Instructor's Manual

by Thomas H. Cormen
Clara Lee
Erica Lin

to Accompany

Introduction to Algorithms

Second Edition

by Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

The MIT Press
Cambridge, Massachusetts London, England

McGraw-Hill Book Company
Boston Burr Ridge, IL Dubuque, IA Madison, WI
New York San Francisco St. Louis Montréal Toronto

Instructor's Manual
by Thomas H. Cormen, Clara Lee, and Erica Lin
to Accompany
Introduction to Algorithms, Second Edition
by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Published by The MIT Press and McGraw-Hill Higher Education, an imprint of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020.

Contents

Revision History

Revisions are listed by date rather than being numbered. Because this revision history is part of each revision, the affected chapters always include the front matter in addition to those listed below.

- 18 January 2005. Corrected an error in the transpose-symmetry properties. Affected chapters: Chapter 3.
- 2 April 2004. Added solutions to Exercises 5.4-6, 11.3-5, 12.4-1, 16.4-2, 16.4-3, 21.3-4, 26.4-2, 26.4-3, and 26.4-6 and to Problems 12-3 and 17-4. Made minor changes in the solutions to Problems 11-2 and 17-2. Affected chapters: Chapters 5, 11, 12, 16, 17, 21, and 26; index.
- 7 January 2004. Corrected two minor typographical errors in the lecture notes for the expected height of a randomly built binary search tree. Affected chapters: Chapter 12.
- 23 July 2003. Updated the solution to Exercise 22.3-4(b) to adjust for a correction in the text. Affected chapters: Chapter 22; index.
- 23 June 2003. Added the link to the website for the `clrscode` package to the preface.
- 2 June 2003. Added the solution to Problem 24-6. Corrected solutions to Exercise 23.2-7 and Problem 26-4. Affected chapters: Chapters 23, 24, and 26; index.
- 20 May 2003. Added solutions to Exercises 24.4-10 and 26.1-7. Affected chapters: Chapters 24 and 26; index.
- 2 May 2003. Added solutions to Exercises 21.4-4, 21.4-5, 21.4-6, 22.1-6, and 22.3-4. Corrected a minor typographical error in the Chapter 22 notes on page 22-6. Affected chapters: Chapters 21 and 22; index.
- 28 April 2003. Added the solution to Exercise 16.1-2, corrected an error in the first adjacency matrix example in the Chapter 22 notes, and made a minor change to the accounting method analysis for dynamic tables in the Chapter 17 notes. Affected chapters: Chapters 16, 17, and 22; index.
- 10 April 2003. Corrected an error in the solution to Exercise 11.3-3. Affected chapters: Chapter 11.
- 3 April 2003. Reversed the order of Exercises 14.2-3 and 14.3-3. Affected chapters: Chapter 13, index.
- 2 April 2003. Corrected an error in the substitution method for recurrences on page 4-4. Affected chapters: Chapter 4.

- 31 March 2003. Corrected a minor typographical error in the Chapter 8 notes on page 8-3. Affected chapters: Chapter 8.
- 14 January 2003. Changed the exposition of indicator random variables in the Chapter 5 notes to correct for an error in the text. Affected pages: 5-4 through 5-6. (The only content changes are on page 5-4; in pages 5-5 and 5-6 only pagination changes.) Affected chapters: Chapter 5.
- 14 January 2003. Corrected an error in the pseudocode for the solution to Exercise 2.2-2 on page 2-16. Affected chapters: Chapter 2.
- 7 October 2002. Corrected a typographical error in EUCLIDEAN-TSP on page 15-23. Affected chapters: Chapter 15.
- 1 August 2002. Initial release.

Preface

This document is an instructor's manual to accompany *Introduction to Algorithms*, Second Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. It is intended for use in a course on algorithms. You might also find some of the material herein to be useful for a CS 2-style course in data structures.

Unlike the instructor's manual for the first edition of the text—which was organized around the undergraduate algorithms course taught by Charles Leiserson at MIT in Spring 1991—we have chosen to organize the manual for the second edition according to chapters of the text. That is, for most chapters we have provided a set of lecture notes and a set of exercise and problem solutions pertaining to the chapter. This organization allows you to decide how to best use the material in the manual in your own course.

We have not included lecture notes and solutions for every chapter, nor have we included solutions for every exercise and problem within the chapters that we have selected. We felt that Chapter 1 is too nontechnical to include here, and Chapter 10 consists of background material that often falls outside algorithms and data-structures courses. We have also omitted the chapters that are not covered in the courses that we teach: Chapters 18–20 and 28–35, as well as Appendices A–C; future editions of this manual may include some of these chapters. There are two reasons that we have not included solutions to all exercises and problems in the selected chapters. First, writing up all these solutions would take a long time, and we felt it more important to release this manual in as timely a fashion as possible. Second, if we were to include all solutions, this manual would be longer than the text itself!

We have numbered the pages in this manual using the format *CC-PP*, where *CC* is a chapter number of the text and *PP* is the page number within that chapter's lecture notes and solutions. The *PP* numbers restart from 1 at the beginning of each chapter's lecture notes. We chose this form of page numbering so that if we add or change solutions to exercises and problems, the only pages whose numbering is affected are those for the solutions for that chapter. Moreover, if we add material for currently uncovered chapters, the numbers of the existing pages will remain unchanged.

The lecture notes

The lecture notes are based on three sources:

- Some are from the first-edition manual, and so they correspond to Charles Leiserson's lectures in MIT's undergraduate algorithms course, 6.046.
- Some are from Tom Cormen's lectures in Dartmouth College's undergraduate algorithms course, CS 25.
- Some are written just for this manual.

You will find that the lecture notes are more informal than the text, as is appropriate for a lecture situation. In some places, we have simplified the material for lecture presentation or even omitted certain considerations. Some sections of the text—usually starred—are omitted from the lecture notes. (We have included lecture notes for one starred section: 12.4, on randomly built binary search trees, which we cover in an optional CS 25 lecture.)

In several places in the lecture notes, we have included "asides" to the instructor. The asides are typeset in a slanted font and are enclosed in square brackets. *[Here is an aside.]* Some of the asides suggest leaving certain material on the board, since you will be coming back to it later. If you are projecting a presentation rather than writing on a blackboard or whiteboard, you might want to mark slides containing this material so that you can easily come back to them later in the lecture.

We have chosen not to indicate how long it takes to cover material, as the time necessary to cover a topic depends on the instructor, the students, the class schedule, and other variables.

There are two differences in how we write pseudocode in the lecture notes and the text:

- Lines are not numbered in the lecture notes. We find them inconvenient to number when writing pseudocode on the board.
- We avoid using the *length* attribute of an array. Instead, we pass the array length as a parameter to the procedure. This change makes the pseudocode more concise, as well as matching better with the description of what it does.

We have also minimized the use of shading in figures within lecture notes, since drawing a figure with shading on a blackboard or whiteboard is difficult.

The solutions

The solutions are based on the same sources as the lecture notes. They are written a bit more formally than the lecture notes, though a bit less formally than the text. We do not number lines of pseudocode, but we do use the *length* attribute (on the assumption that you will want your students to write pseudocode as it appears in the text).

The index lists all the exercises and problems for which this manual provides solutions, along with the number of the page on which each solution starts.

Asides appear in a handful of places throughout the solutions. Also, we are less reluctant to use shading in figures within solutions, since these figures are more likely to be reproduced than to be drawn on a board.

Source files

For several reasons, we are unable to publish or transmit source files for this manual. We apologize for this inconvenience.

In June 2003, we made available a `clrscode` package for LaTeX 2_ε. It enables you to typeset pseudocode in the same way that we do. You can find this package at `http://www.cs.dartmouth.edu/~thc/clrscode/`. That site also includes documentation.

Reporting errors and suggestions

Undoubtedly, instructors will find errors in this manual. Please report errors by sending email to `clrs-manual-bugs@mhhe.com`

If you have a suggestion for an improvement to this manual, please feel free to submit it via email to `clrs-manual-suggestions@mhhe.com`

As usual, if you find an error in the text itself, please verify that it has not already been posted on the errata web page before you submit it. You can use the MIT Press web site for the text, `http://mitpress.mit.edu/algorithms/`, to locate the errata web page and to submit an error report.

We thank you in advance for your assistance in correcting errors in both this manual and the text.

Acknowledgments

This manual borrows heavily from the first-edition manual, which was written by Julie Sussman, P.P.A. Julie did such a superb job on the first-edition manual, finding numerous errors in the first-edition text in the process, that we were thrilled to have her serve as technical copyeditor for the second-edition text. Charles Leiserson also put in large amounts of time working with Julie on the first-edition manual.

The other three *Introduction to Algorithms* authors—Charles Leiserson, Ron Rivest, and Cliff Stein—provided helpful comments and suggestions for solutions to exercises and problems. Some of the solutions are modifications of those written over the years by teaching assistants for algorithms courses at MIT and Dartmouth. At this point, we do not know which TAs wrote which solutions, and so we simply thank them collectively.

We also thank McGraw-Hill and our editors, Betsy Jones and Melinda Dougharty, for moral and financial support. Thanks also to our MIT Press editor, Bob Prior, and to David Jones of The MIT Press for help with TeX macros. Wayne Cripps, John Konkle, and Tim Tregubov provided computer support at Dartmouth, and the MIT sysadmins were Greg Shomo and Matt McKinnon. Phillip Meek of McGraw-Hill helped us hook this manual into their web site.

THOMAS H. CORMEN
CLARA LEE
ERICA LIN
Hanover, New Hampshire
July 2002

Lecture Notes for Chapter 2: Getting Started

Chapter 2 overview

Goals:

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- See how to describe algorithms in pseudocode.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of "divide and conquer" in the context of merge sort.

Insertion sort

The sorting problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The sequences are typically stored in arrays.

We also refer to the numbers as ***keys***. Along with each key may be additional information, known as ***satellite data***. *[You might want to clarify that "satellite data" does not necessarily come from a satellite!]*

We will see several ways to solve the sorting problem. Each way will be expressed as an ***algorithm***: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Expressing algorithms

We express algorithms in whatever way is the clearest and most concise.

English is sometimes the best way.

When issues of control need to be made perfectly clear, we often use ***pseudocode***.

- Pseudocode is similar to C, C++, Pascal, and Java. If you know any of these languages, you should be able to understand pseudocode.
- Pseudocode is designed for *expressing algorithms to humans*. Software engineering issues of data abstraction, modularity, and error handling are often ignored.
- We sometimes embed English statements into pseudocode. Therefore, unlike for "real" programming languages, we cannot create a compiler that translates pseudocode to machine code.

Insertion sort

A good algorithm for sorting a small number of elements.

It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

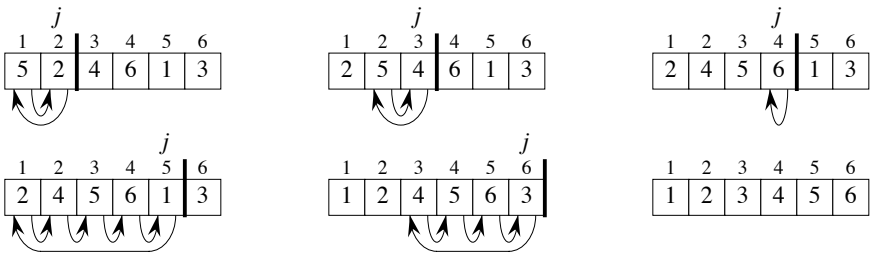
Pseudocode: We use a procedure INSERTION-SORT.

- Takes as parameters an array $A[1 \,..\, n]$ and the length n of the array.
- As in Pascal, we use ". ." to denote a range within an array.
- *[We usually use 1-origin indexing, as we do here. There are a few places in later chapters where we use 0-origin indexing instead. If you are translating pseudocode to C, C++, or Java, which use 0-origin indexing, you need to be careful to get the indices right. One option is to adjust all index calculations in the C, C++, or Java code to compensate. An easier option is, when using an array $A[1 \,..\, n]$, to allocate the array to be one entry longer—$A[0 \,..\, n]$—and just don't use the entry at index 0.]*
- *[In the lecture notes, we indicate array lengths by parameters rather than by using the length attribute that is used in the book. That saves us a line of pseudocode each time. The solutions continue to use the length attribute.]*
- The array A is sorted ***in place***: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

[Leave this on the board, but show only the pseudocode for now. We'll put in the "cost" and "times" columns later.]

Example:



[Read this figure row by row. Each part shows what happens for a particular iteration with the value of j indicated. j indexes the "current card" being inserted into the hand. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position. The heavy vertical lines separate the part of the array in which an iteration works—$A[1\,..\,j]$—from the part of the array that is unaffected by this iteration—$A[j+1\,..\,n]$. The last part of the figure shows the final sorted array.]

Correctness

We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer. Here's the loop invariant for INSERTION-SORT:

> **Loop invariant:** At the start of each iteration of the "outer" **for** loop—the loop indexed by j—the subarray $A[1\,..\,j-1]$ consists of the elements originally in $A[1\,..\,j-1]$ but in sorted order.

To use a loop invariant to prove correctness, we must show three things about it:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like mathematical induction:

- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.
- We can show the three parts in any order.

For insertion sort:

Initialization: Just before the first iteration, $j = 2$. The subarray $A[1 \,..\, j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.

Termination: The outer **for** loop ends when $j > n$; this occurs when $j = n+1$. Therefore, $j-1 = n$. Plugging n in for $j-1$ in the loop invariant, the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$ but in sorted order. In other words, the entire array is sorted!

Pseudocode conventions

[Covering most, but not all, here. See book pages 19–20 for all conventions.]

- Indentation indicates block structure. Saves space and writing time.
- Looping constructs are like in C, C++, Pascal, and Java. We assume that the loop variable in a **for** loop is still defined when the loop exits (unlike in Pascal).
- "$\triangleright$" indicates that the remainder of the line is a comment.
- Variables are local, unless otherwise specified.
- We often use ***objects***, which have ***attributes*** (equivalently, ***fields***). For an attribute *attr* of object x, we write $attr[x]$. (This would be the equivalent of $x.attr$ in Java or $x\text{->}attr$ in C++.)
- Objects are treated as references, like in Java. If x and y denote objects, then the assignment $y \leftarrow x$ makes x and y reference the same object. It does not cause attributes of one object to be copied to another.
- Parameters are passed by value, as in Java and C (and the default mechanism in Pascal and C++). When an object is passed by value, it is actually a reference (or pointer) that is passed; changes to the reference itself are not seen by the caller, but changes to the object's attributes are.
- The boolean operators "and" and "or" are ***short-circuiting***: if after evaluating the left-hand operand, we know the result of the expression, then we don't evaluate the right-hand operand. (If x is FALSE in "x and y" then we don't evaluate y. If x is TRUE in "x or y" then we don't evaluate y.)

Analyzing algorithms

We want to predict the resources that the algorithm requires. Usually, running time.
In order to predict resource requirements, we need a computational model.

Random-access machine (RAM) model

- Instructions are executed one after another. No concurrent operations.
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll use instructions commonly found in real computers:
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling). Also, shift left/shift right (good for multiplying/dividing by 2^k).
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return.

  Each of these instructions takes a constant amount of time.

The RAM model uses integer and floating-point types.

- We don't worry about precision, although it is crucial in certain numerical applications.
- There is a limit on the word size: when working with inputs of size n, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. ($\lg n$ is a very frequently used shorthand for $\log_2 n$.)
  - $c \geq 1 \Rightarrow$ we can hold the value of $n \Rightarrow$ we can index the individual elements.
  - c is a constant $\Rightarrow$ the word size cannot grow arbitrarily.

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

Input size: Depends on the problem being studied.

- Usually, the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Running time: On a particular input, it is the number of primitive operations (steps) executed.

- Want to define steps to be machine-independent.
- Figure that each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line i takes the same amount of time c_i.
- This is assuming that the line consists only of primitive operations.
  - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
  - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: "sort the points by x-coordinate."

Analysis of insertion sort

[Now add statement costs and number of times executed to INSERTION-SORT *pseudocode.]*

- Assume that the ith line takes time c_i, which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, \ldots, n$, let t_j be the number of times that the **while** loop test is executed for that value of j.
- Note that when a **for** or **while** loop exits in the usual way—due to the test in the loop header—the test is executed one time more than the loop body.

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) \ .$$

Let $T(n) =$ running time of INSERTION-SORT.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \ . \end{aligned}$$

The running time depends on the values of t_j. These vary according to the input.

Best case: The array is already sorted.

- Always find that $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j are 1.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow T(n)$ is a *linear function* of n.

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\displaystyle\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\displaystyle\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\displaystyle\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\dfrac{n(n+1)}{2}$.
- Since $\displaystyle\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\dfrac{n(n+1)}{2} - 1$.
  [The parentheses around the summation are not strictly necessary. They are there for clarity, but it might be a good idea to remind the students that the meaning of the expression would be the same even without the parentheses.]
- Letting $k = j - 1$, we see that $\displaystyle\sum_{j=2}^{n} (j - 1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Worst-case and average-case analysis

We usually concentrate on finding the ***worst-case running time***: the longest running time for *any* input of size n.

Reasons:

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it's often about as bad as the worst case.

Example: Suppose that we randomly choose n numbers as the input to insertion sort.

On average, the key in $A[j]$ is less than half the elements in $A[1 \,..\, j-1]$ and it's greater than the other half.
$\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1 \,..\, j-1]$ to decide where to drop *key*.
$\Rightarrow t_j = j/2$.

Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.

Order of growth

Another abstraction to ease analysis and focus on the important features.

Look only at the leading term of the formula for running time.

- Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

Example: For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$.
Drop lower-order terms $\Rightarrow an^2$.
Ignore constant coefficient $\Rightarrow n^2$.

But we cannot say that the worst-case running time $T(n)$ *equals* n^2.

It *grows like* n^2. But it doesn't *equal* n^2.

We say that the running time is $\Theta(n^2)$ to capture the notion that the *order of growth* is n^2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Designing algorithms

There are many ways to design algorithms.

For example, insertion sort is ***incremental***: having sorted $A[1 \,..\, j-1]$, place $A[j]$ correctly, so that $A[1 \,..\, j]$ is sorted.

Divide and conquer

Another common approach.

Divide the problem into a number of subproblems.

Conquer the subproblems by solving them recursively.

Base case: If the subproblems are small enough, just solve them by brute force.

[It would be a good idea to make sure that your students are comfortable with recursion. If they are not, then they will have a hard time understanding divide and conquer.]

Combine the subproblem solutions to give a solution to the original problem.

Merge sort

A sorting algorithm based on divide and conquer. Its worst-case running time has a lower order of growth than insertion sort.

Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p\,..\,r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p\,..\,r]$:

Divide by splitting into two subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$, where q is the halfway point of $A[p\,..\,r]$.

Conquer by recursively sorting the two subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$.

Combine by merging the two sorted subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ to produce a single sorted subarray $A[p\,..\,r]$. To accomplish this step, we'll define a procedure MERGE(A, p, q, r).

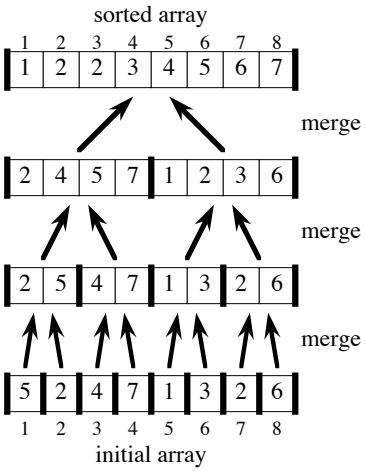
The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

```
MERGE-SORT(A, p, r)
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```

Initial call: MERGE-SORT$(A, 1, n)$

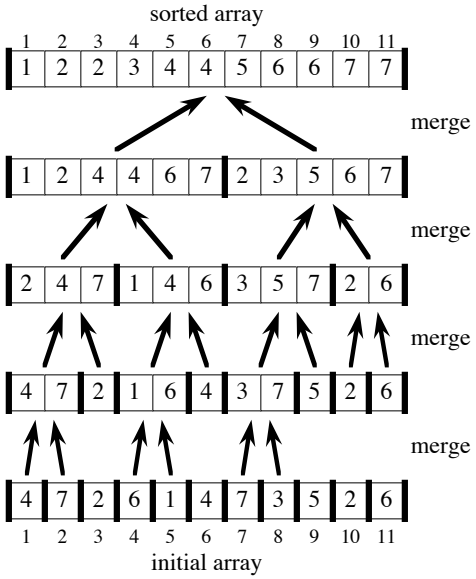
[It is astounding how often students forget how easy it is to compute the halfway point of p and r as their average $(p + r)/2$. We of course have to take the floor to ensure that we get an integer index q. But it is common to see students perform calculations like $p + (r - p)/2$, or even more elaborate expressions, forgetting the easy way to compute an average.]

Example: Bottom-up view for $n = 8$: *[Heavy lines demarcate subarrays used in subproblems.]*



[Examples when n is a power of 2 are most straightforward, but students might also want an example when n is not a power of 2.]

Bottom-up view for $n = 11$:



[Here, at the next-to-last level of recursion, some of the subproblems have only 1 element. The recursion bottoms out on these single-element subproblems.]

Merging

What remains is the MERGE procedure.

Input: Array A and indices p, q, r such that

- $p \le q < r$.
- Subarray $A[p\,..\,q]$ is sorted and subarray $A[q+1\,..\,r]$ is sorted. By the restrictions on p, q, r, neither subarray is empty.

Output: The two subarrays are merged into a single sorted subarray in $A[p\,..\,r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1 =$ the number of elements being merged.

What is n? Until now, n has stood for the size of the original problem. But now we're using it as the size of a subproblem. We will use this technique when we analyze recursive algorithms. Although we may denote the original problem size by n, in general n will be the size of a given subproblem.

Idea behind linear-time merging: Think of two piles of cards.

- Each pile is sorted and placed face-up on a table with the smallest cards on top.
- We will merge these into a single sorted pile, face-down on the table.
- A basic step:
  - Choose the smaller of the two top cards.

  - Remove it from its pile, thereby exposing a new top card.
  - Place the chosen card face-down onto the output pile.
- Repeatedly perform basic steps until one input pile is empty.
- Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.
- Each basic step should take constant time, since we check just the two top cards.
- There are $\leq n$ basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles.
- Therefore, this procedure should take $\Theta(n)$ time.

We don't actually need to check whether a pile is empty before each basic step.

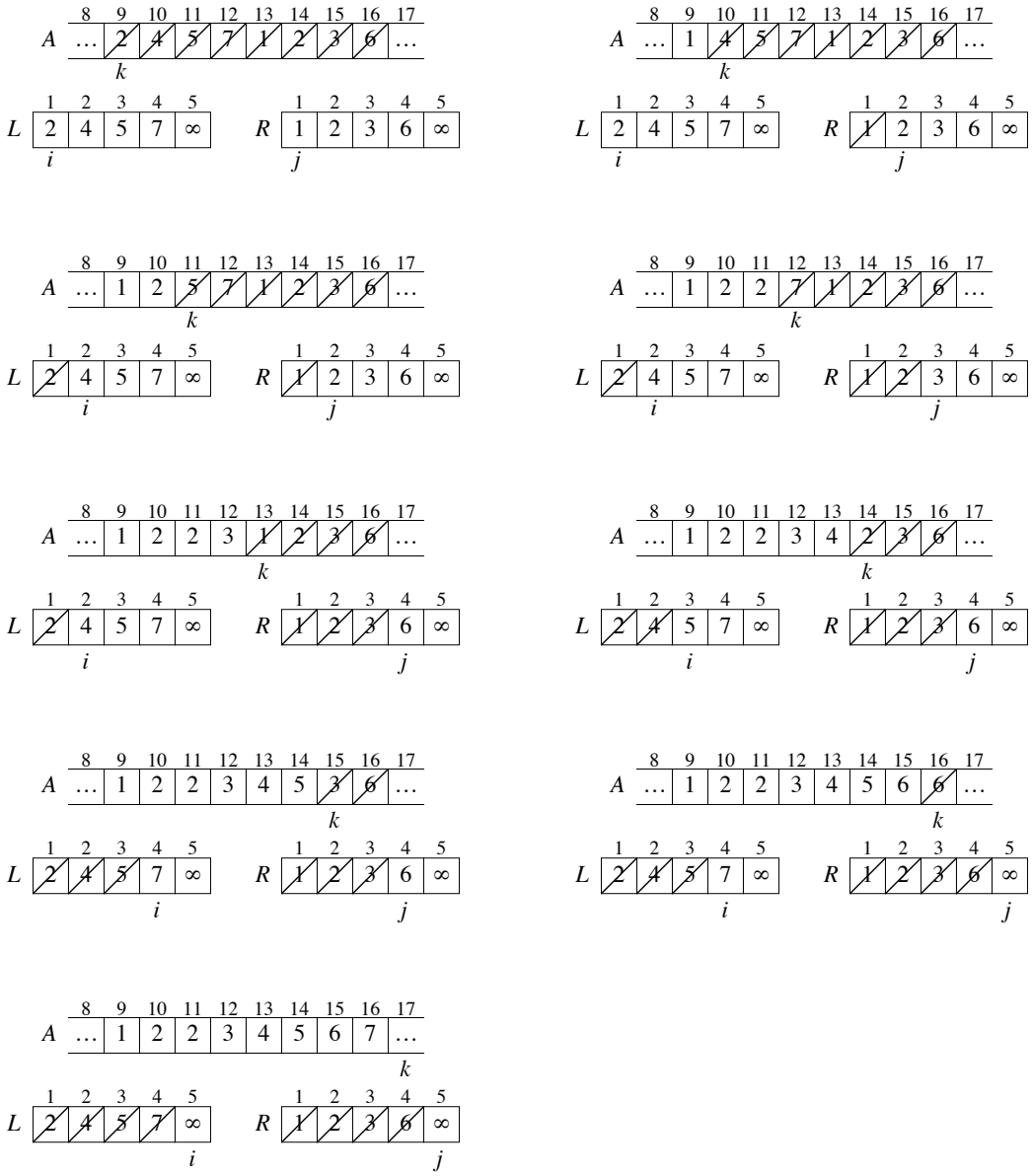
- Put on the bottom of each input pile a special ***sentinel*** card.
- It contains a special value that we use to simplify the code.
- We use ∞, since that's guaranteed to "lose" to any other value.
- The only way that ∞ *cannot* lose is when both piles have ∞ exposed as their top cards.
- But when that happens, all the nonsentinel cards have already been placed into the output pile.
- We know in advance that there are exactly $r - p + 1$ nonsentinel cards $\Rightarrow$ stop once we have performed $r - p + 1$ basic steps. Never a need to check for sentinels, since they'll always lose.
- Rather than even counting basic steps, just fill up the output array from index p up through and including index r.

Pseudocode:

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```

[The book uses a loop invariant to establish that MERGE *works correctly. In a lecture situation, it is probably better to use an example to show that the procedure works correctly.]*

Example: A call of MERGE(9, 12, 16)



*[Read this figure row by row. The first part shows the arrays at the start of the "**for** $k \leftarrow p$ **to** r" loop, where $A[p\,..\,q]$ is copied into $L[1\,..\,n_1]$ and $A[q+1\,..\,r]$ is copied into $R[1\,..\,n_2]$. Succeeding parts show the situation at the start of successive iterations. Entries in A with slashes have had their values copied to either L or R and have not had a value copied back in yet. Entries in L and R with slashes have been copied back into A. The last part shows that the subarrays are merged back into $A[p\,..\,r]$, which is now sorted, and that only the sentinels (∞) are exposed in the arrays L and R.]*

Running time: The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time. The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
Total time: $\Theta(n)$.

Analyzing divide-and-conquer algorithms

Use a ***recurrence equation*** (more commonly, a ***recurrence***) to describe the running time of a divide-and-conquer algorithm.

Let $T(n) =$ running time on a problem of size n.

- If the problem size is small enough (say, $n \le c$ for some constant c), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let the time to divide a size-n problem be $D(n)$.
- There are a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\ . \end{cases}$$

Analyzing merge sort

For simplicity, assume that n is a power of $2 \Rightarrow$ each divide step yields two subproblems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \ge 2$, time for merge sort steps:

Divide: Just compute q as the average of p and $r \Rightarrow D(n) = \Theta(1)$.

Conquer: Recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$.

Combine: MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in n: $\Theta(n) \Rightarrow$ recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\ , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\ . \end{cases}$$

Solving the merge-sort recurrence: By the master theorem in Chapter 4, we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$. *[Reminder:* $\lg n$ *stands for* $\log_2 n$*.]*

Compared to insertion sort ($\Theta(n^2)$ worst-case time), merge sort is faster. Trading a factor of n for a factor of $\lg n$ is a good deal.

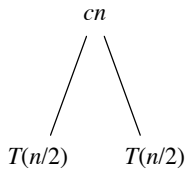
On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

We can understand how to solve the merge-sort recurrence without the master theorem.

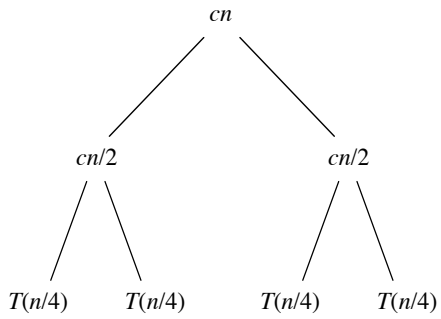
- Let c be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps. *[Of course, we cannot necessarily use the same constant for both. It's not worth going into this detail at this point.]*
- We rewrite the recurrence as

$$T(n) = \begin{cases} c & \text{if } n = 1 \text{ ,} \\ 2T(n/2) + cn & \text{if } n > 1 \text{ .} \end{cases}$$

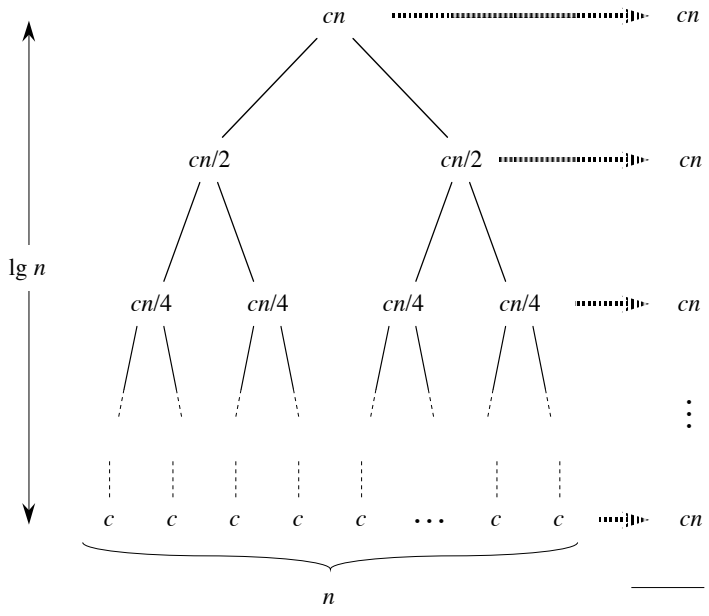
- Draw a ***recursion tree***, which shows successive expansions of the recurrence.
- For the original problem, we have a cost of cn, plus the two subproblems, each costing $T(n/2)$:



- For each of the size-$n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing $T(n/4)$:



- Continue expanding until the problem sizes get down to 1:



Total: $cn \lg n + cn$

- Each level has cost cn.
  - The top level has cost cn.
  - The next level down has 2 subproblems, each contributing cost $cn/2$.
  - The next level has 4 subproblems, each contributing cost $cn/4$.
  - Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves $\Rightarrow$ cost per level stays the same.
- There are $\lg n + 1$ levels (height is $\lg n$).
  - Use induction.
  - Base case: $n = 1 \Rightarrow 1$ level, and $\lg 1 + 1 = 0 + 1 = 1$.
  - Inductive hypothesis is that a tree for a problem size of 2^i has $\lg 2^i + 1 = i + 1$ levels.
  - Because we assume that the problem size is a power of 2, the next problem size up after 2^i is 2^{i+1}.
  - A tree for a problem size of 2^{i+1} has one more level than the size-2^i tree $\Rightarrow$ $i + 2$ levels.
  - Since $\lg 2^{i+1} + 1 = i + 2$, we're done with the inductive argument.
- Total cost is sum of costs at each level. Have $\lg n + 1$ levels, each costing $cn \Rightarrow$ total cost is $cn \lg n + cn$.
- Ignore low-order term of cn and constant coefficient $c \Rightarrow \Theta(n \lg n)$.

Solutions for Chapter 2: Getting Started

Solution to Exercise 2.2-2

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                 then smallest ← i
       exchange A[j] ↔ A[smallest]
```

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1\,..\,j-1]$ consists of the $j-1$ smallest elements in the array $A[1\,..\,n]$, and this subarray is in sorted order. After the first $n-1$ elements, the subarray $A[1\,..\,n-1]$ contains the smallest $n-1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

The running time of the algorithm is $\Theta(n^2)$ for all cases.

Solution to Exercise 2.2-4

Modify the algorithm so it tests whether the input satisfies some special-case condition and, if it does, output a pre-computed answer. The best-case running time is generally not a good measure of an algorithm.

Solution to Exercise 2.3-3

The base case is when $n = 2$, and we have $n \lg n = 2 \lg 2 = 2 \cdot 1 = 2$.

For the inductive step, our inductive hypothesis is that $T(n/2) = (n/2)\lg(n/2)$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2)\lg(n/2) + n \\ &= n(\lg n - 1) + n \\ &= n\lg n - n + n \\ &= n\lg n\ , \end{aligned}$$

which completes the inductive proof for exact powers of 2.

Solution to Exercise 2.3-4

Since it takes $\Theta(n)$ time in the worst case to insert $A[n]$ into the sorted array $A[1\,..\,n-1]$, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\ , \\ T(n-1) + \Theta(n) & \text{if } n > 1\ . \end{cases}$$

The solution to this recurrence is $T(n) = \Theta(n^2)$.

Solution to Exercise 2.3-5

Procedure BINARY-SEARCH takes a sorted array A, a value v, and a range $[low\,..\,high]$ of the array, in which we search for the value v. The procedure compares v to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration. We give both iterative and recursive versions, each of which returns either an index i such that $A[i] = v$, or NIL if no entry of $A[low\,..\,high]$ contains the value v. The initial call to either version should have the parameters $A, v, 1, n$.

```
ITERATIVE-BINARY-SEARCH(A, v, low, high)
while low ≤ high
    do mid ← ⌊(low + high)/2⌋
       if v = A[mid]
         then return mid
       if v > A[mid]
         then low ← mid + 1
         else high ← mid − 1
return NIL
```

```
RECURSIVE-BINARY-SEARCH(A, v, low, high)
if low > high
  then return NIL
mid ← ⌊(low + high)/2⌋
if v = A[mid]
  then return mid
if v > A[mid]
  then return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
  else return RECURSIVE-BINARY-SEARCH(A, v, low, mid − 1)
```

Both procedures terminate the search unsuccessfully when the range is empty (i.e., $low > high$) and terminate it successfully if the value v has been found. Based on the comparison of v to the middle element in the searched range, the search continues with the range halved. The recurrence for these procedures is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = \Theta(\lg n)$.

Solution to Exercise 2.3-6

The **while** loop of lines 5–7 of procedure INSERTION-SORT scans backward through the sorted array $A[1 \ldots j-1]$ to find the appropriate place for $A[j]$. The hitch is that the loop not only searches for the proper place for $A[j]$, but that it also moves each of the array elements that are bigger than $A[j]$ one position to the right (line 6). These movements can take as much as $\Theta(j)$ time, which occurs when all the $j-1$ elements preceding $A[j]$ are larger than $A[j]$. We can use binary search to improve the running time of the search to $\Theta(\lg j)$, but binary search will have no effect on the running time of moving the elements. Therefore, binary search alone cannot improve the worst-case running time of INSERTION-SORT to $\Theta(n \lg n)$.

Solution to Exercise 2.3-7

The following algorithm solves the problem:

1. Sort the elements in S.
2. Form the set $S' = \{z : z = x - y \text{ for some } y \in S\}$.
3. Sort the elements in S'.
4. If any value in S appears more than once, remove all but one instance. Do the same for S'.
5. Merge the two sorted sets S and S'.
6. There exist two elements in S whose sum is exactly x if and only if the same value appears in consecutive positions in the merged output.

To justify the claim in step 4, first observe that if any value appears twice in the merged output, it must appear in consecutive positions. Thus, we can restate the condition in step 5 as there exist two elements in S whose sum is exactly x if and only if the same value appears twice in the merged output.

Suppose that some value w appears twice. Then w appeared once in S and once in S'. Because w appeared in S', there exists some $y \in S$ such that $w = x - y$, or $x = w + y$. Since $w \in S$, the elements w and y are in S and sum to x.

Conversely, suppose that there are values $w, y \in S$ such that $w + y = x$. Then, since $x - y = w$, the value w appears in S'. Thus, w is in both S and S', and so it will appear twice in the merged output.

Steps 1 and 3 require $O(n \lg n)$ steps. Steps 2, 4, 5, and 6 require $O(n)$ steps. Thus the overall running time is $O(n \lg n)$.

Solution to Problem 2-1

[It may be better to assign this problem after covering asymptotic notation in Section 3.1; otherwise part (c) may be too difficult.]

a. Insertion sort takes $\Theta(k^2)$ time per k-element list in the worst case. Therefore, sorting n/k lists of k elements each takes $\Theta(k^2 n/k) = \Theta(nk)$ worst-case time.

b. Just extending the 2-list merge to merge all the lists at once would take $\Theta(n \cdot (n/k)) = \Theta(n^2/k)$ time (n from copying each element once into the result list, n/k from examining n/k lists at each step to select next item for result list).

To achieve $\Theta(n \lg(n/k))$-time merging, we merge the lists pairwise, then merge the resulting lists pairwise, and so on, until there's just one list. The pairwise merging requires $\Theta(n)$ work at each level, since we are still working on n elements, even if they are partitioned among sublists. The number of levels, starting with n/k lists (with k elements each) and finishing with 1 list (with n elements), is $\lceil \lg(n/k) \rceil$. Therefore, the total running time for the merging is $\Theta(n \lg(n/k))$.

c. The modified algorithm has the same asymptotic running time as standard merge sort when $\Theta(nk + n \lg(n/k)) = \Theta(n \lg n)$. The largest asymptotic value of k as a function of n that satisfies this condition is $k = \Theta(\lg n)$.

To see why, first observe that k cannot be more than $\Theta(\lg n)$ (i.e., it can't have a higher-order term than $\lg n$), for otherwise the left-hand expression wouldn't be $\Theta(n \lg n)$ (because it would have a higher-order term than $n \lg n$). So all we need to do is verify that $k = \Theta(\lg n)$ works, which we can do by plugging $k = \lg n$ into $\Theta(nk + n \lg(n/k)) = \Theta(nk + n \lg n - n \lg k)$ to get

$$\Theta(n \lg n + n \lg n - n \lg \lg n) = \Theta(2n \lg n - n \lg \lg n) \ ,$$

which, by taking just the high-order term and ignoring the constant coefficient, equals $\Theta(n \lg n)$.

d. In practice, k should be the largest list length on which insertion sort is faster than merge sort.

Solution to Problem 2-2

a. We need to show that the elements of A' form a permutation of the elements of A.

b. **Loop invariant:** At the start of each iteration of the **for** loop of lines 2–4, $A[j] = \min\{A[k] : j \le k \le n\}$ and the subarray $A[j\,..\,n]$ is a permutation of the values that were in $A[j\,..\,n]$ at the time that the loop started.

Initialization: Initially, $j = n$, and the subarray $A[j\,..\,n]$ consists of single element $A[n]$. The loop invariant trivially holds.

Maintenance: Consider an iteration for a given value of j. By the loop invariant, $A[j]$ is the smallest value in $A[j\,..\,n]$. Lines 3–4 exchange $A[j]$ and $A[j-1]$ if $A[j]$ is less than $A[j-1]$, and so $A[j-1]$ will be the smallest value in $A[j-1\,..\,n]$ afterward. Since the only change to the subarray $A[j-1\,..\,n]$ is this possible exchange, and the subarray $A[j\,..\,n]$ is a permutation of the values that were in $A[j\,..\,n]$ at the time that the loop started, we see that $A[j-1\,..\,n]$ is a permutation of the values that were in $A[j-1\,..\,n]$ at the time that the loop started. Decrementing j for the next iteration maintains the invariant.

Termination: The loop terminates when j reaches i. By the statement of the loop invariant, $A[i] = \min\{A[k] : i \le k \le n\}$ and $A[i\,..\,n]$ is a permutation of the values that were in $A[i\,..\,n]$ at the time that the loop started.

c. **Loop invariant:** At the start of each iteration of the **for** loop of lines 1–4, the subarray $A[1\,..\,i-1]$ consists of the $i-1$ smallest values originally in $A[1\,..\,n]$, in sorted order, and $A[i\,..\,n]$ consists of the $n-i+1$ remaining values originally in $A[1\,..\,n]$.

Initialization: Before the first iteration of the loop, $i = 1$. The subarray $A[1\,..\,i-1]$ is empty, and so the loop invariant vacuously holds.

Maintenance: Consider an iteration for a given value of i. By the loop invariant, $A[1\,..\,i-1]$ consists of the i smallest values in $A[1\,..\,n]$, in sorted order. Part (b) showed that after executing the **for** loop of lines 2–4, $A[i]$ is the smallest value in $A[i\,..\,n]$, and so $A[1\,..\,i]$ is now the i smallest values originally in $A[1\,..\,n]$, in sorted order. Moreover, since the **for** loop of lines 2–4 permutes $A[i\,..\,n]$, the subarray $A[i+1\,..\,n]$ consists of the $n-i$ remaining values originally in $A[1\,..\,n]$.

Termination: The **for** loop of lines 1–4 terminates when $i = n+1$, so that $i-1 = n$. By the statement of the loop invariant, $A[1\,..\,i-1]$ is the entire array $A[1\,..\,n]$, and it consists of the original array $A[1\,..\,n]$, in sorted order.

Note: We have received requests to change the upper bound of the outer **for** loop of lines 1–4 to $length[A] - 1$. That change would also result in a correct algorithm. The loop would terminate when $i = n$, so that according to the loop invariant, $A[1\,..\,n-1]$ would consist of the $n-1$ smallest values originally in $A[1\,..\,n]$, in sorted order, and $A[n]$ would contain the remaining element, which must be the largest in $A[1\,..\,n]$. Therefore, $A[1\,..\,n]$ would be sorted.

In the original pseudocode, the last iteration of the outer **for** loop results in no iterations of the inner **for** loop of lines 1–4. With the upper bound for i set to $length[A]-1$, the last iteration of outer loop would result in one iteration of the inner loop. Either bound, $length[A]$ or $length[A]-1$, yields a correct algorithm.

d. The running time depends on the number of iterations of the **for** loop of lines 2–4. For a given value of i, this loop makes $n-i$ iterations, and i takes on the values $1, 2, \ldots, n$. The total number of iterations, therefore, is

$$\begin{aligned}\sum_{i=1}^{n}(n-i) &= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i \\ &= n^2 - \frac{n(n+1)}{2} \\ &= n^2 - \frac{n^2}{2} - \frac{n}{2} \\ &= \frac{n^2}{2} - \frac{n}{2}\,.\end{aligned}$$

Thus, the running time of bubblesort is $\Theta(n^2)$ in all cases. The worst-case running time is the same as that of insertion sort.

Solution to Problem 2-4

a. The inversions are $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, $(4, 5)$. (Remember that inversions are specified by indices rather than by the values in the array.)

b. The array with elements from $\{1, 2, \ldots, n\}$ with the most inversions is $\langle n, n-1, n-2, \ldots, 2, 1\rangle$. For all $1 \le i < j \le n$, there is an inversion (i, j). The number of such inversions is $\binom{n}{2} = n(n-1)/2$.

c. Suppose that the array A starts out with an inversion (k, j). Then $k < j$ and $A[k] > A[j]$. At the time that the outer **for** loop of lines 1–8 sets $key \leftarrow A[j]$, the value that started in $A[k]$ is still somewhere to the left of $A[j]$. That is, it's in $A[i]$, where $1 \le i < j$, and so the inversion has become (i, j). Some iteration of the **while** loop of lines 5–7 moves $A[i]$ one position to the right. Line 8 will eventually drop *key* to the left of this element, thus eliminating the inversion. Because line 5 moves only elements that are less than *key*, it moves only elements that correspond to inversions. In other words, each iteration of the **while** loop of lines 5–7 corresponds to the elimination of one inversion.

d. We follow the hint and modify merge sort to count the number of inversions in $\Theta(n \lg n)$ time.

To start, let us define a ***merge-inversion*** as a situation within the execution of merge sort in which the MERGE procedure, after copying $A[p\,..\,q]$ to L and $A[q+1\,..\,r]$ to R, has values x in L and y in R such that $x > y$. Consider an inversion (i, j), and let $x = A[i]$ and $y = A[j]$, so that $i < j$ and $x > y$. We claim that if we were to run merge sort, there would be exactly one merge-inversion involving x and y. To see why, observe that the only way in which array elements change their positions is within the MERGE procedure. Moreover,

since MERGE keeps elements within L in the same relative order to each other, and correspondingly for R, the only way in which two elements can change their ordering relative to each other is for the greater one to appear in L and the lesser one to appear in R. Thus, there is at least one merge-inversion involving x and y. To see that there is exactly one such merge-inversion, observe that after any call of MERGE that involves both x and y, they are in the same sorted subarray and will therefore both appear in L or both appear in R in any given call thereafter. Thus, we have proven the claim.

