; every vode, regardless of color, has either two children (with arbitrary colors) or no children.

George and all his descendants are alive and well, and they are quite excited to meet the scientists who wish to study them. Unsurprisingly, these vodes have had some strange mutations in their isolation on Xryltcon. Each vode has a *weirdness* rating; weirder vodes are more interesting to the visiting scientists. (Some vodes even have negative weirdness ratings; they make other vodes more boring just by standing next to them.)

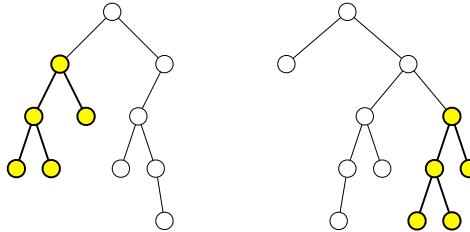
Also, Xryltconian society is strictly governed by a number of sacred cultural traditions.

- No cyan vode may be in the same room as its non-cyan children (if it has any).
- No magenta vode may be in the same room as its parent (if it has one).
- Each yellow vode must be attended at all times by its grandchildren (if it has any).
- George must be present at any gathering of more than fifty vodes.

The scientists have exactly one chance to study a group of vodes in a single room. You are given the family tree of all the vodes on Xryltcon, along with the weirdness value of each vode. Design and analyze an efficient algorithm to decide which vodes the scientists should invite to maximize the sum of the weirdness values of the vodes in the room. Be careful to respect all of the vodes' cultural taboos.

²For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see <http://www.dream-dollars.com>. Really.

4. A *subtree* of a (rooted, ordered) binary tree T consists of a node and all its descendants. Design and analyze an efficient algorithm to compute the *largest common subtree* of two given binary trees T_1 and T_2 , that is, the largest subtree of T_1 that is isomorphic to a subtree in T_2 . The *contents* of the nodes are irrelevant; we are only interested in matching the underlying combinatorial structure.



Two binary trees, with their largest common subtree emphasized

5. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. An *digital subsequence* of D is an sequence of positive integers composed in the usual way from disjoint substrings of D . For example, 3, 4, 5, 6, 23, 38, 62, 64, 83, 279 is an increasing digital subsequence of the first several digits of π :

$$\boxed{3}, 1, \boxed{4}, 1, \boxed{5}, 9, \boxed{6}, \boxed{2, 3}, 4, \boxed{3, 8}, 4, \boxed{6, 2}, \boxed{6, 4}, 3, 3, \boxed{8, 3}, \boxed{2, 7, 9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the previous example has length 10.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D . [Hint: Be careful about your computational assumptions. How long does it take to compare two k -digit numbers?]

- *6. [Extra credit] The *chromatic number* of a graph G is the minimum number of colors needed to color the nodes of G so that no pair of adjacent nodes have the same color.

- (a) Describe and analyze a *recursive* algorithm to compute the chromatic number of an n -vertex graph in $O(4^n \text{ poly}(n))$ time. [Hint: Catalan numbers play a role here.]
- (b) Describe and analyze an algorithm to compute the chromatic number of an n -vertex graph in $O(3^n \text{ poly}(n))$ time. [Hint: Use dynamic programming. What is $(1+x)^n$?]
- (c) Describe and analyze an algorithm to compute the chromatic number of an n -vertex graph in $O((1+3^{1/3})^n \text{ poly}(n))$ time. [Hint: Use (but don't regurgitate) the algorithm in the lecture notes that counts all the maximal independent sets in an n -vertex graph in $O(3^{n/3})$ time.]

Practice Problems

- *1. Describe an algorithm to solve 3SAT in time $O(\phi^n \text{ poly}(n))$, where $\phi = (1 + \sqrt{5})/2$. [Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals.]
2. Describe and analyze an algorithm to compute the longest increasing subsequence in an n -element array of integers in $O(n \log n)$ time. [Hint: Modify the $O(n^2)$ -time algorithm presented in class.]
3. The *edit distance* between two strings A and B , denoted $\text{Edit}(A, B)$, is the minimum number of insertions, deletions, or substitutions required to transform A into B (or vice versa). Edit distance is sometimes also called the *Levenshtein distance*.

Let $\mathbf{A} = \{A_1, A_2, \dots, A_k\}$ be a set of strings. The *edit radius* of \mathbf{A} is the minimum over all strings X of the maximum edit distance from X to any string A_i :

$$\text{>EditRadius}(\mathbf{A}) = \min_{\text{strings } X} \max_{1 \leq i \leq k} \text{Edit}(X, A_i)$$

A string X that achieves this minimum is called an *edit center* of \mathbf{A} . A set of strings may have several edit centers, but the edit radius is unique.

Describe an efficient algorithm to compute the edit radius of three given strings.

4. Given 5 sequences of numbers, each of length n , design and analyze an efficient algorithm to compute the longest common subsequence among all 5 sequences.
5. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the i th word is $W[i]$ pixels wide. We want to break the paragraph into several lines, each exactly L pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. (Look at the paragraph you are reading right now!) There must be at least one pixel of white space between any two words on the same line. Thus, if a line contains words i through j , then the amount of *extra* white space on that line is $L - j + i - \sum_{k=i}^j W[k]$.

Define the *slop* of a paragraph layout as the sum, over all lines *except the last*, of the cube of the extra white space in each line. Describe an efficient algorithm to layout the paragraph with minimum slop, given the list $W[1..n]$ of word widths as input. You can assume that $W[i] < L/2$ for each i , so that each line contains at least two words.

6. A *partition* of a positive integer n is a multiset of positive integers that sum to n . Traditionally, the elements of a partition are written in non-decreasing order, separated by + signs. For example, the integer 7 has exactly twelve partitions:

$$\begin{array}{lll} 1 + 1 + 1 + 1 + 1 + 1 + 1 & 3 + 1 + 1 + 1 + 1 & 4 + 1 + 1 + 1 \\ 2 + 1 + 1 + 1 + 1 + 1 & 3 + 2 + 1 + 1 & 4 + 2 + 1 \\ 2 + 2 + 1 + 1 + 1 & 3 + 2 + 2 & 4 + 3 \\ 2 + 2 + 2 + 1 & 3 + 3 + 1 & 7 \end{array}$$

The *roughness* of a partition $a_1 + a_2 + \dots + a_k$ is defined as follows:

$$\rho(a_1 + a_2 + \dots + a_k) = \sum_{i=1}^{k-1} |a_{i+1} - a_i - 1| + a_k - 1$$

A *smoothest* partition of n is the partition of n with minimum roughness. Intuitively, the smoothest partition is the one closest to a descending arithmetic series $k + \dots + 3 + 2 + 1$, which is the only partition that has roughness 0. For example, the smoothest partitions of 7 are $4 + 2 + 1$ and $3 + 2 + 1 + 1$:

$$\begin{array}{lll} \rho(1 + 1 + 1 + 1 + 1 + 1 + 1) = 6 & \rho(3 + 1 + 1 + 1 + 1) = 4 & \rho(4 + 1 + 1 + 1) = 4 \\ \rho(2 + 1 + 1 + 1 + 1 + 1) = 4 & \rho(3 + 2 + 1 + 1) = 1 & \rho(4 + 2 + 1) = 1 \\ \rho(2 + 2 + 1 + 1 + 1) = 3 & \rho(3 + 2 + 2) = 2 & \rho(4 + 3) = 2 \\ \rho(2 + 2 + 2 + 1) = 2 & \rho(3 + 3 + 1) = 2 & \rho(7) = 7 \end{array}$$

Describe and analyze an algorithm to compute, given a positive integer n , a smoothest partition of n .

CS 473G: Combinatorial Algorithms, Fall 2005

Homework 3

Due Tuesday, October 18, 2005, at midnight

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your homework.

1. Consider the following greedy approximation algorithm to find a vertex cover in a graph:

```
GREEDYVERTEXCOVER( $G$ ):  
     $C \leftarrow \emptyset$   
    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  $C$ 
```

In class we proved that the approximation ratio of this algorithm is $O(\log n)$; your task is to prove a matching lower bound. Specifically, prove that for any integer n , there is a graph G with n vertices such that $\text{GREEDYVERTEXCOVER}(G)$ returns a vertex cover that is $\Omega(\log n)$ times larger than optimal.

2. Prove that for *any* constant k and *any* graph coloring algorithm A , there is a graph G such that $A(G) > OPT(G) + k$, where $A(G)$ is the number of colors generated by algorithm A for graph G , and $OPT(G)$ is the optimal number of colors for G .

[Note: This does not contradict the possibility of a constant **factor** approximation algorithm.]

3. Let R be a set of rectangles in the plane, with horizontal and vertical edges. A *stabbing set* for R is a set of points S such that every rectangle in R contains at least one point in S . The *rectangle stabbing* problem asks, given a set R of rectangles, for the smallest stabbing set S .

- (a) Prove that the rectangle stabbing problem is NP-hard.
- (b) Describe and analyze an efficient approximation algorithm for the rectangle stabbing problem. Give bounds on the approximation ratio of your algorithm.

4. Consider the following approximation scheme for coloring a graph G .

```

TREECOLOR( $G$ ):
 $T \leftarrow$  any spanning tree of  $G$ 
Color the tree  $T$  with two colors
 $c \leftarrow 2$ 
for each edge  $(u, v) \in G \setminus T$ 
     $T \leftarrow T \cup \{(u, v)\}$ 
    if  $\text{color}(u) = \text{color}(v)$     «Try recoloring  $u$  with an existing color»
        for  $i \leftarrow 1$  to  $c$ 
            if no neighbor of  $u$  in  $T$  has color  $i$ 
                 $\text{color}(u) \leftarrow i$ 
    if  $\text{color}(u) = \text{color}(v)$     «Try recoloring  $v$  with an existing color»
        for  $i \leftarrow 1$  to  $c$ 
            if no neighbor of  $v$  in  $T$  has color  $i$ 
                 $\text{color}(v) \leftarrow i$ 
    if  $\text{color}(u) = \text{color}(v)$     «Give up and use a new color»
         $c \leftarrow c + 1$ 
         $\text{color}(u) \leftarrow c$ 
return  $c$ 

```

- (a) Prove that this algorithm correctly colors any bipartite graph.
- (b) Prove an upper bound C on the number of colors used by this algorithm. Give a sample graph and run that requires C colors.
- (c) Does this algorithm approximate the minimum number of colors up to a constant factor? In other words, is there a constant α such that $\text{TREECOLOR}(\mathcal{G}) < \alpha \cdot \text{OPT}(\mathcal{G})$ for any graph \mathcal{G} ? Justify your answer.

5. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is NP-Hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NEXTFIT( $W[1..n]$ ):
```

```

 $b \leftarrow 0$ 
 $Total[0] \leftarrow \infty$ 
for  $i \leftarrow 1$  to  $n$ 
  if  $Total[b] + W[i] > 1$ 
     $b \leftarrow b + 1$ 
     $Total[b] \leftarrow W[i]$ 
  else
     $Total[b] \leftarrow Total[b] + W[i]$ 
return  $b$ 
```

```
FIRSTFIT( $W[1..n]$ ):
```

```

 $b \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$ 
   $j \leftarrow 1$ ;  $found \leftarrow \text{FALSE}$ 
  while  $j \leq b$  and  $found = \text{FALSE}$ 
    if  $Total[j] + W[i] \leq 1$ 
       $Total[j] \leftarrow Total[j] + W[i]$ 
       $found \leftarrow \text{TRUE}$ 
     $j \leftarrow j + 1$ 
  if  $found = \text{FALSE}$ 
     $b \leftarrow b + 1$ 
     $Total[b] = W[i]$ 
return  $b$ 
```

- (a) Prove that NEXTFIT uses at most twice the optimal number of bins.
- (b) Prove that FIRSTFIT uses at most twice the optimal number of bins.
- (c) Prove that if the weight array W is initially sorted in decreasing order, then FIRSTFIT uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):
 - In the packing computed by FIRSTFIT, every item with weight more than $1/3$ is placed in one of the first OPT bins.
 - FIRSTFIT places at most $OPT - 1$ items outside the first OPT bins.

CS 473G: Combinatorial Algorithms, Fall 2005

Homework 4

Due Thursday, October 27, 2005, at midnight

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

Homeworks may be done in teams of up to three people. Each team turns in just one solution; every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your solution to problem 1.

If you are an I2CS student, print “(I2CS)” next to your name. Teams that include both on-campus and I2CS students can have up to four members. Any team containing both on-campus and I2CS students automatically receives 3 points of extra credit.

For the rest of the semester, unless specifically stated otherwise, you may assume that the function $\text{RANDOM}(m)$ returns an integer chosen uniformly at random from the set $\{1, 2, \dots, m\}$ in $O(1)$ time. For example, a fair coin flip is obtained by calling $\text{RANDOM}(2)$.

1. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. [Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]

2. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue containing the empty set.
- **FINDMIN(Q)**: Return the smallest element of Q (if any).
- **DELETEMIN(Q)**: Remove the smallest element in Q (if any).
- **INSERT(Q, x)**: Insert element x into Q , if it is not already there.
- **DECREASEKEY(Q, x, y)**: Replace an element $x \in Q$ with a smaller key y . (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x .
- **DELETE(Q, x)**: Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x .
- **MELD(Q_1, Q_2)**: Return a new priority queue containing all the elements of Q_1 and Q_2 ; this operation destroys Q_1 and Q_2 .

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```

MELD( $Q_1, Q_2$ ):
  if  $Q_1$  is empty return  $Q_2$ 
  if  $Q_2$  is empty return  $Q_1$ 
  if  $key(Q_1) > key(Q_2)$ 
    swap  $Q_1 \leftrightarrow Q_2$ 
  with probability  $1/2$ 
     $left(Q_1) \leftarrow \text{MELD}(left(Q_1), Q_2)$ 
  else
     $right(Q_1) \leftarrow \text{MELD}(right(Q_1), Q_2)$ 
  return  $Q_1$ 
```

- (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of $\text{MELD}(Q_1, Q_2)$ is $O(\log n)$, where $n = |Q_1| + |Q_2|$. [Hint: How long is a random root-to-leaf path in an n -node binary tree if each left/right choice is made with equal probability?]
- (b) Prove that $\text{MELD}(Q_1, Q_2)$ runs in $O(\log n)$ time with high probability.
- (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

3. Let $M[1..n][1..n]$ be an $n \times n$ matrix in which every row and every column is sorted. Such an array is called *totally monotone*. No two elements of M are equal.

- (a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M smaller than $M[i][j]$ and larger than $M[i'][j']$.
- (b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements smaller than $M[i][j]$ and larger than $M[i'][j']$. Assume the requested range is always non-empty.
- (c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

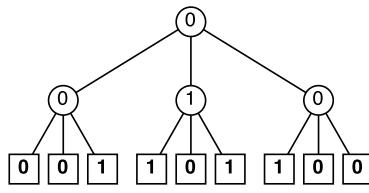
4. Let $X[1..n]$ be an array of n distinct real numbers, and let $N[1..n]$ be an array of indices with the following property: If $X[i]$ is the largest element of X , then $X[N[i]]$ is the smallest element of X ; otherwise, $X[N[i]]$ is the smallest element of X that is larger than $X[i]$.

For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$N[i]$	6	8	9	5	2	3	1	7	4

Describe and analyze a randomized algorithm that determines whether a given number x appears in the array X in $O(\sqrt{n})$ expected time. **Your algorithm may not modify the arrays X and $Next$.**

5. A *majority tree* is a complete ternary tree with depth n , where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, the root has value 0.



A majority tree with depth $n = 2$.

- (a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. [Hint: Consider the special case $n = 1$. Recurse.]
- (b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. [Hint: Consider the special case $n = 1$. Recurse.]

*6. [Extra credit] In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0, 1]$, but in practice, computers only have access to random *bits*. This problem asks you to analyze a modification of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1 .. \infty]$ of random bits, which is interpreted as the rational number

$$\text{priority}(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or ‘reveal’) new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LARGERPRIORITY( $v, w$ ):
for  $i \leftarrow 1$  to  $\infty$ 
    if  $i > \ell_v$ 
         $\ell_v \leftarrow i$ ;  $\pi_v[i] \leftarrow \text{RANDOMBIT}$ 
    if  $i > \ell_w$ 
         $\ell_w \leftarrow i$ ;  $\pi_w[i] \leftarrow \text{RANDOMBIT}$ 
    if  $\pi_v[i] > \pi_w[i]$ 
        return  $v$ 
    else if  $\pi_v[i] < \pi_w[i]$ 
        return  $w$ 
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LARGERPRIORITY during these insertions.

- (a) Prove that $E[L] = \Theta(n)$.
- (b) Prove that $E[\ell_v] = \Theta(1)$ for any node v . [Hint: This is equivalent to part (a). Why?]
- (c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. [Hint: Why doesn’t this contradict part (b)?]

CS 473G: Combinatorial Algorithms, Fall 2005

Homework 5

Due Thursday, November 17, 2005, at midnight

(because you *really* don't want homework due over Thanksgiving break)

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

Homeworks may be done in teams of up to three people. Each team turns in just one solution; every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Attach this sheet (or the equivalent information) to the top of your solution to problem 1.

If you are an I2CS student, print “(I2CS)” next to your name. Teams that include both on-campus and I2CS students can have up to four members. Any team containing both on-campus and I2CS students automatically receives 3 points of extra credit.

Problems labeled \heartsuit are likely to require techniques from next week's lectures on cuts, flows, and matchings. See also Chapter 7 in Kleinberg and Tardos, or Chapter 26 in CLRS.

- \heartsuit 1. Suppose you are asked to construct the minimum spanning tree of a graph G , but you are not completely sure of the edge weights. Specifically, you have a *conjectured* weight $\tilde{w}(e)$ for every edge e in the graph, but you also know that up to k of these conjectured weights are wrong. With the exception of one edge e whose true weight you know exactly, you don't know which edges are wrong, or even how they're wrong; the true weights of those edges could be larger or smaller than the conjectured weights. Given this unreliable information, it is of course impossible to reliably construct the true minimum spanning tree of G , but it is still possible to say something about your special edge.

Describe and analyze an efficient algorithm to determine whether a specific edge e , whose *actual* weight is known, is *definitely not* in the minimum spanning tree of G under the stated conditions. The input consists of the graph G , the conjectured weight function $\tilde{w} : E(G) \rightarrow \mathbb{R}$, the positive integer k , and the edge e .

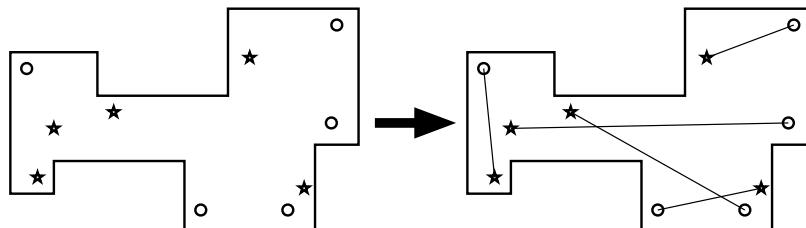
2. Most classical minimum-spanning-tree algorithms use the notions of ‘safe’ and ‘useless’ edges described in the lecture notes, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is *dangerous* if it is the longest edge in some cycle in G , and *useful* if it does not lie in any cycle in G .

- (a) Prove that the minimum spanning tree of G contains every useful edge.
- (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.
- (c) Describe and analyze an efficient implementation of the “anti-Kruskal” MST algorithm: Examine the edges of G in *decreasing* order; if an edge is dangerous, remove it from G . *[Hint: It won’t be as fast as the algorithms you saw in class.]*

γ 3. The UIUC Computer Science department has decided to build a mini-golf course in the basement of the Siebel Center! The playing field is a closed polygon bounded by m horizontal and vertical line segments, meeting at right angles. The course has n *starting points* and n *holes*, in one-to-one correspondence. It is always possible hit the ball along a straight line directly from each starting point to the corresponding hole, without touching the boundary of the playing field. (Players are not allowed to bounce golf balls off the walls; too much glass.) The n starting points and n holes are all at distinct locations.

Sadly, the architect’s computer crashed just as construction was about to begin. Thanks to the herculean efforts of their sysadmins, they were able to recover the *locations* of the starting points and the holes, but all information about which starting points correspond to which holes was lost!

Describe and analyze an algorithm to compute a one-to-one correspondence between the starting points and the holes that meets the straight-line requirement, or to report that no such correspondence exists. The input consists of the x - and y -coordinates of the m corners of the playing field, the n starting points, and the n holes. Assume you can determine in constant time whether two line segments intersect, given the x - and y -coordinates of their endpoints.



A minigolf course with five starting points (\star) and five holes (\circ), and a legal correspondence between them.

γ 4. Let $G = (V, E)$ be a directed graph where the in-degree of each vertex is equal to its out-degree. Prove or disprove the following claim: For any two vertices u and v in G , the number of mutually edge-disjoint paths from u to v is equal to the number of mutually edge-disjoint paths from v to u .

5. You are given a set of n boxes, each specified by its height, width, and depth. The order of the dimensions is unimportant; for example, a $1 \times 2 \times 3$ box is exactly the same as a $3 \times 1 \times 2$ box or a $2 \times 1 \times 3$ box. You can nest box A inside box B if and only if A can be rotated so that it has strictly smaller height, strictly smaller width, and strictly smaller depth than B .
- (a) Design and analyze an efficient algorithm to determine the largest sequence of boxes that can be nested inside one another. [*Hint: Model the nesting relationship as a graph.*]
 - Υ (b) Describe and analyze an efficient algorithm to nest all n boxes into as few groups as possible, where each group consists of a nested sequence. You are not allowed to put two boxes side-by-side inside a third box, even if they are small enough to fit.¹ [*Hint: Model the nesting relationship as a **different** graph.*]
6. [*Extra credit*] Prove that Ford's generic shortest-path algorithm (described in the lecture notes) can take exponential time in the worst case when implemented with a stack instead of a heap (like Dijkstra) or a queue (like Bellman-Ford). Specifically, construct for every positive integer n a weighted directed n -vertex graph G_n , such that the stack-based shortest-path algorithm call RELAX $\Omega(2^n)$ times when G_n is the input graph. [*Hint: Towers of Hanoi.*]

¹Without this restriction, the problem is NP-hard, even for one-dimensional “boxes”.

CS 473G: Combinatorial Algorithms, Fall 2005

Homework 6

Practice only; nothing to turn in.

1. A small airline, Ivy Air, flies between three cities: Ithaca (a small town in upstate New York), Newark (an eyesore in beautiful New Jersey), and Boston (a yuppie town in Massachusetts). They offer several flights but, for this problem, let us focus on the Friday afternoon flight that departs from Ithaca, stops in Newark, and continues to Boston. There are three types of passengers:

- (a) Those traveling from Ithaca to Newark (god only knows why).
- (b) Those traveling from Newark to Boston (a very good idea).
- (c) Those traveling from Ithaca to Boston (it depends on who you know).

The aircraft is a small commuter plane that seats 30 passengers. The airline offers three fare classes:

- (a) Y class: full coach.
- (b) B class: nonrefundable.
- (c) M class: nonrefundable, 3-week advanced purchase.

Ticket prices, which are largely determined by external influences (i.e., competitors), have been set and advertised as follows:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	300	160	360
B	220	130	280
M	100	80	140

Based on past experience, demand forecasters at Ivy Air have determined the following upper bounds on the number of potential customers in each of the 9 possible origin-destination/fare-class combinations:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	4	8	3
B	8	13	10
M	22	20	18

The goal is to decide how many tickets from each of the 9 origin/destination/fare-class combinations to sell. The constraints are that the place cannot be overbooked on either the two legs of the flight and that the number of tickets made available cannot exceed the forecasted maximum demand. The objective is to maximize the revenue.

Formulate this problem as a linear programming problem.

2. (a) Suppose we are given a directed graph $G = (V, E)$, a length function $\ell : E \rightarrow \mathbb{R}$, and a source vertex $s \in V$. Write a linear program to compute the shortest-path distance from s to every other vertex in V . [Hint: Define a variable for each vertex representing its distance from s . What objective function should you use?]
- (b) In the *minimum-cost multicommodity-flow* problem, we are given a directed graph $G = (V, E)$, in which each edge $u \rightarrow v$ has an associated nonnegative *capacity* $c(u \rightarrow v) \geq 0$ and an associated *cost* $\alpha(u \rightarrow v)$. We are given k different commodities, each specified by a triple $K_i = (s_i, t_i, d_i)$, where s_i is the source node of the commodity, t_i is the target node for the commodity i , and d_i is the *demand*: the desired flow of commodity i from s_i to t_i . A *flow* for commodity i is a non-negative function $f_i : E \rightarrow \mathbb{R}_{\geq 0}$ such that the total flow into any vertex other than s_i or t_i is equal to the total flow out of that vertex. The *aggregate flow* $F : E \rightarrow \mathbb{R}$ is defined as the sum of these individual flows: $F(u \rightarrow v) = \sum_{i=1}^k f_i(u \rightarrow v)$. The aggregate flow $F(u \rightarrow v)$ on any edge must not exceed the capacity $c(u \rightarrow v)$. The goal is to find an aggregate flow whose total *cost* $\sum_{u \rightarrow v} F(u \rightarrow v) \cdot \alpha(u \rightarrow v)$ is as small as possible. (Costs may be negative!) Express this problem as a linear program.
3. In class we described the duality transformation only for linear programs in canonical form:

$$\begin{array}{ccc} \text{Primal (II)} & \iff & \text{Dual (II)} \\ \boxed{\begin{array}{ll} \max & c \cdot x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{array}} & & \boxed{\begin{array}{ll} \min & y \cdot b \\ \text{s.t.} & yA \geq c \\ & y \geq 0 \end{array}} \end{array}$$

Describe precisely how to dualize the following more general linear programming problem:

$$\begin{aligned} & \text{maximize} \quad \sum_{j=1}^d c_j x_j \\ & \text{subject to} \quad \sum_{j=1}^d a_{ij} x_j \leq b_i \quad \text{for each } i = 1..p \\ & \quad \sum_{j=1}^d a_{ij} x_j = b_i \quad \text{for each } i = p+1..p+q \\ & \quad \sum_{j=1}^d a_{ij} x_j \geq b_i \quad \text{for each } i = p+q+1..n \end{aligned}$$

Your dual problem should have one variable for each primal constraint, and the dual of your dual program should be precisely the original linear program.

4. (a) Model the maximum-cardinality bipartite matching problem as a linear programming problem. The input is a bipartite graph $G = (U, V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G . Your linear program should have one variable for every edge.
- (b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

5. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.

- (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.
- (b) Prove that finding the optimal feasible solution to an integer program is NP-hard.

[Hint: Almost any NP-hard decision problem can be rephrased as an integer program. Pick your favorite.]

6. Consider the LP formulation of the shortest path problem presented in class:

$$\begin{aligned} & \text{maximize} && d_t \\ & \text{subject to} && d_s = 0 \\ & && d_v - d_u \leq \ell_{u \rightarrow v} \quad \text{for every edge } u \rightarrow v \end{aligned}$$

Characterize the feasible bases for this linear program in terms of the original weighted graph. What does a simplex pivoting operation represent? What is a locally optimal (*i.e.*, dual feasible) basis? What does a dual pivoting operation represent?

7. Consider the LP formulation of the maximum-flow problem presented in class:

$$\begin{aligned} & \text{maximize} && \sum_w f_{s \rightarrow w} - \sum_u f_{u \rightarrow s} \\ & \text{subject to} && \sum_w f_{v \rightarrow w} - \sum_u f_{u \rightarrow v} = 0 \quad \text{for every vertex } v \neq s, t \\ & && f_{u \rightarrow v} \leq c_{u \rightarrow v} \quad \text{for every edge } u \rightarrow v \\ & && f_{u \rightarrow v} \geq 0 \quad \text{for every edge } u \rightarrow v \end{aligned}$$

Is the Ford-Fulkerson augmenting path algorithm an instance of the simplex algorithm applied to this linear program? Why or why not?

- *8. *Helly's theorem* says that for any collection of convex bodies in \mathbb{R}^n , if every $n + 1$ of them intersect, then there is a point lying in the intersection of all of them. Prove Helly's theorem for the special case that the convex bodies are halfspaces. *[Hint: Show that if a system of inequalities $Ax \geq b$ does not have a solution, then we can select $n + 1$ of the inequalities such that the resulting system does not have a solution. Construct a primal LP from the system by choosing a 0 cost vector.]*

You have 90 minutes to answer four of these questions.

Write your answers in the separate answer booklet.

You may take the question sheet with you when you leave.

1. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

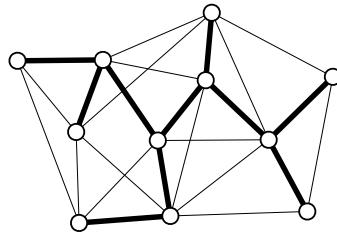
- (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.
 (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.
2. Suppose you are given a magical black box that can tell you in constant time whether or not a given graph has a Hamiltonian cycle. Using this magic black box as a subroutine, describe and analyze a *polynomial-time* algorithm to actually compute a Hamiltonian cycle in a given graph, if one exists.
3. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The *size* of a tiling cover is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X .



A set of intervals. The seven shaded intervals form a tiling path.

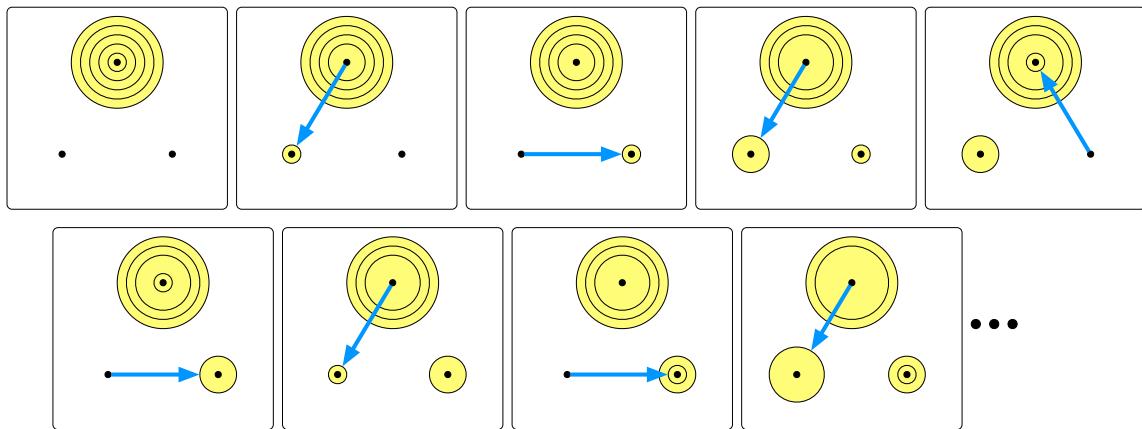
4. Prove that the following problem is NP-complete: Given an undirected graph, does it have a spanning tree in which every node has degree at most 3?



A graph with a spanning tree of maximum degree 3.

5. The *Tower of Hanoi* puzzle, invented by Edouard Lucas in 1883, consists of three pegs and n disks of different sizes. Initially, all n disks are on the same peg, stacked in order by size, with the largest disk on the bottom and the smallest disk on top. In a single move, you can move the topmost disk on any peg to another peg; however, you are never allowed to place a larger disk on top of a smaller one. Your goal is to move all n disks to a different peg.