We have shown that every inversion implies one merge-inversion. In fact, the correspondence between inversions and merge-inversions is one-to-one. Suppose we have a merge-inversion involving values x and y, where x originally was $A[i]$ and y was originally $A[j]$. Since we have a merge-inversion, $x > y$. And since x is in L and y is in R, x must be within a subarray preceding the subarray containing y. Therefore x started out in a position i preceding y's original position j, and so (i, j) is an inversion.

Having shown a one-to-one correspondence between inversions and merge-inversions, it suffices for us to count merge-inversions.

Consider a merge-inversion involving y in R. Let z be the smallest value in L that is greater than y. At some point during the merging process, z and y will be the "exposed" values in L and R, i.e., we will have $z = L[i]$ and $y = R[j]$ in line 13 of MERGE. At that time, there will be merge-inversions involving y and $L[i]$, $L[i+1]$, $L[i+2]$, ..., $L[n_1]$, and these $n_1 - i + 1$ merge-inversions will be the only ones involving y. Therefore, we need to detect the first time that z and y become exposed during the MERGE procedure and add the value of $n_1 - i + 1$ at that time to our total count of merge-inversions.

The following pseudocode, modeled on merge sort, works as we have just described. It also sorts the array A.

```
COUNT-INVERSIONS(A, p, r)
inversions ← 0
if p < r
  then q ← ⌊(p + r)/2⌋
       inversions ← inversions + COUNT-INVERSIONS(A, p, q)
       inversions ← inversions + COUNT-INVERSIONS(A, q + 1, r)
       inversions ← inversions + MERGE-INVERSIONS(A, p, q, r)
return inversions
```

```
MERGE-INVERSIONS(A, p, q, r)
n_1 ← q − p + 1
n_2 ← r − q
create arrays L[1 .. n_1 + 1] and R[1 .. n_2 + 1]
for i ← 1 to n_1
    do L[i] ← A[p + i − 1]
for j ← 1 to n_2
    do R[j] ← A[q + j]
L[n_1 + 1] ← ∞
R[n_2 + 1] ← ∞
i ← 1
j ← 1
inversions ← 0
counted ← FALSE
for k ← p to r
    do if counted = FALSE and R[j] < L[i]
          then inversions ← inversions + n_1 − i + 1
               counted ← TRUE
       if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
               counted ← FALSE
return inversions
```

The initial call is COUNT-INVERSIONS$(A, 1, n)$.

In MERGE-INVERSIONS, the boolean variable *counted* indicates whether we have counted the merge-inversions involving $R[j]$. We count them the first time that both $R[j]$ is exposed and a value greater than $R[j]$ becomes exposed in the L array. We set *counted* to FALSE upon each time that a new value becomes exposed in R. We don't have to worry about merge-inversions involving the sentinel ∞ in R, since no value in L will be greater than ∞.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last **for** loop of the merging procedure, the total running time of the above pseudocode is the same as for merge sort: $\Theta(n \lg n)$.

Lecture Notes for Chapter 3: Growth of Functions

Chapter 3 overview

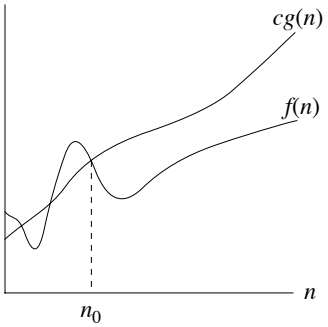
- A way to describe behavior of functions *in the limit*. We're studying ***asymptotic*** efficiency.
- Describe *growth* of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare "sizes" of functions:

O	$\approx$	$\leq$
Ω	$\approx$	$\geq$
Θ	$\approx$	$=$
o	$\approx$	$<$
ω	$\approx$	$>$

Asymptotic notation

O-notation

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$ (will precisely explain this soon).

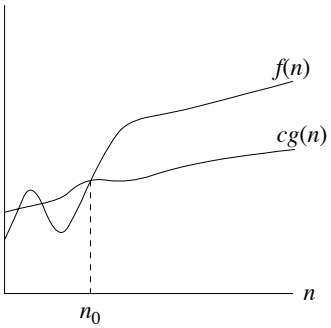
Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Ω-notation

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

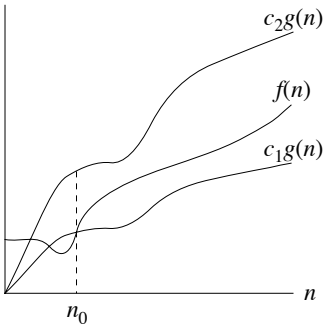
Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Θ-notation

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem
$f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$.

Leading constants and low-order terms don't matter.

Asymptotic notation in equations

When on right-hand side: $O(n^2)$ stands for some anonymous function in the set $O(n^2)$.

$2n^2+3n+1 = 2n^2+\Theta(n)$ means $2n^2+3n+1 = 2n^2+f(n)$ *for some* $f(n) \in \Theta(n)$. In particular, $f(n) = 3n + 1$.

By the way, we interpret # of anonymous functions as = # of times the asymptotic notation appears:

$\sum_{i=1}^{n} O(i)$ OK: 1 anonymous function

$O(1) + O(2) + \cdots + O(n)$ not OK: n hidden constants
$\Rightarrow$ no clean interpretation

When on left-hand side: No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.

Interpret $2n^2 + \Theta(n) = \Theta(n^2)$ as meaning *for all* functions $f(n) \in \Theta(n)$, there exists a function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$.

Can chain together:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \ . \end{aligned}$$

Interpretation:

- First equation: There exists $f(n) \in \Theta(n)$ such that $2n^2+3n+1 = 2n^2+f(n)$.
- Second equation: For all $g(n) \in \Theta(n)$ (such as the $f(n)$ used to make the first equation hold), there exists $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$.

o-notation

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \ne o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \ne o(n^2)$

ω-notation

$\omega(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \ne \omega(n^2)$

Comparisons of functions

Relational properties:

Transitivity:
$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$.
Same for O, Ω, o, and ω.

Reflexivity:
$f(n) = \Theta(f(n))$.
Same for O and Ω.

Symmetry:
$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry:
$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Comparisons:

- $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

No trichotomy. Although intuitively, we can liken O to $\le$, Ω to $\ge$, etc., unlike real numbers, where $a < b$, $a = b$, or $a > b$, we might not be able to compare functions.

Example: $n^{1+\sin n}$ and n, since $1 + \sin n$ oscillates between 0 and 2.

Standard notations and common functions

[You probably do not want to use lecture time going over all the definitions and properties given in Section 3.2, but it might be worth spending a few minutes of lecture time on some of the following.]

Monotonicity

- $f(n)$ is ***monotonically increasing*** if $m \le n \Rightarrow f(m) \le f(n)$.
- $f(n)$ is ***monotonically decreasing*** if $m \ge n \Rightarrow f(m) \ge f(n)$.
- $f(n)$ is ***strictly increasing*** if $m < n \Rightarrow f(m) < f(n)$.
- $f(n)$ is ***strictly decreasing*** if $m > n \Rightarrow f(m) > f(n)$.

Exponentials

Useful identities:

$$\begin{aligned} a^{-1} &= 1/a \ , \\ (a^m)^n &= a^{mn} \ , \\ a^m a^n &= a^{m+n} \ . \end{aligned}$$

Can relate rates of growth of polynomials and exponentials: for all real constants a and b such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 \ ,$$

which implies that $n^b = o(a^n)$.

A suprisingly useful inequality: for all real x,

$$e^x \ge 1 + x \ .$$

As x gets closer to 0, e^x gets closer to $1 + x$.

Logarithms

Notations:

$$\begin{aligned} \lg n &= \log_2 n \quad \text{(binary logarithm)} \ , \\ \ln n &= \log_e n \quad \text{(natural logarithm)} \ , \\ \lg^k n &= (\lg n)^k \quad \text{(exponentiation)} \ , \\ \lg \lg n &= \lg(\lg n) \quad \text{(composition)} \ . \end{aligned}$$

Logarithm functions apply only to the next term in the formula, so that $\lg n + k$ means $(\lg n) + k$, and *not* $\lg(n + k)$.

In the expression $\log_b a$:

- If we hold b constant, then the expression is strictly increasing as a increases.

- If we hold a constant, then the expression is strictly decreasing as b increases.

Useful identities for all real $a > 0$, $b > 0$, $c > 0$, and n, and where logarithm bases are not 1:

$$\begin{aligned}
a &= b^{\log_b a} , \\
\log_c(ab) &= \log_c a + \log_c b , \\
\log_b a^n &= n \log_b a , \\
\log_b a &= \frac{\log_c a}{\log_c b} , \\
\log_b(1/a) &= -\log_b a , \\
\log_b a &= \frac{1}{\log_a b} , \\
a^{\log_b c} &= c^{\log_b a} .
\end{aligned}$$

Changing the base of a logarithm from one constant to another only changes the value by a constant factor, so we usually don't worry about logarithm bases in asymptotic notation. Convention is to use lg within asymptotic notation, unless the base actually matters.

Just as polynomials grow more slowly than exponentials, logarithms grow more slowly than polynomials. In $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$, substitute $\lg n$ for n and 2^a for a:

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 ,$$

implying that $\lg^b n = o(n^a)$.

Factorials

$n! = 1 \cdot 2 \cdot 3 \cdot n$. Special case: $0! = 1$.

Can use ***Stirling's approximation***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) ,$$

to derive that $\lg(n!) = \Theta(n \lg n)$.

Solutions for Chapter 3: Growth of Functions

Solution to Exercise 3.1-1

First, let's clarify what the function $\max(f(n), g(n))$ is. Let's define the function $h(n) = \max(f(n), g(n))$. Then

$$h(n) = \begin{cases} f(n) & \text{if } f(n) \geq g(n) , \\ g(n) & \text{if } f(n) < g(n) . \end{cases}$$

Since $f(n)$ and $g(n)$ are asymptotically nonnegative, there exists n_0 such that $f(n) \geq 0$ and $g(n) \geq 0$ for all $n \geq n_0$. Thus for $n \geq n_0$, $f(n) + g(n) \geq f(n) \geq 0$ and $f(n)+g(n) \geq g(n) \geq 0$. Since for any particular n, $h(n)$ is either $f(n)$ or $g(n)$, we have $f(n) + g(n) \geq h(n) \geq 0$, which shows that $h(n) = \max(f(n), g(n)) \leq c_2(f(n) + g(n))$ for all $n \geq n_0$ (with $c_2 = 1$ in the definition of Θ).

Similarly, since for any particular n, $h(n)$ is the larger of $f(n)$ and $g(n)$, we have for all $n \geq n_0$, $0 \leq f(n) \leq h(n)$ and $0 \leq g(n) \leq h(n)$. Adding these two inequalities yields $0 \leq f(n) + g(n) \leq 2h(n)$, or equivalently $0 \leq (f(n) + g(n))/2 \leq h(n)$, which shows that $h(n) = \max(f(n), g(n)) \geq c_1(f(n)+g(n))$ for all $n \geq n_0$ (with $c_1 = 1/2$ in the definition of Θ).

Solution to Exercise 3.1-2

To show that $(n + a)^b = \Theta(n^b)$, we want to find constants $c_1, c_2, n_0 > 0$ such that $0 \leq c_1 n^b \leq (n + a)^b \leq c_2 n^b$ for all $n \geq n_0$.

Note that

$$\begin{aligned} n + a &\leq n + |a| \\ &\leq 2n \qquad \text{when } |a| \leq n , \end{aligned}$$

and

$$\begin{aligned} n + a &\geq n - |a| \\ &\geq \frac{1}{2}n \qquad \text{when } |a| \leq \tfrac{1}{2}n . \end{aligned}$$

Thus, when $n \geq 2|a|$,

$$0 \leq \frac{1}{2}n \leq n + a \leq 2n .$$

Since $b > 0$, the inequality still holds when all parts are raised to the power b:

$$0 \le \left(\frac{1}{2}n\right)^b \le (n+a)^b \le (2n)^b ,$$

$$0 \le \left(\frac{1}{2}\right)^b n^b \le (n+a)^b \le 2^b n^b .$$

Thus, $c_1 = (1/2)^b$, $c_2 = 2^b$, and $n_0 = 2|a|$ satisfy the definition.

Solution to Exercise 3.1-3

Let the running time be $T(n)$. $T(n) \ge O(n^2)$ means that $T(n) \ge f(n)$ for some function $f(n)$ in the set $O(n^2)$. This statement holds for any running time $T(n)$, since the function $g(n) = 0$ for all n is in $O(n^2)$, and running times are always nonnegative. Thus, the statement tells us nothing about the running time.

Solution to Exercise 3.1-4

$2^{n+1} = O(2^n)$, but $2^{2n} \ne O(2^n)$.

To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that

$0 \le 2^{n+1} \le c \cdot 2^n$ for all $n \ge n_0$.

Since $2^{n+1} = 2 \cdot 2^n$ for all n, we can satisfy the definition with $c = 2$ and $n_0 = 1$.

To show that $2^{2n} \ne O(2^n)$, assume there exist constants $c, n_0 > 0$ such that

$0 \le 2^{2n} \le c \cdot 2^n$ for all $n \ge n_0$.

Then $2^{2n} = 2^n \cdot 2^n \le c \cdot 2^n \Rightarrow 2^n \le c$. But no constant is greater than all 2^n, and so the assumption leads to a contradiction.

Solution to Exercise 3.1-8

$$\Omega(g(n,m)) = \{f(n,m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \le cg(n,m) \le f(n,m) \text{ for all } n \ge n_0 \text{ and } m \ge m_0\} .$$

$$\Theta(g(n,m)) = \{f(n,m) : \text{there exist positive constants } c_1, c_2, n_0, \text{ and } m_0 \text{ such that } 0 \le c_1 g(n,m) \le f(n,m) \le c_2 g(n,m) \text{ for all } n \ge n_0 \text{ and } m \ge m_0\} .$$

Solution to Exercise 3.2-4

$\lceil \lg n \rceil !$ is not polynomially bounded, but $\lceil \lg \lg n \rceil !$ is.

Proving that a function $f(n)$ is polynomially bounded is equivalent to proving that $\lg(f(n)) = O(\lg n)$ for the following reasons.

- If f is polynomially bounded, then there exist constants c, k, n_0 such that for all $n \geq n_0$, $f(n) \leq cn^k$. Hence, $\lg(f(n)) \leq kc \lg n$, which, since c and k are constants, means that $\lg(f(n)) = O(\lg n)$.
- Similarly, if $\lg(f(n)) = O(\lg n)$, then f is polynomially bounded.

In the following proofs, we will make use of the following two facts:

1. $\lg(n!) = \Theta(n \lg n)$ (by equation (3.18)).
2. $\lceil \lg n \rceil = \Theta(\lg n)$, because
   - $\lceil \lg n \rceil \geq \lg n$
   - $\lceil \lg n \rceil < \lg n + 1 \leq 2 \lg n$ for all $n \geq 2$

$$\begin{aligned} \lg(\lceil \lg n \rceil !) &= \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil) \\ &= \Theta(\lg n \lg \lg n) \\ &= \omega(\lg n) \ . \end{aligned}$$

Therefore, $\lg(\lceil \lg n \rceil !) \neq O(\lg n)$, and so $\lceil \lg n \rceil !$ is not polynomially bounded.

$$\begin{aligned} \lg(\lceil \lg \lg n \rceil !) &= \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) \\ &= \Theta(\lg \lg n \lg \lg \lg n) \\ &= o((\lg \lg n)^2) \\ &= o(\lg^2(\lg n)) \\ &= o(\lg n) \ . \end{aligned}$$

The last step above follows from the property that any polylogarithmic function grows more slowly than any positive polynomial function, i.e., that for constants $a, b > 0$, we have $\lg^b n = o(n^a)$. Substitute $\lg n$ for n, 2 for b, and 1 for a, giving $\lg^2(\lg n) = o(\lg n)$.

Therefore, $\lg(\lceil \lg \lg n \rceil !) = O(\lg n)$, and so $\lceil \lg \lg n \rceil !$ is polynomially bounded.

Solution to Problem 3-3

a. Here is the ordering, where functions on the same line are in the same equivalence class, and those higher on the page are Ω of those below them:

$2^{2^{n+1}}$	
2^{2^n}	
$(n+1)!$	
$n!$	see justification 7
e^n	see justification 1
$n \cdot 2^n$	
2^n	
$(3/2)^n$	
$(\lg n)^{\lg n} = n^{\lg \lg n}$	see identity 1
$(\lg n)!$	see justifications 2, 8
n^3	
$n^2 = 4^{\lg n}$	see identity 2
$n \lg n$ and $\lg(n!)$	see justification 6
$n = 2^{\lg n}$	see identity 3
$(\sqrt{2})^{\lg n} (= \sqrt{n})$	see identity 6, justification 3
$2^{\sqrt{2 \lg n}}$	see identity 5, justification 4
$\lg^2 n$	
$\ln n$	
$\sqrt{\lg n}$	
$\ln \ln n$	see justification 5
$2^{\lg^* n}$	
$\lg^* n$ and $\lg^*(\lg n)$	see identity 7
$\lg(\lg^* n)$	
$n^{1/\lg n} (= 2)$ and 1	see identity 4

Much of the ranking is based on the following properties:

- Exponential functions grow faster than polynomial functions, which grow faster than polylogarithmic functions.
- The base of a logarithm doesn't matter asymptotically, but the base of an exponential and the degree of a polynomial do matter.

We have the following *identities*:

1. $(\lg n)^{\lg n} = n^{\lg \lg n}$ because $a^{\log_b c} = c^{\log_b a}$.
2. $4^{\lg n} = n^2$ because $a^{\log_b c} = c^{\log_b a}$.
3. $2^{\lg n} = n$.
4. $2 = n^{1/\lg n}$ by raising identity 3 to the power $1/\lg n$.
5. $2^{\sqrt{2 \lg n}} = n^{\sqrt{2/\lg n}}$ by raising identity 4 to the power $\sqrt{2 \lg n}$.
6. $\left(\sqrt{2}\right)^{\lg n} = \sqrt{n}$ because $\left(\sqrt{2}\right)^{\lg n} = 2^{(1/2)\lg n} = 2^{\lg \sqrt{n}} = \sqrt{n}$.
7. $\lg^*(\lg n) = (\lg^* n) - 1$.

The following *justifications* explain some of the rankings:

1. $e^n = 2^n (e/2)^n = \omega(n 2^n)$, since $(e/2)^n = \omega(n)$.
2. $(\lg n)! = \omega(n^3)$ by taking logs: $\lg(\lg n)! = \Theta(\lg n \lg \lg n)$ by Stirling's approximation, $\lg(n^3) = 3 \lg n$. $\lg \lg n = \omega(3)$.

3. $(\sqrt{2})^{\lg n} = \omega\left(2^{\sqrt{2\lg n}}\right)$ by taking logs: $\lg(\sqrt{2})^{\lg n} = (1/2)\lg n$, $\lg 2^{\sqrt{2\lg n}} = \sqrt{2\lg n}$. $(1/2)\lg n = \omega(\sqrt{2\lg n})$.
4. $2^{\sqrt{2\lg n}} = \omega(\lg^2 n)$ by taking logs: $\lg 2^{\sqrt{2\lg n}} = \sqrt{2\lg n}$, $\lg\lg^2 n = 2\lg\lg n$. $\sqrt{2\lg n} = \omega(2\lg\lg n)$.
5. $\ln\ln n = \omega(2^{\lg^* n})$ by taking logs: $\lg 2^{\lg^* n} = \lg^* n$. $\lg\ln\ln n = \omega(\lg^* n)$.
6. $\lg(n!) = \Theta(n\lg n)$ (equation (3.18)).
7. $n! = \Theta(n^{n+1/2}e^{-n})$ by dropping constants and low-order terms in equation (3.17).
8. $(\lg n)! = \Theta((\lg n)^{\lg n+1/2}e^{-\lg n})$ by substituting $\lg n$ for n in the previous justification. $(\lg n)! = \Theta((\lg n)^{\lg n+1/2}n^{-\lg e})$ because $a^{\log_b c} = c^{\log_b a}$.

b. The following $f(n)$ is nonnegative, and for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

$$f(n) = \begin{cases} 2^{2^{n+2}} & \text{if } n \text{ is even}, \\ 0 & \text{if } n \text{ is odd}. \end{cases}$$

Lecture Notes for Chapter 4: Recurrences

Chapter 4 overview

A ***recurrence*** is a function is defined in terms of

- one or more base cases, and
- itself, with smaller arguments.

Examples:

- $T(n) = \begin{cases} 1 & \text{if } n = 1 \,, \\ T(n-1) + 1 & \text{if } n > 1 \,. \end{cases}$

  Solution: $T(n) = n$.

- $T(n) = \begin{cases} 1 & \text{if } n = 1 \,, \\ 2T(n/2) + n & \text{if } n \geq 1 \,. \end{cases}$

  Solution: $T(n) = n \lg n + n$.

- $T(n) = \begin{cases} 0 & \text{if } n = 2 \,, \\ T(\sqrt{n}) + 1 & \text{if } n > 2 \,. \end{cases}$

  Solution: $T(n) = \lg \lg n$.

- $T(n) = \begin{cases} 1 & \text{if } n = 1 \,, \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 \,. \end{cases}$

  Solution: $T(n) = \Theta(n \lg n)$.

[The notes for this chapter are fairly brief because we teach recurrences in much greater detail in a separate discrete math course.]

Many technical issues:

- Floors and ceilings

  [Floors and ceilings can easily be removed and don't affect the solution to the recurrence. They are better left to a discrete math course.]
- Exact vs. asymptotic functions
- Boundary conditions

In algorithm analysis, we usually express both the recurrence and its solution using asymptotic notation.

- Example: $T(n) = 2T(n/2) + \Theta(n)$, with solution $T(n) = \Theta(n \lg n)$.
- The boundary conditions are usually expressed as "$T(n) = O(1)$ for sufficiently small n."
- When we desire an exact, rather than an asymptotic, solution, we need to deal with boundary conditions.
- In practice, we just use asymptotics most of the time, and we ignore boundary conditions.

[In my course, there are only two acceptable ways of solving recurrences: the substitution method and the master method. Unless the recursion tree is carefully accounted for, I do not accept it as a proof of a solution, though I certainly accept a recursion tree as a way to generate a guess for substitution method. You may choose to allow recursion trees as proofs in your course, in which case some of the substitution proofs in the solutions for this chapter become recursion trees.

I also never use the iteration method, which had appeared in the first edition of Introduction to Algorithms. I find that it is too easy to make an error in parenthesization, and that recursion trees give a better intuitive idea than iterating the recurrence of how the recurrence progresses.]

Substitution method

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 , \\ 2T(n/2) + n & \text{if } n > 1 . \end{cases}$$

1. *Guess:* $T(n) = n \lg n + n$. *[Here, we have a recurrence with an exact function, rather than asymptotic notation, and the solution is also exact rather than asymptotic. We'll have to check boundary conditions and the base case.]*
2. *Induction:*

   Basis: $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

   Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$. We'll use this inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(\frac{n}{2} \lg \frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\ &= n \lg \frac{n}{2} + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n \lg n - n + n + n \\ &= n \lg n + n . \end{aligned}$$

■

Generally, we use asymptotic notation:

- We would write $T(n) = 2T(n/2) + \Theta(n)$.
- We assume $T(n) = O(1)$ for sufficiently small n.
- We express the solution by asymptotic notation: $T(n) = \Theta(n \lg n)$.
- We don't worry about boundary cases, nor do we show base cases in the substitution proof.
  - $T(n)$ is always constant for any constant n.
  - Since we are ultimately interested in an asymptotic solution to a recurrence, it will always be possible to choose base cases that work.
  - When we want an asymptotic solution to a recurrence, we don't worry about the base cases in our proofs.
  - When we want an exact solution, then we have to deal with base cases.

For the substitution method:

- Name the constant in the additive term.
- Show the upper (O) and lower (Ω) bounds separately. Might need to use different constants for each.

Example: $T(n) = 2T(n/2)+\Theta(n)$. If we want to show an upper bound of $T(n) = 2T(n/2) + O(n)$, we write $T(n) \le 2T(n/2) + cn$ for some positive constant c.

1. ***Upper bound:***

   Guess: $T(n) \le dn \lg n$ for some positive constant d. We are given c in the recurrence, and we get to choose d as any positive constant. It's OK for d to depend on c.

   Substitution:

   $$\begin{aligned} T(n) &\le 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\le dn\lg n \qquad \text{if } -dn + cn \le 0\,, \\ &\qquad\qquad\qquad\quad d \ge c \end{aligned}$$

   Therefore, $T(n) = O(n \lg n)$.

2. ***Lower bound:*** Write $T(n) \ge 2T(n/2) + cn$ for some positive constant c.

   Guess: $T(n) \ge dn \lg n$ for some positive constant d.

   Substitution:

   $$\begin{aligned} T(n) &\ge 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\ge dn\lg n \qquad \text{if } -dn + cn \ge 0\,, \\ &\qquad\qquad\qquad\quad d \le c \end{aligned}$$

Therefore, $T(n) = \Omega(n \lg n)$.

Therefore, $T(n) = \Theta(n \lg n)$. *[For this particular recurrence, we can use $d = c$ for both the upper-bound and lower-bound proofs. That won't always be the case.]* ■

Make sure you show the same *exact* form when doing a substitution proof.

Consider the recurrence

$T(n) = 8T(n/2) + \Theta(n^2)$.

For an upper bound:

$T(n) \le 8T(n/2) + cn^2$.

Guess: $T(n) \le dn^3$.

$$\begin{aligned} T(n) &\le 8d(n/2)^3 + cn^2 \\ &= 8d(n^3/8) + cn^2 \\ &= dn^3 + cn^2 \\ &\not\le dn^3 \qquad \text{doesn't work!} \end{aligned}$$

Remedy: *Subtract off* a lower-order term.

Guess: $T(n) \le dn^3 - d'n^2$.

$$\begin{aligned} T(n) &\le 8(d(n/2)^3 - d'(n/2)^2) + cn^2 \\ &= 8d(n^3/8) - 8d'(n^2/4) + cn^2 \\ &= dn^3 - 2d'n^2 + cn^2 \\ &= dn^3 - d'n^2 - d'n^2 + cn^2 \\ &\le dn^3 - d'n^2 \qquad \text{if } -d'n^2 + cn^2 \le 0 , \\ &\qquad\qquad\qquad\qquad\quad d' \ge c \end{aligned}$$

Be careful when using asymptotic notation.

The false proof for the recurrence $T(n) = 4T(n/4) + n$, that $T(n) = O(n)$:

$$\begin{aligned} T(n) &\le 4(c(n/4)) + n \\ &\le cn + n \\ &= O(n) \qquad \text{wrong!} \end{aligned}$$

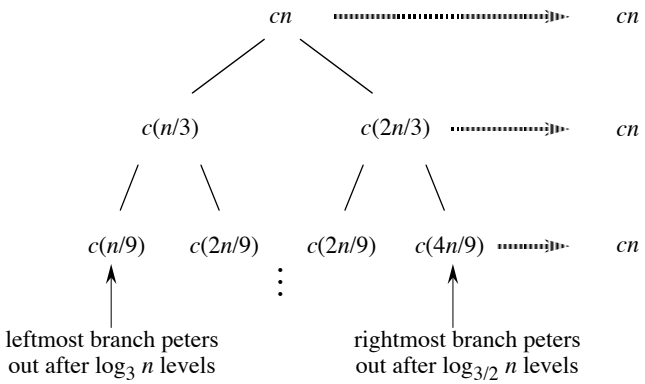
Because we haven't proven the *exact form* of our inductive hypothesis (which is that $T(n) \le cn$), this proof is false.

Recursion trees

Use to generate a guess. Then verify by substitution method.

Example: $T(n) = T(n/3) + T(2n/3) + \Theta(n)$. For upper bound, rewrite as $T(n) \le T(n/3) + T(2n/3) + cn$; for lower bound, as $T(n) \ge T(n/3) + T(2n/3) + cn$.

By summing across each level, the recursion tree shows the cost at each level of recursion (minus the costs of recursive calls, which appear in subtrees):



- There are $\log_3 n$ full levels, and after $\log_{3/2} n$ levels, the problem size is down to 1.
- Each level contributes $\le cn$.
- Lower bound guess: $\ge dn \log_3 n = \Omega(n \lg n)$ for some positive constant d.
- Upper bound guess: $\le dn \log_{3/2} n = O(n \lg n)$ for some positive constant d.
- Then *prove* by substitution.

1. ***Upper bound:***

   Guess: $T(n) \le dn \lg n$.

   Substitution:

$$\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n \qquad \text{if } -dn(\lg 3 - 2/3) + cn \le 0 , \\
&\qquad\qquad\qquad\qquad\qquad\qquad d \ge \frac{c}{\lg 3 - 2/3} .
\end{aligned}$$

   Therefore, $T(n) = O(n \lg n)$.

   Note: Make sure that the symbolic constants used in the recurrence (e.g., c) and the guess (e.g., d) are different.

2. ***Lower bound:***

   Guess: $T(n) \ge dn \lg n$.

   Substitution: Same as for the upper bound, but replacing $\le$ by $\ge$. End up needing

$$0 < d \le \frac{c}{\lg 3 - 2/3} .$$

   Therefore, $T(n) = \Omega(n \lg n)$.

Since $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$, we conclude that $T(n) = \Theta(n \lg n)$. ■

Master method

Used for many divide-and-conquer recurrences of the form

$T(n) = aT(n/b) + f(n)$,

where $a \geq 1$, $b > 1$, and $f(n) > 0$.

Based on the ***master theorem*** (Theorem 4.1).

Compare $n^{\log_b a}$ vs. $f(n)$:

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
($f(n)$ is polynomially smaller than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(n^{\log_b a})$.
(Intuitively: cost is dominated by leaves.)

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.
[This formulation of Case 2 is more general than in Theorem 4.1, and it is given in Exercise 4.4-2.]
($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
(Intuitively: cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)
Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.
($f(n)$ is polynomially greater than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(f(n))$.
(Intuitively: cost is dominated by root.)

What's with the Case 3 regularity condition?

- Generally not a problem.
- It always holds whenever $f(n) = n^k$ and $f(n) = \Omega(n^{\log_b a + \epsilon})$ for constant $\epsilon > 0$. *[Proving this makes a nice homework exercise. See below.]* So you don't need to check it when $f(n)$ is a polynomial.

[Here's a proof that the regularity condition holds when $f(n) = n^k$ and $f(n) = \Omega(n^{\log_b a + \epsilon})$ for constant $\epsilon > 0$.

Since $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $f(n) = n^k$, we have that $k > \log_b a$. Using a base of b and treating both sides as exponents, we have $b^k > b^{\log_b a} = a$, and so $a/b^k < 1$. Since a, b, and k are constants, if we let $c = a/b^k$, then c is a constant strictly less than 1. We have that $af(n/b) = a(n/b)^k = (a/b^k)n^k = cf(n)$, and so the regularity condition is satisfied.]

Examples:

- $T(n) = 5T(n/2) + \Theta(n^2)$
  $n^{\log_2 5}$ vs. n^2
  Since $\log_2 5 - \epsilon = 2$ for some constant $\epsilon > 0$, use Case 1 $\Rightarrow T(n) = \Theta(n^{\lg 5})$

- $T(n) = 27T(n/3) + \Theta(n^3 \lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3 \lg n$
  Use Case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n)$

- $T(n) = 5T(n/2) + \Theta(n^3)$
  $n^{\log_2 5}$ vs. n^3
  Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$
  Check regularity condition (don't really need to since $f(n)$ is a polynomial):
  $af(n/b) = 5(n/2)^3 = 5n^3/8 \le cn^3$ for $c = 5/8 < 1$
  Use Case 3 $\Rightarrow T(n) = \Theta(n^3)$

- $T(n) = 27T(n/3) + \Theta(n^3/\lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3/\lg n = n^3 \lg^{-1} n \ne \Theta(n^3 \lg^k n)$ for any $k \ge 0$.
  Cannot use the master method.

[We don't prove the master theorem in our algorithms course. We sometimes prove a simplified version for recurrences of the form $T(n) = aT(n/b) + n^c$. Section 4.4 of the text has the full proof of the master theorem.]

Solutions for Chapter 4: Recurrences

Solution to Exercise 4.2-2

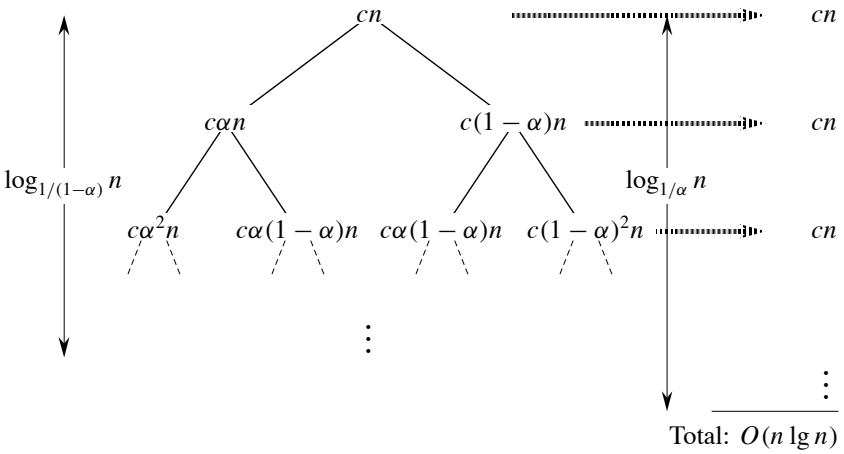
The shortest path from the root to a leaf in the recursion tree is $n \to (1/3)n \to (1/3)^2 n \to \cdots \to 1$. Since $(1/3)^k n = 1$ when $k = \log_3 n$, the height of the part of the tree in which every node has two children is $\log_3 n$. Since the values at each of these levels of the tree add up to n, the solution to the recurrence is at least $n \log_3 n = \Omega(n \lg n)$.

Solution to Exercise 4.2-5

$T(n) = T(\alpha n) + T((1-\alpha)n) + n$

We saw the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + cn$ in the text. This recurrence can be similarly solved.

Without loss of generality, let $\alpha \geq 1-\alpha$, so that $0 < 1-\alpha \leq 1/2$ and $1/2 \leq \alpha < 1$.



The recursion tree is full for $\log_{1/(1-\alpha)} n$ levels, each contributing cn, so we guess $\Omega(n \log_{1/(1-\alpha)} n) = \Omega(n \lg n)$. It has $\log_{1/\alpha} n$ levels, each contributing $\leq cn$, so we guess $O(n \log_{1/\alpha} n) = O(n \lg n)$.

Now we show that $T(n) = \Theta(n \lg n)$ by substitution. To prove the upper bound, we need to show that $T(n) \le dn \lg n$ for a suitable constant $d > 0$.

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\ &\le d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn \\ &= d\alpha n \lg\alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\ &= dn \lg n + dn(\alpha \lg\alpha + (1-\alpha)\lg(1-\alpha)) + cn \\ &\le dn \lg n \ , \end{aligned}$$

if $dn(\alpha \lg\alpha + (1-\alpha)\lg(1-\alpha)) + cn \le 0$. This condition is equivalent to

$$d(\alpha \lg\alpha + (1-\alpha)\lg(1-\alpha)) \le -c \ .$$

Since $1/2 \le \alpha < 1$ and $0 < 1-\alpha \le 1/2$, we have that $\lg\alpha < 0$ and $\lg(1-\alpha) < 0$. Thus, $\alpha \lg\alpha + (1-\alpha)\lg(1-\alpha) < 0$, so that when we multiply both sides of the inequality by this factor, we need to reverse the inequality:

$$d \ge \frac{-c}{\alpha \lg\alpha + (1-\alpha)\lg(1-\alpha)}$$

or

$$d \ge \frac{c}{-\alpha \lg\alpha + -(1-\alpha)\lg(1-\alpha)} \ .$$

The fraction on the right-hand side is a positive constant, and so it suffices to pick any value of d that is greater than or equal to this fraction.

To prove the lower bound, we need to show that $T(n) \ge dn \lg n$ for a suitable constant $d > 0$. We can use the same proof as for the upper bound, substituting $\ge$ for $\le$, and we get the requirement that

$$0 < d \le \frac{c}{-\alpha \lg\alpha - (1-\alpha)\lg(1-\alpha)} \ .$$

Therefore, $T(n) = \Theta(n \lg n)$.

Solution to Problem 4-1

Note: In parts (a), (b), and (d) below, we are applying case 3 of the master theorem, which requires the regularity condition that $af(n/b) \le cf(n)$ for some constant $c < 1$. In each of these parts, $f(n)$ has the form n^k. The regularity condition is satisfied because $af(n/b) = an^k/b^k = (a/b^k)n^k = (a/b^k)f(n)$, and in each of the cases below, a/b^k is a constant strictly less than 1.

a. $T(n) = 2T(n/2) + n^3 = \Theta(n^3)$. This is a divide-and-conquer recurrence with $a = 2$, $b = 2$, $f(n) = n^3$, and $n^{\log_b a} = n^{\log_2 2} = n$. Since $n^3 = \Omega(n^{\log_2 2+2})$ and $a/b^k = 2/2^3 = 1/4 < 1$, case 3 of the master theorem applies, and $T(n) = \Theta(n^3)$.

b. $T(n) = T(9n/10) + n = \Theta(n)$. This is a divide-and-conquer recurrence with $a = 1$, $b = 10/9$, $f(n) = n$, and $n^{\log_b a} = n^{\log_{10/9} 1} = n^0 = 1$. Since $n = \Omega(n^{\log_{10/9} 1+1})$ and $a/b^k = 1/(10/9)^1 = 9/10 < 1$, case 3 of the master theorem applies, and $T(n) = \Theta(n)$.

c. $T(n) = 16T(n/4) + n^2 = \Theta(n^2 \lg n)$. This is another divide-and-conquer recurrence with $a = 16$, $b = 4$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_4 16} = n^2$. Since $n^2 = \Theta(n^{\log_4 16})$, case 2 of the master theorem applies, and $T(n) = \Theta(n^2 \lg n)$.

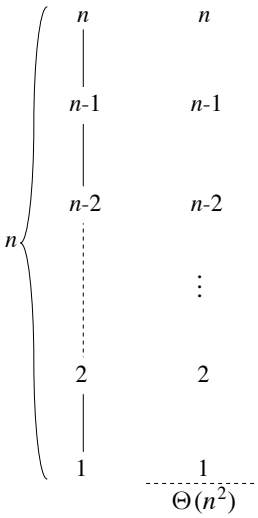
d. $T(n) = 7T(n/3) + n^2 = \Theta(n^2)$. This is a divide-and-conquer recurrence with $a = 7$, $b = 3$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_3 7}$. Since $1 < \log_3 7 < 2$, we have that $n^2 = \Omega(n^{\log_3 7 + \epsilon})$ for some constant $\epsilon > 0$. We also have $a/b^k = 7/3^2 = 7/9 < 1$, so that case 3 of the master theorem applies, and $T(n) = \Theta(n^2)$.

e. $T(n) = 7T(n/2) + n^2 = O(n^{\lg 7})$. This is a divide-and-conquer recurrence with $a = 7$, $b = 2$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_2 7}$. Since $2 < \lg 7 < 3$, we have that $n^2 = O(n^{\log_2 7 - \epsilon})$ for some constant $\epsilon > 0$. Thus, case 1 of the master theorem applies, and $T(n) = \Theta(n^{\lg 7})$.

f. $T(n) = 2T(n/4) + \sqrt{n} = \Theta(\sqrt{n} \lg n)$. This is another divide-and-conquer recurrence with $a = 2$, $b = 4$, $f(n) = \sqrt{n}$, and $n^{\log_b a} = n^{\log_4 2} = \sqrt{n}$. Since $\sqrt{n} = \Theta(n^{\log_4 2})$, case 2 of the master theorem applies, and $T(n) = \Theta(\sqrt{n} \lg n)$.

g. $T(n) = T(n-1) + n$

Using the recursion tree shown below, we get a guess of $T(n) = \Theta(n^2)$.



First, we prove the $T(n) = \Omega(n^2)$ part by induction. The inductive hypothesis is $T(n) \geq cn^2$ for some constant $c > 0$.

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\geq c(n-1)^2 + n \\ &= cn^2 - 2cn + c + n \\ &\geq cn^2 \end{aligned}$$

if $-2cn + n + c \geq 0$ or, equivalently, $n(1 - 2c) + c \geq 0$. This condition holds when $n \geq 0$ and $0 < c \leq 1/2$.

For the upper bound, $T(n) = O(n^2)$, we use the inductive hypothesis that $T(n) \leq cn^2$ for some constant $c > 0$. By a similar derivation, we get that

$T(n) \le cn^2$ if $-2cn + n + c \le 0$ or, equivalently, $n(1 - 2c) + c \le 0$. This condition holds for $c = 1$ and $n \ge 1$.

Thus, $T(n) = \Omega(n^2)$ and $T(n) = O(n^2)$, so we conclude that $T(n) = \Theta(n^2)$.

h. $T(n) = T(\sqrt{n}) + 1$

The easy way to do this is with a change of variables, as on page 66 of the text. Let $m = \lg n$ and $S(m) = T(2^m)$. $T(2^m) = T(2^{m/2}) + 1$, so $S(m) = S(m/2) + 1$. Using the master theorem, $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$ and $f(n) = 1$. Since $1 = \Theta(1)$, case 2 applies and $S(m) = \Theta(\lg m)$. Therefore, $T(n) = \Theta(\lg \lg n)$.

Solution to Problem 4-4

[This problem is solved only for parts a, c, e, f, g, h, and i.]

a. $T(n) = 3T(n/2) + n \lg n$

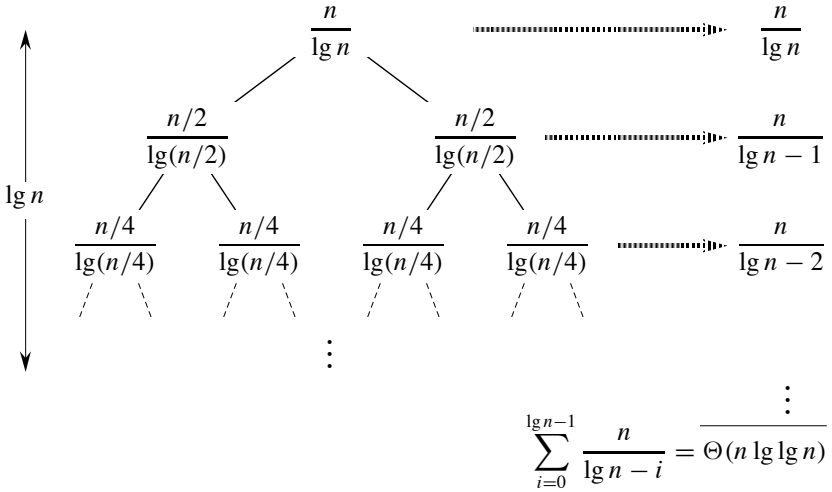
We have $f(n) = n \lg n$ and $n^{\log_b a} = n^{\lg 3} \approx n^{1.585}$. Since $n \lg n = O(n^{\lg 3 - \epsilon})$ for any $0 < \epsilon \le 0.58$, by case 1 of the master theorem, we have $T(n) = \Theta(n^{\lg 3})$.

c. $T(n) = 4T(n/2) + n^2\sqrt{n}$

We have $f(n) = n^2\sqrt{n} = n^{5/2}$ and $n^{\log_b a} = n^{\log_2 4} = n^{\lg 2}$. Since $n^{5/2} = \Omega(n^{\lg 2 + 3/2})$, we look at the regularity condition in case 3 of the master theorem. We have $af(n/b) = 4(n/2)^2\sqrt{n/2} = n^{5/2}/\sqrt{2} \le cn^{5/2}$ for $1/\sqrt{2} \le c < 1$. Case 3 applies, and we have $T(n) = \Theta(n^2\sqrt{n})$.

e. $T(n) = 2T(n/2) + n/\lg n$

We can get a guess by means of a recursion tree:



We get the sum on each level by observing that at depth i, we have 2^i nodes, each with a numerator of $n/2^i$ and a denominator of $\lg(n/2^i) = \lg n - i$, so that the cost at depth i is

$$2^i \cdot \frac{n/2^i}{\lg n - i} = \frac{n}{\lg n - i} .$$

The sum for all levels is

$$\begin{aligned}
\sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} &= n \sum_{i=1}^{\lg n} \frac{n}{i} \\
&= n \sum_{i=1}^{\lg n} 1/i \\
&= n \cdot \Theta(\lg \lg n) \quad \text{(by equation (A.7), the harmonic series)} \\
&= \Theta(n \lg \lg n) .
\end{aligned}$$

We can use this analysis as a guess that $T(n) = \Theta(n \lg \lg n)$. If we were to do a straight substitution proof, it would be rather involved. Instead, we will show by substitution that $T(n) \le n(1 + H_{\lfloor \lg n \rfloor})$ and $T(n) \ge n \cdot H_{\lceil \lg n \rceil}$, where H_k is the kth harmonic number: $H_k = 1/1 + 1/2 + 1/3 + \cdots + 1/k$. We also define $H_0 = 0$. Since $H_k = \Theta(\lg k)$, we have that $H_{\lfloor \lg n \rfloor} = \Theta(\lg \lfloor \lg n \rfloor) = \Theta(\lg \lg n)$ and $H_{\lceil \lg n \rceil} = \Theta(\lg \lceil \lg n \rceil) = \Theta(\lg \lg n)$. Thus, we will have that $T(n) = \Theta(n \lg \lg n)$.

The base case for the proof is for $n = 1$, and we use $T(1) = 1$. Here, $\lg n = 0$, so that $\lg n = \lfloor \lg n \rfloor = \lceil \lg n \rceil$. Since $H_0 = 0$, we have $T(1) = 1 \le 1(1 + H_0)$ and $T(1) = 1 \ge 0 = 1 \cdot H_0$.

For the upper bound of $T(n) \le n(1 + H_{\lfloor \lg n \rfloor})$, we have

$$\begin{aligned}
T(n) &= 2T(n/2) + n/\lg n \\
&\le 2((n/2)(1 + H_{\lfloor \lg(n/2) \rfloor})) + n/\lg n \\
&= n(1 + H_{\lfloor \lg n - 1 \rfloor}) + n/\lg n \\
&= n(1 + H_{\lfloor \lg n \rfloor - 1} + 1/\lg n) \\
&\le n(1 + H_{\lfloor \lg n \rfloor - 1} + 1/\lfloor \lg n \rfloor) \\
&= n(1 + H_{\lfloor \lg n \rfloor}) ,
\end{aligned}$$

where the last line follows from the identity $H_k = H_{k-1} + 1/k$.

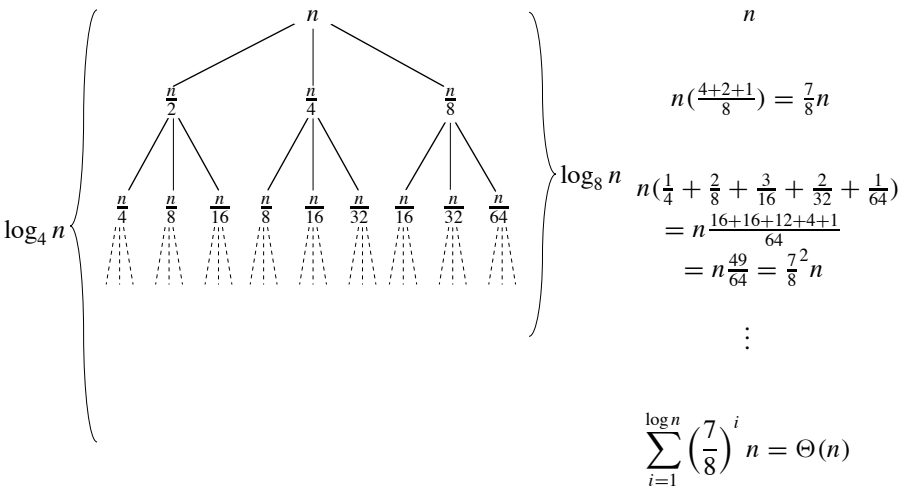
The upper bound of $T(n) \ge n \cdot H_{\lceil \lg n \rceil}$ is similar:

$$\begin{aligned}
T(n) &= 2T(n/2) + n/\lg n \\
&\ge 2((n/2) \cdot H_{\lceil \lg(n/2) \rceil}) + n/\lg n \\
&= n \cdot H_{\lceil \lg n - 1 \rceil} + n/\lg n \\
&= n \cdot (H_{\lceil \lg n \rceil - 1} + 1/\lg n) \\
&\ge n \cdot (H_{\lceil \lg n \rceil - 1} + 1/\lceil \lg n \rceil) \\
&= n \cdot H_{\lceil \lg n \rceil} .
\end{aligned}$$

Thus, $T(n) = \Theta(n \lg \lg n)$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

Using the recursion tree shown below, we get a guess of $T(n) = \Theta(n)$.



We use the substitution method to prove that $T(n) = O(n)$. Our inductive hypothesis is that $T(n) \le cn$ for some constant $c > 0$. We have

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + T(n/8) + n \\ &\le cn/2 + cn/4 + cn/8 + n \\ &= 7cn/8 + n \\ &= (1 + 7c/8)n \\ &\le cn \qquad \text{if } c \ge 8 . \end{aligned}$$

Therefore, $T(n) = O(n)$.

Showing that $T(n) = \Omega(n)$ is easy:

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n \ge n .$$

Since $T(n) = O(n)$ and $T(n) = \Omega(n)$, we have that $T(n) = \Theta(n)$.

g. $T(n) = T(n-1) + 1/n$

This recurrence corresponds to the harmonic series, so that $T(n) = H_n$, where $H_n = 1/1 + 1/2 + 1/3 + \cdots + 1/n$. For the base case, we have $T(1) = 1 = H_1$. For the inductive step, we assume that $T(n-1) = H_{n-1}$, and we have

$$\begin{aligned} T(n) &= T(n-1) + 1/n \\ &= H_{n-1} + 1/n \\ &= H_n . \end{aligned}$$

Since $H_n = \Theta(\lg n)$ by equation (A.7), we have that $T(n) = \Theta(\lg n)$.

h. $T(n) = T(n-1) + \lg n$

We guess that $T(n) = \Theta(n \lg n)$. To prove the upper bound, we will show that $T(n) = O(n \lg n)$. Our inductive hypothesis is that $T(n) \le cn \lg n$ for some constant c. We have

$$\begin{aligned}
T(n) &= T(n-1) + \lg n \\
&\le c(n-1)\lg(n-1) + \lg n \\
&= cn\lg(n-1) - c\lg(n-1) + \lg n \\
&\le cn\lg(n-1) - c\lg(n/2) + \lg n \\
&\qquad \text{(since } \lg(n-1) \ge \lg(n/2) \text{ for } n \ge 2\text{)} \\
&= cn\lg(n-1) - c\lg n + c + \lg n \\
&< cn\lg n - c\lg n + c + \lg n \\
&\le cn\lg n \ ,
\end{aligned}$$

if $-c\lg n + c + \lg n \le 0$. Equivalently,

$$\begin{aligned}
-c\lg n + c + \lg n &\le 0 \\
c &\le (c-1)\lg n \\
\lg n &\ge c/(c-1) \ .
\end{aligned}$$

This works for $c = 2$ and all $n \ge 4$.

To prove the lower bound, we will show that $T(n) = \Omega(n \lg n)$. Our inductive hypothesis is that $T(n) \ge cn\lg n + dn$ for constants c and d. We have

$$\begin{aligned}
T(n) &= T(n-1) + \lg n \\
&\ge c(n-1)\lg(n-1) + d(n-1) + \lg n \\
&= cn\lg(n-1) - c\lg(n-1) + dn - d + \lg n \\
&\ge cn\lg(n/2) - c\lg(n-1) + dn - d + \lg n \\
&\qquad \text{(since } \lg(n-1) \ge \lg(n/2) \text{ for } n \ge 2\text{)} \\
&= cn\lg n - cn - c\lg(n-1) + dn - d + \lg n \\
&\ge cn\lg n \ ,
\end{aligned}$$

if $-cn - c\lg(n-1) + dn - d + \lg n \ge 0$. Since

$$-cn - c\lg(n-1) + dn - d + \lg n > -cn - c\lg(n-1) + dn - d + \lg(n-1) \ ,$$

it suffices to find conditions in which $-cn - c\lg(n-1) + dn - d + \lg(n-1) \ge 0$. Equivalently,

$$\begin{aligned}
-cn - c\lg(n-1) + dn - d + \lg(n-1) &\ge 0 \\
(d-c)n &\ge (c-1)\lg(n-1) + d \ .
\end{aligned}$$

This works for $c = 1$, $d = 2$, and all $n \ge 2$.

Since $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$, we conclude that $T(n) = \Theta(n \lg n)$.

i. $T(n) = T(n-2) + 2\lg n$

We guess that $T(n) = \Theta(n \lg n)$. We show the upper bound of $T(n) = O(n \lg n)$ by means of the inductive hypothesis $T(n) \le cn \lg n$ for some constant $c > 0$. We have

$$\begin{aligned}
T(n) &= T(n-2) + 2\lg n \\
&\le c(n-2)\lg(n-2) + 2\lg n \\
&\le c(n-2)\lg n + 2\lg n \\
&= (cn - 2c + 2)\lg n
\end{aligned}$$

$$\begin{aligned} &= cn\lg n + (2-2c)\lg n \\ &\le cn\lg n \qquad \text{if } c > 1\ . \end{aligned}$$

Therefore, $T(n) = O(n\lg n)$.

For the lower bound of $T(n) = \Omega(n\lg n)$, we'll show that $T(n) \ge cn\lg n + dn$, for constants $c, d > 0$ to be chosen. We assume that $n \ge 4$, which implies that

1. $\lg(n-2) \ge \lg(n/2)$,
2. $n/2 \ge \lg n$, and
3. $n/2 \ge 2$.

(We'll use these inequalities as we go along.) We have

$$\begin{aligned} T(n) &\ge c(n-2)\lg(n-2) + d(n-2) + 2\lg n \\ &= cn\lg(n-2) - 2c\lg(n-2) + dn - 2d + 2\lg n \\ &> cn\lg(n-2) - 2c\lg n + dn - 2d + 2\lg n \\ &\qquad \text{(since } -\lg n < -\lg(n-2)\text{)} \\ &= cn\lg(n-2) - 2(c-1)\lg n + dn - 2d \\ &\ge cn\lg(n/2) - 2(c-1)\lg n + dn - 2d \quad \text{(by inequality (1) above)} \\ &= cn\lg n - cn - 2(c-1)\lg n + dn - 2d \\ &\ge cn\lg n\ , \end{aligned}$$

if $-cn - 2(c-1)\lg n + dn - 2d \ge 0$ or, equivalently, $dn \ge cn + 2(c-1)\lg n + 2d$. Pick any constant $c > 1/2$, and then pick any constant d such that

$$d \ge 2(2c-1)\ .$$

(The requirement that $c > 1/2$ means that d is positive.) Then

$$d/2 \ge 2c - 1 = c + (c-1)\ ,$$

and adding $d/2$ to both sides, we have

$$d \ge c + (c-1) + d/2\ .$$

Multiplying by n yields

$$dn \ge cn + (c-1)n + dn/2\ ,$$

and then both multiplying and dividing the middle term by 2 gives

$$dn \ge cn + 2(c-1)n/2 + dn/2\ .$$

Using inequalities (2) and (3) above, we get

$$dn \ge cn + 2(c-1)\lg n + 2d\ ,$$

which is what we needed to show. Thus $T(n) = \Omega(n\lg n)$. Since $T(n) = O(n\lg n)$ and $T(n) = \Omega(n\lg n)$, we conclude that $T(n) = \Theta(n\lg n)$.

Lecture Notes for Chapter 5: Probabilistic Analysis and Randomized Algorithms

[This chapter introduces probabilistic analysis and randomized algorithms. It assumes that the student is familiar with the basic probability material in Appendix C.

The primary goals of these notes are to

- *explain the difference between probabilistic analysis and randomized algorithms,*
- *present the technique of indicator random variables, and*
- *give another example of the analysis of a randomized algorithm (permuting an array in place).*

These notes omit the technique of permuting an array by sorting, and they omit the starred Section 5.4.]

The hiring problem

Scenario:

- You are using an employment agency to hire a new office assistant.
- The agency sends you one candidate each day.
- You interview the candidate and must immediately decide whether or not to hire that person. But if you hire, you must also fire your current office assistant—even if it's someone you have recently hired.
- Cost to interview is c_i per candidate (interview fee paid to agency).
- Cost to hire is c_h per candidate (includes cost to fire current office assistant + hiring fee paid to agency).
- Assume that $c_h > c_i$.
- You are committed to having hired, at all times, the best candidate seen so far. Meaning that whenever you interview a candidate who is better than your current office assistant, you must fire the current office assistant and hire the candidate. Since you must have someone hired at all times, you will always hire the first candidate that you interview.

Goal: Determine what the price of this strategy will be.

Pseudocode to model this scenario: Assumes that the candidates are numbered 1 to n and that after interviewing each candidate, we can determine if it's better than the current office assistant. Uses a dummy candidate 0 that is worse than all others, so that the first candidate is always hired.

```
HIRE-ASSISTANT(n)
best ← 0        ▷ candidate 0 is a least-qualified dummy candidate
for i ← 1 to n
    do interview candidate i
       if candidate i is better than candidate best
         then best ← i
              hire candidate i
```

Cost: If n candidates, and we hire m of them, the cost is $O(nc_i + mc_h)$.

- Have to pay nc_i to interview, no matter how many we hire.
- So we focus on analyzing the hiring cost mc_h.
- mc_h varies with each run—it depends on the order in which we interview the candidates.
- This is a model of a common paradigm: we need to find the maximum or minimum in a sequence by examining each element and maintaining a current "winner." The variable m denotes how many times we change our notion of which element is currently winning.

Worst-case analysis

In the worst case, we hire all n candidates.

This happens if each one is better than all who came before. In other words, if the candidates appear in increasing order of quality.

If we hire all n, then the cost is $O(nc_i + nc_h) = O(nc_h)$ (since $c_h > c_i$).

Probabilistic analysis

In general, we have no control over the order in which candidates appear.

We could assume that they come in a random order:

- Assign a rank to each candidate: $rank(i)$ is a unique integer in the range 1 to n.
- The ordered list $\langle rank(1), rank(2), \ldots, rank(n)\rangle$ is a permutation of the candidate numbers $\langle 1, 2, \ldots, n\rangle$.
- The list of ranks is equally likely to be any one of the $n!$ permutations.
- Equivalently, the ranks form a ***uniform random permutation***: each of the possible $n!$ permutations appears with equal probability.

Essential idea of probabilistic analysis: We must use knowledge of, or make assumptions about, the distribution of inputs.

- The expectation is over this distribution.
- This technique requires that we can make a reasonable characterization of the input distribution.

Randomized algorithms

We might not know the distribution of inputs, or we might not be able to model it computationally.

Instead, we use randomization within the algorithm in order to impose a distribution on the inputs.

For the hiring problem: Change the scenario:

- The employment agency sends us a list of all n candidates in advance.
- On each day, we randomly choose a candidate from the list to interview (but considering only those we have not yet interviewed).
- Instead of relying on the candidates being presented to us in a random order, we take control of the process and enforce a random order.

What makes an algorithm randomized: An algorithm is ***randomized*** if its behavior is determined in part by values produced by a ***random-number generator***.

- RANDOM(a, b) returns an integer r, where $a \le r \le b$ and each of the $b - a + 1$ possible values of r is equally likely.
- In practice, RANDOM is implemented by a ***pseudorandom-number generator***, which is a deterministic method returning numbers that "look" random and pass statistical tests.

Indicator random variables

A simple yet powerful technique for computing the expected value of a random variable.

Helpful in situations in which there may be dependence.

Given a sample space and an event A, we define the ***indicator random variable***

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur .} \end{cases}$$

Lemma
For an event A, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof Letting $\overline{A}$ be the complement of A, we have

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \quad \text{(definition of expected value)} \\ &= \Pr\{A\} \ . \end{aligned}$$

■ (lemma)

Simple example: Determine the expected number of heads when we flip a fair coin one time.

- Sample space is $\{H, T\}$.
- $\Pr\{H\} = \Pr\{T\} = 1/2$.
- Define indicator random variable $X_H = \mathrm{I}\{H\}$. X_H counts the number of heads in one flip.
- Since $\Pr\{H\} = 1/2$, lemma says that $\mathrm{E}[X_H] = 1/2$.

Slightly more complicated example: Determine the expected number of heads in n coin flips.

- Let X be a random variable for the number of heads in n flips.
- Could compute $\mathrm{E}[X] = \sum_{k=0}^{n} k \cdot \Pr\{X = k\}$. In fact, this is what the book does in equation (C.36).
- Instead, we'll use indicator random variables.
- For $i = 1, 2, \ldots, n$, define $X_i = \mathrm{I}\{\text{the } i\text{th flip results in event } H\}$.
- Then $X = \sum_{i=1}^{n} X_i$.
- Lemma says that $\mathrm{E}[X_i] = \Pr\{H\} = 1/2$ for $i = 1, 2, \ldots, n$.
- Expected number of heads is $\mathrm{E}[X] = \mathrm{E}[\sum_{i=1}^{n} X_i]$.
- ***Problem:*** We want $\mathrm{E}[\sum_{i=1}^{n} X_i]$. We have only the individual expectations $\mathrm{E}[X_1], \mathrm{E}[X_2], \ldots, \mathrm{E}[X_n]$.
- ***Solution:*** Linearity of expectation says that the expectation of the sum equals the sum of the expectations. Thus,

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 \,.
\end{aligned}$$

- Linearity of expectation applies even when there is dependence among the random variables. *[Not an issue in this example, but it can be a great help. The hat-check problem of Exercise 5.2-4 is a problem with lots of dependence. See the solution on page 5-10 of this manual.]*

Analysis of the hiring problem

Assume that the candidates arrive in a random order.

Let X be a random variable that equals the number of times we hire a new office assistant.

Define indicator random variables $X_1, X_2, \ldots, X_n$, where

$X_i = \mathrm{I}\{\text{candidate } i \text{ is hired}\}$.

Useful properties:

- $X = X_1 + X_2 + \cdots + X_n$.
- Lemma $\Rightarrow \mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\}$.

We need to compute $\Pr\{\text{candidate } i \text{ is hired}\}$.

- Candidate i is hired if and only if candidate i is better than each of candidates $1, 2, \ldots, i-1$.
- Assumption that the candidates arrive in random order $\Rightarrow$ candidates $1, 2, \ldots, i$ arrive in random order $\Rightarrow$ any one of these first i candidates is equally likely to be the best one so far.
- Thus, $\Pr\{\text{candidate } i \text{ is the best so far}\} = 1/i$.
- Which implies $\mathrm{E}[X_i] = 1/i$.

Now compute $\mathrm{E}[X]$:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/i \\
&= \ln n + O(1) \quad \text{(equation (A.7): the sum is a harmonic series)} \ .
\end{aligned}$$

Thus, the expected hiring cost is $O(c_h \ln n)$, which is much better than the worst-case cost of $O(nc_h)$.

Randomized algorithms

Instead of assuming a distribution of the inputs, we impose a distribution.

The hiring problem

For the hiring problem, the algorithm is deterministic:

- For any given input, the number of times we hire a new office assistant will always be the same.
- The number of times we hire a new office assistant depends only on the input.
- In fact, it depends only on the ordering of the candidates' ranks that it is given.
- Some rank orderings will always produce a high hiring cost. Example: $\langle 1, 2, 3, 4, 5, 6\rangle$, where each candidate is hired.
- Some will always produce a low hiring cost. Example: any ordering in which the best candidate is the first one interviewed. Then only the best candidate is hired.
- Some may be in between.

Instead of always interviewing the candidates in the order presented, what if we first randomly permuted this order?

- The randomization is now in the algorithm, not in the input distribution.
- Given a particular input, we can no longer say what its hiring cost will be. Each time we run the algorithm, we can get a different hiring cost.
- In other words, each time we run the algorithm, the execution depends on the random choices made.
- No particular input always elicits worst-case behavior.
- Bad behavior occurs only if we get "unlucky" numbers from the random-number generator.

Pseudocode for randomized hiring problem:

RANDOMIZED-HIRE-ASSISTANT(n)
randomly permute the list of candidates
HIRE-ASSISTANT(n)

Lemma
The expected hiring cost of RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.

Proof After permuting the input array, we have a situation identical to the probabilistic analysis of deterministic HIRE-ASSISTANT. ■

Randomly permuting an array

[The book considers two methods of randomly permuting an n-element array. The first method assigns a random priority in the range 1 to n^3 to each position and then reorders the array elements into increasing priority order. We omit this method from these notes. The second method is better: it works in place (unlike the priority-based method), it runs in linear time without requiring sorting, and it needs fewer random bits (n random numbers in the range 1 to n rather than the range 1 to n^3). We present and analyze the second method in these notes.]

Goal: Produce a uniform random permutation (each of the $n!$ permutations is equally likely to be produced).

Non-goal: Show that for each element $A[i]$, the probability that $A[i]$ moves to position j is $1/n$. (See Exercise 5.3-4, whose solution is on page 5-13 of this manual.)

The following procedure permutes the array $A[1 \,..\, n]$ in place (i.e., no auxiliary array is required).

RANDOMIZE-IN-PLACE(A, n)
for $i \leftarrow 1$ **to** n
    do swap $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$

Idea:

- In iteration i, choose $A[i]$ randomly from $A[i \,..\, n]$.
- Will never alter $A[i]$ after iteration i.

Time: $O(1)$ per iteration $\Rightarrow$ $O(n)$ total.

Correctness: Given a set of n elements, a ***k-permutation*** is a sequence containing k of the n elements. There are $n!/(n-k)!$ possible k-permutations.

Lemma
RANDOMIZE-IN-PLACE computes a uniform random permutation.

Proof Use a loop invariant:

> **Loop invariant:** Just prior to the ith iteration of the **for** loop, for each possible $(i-1)$-permutation, subarray $A[1 \,..\, i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

Initialization: Just before first iteration, $i = 1$. Loop invariant says that for each possible 0-permutation, subarray $A[1 \,..\, 0]$ contains this 0-permutation with probability $n!/n! = 1$. $A[1 \,..\, 0]$ is an empty subarray, and a 0-permutation has no elements. So, $A[1 \,..\, 0]$ contains any 0-permutation with probability 1.

Maintenance: Assume that just prior to the ith iteration, each possible $(i-1)$-permutation appears in $A[1 \,..\, i-1]$ with probability $(n-i+1)!/n!$. Will show that after the ith iteration, each possible i-permutation appears in $A[1 \,..\, i]$ with probability $(n-i)!/n!$. Incrementing i for the next iteration then maintains the invariant.

Consider a particular i-permutation $\pi = \langle x_1, x_2, \ldots, x_i \rangle$. It consists of an $(i-1)$-permutation $\pi' = \langle x_1, x_2, \ldots, x_{i-1} \rangle$, followed by x_i.

Let E_1 be the event that the algorithm actually puts π' into $A[1 \,..\, i-1]$. By the loop invariant, $\Pr\{E_1\} = (n-i+1)!/n!$.

Let E_2 be the event that the ith iteration puts x_i into $A[i]$.

We get the i-permutation π in $A[1 \,..\, i]$ if and only if both E_1 and E_2 occur $\Rightarrow$ the probability that the algorithm produces π in $A[1 \,..\, i]$ is $\Pr\{E_2 \cap E_1\}$.

Equation (C.14) $\Rightarrow \Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\}$.

The algorithm chooses x_i randomly from the $n-i+1$ possibilities in $A[i \,..\, n]$ $\Rightarrow \Pr\{E_2 \mid E_1\} = 1/(n-i+1)$. Thus,

$$\begin{aligned}
\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\
&= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\
&= \frac{(n-i)!}{n!} \,.
\end{aligned}$$

Termination: At termination, $i = n+1$, so we conclude that $A[1 \,..\, n]$ is a given n-permutation with probability $(n-n)!/n! = 1/n!$. ■ (lemma)

Solutions for Chapter 5: Probabilistic Analysis and Randomized Algorithms

Solution to Exercise 5.1-3

To get an unbiased random bit, given only calls to BIASED-RANDOM, call BIASED-RANDOM twice. Repeatedly do so until the two calls return different values, and when this occurs, return the first of the two bits:

```
UNBIASED-RANDOM
while TRUE
    do
        x ← BIASED-RANDOM
        y ← BIASED-RANDOM
        if x ≠ y
          then return x
```

To see that UNBIASED-RANDOM returns 0 and 1 each with probability $1/2$, observe that the probability that a given iteration returns 0 is

$$\Pr\{x = 0 \text{ and } y = 1\} = (1 - p)p \ ,$$

and the probability that a given iteration returns 1 is

$$\Pr\{x = 1 \text{ and } y = 0\} = p(1 - p) \ .$$

(We rely on the bits returned by BIASED-RANDOM being independent.) Thus, the probability that a given iteration returns 0 equals the probability that it returns 1. Since there is no other way for UNBIASED-RANDOM to return a value, it returns 0 and 1 each with probability $1/2$.

Assuming that each iteration takes $O(1)$ time, the expected running time of UNBIASED-RANDOM is linear in the expected number of iterations. We can view each iteration as a Bernoulli trial, where "success" means that the iteration returns a value. The probability of success equals the probability that 0 is returned plus the probability that 1 is returned, or $2p(1 - p)$. The number of trials until a success occurs is given by the geometric distribution, and by equation (C.31), the expected number of trials for this scenario is $1/(2p(1 - p))$. Thus, the expected running time of UNBIASED-RANDOM is $\Theta(1/(2p(1 - p)))$.

Solution to Exercise 5.2-1

Since HIRE-ASSISTANT always hires candidate 1, it hires exactly once if and only if no candidates other than candidate 1 are hired. This event occurs when candidate 1 is the best candidate of the n, which occurs with probability $1/n$.

HIRE-ASSISTANT hires n times if each candidate is better than all those who were interviewed (and hired) before. This event occurs precisely when the list of ranks given to the algorithm is $\langle 1, 2, \ldots, n \rangle$, which occurs with probability $1/n!$.

Solution to Exercise 5.2-2

We make three observations:

1. Candidate 1 is always hired.
2. The best candidate, i.e., the one whose rank is n, is always hired.
3. If the best candidate is candidate 1, then that is the only candidate hired.

Therefore, in order for HIRE-ASSISTANT to hire exactly twice, candidate 1 must have rank $i \le n-1$ and all candidates whose ranks are $i+1, i+2, \ldots, n-1$ must be interviewed after the candidate whose rank is n. (When $i = n-1$, this second condition vacuously holds.)

Let E_i be the event in which candidate 1 has rank i; clearly, $\Pr\{E_i\} = 1/n$ for any given value of i.

Letting j denote the position in the interview order of the best candidate, let F be the event in which candidates $2, 3, \ldots, j-1$ have ranks strictly less than the rank of candidate 1. Given that event E_i has occurred, event F occurs when the best candidate is the first one interviewed out of the $n-i$ candidates whose ranks are $i+1, i+2, \ldots, n$. Thus, $\Pr\{F \mid E_i\} = 1/(n-i)$.

Our final event is A, which occurs when HIRE-ASSISTANT hires exactly twice. Noting that the events $E_1, E_2, \ldots, E_n$ are disjoint, we have

$$\begin{aligned} A &= F \cap (E_1 \cup E_2 \cup \cdots \cup E_{n-1}) \\ &= (F \cap E_1) \cup (F \cap E_2) \cup \cdots \cup (F \cap E_{n-1}) \ . \end{aligned}$$

and

$$\Pr\{A\} = \sum_{i=1}^{n-1} \Pr\{F \cap E_i\} \ .$$

By equation (C.14),

$$\begin{aligned} \Pr\{F \cap E_i\} &= \Pr\{F \mid E_i\} \Pr\{E_i\} \\ &= \frac{1}{n-i} \cdot \frac{1}{n} \ , \end{aligned}$$

and so

$$\begin{aligned}
\Pr\{A\} &= \sum_{i=1}^{n-1} \frac{1}{n-i} \cdot \frac{1}{n} \\
&= \frac{1}{n} \sum_{i=1}^{n-1} \frac{1}{n-i} \\
&= \frac{1}{n} \left(\frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{1} \right) \\
&= \frac{1}{n} \cdot H_{n-1} \ ,
\end{aligned}$$

where H_{n-1} is the nth harmonic number.

Solution to Exercise 5.2-4

Another way to think of the hat-check problem is that we want to determine the expected number of fixed points in a random permutation. (A ***fixed point*** of a permutation π is a value i for which $\pi(i) = i$.) One could enumerate all $n!$ permutations, count the total number of fixed points, and divide by $n!$ to determine the average number of fixed points per permutation. This would be a painstaking process, and the answer would turn out to be 1. We can use indicator random variables, however, to arrive at the same answer much more easily.

Define a random variable X that equals the number of customers that get back their own hat, so that we want to compute $\mathrm{E}[X]$.

For $i = 1, 2, \ldots, n$, define the indicator random variable

$X_i = \mathrm{I}\{\text{customer } i \text{ gets back his own hat}\}$.

Then $X = X_1 + X_2 + \cdots + X_n$.

Since the ordering of hats is random, each customer has a probability of $1/n$ of getting back his own hat. In other words, $\Pr\{X_i = 1\} = 1/n$, which, by Lemma 5.1, implies that $\mathrm{E}[X_i] = 1/n$.

Thus,

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i \right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \qquad \text{(linearity of expectation)} \\
&= \sum_{i=1}^{n} 1/n \\
&= 1 \ ,
\end{aligned}$$

and so we expect that exactly 1 customer gets back his own hat.

Note that this is a situation in which the indicator random variables are *not* independent. For example, if $n = 2$ and $X_1 = 1$, then X_2 must also equal 1. Conversely, if $n = 2$ and $X_1 = 0$, then X_2 must also equal 0. Despite the dependence,

$\Pr\{X_i = 1\} = 1/n$ for all i, and linearity of expectation holds. Thus, we can use the technique of indicator random variables even in the presence of dependence.

Solution to Exercise 5.2-5

Let X_{ij} be an indicator random variable for the event where the pair $A[i]$, $A[j]$ for $i < j$ is inverted, i.e., $A[i] > A[j]$. More precisely, we define $X_{ij} = \mathrm{I}\{A[i] > A[j]\}$ for $1 \le i < j \le n$. We have $\Pr\{X_{ij} = 1\} = 1/2$, because given two distinct random numbers, the probability that the first is bigger than the second is $1/2$. By Lemma 5.1, $\mathrm{E}[X_{ij}] = 1/2$.

Let X be the the random variable denoting the total number of inverted pairs in the array, so that

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

We want the expected number of inverted pairs, so we take the expectation of both sides of the above equation to obtain

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \ .$$

We use linearity of expectation to get

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1/2 \\
&= \binom{n}{2}\frac{1}{2} \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{2} \\
&= \frac{n(n-1)}{4} \ .
\end{aligned}$$

Thus the expected number of inverted pairs is $n(n-1)/4$.

Solution to Exercise 5.3-1

Here's the rewritten procedure:

```
RANDOMIZE-IN-PLACE(A)
n ← length[A]
swap A[1] ↔ A[RANDOM(1, n)]
for i ← 2 to n
    do swap A[i] ↔ A[RANDOM(i, n)]
```

The loop invariant becomes

> **Loop invariant:** Just prior to the iteration of the **for** loop for each value of $i = 2, \ldots, n$, for each possible $(i-1)$-permutation, the subarray $A[1\,..\,i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

The maintenance and termination parts remain the same. The initialization part is for the subarray $A[1\,..\,1]$, which contains any 1-permutation with probability $(n-1)!/n! = 1/n$.

Solution to Exercise 5.3-2

Although PERMUTE-WITHOUT-IDENTITY will not produce the identity permutation, there are other permutations that it fails to produce. For example, consider its operation when $n = 3$, when it should be able to produce the $n! - 1 = 5$ non-identity permutations. The **for** loop iterates for $i = 1$ and $i = 2$. When $i = 1$, the call to RANDOM returns one of two possible values (either 2 or 3), and when $i = 2$, the call to RANDOM returns just one value (3). Thus, there are only $2 \cdot 1 = 2$ possible permutations that PERMUTE-WITHOUT-IDENTITY can produce, rather than the 5 that are required.

Solution to Exercise 5.3-3

The PERMUTE-WITH-ALL procedure does not produce a uniform random permutation. Consider the permutations it produces when $n = 3$. There are 3 calls to RANDOM, each of which returns one of 3 values, and so there are 27 possible outcomes of calling PERMUTE-WITH-ALL. Since there are $3! = 6$ permutations, if PERMUTE-WITH-ALL did produce a uniform random permutation, then each permutation would occur $1/6$ of the time. That would mean that each permutation would have to occur an integer number m times, where $m/27 = 1/6$. No integer m satisfies this condition.

In fact, if we were to work out the possible permutations of $\langle 1, 2, 3\rangle$ and how often they occur with PERMUTE-WITH-ALL, we would get the following probabilities:

permutation	probability
$\langle 1, 2, 3\rangle$	$4/27$
$\langle 1, 3, 2\rangle$	$5/27$
$\langle 2, 1, 3\rangle$	$5/27$
$\langle 2, 3, 1\rangle$	$5/27$
$\langle 3, 1, 2\rangle$	$4/27$
$\langle 3, 2, 1\rangle$	$4/27$

Although these probabilities add to 1, none are equal to $1/6$.

Solution to Exercise 5.3-4

PERMUTE-BY-CYCLIC chooses *offset* as a random integer in the range $1 \le \mathit{offset} \le n$, and then it performs a cyclic rotation of the array. That is, $B[((i + \mathit{offset} - 1) \bmod n) + 1] \leftarrow A[i]$ for $i = 1, 2, \ldots, n$. (The subtraction and addition of 1 in the index calculation is due to the 1-origin indexing. If we had used 0-origin indexing instead, the index calculation would have simplied to $B[(i + \mathit{offset}) \bmod n] \leftarrow A[i]$ for $i = 0, 1, \ldots, n - 1$.)

Thus, once *offset* is determined, so is the entire permutation. Since each value of *offset* occurs with probability $1/n$, each element $A[i]$ has a probability of ending up in position $B[j]$ with probability $1/n$.

This procedure does not produce a uniform random permutation, however, since it can produce only n different permutations. Thus, n permutations occur with probability $1/n$, and the remaining $n! - n$ permutations occur with probability 0.

Solution to Exercise 5.4-6

First we determine the expected number of empty bins. We define a random variable X to be the number of empty bins, so that we want to compute $\mathrm{E}[X]$. Next, for $i = 1, 2, \ldots, n$, we define the indicator random variable $Y_i = \mathrm{I}\{\text{bin } i \text{ is empty}\}$. Thus,

$$X = \sum_{i=1}^{n} Y_i \ ,$$

and so

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} Y_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[Y_i] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} \Pr\{\text{bin } i \text{ is empty}\} && \text{(by Lemma 5.1)} \ .
\end{aligned}$$

Let us focus on a specific bin, say bin i. We view a toss as a success if it misses bin i and as a failure if it lands in bin i. We have n independent Bernoulli trials, each with probability of success $1 - 1/n$. In order for bin i to be empty, we need n successes in n trials. Using a binomial distribution, therefore, we have that

$$\begin{aligned}
\Pr\{\text{bin } i \text{ is empty}\} &= \binom{n}{n}\left(1 - \frac{1}{n}\right)^n \left(\frac{1}{n}\right)^0 \\
&= \left(1 - \frac{1}{n}\right)^n \ .
\end{aligned}$$

Thus,

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{n}\left(1-\frac{1}{n}\right)^{n} \\ &= n\left(1-\frac{1}{n}\right)^{n} .\end{aligned}$$

By equation (3.13), as n approaches ∞, the quantity $(1-1/n)^n$ approaches $1/e$, and so $\mathrm{E}[X]$ approaches n/e.

Now we determine the expected number of bins with exactly one ball. We redefine X to be number of bins with exactly one ball, and we redefine Y_i to be $\mathrm{I}\{\text{bin } i \text{ gets exactly one ball}\}$. As before, we find that

$$\mathrm{E}[X] = \sum_{i=1}^{n}\Pr\{\text{bin } i \text{ gets exactly one ball}\} .$$

Again focusing on bin i, we need exactly $n-1$ successes in n independent Bernoulli trials, and so

$$\begin{aligned}\Pr\{\text{bin } i \text{ gets exactly one ball}\} &= \binom{n}{n-1}\left(1-\frac{1}{n}\right)^{n-1}\left(\frac{1}{n}\right)^{1} \\ &= n\cdot\left(1-\frac{1}{n}\right)^{n-1}\frac{1}{n} \\ &= \left(1-\frac{1}{n}\right)^{n-1} ,\end{aligned}$$

and so

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{n}\left(1-\frac{1}{n}\right)^{n-1} \\ &= n\left(1-\frac{1}{n}\right)^{n-1} .\end{aligned}$$

Because

$$n\left(1-\frac{1}{n}\right)^{n-1} = \frac{n\,(1-\frac{1}{n})^{n}}{1-\frac{1}{n}} ,$$

as n approaches ∞, we find that $\mathrm{E}[X]$ approaches

$$\frac{n/e}{1-1/n} = \frac{n^2}{e(n-1)} .$$

Solution to Problem 5-1

a. To determine the expected value represented by the counter after n INCREMENT operations, we define some random variables:

- For $j = 1, 2, \ldots, n$, let X_j denote the increase in the value represented by the counter due to the jth INCREMENT operation.
- Let V_n be the value represented by the counter after n INCREMENT operations.

Then $V_n = X_1 + X_2 + \cdots + X_n$. We want to compute $\mathrm{E}[V_n]$. By linearity of expectation,

$$\mathrm{E}[V_n] = \mathrm{E}[X_1 + X_2 + \cdots + X_n] = \mathrm{E}[X_1] + \mathrm{E}[X_2] + \cdots + \mathrm{E}[X_n] \ .$$

We shall show that $\mathrm{E}[X_j] = 1$ for $j = 1, 2, \ldots, n$, which will prove that $\mathrm{E}[V_n] = n$.

We actually show that $\mathrm{E}[X_j] = 1$ in two ways, the second more rigorous than the first:

1. Suppose that at the start of the jth INCREMENT operation, the counter holds the value i, which represents n_i. If the counter increases due to this INCREMENT operation, then the value it represents increases by $n_{i+1} - n_i$. The counter increases with probability $1/(n_{i+1} - n_i)$, and so

$$\begin{aligned}
\mathrm{E}[X_j] &= (0 \cdot \Pr\{\text{counter does not increase}\}) \\
&\qquad + ((n_{i+1} - n_i) \cdot \Pr\{\text{counter increases}\}) \\
&= \left(0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i}\right)\right) + \left((n_{i+1} - n_i) \cdot \frac{1}{n_{i+1} - n_i}\right) \\
&= 1 \ ,
\end{aligned}$$

   and so $\mathrm{E}[X_j] = 1$ regardless of the value held by the counter.

2. Let C_j be the random variable denoting the value held in the counter at the start of the jth INCREMENT operation. Since we can ignore values of C_j greater than $2^b - 1$, we use a formula for conditional expectation:

$$\begin{aligned}
\mathrm{E}[X_j] &= \mathrm{E}[\mathrm{E}[X_j \mid C_j]] \\
&= \sum_{i=0}^{2^b-1} \mathrm{E}[X_j \mid C_j = i] \cdot \Pr\{C_j = i\} \ .
\end{aligned}$$

   To compute $\mathrm{E}[X_j \mid C_j = i]$, we note that

   - $\Pr\{X_j = 0 \mid C_j = i\} = 1 - 1/(n_{i+1} - n_i)$,
   - $\Pr\{X_j = n_{i+1} - n_i \mid C_j = i\} = 1/(n_{i+1} - n_i)$, and
   - $\Pr\{X_j = k \mid C_j = i\} = 0$ for all other k.

   Thus,

$$\begin{aligned}
\mathrm{E}[X_j \mid C_j = i] &= \sum_k k \cdot \Pr\{X_j = k \mid C_j = i\} \\
&= \left(0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i}\right)\right) + \left((n_{i+1} - n_i) \cdot \frac{1}{n_{i+1} - n_i}\right) \\
&= 1 \ .
\end{aligned}$$

   Therefore, noting that

$$\sum_{i=0}^{2^b-1} \Pr\{C_j = i\} = 1 \ ,$$

   we have

$$\begin{aligned}
\mathrm{E}[X_j] &= \sum_{i=0}^{2^b-1} 1 \cdot \Pr\{C_j = i\} \\
&= 1 \ .
\end{aligned}$$

Why is the second way more rigorous than the first? Both ways condition on the value held in the counter, but only the second way incorporates the conditioning into the expression for $\mathrm{E}[X_j]$.

b. Defining V_n and X_j as in part (a), we want to compute $\mathrm{Var}[V_n]$, where $n_i = 100i$. The X_j are pairwise independent, and so by equation (C.28), $\mathrm{Var}[V_n] = \mathrm{Var}[X_1] + \mathrm{Var}[X_2] + \cdots + \mathrm{Var}[X_n]$.

Since $n_i = 100i$, we see that $n_{i+1} - n_i = 100(i+1) - 100i = 100$. Therefore, with probability $99/100$, the increase in the value represented by the counter due to the jth INCREMENT operation is 0, and with probability $1/100$, the value represented increases by 100. Thus, by equation (C.26),

$$\begin{aligned}
\mathrm{Var}[X_j] &= \mathrm{E}\left[X_j^2\right] - \mathrm{E}^2[X_j] \\
&= \left(\left(0^2 \cdot \frac{99}{100}\right) + \left(100^2 \cdot \frac{1}{100}\right)\right) - 1^2 \\
&= 100 - 1 \\
&= 99 \, .
\end{aligned}$$

Summing up the variances of the X_j gives $\mathrm{Var}[V_n] = 99n$.

Lecture Notes for Chapter 6: Heapsort

Chapter 6 overview

Heapsort

- $O(n \lg n)$ worst case—like merge sort.
- Sorts in place—like insertion sort.
- Combines the best of both algorithms.

To understand heapsort, we'll cover heaps and heap operations, and then we'll take a look at priority queues.

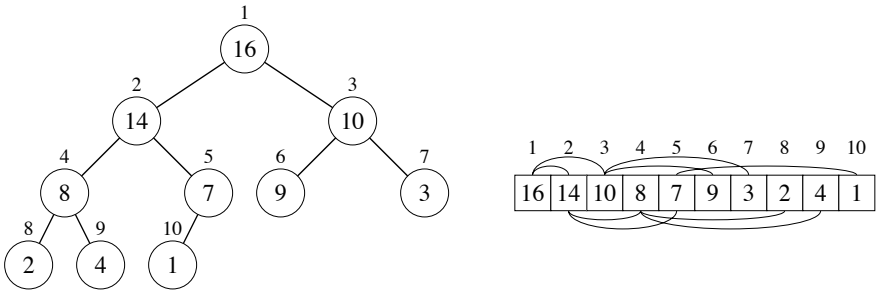
Heaps

Heap data structure

- Heap A (*not* garbage-collected storage) is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array A.
  - Root of tree is $A[1]$.
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
  - Left child of $A[i] = A[2i]$.
  - Right child of $A[i] = A[2i + 1]$.
  - Computing is fast with binary representation implementation.

[In book, have length and heap-size attributes. Here, we bypass these attributes and use parameter values instead.]

Example: of a max-heap. *[Arcs above and below the array on the right go between parents and children. There is no significance to whether an arc is drawn above or below the array.]*



Heap property

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

By induction and transitivity of $\leq$, the max-heap property guarantees that the maximum element of a max-heap is at the root. Similar argument for min-heaps.

The heapsort algorithm we'll show uses max-heaps.

Note: In general, heaps can be k-ary tree instead of binary.

Maintaining the heap property

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

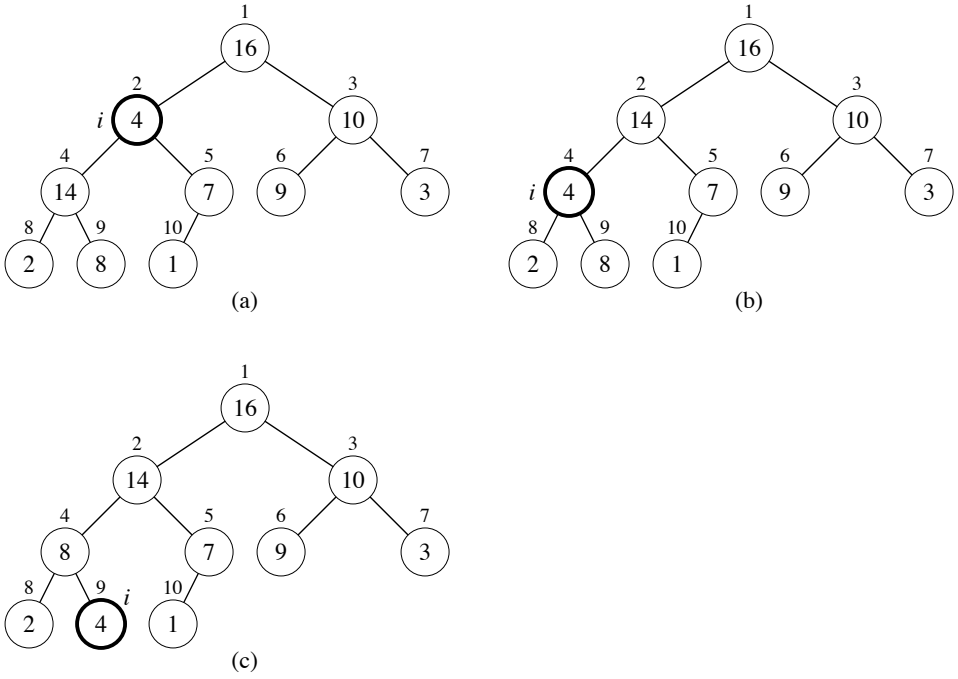
```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
  then largest ← l
  else largest ← i
if r ≤ n and A[r] > A[largest]
  then largest ← r
if largest ≠ i
  then exchange A[i] ↔ A[largest]
       MAX-HEAPIFY(A, largest, n)
```

[Parameter n replaces attribute $heap\text{-}size[A]$.]

The way MAX-HEAPIFY works:

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

Run MAX-HEAPIFY on the following heap example.



- Node 2 violates the max-heap property.
- Compare node 2 with its children, and then swap it with the larger of the two children.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Time: $O(\lg n)$.