- (a) Prove that the Tower of Hanoi puzzle can be solved in exactly $2^n - 1$ moves. [Hint: You've probably seen this before.]
- (b) Now suppose the pegs are arranged in a circle and you are *only* allowed to move disks *counterclockwise*. How many moves do you need to solve this restricted version of the puzzle? Give an upper bound in the form $O(f(n))$ for some function $f(n)$. Prove your upper bound is correct.



A top view of the first eight moves in a clockwise Towers of Hanoi solution

You have 90 minutes to answer four of these questions.

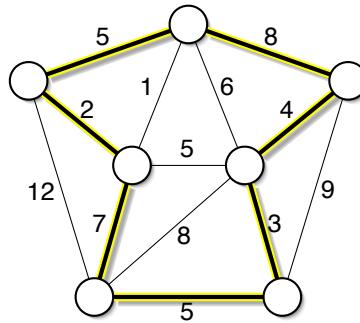
Write your answers in the separate answer booklet.

You may take the question sheet with you when you leave.

Chernoff Bounds: If X is the sum of independent indicator variables and $\mu = \mathbb{E}[X]$, then the following inequalities hold for any $\delta > 0$:

$$\Pr[X < (1 - \delta)\mu] < \left(\frac{e^{-\delta}}{(1 - \delta)^{1-\delta}} \right)^\mu \quad \Pr[X > (1 + \delta)\mu] < \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu$$

1. Describe and analyze an algorithm that randomly shuffles an array $X[1..n]$, so that each of the $n!$ possible permutations is equally likely, in $O(n)$ time. (Assume that the subroutine `RANDOM(m)` returns an integer chosen uniformly at random from the set $\{1, 2, \dots, m\}$ in $O(1)$ time.)
2. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G . Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.



A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

3. A sequence of numbers $\langle a_1, a_2, a_3, \dots, a_n \rangle$ is *oscillating* if $a_i < a_{i+1}$ for every odd index i and $a_i > a_{i+1}$ for every even index i . Describe and analyze an efficient algorithm to compute the longest oscillating subsequence in a sequence of n integers.
4. This problem asks you to how to efficiently modify a maximum flow if one of the edge capacities changes. Specifically, you are given a directed graph $G = (V, E)$ with capacities $c : E \rightarrow \mathbb{Z}_+$, and a maximum flow $F : E \rightarrow \mathbb{Z}$ from some vertex s to some other vertex t in G . Describe and analyze efficient algorithms for the following operations:
 - (a) `INCREMENT(e)` — Increase the capacity of edge e by 1 and update the maximum flow F .
 - (b) `DECREMENT(e)` — Decrease the capacity of edge e by 1 and update the maximum flow F .

Both of your algorithms should be significantly faster than recomputing the maximum flow from scratch.

5.

6. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges.

- (a) Prove that it is NP-hard to compute the most interesting 3-coloring of a graph. [Hint: There is a one-line proof. Use one of the NP-hard problems described in class.]
- (b) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G . Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$. [Hint: There is a one-line proof.]
- (c) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G . Suppose we assign each vertex in G a *random* color from the set {red, green, blue}. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.

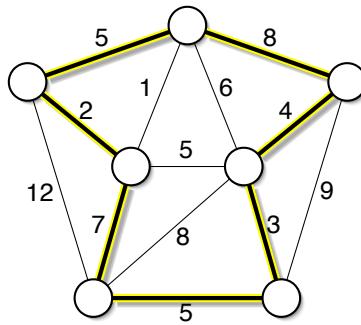
7.

You have 180 minutes to answer six of these questions.

Write your answers in the separate answer booklet.

You may take the question sheet with you when you leave.

1. Describe and analyze an algorithm that randomly shuffles an array $X[1..n]$, so that each of the $n!$ possible permutations is equally likely, in $O(n)$ time. (Assume that the subroutine $\text{RANDOM}(m)$ returns an integer chosen uniformly at random from the set $\{1, 2, \dots, m\}$ in $O(1)$ time.)
2. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G . Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.

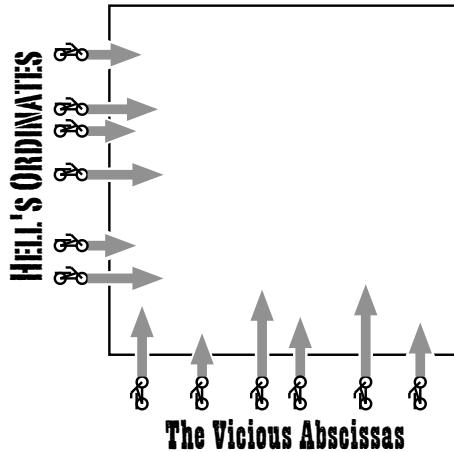


A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

3. Suppose you are given a directed graph $G = (V, E)$ with capacities $c : E \rightarrow \mathbb{Z}_+$ and a maximum flow $F : E \rightarrow \mathbb{Z}$ from some vertex s to some other vertex t in G . Describe and analyze efficient algorithms for the following operations:
 - (a) $\text{INCREMENT}(e)$ — Increase the capacity of edge e by 1 and update the maximum flow F .
 - (b) $\text{DECREMENT}(e)$ — Decrease the capacity of edge e by 1 and update the maximum flow F .
 Both of your algorithms should be significantly faster than recomputing the maximum flow from scratch.
4. Suppose you are given an undirected graph G and two vertices s and t in G . Two paths from s to t are *vertex-disjoint* if the only vertices they have in common are s and t . Describe and analyze an efficient algorithm to compute the maximum number of vertex-disjoint paths between s and t in G . [Hint: Reduce this to a more familiar problem on a suitable directed graph G' .]

5. A sequence of numbers $\langle a_1, a_2, a_3, \dots, a_n \rangle$ is *oscillating* if $a_i < a_{i+1}$ for every *odd* index i and $a_i > a_{i+1}$ for every *even* index i . For example, the sequence $\langle 2, 7, 1, 8, 2, 8, 1, 8, 3 \rangle$ is oscillating. Describe and analyze an efficient algorithm to compute the longest oscillating subsequence in a sequence of n integers.
6. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem we saw in class is a special case.
- Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G . Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.
 - Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G . Suppose we assign each vertex in G a *random* color from the set {red, green, blue}. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.
7. It's time for the 3rd Quasi-Annual Champaign-Urbana Ice Motorcycle Demolition Derby Race-O-Rama and Spaghetti Bake-Off! The main event is a competition between two teams of n motorcycles in a huge square ice-covered arena. All of the motorcycles have spiked tires so that they can ride on the ice. Each motorcycle drags a long metal chain behind it. Whenever a motorcycle runs over a chain, the chain gets caught in the tire spikes, and the motorcycle crashes. Two motorcycles can also crash by running directly into each other. All the motorcycle start simultaneously. Each motorcycle travels in a straight line at a constant speed until it either crashes or reaches the opposite wall—no turning, no braking, no speeding up, no slowing down. The Vicious Abscissas start at the south wall of the arena and ride directly north (vertically). Hell's Ordinates start at the west wall of the arena and ride directly east (horizontally). If any motorcycle completely crosses the arena, that rider's entire team wins the competition.

Describe and analyze an efficient algorithm to decide which team will win, given the starting position and speed of each motorcycle.



CS 473U: Undergraduate Algorithms, Fall 2006

Homework 0

Due Friday, September 1, 2006 at noon in 3229 Siebel Center

Name:	
Net ID:	Alias:

I understand the Homework Instructions and FAQ.

-
- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and submit this page with your solutions. We will list homework and exam grades on the course web site by alias. For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid) your Social Security number. Please use the same alias for every homework and exam.

Federal law forbids us from publishing your grades, even anonymously, without your explicit permission. **By providing an alias, you grant us permission to list your grades on the course web site; if you do not provide an alias, your grades will not be listed.**

- Please carefully read the Homework Instructions and FAQ on the course web page, and then check the box above. This page describes what we expect in your homework solutions—start each numbered problem on a new sheet of paper, write your name and NetID on every page, don’t turn in source code, analyze and prove everything, use good English and good logic, and so on—as well as policies on grading standards, regrading, and plagiarism. **See especially the policies regarding the magic phrases “I don’t know” and “and so on”.** If you have *any* questions, post them to the course newsgroup or ask in lecture.
 - This homework tests your familiarity with prerequisite material—basic data structures, big-Oh notation, recurrences, discrete probability, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Each numbered problem is worth 10 points; not all subproblems have equal weight.
-

#	1	2	3	4	5	6*	Total
Score							
Grader							

Please put your answers to problems 1 and 2 on the same page.

1. Sort the functions listed below from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do not turn in proofs**, but you should probably do them anyway, just for practice.

To simplify your answers, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

$$\begin{array}{cccccccc}
 \lg n & \ln n & \sqrt{n} & n & n \lg n & n^2 & 2^n & n^{1/n} \\
 n^{1+1/\lg n} & \lg^{1000} n & 2^{\sqrt{\lg n}} & (\sqrt{2})^{\lg n} & \lg^{\sqrt{2}} n & n^{\sqrt{2}} & (1 + \frac{1}{n})^n & n^{1/1000} \\
 H_n & H_{\sqrt{n}} & 2^{H_n} & H_{2^n} & F_n & F_{n/2} & \lg F_n & F_{\lg n}
 \end{array}$$

In case you've forgotten:

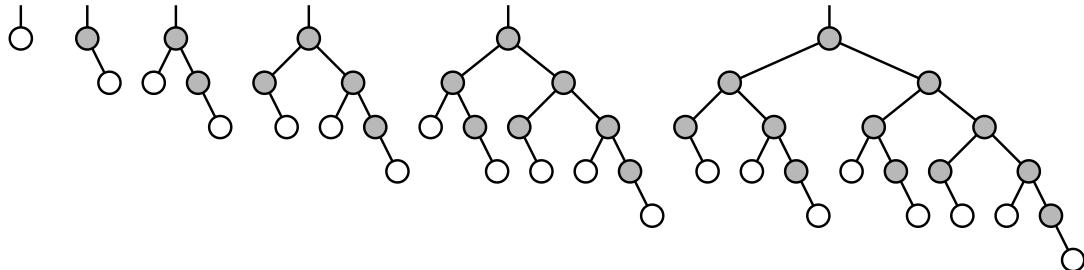
- $\lg n = \log_2 n \neq \ln n = \log_e n$
- $\lg^3 n = (\lg n)^3 \neq \lg \lg \lg n$.
- The harmonic numbers: $H_n = \sum_{i=1}^n 1/i \approx \ln n + 0.577215\dots$
- The Fibonacci numbers: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$

2. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Proofs are *not* required; just give us the list of answers. **Don't turn in proofs**, but you should do them anyway, just for practice. Assume reasonable but nontrivial base cases. **If your solution requires specific base cases, state them.** Extra credit will be awarded for more exact solutions.

- (a) $A(n) = 2A(n/4) + \sqrt{n}$
- (b) $B(n) = 3B(n/3) + n/\lg n$
- (c) $C(n) = \frac{2C(n-1)}{C(n-2)}$ [Hint: This is easy!]
- (d) $D(n) = D(n-1) + 1/n$
- (e) $E(n) = E(n/2) + D(n)$
- (f) $F(n) = 4F\left(\left\lceil \frac{n-8}{2} \right\rceil + \left\lfloor \frac{3n}{\log_\pi n} \right\rfloor\right) + 6\binom{n+5}{2} - 42n \lg^7 n + \sqrt{13n-6} + \frac{\lg \lg n + 1}{\lg n \lg \lg \lg n}$
- (g) $G(n) = 2G(n-1) - G(n-2) + n$
- (h) $H(n) = 2H(n/2) - 2H(n/4) + 2^n$
- (i) $I(n) = I(n/2) + I(n/4) + I(n/6) + I(n/12) + n$
- ★(j) $J(n) = \sqrt{n} \cdot J(2\sqrt{n}) + n$
[Hint: First solve the secondary recurrence $j(n) = 1 + j(2\sqrt{n})$.]

3. The n th *Fibonacci binary tree* \mathcal{F}_n is defined recursively as follows:

- \mathcal{F}_1 is a single root node with no children.
- For all $n \geq 2$, \mathcal{F}_n is obtained from \mathcal{F}_{n-1} by adding a right child to every leaf and adding a left child to every node that has only one child.



The first six Fibonacci binary trees. In each tree \mathcal{F}_n , the subtree of gray nodes is \mathcal{F}_{n-1} .

- Prove that the number of leaves in \mathcal{F}_n is precisely the n th Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$.
- How many nodes does \mathcal{F}_n have? For full credit, give an *exact*, closed-form answer in terms of Fibonacci numbers, and prove your answer is correct.
- Prove that the left subtree of \mathcal{F}_n is a copy of \mathcal{F}_{n-2} .

4. Describe and analyze a data structure that stores set of n records, each with a numerical *key* and a numerical *priority*, such that the following operation can be performed quickly:

$\text{RANGETOP}(a, z) : \text{return the highest-priority record whose key is between } a \text{ and } z.$

For example, if the (key, priority) pairs are

$$(3, 1), (4, 9), (9, 2), (6, 3), (5, 8), (7, 5), (1, 4), (0, 7),$$

then $\text{RANGETOP}(2, 8)$ would return the record with key 4 and priority 9 (the second record in the list).

You may assume that no two records have equal keys or equal priorities, and that no record has a key equal to a or z . Analyze both the size of your data structure and the running time of your RANGETOP algorithm. For full credit, your data structure must be as small as possible and your RANGETOP algorithm must be as fast as possible.

[Hint: How would you compute the number of keys between a and z ? How would you solve the problem if you knew that a is always $-\infty$?]

5. Penn and Teller agree to play the following game. Penn shuffles a standard deck¹ of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs ($3\clubsuit$), at which point the remaining undrawn cards instantly burst into flames.

The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.² To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

- (a) What is the expected number of cards that Teller draws?
- (b) What is the expected *maximum* value among the cards Teller gives to Penn?
- (c) What is the expected *minimum* value among the cards Teller gives to Penn?
- (d) What is the expected number of cards that Teller gives to Penn?

Full credit will be given only for *exact* answers (with correct proofs, of course).

*6. [Extra credit]³

Lazy binary is a variant of standard binary notation for representing natural numbers where we allow each “bit” to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

- The lazy binary representation of zero is 0.
- Given the lazy binary representation of any non-negative integer n , we can construct the lazy binary representation of $n + 1$ as follows:
 - (a) increment the rightmost digit;
 - (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

Here are the first several natural numbers in lazy binary notation:

0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, 102101, 102110, ...

- (a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.
- (b) Prove that for any natural number N , the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

¹In a standard deck of 52 cards, each card has a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ and a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$, and every possible suit-value pair appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

²Specifically, he hurls them from the opposite side of the stage directly into the back of Penn’s right hand.

³The “I don’t know” rule does not apply to extra credit problems. There is no such thing as “partial extra credit”.

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 1

Due Tuesday, September 12, 2006 in 3229 Siebel Center

Starting with this homework, groups of up to three students can submit or present a single joint solution. If your group is submitting a written solution, please remember to **print the names, NetIDs, and aliases of every group member on every page**. Please remember to submit **separate, individually stapled** solutions to each of the problems.

1. Recall from lecture that a *subsequence* of a sequence A consists of a (not necessarily contiguous) collection of elements of A , arranged in the same order as they appear in A . If B is a subsequence of A , then A is a *supersequence* of B .
 - (a) Describe and analyze a **simple** recursive algorithm to compute, given two sequences A and B , the length of the *longest common subsequence* of A and B . For example, given the strings ALGORITHM and ALTRUISTIC, your algorithm would return 5, the length of the longest common subsequence ALRIT.
 - (b) Describe and analyze a **simple** recursive algorithm to compute, given two sequences A and B , the length of a *shortest common supersequence* of A and B . For example, given the strings ALGORITHM and ALTRUISTIC, your algorithm would return 14, the length of the shortest common supersequence ALGTORUISTHIMC.
 - (c) Let $|A|$ denote the length of sequence A . For any two sequences A and B , let $\text{lcs}(A, B)$ denote the length of the longest common subsequence of A and B , and let $\text{scs}(A, B)$ denote the length of the shortest common supersequence of A and B .
Prove that $|A| + |B| = \text{lcs}(A, B) + \text{scs}(A, B)$ for all sequences A and B . [Hint: There is a simple non-inductive proof.]

In parts (a) and (b), we are *not* looking for the most efficient algorithms, but for algorithms with simple and correct recursive structure.

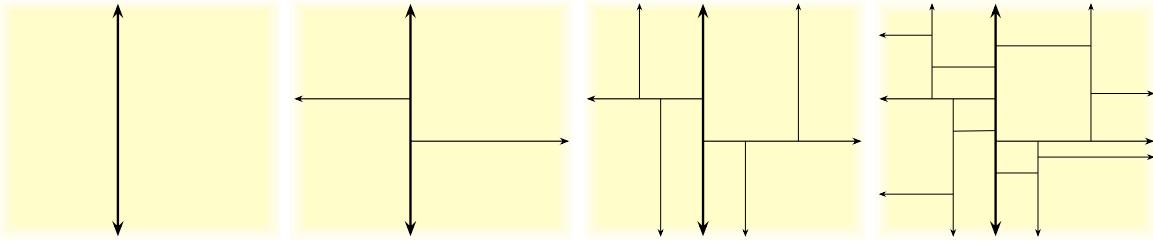
2. You are a contestant on a game show, and it is your turn to compete in the following game. You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.
 - (a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?
 - (b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?
 - (c) [**Extra credit**]¹ Prove that your solution to part (b) is asymptotically optimal.

¹The “I don’t know” rule does not apply to extra credit problems. There is no such thing as “partial extra credit”.

3. A kd-tree is a rooted binary tree with three types of nodes: horizontal, vertical, and leaf. Each vertical node has a *left* child and a *right* child; each horizontal node has a *high* child and a *low* child. The non-leaf node types alternate: non-leaf children of vertical nodes are horizontal and vice versa. Each non-leaf node v stores a real number p_v called its *pivot value*. Each node v has an associated *region* $R(v)$, defined recursively as follows:

- $R(\text{root})$ is the entire plane.
- If v is a horizontal node, the horizontal line $y = p_v$ partitions $R(v)$ into $R(\text{high}(v))$ and $R(\text{low}(v))$ in the obvious way.
- If v is a vertical node, the vertical line $x = p_v$ partitions $R(v)$ into $R(\text{left}(v))$ and $R(\text{right}(v))$ in the obvious way.

Thus, each region $R(v)$ is an axis-aligned rectangle, possibly with one or more sides at infinity. If v is a leaf, we call $R(v)$ a *leaf box*.



The first four levels of a typical kd-tree.

Suppose T is a perfectly balanced kd-tree with n leaves (and thus with depth exactly $\lg n$).

- Consider the horizontal line $y = t$, where $t \neq p_v$ for all nodes v in T . Exactly how many leaf boxes of T does this line intersect? [Hint: The parity of the root node matters.] Prove your answer is correct. A correct $\Theta(\cdot)$ bound is worth significant partial credit.
- Describe and analyze an efficient algorithm to compute, given T and an arbitrary horizontal line ℓ , the number of leaf boxes of T that lie entirely above ℓ .

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 2

Due Tuesday, September 19, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

1. You are given an $m \times n$ matrix M in which each entry is a 0 or 1. A *solid block* is a rectangular subset of M in which each entry is 1. Give a correct efficient algorithm to find a solid block in M with maximum area.

1	1	0	1	1
0	1	1	1	0
1	1	1	1	1
1	1	0	1	1

An algorithm that runs in $\Theta(n^c)$ time will earn $19 - 3c$ points.

2. You are a bus driver with a soda fountain machine in the back and a bus full of very hyper students, who are drinking more soda as they ride along the highway. Your goal is to drop the students off as quickly as possible. More specifically, every minute that a student is on your bus, he drinks another ounce of soda. Your goal is to drop the students off quickly, so that in total they drink as little soda as possible.

You know how many students will get off of the bus at each exit. Your bus begins partway along the highway (probably not at either end), and moves at a constant rate. You must drive the bus along the highway- however you may drive forward to one exit then backward to an exit in the other direction, switching as often as you like (you can stop the bus, drop off students, and turn around instantaneously).

Give an efficient algorithm to drop the students off so that they drink as little soda as possible. The input to the algorithm should be: the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (you may assume it is at an exit).

3. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the i th word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j , then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^j P_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 3

Due Wednesday, October 4, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

1. Consider a perfect tree of height h , where every non-leaf node has 3 children. (Therefore, each of the 3^h leaves is at distance h from the root.) Every leaf has a boolean value associated with it - either 0 or 1. Every internal node gets the boolean value assigned to the majority of its children. Given the values assigned to the leaves, we want to find an algorithm that computes the value (0 or 1) of the root.

It is not hard to find a (deterministic) algorithm that looks at every leaf and correctly determines the value of the root, but this takes $O(3^h)$ time. Describe and analyze a *randomized* algorithm that, on average, looks at asymptotically fewer leaves. That is, the expected number of leaves your algorithm examines should be $o(3^h)$.

2. We define a *meldable heap* to be a binary tree of elements, each of which has a priority, such that the priority of any node is less than the priority of its parent. (Note that the heap does **not** have to be balanced, and that the element with greatest priority is the root.) We also define the priority of a heap to be the priority of its root.

The *meld* operation takes as input two (meldable) heaps and returns a single meldable heap H that contains all the elements of both input heaps. We define *meld* as follows:

- Let H_1 be the input heap with greater priority, and H_2 the input heap with lower priority. (That is, the priority of $\text{root}(H_1)$ is greater than the priority of $\text{root}(H_2)$.) Let H_L be the left subtree of $\text{root}(H_1)$ and H_R be the right subtree of $\text{root}(H_1)$.
 - We set $\text{root}(H) = \text{root}(H_1)$.
 - We now flip a coin that comes up either “Left” or “Right” with equal probability.
 - If it comes up “Left”, we set the left subtree of $\text{root}(H)$ to be H_L , and the right subtree of $\text{root}(H)$ to be $\text{meld}(H_R, H_2)$ (defined recursively).
 - If the coin comes up “Right”, we set the right subtree of $\text{root}(H)$ to be H_R , and the left subtree of $\text{root}(H)$ to be $\text{meld}(H_L, H_2)$.
 - As a base case, melding any heap H_1 with an empty heap gives H_1 .
- (a) Analyze the expected running time of $\text{meld}(H_a, H_b)$ if H_a is a (meldable) heap with n elements, and H_b is a (meldable) heap with m elements.
- (b) Describe how to perform each of the following operations using only melds, and give the running time of each.
- $\text{DeleteMax}(H)$, which deletes the element with greatest priority.
 - $\text{Insert}(H, x)$, which inserts the element x into the heap H .
 - $\text{Delete}(H, x)$, which - given a pointer to element x in heap H - returns the heap with x deleted.

3. Randomized Selection. Given an (unsorted) array of n distinct elements and an integer k , SELECTION is the problem of finding the k th smallest element in the array. One easy solution is to sort the array in increasing order, and then look up the k th entry, but this takes $\Theta(n \log n)$ time. The randomized algorithm below attempts to do better, at least on average.

```

QuickSelect(Array A, n, k)
pivot ← Random(1, n)
S ← { $x \mid x \in A, x < A[pivot]$ }
s ← |S|
L ← { $x \mid x \in A, x > A[pivot]$ }
if ( $k = s + 1$ )
    return  $A[pivot]$ 
else if ( $k \leq s$ )
    return QuickSelect( $S, s, k$ )
else
    return QuickSelect( $L, n - (s + 1), k - (s + 1)$ )

```

Here we assume that $\text{Random}(a, b)$ returns an integer chosen uniformly at random from a to b (inclusive of a and b). The pivot position is randomly chosen; S is the set of elements smaller than the pivot element, and L the set of elements larger than the pivot. The sets S and L are found by comparing every other element of A to the pivot. We partition the elements into these two ‘halves’, and recurse on the appropriate half.

- (a) Write a recurrence relation for the expected running time of QuickSelect.
 - (b) Given any two elements $x, y \in A$, what is the probability that x and y will be compared?
 - (c) Either from part (a) or part (b), find the expected running time of QuickSelect.
4. [Extra Credit]: In the previous problem, we found a $\Theta(n)$ algorithm for selecting the k th smallest element, but the constant hidden in the $\Theta(\cdot)$ notation is somewhat large. It is easy to find the *smallest* element using at most n comparisons; we would like to be able to extend this to larger k . Can you find a randomized algorithm that uses $n + \Theta(k \log k \log n)$ ¹ expected comparisons? (Note that there is no constant multiplying the n .)

Hint: While scanning through a random permutation of n elements, how many times does the smallest element seen so far change? (See HBS 0.) How many times does the k th smallest element so far change?

¹There is an algorithm that uses $n + \Theta(k \log(n/k))$ comparisons, but this is even harder.

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 4

Due Tuesday, October 10, 2006 in 3229 Siebel Center

Remember to submit **separate, individually stapled** solutions to each of the problems.

1. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if every building to the east of i is shorter than i . We present an algorithm that computes which buildings have a good view of Lake Michigan. Use the taxation method of amortized analysis to bound the amortized time spent in each iteration of the for loop. What is the total runtime?

```
GOODVIEW( $A[1..n]$ ):  
    Initialize a stack  $S$   
    for  $i = 1$  to  $n$   
        while ( $S$  not empty and  $A[i] \geq A[S.\text{top}]$ )  
            POP( $S$ )  
        PUSH( $S, i$ )  
    return  $S$ 
```

2. Design and analyze a simple data structure that maintains a list of integers and supports the following operations.
 - (a) $\text{CREATE}()$: creates and returns a new list L
 - (b) $\text{PUSH}(L, x)$: appends x to the end of L
 - (c) $\text{POP}(L)$: deletes the last entry of L and returns it
 - (d) $\text{LOOKUP}(L, k)$: returns the k th entry of L

Your solution may use these primitive data structures: arrays, balanced binary search trees, heaps, queues, single or doubly linked lists, and stacks. If your algorithm uses anything fancier, you must give an explicit implementation. Your data structure should support all operations in amortized constant time. In addition, your data structure should support $\text{LOOKUP}()$ in worst-case $O(1)$ time. At all times, your data structure should use space which is linear in the number of objects it stores.

3. Consider a computer game in which players must navigate through a field of landmines, which are represented as points in the plane. The computer creates new landmines which the players must avoid. A player may ask the computer how many landmines are contained in any simple polygonal region; it is your job to design an algorithm which answers these questions efficiently.

You have access to an efficient static data structure which supports the following operations.

- $\text{CREATE}_S(\{p_1, p_2, \dots, p_n\})$: creates a new data structure S containing the points $\{p_1, \dots, p_n\}$. It has a worst-case running time of $T(n)$. Assume that $T(n)/n \geq T(n-1)/(n-1)$, so that the average processing time of elements does not decrease as n grows.
- $\text{DUMP}_S(S)$: destroys S and returns the set of points that S stored. It has a worst-case running time of $O(n)$, where n is the number of points in S .
- $\text{QUERY}_S(S, R)$: returns the number of points in S that are contained in the region R . It has a worst-case running time of $Q(n)$, where n is the number of points stored in S .

Unfortunately, the data structure does not support point insertion, which is required in your application. Using the given static data structure, design and analyze a dynamic data structure that supports the following operations.

- (a) $\text{CREATED}()$: creates a new data structure D containing no points. It should have a worst-case constant running time.
- (b) $\text{INSERT}_D(D, p)$: inserts p into D . It should run in amortized $O(\log n) \cdot T(n)/n$ time.
- (c) $\text{QUERY}_D(D, R)$: returns the number of points in D that are contained in the region R . It should have a worst-case running time of $O(\log n) \cdot Q(n)$.

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 5

Due Tuesday, October 24, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

1. Makefiles:

In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called ‘make’ that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files that must be compiled. Thus a source file must be recompiled if a file on which it depends is changed.

Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design and analyze an efficient algorithm to recompile only the necessary files. DO NOT worry about the details of parsing a Makefile.

2. Consider a graph G , with n vertices. Show that if any two of the following properties hold for G , then the third property must also hold.

- G is connected.
- G is acyclic.
- G has $n - 1$ edges.

3. The weight of a spanning tree is the sum of the weights on the edges of the tree. Given a graph, G , describe an efficient algorithm (the most efficient one you can) to find the k lightest (with least weight) spanning trees of G .

Analyze the running time of your algorithm. Be sure to prove your algorithm is correct.

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 6

Due Wednesday, November 8, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

1. Dijkstra's algorithm can be used to determine shortest paths on graphs with some negative edge weights (as long as there are no negative cycles), but the worst-case running time is much worse than the $O(E + V \log V)$ it takes when the edge weights are all positive. Construct an infinite family of graphs - with negative edge weights - for which the asymptotic running time of Dijkstra's algorithm is $\Omega(2^{|V|})$.
2. It's a cold and rainy night, and you have to get home from Siebel Center. Your car has broken down, and it's too windy to walk, which means you have to take a bus. To make matters worse, there is no bus that goes directly from Siebel Center to your apartment, so you have to change buses some number of times on your way home. Since it's cold outside, you want to spend as little time as possible waiting in bus shelters.

From a computer in Siebel Center, you can access an online copy of the MTD bus schedule, which lists bus routes and the arrival time of every bus at each stop on its route. Describe an algorithm which, given the schedule, finds a way for you to get home that minimizes the time you spend at bus shelters (the amount of time you spend on the bus doesn't matter). Since Siebel Center is warm and the nearest bus stop is right outside, you can assume that you wait inside Siebel until the first bus you want to take arrives outside. Analyze the efficiency of your algorithm and prove that it is correct.

3. The Floyd-Warshall all-pairs shortest path algorithm computes, for each $u, v \in V$, the shortest path from u to v . However, if the graph has negative cycles, the algorithm fails. Describe a modified version of the algorithm (*with the same asymptotic time complexity*) that correctly returns shortest-path distances, even if the graph contains negative cycles. That is, if there is a path from u to some negative cycle, and a path from that cycle to v , the algorithm should output $dist(u, v) = -\infty$. For any other pair u, v , the algorithm should output the length of the shortest directed path from u to v .

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 6

Due at **4 p.m.** on Friday, November 17, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

1. Given an undirected graph $G(V, E)$, with three vertices $u, v, w \in V$, you want to know whether there exists a path from u to w via v . (That is, the path from u to w must use v as an intermediate vertex.) Describe an efficient algorithm to solve this problem.

2. *Ad-hoc Networks*, made up of cheap, low-powered wireless devices, are often used on battle-fields, in regions that have recently suffered from natural disasters, and in other situations where people might want to monitor conditions in hard-to-reach areas. The idea is that a large collection of the wireless devices could be dropped into the area from an airplane (for instance), and then they could be configured into an efficiently functioning network.

Since the devices are cheap and low-powered, they frequently fail, and we would like our networks to be reliable. If a device detects that it is likely to fail, it should transmit the information it has to some other device (called a *backup*) within range of it. The range is limited; we assume that there is a distance d such that two devices can communicate if and only if they are within distance d of each other. To improve reliability, we don't want a device to transmit information to a neighbor that has already failed, and so we require each device v to have at least k backup devices that it could potentially contact, all of which must be within d meters of it. We call this the *backup set* of v . Also, we do not want any device to be in the backup set of too many other devices; if it were, and it failed, a large fraction of our network would be affected.

The input to our problem is a collection of n devices, and for each pair u, v of devices, the distance between u and v . We are also given the distance d that determines the range of a device, and parameters b and k . Describe an algorithm that determines if, for each device, we can find a backup set of size k , while also requiring that no device appears in the backup set of more than b other devices.