Correctness: *[Instead of book's formal analysis with recurrence, just come up with $O(\lg n)$ intuitively.]* Heap is almost-complete binary tree, hence must process $O(\lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

Building a heap

The following procedure, given an unordered array, will produce a max-heap.

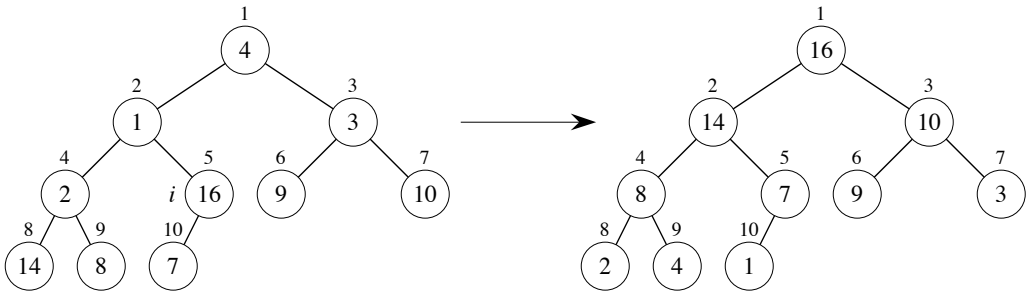
BUILD-MAX-HEAP(A, n)
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i, n)

[Parameter n replaces both attributes $length[A]$ and $heap\text{-}size[A]$.]

Example: Building a max-heap from the following unsorted array results in the first heap example.

- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



Correctness

Loop invariant: At start of every iteration of **for** loop, each node $i+1$, $i+2, \ldots, n$ is root of a max-heap.

Initialization: By Exercise 6.1-7, we know that each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, $\ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analysis

- ***Simple bound:*** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$. (Note: A good approach to analysis in general is to start by proving easy bound, then try to tighten it.)
- ***Tighter analysis:*** Observation: Time to run MAX-HEAPIFY is linear in the height of the node it's run on, and most nodes have small heights. Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 6.3-3), and height of heap is $\lfloor \lg n \rfloor$ (Exercise 6.1-2).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

Evaluate the last summation by substituting $x = 1/2$ in the formula (A.8) ($\sum_{k=0}^{\infty} kx^k$), which yields

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.

The heapsort algorithm

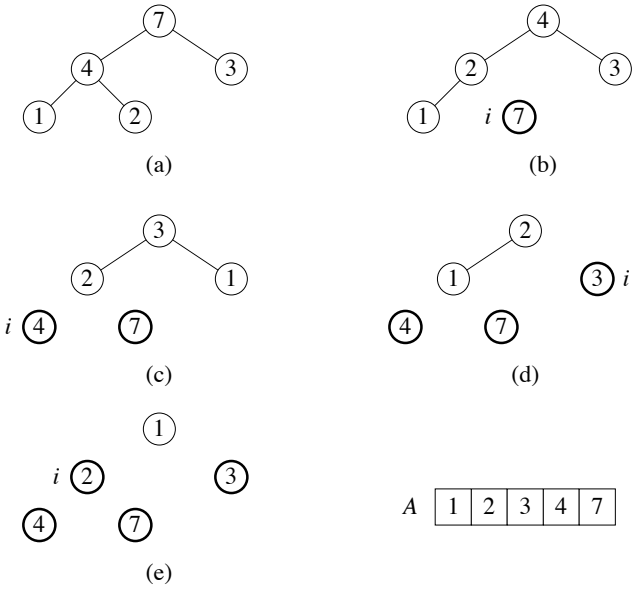
Given an input array, the heapsort algorithm acts as follows:

- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for $i \leftarrow n$ **downto** 2
  do exchange $A[1] \leftrightarrow A[i]$
    MAX-HEAPIFY(A, 1, $i - 1$)

[Parameter n replaces $length[A]$, and parameter value $i - 1$ in MAX-HEAPIFY *call replaces decrementing of heap-size$[A]$.]*

Example: Sort an example heap on the board. *[Nodes with heavy outline are no longer in the heap.]*



Analysis

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

Though heapsort is a great algorithm, a well-implemented quicksort usually beats it in practice.

Heap implementation of priority queue

Heaps efficiently implement priority queues. These notes will deal with max-priority queues implemented with max-heaps. Min-priority queues are implemented with min-heaps similarly.

A heap gives a good compromise between fast insertion but slow extraction and vice versa. Both operations take $O(\lg n)$ time.

Priority queue

- Maintains a dynamic set S of elements.
- Each set element has a ***key***—an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): inserts element x into set S.
  - MAXIMUM(S): returns element of S with largest key.

  - EXTRACT-MAX(S): removes and returns element of S with largest key.
  - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume $k \geq x$'s current key value.
- Example max-priority queue application: schedule jobs on shared computer.
- Min-priority queue supports similar operations:
  - INSERT(S, x): inserts element x into set S.
  - MINIMUM(S): returns element of S with smallest key.
  - EXTRACT-MIN(S): removes and returns element of S with smallest key.
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k. Assume $k \leq x$'s current key value.
- Example min-priority queue application: event-driven simulator.

Note: Actual implementations often have a ***handle*** in each heap element that allows access to an object in the application, and objects in the application often have a handle (likely an array index) to access the heap element.

Will examine how to implement max-priority queue operations.

Finding the maximum element

Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)
return $A[1]$

Time: $\Theta(1)$.

Extracting max element

Given the array A:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)
if $n < 1$
  then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
MAX-HEAPIFY($A, 1, n - 1$) $\quad \triangleright$ remakes heap
return *max*

[Parameter n replaces heap-size$[A]$, and parameter value $n - 1$ in MAX-HEAPIFY *call replaces decrementing of heap-size$[A]$.]*

Analysis: constant time assignments plus time for MAX-HEAPIFY.

Time: $O(\lg n)$.

Example: Run HEAP-EXTRACT-MAX on first heap example.

- Take 16 out of node 1.
- Move 1 from node 10 to node 1.
- Erase node 10.
- MAX-HEAPIFY from the root to preserve max-heap property.
- Note that successive extractions will remove items in reverse sorted order.

Increasing key value

Given set S, element x, and new key value k:

- Make sure $k \geq x$'s current key.
- Update x's key value to k.
- Traverse the tree upward comparing x to its parent and swapping keys if necessary, until x's key is smaller than its parent's key.

HEAP-INCREASE-KEY(A, i, key)
if $key < A[i]$
  then error "new key is smaller than current key"
$A[i] \leftarrow key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$

Analysis: Upward path from node i has length $O(\lg n)$ in an n-element heap.

Time: $O(\lg n)$.

Example: Increase key of node 9 in first heap example to have value 15. Exchange keys of nodes 4 and 9, then of nodes 2 and 4.

Inserting into the heap

Given a key k to insert into the heap:

- Insert a new node in the very last position in the tree with key $-\infty$.
- Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure defined above.

MAX-HEAP-INSERT(A, key, n)
$A[n+1] \leftarrow -\infty$
HEAP-INCREASE-KEY$(A, n+1, key)$

[Parameter n replaces $heap\text{-}size[A]$, and use of value $n+1$ replaces incrementing of $heap\text{-}size[A]$.]

Analysis: constant time assignments + time for HEAP-INCREASE-KEY.

Time: $O(\lg n)$.

Min-priority queue operations are implemented similarly with min-heaps.

Solutions for Chapter 6: Heapsort

Solution to Exercise 6.1-1

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Solution to Exercise 6.1-2

Given an n-element heap of height h, we know from Exercise 6.1-1 that

$2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$.

Thus, $h \le \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor$ (by definition of $\lfloor\ \rfloor$).

Solution to Exercise 6.1-3

Assume the claim is false—i.e., that there is a subtree whose root is not the largest element in the subtree. Then the maximum element is somewhere else in the subtree, possibly even at more than one location. Let m be the index at which the maximum appears (the lowest such index if the maximum appears more than once). Since the maximum is not at the root of the subtree, node m has a parent. Since the parent of a node has a lower index than the node, and m was chosen to be the smallest index of the maximum value, $A[\text{PARENT}(m)] < A[m]$. But by the max-heap property, we must have $A[\text{PARENT}(m)] \ge A[m]$. So our assumption is false, and the claim is true.

Solution to Exercise 6.2-6

If you put a value at the root that is less than every value in the left and right subtrees, then MAX-HEAPIFY will be called recursively until a leaf is reached. To

make the recursive calls traverse the longest path to a leaf, choose values that make MAX-HEAPIFY always recurse on the left child. It follows the left branch when the left child is $\geq$ the right child, so putting 0 at the root and 1 at all the other nodes, for example, will accomplish that. With such values, MAX-HEAPIFY will be called h times (where h is the heap height, which is the number of edges in the longest path from the root to a leaf), so its running time will be $\Theta(h)$ (since each call does $\Theta(1)$ work), which is $\Theta(\lg n)$. Since we have a case in which MAX-HEAPIFY's running time is $\Theta(\lg n)$, its worst-case running time is $\Omega(\lg n)$.

Solution to Exercise 6.3-3

Let H be the height of the heap.

Two subtleties to beware of:

- Be careful not to confuse the height of a node (longest distance from a leaf) with its depth (distance from the root).
- If the heap is not a complete binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same height. For example, although all nodes at depth H have height 0, nodes at depth $H - 1$ can have either height 0 or height 1.

For a complete binary tree, it's easy to show that there are $\lceil n/2^{h+1} \rceil$ nodes of height h. But the proof for an incomplete tree is tricky and is not derived from the proof for a complete tree.

Proof By induction on h.

Basis: Show that it's true for $h = 0$ (i.e., that # of leaves $\leq \lceil n/2^{h+1} \rceil = \lceil n/2 \rceil$). In fact, we'll show that the # of leaves $= \lceil n/2 \rceil$.

The tree leaves (nodes at height 0) are at depths H and $H - 1$. They consist of

- all nodes at depth H, and
- the nodes at depth $H - 1$ that are not parents of depth-H nodes.

Let x be the number of nodes at depth H—that is, the number of nodes in the bottom (possibly incomplete) level.

Note that $n - x$ is odd, because the $n - x$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, and if n is even, x is odd.

To prove the base case, we must consider separately the case in which n is even (x is odd) and the case in which n is odd (x is even). Here are two ways to do this: The first requires more cleverness, and the second requires more algebraic manipulation.

1. First method of proving the base case:
   - If n is odd, then x is even, so all nodes have siblings—i.e., all internal nodes have 2 children. Thus (see Exercise B.5-3), # of internal nodes $=$ # of leaves $- 1$.

So, n = # of nodes = # of leaves + # of internal nodes = $2 \cdot$ # of leaves $- 1$. Thus, # of leaves $= (n+1)/2 = \lceil n/2 \rceil$. (The latter equality holds because n is odd.)

- If n is even, then x is odd, and some leaf doesn't have a sibling. If we gave it a sibling, we would have $n+1$ nodes, where $n+1$ is odd, so the case we analyzed above would apply. Observe that we would also increase the number of leaves by 1, since we added a node to a parent that already had a child. By the odd-node case above, # of leaves $+ 1 = \lceil (n+1)/2 \rceil = \lceil n/2 \rceil + 1$. (The latter equality holds because n is even.)

In either case, # of leaves $= \lceil n/2 \rceil$.

2. Second method of proving the base case:

   Note that at any depth $d < H$ there are 2^d nodes, because all such tree levels are complete.

   - If x is even, there are $x/2$ nodes at depth $H-1$ that are parents of depth H nodes, hence $2^{H-1} - x/2$ nodes at depth $H-1$ that are not parents of depth-H nodes. Thus,

     $$\begin{aligned} \text{total \# of height-0 nodes} &= x + 2^{H-1} - x/2 \\ &= 2^{H-1} + x/2 \\ &= (2^H + x)/2 \\ &= \lceil (2^H + x - 1)/2 \rceil \qquad \text{(because } x \text{ is even)} \\ &= \lceil n/2 \rceil \ . \end{aligned}$$

     ($n = 2^H + x - 1$ because the complete tree down to depth $H-1$ has $2^H - 1$ nodes and depth H has x nodes.)

   - If x is odd, by an argument similar to the even case, we see that

     $$\begin{aligned} \text{\# of height-0 nodes} &= x + 2^{H-1} - (x+1)/2 \\ &= 2^{H-1} + (x-1)/2 \\ &= (2^H + x - 1)/2 \\ &= n/2 \\ &= \lceil n/2 \rceil \qquad \text{(because } x \text{ odd} \Rightarrow n \text{ even)} \ . \end{aligned}$$

Inductive step: Show that if it's true for height $h-1$, it's true for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - n_0 = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$.

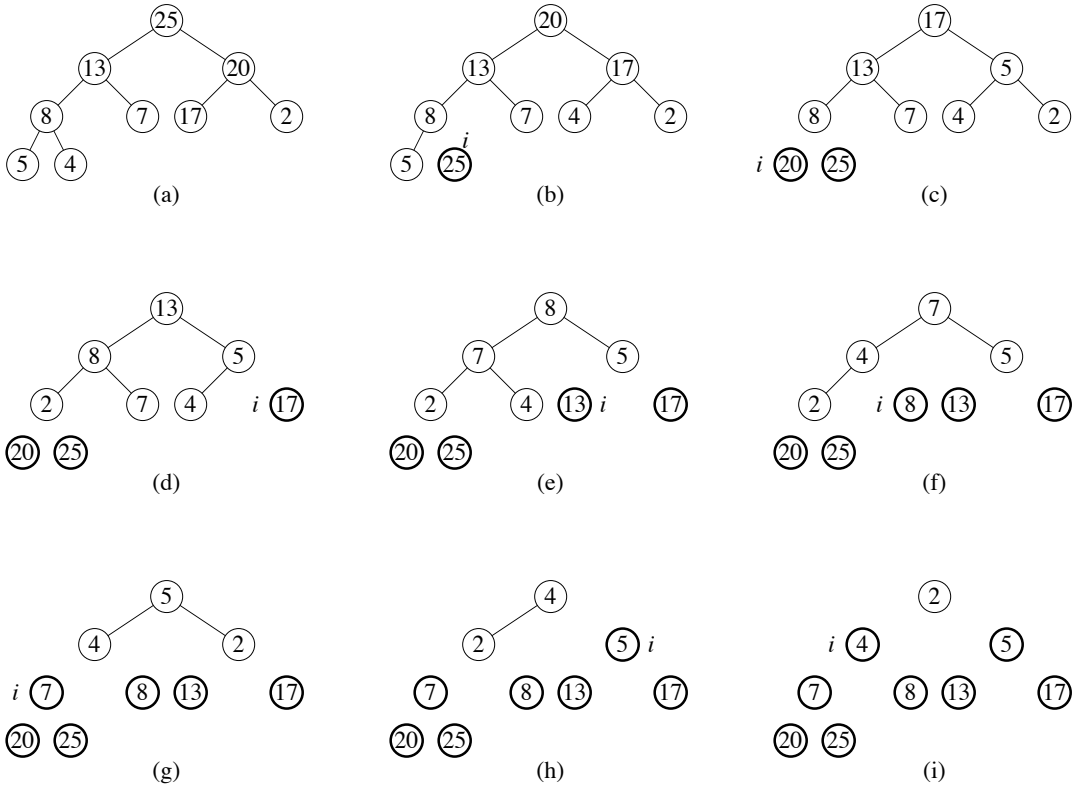
Note that the nodes at height h in T would be at height $h-1$ if the leaves of the tree were removed—that is, they are at height $h-1$ in T'. Letting n'_{h-1} denote the number of nodes at height $h-1$ in T', we have

$$n_h = n'_{h-1} \ .$$

By induction, we can bound n'_{h-1}:

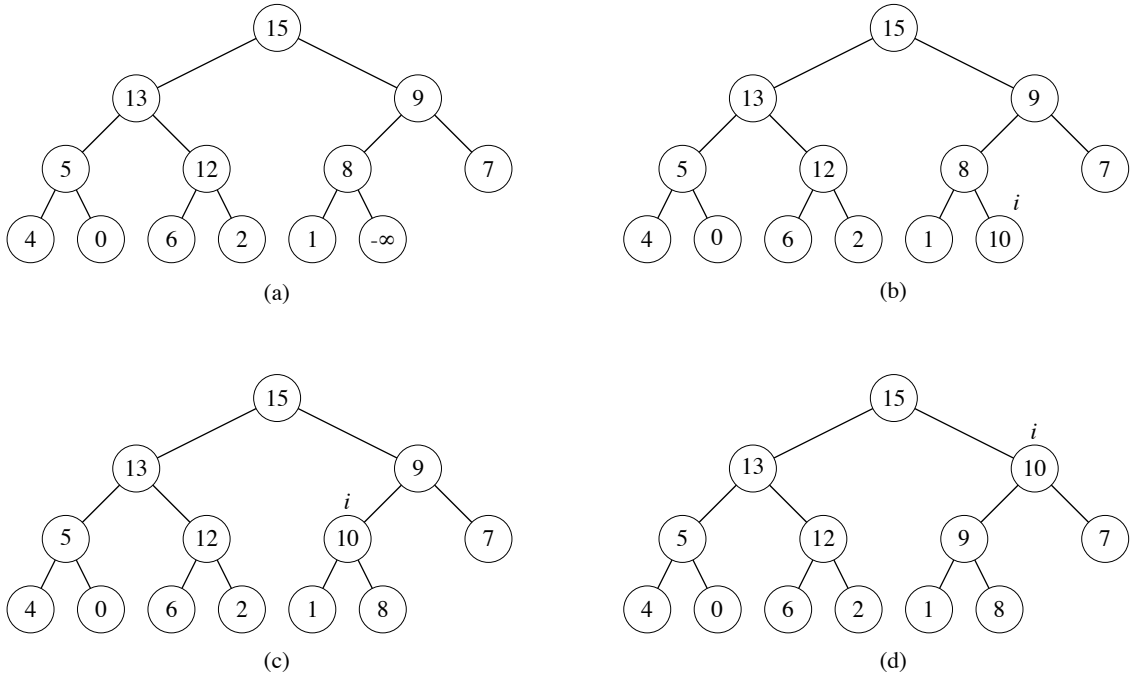
$$n_h = n'_{h-1} \le \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil \ . \qquad \blacksquare$$

Solution to Exercise 6.4-1



A	2	4	5	7	8	13	17	20	25

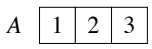
Solution to Exercise 6.5-2



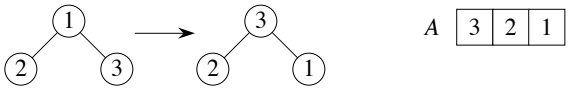
Solution to Problem 6-1

a. The procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ do not always create the same heap when run on the same input array. Consider the following counterexample.

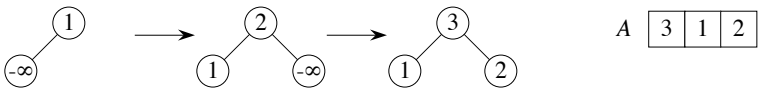
Input array A:



BUILD-MAX-HEAP(A):



BUILD-MAX-HEAP′(A):



b. An upper bound of $O(n \lg n)$ time follows immediately from there being $n - 1$ calls to MAX-HEAP-INSERT, each taking $O(\lg n)$ time. For a lower bound of

$\Omega(n \lg n)$, consider the case in which the input array is given in strictly increasing order. Each call to MAX-HEAP-INSERT causes HEAP-INCREASE-KEY to go all the way up to the root. Since the depth of node i is $\lfloor \lg i \rfloor$, the total time is

$$\begin{aligned}
\sum_{i=1}^{n} \Theta(\lfloor \lg i \rfloor) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg(n/2) \rfloor) \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg n - 1 \rfloor) \\
&\geq n/2 \cdot \Theta(\lg n) \\
&= \Omega(n \lg n) \; .
\end{aligned}$$

In the worst case, therefore, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

Solution to Problem 6-2

a. A d-ary heap can be represented in a 1-dimensional array as follows. The root is kept in $A[1]$, its d children are kept in order in $A[2]$ through $A[d+1]$, their children are kept in order in $A[d+2]$ through $A[d^2+d+1]$, and so on. The following two procedures map a node with index i to its parent and to its jth child (for $1 \leq j \leq d$), respectively.

```
D-ARY-PARENT(i)
return ⌊(i − 2)/d + 1⌋

D-ARY-CHILD(i, j)
return d(i − 1) + j + 1
```

To convince yourself that these procedures really work, verify that

$$\text{D-ARY-PARENT}(\text{D-ARY-CHILD}(i, j)) = i \; ,$$

for any $1 \leq j \leq d$. Notice that the binary heap procedures are a special case of the above procedures when $d = 2$.

b. Since each node has d children, the height of a d-ary heap with n nodes is $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

c. The procedure HEAP-EXTRACT-MAX given in the text for binary heaps works fine for d-ary heaps too. The change needed to support d-ary heaps is in MAX-HEAPIFY, which must compare the argument node to all d children instead of just 2 children. The running time of HEAP-EXTRACT-MAX is still the running time for MAX-HEAPIFY, but that now takes worst-case time proportional to the product of the height of the heap by the number of children examined at each node (at most d), namely $\Theta(d \log_d n) = \Theta(d \lg n / \lg d)$.

d. The procedure MAX-HEAP-INSERT given in the text for binary heaps works fine for d-ary heaps too. The worst-case running time is still $\Theta(h)$, where h is the height of the heap. (Since only parent pointers are followed, the number of children a node has is irrelevant.) For a d-ary heap, this is $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

e. D-ARY-HEAP-INCREASE-KEY can be implemented as a slight modification of MAX-HEAP-INSERT (only the first couple lines are different). Increasing an element may make it larger than its parent, in which case it must be moved higher up in the tree. This can be done just as for insertion, traversing a path from the increased node toward the root. In the worst case, the entire height of the tree must be traversed, so the worst-case running time is $\Theta(h) = \Theta(\log_d n) = \Theta(\lg n / \lg d)$.

```
D-ARY-HEAP-INCREASE-KEY(A, i, k)
A[i] ← max(A[i], k)
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
       i ← PARENT(i)
```

Lecture Notes for Chapter 7: Quicksort

Chapter 7 overview

[The treatment in the second edition differs from that of the first edition. We use a different partitioning method—known as "Lomuto partitioning"—in the second edition, rather than the "Hoare partitioning" used in the first edition. Using Lomuto partitioning helps simplify the analysis, which uses indicator random variables in the second edition.]

Quicksort

- Worst-case running time: $\Theta(n^2)$.
- Expected running time: $\Theta(n \lg n)$.
- Constants hidden in $\Theta(n \lg n)$ are small.
- Sorts in place.

Description of quicksort

Quicksort is based on the three-step process of divide-and-conquer.

- To sort the subarray $A[p \,..\, r]$:

  Divide: Partition $A[p \,..\, r]$, into two (possibly empty) subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$, such that each element in the first subarray $A[p \,..\, q-1]$ is $\le A[q]$ and $A[q]$ is $\le$ each element in the second subarray $A[q+1 \,..\, r]$.

  Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

  Combine: No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

```
QUICKSORT(A, p, r)
if p < r
  then q ← PARTITION(A, p, r)
       QUICKSORT(A, p, q − 1)
       QUICKSORT(A, q + 1, r)
```

Initial call is QUICKSORT$(A, 1, n)$.

Partitioning

Partition subarray $A[p\,..\,r]$ by the following procedure:

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
    do if A[j] ≤ x
          then i ← i + 1
               exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

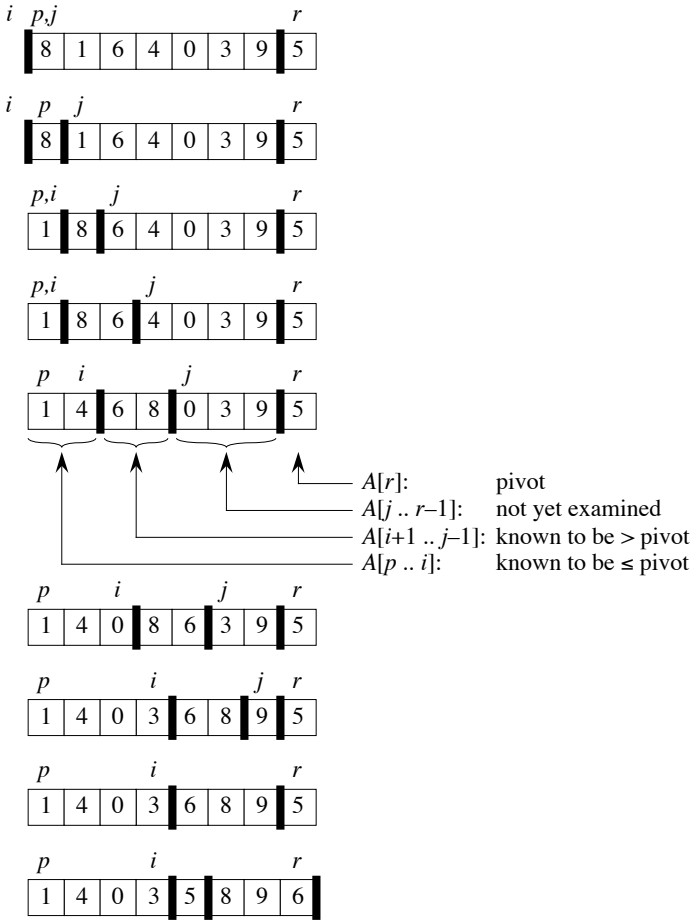
- PARTITION always selects the last element $A[r]$ in the subarray $A[p\,..\,r]$ as the ***pivot***—the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:

  Loop invariant:

  1. All entries in $A[p\,..\,i]$ are $\le$ pivot.
  2. All entries in $A[i + 1\,..\,j - 1]$ are $>$ pivot.
  3. $A[r] =$ pivot.

  It's not needed as part of the loop invariant, but the fourth region is $A[j\,..\,r-1]$, whose entries have not yet been examined, and so we don't know how they compare to the pivot.

Example: On an 8-element subarray.



[The index j disappears because it is no longer needed once the **for** *loop is exited.]*

Correctness: Use the loop invariant to prove correctness of PARTITION:

Initialization: Before the loop starts, all the conditions of the loop invariant are satisfied, because r is the pivot and the subarrays $A[p\,..\,i]$ and $A[i+1\,..\,j-1]$ are empty.

Maintenance: While the loop is running, if $A[j] \le$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot, then increment only j.

Termination: When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases: $A[p\,..\,i] \le$ pivot, $A[i+1\,..\,r-1] >$ pivot, and $A[r] =$ pivot.

The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays. This is done by swapping the pivot and the first element of the second subarray, i.e., by swapping $A[i+1]$ and $A[r]$.

Time for partitioning: $\Theta(n)$ to partition an n-element subarray.

Performance of quicksort

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and $n-1$ elements in the other subarray.
- Get the recurrence
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \ . \end{aligned}$$
- Same running time as insertion sort.
- In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Best case

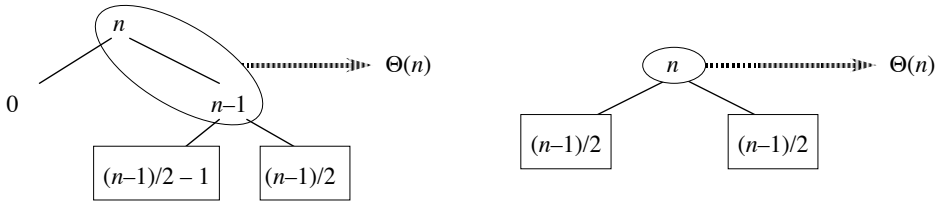
- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\leq n/2$ elements.
- Get the recurrence
$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \ . \end{aligned}$$

Balanced partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence
$$\begin{aligned} T(n) &\leq T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n) \ . \end{aligned}$$
- Intuition: look at the recursion tree.
  - It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$ in Section 4.2.
  - Except that here the constants are different; we get $\log_{10} n$ full levels and $\log_{10/9} n$ levels that are nonempty.
  - As long as it's a constant, the base of the log doesn't matter in asymptotic notation.
  - Any split of constant proportionality will yield a recursion tree of depth $\Theta(\lg n)$.

Intuition for the average case

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of good and bad splits throughout the recursion tree.
- To see that this doesn't affect the asymptotic running time of quicksort, assume that levels alternate between best-case and worst-case splits.



- The extra level in the left-hand figure only adds to the constant hidden in the Θ-notation.
- There are still the same number of subarrays to sort, and only twice as much work was done to get to that point.
- Both figures result in $O(n \lg n)$ time, though the constant for the figure on the left is higher than that of the figure on the right.

Randomized version of quicksort

- We have assumed that all input permutations are equally likely.
- This is not always true.
- To correct this, we add randomization to quicksort.
- We could randomly permute the input array.
- Instead, we use ***random sampling***, or picking one element at random.
- Don't always use $A[r]$ as the pivot. Instead, randomly pick an element from the subarray that is being sorted.

We add this randomization by not always using $A[r]$ as the pivot, but instead randomly picking an element from the subarray that is being sorted.

RANDOMIZED-PARTITION(A, p, r)
$i \leftarrow$ RANDOM(p, r)
exchange $A[r] \leftrightarrow A[i]$
return PARTITION(A, p, r)

Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

RANDOMIZED-QUICKSORT(A, p, r)
if $p < r$
  then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT$(A, p, q-1)$
    RANDOMIZED-QUICKSORT$(A, q+1, r)$

Randomization of quicksort stops any specific type of array from causing worst-case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized QUICKSORT, but not in RANDOMIZED-QUICKSORT.

Analysis of quicksort

We will analyze

- the worst-case running time of QUICKSORT and RANDOMIZED-QUICKSORT (the same), and
- the expected (average-case) running time of RANDOMIZED-QUICKSORT.

Worst-case analysis

We will prove that a worst-case split at every level produces a worst-case running time of $O(n^2)$.

- Recurrence for the worst-case running time of QUICKSORT:
  $T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$.
- Because PARTITION produces two subproblems, totaling size $n-1$, q ranges from 0 to $n-1$.
- ***Guess:*** $T(n) \le cn^2$, for some c.
- Substituting our guess into the above recurrence:
  $$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) \ . \end{aligned}$$
- The maximum value of $(q^2 + (n-q-1)^2)$ occurs when q is either 0 or $n-1$. (Second derivative with respect to q is positive.) This means that
  $$\begin{aligned} \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) &\le (n-1)^2 \\ &= n^2 - 2n + 1 \ . \end{aligned}$$
- Therefore,
  $$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 \qquad \text{if } c(2n-1) \ge \Theta(n) \ . \end{aligned}$$
- Pick c so that $c(2n-1)$ dominates $\Theta(n)$.
- Therefore, the worst-case running time of quicksort is $O(n^2)$.
- Can also show that the recurrence's solution is $\Omega(n^2)$. Thus, the worst-case running time is $\Theta(n^2)$.

Average-case analysis

- The dominant cost of the algorithm is partitioning.
- PARTITION removes the pivot element from future consideration each time.
- Thus, PARTITION is called at most n times.
- QUICKSORT recurses on the partitions.
- The amount of work that each call to PARTITION does is a constant plus the number of comparisons that are performed in its **for** loop.
- Let $X =$ the total number of comparisons performed in all calls to PARTITION.
- Therefore, the total work done over the entire execution is $O(n + X)$.

We will now compute a bound on the overall number of comparisons.

For ease of analysis:

- Rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element.
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ to be the set of elements between z_i and z_j, inclusive.

Each pair of elements is compared at most once, because elements are compared only to the pivot element, and then the pivot element is never in any later call to PARTITION.

Let $X_{ij} = \mathrm{I}\{z_i \text{ is compared to } z_j\}$.
(Considering whether z_i is compared to z_j at any time during the entire quicksort algorithm, not just during one call of PARTITION.)

Since each pair is compared at most once, the total number of comparisons performed by the algorithm is

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

Take expectations of both sides, use Lemma 5.1 and linearity of expectation:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \ . \end{aligned}$$

Now all we have to do is find the probability that two elements are compared.

- Think about when two elements are *not* compared.
  - For example, numbers in separate partitions will not be compared.
  - In the previous example, $\langle 8, 1, 6, 4, 0, 3, 9, 5\rangle$ and the pivot is 5, so that none of the set $\{1, 4, 0, 3\}$ will ever be compared to any of the set $\{8, 6, 9\}$.

- Once a pivot x is chosen such that $z_i < x < z_j$, then z_i and z_j will never be compared at any later time.
- If either z_i or z_j is chosen before any other element of Z_{ij}, then it will be compared to all the elements of Z_{ij}, except itself.
- The probability that z_i is compared to z_j is the probability that either z_i or z_j is the first element chosen.
- There are $j-i+1$ elements, and pivots are chosen randomly and independently. Thus, the probability that any particular one of them is the first one chosen is $1/(j-i+1)$.

Therefore,

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} .
\end{aligned}$$

[The second line follows because the two events are mutually exclusive.]

Substituting into the equation for $\mathrm{E}[X]$:

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

Evaluate by using a change in variables ($k = j-i$) and the bound on the harmonic series in equation (A.7):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) .
\end{aligned}$$

So the expected running time of quicksort, using RANDOMIZED-PARTITION, is $O(n \lg n)$.

Solutions for Chapter 7: Quicksort

Solution to Exercise 7.2-3

PARTITION does a "worst-case partitioning" when the elements are in decreasing order. It reduces the size of the subarray under consideration by only 1 at each step, which we've seen has running time $\Theta(n^2)$.

In particular, PARTITION, given a subarray $A[p\,..\,r]$ of distinct elements in decreasing order, produces an empty partition in $A[p\,..\,q-1]$, puts the pivot (originally in $A[r]$) into $A[p]$, and produces a partition $A[p+1\,..\,r]$ with only one fewer element than $A[p\,..\,r]$. The recurrence for QUICKSORT becomes $T(n) = T(n-1) + \Theta(n)$, which has the solution $T(n) = \Theta(n^2)$.

Solution to Exercise 7.2-5

The minimum depth follows a path that always takes the smaller part of the partition—i.e., that multiplies the number of elements by α. One iteration reduces the number of elements from n to αn, and i iterations reduces the number of elements to $\alpha^i n$. At a leaf, there is just one remaining element, and so at a minimum-depth leaf of depth m, we have $\alpha^m n = 1$. Thus, $\alpha^m = 1/n$. Taking logs, we get $m \lg \alpha = -\lg n$, or $m = -\lg n / \lg \alpha$.

Similarly, maximum depth corresponds to always taking the larger part of the partition, i.e., keeping a fraction $1-\alpha$ of the elements each time. The maximum depth M is reached when there is one element left, that is, when $(1-\alpha)^M n = 1$. Thus, $M = -\lg n / \lg(1-\alpha)$.

All these equations are approximate because we are ignoring floors and ceilings.

Solution to Exercise 7.3-1

We may be interested in the worst-case performance, but in that case, the randomization is irrelevant: it won't improve the worst case. What randomization can do is make the chance of encountering a worst-case scenario small.

Solution to Exercise 7.4-2

To show that quicksort's best-case running time is $\Omega(n \lg n)$, we use a technique similar to the one used in Section 7.4.1 to show that its worst-case running time is $O(n^2)$.

Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \min_{1 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) .$$

We guess that $T(n) \ge cn \lg n$ for some constant c. Substituting this guess into the recurrence, we obtain

$$\begin{aligned} T(n) &\ge \min_{1 \le q \le n-1} (cq \lg q + c(n-q-1) \lg(n-q-1)) + \Theta(n) \\ &= c \cdot \min_{1 \le q \le n-1} (q \lg q + (n-q-1) \lg(n-q-1)) + \Theta(n) . \end{aligned}$$

As we'll show below, the expression $q \lg q + (n-q-1) \lg(n-q-1)$ achieves a minimum over the range $1 \le q \le n-1$ when $q = n-q-1$, or $q = (n-1)/2$, since the first derivative of the expression with respect to q is 0 when $q = (n-1)/2$ and the second derivative of the expression is positive. (It doesn't matter that q is not an integer when n is even, since we're just trying to determine the minimum value of a function, knowing that when we constrain q to integer values, the function's value will be no lower.)

Choosing $q = (n-1)/2$ gives us the bound

$$\begin{aligned} &\min_{1 \le q \le n-1} (q \lg q + (n-q-1) \lg(n-q-1) \\ &\qquad \ge \frac{n-1}{2} \lg \frac{n-1}{2} + \left(n - \frac{n-1}{2} - 1\right) \lg \left(n - \frac{n-1}{2} - 1\right) \\ &\qquad = (n-1) \lg \frac{n-1}{2} . \end{aligned}$$

Continuing with our bounding of $T(n)$, we obtain, for $n \ge 2$,

$$\begin{aligned} T(n) &\ge c(n-1) \lg \frac{n-1}{2} + \Theta(n) \\ &= c(n-1) \lg(n-1) - c(n-1) + \Theta(n) \\ &= cn \lg(n-1) - c \lg(n-1) - c(n-1) + \Theta(n) \\ &\ge cn \lg(n/2) - c \lg(n-1) - c(n-1) + \Theta(n) \qquad \text{(since } n \ge 2\text{)} \\ &= cn \lg n - cn - c \lg(n-1) - cn + c + \Theta(n) \\ &= cn \lg n - (2cn + c \lg(n-1) - c) + \Theta(n) \\ &\ge cn \lg n , \end{aligned}$$

since we can pick the constant c small enough so that the $\Theta(n)$ term dominates the quantity $2cn + c \lg(n-1) - c$. Thus, the best-case running time of quicksort is $\Omega(n \lg n)$.

Letting $f(q) = q \lg q + (n-q-1) \lg(n-q-1)$, we now show how to find the minimum value of this function in the range $1 \le q \le n-1$. We need to find the value of q for which the derivative of f with respect to q is 0. We rewrite this function as

$$f(q) = \frac{q \ln q + (n-q-1)\ln(n-q-1)}{\ln 2} ,$$

and so

$$\begin{aligned} f'(q) &= \frac{d}{dq}\left(\frac{q \ln q + (n-q-1)\ln(n-q-1)}{\ln 2}\right) \\ &= \frac{\ln q + 1 - \ln(n-q-1) - 1}{\ln 2} \\ &= \frac{\ln q - \ln(n-q-1)}{\ln 2} . \end{aligned}$$

The derivative $f'(q)$ is 0 when $q = n-q-1$, or when $q = (n-1)/2$. To verify that $q = (n-1)/2$ is indeed a minimum (not a maximum or an inflection point), we need to check that the second derivative of f is positive at $q = (n-1)/2$:

$$\begin{aligned} f''(q) &= \frac{d}{dq}\left(\frac{\ln q - \ln(n-q-1)}{\ln 2}\right) \\ &= \frac{1}{\ln 2}\left(\frac{1}{q} + \frac{1}{n-q-1}\right) \\ f''\left(\frac{n-1}{2}\right) &= \frac{1}{\ln 2}\left(\frac{2}{n-1} + \frac{2}{n-1}\right) \\ &= \frac{1}{\ln 2} \cdot \frac{4}{n-1} \\ &> 0 \qquad \text{(since } n \ge 2) . \end{aligned}$$

Solution to Problem 7-4

a. QUICKSORT′ does exactly what QUICKSORT does; hence it sorts correctly.

QUICKSORT and QUICKSORT′ do the same partitioning, and then each calls itself with arguments $A, p, q-1$. QUICKSORT then calls itself again, with arguments $A, q+1, r$. QUICKSORT′ instead sets $p \leftarrow q+1$ and performs another iteration of its **while** loop. This executes the same operations as calling itself with $A, q+1, r$, because in both cases, the first and third arguments (A and r) have the same values as before, and p has the old value of $q+1$.

b. The stack depth of QUICKSORT′ will be $\Theta(n)$ on an n-element input array if there are $\Theta(n)$ recursive calls to QUICKSORT′. This happens if every call to PARTITION(A, p, r) returns $q = r$. The sequence of recursive calls in this scenario is

QUICKSORT′$(A, 1, n)$,
QUICKSORT′$(A, 1, n-1)$,
QUICKSORT′$(A, 1, n-2)$,
$\vdots$
QUICKSORT′$(A, 1, 1)$.

Any array that is already sorted in increasing order will cause QUICKSORT′ to behave this way.

c. The problem demonstrated by the scenario in part (b) is that each invocation of QUICKSORT′ calls QUICKSORT′ again with almost the same range. To avoid such behavior, we must change QUICKSORT′ so that the recursive call is on a smaller interval of the array. The following variation of QUICKSORT′ checks which of the two subarrays returned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the array size is reduced by at least half on each recursive call, the number of recursive calls, and hence the stack depth, is $\Theta(\lg n)$ in the worst case. Note that this method works no matter how partitioning is performed (as long as the PARTITION procedure has the same functionality as the procedure given in Section 7.1).

```
QUICKSORT″(A, p, r)
while p < r
    do ▷ Partition and sort the small subarray first
       q ← PARTITION(A, p, r)
       if q − p < r − q
          then QUICKSORT″(A, p, q − 1)
               p ← q + 1
          else QUICKSORT″(A, q + 1, r)
               r ← q − 1
```

The expected running time is not affected, because exactly the same work is done as before: the same partitions are produced, and the same subarrays are sorted.

Lecture Notes for Chapter 8: Sorting in Linear Time

Chapter 8 overview

How fast can we sort?

We will prove a lower bound, then beat it by playing a different game.

Comparison sorting

- The only operation that may be used to gain order information about a sequence is comparison of pairs of elements.
- All sorts seen so far are comparison sorts: insertion sort, selection sort, merge sort, quicksort, heapsort, treesort.

Lower bounds for sorting

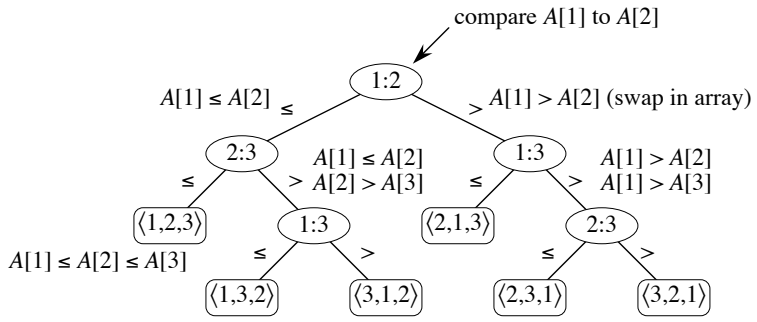
Lower bounds

- $\Omega(n)$ to examine all the input.
- All sorts seen so far are $\Omega(n \lg n)$.
- We'll show that $\Omega(n \lg n)$ is a lower bound for comparison sorts.

Decision tree

- Abstraction of any comparison sort.
- Represents comparisons made by
  - a specific sorting algorithm
  - on inputs of a given size.
- Abstracts away everything else: control and data movement.
- We're counting *only* comparisons.

For insertion sort on 3 elements:



[Each internal node is labeled by indices of array elements ***from their original positions****. Each leaf is labeled by the permutation of orders that the algorithm determines.]*

How many leaves on the decision tree? There are $\geq n!$ leaves, because every permutation appears at least once.

For any comparison sort,

- 1 tree for each n.
- View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point.
- The tree models all possible execution traces.

What is the length of the longest path from root to leaf?

- Depends on the algorithm
- Insertion sort: $\Theta(n^2)$
- Merge sort: $\Theta(n \lg n)$

Lemma

Any binary tree of height h has $\leq 2^h$ leaves.

In other words:

- $l =$ # of leaves,
- $h =$ height,
- Then $l \leq 2^h$.

(We'll prove this lemma later.)

Why is this useful?

Theorem

Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

Proof

- $l \geq n!$
- By lemma, $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Take logs: $h \geq \lg(n!)$
- Use Stirling's approximation: $n! > (n/e)^n$ (by equation (3.16))

$$\begin{aligned} h &\geq \lg(n/e)^n \\ &= n \lg(n/e) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \ . \end{aligned}$$

■ (theorem)

Now to prove the lemma:

Proof By induction on h.

Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$.

Inductive step: Assume true for height $= h - 1$. Extend tree of height $h - 1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves.

$$\begin{aligned} \#\text{ of leaves for height } h &= 2 \cdot (\#\text{ of leaves for height } h-1) \\ &= 2 \cdot 2^{h-1} \qquad \text{(ind. hypothesis)} \\ &= 2^h \ . \end{aligned}$$

■ (lemma)

Corollary
Heapsort and merge sort are asymptotically optimal comparison sorts.

Sorting in linear time

Non-comparison sorts.

Counting sort

Depends on a *key assumption*: numbers to be sorted are integers in $\{0, 1, \ldots, k\}$.

Input: $A[1..n]$, where $A[j] \in \{0, 1, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.

Output: $B[1..n]$, sorted. B is assumed to be already allocated and is given as a parameter.

Auxiliary storage: $C[0..k]$

COUNTING-SORT(A, B, n, k)
for $i \leftarrow 0$ **to** k
  do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$
for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Do an example for $A = 2_1, 5_1, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3$

Counting sort is ***stable*** (keys with same value appear in same order in output as they did in input) because of how the last loop works.

Analysis: $\Theta(n + k)$, which is $\Theta(n)$ if $k = O(n)$.

How big a k is practical?

- Good for sorting 32-bit values? No.
- 16-bit? Probably not.
- 8-bit? Maybe, depending on n.
- 4-bit? Probably (unless n is really small).

Counting sort will be used in radix sort.