3. **UPDATED:** Given a piece of text T and a pattern P (the ‘search string’), an algorithm for the string-matching problem either finds the first occurrence of P in T , or reports that there is none. Modify the Knuth-Morris-Pratt (KMP) algorithm so that it solves the string-matching problem, even if the pattern contains the wildcards ‘?’ and ‘*’. Here, ‘?’ represents any *single* character of the text, and ‘*’ represents any substring of the text (including the empty substring). For example, the pattern “A?B*?A” matches the text “ABACBCABCACBA” starting in position 3 (in three different ways), and position 7 (in two ways). For this input, your algorithm would need to return ‘3’.

UPDATE: You may assume that the pattern you are trying to match containst at most 3 blocks of question marks; the usage of ‘*’ wildcards is still unrestricted. Here, a block refers to a string of consecutive ‘?’s in the pattern. For example, AAB??ACA??????BB contains 2 blocks of question marks; A?B?C?A?C contains 4 blocks of question marks.

4. In the two-dimensional pattern-matching problem, you are given an $m \times n$ matrix M and a $p \times q$ pattern P . You wish to find all positions (i, j) in M such that the the submatrix of M between rows i and $i + p - 1$ and between columns j and $j + q - 1$ is identical to P . (That is, the $p \times q$ sub-matrix of M below and to the right of position (i, j) should be identical to P .) Describe and analyze an efficient algorithm to solve this problem.¹

¹Note that the normal string-matching problem is the special case of the 2-dimensional problem where $m = p = 1$.

CS 473U: Undergraduate Algorithms, Fall 2006

Homework 8

Due Wednesday, December 6, 2006 in 3229 Siebel Center

Remember to submit **separate, individually stapled** solutions to each of the problems.

1. Given an array $A[1..n]$ of $n \geq 2$ distinct integers, we wish to find the second largest element using as few comparisons as possible.
 - (a) Give an algorithm which finds the second largest element and uses at most $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.
 - *(b) Prove that every algorithm which finds the second largest element uses at least $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.
2. Let R be a set of rectangles in the plane. For each point p in the plane, we say that the *rectangle depth* of p is the number of rectangles in R that contain p .
 - (a) (Step 1: Algorithm Design) Design and analyze a polynomial-time algorithm which, given R , computes the maximum rectangle depth.
 - (b) (Step 2: ???) Describe and analyze a polynomial-time reduction from the maximum rectangle depth problem to the maximum clique problem.
 - (c) (Step 3: Profit!) In 2000, the Clay Mathematics Institute described the Millennium Problems: seven challenging open problems which are central to ongoing mathematical research. The Clay Institute established seven prizes, each worth one million dollars, to be awarded to anyone who solves a Millennium problem. One of these problems is the $P = NP$ question. In (a), we developed a polynomial-time algorithm for the maximum rectangle depth problem. In (b), we found a reduction from this problem to an NP-complete problem. We know from class that if we find a polynomial-time algorithm for any NP-complete problem, then we have shown $P = NP$. Why hasn't Jeff used (a) and (b) to show $P = NP$ and become a millionaire?
3. Let G be a complete graph with integer edge weights. If C is a cycle in G , we say that the *cost* of C is the sum of the weights of edges in C . Given G , the traveling salesman problem (TSP) asks us to compute a Hamiltonian cycle of minimum cost. Given G , the traveling salesman cost problem (TSCP) asks us to compute the cost of a minimum cost Hamiltonian cycle. Given G and an integer k , the traveling salesman decision problem (TSDP) asks us to decide if there is a Hamiltonian cycle in G of cost at most k .
 - (a) Describe and analyze a polynomial-time reduction from TSP to TSCP.
 - (b) Describe and analyze a polynomial-time reduction from TSCP to TSDP.
 - (c) Describe and analyze a polynomial-time reduction from TSDP to TSP.

- (d) What can you conclude about the relative computational difficulty of TSP, TSCP, and TSDP?
4. Let G be a graph. A set S of vertices of G is a *dominating set* if every vertex in G is either in S or adjacent to a vertex in S . Show that, given G and an integer k , deciding if G contains a dominating set of size at most k is NP-complete.

1. Probability

- (a) n people have checked their hats with a hat clerk. The clerk is somewhat absent-minded and returns the hats uniformly at random (with no regard for whether each hat is returned to its owner). On average, how many people will get back their own hats?
 - (b) Let S be a uniformly random permutation of $\{1, 2, \dots, n-1, n\}$. As we move from the left to the right of the permutation, let X denote the smallest number seen so far. On average, how many different values will X take?
2. A *tournament* is a directed graph where each pair of distinct vertices u, v has either the edge uv or the edge vu (but not both). A *Hamiltonian path* is a (directed) path that visits each vertex of the (di)graph. Prove that every tournament has a Hamiltonian path.
3. Describe and analyze a data structure that stores a set of n records, each with a numerical *key*, such that the following operation can be performed quickly:

Foo(a): return the sum of the records with keys at least as large as a .

For example, if the keys are:

3 4 9 6 5 8 7 1 0

then **Foo(2)** would return 42, since 3, 4, 5, 6, 7, 8, 9 are all larger than 2 and $3 + 4 + 5 + 6 + 7 + 8 + 9 = 42$. You may assume that no two records have equal keys, and that no record has a key equal to a . Analyze both the size of your data structure and the running time of your **Foo** algorithm. Your data structure must be as small as possible and your **Foo** algorithm must be as fast as possible.

1. The Acme Company is planning a company party. In planning the party, each employee is assigned a *fun value* (a positive real number). The goal of the party planners is to maximize the total fun value (sum of the individual fun values) of the employees invited to the party. However, the planners are not allowed to invite both an employee and his direct boss. Given a tree containing the boss/underling structure of Acme, find the invitation list with the highest allowable fun value.

2. An *inversion* in an array A is a pair i, j such that $i < j$ and $A[i] > A[j]$. (In an n -element array, the number of inversions is between 0 and $\binom{n}{2}$.)

Find an efficient algorithm to count the number of inversions in an n -element array.

3. A *tromino* is a geometric shape made from three squares joined along complete edges. There are only two possible trominoes: the three component squares may be joined in a line or an L-shape.

- (a) Show that it is possible to cover all but one square of a 64×64 checkerboard using L-shape trominoes. (In your covering, each tromino should cover three squares and no square should be covered more than once.)
- (b) Show that you can leave *any* single square uncovered.
- (c) Can you cover all but one square of a 64×64 checkerboard using *line* trominoes? If so, which squares can you leave uncovered?

1. Moving on a Checkerboard

Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:

- 1) the square immediately above
- 2) the square that is one up and one to the left (but only if the checker is not already in the leftmost column)
- 3) the square that is one up and one to the right (but only if the checker is not already in the rightmost column)

Each time you move from square x to square y , you receive $p(x, y)$ dollars. You are given a list of the values $p(x, y)$ for each pair (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

Give an algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. You algorithm is free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination in order to maximize the number of dollars gathered along the way. What is the running time of your algorithm?

2. Maximizing Profit

You are given lists of values h_1, h_2, \dots, h_k and l_1, l_2, \dots, l_k . For each i you can choose $j_i = h_i$, $j_i = l_i$, or $j_i = 0$; the only catch is that if $j_i = h_i$ then j_{i-1} must be 0 (except for $i = 1$). Your goal is to maximize $\sum_{i=1}^k j_i$.

Give an efficient algorithm that returns the maximum possible value of $\sum_{i=1}^k j_i$.

3. Maximum alternating subsequence

An *alternating sequence* is a sequence a_1, a_2, \dots such that no three consecutive terms of the sequence satisfy $a_i > a_{i+1} > a_{i+2}$ or $a_i < a_{i+1} < a_{i+2}$.

Given a sequence, efficiently find the longest alternating subsequence it contains. What is the running time of your algorithm?

1. Championship Showdown

What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basketweaving! The World Champions will be decided by a best of $2n - 1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requires four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1 - p$ that Urbana will win.

Let $P(i, j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0, j) = 1$, $1 \leq j \leq n$, because Champaign needs no more victories to win. $P(i, 0) = 0$, $1 \leq i \leq n$, as Champaign cannot possibly win if Urbana already has. $P(0, 0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q , so

$$P(i, j) = p \cdot P(i - 1, j) + q \cdot P(i, j - 1)$$

for any $i \geq 1$ and $j \geq 1$.

Create and analyze an $O(n^2)$ -time dynamic programming algorithm that takes the parameters n, p , and q and returns the probability that Champaign will win the series (that is, calculate $P(n, n)$).

2. Making Change

Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \dots < c[n]$. (In the U.S., for example, $n = 6$ and the values are 1, 5, 10, 25, 50, and 100 cents.) Your beloved benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

Describe and analyze a dynamic programming algorithm to determine, given a target amount A and a sorted array $c[1..n]$ of coin values, the smallest number of coins needed to make A cents in change. You can assume that $c[1] = 1$, so that it is possible to make change for any amount A .

3. Knapsack

You are a thief, who is trying to choose the best collection of treasure (some subset of the n treasures, numbered 1 through n) to steal. The weight of item i is $w_i \in \mathbb{N}$ and the profit is $p_i \in \mathbb{R}$. Let $C \in \mathbb{N}$ be the maximum weight that your knapsack can hold. Your goal is to choose a subset of elements $S \subseteq \{1, 2, \dots, n\}$ that maximizes your total profit $P(S) = \sum_{i \in S} p_i$, subject to the constraint that the sum of the weights $W(S) = \sum_{i \in S} w_i$ is not more than C .

Give an algorithm that runs in time $O(Cn)$.

1. Randomized Edge Cuts

We will randomly partition the vertex set of a graph G into two sets S and T . The algorithm is to flip a coin for each vertex and with probability $1/2$, put it in S ; otherwise put it in T .

- (a) Show that the expected number of edges with one endpoint in S and the other endpoint in T is exactly half the edges in G .
- (b) Now say the edges have weights. What can you say about the sum of the weights of the edges with one endpoint in S and the other endpoint in T ?

2. Skip Lists

A *skip list* is built in layers. The bottom layer is an ordinary sorted linked list. Each higher layer acts as an “express lane” for the lists below, where an element in layer i appears in layer $i+1$ with some fixed probability p .

```

1
1-----4---6
1---3-4---6-----9
1-2-3-4-5-6-7-8-9-10

```

- (a) What is the probability a node reaches height h .
- (b) What is the probability any node is above $c \log n$ (for some fixed value of c)?
Compute the value explicitly when $p = 1/2$ and $c = 4$.
- (c) To search for an entry x , scan the top layer until you find the last entry y that is less than or equal to x . If $y < x$, drop down one layer and in this new layer (beginning at y) find the last entry that is less than or equal to x . Repeat this process (dropping down a layer, then finding the last entry less than or equal to x) until you either find x or reach the bottom layer and confirm that x is not in the skip list. What is the expected search time?
- (d) Describe an efficient method for insertion. What is the expected insertion time?

3. Clock Solitaire

In a standard deck of 52 cards, put 4 face-down in each of the 12 ‘hour’ positions around a clock, and 4 face-down in a pile in the center. Turn up a card from the center, and look at the number on it. If it’s number x , place the card face-up next to the face-down pile for x , and turn up the next card in the face-down pile for x (that is, the face-down pile corresponding to hour x). You win if, for each $\text{Ace} \leq x \leq \text{Queen}$, all four cards of value x are turned face-up before all four Kings (the center cards) are turned face-up.

What is the probability that you win a game of Clock Solitaire?

1. Simulating Queues with Stacks

A *queue* is a first-in-first-out data structure. It supports two operations *push* and *pop*. Push adds a new item to the back of the queue, while pop removes the first item from the front of the queue. A *stack* is a last-in-first-out data structure. It also supports push and pop. As with a queue, push adds a new item to the back of the queue. However, pop removes the last item from the back of the queue (the one most recently added).

Show how you can simulate a queue by using two stacks. Any sequence of pushes and pops should run in amortized constant time.

2. Multistacks

A *multistack* consists of an infinite series of stacks S_0, S_1, S_2, \dots , where the i th stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i , we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2} , and so on. To clarify, a user can only push elements onto S_0 . All other pushes and pops happen in order to make space to push onto S_0 . Moving a single element from one stack to the next takes $O(1)$ time.

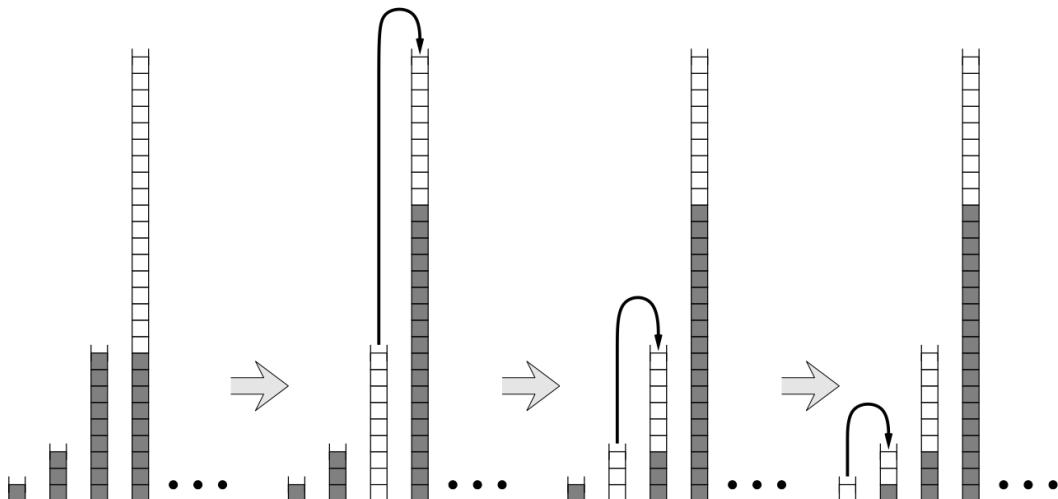


Figure 1. Making room for one new element in a multistack.

- (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?
- (b) Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack.

3. Powerhungry function costs

A sequence of n operations is performed on a data structure. The i th operation costs i if i is an exact power of 2, and 1 otherwise. Determine the amortized cost of the operation.

1. Representation of Integers

- (a) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.
- (b) Prove that any integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.

2. Minimal Dominating Set

Suppose you are given a rooted tree T (not necessarily binary). You want to label each node in T with an integer 0 or 1, such that every node either has the label 1 or is adjacent to a node with the label 1 (or both). The *cost* of a labeling is the number of nodes with label 1. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T .

3. Names in Boxes

The names of 100 prisoners are placed in 100 wooden boxes, one name to a box, and the boxes are lined up on a table in a room. One by one, the prisoners are led into the room; each may look in at most 50 boxes, but must leave the room exactly as he found it and is permitted no further communication with the others.

The prisoners have a chance to plot their strategy in advance, and they are going to need it, because unless *every single prisoner finds his own name* all will subsequently be executed. Find a strategy for them which has probability of success exceeding 30%. You may assume that the names are distributed in the boxes uniformly at random.

- (a) Calculate the probability of success if each prisoner picks 50 boxes uniformly at random.
- *(b) Consider the following strategy.

The prisoners number themselves 1 to 100. Prisoner i begins by looking in box i . There he finds the name of prisoner j . If $j \neq i$, he continues by looking in box j . As long as prisoner i has not found his name, he continues by looking in the box corresponding to the last name he found.

Describe the set of permutations of names in boxes for which this strategy will succeed.

- *(c) Count the number of permutations for which the strategy above succeeds. Use this sum to calculate the probability of success. You may find it useful to do this calculation for general n , then set $n = 100$ at the end.
- (d) We assumed that the names were distributed in the boxes uniformly at random. Explain how the prisoners could augment their strategy to make this assumption unnecessary.

1. Dynamic MSTs

Suppose that you already have a minimum spanning tree (MST) in a graph. Now one of the edge weights changes. Give an efficient algorithm to find an MST in the new graph.

2. Minimum Bottleneck Trees

In a graph G , for any pair of vertices u, v , let $\text{bottleneck}(u, v)$ be the maximum over all paths p_i from u to v of the minimum-weight edge along p_i . Construct a spanning tree T of G such that for each pair of vertices, their bottleneck in G is the same as their bottleneck in T .

One way to think about it is to imagine the vertices of the graph as islands, and the edges as bridges. Each bridge has a maximum weight it can support. If a truck is carrying stuff from u to v , how much can the truck carry? We don't care what route the truck takes; the point is that the smallest-weight edge on the route will determine the load.

3. Eulerian Tours

An *Eulerian tour* is a “walk along edges of a graph” (in which successive edges must have a common endpoint) that uses each edge exactly once and ends at the vertex where it starts. A graph is called Eulerian if it has an Eulerian tour.

Prove that a connected graph is Eulerian iff each vertex has even degree.

1. Alien Abduction

Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road won't be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t .

2. The Only SSSP Algorithm

In the lecture notes, Jeff mentions that all SSSP algorithms are special cases of the following generic SSSP algorithm. Each vertex v in the graph stores two values, which describe a tentative shortest path from s to v .

- $\text{dist}(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path.
- $\text{pred}(v)$ is the predecessor of v in the shortest $s \rightsquigarrow v$ path.

We call an edge *tense* if $\text{dist}(u) + w(u \rightarrow v) < \text{dist}(v)$. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

$$\boxed{\begin{array}{c} \text{Relax}(u \rightarrow v): \\ \hline \text{dist}(v) \leftarrow \text{dist}(u) + w(u \rightarrow v) \\ \text{pred}(v) \leftarrow u \end{array}}$$

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. The first of these is below. Prove it.

- When the algorithm halts, if $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of the predecessor chain ending at v :

$$s \rightarrow \dots \rightarrow (\text{pred}(\text{pred}(v))) \rightarrow \text{pred}(v) \rightarrow v.$$

3. Can't find a Cut-edge

A cut-edge is an edge which when deleted disconnects the graph. Prove or disprove the following. Every 3-regular graph has no cut-edge. (A common approach is induction.)

1. Max-Flow with vertex capacities

In a standard $s - t$ Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a node. In this problem, we consider the variant of Maximum-Flow and Minimum-Cut problems with node capacities.

More specifically, each node, n_i , has a capacity c_i . The edges have unlimited capacity. Show how you can model this problem as a standard Max-flow problem (where the weights are on the edges).

2. Emergency evacuation

Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be reashed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location.

At the same time, we don't want to overload any hospital by sending too many patients its way. We'd like to distribute the people so that each hospital receives at most $\lceil n/k \rceil$ people.

Show how to model this problem as a Max-flow problem.

3. Tracking a Hacker

A computer network (with each edge weight 1) is designed to carry traffic from a source s to a destination t . Recently, a computer hacker destroyed some of the edges in the graph. Normally, the maximum $s - t$ flow in G is k . Unfortunately, there is currently no path from s to t . Fortunately, the sysadmins know that the hacker destroyed at most k edges of the graph.

The sysadmins are trying to diagnose which of the nodes of the graph are no longer reachable. They would like to avoid testing each node. They are using a monitoring tool with the following behavior. If you use the command $\text{ping}(v)$, for a given node v , it will tell you whether there is currently a path from s to v (so $\text{ping}(t)$ will return `False` but $\text{ping}(s)$ will return `True`).

Give an algorithm that accomplishes this task using only $O(k \log n)$ pings. (You may assume that any algorithm you wish to run on the original network (before the hacker destroyed edges) runs for free, since you have a model of that network on your computer.)

1. Updating a maximum flow

Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e , a designated source $s \in V$, and a designated sink $t \in V$. You are also given a maximum $s - t$ flow in G , defined by a flow value f_e on each edge e . The flow $\{f_e\}$ is *acyclic*: There is no cycle in G on which all edges carry positive flow.

Now suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m + n)$, where m is the number of edges in G and n is the number of nodes.

2. Cooking Schedule

You live in a cooperative apartment with n other people. The co-op needs to schedule cooks for the next n days, so that each person cooks one day and each day there is one cook. In addition, each member of the co-op has a list of days they are available to cook (and is unavailable to cook on the other days).

Because of your superior CS473 skills, the co-op selects you to come up with a schedule for cooking, so that everyone cooks on a day they are available.

- (a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible schedule for the co-op.
- (b) A friend of yours tried to help you out by coming up with a cooking schedule. Unfortunately, when you look at the schedule he created, you notice a big problem. $n - 2$ of the people are scheduled for different nights on which they are available: no problem there. But the remaining two people are assigned to cook on the same night (and no one is assigned to the last night).

You want to fix your friend's mistake, but without having to recompute everything from scratch. Show that it's possible, using his "almost correct" schedule to decide in $O(n^2)$ time whether there exists a feasible schedule.

3. Disjoint paths in a digraph

Let $G = (V, E)$ be a directed graph, and suppose that for each node v , the number of edges into v is equal to the number of edges out of v . That is, for all v ,

$$|\{(u, v) : (u, v) \in E\}| = |\{(v, w) : (v, w) \in E\}|.$$

Let x, y be two nodes of G , and suppose that there exist k mutually edge-disjoint paths from x to y . Under these conditions, does it follow that there exist k mutually edge-disjoint paths from y to x . Give a proof or a counterexample with explanation.

1. String matching: an example

- (a) Build a finite automata to search for the string “bababoon”.
- (b) Use the automata from part (a) to build the prefix function for Knuth-Morris-Pratt.
- (c) Use the automata or the prefix function to search for “bababoon” in the string “babybaboonbuysbananasforotherbababoons”.

2. Cooking Schedule Strikes Back

You live in a cooperative apartment with n other people. The co-op needs to schedule cooks for the next $5n$ days, so that each person cooks five days and each day there is one cook. In addition, each member of the co-op has a list of days they are available to cook (and is unavailable to cook on the other days).

Because of your success at headbanging last week, the co-op again asks you to compose a cooking schedule. Unfortunately, you realize that no such schedule is possible Give a schedule for the cooking so that no one has to cook on more than 2 days that they claim to be unavailable.

3. String matching on Trees

You are given a rooted tree T (not necessarily binary), in which each node has a character. You are also given a pattern $P = p_1p_2 \cdots p_l$. Search for the string as a subtree. In other words, search for a subtree in which p_i is on a child of the node containing p_{i-1} for each $2 \leq i \leq l$.

1. Self-reductions

In each case below assume that you are given a black box which can answer the decision version of the indicated problem. Use a polynomial number of calls to the black box to construct the desired set.

- (a) Independent set: Given a graph G and an integer k , does G have a subset of k vertices that are pairwise nonadjacent?
- (b) Subset sum: Given a multiset (elements can appear more than once) $X = \{x_1, x_2, \dots, x_k\}$ of positive integers, and a positive integer S does there exist a subset of X with sum exactly S ?

2. Lower Bounds

Give adversary arguments to prove the indicated lower bounds for the following problems:

- (a) Searching in a sorted array takes at least $1 + \lfloor \lg_2 n \rfloor$ queries.
- (b) Let M be an $n \times n$ array of real values that is increasing in both rows and columns. Prove that searching for a value requires at least n queries.

3. k -coloring

Show that we can solve the problem of constructing a k -coloring of a graph by using a polynomial number of calls to a black box that determines whether a graph has such a k -coloring. (Hint: Try reducing via an intermediate problem that asks whether a partial coloring of a graph can be extended to a proper k -coloring.)

1. NP-hardness Proofs: Restriction

Prove that each of the following problems is NP-hard. In each part, find a special case of the given problem that is equivalent to a known NP-hard problem.

(a) **Longest Path**

Given a graph G and a positive integer k , does G contain a path with k or more edges?

(b) **Partition into Hamiltonian Subgraphs**

Given a graph G and a positive integer k , can the vertices of G be partitioned into at most k disjoint sets such that the graph induced by each set has a Hamiltonian cycle?

(c) **Set Packing**

Given a collection of finite sets C and a positive integer k , does C contain k disjoint sets?

(d) **Largest Common Subgraph**

Given two graphs G_1 and G_2 and a positive integer k , does there exist a graph G_3 such that G_3 is a subgraph of both G_1 and G_2 and G_3 has at least k edges?

2. Domino Line

You are given an unusual set of dominoes; each domino has a number on each end, but the numbers may be arbitrarily large and some numbers appear on many dominoes, while other numbers only appear on a few dominoes. Your goal is to form a line using all the dominoes so that adjacent dominoes have the same number on their adjacent halves. Either give an efficient algorithm to solve the problem or show that it is NP-hard.

3. Set Splitting

Given a finite set S and a collection of subsets C is there a partition of S into two sets S_1 and S_2 such that no subset in C is contained entirely in S_1 or S_2 ? Show that the problem is NP-hard. (Hint: use NAE-3SAT, which is similar to 3SAT except that a satisfying assignment does not allow all 3 variables in a clause to be true.)

You have 120 minutes to answer four of these five questions.

Write your answers in the separate answer booklet.

1. Multiple Choice.

Each of the questions on this page has one of the following five answers:

A: $\Theta(1)$	B: $\Theta(\log n)$	C: $\Theta(n)$	D: $\Theta(n \log n)$	E: $\Theta(n^2)$
----------------	---------------------	----------------	-----------------------	------------------

Choose the correct answer for each question. Each correct answer is worth +1 point; each incorrect answer is worth $-\frac{1}{2}$ point; each “I don’t know” is worth $+\frac{1}{4}$ point. Your score will be rounded to the nearest *non-negative* integer. You do *not* need to justify your answers; just write the correct letter in the box.

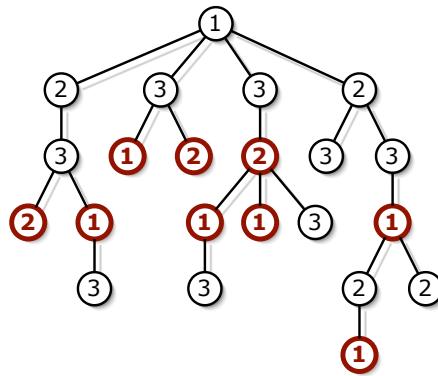
- (a) What is $\frac{5}{n} + \frac{n}{5}$?
- (b) What is $\sum_{i=1}^n \frac{n}{i}$?
- (c) What is $\sum_{i=1}^n \frac{i}{n}$?
- (d) How many bits are required to represent the n th Fibonacci number in binary?
- (e) What is the solution to the recurrence $T(n) = 2T(n/4) + \Theta(n)$?
- (f) What is the solution to the recurrence $T(n) = 16T(n/4) + \Theta(n)$?
- (g) What is the solution to the recurrence $T(n) = T(n - 1) + 1/n^2$?
- (h) What is the worst-case time to search for an item in a binary search tree?
- (i) What is the worst-case running time of quicksort?
- (j) What is the running time of the fastest possible algorithm to solve Sudoku puzzles?
A Sudoku puzzle consists of a 9×9 grid of squares, partitioned into nine 3×3 sub-grids; some of the squares contain digits between 1 and 9. The goal of the puzzle is to enter digits into the blank squares, so that each digit between 1 and 9 appears exactly once in each row, each column, and each 3×3 sub-grid. The initial conditions guarantee that the solution is unique.

2								4
	7		5					
			1		9			
6		4		2				
	8					5		
		9			3		7	
	1		4					
				3		8		
5							6	

A Sudoku puzzle. **Don't try to solve this during the exam!**

2. Oh, no! You have been appointed as the gift czar for Giggle, Inc.'s annual mandatory holiday party! The president of the company, who is certifiably insane, has declared that *every* Giggle employee must receive one of three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy. How do you decide what gifts everyone gets if you want to minimize the number of people that get fired?

More formally, suppose you are given a rooted tree T , representing the company hierarchy. You want to label each node in T with an integer 1, 2, or 3, such that every node has a different label from its parent.. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T . (Your algorithm does *not* have to compute the actual best labeling—just its cost.)



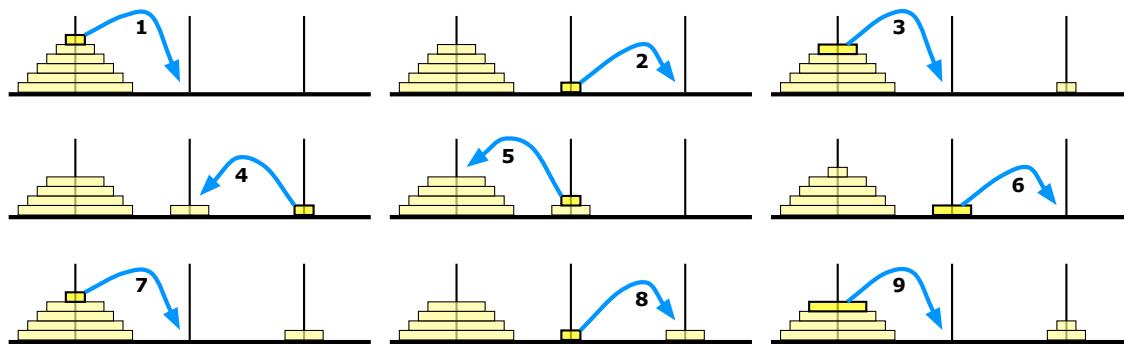
A tree labeling with cost 9. Bold nodes have smaller labels than their parents.
This is *not* the optimal labeling for this tree.

3. Suppose you are given an array $A[1..n]$ of n distinct integers, sorted in increasing order. Describe and analyze an algorithm to determine whether there is an index i such that $A[i] = i$, in $o(n)$ time. [Hint: Yes, that's **little-oh** of n . What can you say about the sequence $A[i] - i$?]
4. Describe and analyze a *polynomial-time* algorithm to compute the length of the longest common subsequence of two strings $A[1..m]$ and $B[1..n]$. For example, given the strings 'DYNAMIC' and 'PROGRAMMING', your algorithm would return the number 3, because the longest common subsequence of those two strings is 'AMI'. You must give a complete, self-contained solution; don't just refer to HW1.

5. Recall that the Tower of Hanoi puzzle consists of three pegs and n disks of different sizes. Initially, all the disks are on one peg, stacked in order by size, with the largest disk on the bottom and the smallest disk on top. In a single move, you can transfer the highest disk on any peg to a different peg, except that you may never place a larger disk on top of a smaller one. The goal is to move all the disks onto one other peg.

Now suppose the pegs are arranged in a row, and you are forbidden to transfer a disk directly between the left and right pegs in a single move; every move must involve the middle peg. How many moves suffice to transfer all n disks from the left peg to the right peg under this restriction? **Prove your answer is correct.**

For full credit, give an *exact* upper bound. A correct upper bound using $O(\cdot)$ notation (with a proof of correctness) is worth 7 points.



The first nine moves in a restricted Towers of Hanoi solution.

1. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, “Get out... while... you...”, thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can’t stand each other’s company, so you’ll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger’s heart, you wisely decide to leave the park immediately.

Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. *Be sure to account for both the vertex probabilities and the edge probabilities!*

2. Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader \bar{x} stores the number of elements of its set in the field $weight(\bar{x})$. Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

MAKESET(x):
$parent(x) \leftarrow x$
$weight(x) \leftarrow 1$

FIND(x):
while $x \neq parent(x)$
$x \leftarrow parent(x)$
return x

UNION(x, y)
$\bar{x} \leftarrow FIND(x)$
$\bar{y} \leftarrow FIND(y)$
if $weight(\bar{x}) > weight(\bar{y})$
$parent(\bar{y}) \leftarrow \bar{x}$
$weight(\bar{x}) \leftarrow weight(\bar{x}) + weight(\bar{y})$
else
$parent(\bar{x}) \leftarrow \bar{y}$
$weight(\bar{x}) \leftarrow weight(\bar{x}) + weight(\bar{y})$

Prove that if we use union-by-weight, the *worst-case* running time of FIND is $O(\log n)$.

3. *Prove or disprove*¹ each of the following statements.

- (a) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G includes the lightest edge in every cycle in G .
- (b) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G excludes the heaviest edge in every cycle in G .

4. In Homework 2, you were asked to analyze the following algorithm to find the k th smallest element from an unsorted array. (The algorithm is presented here in iterative form, rather than the recursive form you saw in the homework, but it's exactly the same algorithm.)

```

QUICKSELECT( $A[1..n], k$ ):
     $i \leftarrow 1; j \leftarrow n$ 
    while  $i \leq j$ 
         $r \leftarrow \text{PARTITION}(A[i..j], \text{RANDOM}(i, j))$ 
        if  $r = k$ 
            return  $A[r]$ 
        else if  $r > k$ 
             $j \leftarrow r - 1$ 
        else
             $i \leftarrow r + 1$ 

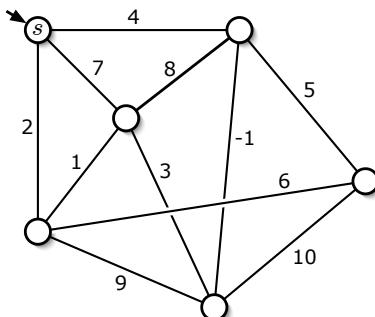
```

The algorithm relies on two subroutines. $\text{RANDOM}(i, j)$ returns an integer chosen uniformly at random from the range $[i..j]$. $\text{PARTITION}(A[i..j], p)$ partitions the subarray $A[i..j]$ using the pivot value $A[p]$ and returns the new index of the pivot value in the partitioned array.

What is the *exact* expected number of iterations of the main loop when $k = 1$? *Prove* your answer is correct. A correct $\Theta(\cdot)$ bound (with proof) is worth 7 points. You may assume that the input array $A[]$ contains n distinct integers.

5. Find the following spanning trees for the weighted graph shown below.

- (a) A breadth-first spanning tree rooted at s .
- (b) A depth-first spanning tree rooted at s .
- (c) A shortest-path tree rooted at s .
- (d) A minimum spanning tree.



You do *not* need to justify your answers; just clearly indicate the edges of each spanning tree. Yes, one of the edges has negative weight.

¹But not both! If you give us *both* a proof *and* a disproof for the same statement, you will get no credit, even if one of your arguments is correct.

1. A *double-Hamiltonian* circuit in an undirected graph G is a closed walk that visits every vertex in G exactly *twice*, possibly by traversing some edges more than once. *Prove* that it is NP-hard to determine whether a given undirected graph contains a double-Hamiltonian circuit.

2. Suppose you are running a web site that is visited by the same set of people every day. Each visitor claims membership in one or more *demographic groups*; for example, a visitor might describe himself as male, 31-40 years old, a resident of Illinois, an academic, a blogger, a Joss Whedon fan¹, and a Sports Racer.² Your site is supported by advertisers. Each advertiser has told you which demographic groups should see its ads and how many of its ads you must show each day. Altogether, there are n visitors, k demographic groups, and m advertisers.

Describe an efficient algorithm to determine, given all the data described in the previous paragraph, whether you can show each visitor exactly *one* ad per day, so that every advertiser has its desired number of ads displayed, and every ad is seen by someone in an appropriate demographic group.

3. Describe and analyze a data structure to support the following operations on an array $X[1..n]$ as quickly as possible. Initially, $X[i] = 0$ for all i .

- Given an index i such that $X[i] = 0$, set $X[i]$ to 1.
- Given an index i , return $X[i]$.
- Given an index i , return the smallest index $j \geq i$ such that $X[j] = 0$, or report that no such index exists.

For full credit, the first two operations should run in *worst-case constant* time, and the amortized cost of the third operation should be as small as possible. [Hint: Use a modified union-find data structure.]

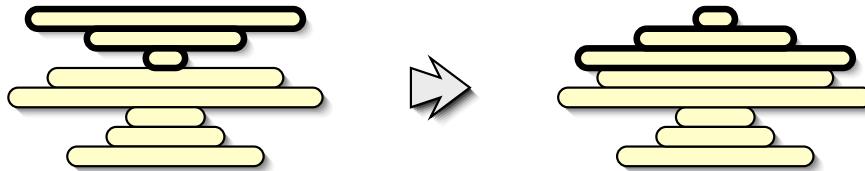
4. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbledypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbledypeg (fixed during the Holy Roman Three-Way Mumbledypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of partygoers know each other and ask you to choose the teams, while he sharpens the knife.

Either describe and analyze a polynomial time algorithm to determine whether the partygoers can be split into three legal Three-Way Mumbledypeg teams, or prove that the problem is NP-hard.

¹Har har har! Mine is an evil laugh! Now die!

²It's Ride the Fire Eagle Danger Day!

5. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n , and flip them all over.



Flipping the top three pancakes.

- (a) Describe an efficient algorithm to sort an arbitrary stack of n pancakes. *Exactly* how many flips does your algorithm perform in the worst case? (For full credit, your algorithm should perform as few flips as possible; an optimal $\Theta()$ bound is worth three points.)
 - (b) Now suppose one side of each pancake is burned. *Exactly* how many flips do you need to sort the pancakes *and* have the burned side of every pancake on the bottom? (For full credit, your algorithm should perform as few flips as possible; an optimal $\Theta()$ bound is worth three points.)
6. Describe and analyze an efficient algorithm to find the length of the longest substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples:

- Given the input string ALGORITHM, your algorithm should return 0.
- Given the input string RECURSION, your algorithm should return 1, for the substring R.
- Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
- Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM.

For full credit, your algorithm should run in $O(n^2)$ time.

7. A *double-Eulerian* circuit in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Describe and analyze a *polynomial-time* algorithm to determine whether a given undirected graph contains a double-Eulerian circuit.

CS 473G: Graduate Algorithms, Spring 2007

Homework 0

Due in class at 11:00am, Tuesday, January 30, 2007

Name:	
Net ID:	Alias:

I understand the Course Policies.

-
- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and staple this page to your solution to problem 1. **By providing an alias, you agree to let us list your grades; if you do not provide an alias, your grades will not be listed.** For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid!) your Social Security number. Please use the same alias for every homework and exam.
 - Read the Course Policies on the course web site, and then check the box above. Among other things, this page describes what we expect in your homework solutions, as well as policies on grading standards, regrading, extra credit, and plagiarism. In particular:
 - Submit each numbered problem separately, on its own piece(s) of paper. If you need more than one page for a problem, staple just *those* pages together, but keep different problems separate. **Do not staple your entire homework together.**
 - You may use *any* source at your disposal—paper, electronic, or human—but you *must* write your answers in your own words, and you *must* cite every source that you use.
 - Algorithms or proofs containing phrases like “and so on” or “repeat this for all n ”, instead of an explicit loop, recursion, or induction, are worth zero points.
 - Answering “I don’t know” to any homework or exam problem is worth 25% partial credit.

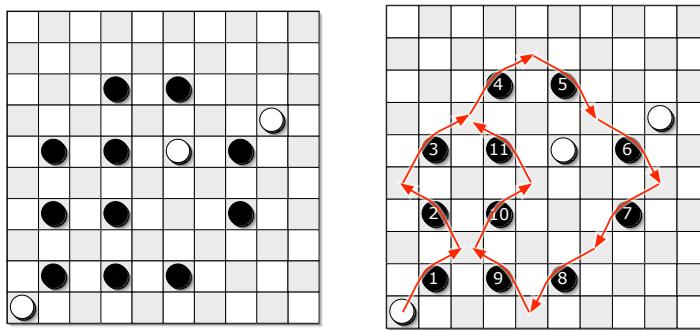
If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup.

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, discrete probability, graphs, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** The early chapters of Kleinberg and Tardos (or any algorithms textbook) should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks.
 - Every homework will have five problems, each worth 10 points. Stars indicate more challenging problems. Many homeworks will also include an extra-credit problem.
-

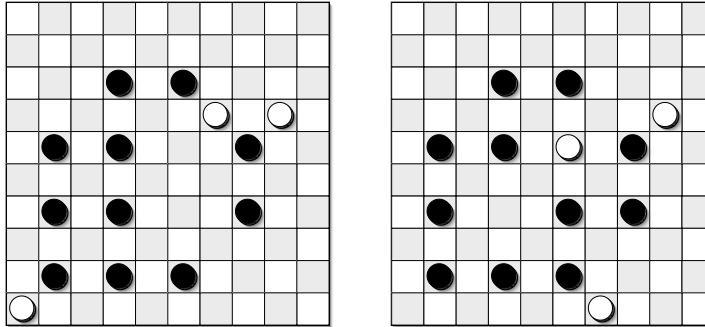
*1. Draughts/checkers is a game played on an $m \times m$ grid of squares, alternately colored light and dark. (The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size.) Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player (“White”) moves the white pieces; the other (“Black”) moves the black pieces.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.¹ Pieces can be moved in any of the four diagonal directions, either one or two steps at a time. On each turn, a player either moves one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her checkers. In a single jump, a piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. Multiple jumps are allowed in a single turn as long as they are made by the same piece. A player wins if her opponent has no pieces left on the board.

Describe an algorithm² that correctly determines whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces, but any algorithm that runs in time polynomial in n and m is worth significant partial credit.



White wins in one turn.



White cannot win in one turn from either of these positions.

[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does **not** always find a winning sequence of jumps even when one exists.]

¹Most variants of draughts have ‘flying kings’, which behave very differently than what’s described here.

²Since you’ve read the Course Policies, you know what this phrase means.

2. (a) Prove that any positive integer can be written as the sum of distinct powers of 2. [*Hint: “Write the number in binary” is **not** a proof; it just restates the problem.*] For example:

$$\begin{aligned}16 + 1 &= 17 = 2^4 + 2^0 \\16 + 4 + 2 + 1 &= 23 = 2^4 + 2^2 + 2^1 + 2^0 \\32 + 8 + 1 &= 42 = 2^5 + 2^3 + 2^1\end{aligned}$$

- (b) Prove that *any* integer (positive, negative, or zero) can be written as the sum of distinct powers of -2 . For example:

$$\begin{aligned}-32 + 16 - 2 + 1 &= -17 = (-2)^5 + (-2)^4 + (-2)^1 + (-2)^0 \\64 - 32 - 8 - 2 + 1 &= 23 = (-2)^6 + (-2)^5 + (-2)^3 + (-2)^1 + (-2)^0 \\64 - 32 + 16 - 8 + 4 - 2 &= 42 = (-2)^6 + (-2)^5 + (-2)^4 + (-2)^3 + (-2)^2 + (-2)^1\end{aligned}$$

3. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left.

4. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

Rosencrantz: H H T

Guildenstern: H T H H

Rosencrantz: T

Guildenstern: (no flips)

Rosencrantz: H H H T

Guildenstern: T H H T H H T H T H H H

- (a) What is the expected number of flips in one of Rosencrantz’s turns?
- (b) Suppose Rosencrantz flips k heads in a row on his turn. What is the expected number of flips in Guildenstern’s next turn?
- (c) What is the expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

Prove that your answers are correct. If you have to appeal to “intuition” or “common sense”, your answer is almost certainly wrong! You must give *exact* answers for full credit, but a correct asymptotic bound for part (b) is worth significant credit.

5. (a) [5 pts] Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases. If your solution requires a particular base case, say so.

$$A(n) = 3A(n/9) + \sqrt{n}$$

$$B(n) = 4B(n-1) - 4B(n-2)$$

$$C(n) = \frac{\pi C(n-1)}{\sqrt{2} C(n-2)}$$

[Hint: This is easy!]

$$D(n) = \max_{n/4 < k < 3n/4} (D(k) + D(n-k) + n)$$

$$E(n) = 2E(n/2) + 4E(n/3) + 2E(n/6) + n^2$$

Do not turn in proofs—just a list of five functions—but you should do them anyway, just for practice. [Hint: On the course web page, you can find a handout describing several techniques for solving recurrences.]

- (b) [5 pts] Sort the functions in the box from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do not turn in proofs**—just a sorted list of 16 functions—but you should do them anyway, just for practice.

To simplify your answer, write $f(n) \ll g(n)$ to indicate that $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

n	$\lg n$	\sqrt{n}	3^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$3\sqrt{n}$	$\sqrt{3^n}$
$3^{\lg n}$	$\lg(3^n)$	$3^{\lg \sqrt{n}}$	$3^{\sqrt{\lg n}}$
$\sqrt{3^{\lg n}}$	$\lg(3^{\sqrt{n}})$	$\lg \sqrt{3^n}$	$\sqrt{\lg(3^n)}$

Recall that $\lg n = \log_2 n$.

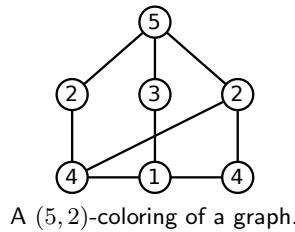
CS 473G: Graduate Algorithms, Spring 2007

Homework 1

Due February 6, 2007

Remember to submit **separate, individually stapled** solutions to each of the problems.

1. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire to students which lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of “strongly favor”, “mostly neutral”, or “strongly oppose”. Each student may respond with “strongly favor” or “strongly oppose” to at most five questions. Because Jeff’s students are very understanding, each student is happy if he or she prevails in just one of his or her strong policy preferences. Either describe a polynomial time algorithm for setting course policy to maximize the number of happy students or show that the problem is NP-hard.
2. Consider a variant 3SAT' of 3SAT which asks, given a formula ϕ in conjunctive normal form in which each clause contains at most 3 literals and each variable appears in at most 3 clauses, is ϕ satisfiable? Prove that 3SAT' is NP-complete.
3. For each problem below, either describe a polynomial-time algorithm to solve the problem or prove that the problem is NP-complete.
 - (a) A *double-Eulerian* circuit in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G , does G have a double-Eulerian circuit?
 - (b) A *double-Hamiltonian* circuit in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G , does G have a double-Hamiltonian circuit?
4. Suppose you have access to a magic black box; if you give it a graph G as input, the black box will tell you, in constant time, if there is a proper 3-coloring of G . Describe a polynomial time algorithm which, given a graph G that is 3-colorable, uses the black box to compute a 3-coloring of G .
5. Let C_5 be the graph which is a cycle on five vertices. A $(5, 2)$ -coloring of a graph G is a function $f : V(G) \rightarrow \{1, 2, 3, 4, 5\}$ such that every pair $\{u, v\}$ of adjacent vertices in G is mapped to a pair $\{f(u), f(v)\}$ of vertices in C_5 which are at distance two from each other.



Using a reduction from 5COLOR, prove that the problem of deciding whether a given graph G has a $(5, 2)$ -coloring is NP-complete.

CS 473G: Graduate Algorithms, Spring 2007

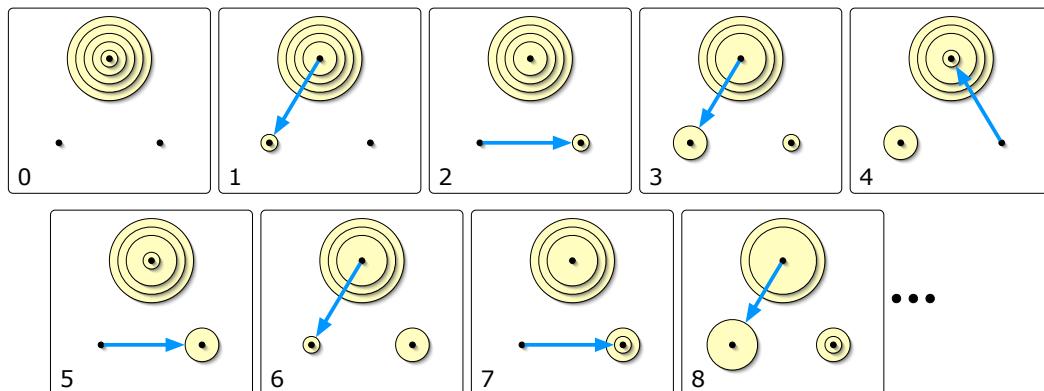
Homework 2

Due Tuesday, February 20, 2007

Remember to submit **separate, individually stapled** solutions to each problem.

As a general rule, a complete full-credit solution to any homework problem should fit into two typeset pages (or five hand-written pages). If your solution is significantly longer than this, you may be including too much detail.

1. Consider a restricted variant of the Tower of Hanoi puzzle, where the three needles are arranged in a triangle, and you are required to move each disk *counterclockwise*. Describe an algorithm to move a stack of n disks from one needle to another. *Exactly* how many moves does your algorithm perform? To receive full credit, your algorithm must perform the minimum possible number of moves. [Hint: Your answer will depend on whether you are moving the stack clockwise or counterclockwise.]



A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

- *2. You find yourself working for The Negation Company (“We Contradict Everything...Not!”), the world’s largest producer of multi-bit Boolean inverters. Thanks to a recent mining discovery, the market prices for amphigen and opoterium, the key elements used in AND and OR gates, have plummeted to almost nothing. Unfortunately, the market price of inverton, the essential element required to build NOT gates, has recently risen sharply as natural supplies are almost exhausted. Your boss is counting on you to radically redesign the company’s only product in response to these radically new market prices.

Design a Boolean circuit that inverts $n = 2^k - 1$ bits, using only k NOT gates but *any* number of AND and OR gates. The input to your circuit consists of n bits x_1, x_2, \dots, x_n , and the output consists of n bits y_1, y_2, \dots, y_n , where each output bit y_i is the inverse of the corresponding input bit x_i . [Hint: Solve the case $k = 2$ first.]

3. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays. A *common supersequence* of X and Y is another sequence that contains both X and Y as subsequences. Give a simple recursive definition for the function $scs(X, Y)$, which gives the length of the *shortest* common supersequence of X and Y .
(b) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i , and $X[i] > X[i+1]$ for all odd i . Give a simple recursive definition for the function $los(X)$, which gives the length of the longest oscillating subsequence of an arbitrary array X of integers.
(c) Call a sequence $X[1..n]$ of integers *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i . Give a simple recursive definition for the function $lxs(X)$, which gives the length of the longest accelerating subsequence of an arbitrary array X of integers.

Each recursive definition should translate directly into a recursive algorithm, *but you do not need to analyze these algorithms*. We are looking for correctness and *simplicity*, not algorithmic efficiency. Not yet, anyway.

4. Describe an algorithm to solve 3SAT in time $O(\phi^n \text{ poly}(n))$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618034$.
[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]
5. (a) Describe an algorithm that determines whether a given set of n integers contains two distinct elements that sum to zero, in $O(n \log n)$ time.
(b) Describe an algorithm that determines whether a given set of n integers contains three distinct elements that sum to zero, in $O(n^2)$ time.
(c) Now suppose the input set X contains n integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three **distinct** elements that sum to zero, in $O(n \log n)$ time.

For example, if the input set is $\{-10, -9, -7, -3, 1, 3, 5, 11\}$, your algorithm for part (a) should return TRUE, because $(-3) + 3 = 0$, and your algorithms for parts (b) and (c) should return FALSE, even though $(-10) + 5 + 5 = 0$.

CS 473G: Graduate Algorithms, Spring 2007

Homework 3

Due Friday, March 9, 2007

Remember to submit **separate, individually stapled** solutions to each problem.

As a general rule, a complete, full-credit solution to any homework problem should fit into two typeset pages (or five hand-written pages). If your solution is significantly longer than this, you may be including too much detail.

1. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays. A *common supersequence* of X and Y is another sequence that contains both X and Y as subsequences. Describe and analyze an efficient algorithm to compute the function $scs(X, Y)$, which gives the length of the shortest common supersequence of X and Y .
- (b) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i , and $X[i] > X[i+1]$ for all odd i . Describe and analyze an efficient algorithm to compute the function $los(X)$, which gives the length of the longest oscillating subsequence of an arbitrary array X of integers.
- (c) Call a sequence $X[1..n]$ of integers *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i . Describe and analyze an efficient algorithm to compute the function $lcs(X)$, which gives the length of the longest accelerating subsequence of an arbitrary array X of integers.

[Hint: Use the recurrences you found in Homework 2. You do not need to prove again that these recurrences are correct.]

2. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \text{poly}(n))$ time. Given an undirected n -vertex graph G with weighted edges, your algorithm should return the weight of the lightest Hamiltonian cycle in G (or ∞ if G has no Hamiltonian cycles).
3. Let G be an arbitrary undirected graph. A set of cycles $\{c_1, \dots, c_k\}$ in G is *redundant* if it is non-empty and every edge in G appears in an even number of c_i 's. A set of cycles is *independent* if it contains no redundant subsets. (In particular, the empty set is independent.) A maximal independent set of cycles is called a *cycle basis* for G .

- (a) Let C be any cycle basis for G . Prove that for any cycle γ in G **that is not an element of C** , there is a subset $A \subseteq C$ such that $A \cup \{\gamma\}$ is redundant. In other words, prove that γ is the ‘exclusive or’ of some subset of basis cycles.

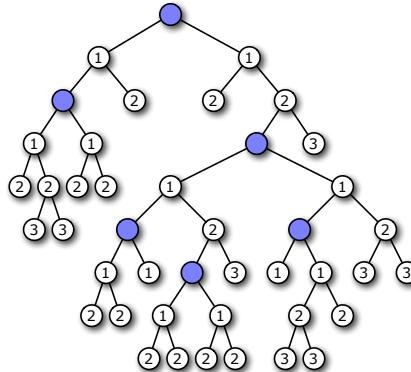
Solution: The claim follows directly from the definitions. A cycle basis is a *maximal* independent set, so if C is a cycle basis, then for any cycle $\gamma \notin C$, the larger set $C \cup \{\gamma\}$ cannot be an independent set, so it must contain a redundant subset. On the other hand, if C is a basis, then C is independent, so C contains no redundant subsets. Thus, $C \cup \{\gamma\}$ must have a redundant subset B that contains γ . Let $A = B \setminus \{\gamma\}$. ■

- (b) Prove that the set of independent cycle sets form a matroid.
- (c) Now suppose each edge of G has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a set of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis of G .
4. Let T be a rooted binary tree with n vertices, and let $k \leq n$ be a positive integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. More formally, we define the *clustering cost* of a clustering of any subset K of vertices as

$$\text{cost}(K) = \max_v \text{cost}(v, K),$$

where the maximum is taken over all vertices v in the tree, and

$$\text{cost}(v, K) = \begin{cases} 0 & \text{if } v \in K \\ \infty & \text{if } v \text{ is the root of } T \text{ and } v \notin K \\ 1 + \text{cost}(\text{parent}(v)) & \text{otherwise} \end{cases}$$

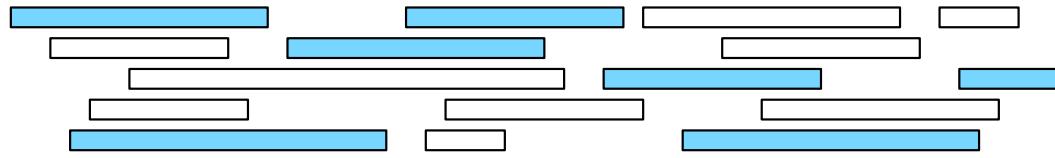


A subset of 5 vertices with clustering cost 3

Describe and analyze a dynamic-programming algorithm to compute the minimum clustering cost of any subset of k vertices in T . For full credit, your algorithm should run in $O(n^2k^2)$ time.

5. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The *size* of a tiling cover is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X . If you use a greedy algorithm, you must prove that it is correct.



CS 473G: Graduate Algorithms, Spring 2007

Homework 4

Due March 29, 2007

Please remember to submit **separate, individually stapled** solutions to each problem.

1. Given a graph G with edge weights and an integer k , suppose we wish to partition the vertices of G into k subsets S_1, S_2, \dots, S_k so that the sum of the weights of the edges that cross the partition (*i.e.*, have endpoints in different subsets) is as large as possible.
 - (a) Describe an efficient $(1 - 1/k)$ -approximation algorithm for this problem.
 - (b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.
2. In class, we saw a $(3/2)$ -approximation algorithm for the metric traveling salesman problem. Here, we consider computing minimum cost Hamiltonian *paths*. Our input consists of a graph G whose edges have weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.
 - (a) If our input includes zero endpoints, describe a $(3/2)$ -approximation to the problem of computing a minimum cost Hamiltonian path.
 - (b) If our input includes one endpoint u , describe a $(3/2)$ -approximation to the problem of computing a minimum cost Hamiltonian path that starts at u .
 - (c) If our input includes two endpoints u and v , describe a $(5/3)$ -approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v .
3. Consider the greedy algorithm for metric TSP: start at an arbitrary vertex u , and at each step, travel to the closest unvisited vertex.
 - (a) Show that the greedy algorithm for metric TSP is an $O(\log n)$ -approximation, where n is the number of vertices. [*Hint: Argue that the k th least expensive edge in the tour output by the greedy algorithm has weight at most $\text{OPT}/(n - k + 1)$; try $k = 1$ and $k = 2$ first.*]
 - *(b) **[Extra Credit]** Show that the greedy algorithm for metric TSP is no better than an $O(\log n)$ -approximation.
4. In class, we saw that the greedy algorithm gives an $O(\log n)$ -approximation for vertex cover. Show that our analysis of the greedy algorithm is asymptotically tight by describing, for any positive integer n , an n -vertex graph for which the greedy algorithm produces a vertex cover of size $\Omega(\log n) \cdot \text{OPT}$.

5. Recall the minimum makespan scheduling problem: Given an array $T[1..n]$ of processing times for n jobs, we wish to schedule the jobs on m machines to minimize the time at which the last job terminates. In class, we proved that the greedy scheduling algorithm has an approximation ratio of at most 2.

- (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2 - 1/m)$ times the makespan of the optimal assignment.
- (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2 - 1/m)$ times the makespan of the optimal assignment.
- (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

CS 473G: Graduate Algorithms, Spring 2007

Homework 5

Due Thursday, April 17, 2007

Please remember to submit **separate, individually stapled** solutions to each problem.

Unless a problem specifically states otherwise, you can assume the function $\text{RANDOM}(k)$, which returns an integer chosen independently and uniformly at random from the set $\{1, 2, \dots, k\}$, in $O(1)$ time. For example, to perform a fair coin flip, you would call $\text{RANDOM}(2)$.

1. Suppose we want to write an efficient function $\text{RANDOMPERMUTATION}(n)$ that returns a permutation of the integers $\langle 1, \dots, n \rangle$ chosen uniformly at random.

- (a) What is the expected running time of the following RANDOMPERMUTATION algorithm?

```
RANDOMPERMUTATION( $n$ ):  
    for  $i \leftarrow 1$  to  $n$   
         $\pi[i] \leftarrow \text{EMPTY}$   
    for  $i \leftarrow 1$  to  $n$   
         $j \leftarrow \text{RANDOM}(n)$   
        while ( $\pi[j] \neq \text{EMPTY}$ )  
             $j \leftarrow \text{RANDOM}(n)$   
         $\pi[j] \leftarrow i$   
    return  $\pi$ 
```

- (b) Consider the following partial implementation of RANDOMPERMUTATION .

```
RANDOMPERMUTATION( $n$ ):  
    for  $i \leftarrow 1$  to  $n$   
         $A[i] \leftarrow \text{RANDOM}(n)$   
     $\pi \leftarrow \text{SOMEFUNCTION}(A)$   
    return  $\pi$ 
```

Prove that if the subroutine SOMEFUNCTION is deterministic, then this algorithm cannot be correct. [Hint: There is a one-line proof.]

- (c) Describe and analyze an RANDOMPERMUTATION algorithm whose expected worst-case running time is $O(n)$.
- *(d) **[Extra Credit]** Describe and analyze an RANDOMPERMUTATION algorithm that uses only fair coin flips; that is, your algorithm can't call $\text{RANDOM}(k)$ with $k > 2$. Your algorithm should run in $O(n \log n)$ time with high probability.

2. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue containing the empty set.
- **FINDMIN(Q)**: Return the smallest element of Q (if any).
- **DELETEMIN(Q)**: Remove the smallest element in Q (if any).
- **INSERT(Q, x)**: Insert element x into Q , if it is not already there.
- **DECREASEKEY(Q, x, y)**: Replace an element $x \in Q$ with a smaller element y . (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q that contains x .
- **DELETE(Q, x)**: Delete the element $x \in Q$. The input is a pointer directly to the node in Q that contains x .
- **MELD(Q_1, Q_2)**: Return a new priority queue containing all the elements of Q_1 and Q_2 ; this operation destroys Q_1 and Q_2 .