Radix sort

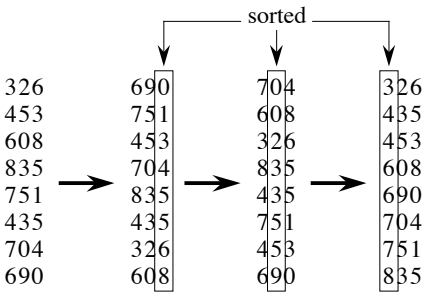
How IBM made its money. Punch card readers for census tabulation in early 1900's. Card sorters, worked on one column at a time. It's the algorithm for using the machine that extends the technique to multi-column sorting. The human operator was part of the algorithm!

Key idea: Sort *least* significant digits first.

To sort d digits:

RADIX-SORT(A, d)
for $i \leftarrow 1$ **to** d
  do use a stable sort to sort array A on digit i

Example:



Correctness:

- Induction on number of passes (i in pseudocode).
- Assume digits $1, 2, \ldots, i-1$ are sorted.
- Show that a stable sort on digit i leaves digits $1, \ldots, i$ sorted:
  - If 2 digits in position i are different, ordering by position i is correct, and positions $1, \ldots, i-1$ are irrelevant.
  - If 2 digits in position i are equal, numbers are already in the right order (by inductive hypothesis). The stable sort on digit i leaves them in the right order.

This argument shows why it's so important to use a stable sort for intermediate sort.

Analysis: Assume that we use counting sort as the intermediate sort.

- $\Theta(n+k)$ per pass (digits in range $0, \ldots, k$)
- d passes
- $\Theta(d(n+k))$ total
- If $k = O(n)$, time $= \Theta(dn)$.

How to break each key into digits?

- n words.
- b bits/word.
- Break into r-bit digits. Have $d = \lceil b/r \rceil$.
- Use counting sort, $k = 2^r - 1$.

  Example: 32-bit words, 8-bit digits. $b = 32$, $r = 8$, $d = \lceil 32/8 \rceil = 4$, $k = 2^8 - 1 = 255$.
- Time $= \Theta\left(\frac{b}{r}(n + 2^r)\right)$.

How to choose r? Balance b/r and $n + 2^r$. Choosing $r \approx \lg n$ gives us $\Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta(bn/\lg n)$.

- If we choose $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ term doesn't improve.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big. Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

So, to sort 2^{16} 32-bit numbers, use $r = \lg 2^{16} = 16$ bits. $\lceil b/r \rceil = 2$ passes.

Compare radix sort to merge sort and quicksort:

- 1 million (2^{20}) 32-bit integers.
- Radix sort: $\lceil 32/20 \rceil = 2$ passes.
- Merge sort/quicksort: $\lg n = 20$ passes.
- Remember, though, that each radix sort "pass" is really 2 passes—one to take census, and one to move data.

How does radix sort violate the ground rules for a comparison sort?

- Using counting sort allows us to gain information about keys by means other than directly comparing 2 keys.
- Used keys as array indices.

Bucket sort

Assumes the input is generated by a random process that distributes elements uniformly over $[0, 1)$.

Idea:

- Divide $[0, 1)$ into n equal-sized *buckets*.
- Distribute the n input values into the buckets.
- Sort each bucket.
- Then go through buckets in order, listing elements in each one.

Input: $A[1 \,..\, n]$, where $0 \le A[i] < 1$ for all i.

Auxiliary array: $B[0 \,..\, n-1]$ of linked lists, each list initially empty.

BUCKET-SORT(A, n)
for $i \leftarrow 1$ **to** n
  do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
for $i \leftarrow 0$ **to** $n-1$
  do sort list $B[i]$ with insertion sort
concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order
return the concatenated lists

Correctness: Consider $A[i]$, $A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Then $\lfloor n \cdot A[i] \rfloor \le \lfloor n \cdot A[j] \rfloor$. So $A[i]$ is placed into the same bucket as $A[j]$ or into a bucket with a lower index.

- If same bucket, insertion sort fixes up.
- If earlier bucket, concatenation of lists fixes up.

Analysis:

- Relies on no bucket getting too many values.
- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow$ $O(n)$ sort time for all buckets.
- We "expect" each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.

Define a random variable:

- n_i = the number of elements placed in bucket $B[i]$.

Because insertion sort runs in quadratic time, bucket sort time is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \; .$$

Take expectations of both sides:

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \qquad \text{(linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) \qquad (\mathrm{E}[aX] = a\mathrm{E}[X])
\end{aligned}$$

Claim

$\mathrm{E}[n_i^2] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim

Define indicator random variables:

- $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
- $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

Then

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \mathrm{E}[X_{ij}X_{ik}] \quad \text{(linearity of expectation)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\} \\
&= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} \\
&= \frac{1}{n}
\end{aligned}$$

$\mathrm{E}[X_{ij}X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} and X_{ik} are independent random variables

$$\begin{aligned}
\Rightarrow \mathrm{E}[X_{ij}X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\
&= \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n^2}
\end{aligned}$$

Therefore:

$$\mathrm{E}[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \frac{1}{n^2}$$

$$
\begin{aligned}
&= n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2} \\
&= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 1 + 1 - \frac{1}{n} \\
&= 2 - \frac{1}{n}
\end{aligned}
$$

■ (claim)

Therefore:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}
$$

- Again, not a comparison sort. Used a function of key values to index into an array.
- This is a ***probabilistic analysis***—we used probability to analyze an algorithm whose running time depends on the distribution of inputs.
- Different from a ***randomized algorithm***, where we use randomization to *impose* a distribution.
- With bucket sort, if the input isn't drawn from a uniform distribution on [0, 1), all bets are off (performance-wise, but the algorithm is still correct).

Solutions for Chapter 8: Sorting in Linear Time

Solution to Exercise 8.1-3

If the sort runs in linear time for m input permutations, then the height h of the portion of the decision tree consisting of the m corresponding leaves and their ancestors is linear.

Use the same argument as in the proof of Theorem 8.1 to show that this is impossible for $m = n!/2$, $n!/n$, or $n!/2^n$.

We have $2^h \geq m$, which gives us $h \geq \lg m$. For all the possible m's given here, $\lg m = \Omega(n \lg n)$, hence $h = \Omega(n \lg n)$.

In particular,

$$\begin{aligned}
\lg \frac{n!}{2} &= \lg n! - 1 \geq n \lg n - n \lg e - 1 \\
\lg \frac{n!}{n} &= \lg n! - \lg n \geq n \lg n - n \lg e - \lg n \\
\lg \frac{n!}{2^n} &= \lg n! - n \geq n \lg n - n \lg e - n
\end{aligned}$$

Solution to Exercise 8.1-4

Let S be a sequence of n elements divided into n/k subsequences each of length k where all of the elements in any subsequence are larger than all of the elements of a preceding subsequence and smaller than all of the elements of a succeeding subsequence.

Claim

Any comparison-based sorting algorithm to sort s must take $\Omega(n \lg k)$ time in the worst case.

Proof First notice that, as pointed out in the hint, we cannot prove the lower bound by multiplying together the lower bounds for sorting each subsequence. That would only prove that there is no faster algorithm *that sorts the subsequences independently*. This was not what we are asked to prove; we cannot introduce *any* extra assumptions.

Now, consider the decision tree of height h for any comparison sort for S. Since the elements of each subsequence can be in any order, any of the $k!$ permutations correspond to the final sorted order of a subsequence. And, since there are n/k such subsequences, each of which can be in any order, there are $(k!)^{n/k}$ permutations of S that could correspond to the sorting of some input order. Thus, any decision tree for sorting S must have at least $(k!)^{n/k}$ leaves. Since a binary tree of height h has no more than 2^h leaves, we must have $2^h \geq (k!)^{n/k}$ or $h \geq \lg((k!)^{n/k})$. We therefore obtain

$$\begin{aligned} h &\geq \lg((k!)^{n/k} \\ &= (n/k)\lg(k!) \\ &\geq (n/k)\lg((k/2)^{k/2}) \\ &= (n/2)\lg(k/2)\ . \end{aligned}$$

The third line comes from $k!$ having its $k/2$ largest terms being at least $k/2$ each. (We implicitly assume here that k is even. We could adjust with floors and ceilings if k were odd.)

Since there exists at least one path in any decision tree for sorting S that has length at least $(n/2)\lg(k/2)$, the worst-case running time of any comparison-based sorting algorithm for S is $\Omega(n \lg k)$. ■

Solution to Exercise 8.2-3

[The following solution also answers Exercise 8.2-2.]

Notice that the correctness argument in the text does not depend on the order in which A is processed. The algorithm is correct no matter what order is used!

But the modified algorithm is not stable. As before, in the final **for** loop an element equal to one taken from A earlier is placed before the earlier one (i.e., at a lower index position) in the output arrray B. The original algorithm was stable because an element taken from A later started out with a lower index than one taken earlier. But in the modified algorithm, an element taken from A later started out with a higher index than one taken earlier.

In particular, the algorithm still places the elements with value k in positions $C[k-1]+1$ through $C[k]$, but in the reverse order of their appearance in A.

Solution to Exercise 8.2-4

Compute the C array as is done in counting sort. The number of integers in the range $[a\,..\,b]$ is $C[b] - C[a-1]$, where we interpret $C[-1]$ as 0.

Solution to Exercise 8.3-2

Insertion sort is stable. When inserting $A[j]$ into the sorted sequence $A[1 \ldots j-1]$, we do it the following way: compare $A[j]$ to $A[i]$, starting with $i = j-1$ and going down to $i = 1$. Continue at long as $A[j] < A[i]$.

Merge sort as defined is stable, because when two elements compared are equal, the tie is broken by taking the element from array L which keeps them in the original order.

Heapsort and quicksort are not stable.

One scheme that makes a sorting algorithm stable is to store the index of each element (the element's place in the original ordering) with the element. When comparing two elements, compare them by their values and break ties by their indices.

Additional space requirements: For n elements, their indices are $1 \ldots n$. Each can be written in $\lg n$ bits, so together they take $O(n \lg n)$ additional space.

Additional time requirements: The worst case is when all elements are equal. The asymptotic time does not change because we add a constant amount of work to each comparison.

Solution to Exercise 8.3-3

Basis: If $d = 1$, there's only one digit, so sorting on that digit sorts the array.

Inductive step: Assuming that radix sort works for $d - 1$ digits, we'll show that it works for d digits.

Radix sort sorts separately on each digit, starting from digit 1. Thus, radix sort of d digits, which sorts on digits $1, \ldots, d$ is equivalent to radix sort of the low-order $d - 1$ digits followed by a sort on digit d. By our induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d, the elements are in order according to their low-order $d - 1$ digits.

The sort on digit d will order the elements by their dth digit. Consider two elements, a and b, with dth digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b, which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b, which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d - 1$ digits when their dth digits are equal, and the elements are already sorted by their low-order $d - 1$ digits.

If the intermediate sort were not stable, it might rearrange elements whose dth digits were equal—elements that *were* in the right order after the sort on their lower-order digits.

Solution to Exercise 8.3-4

Treat the numbers as 2-digit numbers in radix n. Each digit ranges from 0 to $n-1$. Sort these 2-digit numbers with radix sort.

There are 2 calls to counting sort, each taking $\Theta(n+n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

Solution to Exercise 8.4-2

The worst-case running time for the bucket-sort algorithm occurs when the assumption of uniformly distributed input does not hold. If, for example, all the input ends up in the first bucket, then in the insertion sort phase it needs to sort all the input, which takes $O(n^2)$ time.

A simple change that will preserve the linear expected running time and make the worst-case running time $O(n \lg n)$ is to use a worst-case $O(n \lg n)$-time algorithm like merge sort instead of insertion sort when sorting the buckets.

Solution to Problem 8-1

a. For a comparison algorithm A to sort, no two input permutations can reach the same leaf of the decision tree, so there must be at least $n!$ leaves reached in T_A, one for each possible input permutation. Since A is a deterministic algorithm, it must always reach the same leaf when given a particular permutation as input, so at most $n!$ leaves are reached (one for each permutation). Therefore exactly $n!$ leaves are reached, one for each input permutation.

These $n!$ leaves will each have probability $1/n!$, since each of the $n!$ possible permutations is the input with the probability $1/n!$. Any remaining leaves will have probability 0, since they are not reached for any input.

Without loss of generality, we can assume for the rest of this problem that paths leading only to 0-probability leaves aren't in the tree, since they cannot affect the running time of the sort. That is, we can assume that T_A consists of only the $n!$ leaves labeled $1/n!$ and their ancestors.

b. If $k > 1$, then the root of T is not a leaf. This implies that all of T's leaves are leaves in LT and RT. Since every leaf at depth h in LT or RT has depth $h+1$ in T, $D(T)$ must be the sum of $D(LT)$, $D(RT)$, and k, the total number of leaves. To prove this last assertion, let $d_T(x) =$ depth of node x in tree T. Then,

$$\begin{aligned} D(T) &= \sum_{x \in \text{leaves}(T)} d_T(x) \\ &= \sum_{x \in \text{leaves}(LT)} d_T(x) + \sum_{x \in \text{leaves}(RT)} d_T(x) \end{aligned}$$

$$\begin{aligned} &= \sum_{x \in \text{leaves}(LT)} (d_{LT}(x) + 1) + \sum_{x \in \text{leaves}(RT)} (d_{RT}(x) + 1) \\ &= \sum_{x \in \text{leaves}(LT)} d_{LT}(x) + \sum_{x \in \text{leaves}(RT)} d_{RT}(x) + \sum_{x \in \text{leaves}(T)} 1 \\ &= D(LT) + D(RT) + k \ . \end{aligned}$$

c. To show that $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$ we will show separately that

$$d(k) \le \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$$

and

$$d(k) \ge \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\} \ .$$

- To show that $d(k) \le \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$, we need only show that $d(k) \le d(i) + d(k-i) + k$, for $i = 1, 2, \ldots, k-1$. For any i from 1 to $k-1$ we can find trees RT with i leaves and LT with $k-i$ leaves such that $D(RT) = d(i)$ and $D(LT) = d(k-i)$. Construct T such that RT and LT are the right and left subtrees of T's root respectively. Then

  $$\begin{aligned} d(k) &\le D(T) && \text{(by definition of } d \text{ as min } D(T) \text{ value)} \\ &= D(RT) + D(LT) + k && \text{(by part (b))} \\ &= d(i) + d(k-i) + k && \text{(by choice of } RT \text{ and } LT) \ . \end{aligned}$$

- To show that $d(k) \ge \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$, we need only show that $d(k) \ge d(i) + d(k-i) + k$, for some i in $\{1, 2, \ldots, k-1\}$. Take the tree T with k leaves such that $D(T) = d(k)$, let RT and LT be T's right and left subtree, respecitvely, and let i be the number of leaves in RT. Then $k-i$ is the number of leaves in LT and

  $$\begin{aligned} d(k) &= D(T) && \text{(by choice of } T) \\ &= D(RT) + D(LT) + k && \text{(by part (b))} \\ &\ge d(i) + d(k-i) + k && \text{(by defintion of } d \text{ as min } D(T) \text{ value)} \ . \end{aligned}$$

  Neither i nor $k-i$ can be 0 (and hence $1 \le i \le k-1$), since if one of these were 0, either RT or LT would contain all k leaves of T, and that k-leaf subtree would have a D equal to $D(T) - k$ (by part (b)), contradicting the choice of T as the k-leaf tree with the minimum D.

d. Let $f_k(i) = i \lg i + (k-i) \lg(k-i)$. To find the value of i that minimizes f_k, find the i for which the derivative of f_k with respect to i is 0:

$$\begin{aligned} f_k'(i) &= \frac{d}{di}\left(\frac{i \ln i + (k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{\ln i + 1 - \ln(k-i) - 1}{\ln 2} \\ &= \frac{\ln i - \ln(k-i)}{\ln 2} \end{aligned}$$

is 0 at $i = k/2$. To verify this is indeed a minimum (not a maximum), check that the second derivative of f_k is positive at $i = k/2$:

$$f_k''(i) = \frac{d}{di}\left(\frac{\ln i - \ln(k-i)}{\ln 2}\right)$$

$$
\begin{aligned}
&= \frac{1}{\ln 2}\left(\frac{1}{i}+\frac{1}{k-i}\right) .\\
f_k''(k/2) &= \frac{1}{\ln 2}\left(\frac{2}{k}+\frac{2}{k}\right)\\
&= \frac{1}{\ln 2}\cdot\frac{4}{k}\\
&> 0 \qquad\qquad \text{since } k > 1 \ .
\end{aligned}
$$

Now we use substitution to prove $d(k) = \Omega(k \lg k)$. The base case of the induction is satisfied because $d(1) \geq 0 = c \cdot 1 \cdot \lg 1$ for any constant c. For the inductive step we assume that $d(i) \geq ci \lg i$ for $1 \leq i \leq k-1$, where c is some constant to be determined.

$$
\begin{aligned}
d(k) &= \min_{1\leq i\leq k-1}\{d(i)+d(k-i)+k\}\\
&\geq \min_{1\leq i\leq k-1}\{c(i\lg i+(k-i)\lg(k-i))+k\}\\
&= \min_{1\leq i\leq k-1}\{cf_k(i)+k\}\\
&= c\left(\frac{k}{2}\lg\frac{k}{2}\left(k-\frac{k}{2}\right)\lg\left(k-\frac{k}{2}\right)\right)+k\\
&= ck\lg\left(\frac{k}{2}\right)+k\\
&= c(k\lg k-k)+k\\
&= ck\lg k+(k-ck)\\
&\geq ck\lg k \qquad \text{if } c\leq 1 \ ,
\end{aligned}
$$

and so $d(k) = \Omega(k \lg k)$.

e. Using the result of part (d) and the fact that T_A (as modified in our solution to part (a)) has $n!$ leaves, we can conclude that

$$D(T_A) \geq d(n!) = \Omega(n! \lg(n!)) \ .$$

$D(T_A)$ is the sum of the decision-tree path lengths for sorting all input permutations, and the path lengths are proportional to the run time. Since the $n!$ permutations have equal probability $1/n!$, the expected time to sort n random elements (1 input permutation) is the total time for all permutations divided by $n!$:

$$\frac{\Omega(n!\lg(n!))}{n!} = \Omega(\lg(n!)) = \Omega(n\lg n) \ .$$

f. We will show how to modify a randomized decision tree (algorithm) to define a deterministic decision tree (algorithm) that is at least as good as the randomized one in terms of the average number of comparisons.

At each randomized node, pick the child with the smallest subtree (the subtree with the smallest average number of comparisons on a path to a leaf). Delete all the other children of the randomized node and splice out the randomized node itself.

The deterministic algorithm corresponding to this modified tree still works, because the randomized algorithm worked no matter which path was taken from each randomized node.

The average number of comparisons for the modified algorithm is no larger than the average number for the original randomized tree, since we discarded the higher-average subtrees in each case. In particular, each time we splice out a randomized node, we leave the overall average less than or equal to what it was, because

- the same set of input permutations reaches the modified subtree as before, but those inputs are handled in less than or equal to average time than before, and
- the rest of the tree is unmodified.

The randomized algorithm thus takes at least as much time on average as the corresponding deterministic one. (We've shown that the expected running time for a deterministic comparison sort is $\Omega(n \lg n)$, hence the expected time for a randomized comparison sort is also $\Omega(n \lg n)$.)

Solution to Problem 8-3

a. The usual, unadorned radix sort algorithm will not solve this problem in the required time bound. The number of passes, d, would have to be the number of digits in the largest integer. Suppose that there are m integers; we always have $m \le n$. In the worst case, we would have one integer with $n/2$ digits and $n/2$ integers with one digit each. We assume that the range of a single digit is constant. Therefore, we would have $d = n/2$ and $m = n/2 + 1$, and so the running time would be $\Theta(dm) = \Theta(n^2)$.

Let us assume without loss of generality that all the integers are positive and have no leading zeros. (If there are negative integers or 0, deal with the positive numbers, negative numbers, and 0 separately.) Under this assumption, we can observe that integers with more digits are always greater than integers with fewer digits. Thus, we can first sort the integers by number of digits (using counting sort), and then use radix sort to sort each group of integers with the same length. Noting that each integer has between 1 and n digits, let m_i be the number of integers with i digits, for $i = 1, 2, \ldots, n$. Since there are n digits altogether, we have $\sum_{i=1}^{n} i \cdot m_i = n$.

It takes $O(n)$ time to compute how many digits all the integers have and, once the numbers of digits have been computed, it takes $O(m + n) = O(n)$ time to group the integers by number of digits. To sort the group with m_i digits by radix sort takes $\Theta(i \cdot m_i)$ time. The time to sort all groups, therefore, is

$$\begin{aligned}\sum_{i=1}^{n} \Theta(i \cdot m_i) &= \Theta\left(\sum_{i=1}^{n} i \cdot m_i\right) \\ &= \Theta(n) \, .\end{aligned}$$

b. One way to solve this problem is by a radix sort from right to left. Since the strings have varying lengths, however, we have to pad out all strings that are shorter than the longest string. The padding is on the right end of the string, and it's with a special character that is lexicographically less than any other character (e.g., in C, the character `'\0'` with ASCII value 0). Of course, we

don't have to actually change any string; if we want to know the jth character of a string whose length is k, then if $j > k$, the jth character is the pad character.

Unfortunately, this scheme does not always run in the required time bound. Suppose that there are m strings and that the longest string has d characters. In the worst case, one string has $n/2$ characters and, before padding, $n/2$ strings have one character each. As in part (a), we would have $d = n/2$ and $m = n/2 + 1$. We still have to examine the pad characters in each pass of radix sort, even if we don't actually create them in the strings. Assuming that the range of a single character is constant, the running time of radix sort would be $\Theta(dm) = \Theta(n^2)$.

To solve the problem in $O(n)$ time, we use the property that, if the first letter of string x is lexicographically less that the first letter of string y, then x is lexicographically less than y, regardless of the lengths of the two strings. We take advantage of this property by sorting the strings on the first letter, using counting sort. We take an empty string as a special case and put it first. We gather together all strings with the same first letter as a group. Then we recurse, *within each group*, based on each string with the first letter removed.

The correctness of this algorithm is straightforward. Analyzing the running time is a bit trickier. Let us count the number of times that each string is sorted by a call of counting sort. Suppose that the ith string, s_i, has length l_i. Then s_i is sorted by at most $l_i + 1$ counting sorts. (The "+1" is because it may have to be sorted as an empty string at some point; for example, `ab` and `a` end up in the same group in the first pass and are then ordered based on `b` and the empty string in the second pass. The string `a` is sorted its length, 1, time plus one more time.) A call of counting sort on t strings takes $\Theta(t)$ time (remembering that the number of different characters on which we are sorting is a constant.) Thus, the total time for all calls of counting sort is

$$\begin{aligned} O\left(\sum_{i=1}^{m}(l_i + 1)\right) &= O\left(\sum_{i=1}^{m} l_i + m\right) \\ &= O(n + m) \\ &= O(n)\,, \end{aligned}$$

where the second line follows from $\sum_{i=1}^{m} l_i = n$, and the last line is because $m \le n$.

Solution to Problem 8-4

a. Compare each red jug with each blue jug. Since there are n red jugs and n blue jugs, that will take $\Theta(n^2)$ comparisons in the worst case.

b. To solve the problem, an algorithm has to perform a series of comparisons until it has enough information to determine the matching. We can view the computation of the algorithm in terms of a decision tree. Every internal node is labeled with two jugs (one red, one blue) which we compare, and has three outgoing edges (red jug smaller, same size, or larger than the blue jug). The leaves are labeled with a unique matching of jugs.

The height of the decision tree is equal to the worst-case number of comparisons the algorithm has to make to determine the matching. To bound that size, let us first compute the number of possible matchings for n red and n blue jugs.

If we label the red jugs from 1 to n and we also label the blue jugs from 1 to n before starting the comparisons, every outcome of the algorithm can be represented as a set

$$\{(i, \pi(i)) : 1 \le i \le n \text{ and } \pi \text{ is a permutation on } \{1, \ldots, n\}\} ,$$

which contains the pairs of red jugs (first component) and blue jugs (second component) that are matched up. Since every permutation π corresponds to a different outcome, there must be exactly $n!$ different results.

Now we can bound the height h of our decision tree. Every tree with a branching factor of 3 (every inner node has at most three children) has at most 3^h leaves. Since the decison tree must have at least $n!$ children, it follows that

$$3^h \ge n! \ge (n/e)^n \Rightarrow h \ge n \log_3 n - n \log_3 e = \Omega(n \lg n) .$$

So any algorithm solving the problem must use $\Omega(n \lg n)$ comparisons.

c. Assume that the red jugs are labeled with numbers $1, 2, \ldots, n$ and so are the blue jugs. The numbers are arbitrary and do not correspond to the volumes of jugs, but are just used to refer to the jugs in the algorithm description. Moreover, the output of the algorithm will consist of n distinct pairs (i, j), where the red jug i and the blue jug j have the same volume.

The procedure MATCH-JUGS takes as input two sets representing jugs to be matched: $R \subseteq \{1, \ldots, n\}$, representing red jugs, and $B \subseteq \{1, \ldots, n\}$, representing blue jugs. We will call the procedure only with inputs that can be matched; one necessary condition is that $|R| = |B|$.

```
MATCH-JUGS(R, B)
if |R| = 0                          ▷ Sets are empty
  then return
if |R| = 1                          ▷ Sets contain just one jug each
  then let R = {r} and B = {b}
       output "(r, b)"
       return
  else r ← a randomly chosen jug in R
       compare r to every jug of B
       B_< ← the set of jugs in B that are smaller than r
       B_> ← the set of jugs in B that are larger than r
       b ← the one jug in B with the same size as r
       compare b to every jug of R − {r}
       R_< ← the set of jugs in R that are smaller than b
       R_> ← the set of jugs in R that are larger than b
       output "(r, b)"
       MATCH-JUGS(R_<, B_<)
       MATCH-JUGS(R_>, B_>)
```

Correctness can be seen as follows (remember that $|R| = |B|$ in each call). Once we pick r randomly from R, there will be a matching among the jugs in

volume smaller than r (which are in the sets $R_<$ and $B_<$), and likewise between the jugs larger than r (which are in $R_>$ and $B_>$). Termination is also easy to see: since $|R_<| + |R_>| < |R|$ in every recursive step, the size of the first parameter reduces with every recursive call. It eventually must reach 0 or 1, in which case the recursion terminates.

What about the running time? The analysis of the expected number of comparisons is similar to that of the quicksort algorithm in Section 7.4.2. Let us order the jugs as $r_1, \ldots, r_n$ and $b_1, \ldots, b_n$ where $r_i < r_{i+1}$ and $b_i < b_{i+1}$ for $i = 1, \ldots, n$, and $r_i = b_i$. Our analysis uses indicator random variables

$$X_{ij} = \mathrm{I}\{\text{red jug } r_i \text{ is compared to blue jug } b_j\} \ .$$

As in quicksort, a given pair r_i and b_j is compared at most once. When we compare r_i to every jug in B, jug r_i will not be put in either $R_<$ or $R_>$. When we compare b_i to every jug in $R - \{r_i\}$, jug b_i is not put into either $B_<$ or $B_>$. The total number of comparisons is

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

To calculate the expected value of X, we follow the quicksort analysis to arrive at

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{r_i \text{ is compared to } b_j\} \ .$$

As in the quicksort analysis, once we choose a jug r_k such that $r_i < r_k < b_j$, we will put r_i in $R_<$ and b_j in $R_>$, and so r_i and b_j will never be compared again. Let us denote $R_{ij} = \{r_i, \ldots, r_j\}$. Then jugs r_i and b_j will be compared if and only if the first jug in R_{ij} to be chosen is either r_i or r_j.

Still following the quicksort analysis, until a jug from R_{ij} is chosen, the entire set R_{ij} is together. Any jug in R_{ij} is equally likely to be first one chosen. Since $|R_{ij}| = j - i + 1$, the probability of any given jug being the first one chosen in R_{ij} is $1/(j-i+1)$. The remainder of the analysis is the same as the quicksort analysis, and we arrive at the solution of $O(n \lg n)$ comparisons.

Just like in quicksort, in the worst case we always choose the largest (or smallest) jug to partition the sets, which reduces the set sizes by only 1. The running time then obeys the recurrence $T(n) = T(n-1) + \Theta(n)$, and the number of comparisons we make in the worst case is $T(n) = \Theta(n^2)$.

Lecture Notes for Chapter 9: Medians and Order Statistics

Chapter 9 overview

- ***i*th order statistic** is the ith smallest element of a set of n elements.
- The ***minimum*** is the first order statistic ($i = 1$).
- The ***maximum*** is the nth order statistic ($i = n$).
- A ***median*** is the "halfway point" of the set.
- When n is odd, the median is unique, at $i = (n + 1)/2$.
- When n is even, there are two medians:
  - The ***lower median***, at $i = n/2$, and
  - The ***upper median***, at $i = n/2 + 1$.
  - We mean lower median when we use the phrase "the median."

The ***selection problem***:

Input: A set A of n distinct numbers and a number i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in A. In other words, the ith smallest element of A.

The selection problem can be solved in $O(n \lg n)$ time.

- Sort the numbers using an $O(n \lg n)$-time algorithm, such as heapsort or merge sort.
- Then return the ith element in the sorted array.

There are faster algorithms, however.

- First, we'll look at the problem of selecting the minimum and maximum of a set of elements.
- Then, we'll look at a simple general selection algorithm with a time bound of $O(n)$ in the average case.
- Finally, we'll look at a more complicated general selection algorithm with a time bound of $O(n)$ in the worst case.

Minimum and maximum

We can easily obtain an upper bound of $n-1$ comparisons for finding the minimum of a set of n elements.

- Examine each element in turn and keep track of the smallest one.
- This is the best we can do, because each element, except the minimum, must be compared to a smaller element at least once.

The following pseudocode finds the minimum element in array $A[1\,..\,n]$:

```
MINIMUM(A, n)
min ← A[1]
for i ← 2 to n
    do if min > A[i]
          then min ← A[i]
return min
```

The maximum can be found in exactly the same way by replacing the $>$ with $<$ in the above algorithm.

Simultaneous minimum and maximum

Some applications need both the minimum and maximum of a set of elements.

- For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display. To do so, the program must first find the minimum and maximum of each coordinate.

A simple algorithm to find the minimum and maximum is to find each one independently. There will be $n-1$ comparisons for the minimum and $n-1$ comparisons for the maximum, for a total of $2n-2$ comparisons. This will result in $\Theta(n)$ time.

In fact, at most $3\lfloor n/2 \rfloor$ comparisons are needed to find both the minimum and maximum:

- Maintain the minimum and maximum of elements seen so far.
- Don't compare each element to the minimum and maximum separately.
- Process elements in pairs.
- Compare the elements of a pair to each other.
- Then compare the larger element to the maximum so far, and compare the smaller element to the minimum so far.

This leads to only 3 comparisons for every 2 elements.

Setting up the initial values for the min and max depends on whether n is odd or even.

- If n is even, compare the first two elements and assign the larger to max and the smaller to min. Then process the rest of the elements in pairs.
- If n is odd, set both min and max to the first element. Then process the rest of the elements in pairs.

Analysis of the total number of comparisons

- If n is even, we do 1 initial comparison and then $3(n-2)/2$ more comparisons.

$$\begin{aligned} \#\text{ of comparisons} &= \frac{3(n-2)}{2} + 1 \\ &= \frac{3n-6}{2} + 1 \\ &= \frac{3n}{2} - 3 + 1 \\ &= \frac{3n}{2} - 2\,. \end{aligned}$$

- If n is odd, we do $3(n-1)/2 = 3\lfloor n/2 \rfloor$ comparisons.

In either case, the maximum number of comparisons is $\leq 3\lfloor n/2 \rfloor$.

Selection in expected linear time

Selection of the ith smallest element of the array A can be done in $\Theta(n)$ time.

The function RANDOMIZED-SELECT uses RANDOMIZED-PARTITION from the quicksort algorithm in Chapter 7. RANDOMIZED-SELECT differs from quicksort because it recurses on one side of the partition only.

```
RANDOMIZED-SELECT(A, p, r, i)
if p = r
  then return A[p]
q ← RANDOMIZED-PARTITION(A, p, r)
k ← q − p + 1
if i = k          ▷ pivot value is the answer
  then return A[q]
elseif i < k
  then return RANDOMIZED-SELECT(A, p, q − 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

After the call to RANDOMIZED-PARTITION, the array is partitioned into two subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$, along with a ***pivot*** element $A[q]$.

- The elements of subarray $A[p\,..\,q-1]$ are all $\leq A[q]$.
- The elements of subarray $A[q+1\,..\,r]$ are all $> A[q]$.
- The pivot element is the kth element of the subarray $A[p\,..\,r]$, where $k = q-p+1$.
- If the pivot element is the ith smallest element (i.e., $i = k$), return $A[q]$.
- Otherwise, recurse on the subarray containing the ith smallest element.
  - If $i < k$, this subarray is $A[p\,..\,q-1]$, and we want the ith smallest element.
  - If $i > k$, this subarray is $A[q+1\,..\,r]$ and, since there are k elements in $A[p\,..\,r]$ that precede $A[q+1\,..\,r]$, we want the $(i-k)$th smallest element of this subarray.

Analysis

Worst-case running time: $\Theta(n^2)$, because we could be extremely unlucky and always recurse on a subarray that is only 1 element smaller than the previous subarray.

Expected running time: RANDOMIZED-SELECT works well on average. Because it is randomized, no particular input brings out the worst-case behavior consistently.

The running time of RANDOMIZED-SELECT is a random variable that we denote by $T(n)$. We obtain an upper bound on $\mathrm{E}[T(n)]$ as follows:

- RANDOMIZED-PARTITION is equally likely to return any element of A as the pivot.
- For each k such that $1 \le k \le n$, the subarray $A[p\,..\,q]$ has k elements (all $\le$ pivot) with probability $1/n$. *[Note that we're now considering a subarray that includes the pivot, along with elements less than the pivot.]*
- For $k = 1, 2, \ldots, n$, define indicator random variable

  $X_k = \mathrm{I}\{\text{subarray } A[p\,..\,q] \text{ has exactly } k \text{ elements}\}$.
- Since $\Pr\{\text{subarray } A[p\,..\,q] \text{ has exactly } k \text{ elements}\} = 1/n$, Lemma 5.1 says that $\mathrm{E}[X_k] = 1/n$.
- When we call RANDOMIZED-SELECT, we don't know if it will terminate immediately with the correct answer, recurse on $A[p\,..\,q-1]$, or recurse on $A[q+1\,..\,r]$. It depends on whether the ith smallest element is less than, equal to, or greater than the pivot element $A[q]$.
- To obtain an upper bound, we assume that $T(n)$ is monotonically increasing and that the ith smallest element is always in the larger subarray.
- For a given call of RANDOMIZED-SELECT, $X_k = 1$ for exactly one value of k, and $X_k = 0$ for all other k.
- When $X_k = 1$, the two subarrays have sizes $k-1$ and $n-k$.
- For a subproblem of size n, RANDOMIZED-PARTITION takes $O(n)$ time. *[Actually, it takes $\Theta(n)$ time, but $O(n)$ suffices, since we're obtaining only an upper bound on the expected running time.]*
- Therefore, we have the recurrence

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) \,. \end{aligned}$$

- Taking expected values gives

$$\begin{aligned} &\mathrm{E}[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) \qquad \text{(linearity of expectation)} \end{aligned}$$

$$= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) \quad \text{(equation (C.23))}$$

$$= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) \; .$$

- We rely on X_k and $T(\max(k-1, n-k))$ being independent random variables in order to apply equation (C.23).
- Looking at the expression $\max(k-1, n-k)$, we have

  $$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \; , \\ n-k & \text{if } k \le \lceil n/2 \rceil \; . \end{cases}$$

  - If n is even, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears exactly twice in the summation.
  - If n is odd, these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once.
- Either way,

  $$\mathrm{E}[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n) \; .$$

- Solve this recurrence by substitution:
  - Guess that $T(n) \le cn$ for some constant c that satisfies the initial conditions of the recurrence.
  - Assume that $T(n) = O(1)$ for $n <$ some constant. We'll pick this constant later.
  - Also pick a constant a such that the function described by the $O(n)$ term is bounded from above by an for all $n > 0$.
  - Using this guess and constants c and a, we have

$$\begin{aligned}
\mathrm{E}[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\,\lfloor n/2 \rfloor}{2} \right) + an \\
&\le \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\
&= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\
&\le \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right) \; .
\end{aligned}$$

- To complete this proof, we choose c such that

$$\begin{aligned} cn/4 - c/2 - an &\geq 0 \\ cn/4 - an &\geq c/2 \\ n(c/4 - a) &\geq c/2 \\ n &\geq \frac{c/2}{c/4 - a} \\ n &\geq \frac{2c}{c - 4a} . \end{aligned}$$

- Thus, as long as we assume that $T(n) = O(1)$ for $n < 2c/(c - 4a)$, we have $\mathrm{E}[T(n)] = O(n)$.

Therefore, we can determine any order statistic in linear time on average.

Selection in worst-case linear time

We can find the ith smallest element in $O(n)$ time *in the worst case*. We'll describe a procedure SELECT that does so.

SELECT recursively partitions the input array.

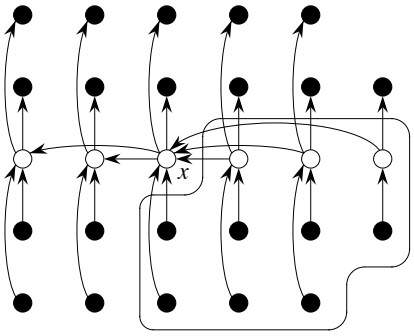
- ***Idea:*** Guarantee a good split when the array is partitioned.
- Will use the deterministic procedure PARTITION, but with a small modification. Instead of assuming that the last element of the subarray is the pivot, the modified PARTITION procedure is told which element to use as the pivot.

SELECT works on an array of $n > 1$ elements. It executes the following steps:

1. Divide the n elements into groups of 5. Get $\lceil n/5 \rceil$ groups: $\lfloor n/5 \rfloor$ groups with exactly 5 elements and, if 5 does not divide n, one group with the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups:
   - Run insertion sort on each group. Takes $O(1)$ time per group since each group has ≤ 5 elements.
   - Then just pick the median from each group, in $O(1)$ time.
3. Find the median x of the $\lceil n/5 \rceil$ medians by a recursive call to SELECT. (If $\lceil n/5 \rceil$ is even, then follow our convention and find the lower median.)
4. Using the modified version of PARTITION that takes the pivot element as input, partition the input array around x. Let x be the kth element of the array after partitioning, so that there are $k - 1$ elements on the low side of the partition and $n - k$ elements on the high side.
5. Now there are three possibilities:
   - If $i = k$, just return x.
   - If $i < k$, return the ith smallest element on the low side of the partition by making a recursive call to SELECT.
   - If $i > k$, return the $(i - k)$th smallest element on the high side of the partition by making a recursive call to SELECT.

Analysis

Start by getting a lower bound on the number of elements that are greater than the partitioning element x:



[Each group is a column. Each white circle is the median of a group, as found in step 2. Arrows go from larger elements to smaller elements, based on what we know after step 4. Elements in the region on the lower right are known to be greater than x.]

- At least half of the medians found in step 2 are $\geq x$.
- Look at the groups containing these medians that are $\geq x$. All of them contribute 3 elements that are $> x$ (the median of the group and the 2 elements in the group greater than the group's median), except for 2 of the groups: the group containing x (which has only 2 elements $> x$) and the group with < 5 elements.
- Forget about these 2 groups. That leaves $\geq \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2$ groups with 3 elements known to be $> x$.
- Thus, we know that at least

  $$3\left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

  elements are $> x$.

Symmetrically, the number of elements that are $< x$ is at least $3n/10 - 6$.

Therefore, when we call SELECT recursively in step 5, it's on $\leq 7n/10 + 6$ elements.

Develop a recurrence for the worst-case running time of SELECT:

- Steps 1, 2, and 4 each take $O(n)$ time:
  - Step 1: making groups of 5 elements takes $O(n)$ time.
  - Step 2: sorting $\lceil n/5 \rceil$ groups in $O(1)$ time each.
  - Step 4: partitioning the n-element array around x takes $O(n)$ time.
- Step 3 takes time $T(\lceil n/5 \rceil)$.
- Step 5 takes time $\leq T(7n/10 + 6)$, assuming that $T(n)$ is monotonically increasing.

- Assume that $T(n) = O(1)$ for small enough n. We'll use $n < 140$ as "small enough." Why 140? We'll see why later.
- Thus, we get the recurrence

$$T(n) \le \begin{cases} O(1) & \text{if } n < 140 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \ge 140 . \end{cases}$$

Solve this recurrence by substitution:

- ***Inductive hypothesis:*** $T(n) \le cn$ for some constant c and all $n > 0$.
- Assume that c is large enough that $T(n) \le cn$ for all $n < 140$. So we are concerned only with the case $n \ge 140$.
- Pick a constant a such that the function described by the $O(n)$ term in the recurrence is $\le an$ for all $n > 0$.
- Substitute the inductive hypothesis in the right-hand side of the recurrence:

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) . \end{aligned}$$

- This last quantity is $\le cn$ if

$$\begin{aligned} -cn/10 + 7c + an &\le 0 \\ cn/10 - 7c &\ge an \\ cn - 70c &\ge 10an \\ c(n - 70) &\ge 10an \\ c &\ge 10a(n/(n - 70)) . \end{aligned}$$

- Because we assumed that $n \ge 140$, we have $n/(n - 70) \le 2$.
- Thus, $20a \ge 10a(n/(n-70))$, so choosing $c \ge 20a$ gives $c \ge 10a(n/(n-70))$, which in turn gives us the condition we need to show that $T(n) \le cn$.
- We conclude that $T(n) = O(n)$, so that SELECT runs in linear time in all cases.
- Why 140? We could have used any integer strictly greater than 70.
  - Observe that for $n > 70$, the fraction $n/(n - 70)$ decreases as n increases.
  - We picked $n \ge 140$ so that the fraction would be ≤ 2, which is an easy constant to work with.
  - We could have picked, say, $n \ge 71$, so that for all $n \ge 71$, the fraction would be $\le 71/(71 - 70) = 71$. Then we would have had $20a \ge 710a$, so we'd have needed to choose $c \ge 710a$.

Notice that SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements.

- Sorting requires $\Omega(n \lg n)$ time in the comparison model.
- Sorting algorithms that run in linear time need to make assumptions about their input.
- Linear-time *selection* algorithms do not require any assumptions about their input.
- Linear-time selection algorithms solve the selection problem without sorting and therefore are not subject to the $\Omega(n \lg n)$ lower bound.

Solutions for Chapter 9: Medians and Order Statistics

Solution to Exercise 9.1-1

The smallest of n numbers can be found with $n-1$ comparisons by conducting a tournament as follows: Compare all the numbers in pairs. Only the smaller of each pair could possibly be the smallest of all n, so the problem has been reduced to that of finding the smallest of $\lceil n/2 \rceil$ numbers. Compare those numbers in pairs, and so on, until there's just one number left, which is the answer.

To see that this algorithm does exactly $n-1$ comparisons, notice that each number except the smallest loses exactly once. To show this more formally, draw a binary tree of the comparisons the algorithm does. The n numbers are the leaves, and each number that came out smaller in a comparison is the parent of the two numbers that were compared. Each non-leaf node of the tree represents a comparison, and there are $n-1$ internal nodes in an n-leaf full binary tree (see Exercise (B.5-3)), so exactly $n-1$ comparisons are made.

In the search for the smallest number, the second smallest number must have come out smallest in every comparison made with it until it was eventually compared with the smallest. So the second smallest is among the elements that were compared with the smallest during the tournament. To find it, conduct another tournament (as above) to find the smallest of these numbers. At most $\lceil \lg n \rceil$ (the height of the tree of comparisons) elements were compared with the smallest, so finding the smallest of these takes $\lceil \lg n \rceil - 1$ comparisons in the worst case.

The total number of comparisons made in the two tournaments was

$$n - 1 + \lceil \lg n \rceil - 1 = n + \lceil \lg n \rceil - 2$$

in the worst case.

Solution to Exercise 9.3-1

For groups of 7, the algorithm still works in linear time. The number of elements greater than x (and similarly, the number less than x) is at least

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7} \right\rceil \right\rceil - 2\right) \geq \frac{2n}{7} - 8 ,$$

and the recurrence becomes

$$T(n) \le T(\lceil n/7 \rceil) + T(5n/7 + 8) + O(n) ,$$

which can be shown to be $O(n)$ by substitution, as for the groups of 5 case in the text.

For groups of 3, however, the algorithm no longer works in linear time. The number of elements greater than x, and the number of elements less than x, is at least

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 2\right) \ge \frac{n}{3} - 4 ,$$

and the recurrence becomes

$$T(n) \le T(\lceil n/3 \rceil) + T(2n/3 + 4) + O(n) ,$$

which does not have a linear solution.

We can prove that the worst-case time for groups of 3 is $\Omega(n \lg n)$. We do so by deriving a recurrence for a particular case that takes $\Omega(n \lg n)$ time.

In counting up the number of elements greater than x (and similarly, the number less than x), consider the particular case in which there are exactly $\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil$ groups with medians $\ge x$ and in which the "leftover" group does contribute 2 elements greater than x. Then the number of elements greater than x is exactly $2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 1\right) + 1$ (the -1 discounts x's group, as usual, and the $+1$ is contributed by x's group) $= 2\lceil n/6 \rceil - 1$, and the recursive step for elements $\le x$ has $n - (2\lceil n/6 \rceil - 1) \ge n - (2(n/6 + 1) - 1) = 2n/3 - 1$ elements. Observe also that the $O(n)$ term in the recurrence is really $\Theta(n)$, since the partitioning in step 4 takes $\Theta(n)$ (not just $O(n)$) time. Thus, we get the recurrence

$$T(n) \ge T(\lceil n/3 \rceil) + T(2n/3 - 1) + \Theta(n) \ge T(n/3) + T(2n/3 - 1) + \Theta(n) ,$$

from which you can show that $T(n) \ge cn \lg n$ by substitution. You can also see that $T(n)$ is nonlinear by noticing that each level of the recursion tree sums to n.

[In fact, any odd group size ≥ 5 works in linear time.]

Solution to Exercise 9.3-3

A modification to quicksort that allows it to run in $O(n \lg n)$ time in the worst case uses the deterministic PARTITION algorithm that was modified to take an element to partition around as an input parameter.

SELECT takes an array A, the bounds p and r of the subarray in A, and the rank i of an order statistic, and in time linear in the size of the subarray $A[p \,..\, r]$ it returns the ith smallest element in $A[p \,..\, r]$.