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```

MELD( $Q_1, Q_2$ ):
  if  $Q_1$  is empty, return  $Q_2$ 
  if  $Q_2$  is empty, return  $Q_1$ 
  if  $key(Q_1) > key(Q_2)$ 
    swap  $Q_1 \leftrightarrow Q_2$ 
  with probability  $1/2$ 
     $left(Q_1) \leftarrow \text{MELD}(left(Q_1), Q_2)$ 
  else
     $right(Q_1) \leftarrow \text{MELD}(right(Q_1), Q_2)$ 
  return  $Q_1$ 
```

- (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of $\text{MELD}(Q_1, Q_2)$ is $O(\log n)$, where $n = |Q_1| + |Q_2|$. [Hint: How long is a random root-to-leaf path in an n -node binary tree if each left/right choice is made with equal probability?]
 - (b) Prove that $\text{MELD}(Q_1, Q_2)$ runs in $O(\log n)$ time with high probability.
 - (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)
3. Prove that GUESSMINCUT returns the *second* smallest cut in its input graph with probability $\Omega(1/n^2)$. (The second smallest cut could be significantly larger than the minimum cut.)

4. A *heater* is a sort of dual treap, in which the priorities of the nodes are given by the user, but their search keys are random (specifically, independently and uniformly distributed in the unit interval $[0, 1]$).

- (a) Prove that for any r , the node with the r th smallest priority has expected depth $O(\log r)$.
- (b) Prove that an n -node heater has depth $O(\log n)$ with high probability.
- (c) Describe algorithms to perform the operations INSERT and DELETEMIN in a heater. What are the expected worst-case running times of your algorithms?

You may assume all priorities and keys are distinct. [Hint: Cite the relevant parts (but only the relevant parts!) of the treap analysis instead of repeating them.]

5. Let n be an arbitrary positive integer. Describe a set \mathcal{T} of binary search trees with the following properties:

- Every tree in \mathcal{T} has n nodes, which store the search keys $1, 2, 3, \dots, n$.
- For any integer k , if we choose a tree uniformly at random from \mathcal{T} , the expected depth of node k in that tree is $O(\log n)$.
- Every tree in \mathcal{T} has depth $\Omega(\sqrt{n})$.

(This is why we had to prove via Chernoff bounds that the maximum depth of an n -node treap is $O(\log n)$ with high probability.)

*6. [Extra Credit] Recall that F_k denotes the k th Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for all $k \geq 2$. Suppose we are building a hash table of size $m = F_k$ using the hash function

$$h(x) = (F_{k-1} \cdot x) \bmod F_k$$

Prove that if the consecutive integers $0, 1, 2, \dots, F_k - 1$ are inserted in order into an initially empty table, each integer is hashed into one of the largest contiguous empty intervals in the table. Among other things, this implies that there are no collisions.

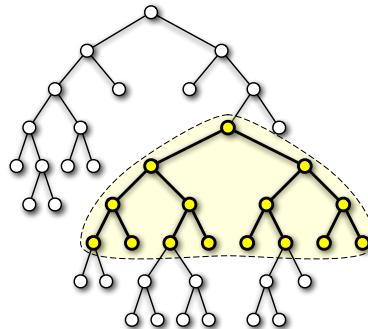
For example, when $m = 13$, the hash table is filled as follows.

0												
0									1			
0			2					1				
0			2					1			3	
0			2			4		1			3	
0	5		2			4		1			3	
0	5		2			4		1	6		3	
0	5		2	7		4		1	6		3	8
0	5		2	7		4	9	1	6		3	8
0	5	10	2	7		4	9	1	6		3	8
0	5	10	2	7	12	4	9	1	6	11	3	8
0	5	10	2	7	12	4	9	1	6	11	3	8

You have 90 minutes to answer four of these questions.
Write your answers in the separate answer booklet.
 You may take the question sheet with you when you leave.

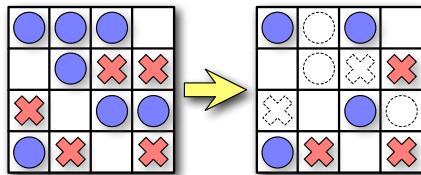
1. Recall that a binary tree is *complete* if every internal node has two children and every leaf has the same depth. An *internal subtree* of a binary tree is a connected subgraph, consisting of a node and some (possibly all or none) of its descendants.

Describe and analyze an algorithm that computes the depth of the *largest complete internal subtree* of a given n -node binary tree. For full credit, your algorithm should run in $O(n)$ time.

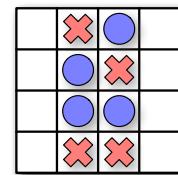


The largest complete internal subtree in this binary tree has depth 3.

2. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

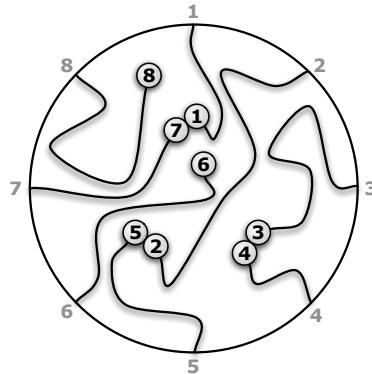
Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

3. Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k . Describe and analyze an algorithm to find the k th largest element in the union of A and B in $O(\log n)$ time. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 21], \quad B[1..8] = [2, 4, 5, 8, 14, 17, 19, 20], \quad k = 10,$$

your algorithm should return 13. You can assume that the arrays contain no duplicates. [Hint: What can you learn from comparing one element of A to one element of B ?]

4. Every year, as part of its annual meeting, the Antarctic Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n . During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.



The end of a typical Antarctic SLUG race. Snails 6 and 8 never find mates.

The organizers must pay $M[3, 4] + M[2, 5] + M[1, 7]$.

For every pair of snails, the Antarctic SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

5. SUBSETSUM and PARTITION are two closely-related NP-hard problems.

- SUBSETSUM: Given a set X of positive integers and an integer t , determine whether there is a subset of X whose elements sum to t .
- PARTITION: Given a set X of positive integers, determine whether X can be partitioned into two subsets whose elements sum to the same value.
 - (a) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.
 - (b) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

Don't forget to **prove** that your reductions are correct.

You have 120 minutes to answer four of these questions.
Write your answers in the separate answer booklet.
 You may take the question sheet with you when you leave.

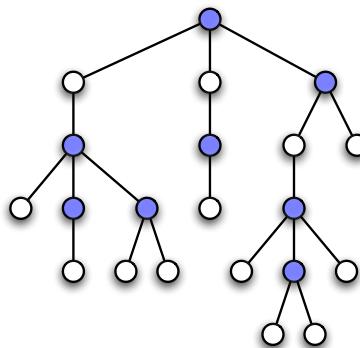
1. Consider the following algorithm for finding the smallest element in an unsorted array:

```
RANDOMMIN( $A[1..n]$ ):
   $min \leftarrow \infty$ 
  for  $i \leftarrow 1$  to  $n$  in random order
    if  $A[i] < min$ 
       $min \leftarrow A[i]$  (*)
  return  $min$ 
```

- (a) [1 pt] In the worst case, how many times does RANDOMMIN execute line (*)?
- (b) [3 pts] What is the probability that line (*) is executed during the last iteration of the for loop?
- (c) [6 pts] What is the *exact* expected number of executions of line (*)? (A correct $\Theta()$ bound is worth 4 points.)

2. Describe and analyze an efficient algorithm to find the size of the smallest vertex cover of a given tree. That is, given a tree T , your algorithm should find the size of the smallest subset C of the vertices, such that every edge in T has at least one endpoint in C .

The following hint may be helpful. Suppose C is a vertex cover that contains a leaf ℓ . If we remove ℓ from the cover and insert its parent, we get another vertex cover of the same size as C . Thus, there is a minimum vertex cover that includes none of the leaves of T (except when the tree has only one or two vertices).

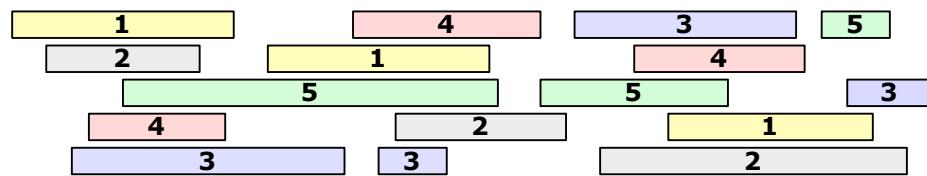


A tree whose smallest vertex cover has size 8.

3. A *dominating set* for a graph G is a subset D of the vertices, such that every vertex in G is either in D or has a neighbor in D . The MINDOMINATINGSET problem asks for the size of the smallest dominating set for a given graph.

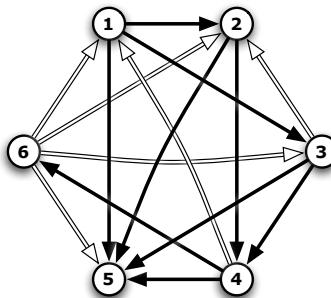
Recall the MINSETCOVER problem from lecture. The input consists of a *ground set* X and a collection of subsets $S_1, S_2, \dots, S_k \subseteq X$. The problem is to find the minimum number of subsets S_i that completely cover X . This problem is NP-hard, because it is a generalization of the vertex cover problem.

- (a) [7 pts] Describe a polynomial-time reduction from MINDOMINATINGSET to MINSET-COVER.
 - (b) [3 pts] Describe a polynomial-time $O(\log n)$ -approximation algorithm for MINDOMINATINGSET.
[Hint: There is a two-line solution.]
4. Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color X . Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, where $L[i]$ and $R[i]$ are the left and right endpoints of the i th interval. As usual, if you use a greedy algorithm, you must prove that it is correct.



A proper coloring of a set of intervals using five colors.

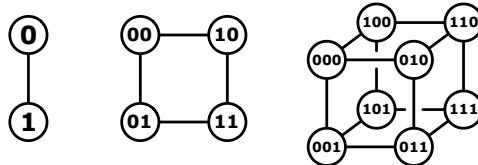
5. The *linear arrangement problem* asks, given an n -vertex directed graph as input, for an ordering v_1, v_2, \dots, v_n of the vertices that maximizes the number of *forward edges*: directed edges $v_i \rightarrow v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem.



A directed graph with six vertices with nine forward edges (black) and six backward edges (white)

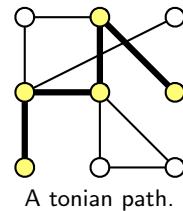
You have 180 minutes to answer six of these questions.
Write your answers in the separate answer booklet.

1. The d -dimensional hypercube is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if and only if their labels differ in exactly one bit.



The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

- (a) [8 pts] Recall that a Hamiltonian cycle is a closed walk that visits each vertex in a graph exactly once. **Prove** that for all $d \geq 2$, the d -dimensional hypercube has a Hamiltonian cycle.
- (b) [2 pts] Recall that an Eulerian circuit is a closed walk that traverses each edge in a graph exactly once. Which hypercubes have an Eulerian circuit? [Hint: This is very easy.]
2. The University of Southern North Dakota at Hoople has hired you to write an algorithm to schedule their final exams. Each semester, USNDH offers n different classes. There are r different rooms on campus and t different time slots in which exams can be offered. You are given two arrays $E[1..n]$ and $S[1..r]$, where $E[i]$ is the number of students enrolled in the i th class, and $S[j]$ is the number of seats in the j th room. At most one final exam can be held in each room during each time slot. Class i can hold its final exam in room j only if $E[i] < S[j]$. Describe and analyze an efficient algorithm to assign a room and a time slot to each class (or report correctly that no such assignment is possible).
3. What is the *exact* expected number of leaves in an n -node treap? (The answer is obviously at most n , so no partial credit for writing " $O(n)$ ".) [Hint: What is the probability that the node with the k th largest key is a leaf?]
4. A *tonian path* in a graph G is a simple path in G that visits more than half of the vertices of G . (Intuitively, a tonian path is “most of a Hamiltonian path”.) **Prove** that it is NP-hard to determine whether or not a given graph contains a tonian path.



A tonian path.

5. A *palindrome* is a string that reads the same forwards and backwards, like x, pop, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. Any string can be broken into sequence of palindromes. For example, the string bubbasesabana ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example,

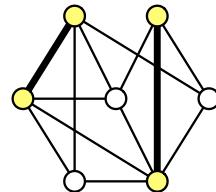
$$\begin{aligned} &\text{bub} + \text{baseesab} + \text{anana} \\ &\text{b} + \text{u} + \text{bb} + \text{a} + \text{sees} + \text{aba} + \text{nan} + \text{a} \\ &\text{b} + \text{u} + \text{bb} + \text{a} + \text{sees} + \text{a} + \text{b} + \text{anana} \\ &\text{b} + \text{u} + \text{b} + \text{b} + \text{a} + \text{s} + \text{e} + \text{e} + \text{s} + \text{a} + \text{b} + \text{a} + \text{n} + \text{a} + \text{n} + \text{a} \end{aligned}$$

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string bubbasesabana, your algorithm would return the integer 3.

6. Consider the following modification of the 2-approximation algorithm for minimum vertex cover that we saw in class. The only real change is that we compute a set of edges instead of a set of vertices.

```
APPROXMINMAXMATCHING( $G$ ):
   $M \leftarrow \emptyset$ 
  while  $G$  has at least one edge
     $(u, v) \leftarrow$  any edge in  $G$ 
     $G \leftarrow G \setminus \{u, v\}$ 
     $M \leftarrow M \cup \{(u, v)\}$ 
  return  $M$ 
```

- (a) [2 pts] **Prove** that the output graph M is a *matching*—no pair of edges in M share a common vertex.
- (b) [2 pts] **Prove** that M is a *maximal matching*— M is not a proper subgraph of another matching in G .
- (c) [6 pts] **Prove** that M contains at most twice as many edges as the *smallest maximal matching* in G .



The smallest maximal matching in a graph.

7. Recall that in the standard maximum-flow problem, the flow through an edge is limited by the capacity of that edge, but there is no limit on how much flow can pass through a vertex. Suppose each vertex v in our input graph has a capacity $c(v)$ that limits the total flow through v , in addition to the usual edge capacities. Describe and analyze an efficient algorithm to compute the maximum (s, t) -flow with these additional constraints. [Hint: Reduce to the standard max-flow problem.]

CS 573: Graduate Algorithms, Fall 2008

Homework 0

Due in class at 12:30pm, Wednesday, September 3, 2008

Name:	
Net ID:	Alias:

I understand the course policies.

-
- Each student must submit their own solutions for this homework. For all future homeworks, groups of up to three students may submit a single, common solution.
 - Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and staple this page to the front of your homework solutions. We will list homework and exam grades on the course web site by alias.

Federal privacy law and university policy forbid us from publishing your grades, even anonymously, without your explicit written permission. *By providing an alias, you grant us permission to list your grades on the course web site. If you do not provide an alias, your grades will not be listed.* For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid) your Social Security number.

- Please carefully read the course policies linked from the course web site. If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup. Once you understand the policies, please check the box at the top of this page. In particular:
 - You may use any source at your disposal—paper, electronic, or human—but you **must** write your solutions in your own words, and you **must** cite every source that you use.
 - Unless explicitly stated otherwise, **every** homework problem requires a proof.
 - Answering “I don’t know” to any homework or exam problem is worth 25% partial credit.
 - Algorithms or proofs containing phrases like “and so on” or “repeat this for all n ”, instead of an explicit loop, recursion, or induction, will receive 0 points.
 - This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, discrete probability, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The early chapters of any algorithms textbook should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks. If you need help, please ask in office hours and/or on the course newsgroup.
-

1. (a) [5 pts] Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases. If your solution requires a particular base case, say so.

$$A(n) = 4A(n/8) + \sqrt{n}$$

$$B(n) = B(n/3) + 2B(n/4) + B(n/6) + n$$

$$C(n) = 6C(n-1) - 9C(n-2)$$

$$D(n) = \max_{n/3 < k < 2n/3} (D(k) + D(n-k) + n)$$

$$E(n) = (E(\sqrt{n}))^2 \cdot n$$

- (b) [5 pts] Sort the functions in the box from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do not turn in proofs**—just a sorted list of 16 functions—but you should do them anyway, just for practice. We use the notation $\lg n = \log_2 n$.

n	$\lg n$	\sqrt{n}	3^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$3^{\sqrt{n}}$	$\sqrt{3^n}$
$3^{\lg n}$	$\lg(3^n)$	$3^{\lg \sqrt{n}}$	$3^{\sqrt{\lg n}}$
$\sqrt{3^{\lg n}}$	$\lg(3^{\sqrt{n}})$	$\lg \sqrt{3^n}$	$\sqrt{\lg(3^n)}$

2. Describe and analyze a data structure that stores set of n records, each with a numerical *key* and a numerical *priority*, such that the following operation can be performed quickly:

- $\text{RANGE TOP}(a, z)$: return the highest-priority record whose key is between a and z .

For example, if the (key, priority) pairs are

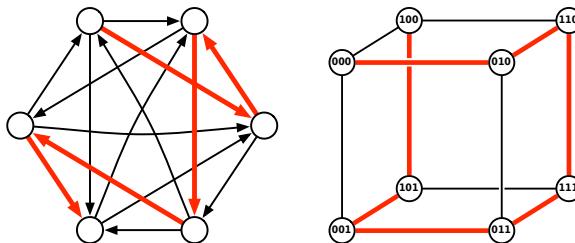
$$(3, 1), (4, 9), (9, 2), (6, 3), (5, 8), (7, 5), (1, 10), (0, 7),$$

then $\text{RANGE TOP}(2, 8)$ would return the record with key 4 and priority 9 (the second in the list).

Analyze both the size of your data structure and the running time of your RANGE TOP algorithm. For full credit, your space and time bounds must both be as small as possible. You may assume that no two records have equal keys or equal priorities, and that no record has a or z as its key. *[Hint: How would you compute the number of keys between a and z ? How would you solve the problem if you knew that a is always $-\infty$?]*

3. A *Hamiltonian path* in G is a path that visits every vertex of G exactly once. In this problem, you are asked to prove that two classes of graphs always contain a Hamiltonian path.

- (a) [5 pts] A *tournament* is a directed graph with exactly one edge between each pair of vertices. (Think of the nodes in a round-robin tournament, where edges represent games, and each edge points from the loser to the winner.) Prove that every tournament contains a *directed* Hamiltonian path.
- (b) [5 pts] Let d be an arbitrary non-negative integer. The d -dimensional *hypercube* is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if and only if their labels differ in exactly one bit. Prove that the d -dimensional hypercube contains a Hamiltonian path.



Hamiltonian paths in a 6-node tournament and a 3-dimensional hypercube.

4. Penn and Teller agree to play the following game. Penn shuffles a standard deck¹ of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs ($3\clubsuit$), at which point the remaining undrawn cards instantly burst into flames.

The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.² To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

- (a) What is the expected number of cards that Teller draws?
- (b) What is the expected *maximum* value among the cards Teller gives to Penn?
- (c) What is the expected *minimum* value among the cards Teller gives to Penn?
- (d) What is the expected number of cards that Teller gives to Penn?

Full credit will be given only for *exact* answers (with correct proofs, of course). [Hint: Let $13 = n$.]

¹In a standard deck of playing cards, each card has a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$ and a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$; each of the 52 possible suit-value pairs appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

²Specifically, he hurls them from the opposite side of the stage directly into the back of Penn's right hand. Ouch!

5. (a) The **Fibonacci numbers** F_n are defined by the recurrence $F_n = F_{n-1} + F_{n-2}$, with base cases $F_0 = 0$ and $F_1 = 1$. Here are the first several Fibonacci numbers:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Prove that any non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers. That is, if the Fibonacci number F_i appears in the sum, it appears exactly once, and its neighbors F_{i-1} and F_{i+1} do not appear at all. For example:

$$17 = F_7 + F_4 + F_2, \quad 42 = F_9 + F_6, \quad 54 = F_9 + F_7 + F_5 + F_3 + F_1.$$

- (b) The Fibonacci sequence can be extended backward to negative indices by rearranging the defining recurrence: $F_n = F_{n+2} - F_{n+1}$. Here are the first several negative-index Fibonacci numbers:

F_{-10}	F_{-9}	F_{-8}	F_{-7}	F_{-6}	F_{-5}	F_{-4}	F_{-3}	F_{-2}	F_{-1}
-55	34	-21	13	-8	5	-3	2	-1	1

Prove that $F_{-n} = -F_n$ if and only if n is even.

- (c) Prove that *any* integer—positive, negative, or zero—can be written as the sum of distinct, non-consecutive Fibonacci numbers *with negative indices*. For example:

$$17 = F_{-7} + F_{-5} + F_{-2}, \quad -42 = F_{-10} + F_{-7}, \quad 54 = F_{-9} + F_{-7} + F_{-5} + F_{-3} + F_{-1}.$$

[Hint: Zero is both non-negative and even. Don't use weak induction!]

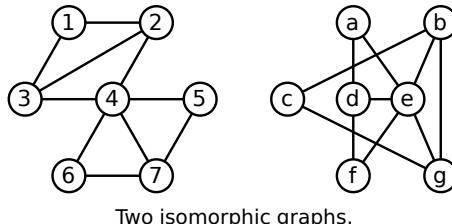
CS 573: Graduate Algorithms, Fall 2008

Homework 1

Due at 11:59:59pm, Wednesday, September 17, 2008

For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and alias (if any) of every group member on the first page of your submission.

1. Two graphs are said to be **isomorphic** if one can be transformed into the other just by relabeling the vertices. For example, the graphs shown below are isomorphic; the left graph can be transformed into the right graph by the relabeling $(1, 2, 3, 4, 5, 6, 7) \mapsto (c, g, b, e, a, f, d)$.



Consider the following related decision problems:

- **GRAPHISOMORPHISM:** Given two graphs G and H , determine whether G and H are isomorphic.
 - **EVENGRAPHISOMORPHISM:** Given two graphs G and H , such that every vertex in G and H has even degree, determine whether G and H are isomorphic.
 - **SUBGRAPHISOMORPHISM:** Given two graphs G and H , determine whether G is isomorphic to a subgraph of H .
- (a) Describe a polynomial-time reduction from **EVENGRAPHISOMORPHISM** to **GRAPHISOMORPHISM**.
 - (b) Describe a polynomial-time reduction from **GRAPHISOMORPHISM** to **EVENGRAPHISOMORPHISM**.
 - (c) Describe a polynomial-time reduction from **GRAPHISOMORPHISM** to **SUBGRAPHISOMORPHISM**.
 - (d) Prove that **SUBGRAPHISOMORPHISM** is NP-complete.
 - (e) What can you conclude about the NP-hardness of **GRAPHISOMORPHISM**? Justify your answer.

[Hint: These are all easy!]

2. (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).]
3. The following variant of 3SAT is called either EXACT3SAT or 1IN3SAT, depending on who you ask.

Given a boolean formula in conjunctive normal form with 3 literals per clause, is there an assignment that makes **exactly one** literal in each clause TRUE?

Prove that this problem is NP-complete.

4. Suppose you are given a magic black box that can solve the MAXCLIQUE problem *in polynomial time*. That is, given an arbitrary graph G as input, the magic black box computes 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.
 5. A boolean formula in *exclusive-or conjunctive normal form* (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive-or* of several literals. The XCNF-SAT problem asks whether a given XCNF boolean formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-complete.
- *6. [Extra credit] Describe and analyze an algorithm to solve 3SAT in $O(\phi^n \text{poly}(n))$ time, where $\phi = (1 + \sqrt{5})/2 \approx 1.618034$. [Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]

⁰In class, I asserted that Gaussian elimination was probably discovered by Gauss, in violation of Stigler's Law of Eponymy. In fact, a method very similar to Gaussian elimination appears in the Chinese treatise *Nine Chapters on the Mathematical Art*, believed to have been finalized before 100AD, although some material may predate emperor Qin Shi Huang's infamous 'burning of the books and burial of the scholars' in 213BC. The great Chinese mathematician Liu Hui, in his 3rd-century commentary on *Nine Chapters*, compares two variants of the method and counts the number of arithmetic operations used by each, with the explicit goal of find the more efficient method. This is arguably the earliest recorded analysis of any algorithm.

CS 573: Graduate Algorithms, Fall 2008

Homework 2

Due at 11:59:59pm, Wednesday, October 1, 2008

-
- For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and alias (if any) of every group member on the first page of your submission.
 - We will use the following point breakdown to grade dynamic programming algorithms: 60% for a correct recurrence (including base cases), 20% for correct running time analysis of the memoized recurrence, 10% for correctly transforming the memoized recursion into an iterative algorithm.
 - A greedy algorithm *must* be accompanied by a proof of correctness in order to receive *any* credit.
-

1. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays of numbers. A **common supersequence** of X and Y is another sequence that contains both X and Y as subsequences. Describe and analyze an efficient algorithm to compute the function $scs(X, Y)$, which gives the length of the shortest common supersequence of X and Y .
(b) Call a sequence $X[1..n]$ of numbers **oscillating** if $X[i] < X[i + 1]$ for all even i , and $X[i] > X[i + 1]$ for all odd i . Describe and analyze an efficient algorithm to compute the function $los(X)$, which gives the length of the longest oscillating subsequence of an arbitrary array X of integers.
(c) Call a sequence $X[1..n]$ of numbers **accelerating** if $2 \cdot X[i] < X[i - 1] + X[i + 1]$ for all i . Describe and analyze an efficient algorithm to compute the function $lxs(X)$, which gives the length of the longest accelerating subsequence of an arbitrary array X of integers.
2. A **palindrome** is a string that reads the same forwards and backwards, like x, pop, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. Any string can be broken into sequence of palindromes. For example, the string **bubbasesabana** ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example:

$$\begin{aligned} &\text{bub} + \text{baseesab} + \text{anana} \\ &\text{b} + \text{u} + \text{bb} + \text{a} + \text{sees} + \text{aba} + \text{nan} + \text{a} \\ &\text{b} + \text{u} + \text{bb} + \text{a} + \text{sees} + \text{a} + \text{b} + \text{anana} \\ &\text{b} + \text{u} + \text{b} + \text{b} + \text{a} + \text{s} + \text{e} + \text{e} + \text{s} + \text{a} + \text{b} + \text{a} + \text{n} + \text{a} + \text{n} + \text{a} \end{aligned}$$

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string **bubbasesabana**, your algorithm would return the integer 3.

3. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \text{poly}(n))$ time. Given an undirected n -vertex graph G with weighted edges, your algorithm should return the weight of the lightest Hamiltonian cycle in G , or ∞ if G has no Hamiltonian cycles. [Hint: The obvious recursive algorithm takes $O(n!)$ time.]

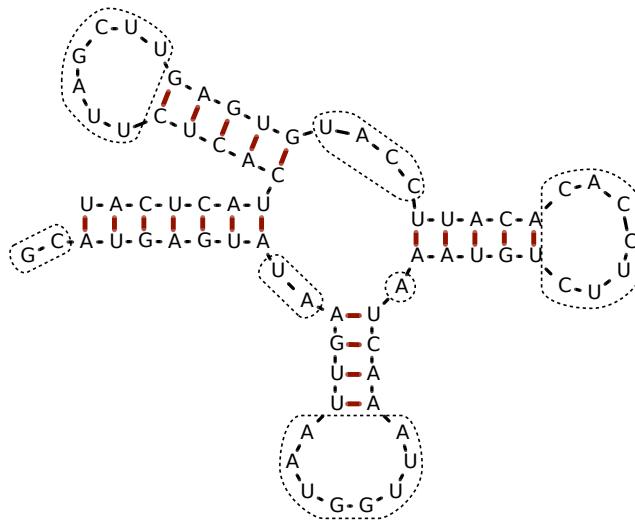
4. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{A, C, G, U\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule.

We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ **overlap** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

- Each base can be paired with at most one other base.
- Only A-U pairs and C-G pairs can bond.
- Pairs of the form $(i, i + 1)$ and $(i, i + 2)$ cannot bond.
- Overlapping base pairs cannot bond.

The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.

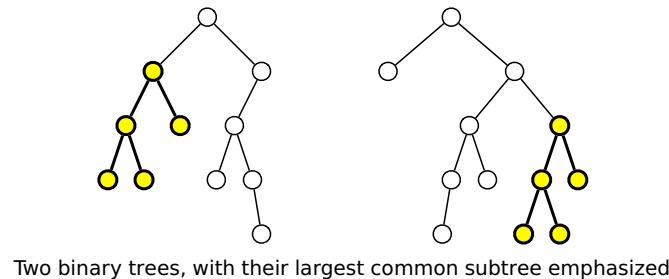


Example RNA secondary structure with 21 base pairs, indicated by heavy red lines.
Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$

- Describe and analyze an algorithm that computes the maximum possible *number* of base pairs in a secondary structure for a given RNA sequence.
- A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths.¹ Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

¹This score function has absolutely no connection to reality; I just made it up. Real RNA structure prediction requires much more complicated scoring functions.

5. A *subtree* of a (rooted, ordered) binary tree T consists of a node and all its descendants. Design and analyze an efficient algorithm to compute the *largest common subtree* of two given binary trees T_1 and T_2 ; this is the largest subtree of T_1 that is isomorphic to a subtree in T_2 . The contents of the nodes are irrelevant; we are only interested in matching the underlying combinatorial structure.



- *6. [Extra credit] Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. A *digital subsequence* of D is an sequence of positive integers composed in the usual way from disjoint substrings of D . For example, 3, 4, 5, 6, 23, 38, 62, 64, 83, 279 is an increasing digital subsequence of the first several digits of π :

3, 1, 4, 1, 5, 9, 6, 2, 3, 4, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the previous example has length 10.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D . [Hint: Be careful about your computational assumptions. How long does it take to compare two k -digit numbers?]

CS 573: Graduate Algorithms, Fall 2008

Homework 3

Due at 11:59:59pm, Wednesday, October 22, 2008

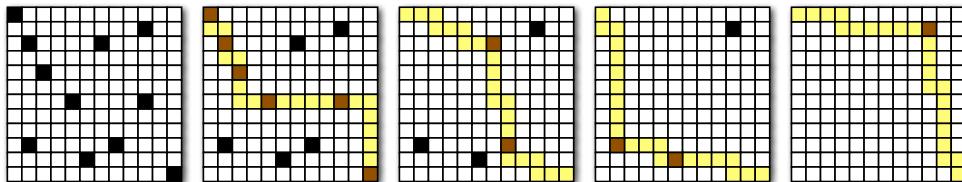
-
- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.
-

- Consider an $n \times n$ grid, some of whose cells are marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. We want to compute the minimum number of monotone paths that cover all marked cells. The input to our problem is an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{TRUE}$ if and only if cell (i, j) is marked.

One of your friends suggests the following greedy strategy:

- Find (somehow) one “good” path π that covers the maximum number of marked cells.
- Unmark the cells covered by π .
- If any cells are still marked, recursively cover them.

Does this greedy strategy always compute an optimal solution? If yes, give a proof. If no, give a counterexample.



Greedily covering the marked cells in a grid with four monotone paths.

- Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X , that is, any real value that is contained in some interval in X is also contained in some interval in Y . The *size* of a tiling path is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X . If you use a greedy algorithm, you must prove that it is correct.



A set of intervals. The seven shaded intervals form a tiling path.

3. Given a graph G with edge weights and an integer k , suppose we wish to partition the vertices of G into k subsets S_1, S_2, \dots, S_k so that the sum of the weights of the edges that cross the partition (i.e., that have endpoints in different subsets) is as large as possible.
- Describe an efficient $(1 - 1/k)$ -approximation algorithm for this problem. [Hint: Solve the special case $k = 2$ first.]
 - Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for this new problem? Justify your answer.
4. Consider the following heuristic for constructing a vertex cover of a connected graph G : ***Return the set of all non-leaf nodes of any depth-first spanning tree.*** (Recall that a depth-first spanning tree is a *rooted* tree; the root is not considered a leaf, even if it has only one neighbor in the tree.)
- Prove that this heuristic returns a vertex cover of G .
 - Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G .
 - Prove that for any $\varepsilon > 0$, there is a graph for which this heuristic returns a vertex cover of size at least $(2 - \varepsilon) \cdot OPT$.
5. Consider the following greedy approximation algorithm to find a vertex cover in a graph:

```

GREEDYVERTEXCOVER( $G$ ):
   $C \leftarrow \emptyset$ 
  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  $C$ 

```

In class we proved that the approximation ratio of this algorithm is $O(\log n)$; your task is to prove a matching lower bound. Specifically, for any positive integer n , describe an n -vertex graph G such that $\text{GREEDYVERTEXCOVER}(G)$ returns a vertex cover that is $\Omega(\log n)$ times larger than optimal. [Hint: $H_n = \Omega(\log n)$.]

- *6. [*Extra credit*] Consider the greedy algorithm for metric TSP: Start at an arbitrary vertex u , and at each step, travel to the closest unvisited vertex.
- Prove that this greedy algorithm is an $O(\log n)$ -approximation algorithm, where n is the number of vertices. [Hint: Show that the k th least expensive edge in the tour output by the greedy algorithm has weight at most $OPT/(n - k + 1)$; try $k = 1$ and $k = 2$ first.]
 - Prove that the greedy algorithm for metric TSP is no better than an $O(\log n)$ -approximation. That is, describe an infinite family of weighted graphs that satisfy the triangle inequality, such that the greedy algorithm returns a cycle whose length is $\Omega(\log n)$ times the optimal TSP tour.

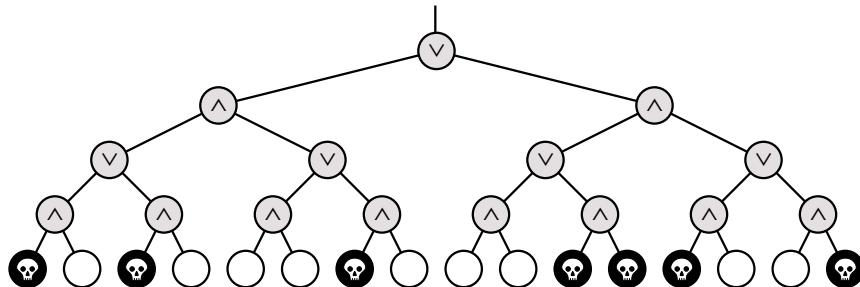
CS 573: Graduate Algorithms, Fall 2008

Homework 4

Due at 11:59:59pm, Wednesday, October 31, 2008

-
- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.
 - Unless a problem explicitly states otherwise, you can assume the existence of a function $\text{RANDOM}(k)$, which returns an integer uniformly distributed in the range $\{1, 2, \dots, k\}$ in $O(1)$ time; the argument k must be a positive integer. For example, $\text{RANDOM}(2)$ simulates a fair coin flip, and $\text{RANDOM}(1)$ *always* returns 1.
-

- Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black leaves stand represent TRUE and FALSE inputs, respectively. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*
- Unfortunately, Death won't let you even look at every node in the tree. Describe and analyze a randomized algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*
- [Extra credit]** Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm at all.]*

2. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. [Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]

3. (a) Prove that the expected number of proper descendants of any node in a treap is *exactly* equal to the expected depth of that node.
- (b) Why doesn't the Chernoff-bound argument for depth imply that, with high probability, *every* node in a treap has $O(\log n)$ descendants? The conclusion is obviously bogus—every n -node treap has one node with exactly n descendants!—but what is the flaw in the argument?
- (c) What is the expected number of leaves in an n -node treap? [Hint: What is the probability that in an n -node treap, the node with k th smallest search key is a leaf?]
4. The following randomized algorithm, sometimes called “one-armed quicksort”, selects the r th smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call `RANDOMSELECT(A, 1)`; to find the median element, you would call `RANDOMSELECT(A, $\lfloor n/2 \rfloor$)`. The subroutine `PARTITION(A[1..n], p)` splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element.

```

RANDOMSELECT( $A[1..n]$ ,  $r$ ) :
   $k \leftarrow \text{PARTITION}(A[1..n], \text{RANDOM}(n))$ 
  if  $r < k$ 
    return RANDOMSELECT( $A[1..k - 1]$ ,  $r$ )
  else if  $r > k$ 
    return RANDOMSELECT( $A[k + 1..n]$ ,  $r - k$ )
  else
    return  $A[k]$ 

```

- (a) State a recurrence for the expected running time of `RANDOMSELECT`, as a function of n and r .
- (b) What is the *exact* probability that `RANDOMSELECT` compares the i th smallest and j th smallest elements in the input array? The correct answer is a simple function of i , j , and r . [Hint: Check your answer by trying a few small examples.]
- (c) Show that for any n and r , the expected running time of `RANDOMSELECT` is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b).
- *(d) [*Extra Credit*] Find the *exact* expected number of comparisons executed by `RANDOMSELECT`, as a function of n and r .

5. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue containing the empty set.
- **FINDMIN(Q)**: Return the smallest element of Q (if any).
- **DELETEMIN(Q)**: Remove the smallest element in Q (if any).
- **INSERT(Q, x)**: Insert element x into Q , if it is not already there.
- **DECREASEKEY(Q, x, y)**: Replace an element $x \in Q$ with a smaller key y . (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x .
- **DELETE(Q, x)**: Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x .
- **MELD(Q_1, Q_2)**: Return a new priority queue containing all the elements of Q_1 and Q_2 ; this operation destroys Q_1 and Q_2 .

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```

MELD( $Q_1, Q_2$ ):
    if  $Q_1$  is empty return  $Q_2$ 
    if  $Q_2$  is empty return  $Q_1$ 
    if  $key(Q_1) > key(Q_2)$ 
        swap  $Q_1 \leftrightarrow Q_2$ 
    with probability 1/2
         $left(Q_1) \leftarrow \text{MELD}(left(Q_1), Q_2)$ 
    else
         $right(Q_1) \leftarrow \text{MELD}(right(Q_1), Q_2)$ 
    return  $Q_1$ 
```

- (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (*not* just those constructed by the operations listed above), the expected running time of $\text{MELD}(Q_1, Q_2)$ is $O(\log n)$, where n is the total number of nodes in both trees. [Hint: How long is a random root-to-leaf path in an n -node binary tree if each left/right choice is made with equal probability?]
- (b) Prove that $\text{MELD}(Q_1, Q_2)$ runs in $O(\log n)$ time with high probability. [Hint: You don't need Chernoff bounds, but you might use the identity $\binom{ck}{k} \leq (ce)^k$.]
- (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

*6. [*Extra credit*] In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0, 1]$, but in practice, computers only have access to random *bits*. This problem asks you to analyze a modification of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1 .. \infty]$ of random bits, which is interpreted as the rational number

$$\text{priority}(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or ‘reveal’) new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LARGERPRIORITY( $v, w$ ):  
for  $i \leftarrow 1$  to  $\infty$   
  if  $i > \ell_v$   
     $\ell_v \leftarrow i$ ;  $\pi_v[i] \leftarrow \text{RANDOMBIT}$   
  if  $i > \ell_w$   
     $\ell_w \leftarrow i$ ;  $\pi_w[i] \leftarrow \text{RANDOMBIT}$   
  if  $\pi_v[i] > \pi_w[i]$   
    return  $v$   
  else if  $\pi_v[i] < \pi_w[i]$   
    return  $w$ 
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LARGERPRIORITY during these insertions.

- (a) Prove that $E[L] = \Theta(n)$.
- (b) Prove that $E[\ell_v] = \Theta(1)$ for any node v . [Hint: This is equivalent to part (a). Why?]
- (c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. [Hint: Why doesn’t this contradict part (b)?]

CS 573: Graduate Algorithms, Fall 2008

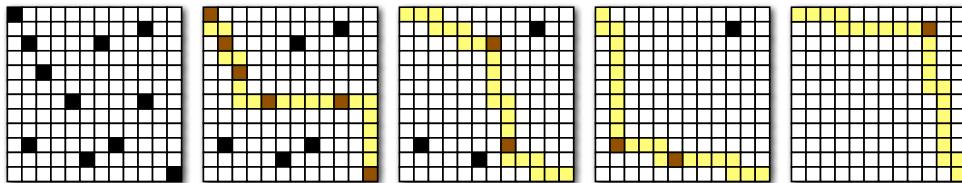
Homework 5

Due at 11:59:59pm, Wednesday, November 19, 2008

- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.

- Recall the following problem from Homework 3: You are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{TRUE}$ if and only if cell (i, j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell.

Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.



Greedyly covering the marked cells in a grid with four monotone paths.

- Suppose we are given a directed graph $G = (V, E)$, two vertices s and t , and a capacity function $c: V \rightarrow \mathbb{R}^+$. A flow f is *feasible* if the total flow into every vertex v is at most $c(v)$:

$$\sum_u f(u \rightarrow v) \leq c(v) \quad \text{for every vertex } v.$$

Describe and analyze an efficient algorithm to compute a feasible flow of maximum value.

- Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A . For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

4. *Ad-hoc networks* are made up of cheap, low-powered wireless devices. In principle¹, these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other situations where people might want to monitor conditions in hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be dropped into the area from an airplane (for instance), and then they would somehow automatically configure themselves into an efficiently functioning wireless network.

The devices can communication only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D .

We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit the information it has to some other *backup* device within its communication range. To improve reliability, we require each device x to have k potential backup devices, all within distance D of x ; we call these k devices the *backup set* of x . Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

So suppose we are given the communication radius D , parameters b and k , and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j . Describe an algorithm that either computes a backup set of size k for each of the n devices, such that that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

5. Let $G = (V, E)$ be a directed graph where for each vertex v , the in-degree and out-degree of v are equal. Let u and v be two vertices G , and suppose G contains k edge-disjoint paths from u to v . Under these conditions, must G also contain k edge-disjoint paths from v to u ? Give a proof or a counterexample with explanation.

- *6. [*Extra credit*] A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe a polynomial-time algorithm to compute, given two rooted trees A and B , the largest common rooted subtree of A and B .

[Hint: Let $LCS(u, v)$ denote the largest common subtree whose root in A is u and whose root in B is v . Your algorithm should compute $LCS(u, v)$ for all vertices u and v using dynamic programming. This would be easy if every vertex had $O(1)$ children, and still straightforward if the children of each node were ordered from left to right and the common subtree had to respect that ordering. But for unordered trees with large degree, you need another trick to combine recursive subproblems efficiently. Don't waste your time trying to reduce the polynomial running time.]

¹but not really in practice

CS 573: Graduate Algorithms, Fall 2008

Homework 6

Practice only

-
- This homework is only for practice; do not submit solutions. At least one (sub)problem (or something *very* similar) will appear on the final exam.
-

1. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.
 - (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.
 - (b) Prove that finding the optimal solution to an integer program is NP-hard.

[Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]

2. Describe precisely how to dualize a linear program written in general form:

$$\begin{aligned} & \text{maximize} \sum_{j=1}^d c_j x_j \\ & \text{subject to} \sum_{j=1}^d a_{ij} x_j \leq b_i \quad \text{for each } i = 1..p \\ & \quad \sum_{j=1}^d a_{ij} x_j = b_i \quad \text{for each } i = p+1..p+q \\ & \quad \sum_{j=1}^d a_{ij} x_j \geq b_i \quad \text{for each } i = p+q+1..n \end{aligned}$$

Keep the number of dual variables as small as possible. The dual of the dual of any linear program should be *syntactically identical* to the original linear program.

3. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time, using this subroutine as a black box. Your algorithm should return an optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one constraint to guarantee boundedness; add one variable to guarantee feasibility.]*

4. Suppose you are given a set P of n points in some high-dimensional space \mathbb{R}^d , each labeled either *black* or *white*. A *linear classifier* is a d -dimensional vector c with the following properties:

- If p is a black point, then $p \cdot c > 0$.
- If p is a white point, then $p \cdot c < 0$.

Describe an efficient algorithm to find a linear classifier for the given data points, or correctly report that none exists. [*Hint: This is almost trivial, but not quite.*]

Lots more linear programming problems can be found at <http://www.ee.ucla.edu/ee236a/homework/problems.pdf>. Enjoy!

You have 120 minutes to answer all five questions.
Write your answers in the separate answer booklet.
Please turn in your question sheet and your cheat sheet with your answers.

1. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, several cards are dealt face up in a long row. Then you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. Each card is worth a different number of points. The player that collects the most points wins the game.

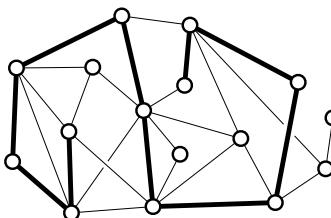
Like most eight-year-olds who haven't studied algorithms, Elmo follows the obvious greedy strategy every time he plays: ***Elmo always takes the card with the higher point value.*** Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

- (a) Describe an initial sequence of cards that allows you to win against Elmo, no matter who moves first, but *only* if you do *not* follow Elmo's greedy strategy.
- (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

Here is a sample game, where both you and Elmo are using the greedy strategy. Elmo wins 8–7. You cannot win this particular game, no matter what strategy you use.

Initial cards	2	4	5	1	3
Elmo takes the 3	2	4	5	1	\$
You take the 2	\$	4	5	1	
Elmo takes the 4		\$	5	1	
You take the 5			\$	1	
Elmo takes the 1				\$	

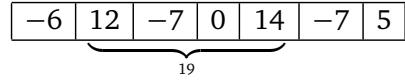
2. **Prove** that the following problem is NP-hard: Given an undirected graph G , find the longest path in G whose length is a multiple of 5.



This graph has a path of length 10, but no path of length 15.

3. Suppose you are given an array $A[1..n]$ of integers. Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray $A[i..j]$.

For example, if the array A contains the numbers $[-6, 12, -7, 0, 14, -7, 5]$, your algorithm should return the number 19:



4. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, ‘bananaanananas’ is a shuffle of ‘banana’ and ‘ananas’ in several different ways:

banana_ananas ban_ana_ana_anas b_an_aa_ana_as

The strings ‘prodgyrnamatammiiincg’ and ‘dyprongarmammicing’ are both shuffles of ‘dynamic’ and ‘programming’:

pro^dg^yrⁿam^matammiiⁱn^cg dy^proⁿg^ar^mammicⁱng

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B .

5. Suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k . Describe an algorithm to find the k th smallest element in the union of A and B in $\Theta(\log(m+n))$ time. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \quad B[1..5] = [2, 5, 8, 17, 19] \quad k = 6$$

your algorithm should return 8. You can assume that the arrays contain no duplicates. An algorithm that works only in the special case $n = m = k$ is worth 7 points.

[Hint: What can you learn from comparing one element of A to one element of B?]

You have 120 minutes to answer all five questions.
Write your answers in the separate answer booklet.
Please turn in your question sheet and your cheat sheet with your answers.

1. Consider the following modification of the ‘dumb’ 2-approximation algorithm for minimum vertex cover that we saw in class. The only change is that we output a set of edges instead of a set of vertices.

```
APPROXMINMAXMATCHING( $G$ ):  

 $M \leftarrow \emptyset$   

while  $G$  has at least one edge  

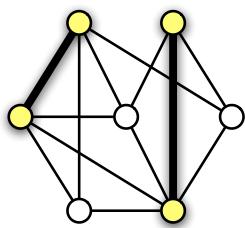
    let  $(u, v)$  be any edge in  $G$   

    remove  $u$  and  $v$  (and their incident edges) from  $G$   

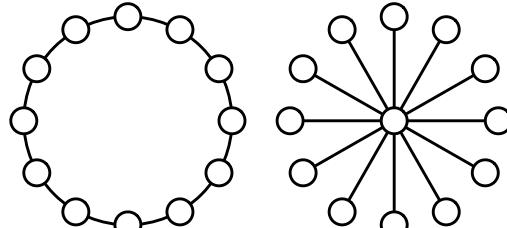
    add  $(u, v)$  to  $M$   

return  $M$ 
```

- (a) **Prove** that this algorithm computes a *matching*—no two edges in M share a common vertex.
- (b) **Prove** that M is a *maximal matching*— M is not a proper subgraph of another matching in G .
- (c) **Prove** that M contains at most twice as many edges as the *smallest maximal matching* in G .



The smallest maximal matching in a graph.



A cycle and a star.

2. Consider the following heuristic for computing a small vertex cover of a graph.

- Assign a random *priority* to each vertex, chosen independently and uniformly from the real interval $[0, 1]$ (just like treaps).
- Mark every vertex that does *not* have larger priority than *all* of its neighbors.

For any graph G , let $OPT(G)$ denote the size of the smallest vertex cover of G , and let $M(G)$ denote the number of nodes marked by this algorithm.

- (a) **Prove** that the set of vertices marked by this heuristic is *always* a vertex cover.
- (b) Suppose the input graph G is a *cycle*, that is, a connected graph where every vertex has degree 2. What is the expected value of $M(G)/OPT(G)$? **Prove** your answer is correct.
- (c) Suppose the input graph G is a *star*, that is, a tree with one central vertex of degree $n - 1$. What is the expected value of $M(G)/OPT(G)$? **Prove** your answer is correct.

3. Suppose we want to write an efficient function $\text{SHUFFLE}(A[1..n])$ that randomly permutes the array A , so that each of the $n!$ permutations is equally likely.

- (a) *Prove* that the following SHUFFLE algorithm is *not* correct. [Hint: There is a two-line proof.]

```

SHUFFLE( $A[1..n]$ ):
  for  $i = 1$  to  $n$ 
    swap  $A[i] \leftrightarrow A[\text{RANDOM}(n)]$ 
```

- (b) Describe and analyze a correct SHUFFLE algorithm whose expected running time is $O(n)$.

Your algorithm may call the function $\text{RANDOM}(k)$, which returns an integer uniformly distributed in the range $\{1, 2, \dots, k\}$ in $O(1)$ time. For example, $\text{RANDOM}(2)$ simulates a fair coin flip, and $\text{RANDOM}(1)$ *always* returns 1.

4. Let Φ be a legal input for 3SAT—a boolean formula in conjunctive normal form, with exactly three literals in each clause. Recall that an assignment of boolean values to the variables in Φ **satisfies** a clause if at least one of its literals is TRUE. The **maximum satisfiability problem**, sometimes called MAX3SAT, asks for the maximum number of clauses that can be simultaneously satisfied by a single assignment. Solving MAXSAT exactly is clearly also NP-hard; this problem asks about approximation algorithms.

- (a) Let $\text{MaxSat}(\Phi)$ denote the maximum number of clauses that can be simultaneously satisfied by one variable assignment. Suppose we randomly assign each variable in Φ to be TRUE or FALSE, each with equal probability. *Prove* that the expected number of satisfied clauses is at least $\frac{7}{8}\text{MaxSat}(\Phi)$.
- (b) Let $\text{MinUnsat}(\Phi)$ denote the *minimum* number of clauses that can be simultaneously *unsatisfied* by a single assignment. *Prove* that it is NP-hard to approximate $\text{MinUnsat}(\Phi)$ within a factor of $10^{10^{100}}$.

5. Consider the following randomized algorithm for generating biased random bits. The subroutine FAIRCOIN returns either 0 or 1 with equal probability; the random bits returned by FAIRCOIN are mutually independent.

```

ONEINTHREE:
  if FAIRCOIN = 0
    return 0
  else
    return 1 - ONEINTHREE
```

- (a) *Prove* that ONEINTHREE returns 1 with probability $1/3$.
- (b) What is the *exact* expected number of times that this algorithm calls FAIRCOIN? *Prove* your answer is correct.
- (c) Now suppose you are given a subroutine ONEINTHREE that generates a random bit that is equal to 1 with probability $1/3$. Describe a FAIRCOIN algorithm that returns either 0 or 1 with equal probability, using ONEINTHREE as a subroutine. **Your only source of randomness is ONEINTHREE; in particular, you may not use the RANDOM function from problem 3.**
- (d) What is the *exact* expected number of times that your FAIRCOIN algorithm calls ONEINTHREE? *Prove* your answer is correct.

You have 180 minutes to answer all seven questions.
Write your answers in the separate answer booklet.
You can keep everything except your answer booklet when you leave.

1. An *integer program* is a linear program with the additional constraint that the variables must take only integer values. **Prove** that deciding whether an integer program has a feasible solution is NP-complete. [Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]
2. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A *heater* is a priority search tree in which the priorities are given by the user, and the search keys are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is the ‘opposite’ of a treap.

The following problems consider an n -node heater T whose node priorities are the integers from 1 to n . We identify nodes in T by their priorities; thus, ‘node 5’ means the node in T with priority 5. The min-heap property implies that node 1 is the root of T . Finally, let i and j be integers with $1 \leq i < j \leq n$.

- (a) **Prove** that in a random permutation of the $(i + 1)$ -element set $\{1, 2, \dots, i, j\}$, elements i and j are adjacent with probability $2/(i + 1)$.
 - (b) **Prove** that node i is an ancestor of node j with probability $2/(i + 1)$. [Hint: Use part (a)!]
 - (c) What is the probability that node i is a descendant of node j ? [Hint: Don’t use part (a)!]
 - (d) What is the exact expected depth of node j ?
-
3. The UIUC Faculty Senate has decided to convene a committee to determine whether Chief Illiniwek should become the official *maseot symbol* of the University of Illinois Global Campus. Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member will represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that any committee on virtual *maseots symbols* must contain the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.
- Describe an efficient algorithm to select the membership of the Global Illiniwek Committee. Your input is a list of all UIUC faculty members, their ranks (assistant, associate, or full), and their departmental affiliation(s). There are n faculty members and $3k$ departments.
-
4. Let $\alpha(G)$ denote the number of vertices in the largest independent set in a graph G . **Prove** that the following problem is NP-hard: Given a graph G , return *any* integer between $\alpha(G) - 31337$ and $\alpha(G) + 31337$.

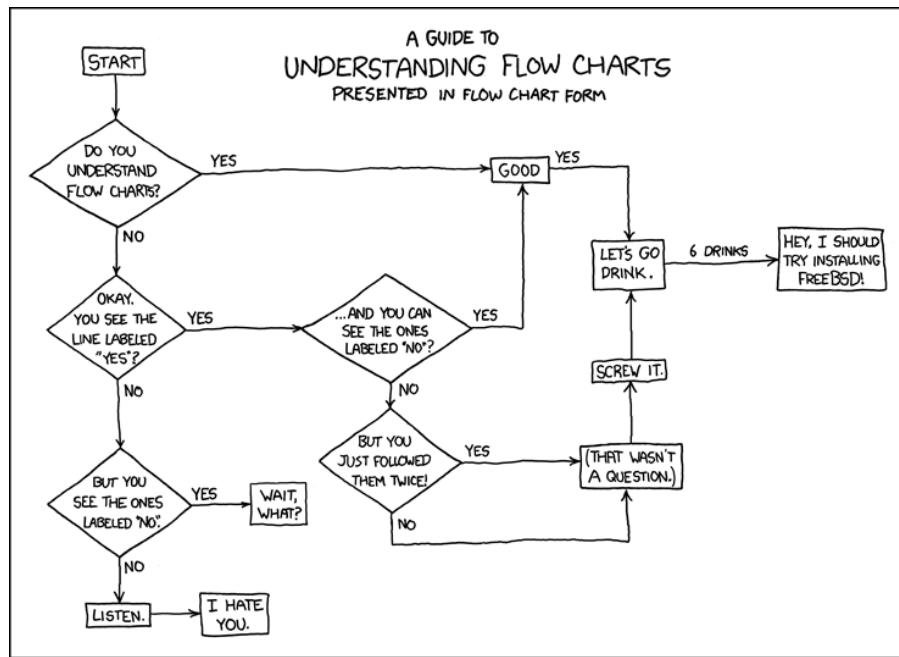
5. Let $G = (V, E)$ be a directed graph with capacities $c: E \rightarrow \mathbb{R}^+$, a source vertex s , and a target vertex t . Suppose someone hands you a function $f: E \rightarrow \mathbb{R}$. Describe and analyze a fast algorithm to determine whether f is a maximum (s, t) -flow in G .

6. For some strange reason, you decide to ride your bicycle 3688 miles from Urbana to Wasilla, Alaska, to join in the annual Wasilla Mining Festival and Helicopter Wolf Hunt. The festival starts exactly 32 days from now, so you need to bike an average of 109 miles each day. Because you are a poor starving student, you can only afford to sleep at campgrounds, which are unfortunately *not* spaced exactly 109 miles apart. So some days you will have to ride more than average, and other days less, but you would like to keep the variation as small as possible. You settle on a formal scoring system to help decide where to sleep; if you ride x miles in one day, your score for that day is $(109 - x)^2$. What is the minimum possible total score for all 32 days?

More generally, suppose you have D days to travel DP miles, there are n campgrounds along your route, and your score for traveling x miles in one day is $(x - P)^2$. You are given a sorted array $dist[1..n]$ of real numbers, where $dist[i]$ is the distance from your starting location to the i th campground; it may help to also set $dist[0] = 0$ and $dist[n + 1] = DP$. Describe and analyze a fast algorithm to compute the minimum possible score for your trip. The running time of your algorithm should depend on the integers D and n , but not on the real number P .

7. Describe and analyze efficient algorithms for the following problems.

- (a) Given a set of n integers, does it contain elements a and b such that $a + b = 0$?
- (b) Given a set of n integers, does it contain elements a , b , and c such that $a + b = c$?



— Randall Munroe, *xkcd*, December 17, 2008 (<http://xkcd.com/518/>)

CS 473: Undergraduate Algorithms, Spring 2009

Homework 0

Due in class at 11:00am, Tuesday, January 27, 2009

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The early chapters of any algorithms textbook should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks. If you need help, please ask in office hours and/or on the course newsgroup.
- Each student must submit individual solutions for this homework. For all future homeworks, groups of up to three students may submit a single, common solution.
- Please carefully read the course policies linked from the course web site. If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup. In particular:
 - Submit five separately stapled solutions, one for each numbered problem, with your name and NetID clearly printed on each page. Please do not staple everything together.
 - You may use any source at your disposal—paper, electronic, or human—but you **must** write your solutions in your own words, and you **must** cite every source that you use.
 - Unless explicitly stated otherwise, **every** homework problem requires a proof.
 - Answering “I don’t know” to any homework or exam problem (except for extra credit problems) is worth 25% partial credit.
 - Algorithms or proofs containing phrases like “and so on” or “repeat this process for all n ”, instead of an explicit loop, recursion, or induction, will receive 0 points.

Write the sentence “I understand the course policies.” at the top of your solution to problem 1.

1. Professor George O’Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character &. Preorder and postorder traversals of the tree visit the nodes in the following order:
 - Preorder: I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X
 - Postorder: H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I
 - (a) List the nodes in George’s tree in the order visited by an inorder traversal.
 - (b) Draw George’s tree.

2. (a) [5 pts] Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases. If your solution requires a particular base case, say so. ***Do not submit proofs***—just a list of five functions—but you should do them anyway, just for practice.

$$A(n) = 10A(n/5) + n$$

$$B(n) = 2B\left(\left\lceil \frac{n+3}{4} \right\rceil\right) + 5n^{6/7} - 8\sqrt{\frac{n}{\log n}} + 9\lfloor \log^{10} n \rfloor - 11$$

$$C(n) = 3C(n/2) + C(n/3) + 5C(n/6) + n^2$$

$$D(n) = \max_{0 < k < n} (D(k) + D(n - k) + n)$$

$$E(n) = \frac{E(n-1)E(n-3)}{E(n-2)}$$

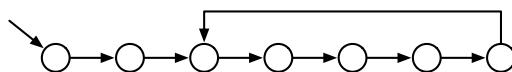
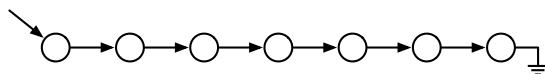
[Hint: Write out the first 20 terms.]

- (b) [5 pts] Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. ***Do not submit proofs***—just a sorted list of 16 functions—but you should do them anyway, just for practice.

Write $f(n) \ll g(n)$ to indicate that $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. We use the notation $\lg n = \log_2 n$.

n	$\lg n$	\sqrt{n}	3^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$3^{\sqrt{n}}$	$\sqrt{3^n}$
$3^{\lg n}$	$\lg(3^n)$	$3^{\lg \sqrt{n}}$	$3^{\sqrt{\lg n}}$
$\sqrt{3^{\lg n}}$	$\lg(3^{\sqrt{n}})$	$\lg \sqrt{3^n}$	$\sqrt{\lg(3^n)}$

3. Suppose you are given a pointer to the head of singly linked list. Normally, each node in the list has a pointer to the next element, and the last node's pointer is NULL. Unfortunately, your list might have been corrupted (by a bug in *somebody else's* code, of course), so that some node's pointer leads back to an earlier node in the list.



Top: A standard singly-linked list. Bottom: A corrupted singly-linked list.

Describe an algorithm¹ that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

¹Since you understand the course policies, you know what this phrase means. Right?

4. (a) Prove that any integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example:

$$42 = 3^4 - 3^3 - 3^2 - 3^1$$

$$25 = 3^3 - 3^1 + 3^0$$

$$17 = 3^3 - 3^2 - 3^0$$

- (b) Prove that any integer (positive, negative, or zero) can be written in the form $\sum_i (-2)^i$, where the exponents i are distinct non-negative integers. For example:

$$42 = (-2)^6 + (-2)^5 + (-2)^4 + (-2)^0$$

$$25 = (-2)^6 + (-2)^5 + (-2)^3 + (-2)^0$$

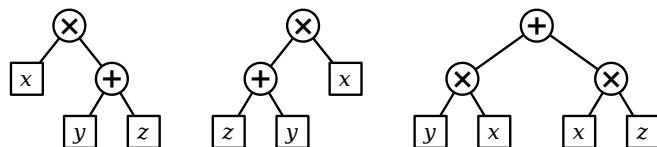
$$17 = (-2)^4 + (-2)^0$$

[Hint: Don't use weak induction. In fact, **never** use weak induction.]

5. An **arithmetic expression tree** is a binary tree where every leaf is labeled with a variable, every internal node is labeled with an arithmetic operation, and every internal node has exactly two children. For this problem, assume that the only allowed operations are $+$ and \times . Different leaves may or may not represent different variables.

Every arithmetic expression tree represents a function, transforming input values for the leaf variables into an output value for the root, by following two simple rules: (1) The value of any $+$ -node is the sum of the values of its children. (2) The value of any \times -node is the product of the values of its children.

Two arithmetic expression trees are **equivalent** if they represent the same function; that is, the same input values for the leaf variables always leads to the same output value at both roots. An arithmetic expression tree is in **normal form** if the parent of every $+$ -node (if any) is another $+$ -node.



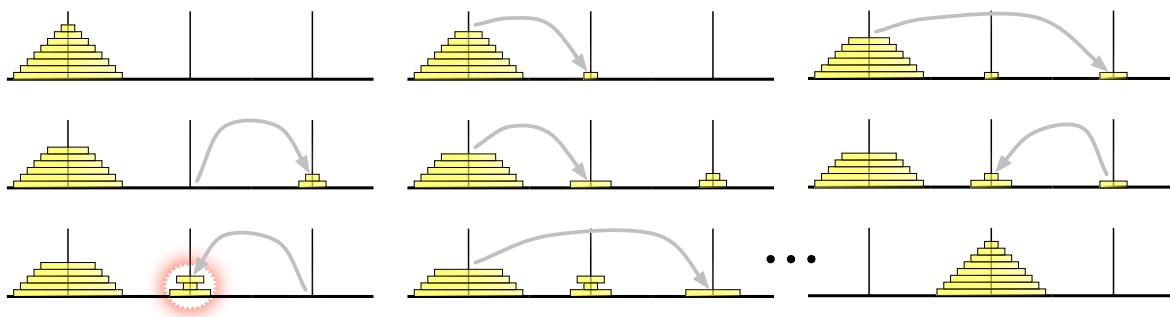
Three equivalent expression trees. Only the third is in normal form.

Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form.

*6. [Extra credit] You may be familiar with the story behind the famous Tower of Hanoi puzzle:

At the great temple of Benares, there is a brass plate on which three vertical diamond shafts are fixed. On the shafts are mounted n golden disks of decreasing size. At the time of creation, the god Brahma placed all of the disks on one pin, in order of size with the largest at the bottom. The Hindu priests unceasingly transfer the disks from peg to peg, one at a time, never placing a larger disk on a smaller one. When all of the disks have been transferred to the last pin, the universe will end.

Recently the temple at Benares was relocated to southern California, where the monks are considerably more laid back about their job. At the “Towers of Hollywood”, the golden disks have been replaced with painted plywood, and the diamond shafts have been replaced with Plexiglas. More importantly, the restriction on the order of the disks has been relaxed. While the disks are being moved, the bottom disk on any pin must be the *largest* disk on that pin, but disks further up in the stack can be in any order. However, after all the disks have been moved, they must be in sorted order again.



The Towers of Hollywood. The sixth move leaves the disks out of order.

Describe an algorithm that moves a stack of n disks from one pin to the another using the smallest possible number of moves. Exactly how many moves does your algorithm perform? [Hint: The Hollywood monks can bring about the end of the universe considerably faster than their Benaresian counterparts.]

CS 473: Undergraduate Algorithms, Spring 2009

Homework 1

Due Tuesday, February 3, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.

- The traditional Devonian/Cornish drinking song "The Barley Mow" has the following pseudolyrics, where $\text{container}[i]$ is the name of a container that holds 2^i ounces of beer. One version of the song uses the following containers: nippertkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. (Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.)

`BARLEYMOW(n):`

```

"Here's a health to the barley-mow, my brave boys,"
"Here's a health to the barley-mow!"

"We'll drink it out of the jolly brown bowl,"
"Here's a health to the barley-mow!"
"Here's a health to the barley-mow, my brave boys,"
"Here's a health to the barley-mow!"

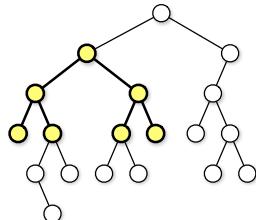
for  $i \leftarrow 1$  to  $n$ 
    "We'll drink it out of the container[ $i$ ], boys,"
    "Here's a health to the barley-mow!"
    for  $j \leftarrow i$  downto 1
        "The container[j],"
        "And the jolly brown bowl!"
        "Here's a health to the barley-mow!"
        "Here's a health to the barley-mow, my brave boys,"
        "Here's a health to the barley-mow!"