```
BEST-CASE-QUICKSORT(A, p, r)
if p < r
  then i ← ⌊(r − p + 1)/2⌋
       x ← SELECT(A, p, r, i)
       q ← PARTITION(x)
       BEST-CASE-QUICKSORT(A, p, q − 1)
       BEST-CASE-QUICKSORT(A, q + 1, r)
```

For an n-element array, the largest subarray that BEST-CASE-QUICKSORT recurses on has $n/2$ elements. This situation occurs when $n = r - p + 1$ is even; then the subarray $A[q+1\,..\,r]$ has $n/2$ elements, and the subarray $A[p\,..\,q-1]$ has $n/2-1$ elements.

Because BEST-CASE-QUICKSORT always recurses on subarrays that are at most half the size of the original array, the recurrence for the worst-case running time is $T(n) \le 2T(n/2) + \Theta(n) = O(n \lg n)$.

Solution to Exercise 9.3-5

We assume that are given a procedure MEDIAN that takes as parameters an array A and subarray indices p and r, and returns the value of the median element of $A[p\,..\,r]$ in $O(n)$ time in the worst case.

Given MEDIAN, here is a linear-time algorithm SELECT$'$ for finding the ith smallest element in $A[p\,..\,r]$. This algorithm uses the deterministic PARTITION algorithm that was modified to take an element to partition around as an input parameter.

```
SELECT'(A, p, r, i)
if p = r
  then return A[p]
x ← MEDIAN(A, p, r)
q ← PARTITION(x)
k ← q − p + 1
if i = k
  then return A[q]
elseif i < k
  then return SELECT'(A, p, q − 1, i)
else return SELECT'(A, q + 1, r, i − k)
```

Because x is the median of $A[p\,..\,r]$, each of the subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ has at most half the number of elements of $A[p\,..\,r]$. The recurrence for the worst-case running time of SELECT$'$ is $T(n) \le T(n/2) + O(n) = O(n)$.

Solution to Exercise 9.3-8

Let's start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let's call the median value m, and let's suppose that it's in $X[k]$. Then k elements of X are less than or equal to m and $n-k$ elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than or equal to m and n elements greater than or equal to m, and so there must be $n-k$ elements of Y that are less than or equal to m and $n-(n-k) = k$ elements of Y that are greater than or equal to m.

Thus, we can check that $X[k]$ is the lower median by checking whether $Y[n-k] \le X[k] \le Y[n-k+1]$. A boundary case occurs for $k = n$. Then $n - k = 0$, and there is no array entry $Y[0]$; we only need to check that $X[n] \le Y[1]$.

Now, if the median is in X but is not in $X[k]$, then the above condition will not hold. If the median is in $X[k']$, where $k' < k$, then $X[k]$ is above the median, and $Y[n-k+1] < X[k]$. Conversely, if the median is in $X[k'']$, where $k'' > k$, then $X[k]$ is below the median, and $X[k] < Y[n-k]$.

Thus, we can use a binary search to determine whether there is an $X[k]$ such that either $k < n$ and $Y[n-k] \le X[k] \le Y[n-k+1]$ or $k = n$ and $X[k] \le Y[n-k+1]$; if we find such an $X[k]$, then it is the median. Otherwise, we know that the median is in Y, and we use a binary search to find a $Y[k]$ such that either $k < n$ and $X[n-k] \le Y[k] \le X[n-k+1]$ or $k = n$ and $Y[k] \le X[n-k+1]$; such a $Y[k]$ is the median. Since each binary search takes $O(\lg n)$ time, we spend a total of $O(\lg n)$ time.

Here's how we write the algorithm in pseudocode:

```
TWO-ARRAY-MEDIAN(X, Y)
n ← length[X]                    ▷ n also equals length[Y]
median ← FIND-MEDIAN(X, Y, n, 1, n)
if median = NOT-FOUND
  then median ← FIND-MEDIAN(Y, X, n, 1, n)
return median

FIND-MEDIAN(A, B, n, low, high)
if low > high
  then return NOT-FOUND
  else k ← ⌊(low + high)/2⌋
       if k = n and A[n] ≤ B[1]
         then return A[n]
       elseif k < n and B[n − k] ≤ A[k] ≤ B[n − k + 1]
         then return A[k]
       elseif A[k] > B[n − k + 1]
         then return FIND-MEDIAN(A, B, n, low, k − 1)
       else return FIND-MEDIAN(A, B, n, k + 1, high)
```

Solution to Exercise 9.3-9

In order to find the optimal placement for Professor Olay's pipeline, we need only find the median(s) of the y-coordinates of his oil wells, as the following proof explains.

Claim

The optimal y-coordinate for Professor Olay's east-west oil pipeline is as follows:

- If n is even, then on either the oil well whose y-coordinate is the lower median or the one whose y-coordinate is the upper median, or anywhere between them.
- If n is odd, then on the oil well whose y-coordinate is the median.

Proof We examine various cases. In each case, we will start out with the pipeline at a particular y-coordinate and see what happens when we move it. We'll denote by s the sum of the north-south spurs with the pipeline at the starting location, and s' will denote the sum after moving the pipeline.

We start with the case in which n is even. Let us start with the pipeline somewhere on or between the two oil wells whose y-coordinates are the lower and upper medians. If we move the pipeline by a vertical distance d without crossing either of the median wells, then $n/2$ of the wells become d farther from the pipeline and $n/2$ become d closer, and so $s' = s + dn/2 - dn/2 = s$; thus, all locations on or between the two medians are equally good.

Now suppose that the pipeline goes through the oil well whose y-coordinate is the upper median. What happens when we increase the y-coordinate of the pipeline by $d > 0$ units, so that it moves above the oil well that achieves the upper median? All oil wells whose y-coordinates are at or below the upper median become d units farther from the pipeline, and there are at least $n/2 + 1$ such oil wells (the upper median, and every well at or below the lower median). There are at most $n/2 - 1$ oil wells whose y-coordinates are above the upper median, and each of these oil wells becomes at most d units closer to the pipeline when it moves up. Thus, we have a lower bound on s' of $s' \geq s + d(n/2 + 1) - d(n/2 - 1) = s + 2d > s$. We conclude that moving the pipeline up from the oil well at the upper median increases the total spur length. A symmetric argument shows that if we start with the pipeline going through the oil well whose y-coordinate is the lower median and move it down, then the total spur length increases.

We see, therefore, that when n is even, an optimal placement of the pipeline is anywhere on or between the two medians.

Now we consider the case when n is odd. We start with the pipeline going through the oil well whose y-coordinate is the median, and we consider what happens when we move it up by $d > 0$ units. All oil wells at or below the median become d units farther from the pipeline, and there are at least $(n + 1)/2$ such wells (the one at the median and the $(n - 1)/2$ at or below the median. There are at most $(n - 1)/2$ oil wells above the median, and each of these becomes at most d units closer to the pipeline. We get a lower bound on s' of $s' \geq s + d(n + 1)/2 - d(n - 1)/2 = s + d > s$, and we conclude that moving the pipeline up from the oil well at the median increases the total spur length. A symmetric argument shows that moving the pipeline down from the median also increases the total spur length, and so the optimal placement of the pipeline is on the median. ■ (claim)

Since we know we are looking for the median, we can use the linear-time median-finding algorithm.

Solution to Problem 9-1

We assume that the numbers start out in an array.

a. Sort the numbers using merge sort or heapsort, which take $\Theta(n \lg n)$ worst-case time. (Don't use quicksort or insertion sort, which can take $\Theta(n^2)$ time.) Put

the i largest elements (directly accessible in the sorted array) into the output array, taking $\Theta(i)$ time.

Total worst-case running time: $\Theta(n \lg n + i) = \Theta(n \lg n)$ (because $i \le n$).

b. Implement the priority queue as a heap. Build the heap using BUILD-HEAP, which takes $\Theta(n)$ time, then call HEAP-EXTRACT-MAX i times to get the i largest elements, in $\Theta(i \lg n)$ worst-case time, and store them in reverse order of extraction in the output array. The worst-case extraction time is $\Theta(i \lg n)$ because

- i extractions from a heap with $O(n)$ elements takes $i \cdot O(\lg n) = O(i \lg n)$ time, and
- half of the i extractions are from a heap with $\ge n/2$ elements, so those $i/2$ extractions take $(i/2)\Omega(\lg(n/2)) = \Omega(i \lg n)$ time in the worst case.

Total worst-case running time: $\Theta(n + i \lg n)$.

c. Use the SELECT algorithm of Section 9.3 to find the ith largest number in $\Theta(n)$ time. Partition around that number in $\Theta(n)$ time. Sort the i largest numbers in $\Theta(i \lg i)$ worst-case time (with merge sort or heapsort).

Total worst-case running time: $\Theta(n + i \lg i)$.

Note that method (c) is always asymptotically at least as good as the other two methods, and that method (b) is asymptotically at least as good as (a). (Comparing (c) to (b) is easy, but it is less obvious how to compare (c) and (b) to (a). (c) and (b) are asymptotically at least as good as (a) because n, $i \lg i$, and $i \lg n$ are all $O(n \lg n)$. The sum of two things that are $O(n \lg n)$ is also $O(n \lg n)$.)

Solution to Problem 9-2

a. The median x of the elements $x_1, x_2, \ldots, x_n$, is an element $x = x_k$ satisfying $|\{x_i : 1 \le i \le n \text{ and } x_i < x\}| \le n/2$ and $|\{x_i : 1 \le i \le n \text{ and } x_i > x\}| \le n/2$. If each element x_i is assigned a weight $w_i = 1/n$, then we get

$$\begin{aligned}
\sum_{x_i < x} w_i &= \sum_{x_i < x} \frac{1}{n} \\
&= \frac{1}{n} \cdot \sum_{x_i < x} 1 \\
&= \frac{1}{n} \cdot |\{x_i : 1 \le i \le n \text{ and } x_i < x\}| \\
&\le \frac{1}{n} \cdot \frac{n}{2} \\
&= \frac{1}{2} ,
\end{aligned}$$

and

$$\sum_{x_i > x} w_i = \sum_{x_i > x} \frac{1}{n}$$

$$\begin{aligned} &= \frac{1}{n} \cdot \sum_{x_i > x} 1 \\ &= \frac{1}{n} \cdot |\{x_i : 1 \le i \le n \text{ and } x_i > x\}| \\ &\le \frac{1}{n} \cdot \frac{n}{2} \\ &= \frac{1}{2}, \end{aligned}$$

which proves that x is also the weighted median of $x_1, x_2, \ldots, x_n$ with weights $w_i = 1/n$, for $i = 1, 2, \ldots, n$.

b. We first sort the n elements into increasing order by x_i values. Then we scan the array of sorted x_i's, starting with the smallest element and accumulating weights as we scan, until the total exceeds $1/2$. The last element, say x_k, whose weight caused the total to exceed $1/2$, is the weighted median. Notice that the total weight of all elements smaller than x_k is less than $1/2$, because x_k was the first element that caused the total weight to exceed $1/2$. Similarly, the total weight of all elements larger than x_k is also less than $1/2$, because the total weight of all the other elements exceeds $1/2$.

The sorting phase can be done in $O(n \lg n)$ worst-case time (using merge sort or heapsort), and the scanning phase takes $O(n)$ time. The total running time in the worst case, therefore, is $O(n \lg n)$.

c. We find the weighted median in $\Theta(n)$ worst-case time using the $\Theta(n)$ worst-case median algorithm in Section 9.3. (Although the first paragraph of the section only claims an $O(n)$ upper bound, it is easy to see that the more precise running time of $\Theta(n)$ applies as well, since steps 1, 2, and 4 of SELECT actually take $\Theta(n)$ time.)

The weighted-median algorithm works as follows. If $n \le 2$, we just return the brute-force solution. Otherwise, we proceed as follows. We find the actual median x_k of the n elements and then partition around it. We then compute the total weights of the two halves. If the weights of the two halves are each strictly less than $1/2$, then the weighted median is x_k. Otherwise, the weighted median should be in the half with total weight exceeding $1/2$. The total weight of the "light" half is lumped into the weight of x_k, and the search continues within the half that weighs more than $1/2$. Here's pseudocode, which takes as input a set $X = \{x_1, x_2, \ldots, x_n\}$:

```
WEIGHTED-MEDIAN(X)
if n = 1
  then return x_1
elseif n = 2
  then if w_1 ≥ w_2
         then return x_1
         else  return x_2
else
       find the median x_k of X = {x_1, x_2, ..., x_n}
       partition the set X around x_k
       compute W_L = Σ_{x_i<x_k} w_i and W_G = Σ_{x_i>x_k} w_i
       if W_L < 1/2 and W_G < 1/2
         then return x_k
       elseif W_L > 1/2
         then w_k ← w_k + W_G
              X' ← {x_i ∈ X : x_i ≤ x_k}
              return WEIGHTED-MEDIAN(X')
       else w_k ← w_k + W_L
              X' ← {x_i ∈ X : x_i ≥ x_k}
              return WEIGHTED-MEDIAN(X')
```

The recurrence for the worst-case running time of WEIGHTED-MEDIAN is $T(n) = T(n/2+1) + \Theta(n)$, since there is at most one recursive call on half the number of elements, plus the median element x_k, and all the work preceding the recursive call takes $\Theta(n)$ time. The solution of the recurrence is $T(n) = \Theta(n)$.

d. Let the n points be denoted by their coordinates $x_1, x_2, \ldots, x_n$, let the corresponding weights be $w_1, w_2, \ldots, w_n$, and let $x = x_k$ be the weighted median. For any point p, let $f(p) = \sum_{i=1}^{n} w_i \, |p - x_i|$; we want to find a point p such that $f(p)$ is minimum. Let y be any point (real number) other than x. We show the optimality of the weighted median x by showing that $f(y) - f(x) \geq 0$. We examine separately the cases in which $y > x$ and $x > y$. For any x and y, we have

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^{n} w_i \, |y - x_i| - \sum_{i=1}^{n} w_i \, |x - x_i| \\ &= \sum_{i=1}^{n} w_i (|y - x_i| - |x - x_i|) \; . \end{aligned}$$

When $y > x$, we bound the quantity $|y - x_i| - |x - x_i|$ from below by examining three cases:

1. $x < y \leq x_i$: Here, $|x - y| + |y - x_i| = |x - x_i|$ and $|x - y| = y - x$, which imply that $|y - x_i| - |x - x_i| = -|x - y| = x - y$.
2. $x < x_i \leq y$: Here, $|y - x_i| \geq 0$ and $|x_i - x| \leq y - x$, which imply that $|y - x_i| - |x - x_i| \geq -(y - x) = x - y$.
3. $x_i \leq x < y$: Here, $|x - x_i| + |y - x| = |y - x_i|$ and $|y - x| = y - x$, which imply that $|y - x_i| - |x - x_i| = |y - x| = y - x$.

Separating out the first two cases, in which $x < x_i$, from the third case, in which $x \geq x_i$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^{n} w_i(|y - x_i| - |x - x_i|) \\ &\geq \sum_{x<x_i} w_i(x - y) + \sum_{x\geq x_i} w_i(y - x) \\ &= (y - x)\left(\sum_{x\geq x_i} w_i - \sum_{x<x_i} w_i\right) . \end{aligned}$$

The property that $\sum_{x_i<x} w_i < 1/2$ implies that $\sum_{x\geq x_i} w_i \geq 1/2$. This fact, combined with $y - x > 0$ and $\sum_{x<x_i} w_i \leq 1/2$, yields that $f(y) - f(x) \geq 0$.

When $x > y$, we again bound the quantity $|y - x_i| - |x - x_i|$ from below by examining three cases:

1. $x_i \leq y < x$: Here, $|y - x_i| + |x - y| = |x - x_i|$ and $|x - y| = x - y$, which imply that $|y - x_i| - |x - x_i| = -|x - y| = y - x$.
2. $y \leq x_i < x$: Here, $|y - x_i| \geq 0$ and $|x - x_i| \leq x - y$, which imply that $|y - x_i| - |x - x_i| \geq -(x - y) = y - x$.
3. $y < x \leq x_i$. Here, $|x - y| + |x - x_i| = |y - x_i|$ and $|x - y| = x - y$, which imply that $|y - x_i| - |x - x_i| = |x - y| = x - y$.

Separating out the first two cases, in which $x > x_i$, from the third case, in which $x \leq x_i$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^{n} w_i(|y - x_i| - |x - x_i|) \\ &\geq \sum_{x>x_i} w_i(y - x) + \sum_{x\leq x_i} w_i(x - y) \\ &= (x - y)\left(\sum_{x\leq x_i} w_i - \sum_{x>x_i} w_i\right) . \end{aligned}$$

The property that $\sum_{x_i>x} w_i \leq 1/2$ implies that $\sum_{x\leq x_i} w_i > 1/2$. This fact, combined with $x - y > 0$ and $\sum_{x>x_i} w_i < 1/2$, yields that $f(y) - f(x) > 0$.

e. We are given n 2-dimensional points $p_1, p_2, \ldots, p_n$, where each p_i is a pair of real numbers $p_i = (x_i, y_i)$, and positive weights $w_1, w_2, \ldots, w_n$. The goal is to find a point $p = (x, y)$ that minimizes the sum

$$f(x, y) = \sum_{i=1}^{n} w_i \left(|x - x_i| + |y - y_i|\right) .$$

We can express the cost function of the two variables, $f(x, y)$, as the sum of two functions of one variable each: $f(x, y) = g(x) + h(y)$, where $g(x) = \sum_{i=1}^{n} w_i |x - x_i|$, and $h(y) = \sum_{i=1}^{n} w_i |y - y_i|$. The goal of finding a point $p = (x, y)$ that minimizes the value of $f(x, y)$ can be achieved by treating each dimension independently, because g does not depend on y and h does not depend on x. Thus,

$$\min_{x,y} f(x, y) = \min_{x,y} \left(g(x) + h(y)\right)$$

$$\begin{aligned} &= \min_x \left(\min_y (g(x) + h(y)) \right) \\ &= \min_x \left(g(x) + \min_y h(y) \right) \\ &= \min_x g(x) + \min_y h(y) \ . \end{aligned}$$

Consequently, finding the best location in 2 dimensions can be done by finding the weighted median x_k of the x-coordinates and then finding the weighted median y_j of the y-coordinates. The point (x_k, y_j) is an optimal solution for the 2-dimensional post-office location problem.

Solution to Problem 9-3

a. Our algorithm relies on a particular property of SELECT: that not only does it return the ith smallest element, but that it also partitions the input array so that the first i positions contain the i smallest elements (though not necessarily in sorted order). To see that SELECT has this property, observe that there are only two ways in which returns a value: when $n = 1$, and when immediately after partitioning in step 4, it finds that there are exactly i elements on the low side of the partition.

Taking the hint from the book, here is our modified algorithm to select the ith smallest element of n elements. Whenever it is called with $i \geq n/2$, it just calls SELECT and returns its result; in this case, $U_i(n) = T(n)$.

When $i < n/2$, our modified algorithm works as follows. Assume that the input is in a subarray $A[p+1\,..\,p+n]$, and let $m = \lfloor n/2 \rfloor$. In the initial call, $p = 1$.

1. Divide the input as follows. If n is even, divide the input into two parts: $A[p+1\,..\,p+m]$ and $A[p+m+1\,..\,p+n]$. If n is odd, divide the input into three parts: $A[p+1\,..\,p+m]$, $A[p+m+1\,..\,p+n-1]$, and $A[p+n]$ as a leftover piece.
2. Compare $A[p+i]$ and $A[p+i+m]$ for $i = 1, 2, \ldots, m$, putting the smaller of the the two elements into $A[p+i+m]$ and the larger into $A[p+i]$.
3. Recursively find the ith smallest element in $A[p+m+1\,..\,p+n]$, but with an additional action performed by the partitioning procedure: whenever it exchanges $A[j]$ and $A[k]$ (where $p+m+1 \leq j, k \leq p+2m$), it also exchanges $A[j-m]$ and $A[k-m]$. The idea is that after recursively finding the ith smallest element in $A[p+m+1\,..\,p+n]$, the subarray $A[p+m+1\,..\,p+m+i]$ contains the i smallest elements that had been in $A[p+m+1\,..\,p+n]$ and the subarray $A[p+1\,..\,p+i]$ contains their larger counterparts, as found in step 1. The ith smallest element of $A[p+1\,..\,p+n]$ must be either one of the i smallest, as placed into $A[p+m+1\,..\,p+m+i]$, or it must be one of the larger counterparts, as placed into $A[p+1\,..\,p+i]$.
4. Collect the subarrays $A[p+1\,..\,p+i]$ and $A[p+m+1\,..\,p+m+i]$ into a single array $B[1\,..\,2i]$, call SELECT to find the ith smallest element of B, and return the result of this call to SELECT.

The number of comparisons in each step is as follows:

1. No comparisons.
2. $m = \lfloor n/2 \rfloor$ comparisons.
3. Since we recurse on $A[p+m+1\,..\,p+n]$, which has $\lceil n/2 \rceil$ elements, the number of comparisons is $U_i(\lceil n/2 \rceil)$.
4. Since we call SELECT on an array with $2i$ elements, the number of comparisons is $T(2i)$.

Thus, when $i < n/2$, the total number of comparisons is $\lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i)$.

b. We show by substitution that if $i < n/2$, then $U_i(n) = n + O(T(2i)\lg(n/i))$. In particular, we shall show that $U_i(n) \le n + cT(2i)\lg(n/i) - d(\lg\lg n)T(2i) = n + cT(2i)\lg n - cT(2i)\lg i - d(\lg\lg n)T(2i)$ for some positive constant c, some positive constant d to be chosen later, and $n \ge 4$. We have

$$\begin{aligned}
U_i(n) &= \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) \\
&\le \lfloor n/2 \rfloor + \lceil n/2 \rceil + cT(2i)\lg\lceil n/2 \rceil - cT(2i)\lg i \\
&\qquad - d(\lg\lg\lceil n/2 \rceil)T(2i) \\
&= n + cT(2i)\lg\lceil n/2 \rceil - cT(2i)\lg i - d(\lg\lg\lceil n/2 \rceil)T(2i) \\
&\le n + cT(2i)\lg(n/2+1) - cT(2i)\lg i - d(\lg\lg(n/2))T(2i) \\
&= n + cT(2i)\lg(n/2+1) - cT(2i)\lg i - d(\lg(\lg n - 1))T(2i) \\
&\le n + cT(2i)\lg n - cT(2i)\lg i - d(\lg\lg n)T(2i)
\end{aligned}$$

if $cT(2i)\lg(n/2+1) - d(\lg(\lg n - 1))T(2i) \le cT(2i)\lg n - d(\lg\lg n)T(2i)$. Simple algebraic manipulations gives the following sequence of equivalent conditions:

$$cT(2i)\lg(n/2+1) - d(\lg(\lg n - 1))T(2i) \le cT(2i)\lg n - d(\lg\lg n)T(2i)$$

$$c\lg(n/2+1) - d(\lg(\lg n - 1)) \le c\lg n - d(\lg\lg n)$$

$$c(\lg(n/2+1) - \lg n) \le d(\lg(\lg n - 1) - \lg\lg n)$$

$$c\left(\lg\frac{n/2+1}{n}\right) \le d\lg\frac{\lg n - 1}{\lg n}$$

$$c\left(\lg\left(\frac{1}{2} + \frac{1}{n}\right)\right) \le d\lg\frac{\lg n - 1}{\lg n}$$

Observe that $1/2 + 1/n$ decreases as n increases, but $(\lg n - 1)/\lg n$ increases as n increases. When $n = 4$, we have $1/2 + 1/n = 3/4$ and $(\lg n - 1)/\lg n = 1/2$. Thus, we just need to choose d such that $c\lg(3/4) \le d\lg(1/2)$ or, equivalently, $c\lg(3/4) \le -d$. Multiplying both sides by -1, we get $d \le -c\lg(3/4) = c\lg(4/3)$. Thus, any value of d that is at most $c\lg(4/3)$ suffices.

c. When i is a constant, $T(2i) = O(1)$ and $\lg(n/i) = \lg n - \lg i = O(\lg n)$. Thus, when i is a constant less than $n/2$, we have that

$$\begin{aligned}
U_i(n) &= n + O(T(2i)\lg(n/i)) \\
&= n + O(O(1)\cdot O(\lg n)) \\
&= n + O(\lg n)\,.
\end{aligned}$$

d. Suppose that $i = n/k$ for $k \ge 2$. Then $i \le n/2$. If $k > 2$, then $i < n/2$, and we have

$$U_i(n) = n + O(T(2i)\lg(n/i))$$

$$\begin{aligned} &= n + O(T(2n/k)\lg(n/(n/k)) \\ &= n + O(T(2n/k)\lg k)\ . \end{aligned}$$

If $k = 2$, then $n = 2i$ and $\lg k = 1$. We have

$$\begin{aligned} U_i(n) &= T(n) \\ &= n + (T(n) - n) \\ &\le n + (T(2i) - n) \\ &= n + (T(2n/k) - n) \\ &= n + (T(2n/k)\lg k - n) \\ &= n + O(T(2n/k)\lg k)\ . \end{aligned}$$

Lecture Notes for Chapter 11: Hash Tables

Chapter 11 overview

Many applications require a dynamic set that supports only the ***dictionary operations*** INSERT, SEARCH, and DELETE. Example: a symbol table in a compiler.

A hash table is effective for implementing a dictionary.

- The expected time to search for an element in a hash table is $O(1)$, under some reasonable assumptions.
- Worst-case search time is $\Theta(n)$, however.

A hash table is a generalization of an ordinary array.

- With an ordinary array, we store the element whose key is k in position k of the array.
- Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called ***direct addressing***.
- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.

We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.

- Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
- A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
- Given a key k, don't just use k as the index into the array.
- Instead, compute a function of k, and use that value to index into the array. We call this function a ***hash function***.

Issues that we'll explore in hash tables:

- How to compute hash functions. We'll look at the multiplication and division methods.
- What to do when the hash function maps multiple keys to the same table entry. We'll look at chaining and open addressing.

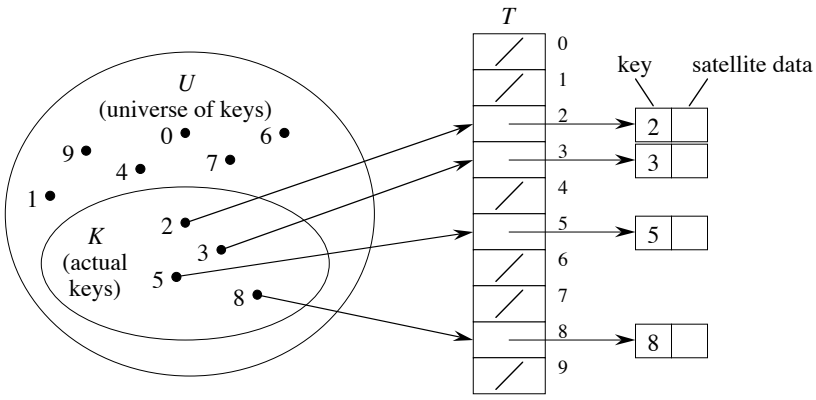
Direct-address tables

Scenario:

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \ldots, m-1\}$ where m isn't too large.
- No two elements have the same key.

Represent by a ***direct-address table***, or array, $T[0 \ldots m-1]$:

- Each ***slot***, or position, corresponds to a key in U.
- If there's an element x with key k, then $T[k]$ contains a pointer to x.
- Otherwise, $T[k]$ is empty, represented by NIL.



Dictionary operations are trivial and take $O(1)$ time each:

DIRECT-ADDRESS-SEARCH(T, k)
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
$T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
$T[key[x]] \leftarrow$ NIL

Hash tables

The problem with direct addressing is if the universe U is large, storing a table of size $|U|$ may be impractical or impossible.

Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.

- When K is much smaller than U, a hash table requires much less space than a direct-address table.
- Can reduce storage requirements to $\Theta(|K|)$.
- Can still get $O(1)$ search time, but in the *average case*, not the *worst case*.

Idea: Instead of storing an element with key k in slot k, use a function h and store the element in slot $h(k)$.

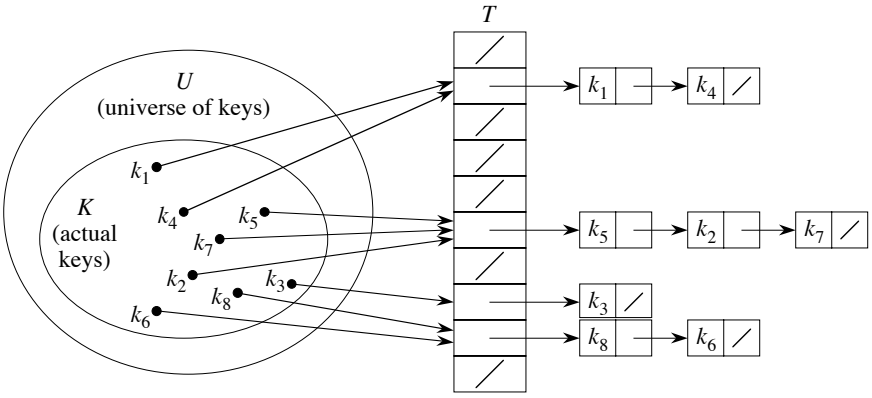
- We call h a ***hash function***.
- $h : U \to \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k ***hashes*** to slot $h(k)$.

Collisions: When two or more keys hash to the same slot.

- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing.
- Chaining is usually better than open addressing. We'll examine both.

Collision resolution by chaining

Put all elements that hash to the same slot into a linked list.



[This figure shows singly linked lists. If we want to delete elements, it's better to use doubly linked lists.]

- Slot j contains a pointer to the head of the list of all stored elements that hash to j *[or to the sentinel if using a circular, doubly linked list with a sentinel]* ,
- If there are no such elements, slot j contains NIL.

How to implement dictionary operations with chaining:

- ***Insertion:***

  CHAINED-HASH-INSERT(T, x)
  insert x at the head of list $T[h(key[x])]$

  - Worst-case running time is $O(1)$.
  - Assumes that the element being inserted isn't already in the list.
  - It would take an additional search to check if it was already inserted.

- ***Search:***

  CHAINED-HASH-SEARCH(T, k)
  search for an element with key k in list $T[h(k)]$

  Running time is proportional to the length of the list of elements in slot $h(k)$.

- ***Deletion:***

  CHAINED-HASH-DELETE(T, x)
  delete x from the list $T[h(key[x])]$

  - Given pointer x to the element to delete, so no search is needed to find this element.
  - Worst-case running time is $O(1)$ time if the lists are doubly linked.
  - If the lists are singly linked, then deletion takes as long as searching, because we must find x's predecessor in its list in order to correctly update *next* pointers.

Analysis of hashing with chaining

Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?

- Analysis is in terms of the ***load factor*** $\alpha = n/m$:
  - $n =$ # of elements in the table.
  - $m =$ # of slots in the table $=$ # of (possibly empty) linked lists.
  - Load factor is average number of elements per linked list.
  - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.
- Worst case is when all n keys hash to the same slot $\Rightarrow$ get a single list of length n $\Rightarrow$ worst-case time to search is $\Theta(n)$, plus time to compute hash function.
- Average case depends on how well the hash function distributes the keys among the slots.

We focus on average-case performance of hashing with chaining.

- Assume ***simple uniform hashing***: any given element is equally likely to hash into any of the m slots.

- For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ by n_j. Then $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Average value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.
- Assume that we can compute the hash function in $O(1)$ time, so that the time required to search for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.

We consider two cases:

- If the hash table contains no element with key k, then the search is unsuccessful.
- If the hash table does contain an element with key k, then the search is successful.

[In the theorem statements that follow, we omit the assumptions that we're resolving collisions by chaining and that simple uniform hashing applies.]

Unsuccessful search:

Theorem

An unsuccessful search takes expected time $\Theta(1 + \alpha)$.

Proof Simple uniform hashing $\Rightarrow$ any key not already in the table is equally likely to hash to any of the m slots.

To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$. This list has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Therefore, the expected number of elements examined in an unsuccessful search is α.

Adding in the time to compute the hash function, the total time required is $\Theta(1 + \alpha)$. ■

Successful search:

- The expected time for a successful search is also $\Theta(1 + \alpha)$.
- The circumstances are slightly different from an unsuccessful search.
- The probability that each list is searched is proportional to the number of elements it contains.

Theorem

A successful search takes expected time $\Theta(1 + \alpha)$.

Proof Assume that the element x being searched for is equally likely to be any of the n elements stored in the table.

The number of elements examined during a successful search for x is 1 more than the number of elements that appear before x in x's list. These are the elements inserted *after* x was inserted (because we insert at the head of the list).

So we need to find the average, over the n elements x in the table, of how many elements were inserted into x's list after x was inserted.

For $i = 1, 2, \ldots, n$, let x_i be the ith element inserted into the table, and let $k_i = key[x_i]$.

For all i and j, define indicator random variable $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$.

Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m \Rightarrow \mathrm{E}[X_{ij}] = 1/m$ (by Lemma 5.1).

Expected number of elements examined in a successful search is

$$\begin{aligned}
&\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&\quad= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) \quad \text{(linearity of expectation)} \\
&\quad= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&\quad= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&\quad= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&\quad= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{(equation (A.1))} \\
&\quad= 1+\frac{n-1}{2m} \\
&\quad= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}$$

Adding in the time for computing the hash function, we get that the expected total time for a successful search is $\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha)$.

Alternative analysis, using indicator random variables even more:

For each slot l and for each pair of keys k_i and k_j, define the indicator random variable $X_{ijl} = \mathrm{I}\{\text{the search is for } x_i, h(k_i) = l, \text{ and } h(k_j) = l\}$. $X_{ijl} = 1$ when keys k_i and k_j collide at slot l and when we are searching for x_i.

Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = l\} = 1/m$ and $\Pr\{h(k_j) = l\} = 1/m$. Also have $\Pr\{\text{the search is for } x_i\} = 1/n$. These events are all independent $\Rightarrow \Pr\{X_{ijl} = 1\} = 1/nm^2 \Rightarrow \mathrm{E}[X_{ijl}] = 1/nm^2$ (by Lemma 5.1).

Define, for each element x_j, the indicator random variable

$Y_j = \mathrm{I}\{x_j \text{ appears in a list prior to the element being searched for}\}$.

$Y_j = 1$ if and only if there is some slot l that has both elements x_i and x_j in its list, and also $i < j$ (so that x_i appears after x_j in the list). Therefore,

$$Y_j = \sum_{i=1}^{j-1}\sum_{l=0}^{m-1}X_{ijl}\,.$$

One final random variable: Z, which counts how many elements appear in the list prior to the element being searched for: $Z = \sum_{j=1}^{n}Y_j$. We must count the element

being searched for as well as all those preceding it in its list $\Rightarrow$ compute $\mathrm{E}[Z+1]$:

$$\begin{aligned}
\mathrm{E}[Z+1] &= \mathrm{E}\left[1+\sum_{j=1}^{n} Y_j\right] \\
&= 1+\mathrm{E}\left[\sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{l=0}^{m-1} X_{ijl}\right] \quad \text{(linearity of expectation)} \\
&= 1+\sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{l=0}^{m-1}\mathrm{E}[X_{ijl}] \quad \text{(linearity of expectation)} \\
&= 1+\sum_{j=1}^{n}\sum_{i=1}^{j-1}\sum_{l=0}^{m-1}\frac{1}{nm^2} \\
&= 1+\binom{n}{2}\cdot m\cdot\frac{1}{nm^2} \\
&= 1+\frac{n(n-1)}{2}\cdot\frac{1}{nm} \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{n}{2m}-\frac{1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}$$

Adding in the time for computing the hash function, we get that the expected total time for a successful search is $\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha)$. ■

Interpretation: If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$, which means that searching takes constant time on average.

Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, all dictionary operations take $O(1)$ time on average.

Hash functions

We discuss some issues regarding hash-function design and present schemes for hash function creation.

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of simple uniform hashing.
- In practice, it's not possible to satisfy this assumption, since we don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.

Keys as natural numbers

- Hash functions assume that the keys are natural numbers.
- When they're not, have to interpret them as natural numbers.
- ***Example:*** Interpret a character string as an integer expressed in some radix notation. Suppose the string is `CLRS`:
  - ASCII values: `C` = 67, `L` = 76, `R` = 82, `S` = 83.
  - There are 128 basic ASCII values.
  - So interpret `CLRS` as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) =$ 141,764,947.

Division method

$h(k) = k \bmod m$.

Example: $m = 20$ and $k = 91 \Rightarrow h(k) = 11$.

Advantage: Fast, since requires just one division operation.

Disadvantage: Have to avoid certain values of m:

- Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
- If k is a character string interpreted in radix 2^p (as in `CLRS` example), then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value (Exercise 11.3-3).

Good choice for m: A prime not too close to an exact power of 2.

Multiplication method

1. Choose constant A in the range $0 < A < 1$.
2. Multiply key k by A.
3. Extract the fractional part of kA.
4. Multiply the fractional part by m.
5. Take the floor of the result.

Put another way, $h(k) = \lfloor m\,(k\,A \bmod 1) \rfloor$, where $k\,A \bmod 1 = kA - \lfloor kA \rfloor =$ fractional part of kA.

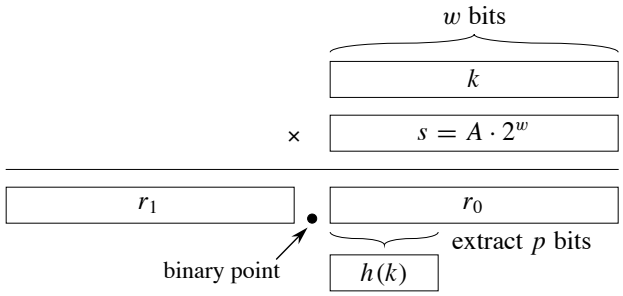
Disadvantage: Slower than division method.

Advantage: Value of m is not critical.

(Relatively) easy implementation:

- Choose $m = 2^p$ for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let s be an integer in the range $0 < s < 2^w$. (s takes w bits.)

- Restrict A to be of the form $s/2^w$.



- Multiply k by s.
- Since we're multiplying two w-bit words, the result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($k\ A \bmod 1 = kA - \lfloor kA \rfloor$). Think of the "binary point" (analog of decimal point, but for binary representation) as being between r_1 and r_0. Since we don't care about the integer part of kA, we can forget about r_1 and just use r_0.
- Since we want $\lfloor m\ (k\ A \bmod 1) \rfloor$, we could get that value by shifting r_0 to the left by $p = \lg m$ bits and then taking the p bits that were shifted to the left of the binary point.
- We don't need to shift. The p bits that would have been shifted to the left of the binary point are the p most significant bits of r_0. So we can just take these bits after having formed r_0 by multiplying k by s.
- ***Example:*** $m = 8$ (implies $p = 3$), $w = 5$, $k = 21$. Must have $0 < s < 2^5$; choose $s = 13 \Rightarrow A = 13/32$.
  - Using just the formula to compute $h(k)$: $kA = 21 \cdot 13/32 = 273/32 = 8\frac{17}{32}$ $\Rightarrow k\ A \bmod 1 = 17/32 \Rightarrow m\ (k\ A \bmod 1) = 8 \cdot 17/32 = 17/4 = 4\frac{1}{4} \Rightarrow$ $\lfloor m\ (k\ A \bmod 1) \rfloor = 4$, so that $h(k) = 4$.
  - Using the implementation: $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17 \Rightarrow r_1 = 8$, $r_0 = 17$. Written in $w = 5$ bits, $r_0 = 10001$. Take the $p = 3$ most significant bits of r_0, get 100 in binary, or 4 in decimal, so that $h(k) = 4$.

How to choose A:

- The multiplication method works with any legal value of A.
- But it works better with some values than with others, depending on the keys being hashed.
- Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

Universal hashing

[We just touch on universal hashing in these notes. See the book for a full treatment.]

Suppose that a malicious adversary, who gets to choose the keys to be hashed, has seen your hashing program and knows the hash function in advance. Then he could choose keys that all hash to the same slot, giving worst-case behavior.

One way to defeat the adversary is to use a different hash function each time. You choose one at random at the beginning of your program. Unless the adversary knows how you'll be randomly choosing which hash function to use, he cannot intentionally defeat you.

Just because we choose a hash function randomly, that doesn't mean it's a good hash function. What we want is to randomly choose a single hash function from a set of good candidates.

Consider a finite collection $\mathcal{H}$ of hash functions that map a universe U of keys into the range $\{0, 1, \ldots, m-1\}$. $\mathcal{H}$ is ***universal*** if for each pair of keys $k, l \in U$, where $k \neq l$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is $\leq |\mathcal{H}|/m$.

Put another way, $\mathcal{H}$ is universal if, with a hash function h chosen randomly from $\mathcal{H}$, the probability of a collision between two different keys is no more than than $1/m$ chance of just choosing two slots randomly and independently.

Why are universal hash functions good?

- They give good hashing behavior:

  Theorem

  Using chaining and universal hashing on key k:

  - If k is not in the table, the expected length $\mathrm{E}[n_{h(k)}]$ of the list that k hashes to is $\leq \alpha$.
  - If k is in the table, the expected length $\mathrm{E}[n_{h(k)}]$ of the list that holds k is $\leq 1 + \alpha$.

  Corollary

  Using chaining and universal hashing, the expected time for each SEARCH operation is $O(1)$.

- They are easy to design.

[See book for details of behavior and design of a universal class of hash functions.]

Open addressing

An alternative to chaining for handling collisions.

Idea:

- Store all keys in the hash table itself.
- Each slot contains either a key or NIL.
- To search for key k:
  - Compute $h(k)$ and examine slot $h(k)$. Examining a slot is known as a ***probe***.
  - If slot $h(k)$ contains key k, the search is successful. If this slot contains NIL, the search is unsuccessful.
  - There's a third possibility: slot $h(k)$ contains a key that is not k. We compute the index of some other slot, based on k and on which probe (count from 0: 0th, 1st, 2nd, etc.) we're on.

  - Keep probing until we either find key k (successful search) or we find a slot holding NIL (unsuccessful search).

- We need the sequence of slots probed to be a permutation of the slot numbers $\langle 0, 1, \ldots, m-1 \rangle$ (so that we examine all slots if we have to, and so that we don't examine any slot more than once).
- Thus, the hash function is $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$.
- The requirement that the sequence of slots be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ is equivalent to requiring that the ***probe sequence*** $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- To insert, act as though we're searching, and insert at the first NIL slot we find.

Pseudocode for searching:

```
HASH-SEARCH(T, k)
i ← 0
repeat j ← h(k, i)
       if T[j] = k
         then return j
       i ← i + 1
  until T[j] = NIL or i = m
return NIL
```

HASH-SEARCH returns the index of a slot containing key k, or NIL if the search is unsuccessful.

Pseudocode for insertion:

```
HASH-INSERT(T, k)
i ← 0
repeat j ← h(k, i)
       if T[j] = NIL
         then T[j] ← k
              return j
         else i ← i + 1
  until i = m
error "hash table overflow"
```

HASH-INSERT returns the number of the slot that gets key k, or it flags a "hash table overflow" error if there is no empty slot in which to put key k.

Deletion: Cannot just put NIL into the slot containing the key we want to delete.

- Suppose we want to delete key k in slot j.
- And suppose that sometime after inserting key k, we were inserting key k', and during this insertion we had probed slot j (which contained key k).
- And suppose we then deleted key k by storing NIL into slot j.
- And then we search for key k'.

- During the search, we would probe slot j *before* probing the slot into which key k' was eventually stored.
- Thus, the search would be unsuccessful, even though key k' is in the table.

Solution: Use a special value DELETED instead of NIL when marking a slot as empty during deletion.

- Search should treat DELETED as though the slot holds a key that does not match the one being searched for.
- Insertion should treat DELETED as though the slot were empty, so that it can be reused.

The disadvantage of using DELETED is that now search time is no longer dependent on the load factor α.

How to compute probe sequences

The ideal situation is ***uniform hashing***: each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence. (This generalizes simple uniform hashing for a hash function that produces a whole probe sequence rather than just a single number.)

It's hard to implement true uniform hashing, so we approximate it with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

None of these techniques can produce all $m!$ probe sequences. They will make use of ***auxiliary hash functions***, which map $U \to \{0, 1, \ldots, m-1\}$.

Linear probing: Given auxiliary hash function h', the probe sequence starts at slot $h'(k)$ and continues sequentially through the table, wrapping after slot $m-1$ to slot 0.

Given key k and probe number i $(0 \le i < m)$, $h(k, i) = (h'(k) + i) \bmod m$.

The initial probe determines the entire sequence $\Rightarrow$ only m possible sequences.

Linear probing suffers from ***primary clustering***: long runs of occupied sequences build up. And long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$. Result is that the average search and insertion times increase.

Quadratic probing: As in linear probing, the probe sequence starts at $h'(k)$. Unlike linear probing, it jumps around in the table according to a quadratic function of the probe number: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $c_1, c_2 \ne 0$ are constants.

Must constrain c_1, c_2, and m in order to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$. (Problem 11-3 explores one way to implement quadratic probing.)

Can get ***secondary clustering***: if two distinct keys have the same h' value, then they have the same probe sequence.

Double hashing: Use two auxiliary hash functions, h_1 and h_2. h_1 gives the initial probe, and h_2 gives the remaining probes: $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$.

Must have $h_2(k)$ be relatively prime to m (no factors in common other than 1) in order to guarantee that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

- Could choose m to be a power of 2 and h_2 to always produce an odd number > 1.
- Could let m be prime and have $1 < h_2(k) < m$.

$\Theta(m^2)$ different probe sequences, since each possible combination of $h_1(k)$ and $h_2(k)$ gives a different probe sequence.

Analysis of open-address hashing

Assumptions:

- Analysis is in terms of load factor α. We will assume that the table never completely fills, so we always have $0 \le n < m \Rightarrow 0 \le \alpha < 1$.
- Assume uniform hashing.
- No deletion.
- In a successful search, each key is equally likely to be searched for.

Theorem
The expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Proof Since the search is unsuccessful, every probe is to an occupied slot, except for the last probe, which is to an empty slot.

Define random variable $X = \#$ of probes made in an unsuccessful search.

Define events A_i, for $i = 1, 2, \ldots$, to be the event that there is an ith probe and that it's to an occupied slot.

$X \ge i$ if and only if probes $1, 2, \ldots, i-1$ are made and are to occupied slots $\Rightarrow$ $\Pr\{X \ge i\} = \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$.

By Exercise C.2-6,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\quad \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} \ .\end{aligned}$$

Claim
$\Pr\{A_j \mid A_1 \cap A_2 \cap \cdots \cap A_{j-1}\} = (n-j+1)/(m-j+1)$. Boundary case: $j = 1$ $\Rightarrow \Pr\{A_1\} = n/m$.

Proof For the boundary case $j = 1$, there are n stored keys and m slots, so the probability that the first probe is to an occupied slot is n/m.

Given that $j-1$ probes were made, all to occupied slots, the assumption of uniform hashing says that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, which in turn implies that the next probe is to a slot that we have not yet probed. There are $m-j+1$ slots remaining, $n-j+1$ of which are occupied. Thus, the probability that the jth probe is to an occupied slot is $(n-j+1)/(m-j+1)$. ■ (claim)

Using this claim,

$$\Pr\{X \geq i\} = \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}}_{i-1 \text{ factors}} .$$

$n < m \Rightarrow (n-j)/(m-j) \leq n/m$ for $j \geq 0$, which implies

$$\begin{aligned} \Pr\{X \geq i\} &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} . \end{aligned}$$

By equation (C.24),

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} \qquad \text{(equation (A.6))} . \end{aligned}$$

■ (theorem)

Interpretation: If α is constant, an unsuccessful search takes $O(1)$ time.

- If $\alpha = 0.5$, then an unsuccessful search takes an average of $1/(1-0.5) = 2$ probes.
- If $\alpha = 0.9$, takes an average of $1/(1-0.9) = 10$ probes.

Corollary

The expected number of probes to insert is at most $1/(1-\alpha)$.

Proof Since there is no deletion, insertion uses the same probe sequence as an unsuccessful search. ■

Theorem

The expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

Proof A successful search for key k follows the same probe sequence as when key k was inserted.

By the previous corollary, if k was the $(i+1)$st key inserted, then α equaled i/m at the time. Thus, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$.

That was assuming that k was the $(i+1)$st key inserted. We need to average over all n keys:

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) , \end{aligned}$$

where $H_i = \sum_{j=1}^{i} 1/j$ is the ith harmonic number.

Simplify by using the technique of bounding a summation by an integral:

$$\begin{aligned}
\frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} 1/k \\
&\le \frac{1}{\alpha} \int_{m-n}^{m} (1/x)\, dx \quad \text{(inequality (A.12))} \\
&= \frac{1}{\alpha} \ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}
\end{aligned}$$

■ (theorem)

Solutions for Chapter 11: Hash Tables

Solution to Exercise 11.1-4

We denote the huge array by T and, taking the hint from the book, we also have a stack implemented by an array S. The size of S equals the number of keys actually stored, so that S should be allocated at the dictionary's maximum size. The stack has an attribute $top[S]$, so that only entries $S[1\,..\,top[S]]$ are valid.

The idea of this scheme is that entries of T and S validate each other. If key k is actually stored in T, then $T[k]$ contains the index, say j, of a valid entry in S, and $S[j]$ contains the value k. Let us call this situation, in which $1 \le T[k] \le top[S]$, $S[T[k]] = k$, and $T[S[j]] = j$, a ***validating cycle***.

Assuming that we also need to store pointers to objects in our direct-address table, we can store them in an array that is parallel to either T or S. Since S is smaller than T, we'll use an array S', allocated to be the same size as S, for these pointers. Thus, if the dictionary contains an object x with key k, then there is a validating cycle and $S'[T[k]]$ points to x.

The operations on the dictionary work as follows:

- Initialization: Simply set $top[S] = 0$, so that there are no valid entries in the stack.
- SEARCH: Given key k, we check whether we have a validating cycle, i.e., whether $1 \le T[k] \le top[S]$ and $S[T[k]] = k$. If so, we return $S'[T[k]]$, and otherwise we return NIL.
- INSERT: To insert object x with key k, assuming that this object is not already in the dictionary, we increment $top[S]$, set $S[top[S]] \leftarrow k$, set $S'[top[S]] \leftarrow x$, and set $T[k] \leftarrow top[S]$.
- DELETE: To delete object x with key k, assuming that this object is in the dictionary, we need to break the validating cycle. The trick is to also ensure that we don't leave a "hole" in the stack, and we solve this problem by moving the top entry of the stack into the position that we are vacating—and then fixing up *that* entry's validating cycle. That is, we execute the following sequence of assignments:

$S[T[k]] \leftarrow S[top[S]]$
$S'[T[k]] \leftarrow S'[top[S]]$
$T[S[T[k]]] \leftarrow T[k]$
$T[k] \leftarrow 0$
$top[S] \leftarrow top[S] - 1$

Each of these operations—initialization, SEARCH, INSERT, and DELETE—takes $O(1)$ time.

Solution to Exercise 11.2-1

For each pair of keys k, l, where $k \neq l$, define the indicator random variable $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Since we assume simple uniform hashing, $\Pr\{X_{kl} = 1\} = \Pr\{h(k) = h(l)\} = 1/m$, and so $\mathrm{E}[X_{kl}] = 1/m$.

Now define the random variable Y to be the total number of collisions, so that $Y = \sum_{k \neq l} X_{kl}$. The expected number of collisions is

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{k \neq l} X_{kl}\right] \\
&= \sum_{k \neq l} \mathrm{E}[X_{kl}] \qquad \text{(linearity of expectation)} \\
&= \binom{n}{2}\frac{1}{m} \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{m} \\
&= \frac{n(n-1)}{2m} .
\end{aligned}$$

Solution to Exercise 11.2-4

The flag in each slot will indicate whether the slot is free.

- A free slot is in the free list, a doubly linked list of all free slots in the table. The slot thus contains two pointers.
- A used slot contains an element and a pointer (possibly NIL) to the next element that hashes to this slot. (Of course, that pointer points to another slot in the table.)

Operations

- ***Insertion:***
  - If the element hashes to a free slot, just remove the slot from the free list and store the element there (with a NIL pointer). The free list must be doubly linked in order for this deletion to run in $O(1)$ time.

- If the element hashes to a used slot j, check whether the element x already there "belongs" there (its key also hashes to slot j).
  - If so, add the new element to the chain of elements in this slot. To do so, allocate a free slot (e.g., take the head of the free list) for the new element and put this new slot at the head of the list pointed to by the hashed-to slot (j).
  - If not, E is part of another slot's chain. Move it to a new slot by allocating one from the free list, copying the old slot's (j's) contents (element x and pointer) to the new slot, and updating the pointer in the slot that pointed to j to point to the new slot. Then insert the new element in the now-empty slot as usual.
    To update the pointer to j, it is necessary to find it by searching the chain of elements starting in the slot x hashes to.
- ***Deletion:*** Let j be the slot the element x to be deleted hashes to.
  - If x is the only element in j (j doesn't point to any other entries), just free the slot, returning it to the head of the free list.
  - If x is in j but there's a pointer to a chain of other elements, move the first pointed-to entry to slot j and free the slot it was in.
  - If x is found by following a pointer from j, just free x's slot and splice it out of the chain (i.e., update the slot that pointed to x to point to x's successor).
- ***Searching:*** Check the slot the key hashes to, and if that is not the desired element, follow the chain of pointers from the slot.

  All the operations take expected $O(1)$ times for the same reason they do with the version in the book: The expected time to search the chains is $O(1 + \alpha)$ regardless of where the chains are stored, and the fact that all the elements are stored in the table means that $\alpha \leq 1$. If the free list were singly linked, then operations that involved removing an arbitrary slot from the free list would not run in $O(1)$ time.

Solution to Exercise 11.3-3

First, we observe that we can generate any permutation by a sequence of interchanges of pairs of characters. One can prove this property formally, but informally, consider that both heapsort and quicksort work by interchanging pairs of elements and that they have to be able to produce any permutation of their input array. Thus, it suffices to show that if string x can be derived from string y by interchanging a single pair of characters, then x and y hash to the same value.

Let us denote the ith character in x by x_i, and similarly for y. The interpretation of x in radix 2^p is $\sum_{i=0}^{n-1} x_i 2^{ip}$, and so $h(x) = \left(\sum_{i=0}^{n-1} x_i 2^{ip}\right) \bmod (2^p - 1)$. Similarly, $h(y) = \left(\sum_{i=0}^{n-1} y_i 2^{ip}\right) \bmod (2^p - 1)$.

Suppose that x and y are identical strings of n characters except that the characters in positions a and b are interchanged: $x_a = y_b$ and $y_a = x_b$. Without loss of generality, let $a > b$. We have

$$h(x) - h(y) = \left(\sum_{i=0}^{n-1} x_i 2^{ip}\right) \bmod (2^p - 1) - \left(\sum_{i=0}^{n-1} y_i 2^{ip}\right) \bmod (2^p - 1) .$$

Since $0 \le h(x), h(y) < 2^p - 1$, we have that $-(2^p - 1) < h(x) - h(y) < 2^p - 1$. If we show that $(h(x) - h(y)) \bmod (2^p - 1) = 0$, then $h(x) = h(y)$.

Since the sums in the hash functions are the same except for indices a and b, we have

$$\begin{aligned}
&(h(x) - h(y)) \bmod (2^p - 1) \\
&\quad = ((x_a 2^{ap} + x_b 2^{bp}) - (y_a 2^{ap} + y_b 2^{bp})) \bmod (2^p - 1) \\
&\quad = ((x_a 2^{ap} + x_b 2^{bp}) - (x_b 2^{ap} + x_a 2^{bp})) \bmod (2^p - 1) \\
&\quad = ((x_a - x_b)2^{ap} - (x_a - x_b)2^{bp}) \bmod (2^p - 1) \\
&\quad = ((x_a - x_b)(2^{ap} - 2^{bp})) \bmod (2^p - 1) \\
&\quad = ((x_a - x_b)2^{bp}(2^{(a-b)p} - 1)) \bmod (2^p - 1) .
\end{aligned}$$

By equation (A.5),

$$\sum_{i=0}^{a-b-1} 2^{pi} = \frac{2^{(a-b)p} - 1}{2^p - 1} ,$$

and multiplying both sides by $2^p - 1$, we get $2^{(a-b)p} - 1 = \left(\sum_{i=0}^{a-b-1} 2^{pi}\right)(2^p - 1)$. Thus,

$$\begin{aligned}
&(h(x) - h(y)) \bmod (2^p - 1) \\
&\quad = \left((x_a - x_b)2^{bp}\left(\sum_{i=0}^{a-b-1} 2^{pi}\right)(2^p - 1)\right) \bmod (2^p - 1) \\
&\quad = 0 ,
\end{aligned}$$

since one of the factors is $2^p - 1$.

We have shown that $(h(x) - h(y)) \bmod (2^p - 1) = 0$, and so $h(x) = h(y)$.

Solution to Exercise 11.3-5

Let $b = |B|$ and $u = |U|$. We start by showing that the total number of collisions is minimized by a hash function that maps u/b elements of U to each of the b values in B. For a given hash function, let u_j be the number of elements that map to $j \in B$. We have $u = \sum_{j \in B} u_j$. We also have that the number of collisions for a given value of $j \in B$ is $\binom{u_j}{2} = u_j(u_j - 1)/2$.

Lemma

The total number of collisions is minimized when $u_j = u/b$ for each $j \in B$.

Proof If $u_j \le u/b$, let us call j ***underloaded***, and if $u_j \ge u/b$, let us call j ***overloaded***. Consider an unbalanced situation in which $u_j \ne u/b$ for at least one value $j \in B$. We can think of converting a balanced situation in which all u_j equal u/b into the unbalanced situation by repeatedly moving an element that maps to an underloaded value to map instead to an overloaded value. (If you think

of the values of B as representing buckets, we are repeatedly moving elements from buckets containing at most u/b elements to buckets containing at least u/b elements.)

We now show that each such move increases the number of collisions, so that all the moves together must increase the number of collisions. Suppose that we move an element from an underloaded value j to an overloaded value k, and we leave all other elements alone. Because j is underloaded and k is overloaded, $u_j \le u/b \le u_k$. Considering just the collisions for values j and k, we have $u_j(u_j - 1)/2 + u_k(u_k - 1)/2$ collisions before the move and $(u_j - 1)(u_j - 2)/2 + (u_k + 1)u_k/2$ collisions afterward. We wish to show that $u_j(u_j - 1)/2 + u_k(u_k - 1)/2 < (u_j - 1)(u_j - 2)/2 + (u_k + 1)u_k/2$. We have the following sequence of equivalent inequalities:

$$\begin{aligned}
u_j &< u_k + 1 \\
2u_j &< 2u_k + 2 \\
-u_k &< u_k - 2u_j + 2 \\
u_j^2 - u_j + u_k^2 - u_k &< u_j^2 - 3u_j + 2 + u_k^2 + u_k \\
u_j(u_j - 1) + u_k(u_k - 1) &< (u_j - 1)(u_j - 2) + (u_k + 1)u_k \\
u_j(u_j - 1)/2 + u_k(u_k - 1)/2 &< (u_j - 1)(u_j - 2)/2 + (u_k + 1)u_k/2 \; .
\end{aligned}$$

Thus, each move increases the number of collisions. We conclude that the number of collisions is minimized when $u_j = u/b$ for each $j \in B$. ■

By the above lemma, for any hash function, the total number of collisions must be at least $b(u/b)(u/b - 1)/2$. The number of pairs of distinct elements is $\binom{u}{2} = u(u - 1)/2$. Thus, the number of collisions per pair of distinct elements must be at least

$$\begin{aligned}
\frac{b(u/b)(u/b - 1)/2}{u(u - 1)/2} &= \frac{u/b - 1}{u - 1} \\
&> \frac{u/b - 1}{u} \\
&= \frac{1}{b} - \frac{1}{u} \; .
\end{aligned}$$

Thus, the bound ϵ on the probability of a collision for any pair of distinct elements can be no less than $1/b - 1/u = 1/|B| - 1/|U|$.

Solution to Problem 11-1

a. Since we assume uniform hashing, we can use the same observation as is used in Corollary 11.7: that inserting a key entails an unsuccessful search followed by placing the key into the first empty slot found. As in the proof of Theorem 11.6, if we let X be the random variable denoting the number of probes in an unsuccessful search, then $\Pr\{X \ge i\} \le \alpha^{i-1}$. Since $n \le m/2$, we have $\alpha \le 1/2$. Letting $i = k + 1$, we have $\Pr\{X > k\} = \Pr\{X \ge k + 1\} \le (1/2)^{(k+1)-1} = 2^{-k}$.

b. Substituting $k = 2\lg n$ into the statement of part (a) yields that the probability that the ith insertion requires more than $k = 2\lg n$ probes is at most $2^{-2\lg n} = (2^{\lg n})^{-2} = n^{-2} = 1/n^2$.

c. Let the event A be $X > 2\lg n$, and for $i = 1, 2, \ldots, n$, let the event A_i be $X_i > 2\lg n$. In part (b), we showed that $\Pr\{A_i\} \le 1/n^2$ for $i = 1, 2, \ldots, n$. From how we defined these events, $A = A_1 \cup A_2 \cup \cdots \cup A_n$. Using Boole's inequality, (C.18), we have

$$\begin{aligned}\Pr\{A\} &\le \Pr\{A_1\} + \Pr\{A_2\} + \cdots + \Pr\{A_n\} \\ &\le n \cdot \frac{1}{n^2} \\ &= 1/n \ .\end{aligned}$$

d. We use the definition of expectation and break the sum into two parts:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{k=1}^{n} k \cdot \Pr\{X = k\} \\ &= \sum_{k=1}^{\lceil 2\lg n\rceil} k \cdot \Pr\{X = k\} + \sum_{k=\lceil 2\lg n\rceil+1}^{n} k \cdot \Pr\{X = k\} \\ &\le \sum_{k=1}^{\lceil 2\lg n\rceil} \lceil 2\lg n\rceil \cdot \Pr\{X = k\} + \sum_{k=\lceil 2\lg n\rceil+1}^{n} n \cdot \Pr\{X = k\} \\ &= \lceil 2\lg n\rceil \sum_{k=1}^{\lceil 2\lg n\rceil} \Pr\{X = k\} + n \sum_{k=\lceil 2\lg n\rceil+1}^{n} \Pr\{X = k\} \ .\end{aligned}$$

Since X takes on exactly one value, we have that $\sum_{k=1}^{\lceil 2\lg n\rceil} \Pr\{X = k\} = \Pr\{X \le \lceil 2\lg n\rceil\} \le 1$ and $\sum_{k=\lceil 2\lg n\rceil+1}^{n} \Pr\{X = k\} \le \Pr\{X > 2\lg n\} \le 1/n$, by part (c). Therefore,

$$\begin{aligned}\mathrm{E}[X] &\le \lceil 2\lg n\rceil \cdot 1 + n \cdot (1/n) \\ &= \lceil 2\lg n\rceil + 1 \\ &= O(\lg n) \ .\end{aligned}$$

Solution to Problem 11-2

a. A particular key is hashed to a particular slot with probability $1/n$. Suppose we select a specific set of k keys. The probability that these k keys are inserted into the slot in question and that all other keys are inserted elsewhere is

$$\left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \ .$$

Since there are $\binom{n}{k}$ ways to choose our k keys, we get

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} \ .$$

b. For $i = 1, 2, \ldots, n$, let X_i be a random variable denoting the number of keys that hash to slot i, and let A_i be the event that $X_i = k$, i.e., that exactly k keys hash to slot i. From part (a), we have $\Pr\{A\} = Q_k$. Then,

$$\begin{aligned}
P_k &= \Pr\{M = k\} \\
&= \Pr\left\{\left(\max_{1 \le i \le n} X_i\right) = k\right\} \\
&= \Pr\{\text{there exists } i \text{ such that } X_i = k \text{ and that } X_i \le k \text{ for } i = 1, 2, \ldots, n\} \\
&\le \Pr\{\text{there exists } i \text{ such that } X_i = k\} \\
&= \Pr\{A_1 \cup A_2 \cup \cdots \cup A_n\} \\
&\le \Pr\{A_1\} + \Pr\{A_2\} + \cdots + \Pr\{A_n\} \qquad \text{(by inequality (C.18))} \\
&= n Q_k \ .
\end{aligned}$$

c. We start by showing two facts. First, $1 - 1/n < 1$, which implies $(1 - 1/n)^{n-k} < 1$. Second, $n!/(n-k)! = n \cdot (n-1) \cdot (n-2) \cdots (n-k+1) < n^k$. Using these facts, along with the simplification $k! > (k/e)^k$ of equation (3.17), we have

$$\begin{aligned}
Q_k &= \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \frac{n!}{k!(n-k)!} \\
&< \frac{n!}{n^k k!(n-k)!} && ((1 - 1/n)^{n-k} < 1) \\
&< \frac{1}{k!} && (n!/(n-k)! < n^k) \\
&< \frac{e^k}{k^k} && (k! > (k/e)^k) \ .
\end{aligned}$$

d. Notice that when $n = 2$, $\lg \lg n = 0$, so to be precise, we need to assume that $n \ge 3$.

In part (c), we showed that $Q_k < e^k/k^k$ for any k; in particular, this inequality holds for k_0. Thus, it suffices to show that $e^{k_0}/{k_0}^{k_0} < 1/n^3$ or, equivalently, that $n^3 < {k_0}^{k_0}/e^{k_0}$.

Taking logarithms of both sides gives an equivalent condition:

$$\begin{aligned}
3 \lg n &< k_0(\lg k_0 - \lg e) \\
&= \frac{c \lg n}{\lg \lg n}(\lg c + \lg \lg n - \lg \lg \lg n - \lg e) \ .
\end{aligned}$$

Dividing both sides by $\lg n$ gives the condition

$$\begin{aligned}
3 &< \frac{c}{\lg \lg n}(\lg c + \lg \lg n - \lg \lg \lg n - \lg e) \\
&= c\left(1 + \frac{\lg c - \lg e}{\lg \lg n} - \frac{\lg \lg \lg n}{\lg \lg n}\right) \ .
\end{aligned}$$

Let x be the last expression in parentheses:

$$x = \left(1 + \frac{\lg c - \lg e}{\lg \lg n} - \frac{\lg \lg \lg n}{\lg \lg n}\right) \ .$$

We need to show that there exists a constant $c > 1$ such that $3 < cx$.

Noting that $\lim_{n \to \infty} x = 1$, we see that there exists n_0 such that $x \ge 1/2$ for all $n \ge n_0$. Thus, any constant $c > 6$ works for $n \ge n_0$.

We handle smaller values of n—in particular, $3 \le n < n_0$—as follows. Since n is constrained to be an integer, there are a finite number of n in the range $3 \le n < n_0$. We can evaluate the expression x for each such value of n and determine a value of c for which $3 < cx$ for all values of n. The final value of c that we use is the larger of

- 6, which works for all $n \ge n_0$, and
- $\max_{3 \le n < n_0} \{c : 3 < cx\}$, i.e., the largest value of c that we chose for the range $3 \le n < n_0$.

Thus, we have shown that $Q_{k_0} < 1/n^3$, as desired.

To see that $P_k < 1/n^2$ for $k \ge k_0$, we observe that by part (b), $P_k \le nQ_k$ for all k. Choosing $k = k_0$ gives $P_{k_0} \le nQ_{k_0} < n \cdot (1/n^3) = 1/n^2$. For $k > k_0$, we will show that we can pick the constant c such that $Q_k < 1/n^3$ for all $k \ge k_0$, and thus conclude that $P_k < 1/n^2$ for all $k \ge k_0$.

To pick c as required, we let c be large enough that $k_0 > 3 > e$. Then $e/k < 1$ for all $k \ge k_0$, and so e^k/k^k decreases as k increases. Thus,

$$\begin{aligned} Q_k &< e^k/k^k \\ &\le e^{k_0}/k^{k_0} \\ &< 1/n^3 \end{aligned}$$

for $k \ge k_0$.

e. The expectation of M is

$$\begin{aligned} \mathrm{E}[M] &= \sum_{k=0}^{n} k \cdot \Pr\{M = k\} \\ &= \sum_{k=0}^{k_0} k \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} k \cdot \Pr\{M = k\} \\ &\le \sum_{k=0}^{k_0} k_0 \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} n \cdot \Pr\{M = k\} \\ &\le k_0 \sum_{k=0}^{k_0} \Pr\{M = k\} + n \sum_{k=k_0+1}^{n} \Pr\{M = k\} \\ &= k_0 \cdot \Pr\{M \le k_0\} + n \cdot \Pr\{M > k_0\} \ , \end{aligned}$$

which is what we needed to show, since $k_0 = c \lg n / \lg \lg n$.

To show that $\mathrm{E}[M] = O(\lg n / \lg \lg n)$, note that $\Pr\{M \le k_0\} \le 1$ and

$$\begin{aligned} \Pr\{M > k_0\} &= \sum_{k=k_0+1}^{n} \Pr\{M = k\} \\ &= \sum_{k=k_0+1}^{n} P_k \\ &< \sum_{k=k_0+1}^{n} 1/n^2 \qquad \text{(by part (d))} \\ &< n \cdot (1/n^2) \\ &= 1/n \ . \end{aligned}$$

We conclude that

$$\begin{aligned} \mathrm{E}[M] &\leq k_0 \cdot 1 + n \cdot (1/n) \\ &= k_0 + 1 \\ &= O(\lg n / \lg\lg n) \ . \end{aligned}$$

Solution to Problem 11-3

a. From how the probe-sequence computation is specified, it is easy to see that the probe sequence is $\langle h(k), h(k)+1, h(k)+1+2, h(k)+1+2+3, \ldots, h(k)+1+2+3+\cdots+i, \ldots\rangle$, where all the arithmetic is modulo m. Starting the probe numbers from 0, the ith probe is offset (modulo m) from $h(k)$ by

$$\sum_{j=0}^{i} j = \frac{i(i+1)}{2} = \frac{1}{2}\,i^2 + \frac{1}{2}\,i \ .$$

Thus, we can write the probe sequence as

$$h'(k,i) = \left(h(k) + \frac{1}{2}\,i + \frac{1}{2}\,i^2\right) \bmod m \ ,$$

which demonstrates that this scheme is a special case of quadratic probing.

b. Let $h'(k,i)$ denote the ith probe of our scheme. We saw in part (a) that $h'(k,i) = (h(k) + i(i+1)/2) \bmod m$. To show that our algorithm examines every table position in the worst case, we show that for a given key, each of the first m probes hashes to a distinct value. That is, for any key k and for any probe numbers i and j such that $0 \leq i < j < m$, we have $h'(k,i) \neq h'(k,j)$. We do so by showing that $h'(k,i) = h'(k,j)$ yields a contradiction.

Let us assume that there exists a key k and probe numbers i and j satsifying $0 \leq i < j < m$ for which $h'(k,i) = h'(k,j)$. Then

$$h(k) + i(i+1)/2 \equiv h(k) + j(j+1)/2 \pmod{m} \ ,$$

which in turn implies that

$$i(i+1)/2 \equiv j(j+1)/2 \pmod{m} \ ,$$

or

$$j(j+1)/2 - i(i+1)/2 \equiv 0 \pmod{m} \ .$$

Since $j(j+1)/2 - i(i+1)/2 = (j-i)(j+i+1)/2$, we have

$$(j-i)(j+i+1)/2 \equiv 0 \pmod{m} \ .$$

The factors $j-i$ and $j+i+1$ must have different parities, i.e., $j-i$ is even if and only if $j+i+1$ is odd. (Work out the various cases in which i and j are even and odd.) Since $(j-i)(j+i+1)/2 \equiv 0 \pmod{m}$, we have $(j-i)(j+i+1)/2 = rm$ for some integer r or, equivalently, $(j-i)(j+i+1) = r \cdot 2m$. Using the assumption that m is a power of 2, let $m = 2^p$ for some nonnegative integer p, so that now we have $(j-i)(j+i+1) = r \cdot 2^{p+1}$. Because exactly one of

the factors $j - i$ and $j + i + 1$ is even, 2^{p+1} must divide one of the factors. It cannot be $j - i$, since $j - i < m < 2^{p+1}$. But it also cannot be $j + i + 1$, since $j + i + 1 \leq (m - 1) + (m - 2) + 1 = 2m - 2 < 2^{p+1}$. Thus we have derived the contradiction that 2^{p+1} divides neither of the factors $j - i$ and $j + i + 1$. We conclude that $h'(k, i) \neq h'(k, j)$.

Lecture Notes for Chapter 12: Binary Search Trees

Chapter 12 overview

Search trees

- Data structures that support many dynamic-set operations.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the height of the tree.
  - For complete binary tree with n nodes: worst case $\Theta(\lg n)$.
  - For linear chain of n nodes: worst case $\Theta(n)$.
- Different types of search trees include binary search trees, red-black trees (covered in Chapter 13), and B-trees (covered in Chapter 18).

We will cover binary search trees, tree walks, and operations on binary search trees.

Binary search trees

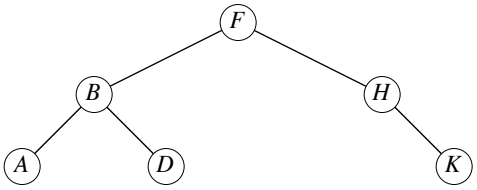
Binary search trees are an important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- As in Section 10.4, we represent a binary tree by a linked data structure in which each node is an object.
- $root[T]$ points to the root of tree T.
- Each node contains the fields
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - p: points to parent. $p[root[T]] = \text{NIL}$.
- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in left subtree of x, then $key[y] \leq key[x]$.

- If y is in right subtree of x, then $key[y] \geq key[x]$.

Draw sample tree.

[This is Figure 12.1(a) from the text, using A, B, D, F, H, K in place of 2, 3, 5, 5, 7, 8, with alphabetic comparisons. It's OK to have duplicate keys, though there are none in this example. Show that the binary-search-tree property holds.]



The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an ***inorder tree walk***. Elements are printed in monotonically increasing order.

How INORDER-TREE-WALK works:

- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(left[x])
       print key[x]
       INORDER-TREE-WALK(right[x])
```

Example: Do the inorder tree walk on the example above, getting the output $A\,B\,D\,F\,H\,K$.

Correctness: Follows by induction directly from the binary-search-tree property.

Time: Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once. *[Book has formal proof.]*

Querying a binary search tree

Searching

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
  then return x
if k < key[x]
  then return TREE-SEARCH(left[x], k)
  else  return TREE-SEARCH(right[x], k)
```

Initial call is TREE-SEARCH($root[T]$, k).

Example: Search for values D and C in the example tree from above.

Time: The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

[The text also gives an iterative version of TREE-SEARCH*, which is more efficient on most computers. The above recursive procedure is more straightforward, however.]*

Minimum and maximum

The binary-search-tree property guarantees that

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node.

Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
    do x ← left[x]
return x
```

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

Time: Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

Successor and predecessor

Assuming that all keys are distinct, the successor of a node x is the node y such that $key[y]$ is the smallest key $> key[x]$. (We can find x's successor based entirely on the tree structure. No key comparisons are necessary.) If x has the largest key in the binary search tree, then we say that x's successor is NIL.

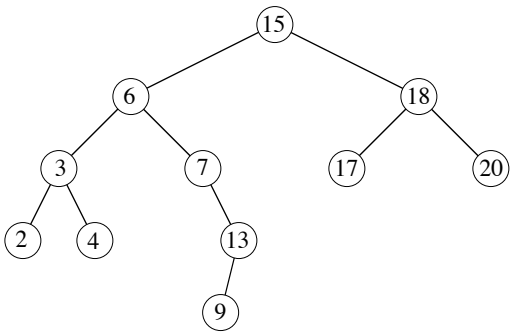
There are two cases:

1. If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.
2. If node x has an empty right subtree, notice that:
   - As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
   - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
  then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
    do x ← y
       y ← p[y]
return y
```

TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

Example:



- Find the successor of the node with key value 15. (Answer: Key value 17)
- Find the successor of the node with key value 6. (Answer: Key value 7)
- Find the successor of the node with key value 4. (Answer: Key value 6)
- Find the predecessor of the node with key value 6. (Answer: Key value 4)

Time: For both the TREE-SUCCESSOR and TREE-PREDECESSOR procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is $O(h)$, where h is the height of the tree.

Insertion and deletion

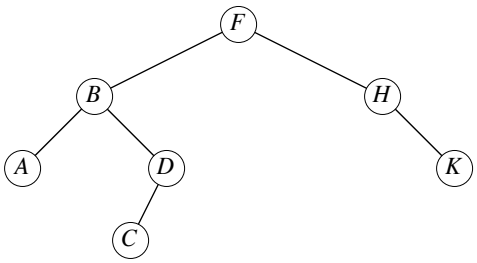
Insertion and deletion allows the dynamic set represented by a binary search tree to change. The binary-search-tree property must hold after the change. Insertion is more straightforward than deletion.

Insertion

TREE-INSERT(T, z)
$y \leftarrow$ NIL
$x \leftarrow root[T]$
while $x \neq$ NIL
  do $y \leftarrow x$
    if $key[z] < key[x]$
      then $x \leftarrow left[x]$
      else $x \leftarrow right[x]$
$p[z] \leftarrow y$
if $y =$ NIL
  then $root[T] \leftarrow z$ ▷ Tree T was empty
  else if $key[z] < key[y]$
      then $left[y] \leftarrow z$
      else $right[y] \leftarrow z$

- To insert value v into the binary search tree, the procedure is given node z, with $key[z] = v$, $left[z] =$ NIL, and $right[z] =$ NIL.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x: traces the downward path.
  - Pointer y: "trailing pointer" to keep track of parent of x.
- Traverse the tree downward by comparing the value of node at x with v, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare z's value with y's value, and insert z at either y's *left* or *right*, appropriately.

Example: Run TREE-INSERT(C) on the first sample binary search tree. Result:



Time: Same as TREE-SEARCH. On a tree of height h, procedure takes $O(h)$ time.

TREE-INSERT can be used with INORDER-TREE-WALK to sort a given set of numbers. (See Exercise 12.3-3.)

Deletion

TREE-DELETE is broken into three cases.

Case 1: z has no children.

- Delete z by making the parent of z point to NIL, instead of to z.

Case 2: z has one child.

- Delete z by making the parent of z point to z's child, instead of to z.

Case 3: z has two children.

- z's successor y has either no children or one child. (y is the minimum node—with no left child—in z's right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace z's key and satellite data with y's.

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out: either z or z's successor.
if left[z] = NIL or right[z] = NIL
  then y ← z
  else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children.
if left[y] ≠ NIL
  then x ← left[y]
  else x ← right[y]
▷ y is removed from the tree by manipulating pointers of p[y] and x.
if x ≠ NIL
  then p[x] ← p[y]
if p[y] = NIL
  then root[T] ← x
  else if y = left[p[y]]
         then left[p[y]] ← x
         else right[p[y]] ← x
▷ If it was z's successor that was spliced out, copy its data into z.
if y ≠ z
  then key[z] ← key[y]
       copy y's satellite data into z
return y
```

Example: We can demonstrate on the above sample tree.

- For Case 1, delete K.
- For Case 2, delete H.
- For Case 3, delete B, swapping it with C.

Time: $O(h)$, on a tree of height h.

Minimizing running time

We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- Problem: Worst case for binary search tree is $\Theta(n)$—no better than linked list.
- Solution: Guarantee small height (balanced tree)—$h = O(\lg n)$.

In later chapters, by varying the properties of binary search trees, we will be able to analyze running time in terms of n.

- Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).

Red-black trees are a special class of binary trees that avoids the worst-case behavior of $O(n)$ like "plain" binary search trees. Red-black trees are covered in detail in Chapter 13.

Expected height of a randomly built binary search tree

[These are notes on a starred section in the book. I covered this material in an optional lecture.]

Given a set of n distinct keys. Insert them in random order into an initially empty binary search tree.

- Each of the $n!$ permutations is equally likely.
- Different from assuming that every binary search tree on n keys is equally likely.

  Try it for $n = 3$. Will get 5 different binary search trees. When we look at the binary search trees resulting from each of the 3! input permutations, 4 trees will appear once and 1 tree will appear twice. *[This gives the idea for the solution to Exercise 12.4-3.]*
- Forget about deleting keys.

We will show that the expected height of a randomly built binary search tree is $O(\lg n)$.

Random variables

Define the following random variables:

- X_n = height of a randomly built binary search tree on n keys.
- $Y_n = 2^{X_n}$ = ***exponential height***.
- R_n = rank of the root within the set of n keys used to build the binary search tree.
  - Equally likely to be any element of $\{1, 2, \ldots, n\}$.
  - If $R_n = i$, then
    - Left subtree is a randomly-built binary search tree on $i - 1$ keys.
    - Right subtree is a randomly-built binary search tree on $n - i$ keys.

Foreshadowing

We will need to relate $\mathrm{E}[Y_n]$ to $\mathrm{E}[X_n]$.

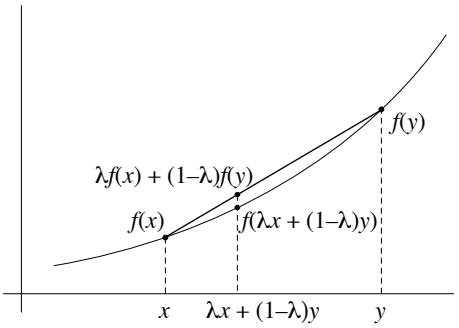
We'll use ***Jensen's inequality***:

$\mathrm{E}[f(X)] \geq f(\mathrm{E}[X])$, *[leave on board]*

provided

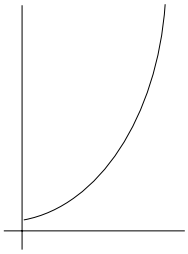
- the expectations exist and are finite, and
- $f(x)$ is ***convex***: for all x, y and all $0 \leq \lambda \leq 1$

  $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$.



Convex ≡ "curves upward"

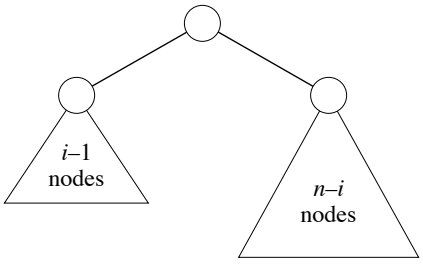
We'll use Jensen's inequality for $f(x) = 2^x$.



Since 2^x curves upward, it's convex.

Formula for Y_n

Think about Y_n, if we know that $R_n = i$:



Height of root is 1 more than the maximum height of its children:

$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i})$.

Base cases:

- $Y_1 = 1$ (expected height of a 1-node tree is $2^0 = 1$).
- Define $Y_0 = 0$.

Define indicator random variables $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$:

$Z_{n,i} = \mathrm{I}\{R_n = i\}$.

R_n is equally likely to be any element of $\{1, 2, \ldots, n\}$
$\Rightarrow \Pr\{R_n = i\} = 1/n$
$\Rightarrow \mathrm{E}[Z_{n,i}] = 1/n$ *[leave on board]*
(since $\mathrm{E}[\mathrm{I}\{A\}] = \Pr\{A\}$)

Consider a given n-node binary search tree (which could be a subtree). Exactly one $Z_{n,i}$ is 1, and all others are 0. Hence,

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \cdot (2 \cdot \max(Y_{i-1}, Y_{n-i})) \; . \qquad \textit{[leave on board]}$$

[Recall: $Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i})$ was assuming that $R_n = i$.]

Bounding $\mathrm{E}[Y_n]$

We will show that $\mathrm{E}[Y_n]$ is polynomial in n, which will imply that $\mathrm{E}[X_n] = O(\lg n)$.

Claim
$Z_{n,i}$ is independent of Y_{i-1} and Y_{n-i}.

Justification If we choose the root such that $R_n = i$, the left subtree contains $i - 1$ nodes, and it's like any other randomly built binary search tree with $i - 1$ nodes. Other than the number of nodes, the left subtree's structure has nothing to do with it being the left subtree of the root. Hence, Y_{i-1} and $Z_{n,i}$ are independent.

Similarly, Y_{n-i} and $Z_{n,i}$ are independent. ■ (claim)

Fact
If X and Y are nonnegative random variables, then $\mathrm{E}[\max(X, Y)] \le \mathrm{E}[X] + \mathrm{E}[Y]$.
[Leave on board. This is Exercise C.3-4 from the text.]

Thus,

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i} \, (2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i} \cdot (2 \cdot \max(Y_{i-1}, Y_{n-i}))] \qquad \text{(linearity of expectation)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}] \cdot \mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad \text{(independence)}
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[2 \cdot \max(Y_{i-1}, Y_{n-i})\right] && (\mathrm{E}\left[Z_{n,i}\right] = 1/n) \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}\left[\max(Y_{i-1}, Y_{n-i})\right] && (\mathrm{E}\left[aX\right] = a\,\mathrm{E}\left[X\right]) \\
&\le \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}\left[Y_{i-1}\right] + \mathrm{E}\left[Y_{n-i}\right]) && \text{(earlier fact)} \ .
\end{aligned}
$$

Observe that the last summation is

$$
\begin{aligned}
&(\mathrm{E}\left[Y_0\right] + \mathrm{E}\left[Y_{n-1}\right]) + (\mathrm{E}\left[Y_1\right] + \mathrm{E}\left[Y_{n-2}\right]) + (\mathrm{E}\left[Y_2\right] + \mathrm{E}\left[Y_{n-3}\right]) \\
&\qquad + \cdots + (\mathrm{E}\left[Y_{n-1}\right] + \mathrm{E}\left[Y_0\right]) = 2 \sum_{i=0}^{n-1} \mathrm{E}\left[Y_i\right] \ ,
\end{aligned}
$$

and so we get the recurrence

$$
\mathrm{E}\left[Y_n\right] \le \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}\left[Y_i\right] \ . \qquad \textit{[leave on board]}
$$

Solving the recurrence

We will show that for all integers $n > 0$, this recurrence has the solution

$$
\mathrm{E}\left[Y_n\right] \le \frac{1}{4}\binom{n+3}{3} \ .
$$

Lemma

$$
\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4} \ .
$$

[This lemma solves Exercise 12.4-1.]

Proof Use Pascal's identity (Exercise C.1-7): $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

Also using the simple identity $\binom{4}{4} = 1 = \binom{3}{3}$, we have

$$
\begin{aligned}
\binom{n+3}{4} &= \binom{n+2}{3} + \binom{n+2}{4} \\
&= \binom{n+2}{3} + \binom{n+1}{3} + \binom{n+1}{4} \\
&= \binom{n+2}{3} + \binom{n+1}{3} + \binom{n}{3} + \binom{n}{4} \\
&\ \ \vdots \\
&= \binom{n+2}{3} + \binom{n+1}{3} + \binom{n}{3} + \cdots + \binom{4}{3} + \binom{4}{4} \\
&= \binom{n+2}{3} + \binom{n+1}{3} + \binom{n}{3} + \cdots + \binom{4}{3} + \binom{3}{3} \\
&= \sum_{i=0}^{n-1} \binom{i+3}{3} \ .
\end{aligned}
$$

■ (lemma)

We solve the recurrence by induction on n.

Basis: $n = 1$.

$$1 = Y_1 = \mathrm{E}[Y_1] \le \frac{1}{4}\binom{1+3}{3} = \frac{1}{4} \cdot 4 = 1 \ .$$

Inductive step: Assume that $\mathrm{E}[Y_i] \le \frac{1}{4}\binom{i+3}{3}$ for all $i < n$. Then

$$\begin{aligned}
\mathrm{E}[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] && \text{(from before)} \\
&\le \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(inductive hypothesis)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(lemma)} \\
&= \frac{1}{n} \cdot \frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4} \cdot \frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3} \ .
\end{aligned}$$

Thus, we've proven that $\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3}$.

Bounding $\mathrm{E}[X_n]$

With our bound on $\mathrm{E}[Y_n]$, we use Jensen's inequality to bound $\mathrm{E}[X_n]$:

$$2^{\mathrm{E}[X_n]} \le \mathrm{E}[2^{X_n}] = \mathrm{E}[Y_n] \ .$$

Thus,

$$\begin{aligned}
2^{\mathrm{E}[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\
&= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\
&= O(n^3) \ .
\end{aligned}$$

Taking logs of both sides gives $\mathrm{E}[X_n] = O(\lg n)$.

Done!

Solutions for Chapter 12: Binary Search Trees

Solution to Exercise 12.1-2

In a heap, a node's key is $\geq$ both of its children's keys. In a binary search tree, a node's key is $\geq$ its left child's key, but $\leq$ its right child's key.

The heap property, unlike the binary-searth-tree property, doesn't help print the nodes in sorted order because it doesn't tell which subtree of a node contains the element to print before that node. In a heap, the largest element smaller than the node could be in either subtree.

Note that if the heap property could be used to print the keys in sorted order in $O(n)$ time, we would have an $O(n)$-time algorithm for sorting, because building the heap takes only $O(n)$ time. But we know (Chapter 8) that a comparison sort must take $\Omega(n \lg n)$ time.

Solution to Exercise 12.2-5

Let x be a node with two children. In an inorder tree walk, the nodes in x's left subtree immediately precede x and the nodes in x's right subtree immediately follow x. Thus, x's predecessor is in its left subtree, and its successor is in its right subtree.

Let s be x's successor. Then s cannot have a left child, for a left child of s would come between x and s in the inorder walk. (It's after x because it's in x's right subtree, and it's before s because it's in s's left subtree.) If any node were to come between x and s in an inorder walk, then s would not be x's successor, as we had supposed.