```

- Suppose each container name $\text{container}[i]$ is a single word, and you can sing four words a second. How long would it take you to sing $\text{BARLEYMOW}(n)$? (Give a tight asymptotic bound.) [Hint: Is 'barley-mow' one word or two? Does it matter?]
- If you want to sing this song for $n > 20$, you'll have to make up your own container names. To avoid repetition, these names will get progressively longer as n increases¹. Suppose $\text{container}[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing $\text{BARLEYMOW}(n)$? (Give a tight asymptotic bound.)
- Suppose each time you mention the name of a container, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $\text{container}[i]$. Assuming for purposes of this problem that you are at least 21 years old, exactly how many ounces of beer would you drink if you sang $\text{BARLEYMOW}(n)$? (Give an exact answer, not just an asymptotic bound.)

¹"We'll drink it out of the hemisemidemiyottapint, boys!"

2. For this problem, a *subtree* of a binary tree means any connected subgraph; a binary tree is *complete* if every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

3. (a) Describe and analyze a recursive algorithm to reconstruct a binary tree, given its preorder and postorder node sequences (as in Homework 0, problem 1).
- (b) Describe and analyze a recursive algorithm to reconstruct a binary tree, given its preorder and *inorder* node sequences.

CS 473: Undergraduate Algorithms, Spring 2009

Homework 10

Due Tuesday, May 5, 2009 at 11:59:59pm

- Groups of up to three students may submit a single, common solution. Please clearly write every group member's name and NetID on every page of your submission.
 - ***This homework is optional.*** If you submit solutions, they will be graded, and your overall homework grade will be the average of ten homeworks (Homeworks 0–10, dropping the lowest). If you do not submit solutions, your overall homework grade will be the average of *nine* homeworks (Homeworks 0–9, dropping the lowest).
-

1. 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.
2. PLANARCIRCUITSAT is a special case of CIRCUITSAT where the input circuit is drawn ‘nicely’ in the plane — no two wires cross, no two gates touch, and each wire touches only the gates it connects. (Not every circuit can be drawn this way!) As in the general CIRCUITSAT problem, we want to determine if there is an input that makes the circuit output TRUE?

Prove that PLANARCIRCUITSAT is NP-complete. [*Hint: XOR.*]

3. For each problem below, either describe a polynomial-time algorithm or prove that the problem is NP-complete.
 - (a) A ***double-Eulerian*** circuit in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G , does G have a ***double-Eulerian*** circuit?
 - (b) A ***double-Hamiltonian*** circuit in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G , does G have a ***double-Hamiltonian*** circuit?

CS 473: Undergraduate Algorithms, Spring 2009

Homework 2

Written solutions due Tuesday, February 10, 2009 at 11:59:59pm.

- Roughly 1/3 of the students will give oral presentations of their solutions to the TAs. *Please check Compass to check whether you are supposed give an oral presentation for this homework.* Please see the course web page for further details.
 - Groups of up to three students may submit a common solution. Please clearly write every group member's name and NetID on every page of your submission.
 - Please start your solution to each numbered problem on a new sheet of paper. Please *don't* staple solutions for different problems together.
 - **For this homework only:** These homework problems ask you to describe recursive backtracking algorithms for various problems. *Don't* use memoization or dynamic programming to make your algorithms more efficient; you'll get to do that on HW3. *Don't* analyze the running times of your algorithms. The **only** things you should submit for each problem are (1) a description of your recursive algorithm, and (2) a brief justification for its correctness.
-

1. A **basic arithmetic expression** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$\begin{aligned}
 & 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
 & ((1 + 1) \times (1 + 1 + 1 + 1 + 1)) + ((1 + 1) \times (1 + 1)) \\
 & (1 + 1) \times (1 + 1 + 1 + 1 + 1 + 1 + 1) \\
 & (1 + 1) \times (((1 + 1 + 1) \times (1 + 1)) + 1)
 \end{aligned}$$

Describe a recursive algorithm to compute, given an integer n as input, the minimum number of 1's in a basic arithmetic expression whose value is n . The number of parentheses doesn't matter, just the number of 1's. For example, when $n = 14$, your algorithm should return 8, for the final expression above.

2. A sequence $A = \langle a_1, a_2, \dots, a_n \rangle$ is **bitonic** if there is an index i with $1 < i < n$, such that the prefix $\langle a_1, a_2, \dots, a_i \rangle$ is strictly increasing and the suffix $\langle a_i, a_{i+1}, \dots, a_n \rangle$ is strictly decreasing. In particular, a bitonic sequence must contain at least three elements.

Describe a recursive algorithm to compute, given a sequence A , the length of the longest bitonic subsequence of A .

3. A *palindrome* is a string that reads the same forwards and backwards, like x, pop, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. Any string can be broken into sequence of palindromes. For example, the string bubbbaseesabana ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example:

bub + baseesab + anana
b + u + bb + a + sees + aba + nan + a
b + u + bb + a + sees + a + b + anana
b + u + b + b + a + s + e + e + s + a + b + a + n + a + n + a

Describe a recursive algorithm to compute the minimum number of palindromes that make up a given input string. For example, given the input string bubbbaseesabana, your algorithm would return the integer 3.

CS 473: Undergraduate Algorithms, Spring 2009

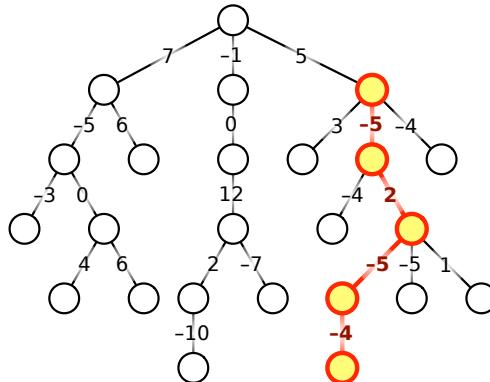
Homework 3

Written solutions due Tuesday, February 17, 2009 at 11:59:59pm.

1. Redo Homework 2, but now with dynamic programming!
 - (a) Describe and analyze an efficient algorithm to compute the minimum number of 1's in a basic arithmetic expression whose value is a given positive integer.
 - (b) Describe and analyze an efficient algorithm to compute the length of the longest bitonic subsequence of a given input sequence.
 - (c) Describe and analyze an efficient algorithm to compute the minimum number of palindromes that make up a given input string.

Please see Homework 2 for more detailed descriptions of each problem. ***Solutions for Homework 2 will be posted Friday, after the HW2 oral presentations.*** You may (and should!) use anything from those solutions without justification.

2. Let T be a rooted tree with integer weights on its edges, which could be positive, negative, or zero. Design an algorithm to find the minimum-length path from a node in T down to one of its descendants. The length of a path is the sum of the weights of its edges. For example, given the tree shown below, your algorithm should return the number -12 . For full credit, your algorithm should run in $O(n)$ time.



The minimum-weight downward path in this tree has weight -12 .

3. Describe and analyze an efficient algorithm to compute the longest common subsequence of *three* given strings. For example, given the input strings EPIDEMIOLOGIST, REFRIGERATION, and SUPERCALIFRAGILISTICEXPIALODOCIOUS, your algorithm should return the number 5, because the longest common subsequence is EIEIO.

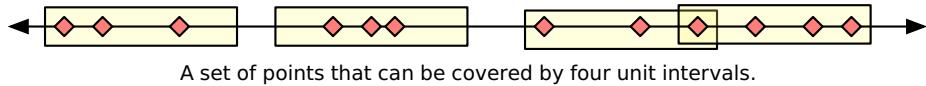
CS 473: Undergraduate Algorithms, Spring 2009

Homework 3½

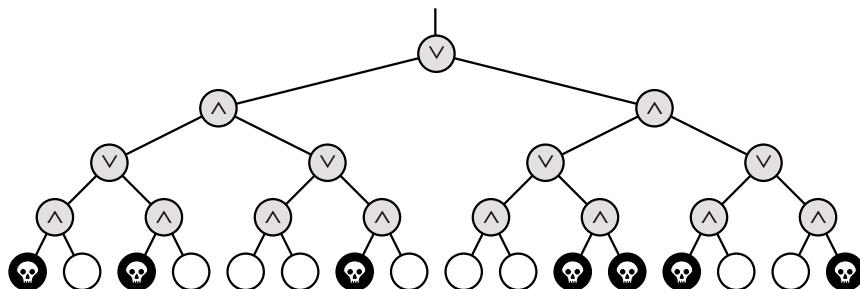
Practice only

1. After graduating from UIUC, you are hired by a mobile phone company to plan the placement of new cell towers along a long, straight, nearly-deserted highway out west. Each cell tower can transmit the same fixed distance from its location. Federal law requires that any building along the highway must be within the broadcast range of at least one tower. On the other hand, your company wants to build as few towers as possible. Given the locations of the buildings, where should you build the towers?

More formally, suppose you are given a set $X = \{x_1, x_2, \dots, x_n\}$ of points on the real number line. Describe an algorithm to compute the minimum number of intervals of length 1 that can cover all the points in X . For full credit, your algorithm should run in $O(n \log n)$ time.



2. (a) The *left spine* of a binary tree is a path starting at the root and following only left-child pointers down to a leaf. What is the expected number of nodes in the left spine of an n -node treap?
- (b) What is the expected number of leaves in an n -node treap? [Hint: What is the probability that in an n -node treap, the node with k th smallest search key is a leaf?]
- (c) Prove that the expected number of proper descendants of any node in a treap is exactly equal to the expected depth of that node.
3. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- (a) Describe and analyze a deterministic algorithm to determine whether or not you can win. [*Hint: This is easy!*]
- (b) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $O(3^n)$ expected time. [*Hint: Consider the case $n = 1$.*]
- *(c) Describe a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. [*Hint: You may not need to change your algorithm from part (b) at all!*]

CS 473: Undergraduate Algorithms, Spring 2009

Homework 3

Written solutions due Tuesday, March 2, 2009 at 11:59:59pm.

1. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue containing the empty set.
- **FINDMIN(Q)**: Return the smallest element of Q (if any).
- **DELETEMIN(Q)**: Remove the smallest element in Q (if any).
- **INSERT(Q, x)**: Insert element x into Q , if it is not already there.
- **DECREASEKEY(Q, x, y)**: Replace an element $x \in Q$ with a smaller key y . (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x .
- **DELETE(Q, x)**: Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x .
- **MELD(Q_1, Q_2)**: Return a new priority queue containing all the elements of Q_1 and Q_2 ; this operation destroys Q_1 and Q_2 .

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```

MELD( $Q_1, Q_2$ ):
    if  $Q_1$  is empty return  $Q_2$ 
    if  $Q_2$  is empty return  $Q_1$ 
    if  $key(Q_1) > key(Q_2)$ 
        swap  $Q_1 \leftrightarrow Q_2$ 
    with probability 1/2
         $left(Q_1) \leftarrow \text{MELD}(left(Q_1), Q_2)$ 
    else
         $right(Q_1) \leftarrow \text{MELD}(right(Q_1), Q_2)$ 
    return  $Q_1$ 

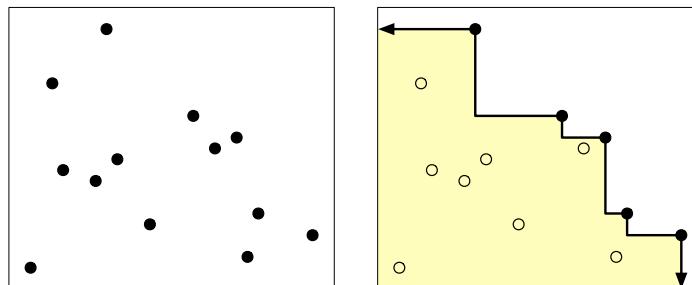
```

- (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (*not* just those constructed by the operations listed above), the expected running time of $\text{MELD}(Q_1, Q_2)$ is $O(\log n)$, where n is the total number of nodes in both trees. [Hint: How long is a random root-to-leaf path in an n -node binary tree if each left/right choice is made with equal probability?]
- (b) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ expected time.)

2. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A *heater* is a priority search tree in which the priorities are given by the user, and the search keys are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is the ‘opposite’ of a treap.

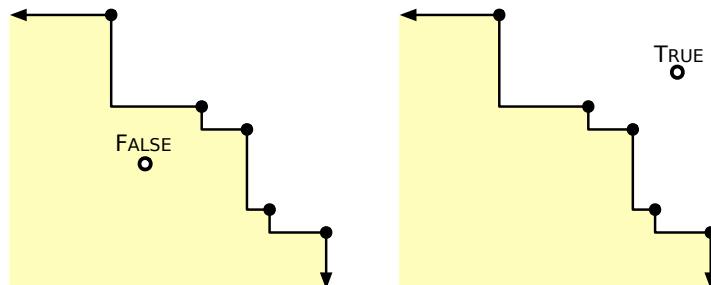
The following problems consider an n -node heater T whose node priorities are the integers from 1 to n . We identify nodes in T by their priorities; thus, ‘node 5’ means the node in T with priority 5. The min-heap property implies that node 1 is the root of T . Finally, let i and j be integers with $1 \leq i < j \leq n$.

- (a) **Prove** that in a random permutation of the $(i + 1)$ -element set $\{1, 2, \dots, i, j\}$, elements i and j are adjacent with probability $2/(i + 1)$.
 - (b) **Prove** that node i is an ancestor of node j with probability $2/(i + 1)$. [Hint: Use part (a)!]
 - (c) What is the probability that node i is a descendant of node j ? [Hint: Don’t use part (a)!]
 - (d) What is the exact expected depth of node j ?
3. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.



A set of points in the plane and its staircase (shaded).

- (a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.
- (b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm ABOVE?(x, y) that returns TRUE if the point (x, y) is above the staircase, or FALSE otherwise. Your data structure should use $O(n)$ space, and your ABOVE? algorithm should run in $O(\log n)$ time.



Two staircase queries.

CS 473: Undergraduate Algorithms, Spring 2009

Homework 5

Written solutions due Tuesday, March 9, 2009 at 11:59:59pm.

1. Remember the difference between stacks and queues? Good.
 - (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard methods PUSH and POP.
 - (b) A *quack* is an abstract data type that combines properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:
 - **Push:** add a new item to the left end of the list;
 - **Pop:** remove the item on the left end of the list;
 - **Pull:** remove the item on the right end of the list.
 Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any push, pop, or pull operation is $O(1)$. Again, you are *only* allowed to access the stacks through the standard methods PUSH and POP.

2. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

Describe and analyze an algorithm to purge an *arbitrary* n -node dirty binary search tree in $O(n)$ time, using at most $O(\log n)$ space (in addition to the tree itself). Don't forget to include the recursion stack in your space bound. An algorithm that uses $\Theta(n)$ additional space in the worst case is worth half credit.

3. Some applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Maintaining these secondary structures usually complicates algorithms for keeping the top-level search tree balanced.

Let T be an arbitrary binary tree. Suppose every node v in T stores a secondary structure of size $O(\text{size}(v))$, which can be built in $O(\text{size}(v))$ time, where $\text{size}(v)$ denotes the number of descendants of v . Performing a rotation at any node v now requires $O(\text{size}(v))$ time, because we have to rebuild one of the secondary structures.

 - (a) [1 pt] Overall, how much space does this data structure use *in the worst case*?
 - (b) [1 pt] How much space does this structure use if the primary search tree T is perfectly balanced?
 - (c) [2 pts] Suppose T is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. [*Hint: This is easy!*]

- (d) [3 pts] Now suppose T is a scapegoat tree, and that rebuilding the subtree rooted at v requires $\Theta(\text{size}(v) \log \text{size}(v))$ time (because we also have to rebuild the secondary structures at every descendant of v). What is the *amortized* cost of inserting a new element into T ?
- (e) [3 pts] Finally, suppose T is a treap. What's the worst-case *expected* time for inserting a new element into T ?

CS 473: Undergraduate Algorithms, Spring 2009

Homework 6

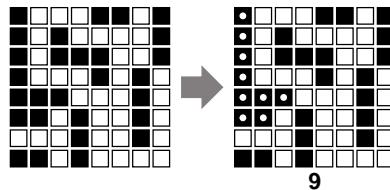
Written solutions due Tuesday, March 17, 2009 at 11:59:59pm.

1. Let G be an undirected graph with n nodes. Suppose that G contains two nodes s and t , such that every path from s to t contains more than $n/2$ edges.
 - (a) Prove that G must contain a vertex v that lies on *every* path from s to t .
 - (b) Describe an algorithm that finds such a vertex v in $O(V + E)$ time.

2. Suppose you are given a graph G with weighted edges and a minimum spanning tree T of G .
 - (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.
 - (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

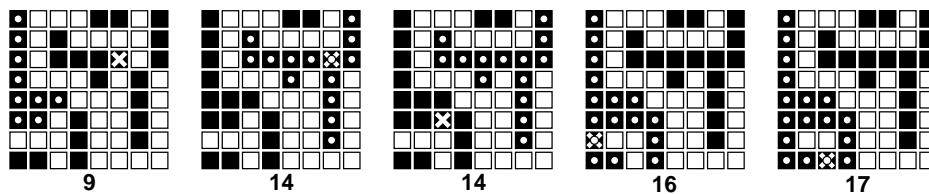
In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. [Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]

3. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.
 For example, given the bitmap below as input, your algorithm should return the number 9, because the largest connected black component (marked with white dots on the right) contains nine pixels.



- (b) Design and analyze an algorithm $\text{BLACKEN}(i, j)$ that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.



- (c) What is the *worst-case* running time of your BLACKEN algorithm?

CS 473: Undergraduate Algorithms, Spring 2009

Homework 6½

Practice only—do not submit solutions

1. In class last Tuesday, we discussed Ford's generic shortest-path algorithm—relax arbitrary tense edges until no edge is tense. This problem asks you to fill in part of the proof that this algorithm is correct.
 - (a) Prove that after *every* call to RELAX, for every vertex v , either $dist(v) = \infty$ or $dist(v)$ is the total weight of some path from s to v .
 - (b) Prove that for every vertex v , when the generic algorithm halts, either $pred(v) = \text{NULL}$ and $dist(v) = \infty$, or $dist(v)$ is the total weight of the predecessor chain ending at v :

$$s \rightarrow \dots \rightarrow pred(pred(v)) \rightarrow pred(v) \rightarrow v.$$

2. Describe a modification of Shimbels shortest-path algorithm that actually computes a negative-weight cycle if any such cycle is reachable from s , or a shortest-path tree rooted at s if there is no such cycle. Your modified algorithm should still run in $O(VE)$ time.
3. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

Suppose one of your customers wants to fly from city X to city Y . Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. [Hint: Modify the input data and apply Dijkstra's algorithm.]

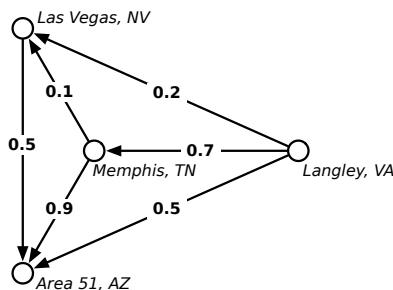
CS 473: Undergraduate Algorithms, Spring 2009

Homework 6 $\frac{3}{4}$

Practice only—do not submit solutions

1. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t .



For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.)

2. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets V_1, V_2, \dots, V_k ; that is, every vertex of G belongs to exactly one subset V_i . For each i and j , let $\delta(i, j)$ denote the minimum shortest-path distance between any vertex in V_i and any vertex in V_j :

$$\delta(i, j) = \min\{\text{dist}(u, v) \mid u \in V_i \text{ and } v \in V_j\}.$$

Describe an algorithm to compute $\delta(i, j)$ for all i and j in time $O(VE + kE \log E)$. The output from your algorithm is a $k \times k$ array.

3. Recall¹ that a deterministic finite automaton (DFA) is formally defined as a tuple $M = (\Sigma, Q, q_0, F, \delta)$, where the finite set Σ is the input alphabet, the finite set Q is the set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of final (accepting) states, and $\delta: Q \times \Sigma \rightarrow Q$ is the transition function. Equivalently, a DFA is a directed (multi-)graph with labeled edges, such that each symbol in Σ is the label of exactly one edge leaving any vertex. There is a special ‘start’ vertex q_0 , and a subset of the vertices are marked as ‘accepting states’. Any string in Σ^* describes a unique walk starting at q_0 .

Stephen Kleene² proved that the language accepted by any DFA is identical to the language described by some regular expression. This problem asks you to develop a variant of the Floyd-Warshall all-pairs shortest path algorithm that computes a regular expression that is equivalent to the language accepted by a given DFA.

Suppose the input DFA M has n states, numbered from 1 to n , where (without loss of generality) the start state is state 1. Let $L(i, j, r)$ denote the set of all words that describe walks in M from state i to state j , where every intermediate state lies in the subset $\{1, 2, \dots, r\}$; thus, the language accepted by the DFA is exactly

$$\bigcup_{q \in F} L(1, q, n).$$

Let $R(i, j, r)$ be a regular expression that describes the language $L(i, j, r)$.

- (a) What is the regular expression $R(i, j, 0)$?
- (b) Write a recurrence for the regular expression $R(i, j, r)$ in terms of regular expressions of the form $R(i', j', r - 1)$.
- (c) Describe a polynomial-time algorithm to compute $R(i, j, n)$ for all states i and j . (Assume that you can concatenate two regular expressions in $O(1)$ time.)

¹No, really, you saw this in CS 273/373.

²Pronounced ‘clay knee’, not ‘clean’ or ‘clean-ee’ or ‘clay-nuh’ or ‘dimaggio’.

CS 473: Undergraduate Algorithms, Spring 2009

Homework 7

Due Tuesday, April 14, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.
-
1. A graph is *bipartite* if its vertices can be colored black or white such that every edge joins vertices of two different colors. A graph is *d-regular* if every vertex has degree d . A *matching* in a graph is a subset of the edges with no common endpoints; a matching is *perfect* if it touches every vertex.
 - (a) Prove that every regular bipartite graph contains a perfect matching.
 - (b) Prove that every d -regular bipartite graph is the union of d perfect matchings.
 2. Let $G = (V, E)$ be a directed graph where for each vertex v , the in-degree of v and out-degree of v are equal. Let u and v be two vertices G , and suppose G contains k edge-disjoint paths from u to v . Under these conditions, must G also contain k edge-disjoint paths from v to u ? Give a proof or a counterexample with explanation.
 3. A flow f is called *acyclic* if the subgraph of directed edges with positive flow contains no directed cycles. A flow is *positive* if its value is greater than 0.
 - (a) A *path flow* assigns positive values only to the edges of one simple directed path from s to t . Prove that every positive acyclic flow can be written as the sum of a finite number of path flows.
 - (b) Describe a flow in a directed graph that *cannot* be written as the sum of path flows.
 - (c) A *cycle flow* assigns positive values only to the edges of one simple directed cycle. Prove that every flow can be written as the sum of a finite number of path flows and cycle flows.
 - (d) Prove that for any flow f , there is an acyclic flow with the same value as f . (In particular, this implies that some maximum flow is acyclic.)

CS 473: Undergraduate Algorithms, Spring 2009

Homework 8

Due Tuesday, April 21, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.

- A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover all the vertices. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that none exists. [*Hint: Use bipartite matching!*]
- Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A . For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

- Ad-hoc networks* are made up of cheap, low-powered wireless devices. In principle¹, these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that several simple devices could be distributed randomly in the area of interest (for example, dropped from an airplane), and then they would somehow automatically configure themselves into an efficiently functioning wireless network.

The devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D .

We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit all its information to some other *backup* device within its communication range. To improve reliability, we require each device x to have k potential backup devices, all within distance D of x ; we call these k devices the **backup set** of x . Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

Suppose we are given the communication distance D , parameters b and k , and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j . Describe and analyze an algorithm that either computes a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or correctly reports that no good collection of backup sets exists.

¹but not so much in practice

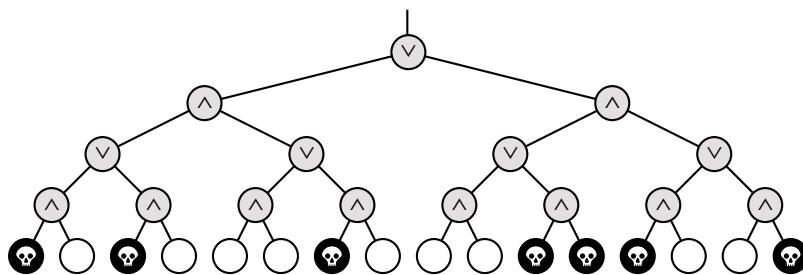
CS 473: Undergraduate Algorithms, Spring 2009

Homework 9

Due Tuesday, April 28, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.

- Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it is white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it is worth playing or not as follows. Imagine that the nodes at even levels (where it is your turn) are OR gates, the nodes at odd levels (where it is Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black leaves stand represent TRUE and FALSE inputs, respectively. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- (a) Describe and analyze a deterministic algorithm to determine whether or not you can win.
[Hint: This is easy.]
 - (b) Prove that *every* deterministic algorithm must examine *every* leaf of the tree in the worst case. Since there are 4^n leaves, this implies that any deterministic algorithm must take $\Omega(4^n)$ time in the worst case. Use an adversary argument; in other words, assume that Death cheats.
 - (c) *[Extra credit]* Describe a *randomized* algorithm that runs in $O(3^n)$ expected time.
-
- We say that an array $A[1..n]$ is *k-sorted* if it can be divided into k blocks, each of size n/k , such that the elements in each block are larger than the elements in earlier blocks, and smaller than elements in later blocks. The elements within each block need not be sorted.

For example, the following array is 4-sorted:

1	2	4	3		7	6	8	5		10	11	9	12		15	13	16	14
---	---	---	---	--	---	---	---	---	--	----	----	---	----	--	----	----	----	----

- (a) Describe an algorithm that k -sorts an arbitrary array in time $O(n \log k)$.
- (b) Prove that any comparison-based k -sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.
- (c) Describe an algorithm that completely sorts an already k -sorted array in time $O(n \log(n/k))$.
- (d) Prove that any comparison-based algorithm to completely sort a k -sorted array requires $\Omega(n \log(n/k))$ comparisons in the worst case.

In all cases, you can assume that n/k is an integer.

3. UIUC has just finished constructing the new Reingold Building, the tallest dormitory on campus. In order to determine how much insurance to buy, the university administration needs to determine the highest safe floor in the building. A floor is considered *safe* if a ~~drunk~~ student *an egg* can fall from a window on that floor and land without breaking; if the egg breaks, the floor is considered *unsafe*. Any floor that is higher than an unsafe floor is also considered unsafe. The only way to determine whether a floor is safe is to drop an egg from a window on that floor.

You would like to find the lowest unsafe floor L by performing as few tests as possible; unfortunately, you have only a very limited supply of eggs.

- (a) Prove that if you have only one egg, you can find the lowest unsafe floor with L tests. [*Hint: Yes, this is trivial.*]
- (b) Prove that if you have only one egg, you must perform at least L tests in the worst case. In other words, prove that your algorithm from part (a) is optimal. [*Hint: Use an adversary argument.*]
- (c) Describe an algorithm to find the lowest unsafe floor using *two* eggs and only $O(\sqrt{L})$ tests. [*Hint: Ideally, each egg should be dropped the same number of times. How many floors can you test with n drops?*]
- (d) Prove that if you start with two eggs, you must perform at least $\Omega(\sqrt{L})$ tests in the worst case. In other words, prove that your algorithm from part (c) is optimal.
- *(e) [**Extra credit!**] Describe an algorithm to find the lowest unsafe floor using k eggs, using as few tests as possible, and prove your algorithm is optimal for all values of k .

CS 473: Undergraduate Algorithms, Spring 2009

Head Banging Session 0

January 20 and 21, 2009

1. Solve the following recurrences. If base cases are provided, find an *exact* closed-form solution. Otherwise, find a solution of the form $\Theta(f(n))$ for some function f .

- **Warmup:** You should be able to solve these almost as fast as you can write down the answers.

- (a) $A(n) = A(n - 1) + 1$, where $A(0) = 0$.
- (b) $B(n) = B(n - 5) + 2$, where $B(0) = 17$.
- (c) $C(n) = C(n - 1) + n^2$
- (d) $D(n) = 3D(n/2) + n^2$
- (e) $E(n) = 4E(n/2) + n^2$
- (f) $F(n) = 5F(n/2) + n^2$

- **Real practice:**

- (a) $A(n) = A(n/3) + 3A(n/5) + A(n/15) + n$
- (b) $B(n) = \min_{0 < k < n} (B(k) + B(n - k) + n)$
- (c) $C(n) = \max_{n/4 < k < 3n/4} (C(k) + C(n - k) + n)$
- (d) $D(n) = \max_{0 < k < n} (D(k) + D(n - k) + k(n - k))$, where $D(1) = 0$
- (e) $E(n) = 2E(n - 1) + E(n - 2)$, where $E(0) = 1$ and $E(1) = 2$
- (f) $F(n) = \frac{1}{F(n - 1)F(n - 2)}$, where $F(0) = 1$ and $F(2) = 2$
- (g) $G(n) = nG(\sqrt{n}) + n^2$

2. The *Fibonacci numbers* F_n are defined recursively as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for every integer $n \geq 2$. The first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,

Prove that any non-negative integer can be written as the sum of distinct *non-consecutive* Fibonacci numbers. That is, if any Fibonacci number F_n appears in the sum, then its neighbors F_{n-1} and F_{n+1} do not. For example:

$$\begin{aligned}
 88 &= 55 + 21 + 8 + 3 + 1 &= F_{10} + F_8 + F_6 + F_4 + F_2 \\
 42 &= &= F_9 + F_6 \\
 17 &= 13 + 3 + 1 &= F_7 + F_4 + F_2
 \end{aligned}$$

3. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. Pretty please.

1. An *inversion* in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n -element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward).

Describe and analyze an algorithm to count the number of inversions in an n -element array in $O(n \log n)$ time.

2. (a) Prove that the following algorithm actually sorts its input.

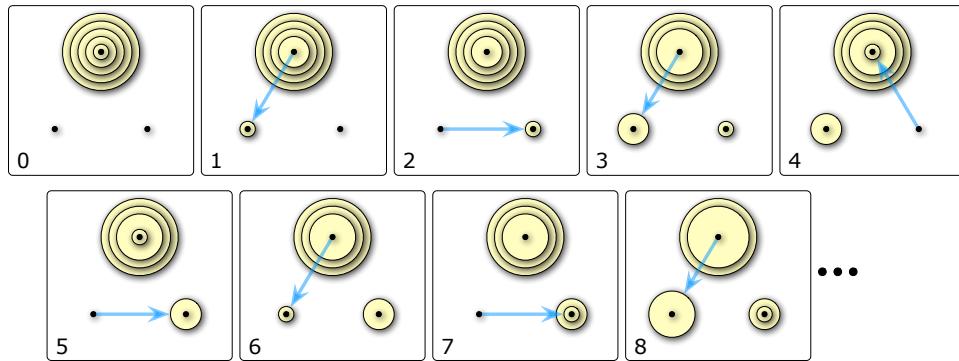
```

STOOGESORT( $A[0..n-1]$ ) :
  if  $n = 2$  and  $A[0] > A[1]$ 
    swap  $A[0] \leftrightarrow A[1]$ 
  else if  $n > 2$ 
     $m = \lceil 2n/3 \rceil$ 
    STOOGESORT( $A[0..m-1]$ )
    STOOGESORT( $A[n-m..n-1]$ )
    STOOGESORT( $A[0..m-1]$ )
  
```

- (b) Would STOOGESORT still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.
- (c) State a recurrence (including base case(s)) for the number of comparisons executed by STOOGESORT.
- (d) Solve this recurrence. [Hint: Ignore the ceiling.]
- (e) **To think about on your own:** Prove that the number of swaps executed by STOOGESORT is at most $\binom{n}{2}$.

3. Consider the following restricted variants of the Tower of Hanoi puzzle. In each problem, the needles are numbered 0, 1, and 2, and your task is to move a stack of n disks from needle 1 to needle 2.

- (a) Suppose you are forbidden to move any disk directly between needle 1 and needle 2; every move must involve needle 0. Describe an algorithm to solve this version of the puzzle using as few moves as possible. Exactly how many moves does your algorithm make?
- (b) Suppose you are only allowed to move disks from needle 0 to needle 2, from needle 2 to needle 1, or from needle 1 to needle 0. Equivalently, Suppose the needles are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle using as few moves as possible. Exactly how many moves does your algorithm make?



The first eight moves in a counterclockwise Towers of Hanoi solution

- ★(c) Finally, suppose you are forbidden to move any disk directly from needle 1 to 2, but any other move is allowed. Describe an algorithm to solve this version of the puzzle using as few moves as possible. *Exactly* how many moves does your algorithm make?

[Hint: This version is considerably harder than the other two.]

CS 473: Undergraduate Algorithms, Spring 2009

HBS 10

1. Consider the following problem, called *BOX-DEPTH*: 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 *BOX-DEPTH* to *MAX-CLIQUE*.
 - (b) Describe and analyze a polynomial-time algorithm for *BOX-DEPTH*. [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 $P = NP$?
2. 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.
3. Prove that the following problems are NP-complete.
 - (a) Given an undirected graph G , does G have a spanning tree in which every node has degree at most 17?
 - (b) Given an undirected graph G , does G have a spanning tree with at most 42 leaves?

CS 473: Undergraduate Algorithms, Spring 2009

HBS 11

1. You step in a party with a camera in your hand. Each person attending the party has some friends there. You want to have exactly one picture of each person in your camera. You want to use the following protocol to collect photos. At each step, the person that has the camera in his hand takes a picture of one of his/her friends and pass the camera to him/her. Of course, you only like the solution if it finishes when the camera is in your hand. Given the friendship matrix of the people in the party, design a polynomial algorithm that decides whether this is possible, or prove that this decision problem is NP-hard.
2. A boolean formula is in disjunctive normal form (DNF) if it is a disjunctions (OR) of several clauses, each of which is the conjunction (AND) of several literals, each of which is either a variable or its negation. For example:

$$(a \wedge b \wedge c) \vee (\bar{a} \wedge b) \vee (\bar{c} \wedge x)$$

Given a DNF formula give a polynomial algorithm to check whether it is satisfiable or not. Why this does not imply $P = NP$.

3. Prove that the following problems are NP-complete.
 - (a) Given an undirected graph G , does G have a spanning tree in which every node has degree at most 17?
 - (b) Given an undirected graph G , does G have a spanning tree with at most 42 leaves?

CS 473: Undergraduate Algorithms, Spring 2009

HBS 2

1. Consider two horizontal lines l_1 and l_2 in the plane. There are n points on l_1 with x -coordinates $A = a_1, a_2, \dots, a_n$ and there are n points on l_2 with x -coordinates $B = b_1, b_2, \dots, b_n$. Design an algorithm to compute, given A and B , a largest set S of non-intersecting line segments subject to the following restrictions:
 - (a) Any segment in S connects a_i to b_i for some $i (1 \leq i \leq n)$.
 - (b) Any two segments in S do not intersect.
2. Consider a $2^n \times 2^n$ chess board with one (arbitrarily chosen) square removed. Prove that any such chessboard can be tiled without gaps or overlaps by L-shaped pieces of 3 squares each. Can you give an algorithm to do the tiling?
3. Given a string of letters $Y = y_1 y_2 \dots y_n$, a segmentation of Y is a partition of its letters into contiguous blocks of letters (also called words). Each word has a quality that can be computed by a given oracle (e.g. you can call `quality("meet")` to get the quality of the word "meet"). The quality of a segmentation is equal to the sum over the qualities of its words. Each call to the oracle takes linear time in terms of the argument; that is `quality(S)` takes $O(|S|)$.

Using the given oracle, give an algorithm that takes a string Y and computes a segmentation of maximum total quality.

CS 473: Undergraduate Algorithms, Spring 2009

HBS 3

1. Change your recursive solutions for the following problems to efficient algorithms (Hint: use dynamic programming!).
 - (a) Consider two horizontal lines l_1 and l_2 in the plane. There are n points on l_1 with x -coordinates $A = a_1, a_2, \dots, a_n$ and there are n points on l_2 with x -coordinates $B = b_1, b_2, \dots, b_n$. Design an algorithm to compute, given A and B , a largest set S of non-intersecting line segments subject to the following restrictions:
 - i. Any segment in S connects a_i to b_i for some i ($1 \leq i \leq n$).
 - ii. Any two segments in S do not intersect.
 - (b) Given a string of letters $Y = y_1y_2\dots y_n$, a segmentation of Y is a partition of its letters into contiguous blocks of letters (also called words). Each word has a quality that can be computed by a given oracle (e.g. you can call *quality("meet")* to get the quality of the word "meet"). The quality of a segmentation is equal to the sum over the qualities of its words. Each call to the oracle takes linear time in terms of the argument; that is *quality*(S) takes $O(|S|)$.Using the given oracle, give an algorithm that takes a string Y and computes a segmentation of maximum total quality.
2. Give a polynomial time algorithm which given two strings A and B returns the longest sequence S that is a subsequence of A and B .
3. Consider a rooted tree T . Assume the root has a message to send to all nodes. At the beginning only the root has the message. If a node has the message, it can forward it to one of its children at each time step. Design an algorithm to find the minimum number of time steps required for the message to be delivered to all nodes.

CS 473: Undergraduate Algorithms, Spring 2009

HBS 3.5

1. Say you are given n jobs to run on a machine. Each job has a start time and an end time. If a job is chosen to be run, then it must start at its start time and end at its end time. Your goal is to accept as many jobs as possible, regardless of the job lengths, subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n . You may assume for simplicity that no two jobs have the same start or end times, but the start time and end time of two jobs can overlap.
2. Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, without knowing the length of the stream in advance. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself).
3. Design and analyze an algorithm that return a permutation of the integers $\{1, 2, \dots, n\}$ chosen uniformly at random.

CS 473: Undergraduate Algorithms, Spring 2009

HBS 4

1. Let x and y be two elements of a set S whose ranks differ by exactly r . Prove that in a treap for S , the expected length of the unique path from x to y is $O(\log r)$
2. Consider the problem of making change for n cents using the least number of coins.
 - (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
 - (b) Suppose that the available coins have the values c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.
 - (c) Give a set of 4 coin values for which the greedy algorithm does not yield an optimal solution, show why.
 - (d) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of coin values.
3. A heater is a sort of dual treap, in which the priorities of the nodes are given, but their search keys are generated independently and uniformly from the unit interval $[0,1]$. You can assume all priorities and keys are distinct. Describe algorithms to perform the operations INSERT and DELETEMIN in a heater. What are the expected worst-case running times of your algorithms? In particular, can you express the expected running time of INSERT in terms of the priority rank of the newly inserted item?

CS 473: Undergraduate Algorithms, Spring 2009

HBS 5

1. Recall that the staircase of a set of points consists of the points with no other point both above and to the right. Describe a method to maintain the staircase as new points are added to the set. Specifically, describe and analyze a data structure that stores the staircase of a set of points, and an algorithm $INSERT(x, y)$ that adds the point (x, y) to the set and returns *TRUE* or *FALSE* to indicate whether the staircase has changed. Your data structure should use $O(n)$ space, and your $INSERT$ algorithm should run in $O(\log n)$ amortized time.
2. In some applications, we do not know in advance how much space we will require. So, we start the program by allocating a (dynamic) table of some fixed size. Later, as new objects are inserted, we may have to allocate a larger table and copy the previous table to it. So, we may need more than $O(1)$ time for copying. In addition, we want to keep the table size small enough, avoiding a very large table to keep only few items. One way to manage a dynamic table is by the following rules:
 - (a) Double the size of the table if an item is inserted into a full table
 - (b) Halve the table size if a deletion causes the table to become less than 1/4 fullShow that, in such a dynamic table we only need $O(1)$ amortized time, per operation.
3. Consider a stack data structure with the following operations:
 - $PUSH(x)$: adds the element x to the top of the stack
 - POP : removes and returns the element that is currently on top of the stack (if the stack is non-empty)
 - $SEARCH(x)$: repeatedly removes the element on top of the stack until x is found or the stack becomes empty

What is the amortized cost of an operation?

CS 473: Undergraduate Algorithms, Spring 2009

HBS 6

1. Let G be a connected graph and let v be a vertex in G . Show that T is both a DFS tree and a BFS tree rooted at v , then $G = T$.
2. An Euler tour of a graph G is a walk that starts from a vertex v , visits every edge of G exactly once and gets back to v . Prove that G has an Euler tour if and only if all the vertices of G has even degrees. Can you give an efficient algorithm to find an Euler tour of such a graph.
3. You are helping a group of ethnographers analyze some oral history data they have collected by interviewing members of a village to learn about the lives of people lived there over the last two hundred years. From the interviews, you have learned about a set of people, all now deceased, whom we will denote P_1, P_2, \dots, P_n . The ethnographers have collected several facts about the lifespans of these people, of one of the following forms:
 - (a) P_i died before P_j was born.
 - (b) P_i and P_j were both alive at some moment.

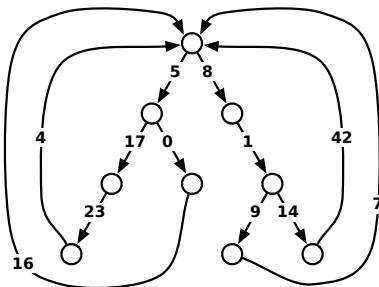
Naturally, the ethnographers are not sure that their facts are correct; memories are not so good, and all this information was passed down by word of mouth. So they'd like you to determine whether the data they have collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they have learned simultaneously hold.

Describe and analyze an algorithm to answer the ethnographers' problem. Your algorithm should either output possible dates of birth and death that are consistent with all the stated facts, or it should report correctly that no such dates exist.

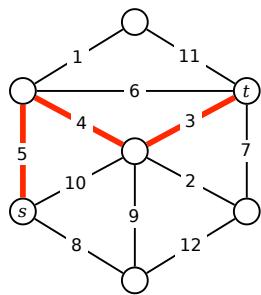
CS 473: Undergraduate Algorithms, Spring 2009

HBS 6.5

1. (a) Describe and analyze an algorithm to find the *second smallest spanning tree* of a given graph G , that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.
 *(b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k , the k smallest spanning trees of G .
2. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.



- (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?
 (b) Describe and analyze a faster algorithm.
3. Consider a path between two vertices s and t in an undirected weighted graph G . The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t . (If there are no paths from s to t , the bottleneck distance between s and t is ∞ .)



The bottleneck distance between s and t is 5.

Describe and analyze an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

CS 473: Undergraduate Algorithms, Spring 2009

HBS 6.55

1. Suppose you are given a directed graph $G = (V, E)$ with non-negative edge lengths; $\ell(e)$ is the length of $e \in E$. You are interested in the shortest path distance between two given locations/nodes s and t . It has been noticed that the existing shortest path distance between s and t in G is not satisfactory and there is a proposal to add exactly one edge to the graph to improve the situation. The candidate edges from which one has to be chosen is given by $E' = \{e_1, e_2, \dots, e_k\}$ and you can assume that $E \cup E' = \emptyset$. The length of the e_i is $\alpha_i \geq 0$. Your goal is figure out which of these k edges will result in the most reduction in the shortest path distance from s to t . Describe an algorithm for this problem that runs in time $O((m + n) \log n + k)$ where $m = |E|$ and $n = |V|$. Note that one can easily solve this problem in $O(k(m + n) \log n)$ by running Dijkstra's algorithm k times, one for each G_i where G_i is the graph obtained by adding e_i to G .
2. Let G be an undirected graph with non-negative edge weights. Let s and t be two vertices such that the shortest path between s and t in G contains all the vertices in the graph. For each edge e , let $G \setminus e$ be the graph obtained from G by deleting the edge e . Design an $O(E \log V)$ algorithm that finds the shortest path distance between s and t in $G \setminus e$ for all e . [Note that you need to output E distances, one for each graph $G \setminus e$]
3. Given a Directed Acyclic Graph (DAG) and two vertices s and t you want to determine if there is an s to t path that includes at least k vertices.

CS 473: Undergraduate Algorithms, Spring 2009

HBS 7

1. Let $G = (V, E)$ be a directed graph with non-negative capacities. Give an efficient algorithm to check whether there is a unique max-flow on G ?
2. Let $G = (V, E)$ be a graph and $s, t \in V$ be two specific vertices of G . We call $(S, T = V \setminus S)$ an (s, t) -cut if $s \in S$ and $t \in T$. Moreover, it is a minimum cut if the sum of the capacities of the edges that have one endpoint in S and one endpoint in T equals the maximum (s, t) -flow. Show that, both intersection and union of two min-cuts is a min-cut itself.
3. Let $G = (V, E)$ be a graph. For each edge e let $d(e)$ be a demand value attached to it. A flow is feasible if it sends more than $d(e)$ through e . Assume you have an oracle that is capable of solving the maximum flow problem. Give efficient algorithms for the following problems that call the oracle at most once.
 - (a) Find a feasible flow.
 - (b) Find a feasible flow of minimum possible value.

CS 473: Undergraduate Algorithms, Spring 2009

HBS 8

1. A box i can be specified by the values of its sides, say (i_1, i_2, i_3) . We know all the side lengths are larger than 10 and smaller than 20 (i.e. $10 < i_1, i_2, i_3 < 20$). Geometrically, you know what it means for one box to nest in another: It is possible if you can rotate the smaller so that it fits inside the larger in each dimension. Of course, nesting is recursive, that is if i nests in j and j nests in k then i nests in k . After doing some nesting operations, we say a box is visible if it is not nested in any other one. Given a set of boxes (each specified by the lengths of their sides) the goal is to find a set of nesting operations to minimize the number of visible boxes. Design and analyze an efficient algorithm to do this.

2. Let the number of papers submitted to a conference be n and the number of available reviewers be m . Each reviewer has a list of papers that he/she can review and each paper should be reviewed by three different persons. Also, each reviewer can review at most 5 papers. Design and analyze an algorithm to make the assignment or decide no feasible assignment exists.

3. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like Yahoo! was in the "eyeballs" - the simple fact that it gets millions of people looking at its pages every day. And further, by convincing people to register personal data with the site, it can show each user an extremely targeted advertisement whenever he or she visits the site, in away that TV networks or magazines could not hope to match. So if the user has told Yahoo! that he is a 20-year old computer science major from Cornell University, the site can throw up a banner ad for apartments in Ithaca, NY; on the other hand, if he is a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead.

But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct demographic groups G_1, G_2, \dots, G_k . (These groups can overlap; for example G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different advertisers, to show a certain number of copies of their ads to users of the site. Here is what the contract with the i^{th} advertiser looks like:

- (a) For a subset $X_i \subset \{G_1, \dots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i
- (b) For a number r_i , advertiser i wants its ads shown to at least r_i users each minute.

Now, consider the problem of designing a good advertising policy - a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for $j = 1, 2, \dots, n$) belongs to a subset $U_j \subset \{G_1, \dots, G_k\}$ of the demographic groups. The problem is: is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \dots, m$, at least r_i of the n users, each belonging to at least one demographic group in X_i , are shown an ad provided by advertiser i .)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

CS 473: Undergraduate Algorithms, Spring 2009

HBS 9

1. Prove that any algorithm to merge two sorted arrays, each of size n , requires at least $2n - 1$ comparisons.
2. 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.]
3. Let P be a set of n points in the plane. The staircase of P is the set of all points in the plane that have at least one point in P both above and to the right. Prove that computing the staircase requires at least $\Omega(n \log n)$ comparisons in two ways,
 - (a) Reduction from sorting.
 - (b) Directly.

You have 90 minutes to answer four of the five questions.
Write your answers in the separate answer booklet.
 You may take the question sheet with you when you leave.

1. Each of these ten questions has one of the following five answers:

A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$

Choose the correct answer for each question. Each correct answer is worth +1 point; each incorrect answer is worth $-1/2$ point; each "I don't know" is worth $+1/4$ point. Your score will be rounded to the nearest *non-negative* integer.

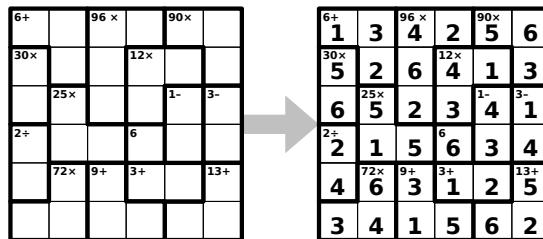
- (a) What is $\sum_{i=1}^n \frac{n}{i}$?
- (b) What is $\sqrt{\sum_{i=1}^n i}$?
- (c) How many digits are required to write 3^n in decimal?
- (d) What is the solution to the recurrence $D(n) = D(n/\pi) + \sqrt{2}$?
- (e) What is the solution to the recurrence $E(n) = E(n - \sqrt{2}) + \pi$?
- (f) What is the solution to the recurrence $F(n) = 4F(n/2) + 3n$?
- (g) What is the worst-case time to search for an item in a binary search tree?
- (h) What is the worst-case running time of quicksort?
- (i) Let $H[1..n, 1..n]$ be a fixed array of numbers. Consider the following recursive function:

$$Glub(i, j) = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } i > n \text{ or } j = 0 \\ \max \{Glub(i - 1, j), H[i, j] + Glub(i + 1, j - 1)\} & \text{otherwise} \end{cases}$$

How long does it take to compute $Glub(n, n)$ using dynamic programming?

- (j) What is the running time of the fastest possible algorithm to solve KenKen puzzles?

A KenKen puzzle is a 6×6 grid, divided into regions called *cages*. Each cage is labeled with a numerical *value* and an arithmetic *operation*: $+$, $-$, \times , or \div . (The operation can be omitted if the cage consists of a single cell.) The goal is to place an integer between 1 and 6 in each grid cell, so that no number appears twice in any row or column, and the numbers inside each cage can be combined using *only* that cage's operation to obtain that cage's value. The solution is guaranteed to be unique.

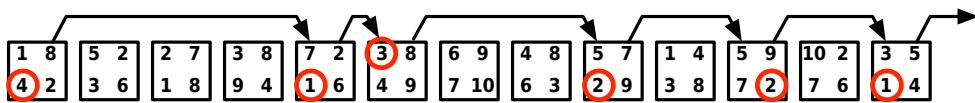


6+		96 ×		90 ×	
30x			12x		
	25x			1-	3-
2+			6		
	72x	9+	3+		13+

6+	1	3	4	2	5	6
30x	5	2	6	4	1	3
	6	5	2	3	4	1
2+	2	1	5	6	3	4
	4	6	3	1	2	5
	3	4	1	5	6	2

A Kenken puzzle and its solution

2. (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \dots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe an efficient algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. An algorithm that runs in $\Theta(n)$ time is worth at most 3 points.
- (b) Now suppose $A[1..n]$ is a sorted array of n distinct **positive** integers. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. [Hint: This is **really** easy!]
3. *Moby Selene* is a solitaire game played on a row of n squares. Each square contains four positive integers. The player begins by placing a token on the leftmost square. On each move, the player chooses one of the numbers on the token's current square, and then moves the token that number of squares to the right. The game ends when the token moves past the rightmost square. The object of the game is to make as many moves as possible before the game ends.



A Moby Selene puzzle that allows six moves. (This is **not** the longest legal sequence of moves.)

- (a) **Prove** that the obvious greedy strategy (always choose the smallest number) does not give the largest possible number of moves for every Moby Selene puzzle.
- (b) Describe and analyze an efficient algorithm to find the largest possible number of legal moves for a given Moby Selene puzzle.
4. Consider the following algorithm for finding the largest element in an unsorted array:

```
RANDOMMAX( $A[1..n]$ ):
   $max \leftarrow \infty$ 
  for  $i \leftarrow 1$  to  $n$  in random order
    if  $A[i] > max$ 
       $max \leftarrow A[i]$  (*)
```

return max

- (a) In the worst case, how many times does RANDOMMAX execute line (*)?
- (b) What is the **exact** probability that line (*) is executed during the last iteration of the for loop?
- (c) What is the **exact** expected number of executions of line (*)? (A correct $\Theta()$ bound is worth half credit.)
5. *This question is taken directly from HBS 0.* Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. Pretty please.

You have 90 minutes to answer four of the five questions.
Write your answers in the separate answer booklet.
You may take the question sheet with you when you leave.

1. Recall that a *tree* is a connected graph with no cycles. A graph is *bipartite* if we can color its vertices black and white, so that every edge connects a white vertex to a black vertex.
 - (a) **Prove** that every tree is bipartite.
 - (b) Describe and analyze a fast algorithm to determine whether a given graph is bipartite.
2. Describe and analyze an algorithm $\text{SHUFFLE}(A[1..n])$ that randomly permutes the input array A , so that each of the $n!$ possible permutations is equally likely. You can assume the existence of a subroutine $\text{RANDOM}(k)$ that returns a random integer chosen uniformly between 1 and k in $O(1)$ time. For full credit, your SHUFFLE algorithm should run in $O(n)$ time. [Hint: This problem appeared in HBS 3½.]
3. Let G be an undirected graph with weighted edges.
 - (a) Describe and analyze an algorithm to compute the *maximum weight spanning tree* of G .
 - (b) A *feedback edge set* of G is a subset F of the edges such that every cycle in G contains at least one edge in F . In other words, removing every edge in F makes G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of G .

[Hint: Don't reinvent the wheel!]
4. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G , and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H , so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm to choose the best edge to reinsert. For full credit, your algorithm should run in $O(E \log V)$ time. [Hint: This problem appeared in HBS 6¾.]
5. Describe and analyze an efficient data structure to support the following operations on an array $X[1..n]$ as quickly as possible. Initially, $X[i] = 0$ for all i .
 - Given an index i such that $X[i] = 0$, set $X[i]$ to 1.
 - Given an index i , return $X[i]$.
 - Given an index i , return the smallest index $j \geq i$ such that $X[j] = 0$, or report that no such index exists.

For full credit, the first two operations should run in *worst-case constant* time, and the amortized cost of the third operation should be as small as possible.

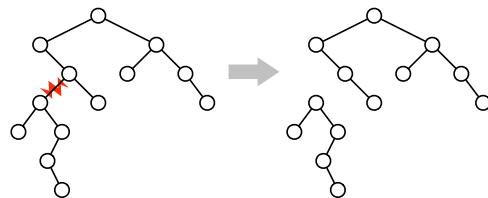
You have 180 minutes to answer six of the seven questions.
 Write your answers in the separate answer booklet.
 You may take the question sheet with you when you leave.

1. SUBSETSUM and PARTITION are two closely related NP-hard problems, defined as follows.

SUBSETSUM: Given a set X of positive integers and a positive integer k , does X have a subset whose elements sum up to k ?

PARTITION: Given a set Y of positive integers, can Y be partitioned into two subsets whose sums are equal?

- (a) [2 pts] **Prove** that PARTITION and SUBSETSUM are both in NP.
 - (b) [1 pt] Suppose you already know that SUBSETSUM is NP-hard. Which of the following arguments could you use to prove that PARTITION is NP-hard? **You do not need to justify your answer** — just answer ① or ②.
 - ① Given a set X and an integer k , construct a set Y in polynomial time, such that $\text{PARTITION}(Y)$ is true if and only if $\text{SUBSETSUM}(X, k)$ is true.
 - ② Given a set Y , construct a set X and an integer k in polynomial time, such that $\text{PARTITION}(Y)$ is true if and only if $\text{SUBSETSUM}(X, k)$ is true.
 - (c) [3 pts] Describe and analyze a polynomial-time reduction from PARTITION to SUBSETSUM. **You do not need to prove that your reduction is correct.**
 - (d) [4 pts] Describe and analyze a polynomial-time reduction from SUBSETSUM to PARTITION. **You do not need to prove that your reduction is correct.**
2. (a) [4 pts] For any node v in a binary tree, let $\text{size}(v)$ denote the number of nodes in the subtree rooted at v . Let k be an arbitrary positive number. **Prove** that every binary tree with at least k nodes contains a node v such that $k \leq \text{size}(v) \leq 2k$.
- (b) [2 pts] Removing any edge from an n -node binary tree T separates it into two smaller binary trees. An edge is called a **balanced separator** if both of these subtrees have at least $n/3$ nodes (and therefore at most $2n/3$ nodes). **Prove** that every binary tree with more than one node has a balanced separator. [Hint: Use part (a).]
- (c) [4 pts] Describe and analyze an algorithm to find a balanced separator in a given binary tree. [Hint: Use part (a).]



Removing a balanced separator from a binary tree.

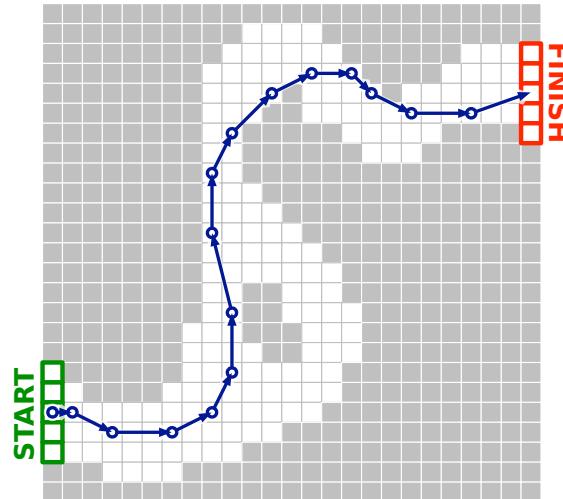
3. **Racetrack** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.¹ The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

Each car has a *position* and a *velocity*, both with integer *x*- and *y*-coordinates. The initial position is a point on the starting line, chosen by the player; the initial velocity is always $(0, 0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position on the finish line.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting line' is the first column, and the 'finish line' is the last column.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. [Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]

velocity	position
$(0, 0)$	$(1, 5)$
$(1, 0)$	$(2, 5)$
$(2, -1)$	$(4, 4)$
$(3, 0)$	$(7, 4)$
$(2, 1)$	$(9, 5)$
$(1, 2)$	$(10, 7)$
$(0, 3)$	$(10, 10)$
$(-1, 4)$	$(9, 14)$
$(0, 3)$	$(9, 17)$
$(1, 2)$	$(10, 19)$
$(2, 2)$	$(12, 21)$
$(2, 1)$	$(14, 22)$
$(2, 0)$	$(16, 22)$
$(1, -1)$	$(17, 21)$
$(2, -1)$	$(19, 20)$
$(3, 0)$	$(22, 20)$
$(3, 1)$	$(25, 21)$



A 16-step Racetrack run, on a 25×25 track.

4. A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA. Describe and analyze an algorithm to find the length of the longest *subsequence* of a given string that is also a palindrome.

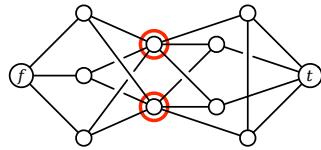
For example, the longest palindrome subsequence of MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11.

¹The actual game is a bit more complicated than the version described here.

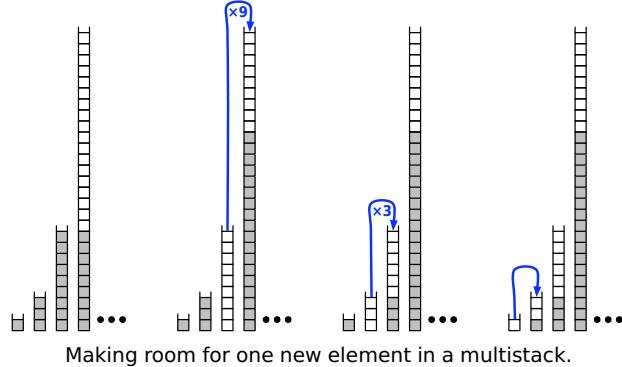
5. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the number 2.



6. A *mystack* consists of an infinite series of stacks S_0, S_1, S_2, \dots , where the i th stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i , we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2} .) Moving a single element from one stack to the next takes $O(1)$ time.



- (a) In the worst case, how long does it take to push one more element onto a mystack containing n elements?
- (b) **Prove** that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the mystack.
7. Recall the problem 3COLOR: Given a graph, can we color each vertex with one of 3 colors, so that every edge touches two different colors? We proved in class that 3COLOR is NP-hard.

Now consider the related problem 12COLOR: Given a graph, can we color each vertex with one of twelve colors, so that every edge touches two different colors? **Prove** that 12COLOR is NP-hard.

You may assume the following problems are NP-hard:

CIRCUITSAT: Given a boolean circuit, are there any input values that make the circuit output True?

PLANAR CIRCUITSAT: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3SAT: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

MAXINDEPENDENTSET: Given an undirected graph G , what is the size of the largest subset of vertices in G that have no edges among them?

MAXCLIQUE: Given an undirected graph G , what is the size of the largest complete subgraph of G ?

MINVERTEXCOVER: Given an undirected graph G , what is the size of the smallest subset of vertices that touch every edge in G ?

MINSETCOVER: Given a collection of subsets S_1, S_2, \dots, S_m of a set S , what is the size of the smallest subcollection whose union is S ?

MINHITTINGSET: Given a collection of subsets S_1, S_2, \dots, S_m of a set S , what is the size of the smallest subset of S that intersects every subset S_i ?

3COLOR: Given an undirected graph G , can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HAMILTONIANCYCLE: Given a graph G , can is there a cycle in G that visits every vertex once?

HAMILTONIANPATH: Given a graph G , can is there a path in G that visits every vertex once?

DOUBLEHAMILTONIANCYCLE: Given a graph G , can is there a closed walk in G that visits every vertex twice?

DOUBLEHAMILTONIANPATH: Given a graph G , can is there an open walk in G that visits every vertex twice?

MINDEGREESPANNINGTREE: Given an undirected graph G , what is the minimum degree of any spanning tree of G ?

MINLEAVESPANNINGTREE: Given an undirected graph G , what is the minimum number of leaves in any spanning tree of G ?

TRAVELINGSalesman: Given a graph G with weighted edges, what is the minimum cost of any Hamiltonian path/cycle in G ?

LONGESTPATH: Given a graph G with weighted edges and two vertices s and t , what is the length of the longest *simple* path from s to t in G ?

SUBSETSUM: Given a set X of positive integers and an integer k , does X have a subset whose elements sum to k ?

PARTITION: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3PARTITION: Given a set X of n positive integers, can X be partitioned into $n/3$ three-element subsets, all with the same sum?

MINESWEEPER: Given a Minesweeper configuration and a particular square x , is it safe to click on x ?

TETRIS: Given a sequence of N Tetris pieces and a partially filled $n \times k$ board, is it possible to play every piece in the sequence without overflowing the board?

SUDOKU: Given an $n \times n$ Sudoku puzzle, does it have a solution?

KENKEN: Given an $n \times n$ Ken-Ken puzzle, does it have a solution?