Symmetrically, x's predecessor has no right child.

Solution to Exercise 12.2-7

Note that a call to TREE-MINIMUM followed by $n-1$ calls to TREE-SUCCESSOR performs exactly the same inorder walk of the tree as does the procedure INORDER-TREE-WALK. INORDER-TREE-WALK prints the TREE-MINIMUM first, and by

definition, the TREE-SUCCESSOR of a node is the next node in the sorted order determined by an inorder tree walk.

This algorithm runs in $\Theta(n)$ time because:

- It requires $\Omega(n)$ time to do the n procedure calls.
- It traverses each of the $n - 1$ tree edges at most twice, which takes $O(n)$ time.

To see that each edge is traversed at most twice (once going down the tree and once going up), consider the edge between any node u and either of its children, node v. By starting at the root, we must traverse (u, v) downward from u to v, before traversing it upward from v to u. The only time the tree is traversed downward is in code of TREE-MINIMUM, and the only time the tree is traversed upward is in code of TREE-SUCCESSOR when we look for the successor of a node that has no right subtree.

Suppose that v is u's left child.

- Before printing u, we must print all the nodes in its left subtree, which is rooted at v, guaranteeing the downward traversal of edge (u, v).
- After all nodes in u's left subtree are printed, u must be printed next. Procedure TREE-SUCCESSOR traverses an upward path to u from the maximum element (which has no right subtree) in the subtree rooted at v. This path clearly includes edge (u, v), and since all nodes in u's left subtree are printed, edge (u, v) is never traversed again.

Now suppose that v is u's right child.

- After u is printed, TREE-SUCCESSOR(u) is called. To get to the minimum element in u's right subtree (whose root is v), the edge (u, v) must be traversed downward.
- After all values in u's right subtree are printed, TREE-SUCCESSOR is called on the maximum element (again, which has no right subtree) in the subtree rooted at v. TREE-SUCCESSOR traverses a path up the tree to an element after u, since u was already printed. Edge (u, v) must be traversed upward on this path, and since all nodes in u's right subtree have been printed, edge (u, v) is never traversed again.

Hence, no edge is traversed twice in the same direction.

Therefore, this algorithm runs in $\Theta(n)$ time.

Solution to Exercise 12.3-3

Here's the algorithm:

TREE-SORT(A)
let T be an empty binary search tree
for $i \leftarrow 1$ **to** n
    do TREE-INSERT$(T, A[i])$
INORDER-TREE-WALK$(root[T])$

Worst case: $\Theta(n^2)$—occurs when a linear chain of nodes results from the repeated TREE-INSERT operations.

Best case: $\Theta(n \lg n)$—occurs when a binary tree of height $\Theta(\lg n)$ results from the repeated TREE-INSERT operations.

Solution to Exercise 12.4-1

We will answer the second part first. We shall show that if the average depth of a node is $\Theta(\lg n)$, then the height of the tree is $O(\sqrt{n \lg n})$. Then we will answer the first part by exhibiting that this bound is tight: there is a binary search tree with average node depth $\Theta(\lg n)$ and height $\Theta(\sqrt{n \lg n}) = \omega(\lg n)$.

Lemma
If the average depth of a node in an n-node binary search tree is $\Theta(\lg n)$, then the height of the tree is $O(\sqrt{n \lg n})$.

Proof Suppose that an n-node binary search tree has average depth $\Theta(\lg n)$ and height h. Then there exists a path from the root to a node at depth h, and the depths of the nodes on this path are $0, 1, \ldots, h$. Let P be the set of nodes on this path and Q be all other nodes. Then the average depth of a node is

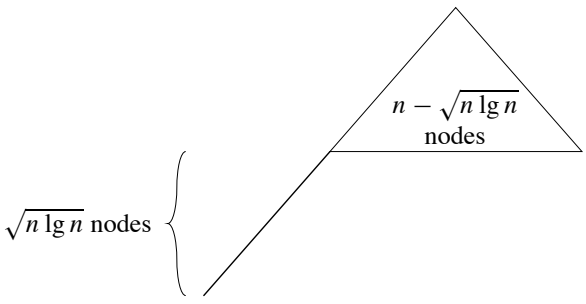
$$\begin{aligned}
\frac{1}{n}\left(\sum_{x \in P} \text{depth}(x) + \sum_{y \in Q} \text{depth}(y)\right) &\geq \frac{1}{n}\sum_{x \in P} \text{depth}(x) \\
&= \frac{1}{n}\sum_{d=0}^{h} d \\
&= \frac{1}{n} \cdot \Theta(h^2) \; .
\end{aligned}$$

For the purpose of contradiction, suppose that h is not $O(\sqrt{n \lg n})$, so that $h = \omega(\sqrt{n \lg n})$. Then we have

$$\begin{aligned}
\frac{1}{n} \cdot \Theta(h^2) &= \frac{1}{n} \cdot \omega(n \lg n) \\
&= \omega(\lg n) \; ,
\end{aligned}$$

which contradicts the assumption that the average depth is $\Theta(\lg n)$. Thus, the height is $O(\sqrt{n \lg n})$. ■

Here is an example of an n-node binary search tree with average node depth $\Theta(\lg n)$ but height $\omega(\lg n)$:



In this tree, $n - \sqrt{n \lg n}$ nodes are a complete binary tree, and the other $\sqrt{n \lg n}$ nodes protrude from below as a single chain. This tree has height

$$\begin{aligned}\Theta(\lg(n - \sqrt{n \lg n})) + \sqrt{n \lg n} &= \Theta(\sqrt{n \lg n}) \\ &= \omega(\lg n) .\end{aligned}$$

To compute an upper bound on the average depth of a node, we use $O(\lg n)$ as an upper bound on the depth of each of the $n - \sqrt{n \lg n}$ nodes in the complete binary tree part and $O(\lg n + \sqrt{n \lg n})$ as an upper bound on the depth of each of the $\sqrt{n \lg n}$ nodes in the protruding chain. Thus, the average depth of a node is bounded from above by

$$\begin{aligned}\frac{1}{n} \cdot O(\sqrt{n \lg n}\ (\lg n + \sqrt{n \lg n}) + (n - \sqrt{n \lg n}) \lg n) &= \frac{1}{n} \cdot O(n \lg n) \\ &= O(\lg n) .\end{aligned}$$

To bound the average depth of a node from below, observe that the bottommost level of the complete binary tree part has $\Theta(n - \sqrt{n \lg n})$ nodes, and each of these nodes has depth $\Theta(\lg n)$. Thus, the average node depth is at least

$$\begin{aligned}\frac{1}{n} \cdot \Theta((n - \sqrt{n \lg n}) \lg n) &= \frac{1}{n} \cdot \Omega(n \lg n) \\ &= \Omega(\lg n) .\end{aligned}$$

Because the average node depth is both $O(\lg n)$ and $\Omega(\lg n)$, it is $\Theta(\lg n)$.

Solution to Exercise 12.4-4

We'll go one better than showing that the function 2^x is convex. Instead, we'll show that the function c^x is convex, for any positive constant c. According to the definition of convexity on page 1109 of the text, a function $f(x)$ is convex if for all x and y and for all $0 \le \lambda \le 1$, we have $f(\lambda x + (1 - \lambda)y) \le \lambda f(x) + (1 - \lambda) f(y)$. Thus, we need to show that for all $0 \le \lambda \le 1$, we have $c^{\lambda x + (1-\lambda)y} \le \lambda c^x + (1-\lambda)c^y$.

We start by proving the following lemma.

Lemma

For any real numbers a and b and any positive real number c,

$$c^a \ge c^b + (a - b)c^b \ln c .$$

Proof We first show that for all real r, we have $c^r \ge 1 + r \ln c$. By equation (3.11) from the text, we have $e^x \ge 1 + x$ for all real x. Let $x = r \ln c$, so that $e^x = e^{r \ln c} = (e^{\ln c})^r = c^r$. Then we have $c^r = e^{r \ln c} \ge 1 + r \ln c$.

Substituting $a - b$ for r in the above inequality, we have $c^{a-b} \ge 1 + (a - b) \ln c$. Multiplying both sides by c^b gives $c^a \ge c^b + (a - b)c^b \ln c$. ■ (lemma)

Now we can show that $c^{\lambda x + (1-\lambda)y} \le \lambda c^x + (1 - \lambda)c^y$ for all $0 \le \lambda \le 1$. For convenience, let $z = \lambda x + (1 - \lambda)y$.

In the inequality given by the lemma, substitute x for a and z for b, giving

$$c^x \ge c^z + (x - z)c^z \ln c .$$

Also substitute y for a and z for b, giving

$c^y \geq c^z + (y - z)c^z \ln c$.

If we multiply the first inequality by λ and the second by $1 - \lambda$ and then add the resulting inequalities, we get

$$\begin{aligned}
&\lambda c^x + (1 - \lambda)c^y \\
&\quad \geq \lambda(c^z + (x - z)c^z \ln c) + (1 - \lambda)(c^z + (y - z)c^z \ln c) \\
&\quad = \lambda c^z + \lambda x c^z \ln c - \lambda z c^z \ln c + (1 - \lambda)c^z + (1 - \lambda)y c^z \ln c - (1 - \lambda)z c^z \ln c \\
&\quad = (\lambda + (1 - \lambda))c^z + (\lambda x + (1 - \lambda)y)c^z \ln c - (\lambda + (1 - \lambda))z c^z \ln c \\
&\quad = c^z + z c^z \ln c - z c^z \ln c \\
&\quad = c^z \\
&\quad = c^{\lambda x + (1-\lambda)y} ,
\end{aligned}$$

as we wished to show.

Solution to Problem 12-2

To sort the strings of S, we first insert them into a radix tree, and then use a preorder tree walk to extract them in lexicographically sorted order. The tree walk outputs strings only for nodes that indicate the existence of a string (i.e., those that are lightly shaded in Figure 12.5 of the text).

Correctness: The preorder ordering is the correct order because:

- Any node's string is a prefix of all its descendants' strings and hence belongs before them in the sorted order (rule 2).
- A node's left descendants belong before its right descendants because the corresponding strings are identical up to that parent node, and in the next position the left subtree's strings have 0 whereas the right subtree's strings have 1 (rule 1).

Time: $\Theta(n)$.

- Insertion takes $\Theta(n)$ time, since the insertion of each string takes time proportional to its length (traversing a path through the tree whose length is the length of the string), and the sum of all the string lengths is n.
- The preorder tree walk takes $O(n)$ time. It is just like INORDER-TREE-WALK (it prints the current node and calls itself recursively on the left and right subtrees), so it takes time proportional to the number of nodes in the tree. The number of nodes is at most 1 plus the sum (n) of the lengths of the binary strings in the tree, because a length-i string corresponds to a path through the root and i other nodes, but a single node may be shared among many string paths.

Solution to Problem 12-3

a. The total path length $P(T)$ is defined as $\sum_{x \in T} d(x, T)$. Dividing both quantities by n gives the desired equation.

b. For any node x in T_L, we have $d(x, T_L) = d(x, T) - 1$, since the distance to the root of T_L is one less than the distance to the root of T. Similarly, for any node x in T_R, we have $d(x, T_R) = d(x, T) - 1$. Thus, if T has n nodes, we have

$$P(T) = P(T_L) + P(T_R) + n - 1 \ ,$$

since each of the n nodes of T (except the root) is in either T_L or T_R.

c. If T is a randomly built binary search tree, then the root is equally likely to be any of the n elements in the tree, since the root is the first element inserted. It follows that the number of nodes in subtree T_L is equally likely to be any integer in the set $\{0, 1, \ldots, n-1\}$. The definition of $P(n)$ as the average total path length of a randomly built binary search tree, along with part (b), gives us the recurrence

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n - 1) \ .$$

d. Since $P(0) = 0$, and since for $k = 1, 2, \ldots, n-1$, each term $P(k)$ in the summation appears once as $P(i)$ and once as $P(n-i-1)$, we can rewrite the equation from part (c) as

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n) \ .$$

e. Observe that if, in the recurrence (7.6) in part (c) of Problem 7-2, we replace $\mathrm{E}[T(\cdot)]$ by $P(\cdot)$ and we replace q by k, we get almost the same recurrence as in part (d) of Problem 12-3. The remaining difference is that in Problem 12-3(d), the summation starts at 1 rather than 2. Observe, however, that a binary tree with just one node has a total path length of 0, so that $P(1) = 0$. Thus, we can rewrite the recurrence in Problem 12-3(d) as

$$P(n) = \frac{2}{n} \sum_{k=2}^{n-1} P(k) + \Theta(n)$$

and use the same technique as was used in Problem 7-2 to solve it.

We start by solving part (d) of Problem 7-2: showing that

$$\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \ .$$

Following the hint in Problem 7-2(d), we split the summation into two parts:

$$\sum_{k=2}^{n-1} k \lg k = \sum_{k=2}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \ .$$

The $\lg k$ in the first summation on the right is less than $\lg(n/2) = \lg n - 1$, and the $\lg k$ in the second summation is less than $\lg n$. Thus,

$$\begin{aligned}
\sum_{k=2}^{n-1} k \lg k &< (\lg n - 1) \sum_{k=2}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=2}^{n-1} k - \sum_{k=2}^{\lceil n/2 \rceil - 1} k \\
&\le \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \\
&\le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}$$

if $n \ge 2$.

Now we show that the recurrence

$$P(n) = \frac{2}{n} \sum_{k=2}^{n-1} P(k) + \Theta(n)$$

has the solution $P(n) = O(n \lg n)$. We use the substitution method. Assume inductively that $P(n) \le an \lg n + b$ for some positive constants a and b to be determined. We can pick a and b sufficiently large so that $an \lg n + b \ge P(1)$. Then, for $n > 1$, we have by substitution

$$\begin{aligned}
P(n) &= \frac{2}{n} \sum_{k=2}^{n-1} P(k) + \Theta(n) \\
&\le \frac{2}{n} \sum_{k=2}^{n-1} (ak \lg k + b) + \Theta(n) \\
&= \frac{2a}{n} \sum_{k=2}^{n-1} k \lg k + \frac{2b}{n}(n-2) + \Theta(n) \\
&\le \frac{2a}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \frac{2b}{n}(n-2) + \Theta(n) \\
&\le an \lg n - \frac{a}{4} n + 2b + \Theta(n) \\
&= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4} n \right) \\
&\le an \lg n + b \,,
\end{aligned}$$

since we can choose a large enough so that $\frac{a}{4} n$ dominates $\Theta(n) + b$. Thus, $P(n) = O(n \lg n)$.

f. We draw an analogy between inserting an element into a subtree of a binary search tree and sorting a subarray in quicksort. Observe that once an element x is chosen as the root of a subtree T, all elements that will be inserted after x into T will be compared to x. Similarly, observe that once an element y is chosen as the pivot in a subarray S, all other elements in S will be compared to y. Therefore, the quicksort implementation in which the comparisons are the same as those made when inserting into a binary search tree is simply to consider the pivots in the same order as the order in which the elements are inserted into the tree.

Lecture Notes for Chapter 13: Red-Black Trees

Chapter 13 overview

Red-black trees

- A variation of binary search trees.
- ***Balanced***: height is $O(\lg n)$, where n is the number of nodes.
- Operations will take $O(\lg n)$ time in the worst case.

[These notes are a bit simpler than the treatment in the book, to make them more amenable to a lecture situation. Our students first see red-black trees in a course that precedes our algorithms course. This set of lecture notes is intended as a refresher for the students, bearing in mind that some time may have passed since they last saw red-black trees.

The procedures in this chapter are rather long sequences of pseudocode. You might want to make arrangements to project them rather than spending time writing them on a board.]

Red-black trees

A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute *color*, which is either red or black.

All leaves are empty (nil) and colored black.

- We use a single sentinel, $nil[T]$, for all the leaves of red-black tree T.
- $color[nil[T]]$ is black.
- The root's parent is also $nil[T]$.

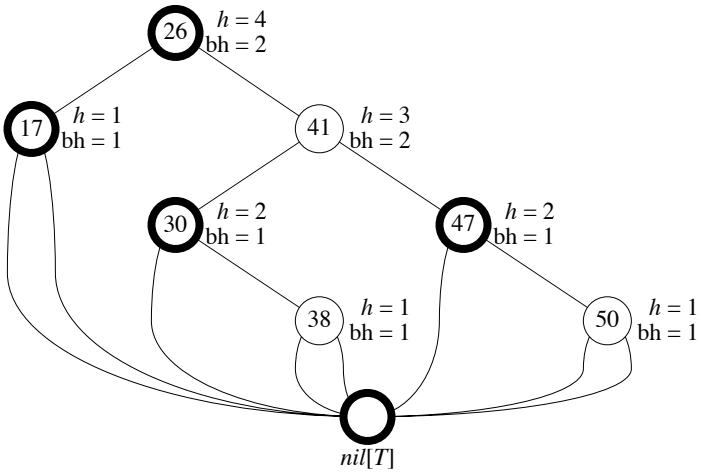
All other attributes of binary search trees are inherited by red-black trees (*key*, *left*, *right*, and p). We don't care about the key in $nil[T]$.

Red-black properties

[Leave these up on the board.]

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($nil[T]$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Example:



[Nodes with bold outline indicate black nodes. Don't add heights and black-heights yet. We won't bother with drawing $nil[T]$ any more.]

Height of a red-black tree

- ***Height of a node*** is the number of edges in a longest path to a leaf.
- ***Black-height*** of a node x: $\mathrm{bh}(x)$ is the number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x. By property 5, black-height is well defined.

[Now label the example tree with height h and bh *values.]*

Claim

Any node with height h has black-height $\geq h/2$.

Proof By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. ■ (claim)

Claim

The subtree rooted at any node x contains $\geq 2^{\mathrm{bh}(x)} - 1$ internal nodes.

Proof By induction on height of x.

Basis: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow \text{bh}(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.

Inductive step: Let the height of x be h and $\text{bh}(x) = b$. Any child of x has height $h - 1$ and black-height either b (if the child is red) or $b - 1$ (if the child is black). By the inductive hypothesis, each child has $\geq 2^{\text{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ internal nodes. (The $+1$ is for x itself.) ■ (claim)

Lemma
A red-black tree with n internal nodes has height $\leq 2\lg(n + 1)$.

Proof Let h and b be the height and black-height of the root, respectively. By the above two claims,

$$n \geq 2^b - 1 \geq 2^{h/2} - 1 .$$

Adding 1 to both sides and then taking logs gives $\lg(n + 1) \geq h/2$, which implies that $h \leq 2\lg(n + 1)$. ■ (theorem)

Operations on red-black trees

The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(\text{height})$ time. Thus, they take $O(\lg n)$ time on red-black trees.

Insertion and deletion are not so easy.

If we insert, what color to make the new node?

- Red? Might violate property 4.
- Black? Might violate property 5.

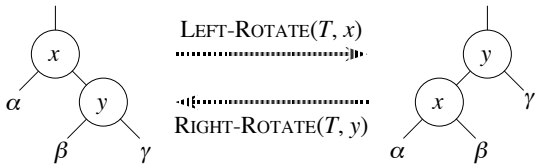
If we delete, thus removing a node, what color was the node that was removed?

- Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
- Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

Rotations

- The basic tree-restructuring operation.
- Needed to maintain red-black trees as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)

- Won't upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.



LEFT-ROTATE(T, x)

```
y ← right[x]            ▷ Set y.
right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree.
if left[y] ≠ nil[T]
  then p[left[y]] ← x
p[y] ← p[x]             ▷ Link x's parent to y.
if p[x] = nil[T]
  then root[T] ← y
  else if x = left[p[x]]
         then left[p[x]] ← y
         else right[p[x]] ← y
left[y] ← x             ▷ Put x on y's left.
p[x] ← y
```

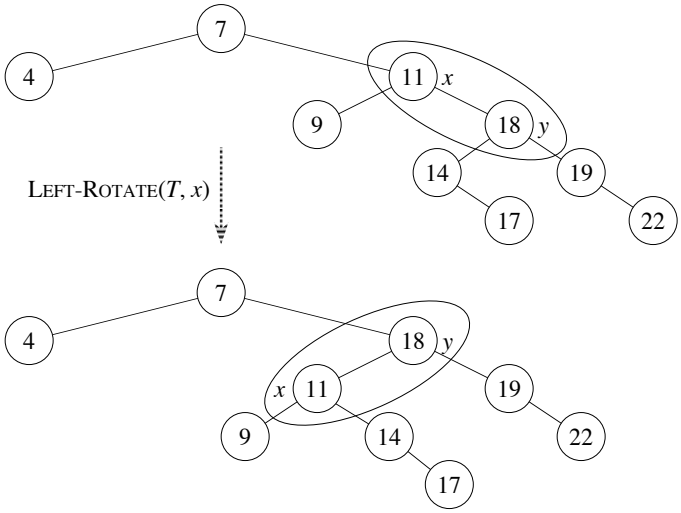
[In the first two printings of the second edition, this procedure contains a bug that is corrected above (and in the third and subsequent printings). The bug is that the assignment in line 4 ($p[left[y]] \leftarrow x$) should be performed only when y's left child is not the sentinel (which is tested in line 3). The first two printings omitted this test.]

The pseudocode for LEFT-ROTATE assumes that

- $right[x] \neq nil[T]$, and
- root's parent is $nil[T]$.

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

Example: *[Use to demonstrate that rotation maintains inorder ordering of keys. Node colors omitted.]*



- Before rotation: keys of x's left subtree $\leq 11 \leq$ keys of y's left subtree $\leq 18 \leq$ keys of y's right subtree.
- Rotation makes y's left subtree into x's right subtree.
- After rotation: keys of x's left subtree $\leq 11 \leq$ keys of x's right subtree $\leq 18 \leq$ keys of y's right subtree.

Time: $O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.

Notes:

- Rotation is a very basic operation, also used in AVL trees and splay trees.
- Some books talk of rotating on an edge rather than on a node.

Insertion

Start by doing regular binary-search-tree insertion:

```
RB-INSERT(T, z)
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
       if key[z] < key[x]
         then x ← left[x]
         else x ← right[x]
p[z] ← y
if y = nil[T]
  then root[T] ← z
  else if key[z] < key[y]
         then left[y] ← z
         else right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

Which property might be violated?

1. OK.
2. If z is the root, then there's a violation. Otherwise, OK.
3. OK.
4. If $p[z]$ is red, there's a violation: both z and $p[z]$ are red.
5. OK.

Remove the violation by calling RB-INSERT-FIXUP:

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
          then y ← right[p[p[z]]]
               if color[y] = RED
                 then color[p[z]] ← BLACK                          ▷ Case 1
                      color[y] ← BLACK                             ▷ Case 1
                      color[p[p[z]]] ← RED                         ▷ Case 1
                      z ← p[p[z]]                                  ▷ Case 1
                 else if z = right[p[z]]
                        then z ← p[z]                              ▷ Case 2
                             LEFT-ROTATE(T, z)                     ▷ Case 2
                      color[p[z]] ← BLACK                          ▷ Case 3
                      color[p[p[z]]] ← RED                         ▷ Case 3
                      RIGHT-ROTATE(T, p[p[z]])                     ▷ Case 3
          else (same as then clause
                     with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

Loop invariant:

At the start of each iteration of the **while** loop,

a. z is red.

b. There is at most one red-black violation:

- Property 2: z is a red root, or
- Property 4: z and $p[z]$ are both red.

[The book has a third part of the loop invariant, but we omit it for lecture.]

Initialization: We've already seen why the loop invariant holds initially.

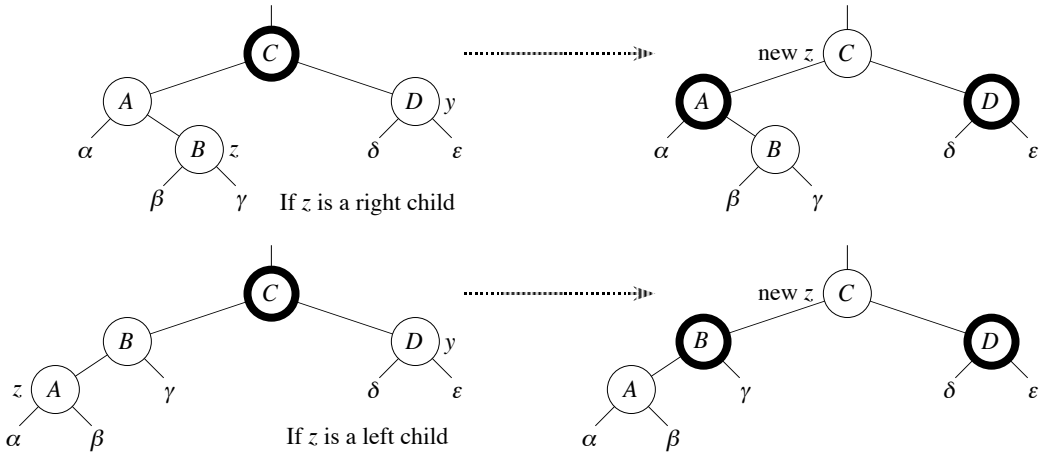
Termination: The loop terminates because $p[z]$ is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.

Maintenance: We drop out when z is the root (since then $p[z]$ is the sentinel $nil[T]$, which is black). When we start the loop body, the only violation is of property 4.

There are 6 cases, 3 of which are symmetric to the other 3. The cases are not mutually exclusive. We'll consider cases in which $p[z]$ is a left child.

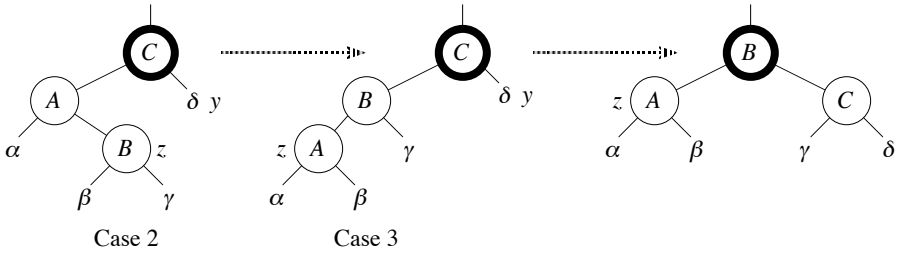
Let y be z's uncle ($p[z]$'s sibling).

Case 1: y is red



- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black $\Rightarrow$ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Rightarrow$ restores property 5.
- The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).

Case 2: y is black, z is a right child



Case 2 Case 3

- Left rotate around $p[z]$ $\Rightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Case 3: y is black, z is a left child

- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

Analysis

$O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

Within RB-INSERT-FIXUP:

- Each iteration takes $O(1)$ time.
- Each iteration is either the last one or it moves z up 2 levels.
- $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

Deletion

Start by doing regular binary-search-tree deletion:

```
RB-DELETE(T, z)
if left[z] = nil[T] or right[z] = nil[T]
  then y ← z
  else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
  then x ← left[y]
  else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
  then root[T] ← x
  else if y = left[p[y]]
         then left[p[y]] ← x
         else right[p[y]] ← x
if y ≠ z
  then key[z] ← key[y]
       copy y's satellite data into z
if color[y] = BLACK
  then RB-DELETE-FIXUP(T, x)
return y
```

- y is the node that was actually spliced out.
- x is either
  - y's sole non-sentinel child before y was spliced out, or
  - the sentinel, if y had no children.

  In both cases, $p[x]$ is now the node that was previously y's parent.

If y is black, we could have violations of red-black properties:

1. OK.
2. If y is the root and x is red, then the root has become red.
3. OK.
4. Violation if $p[y]$ and x are both red.
5. Any path containing y now has 1 fewer black node.
   - Correct by giving x an "extra black."

- Add 1 to count of black nodes on paths containing x.
- Now property 5 is OK, but property 1 is not.
- x is either ***doubly black*** (if $color[x] = \text{BLACK}$) or ***red & black*** (if $color[x] = \text{RED}$).
- The attribute $color[x]$ is still either RED or BLACK. No new values for *color* attribute.
- In other words, the extra blackness on a node is by virtue of x pointing to the node.

Remove the violations by calling RB-DELETE-FIXUP:

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w ← right[p[x]]
               if color[w] = RED
                  then color[w] ← BLACK                          ▷ Case 1
                       color[p[x]] ← RED                         ▷ Case 1
                       LEFT-ROTATE(T, p[x])                      ▷ Case 1
                       w ← right[p[x]]                           ▷ Case 1
               if color[left[w]] = BLACK and color[right[w]] = BLACK
                  then color[w] ← RED                            ▷ Case 2
                       x ← p[x]                                  ▷ Case 2
                  else if color[right[w]] = BLACK
                          then color[left[w]] ← BLACK            ▷ Case 3
                               color[w] ← RED                    ▷ Case 3
                               RIGHT-ROTATE(T, w)                ▷ Case 3
                               w ← right[p[x]]                   ▷ Case 3
                       color[w] ← color[p[x]]                    ▷ Case 4
                       color[p[x]] ← BLACK                       ▷ Case 4
                       color[right[w]] ← BLACK                   ▷ Case 4
                       LEFT-ROTATE(T, p[x])                      ▷ Case 4
                       x ← root[T]                               ▷ Case 4
          else (same as then clause with "right" and "left" exchanged)
color[x] ← BLACK
```

Idea: Move the extra black up the tree until

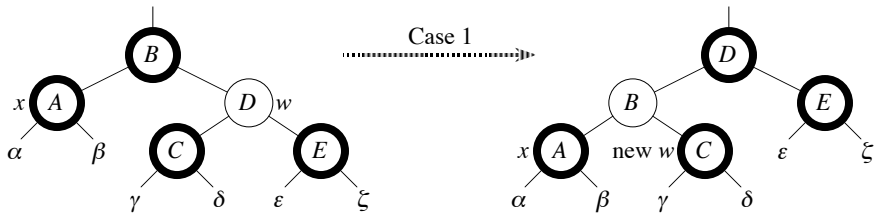
- x points to a red & black node $\Rightarrow$ turn it into a black node,
- x points to the root $\Rightarrow$ just remove the extra black, or
- we can do certain rotations and recolorings and finish.

Within the **while** loop:

- x always points to a nonroot doubly black node.
- w is x's sibling.
- w cannot be $nil[T]$, since that would violate property 5 at $p[x]$.

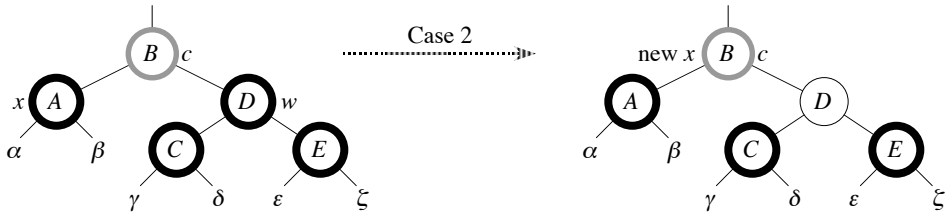
There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which x is a left child.

Case 1: w is red



- w must have black children.
- Make w black and $p[x]$ red.
- Then left rotate on $p[x]$.
- New sibling of x was a child of w before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

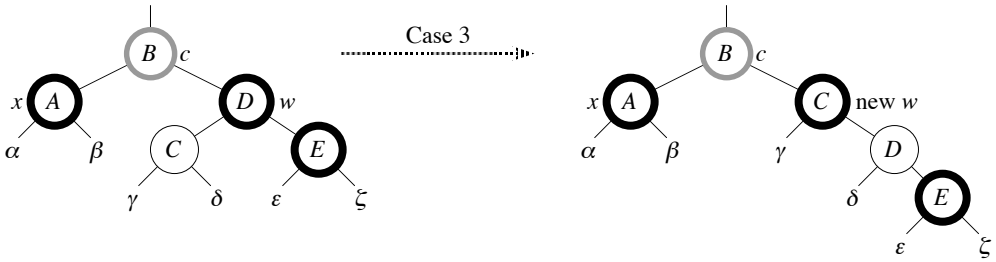
Case 2: w is black and both of w's children are black



[Node with gray outline is of unknown color, denoted by c.]

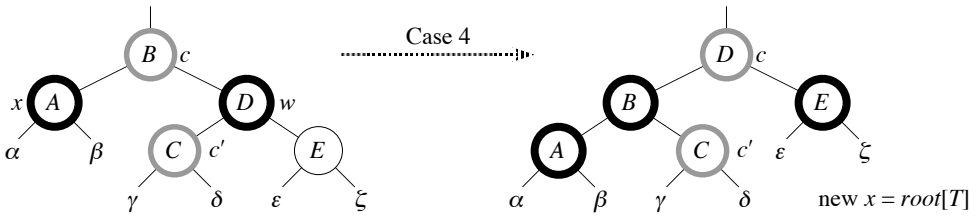
- Take 1 black off x ($\Rightarrow$ singly black) and off w ($\Rightarrow$ red).
- Move that black to $p[x]$.
- Do the next iteration with $p[x]$ as the new x.
- If entered this case from case 1, then $p[x]$ was red $\Rightarrow$ new x is red & black $\Rightarrow$ color attribute of new x is RED $\Rightarrow$ loop terminates. Then new x is made black in the last line.

Case 3: w is black, w's left child is red, and w's right child is black



- Make w red and w's left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Case 4: w is black, w's left child is black, and w's right child is red



[Now there are two nodes of unknown colors, denoted by c and c'.]

- Make w be $p[x]$'s color (c).
- Make $p[x]$ black and w's right child black.
- Then left rotate on $p[x]$.
- Remove the extra black on x ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

Analysis

$O(\lg n)$ time to get through RB-DELETE up to the call of RB-DELETE-FIXUP.

Within RB-DELETE-FIXUP:

- Case 2 is the only case in which more iterations occur.
  - x moves up 1 level.
  - Hence, $O(\lg n)$ iterations.
- Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
- Hence, $O(\lg n)$ time.

[In Chapter 14, we'll see a theorem that relies on red-black tree operations causing at most a constant number of rotations. This is where red-black trees enjoy an advantage over AVL trees: in the worst case, an operation on an n-node AVL tree causes $\Omega(\lg n)$ rotations.]

Solutions for Chapter 13: Red-Black Trees

Solution to Exercise 13.1-3

If we color the root of a relaxed red-black tree black but make no other changes, the resulting tree is a red-black tree. Not even any black-heights change.

Solution to Exercise 13.1-4

After absorbing each red node into its black parent, the degree of each node black node is

- 2, if both children were already black,
- 3, if one child was black and one was red, or
- 4, if both children were red.

All leaves of the resulting tree have the same depth.

Solution to Exercise 13.1-5

In the longest path, at least every other node is black. In the shortest path, at most every node is black. Since the two paths contain equal numbers of black nodes, the length of the longest path is at most twice the length of the shortest path.

We can say this more precisely, as follows:

Since every path contains $\mathrm{bh}(x)$ black nodes, even the shortest path from x to a descendant leaf has length at least $\mathrm{bh}(x)$. By definition, the longest path from x to a descendant leaf has length $\mathrm{height}(x)$. Since the longest path has $\mathrm{bh}(x)$ black nodes and at least half the nodes on the longest path are black (by property 4), $\mathrm{bh}(x) \geq \mathrm{height}(x)/2$, so

$$\text{length of longest path} = \mathrm{height}(x) \leq 2 \cdot \mathrm{bh}(x) \leq \text{ twice length of shortest path .}$$

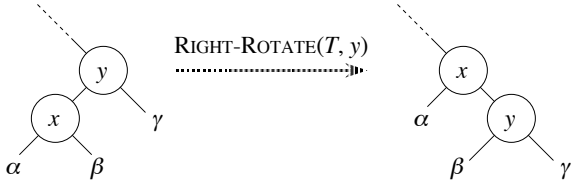
Solution to Exercise 13.2-4

Since the exercise asks about binary search trees rather than the more specific red-black trees, we assume here that leaves are full-fledged nodes, and we ignore the sentinels.

Taking the book's hint, we start by showing that with at most $n - 1$ right rotations, we can convert any binary search tree into one that is just a right-going chain.

The idea is simple. Let us define the ***right spine*** as the root and all descendants of the root that are reachable by following only *right* pointers from the root. A binary search tree that is just a right-going chain has all n nodes in the right spine.

As long as the tree is not just a right spine, repeatedly find some node y on the right spine that has a non-leaf left child x and then perform a right rotation on y:



(In the above figure, note that any of α, β, and γ can be an empty subtree.)

Observe that this right rotation adds x to the right spine, and no other nodes leave the right spine. Thus, this right rotation increases the number of nodes in the right spine by 1. Any binary search tree starts out with at least one node—the root—in the right spine. Moreover, if there are any nodes not on the right spine, then at least one such node has a parent on the right spine. Thus, at most $n - 1$ right rotations are needed to put all nodes in the right spine, so that the tree consists of a single right-going chain.

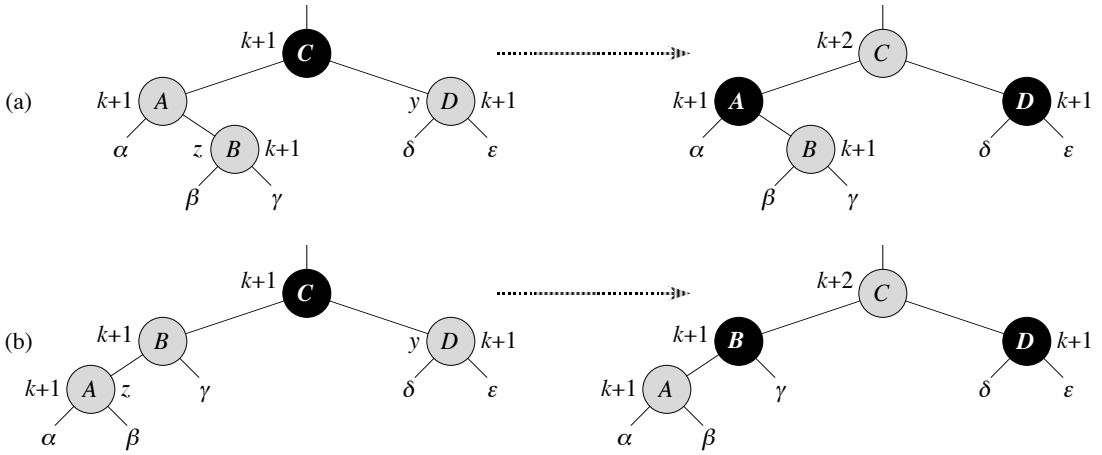
If we knew the sequence of right rotations that transforms an arbitrary binary search tree T to a single right-going chain T', then we could perform this sequence in reverse—turning each right rotation into its inverse left rotation—to transform T' back into T.

Therefore, here is how we can transform any binary search tree T_1 into any other binary search tree T_2. Let T' be the unique right-going chain consisting of the nodes of T_1 (which is the same as the nodes of T_2). Let $r = \langle r_1, r_2, \ldots, r_k \rangle$ be a sequence of right rotations that transforms T_1 to T', and let $r' = \langle r'_1, r'_2, \ldots, r'_{k'} \rangle$ be a sequence of right rotations that transforms T_2 to T'. We know that there exist sequences r and r' with $k, k' \leq n - 1$. For each right rotation r'_i, let l'_i be the corresponding inverse left rotation. Then the sequence $\langle r_1, r_2, \ldots, r_k, l'_{k'}, l'_{k'-1}, \ldots, l'_2, l'_1 \rangle$ transforms T_1 to T_2 in at most $2n - 2$ rotations.

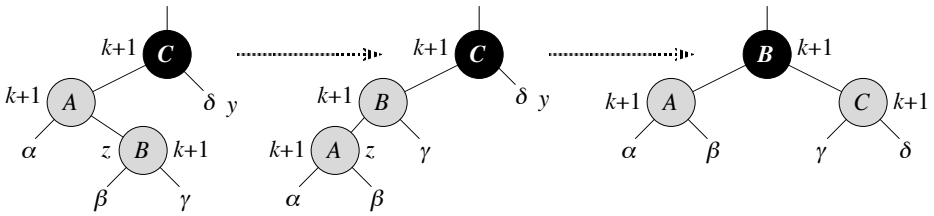
Solution to Exercise 13.3-3

In Figure 13.5, nodes A, B, and D have black-height $k+1$ in all cases, because each of their subtrees has black-height k and a black root. Node C has black-height $k+1$

on the left (because its red children have black-height $k+1$) and black-height $k+2$ on the right (because its black children have black-height $k+1$).



In Figure 13.6, nodes A, B, and C have black-height $k+1$ in all cases. At left and in the middle, each of A's and B's subtrees has black-height k and a black root, while C has one such subtree and a red child with black-height $k+1$. At the right, each of A's and C's subtrees has black-height k and a black root, while B's red children each have black-height $k+1$.



Property 5 is preserved by the transformations. We have shown above that the black-height is well-defined within the subtrees pictured, so property 5 is preserved within those subtrees. Property 5 is preserved for the tree containing the subtrees pictured, because every path through these subtrees to a leaf contributes $k+2$ black nodes.

Solution to Exercise 13.3-4

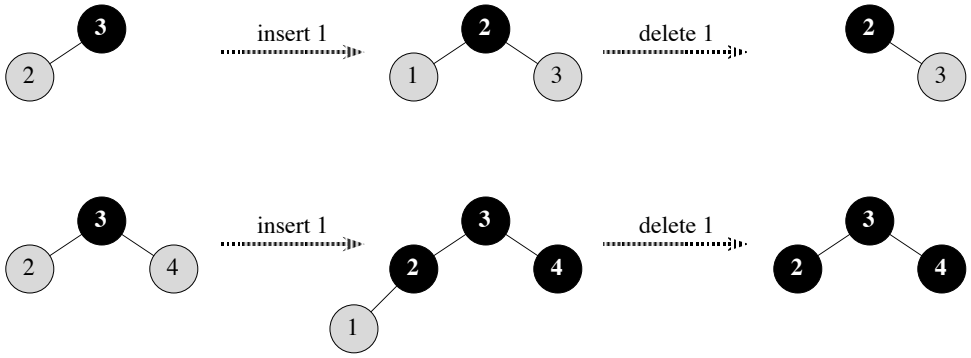
Colors are set to red only in cases 1 and 3, and in both situations, it is $p[p[z]]$ that is reddened. If $p[p[z]]$ is the sentinel, then $p[z]$ is the root. By part (b) of the loop invariant and line 1 of RB-INSERT-FIXUP, if $p[z]$ is the root, then we have dropped out of the loop. The only subtlety is in case 2, where we set $z \leftarrow p[z]$ before coloring $p[p[z]]$ red. Because we rotate before the recoloring, the identity of $p[p[z]]$ is the same before and after case 2, so there's no problem.

Solution to Exercise 13.4-6

Case 1 occurs only if x's sibling w is red. If $p[x]$ were red, then there would be two reds in a row, namely $p[x]$ (which is also $p[w]$) and w, and we would have had these two reds in a row even before calling RB-DELETE.

Solution to Exercise 13.4-7

No, the red-black tree will not necessarily be the same. Here are two examples: one in which the tree's shape changes, and one in which the shape remains the same but the node colors change.



Solution to Problem 13-1

a. When inserting key k, all nodes on the path from the root to the added node (a new leaf) must change, since the need for a new child pointer propagates up from the new node to all of its ancestors.

When deleting a node, let y be the node actually removed and z be the node given to the delete procedure.

- If y has at most one child, it will be removed or spliced out (see Figure 12.4, parts (a) and (b), where y and z are the same node). All ancestors of y will be changed. (As with insertion, the need for a new child pointer propagates up from the removed node.)
- If z has two children, y is its successor; it is y that will be spliced out and moved to z's position (see Figure 12.4(c)). Therefore all ancestors of both z and y must be changed. (Actually, this is just all ancestors of z, since z is an ancestor of y in this case.)

In either case, y's children (if any) are unchanged, because we have assumed that there is no parent field.

b. We assume that we can call two procedures:

- MAKE-NEW-NODE(k) creates a new node whose *key* field has value k and with *left* and *right* fields NIL, and it returns a pointer to the new node.
- COPY-NODE(x) creates a new node whose *key*, *left*, and *right* fields have the same values as those of node x, and it returns a pointer to the new node.

Here are two ways to write PERSISTENT-TREE-INSERT. The first is a version of TREE-INSERT, modified to create new nodes along the path to where the new node will go, and to not use parent fields. It returns the root of the new tree.

```
PERSISTENT-TREE-INSERT(T, k)
z ← MAKE-NEW-NODE(k)
new-root ← COPY-NODE(root[T])
y ← NIL
x ← new-root
while x ≠ NIL
    do y ← x
       if key[z] < key[x]
         then x ← COPY-NODE(left[x])
              left[y] ← x
         else x ← COPY-NODE(right[x])
              right[y] ← x
if y = NIL
  then new-root ← z
  else if key[z] < key[y]
         then left[y] ← z
         else right[y] ← z
return new-root
```

The second is a rather elegant recursive procedure. It must be called with $root[T]$ instead of T as its first argument (because the recursive calls pass a node for this argument), and it returns the root of the new tree.

```
PERSISTENT-TREE-INSERT(r, k)
if r = NIL
  then x ← MAKE-NEW-NODE(k)
  else x ← COPY-NODE(r)
       if k < key[r]
         then left[x] ← PERSISTENT-TREE-INSERT(left[r], k)
         else right[x] ← PERSISTENT-TREE-INSERT(right[r], k)
return x
```

c. Like TREE-INSERT, PERSISTENT-TREE-INSERT does a constant amount of work at each node along the path from the root to the new node. Since the length of the path is at most h, it takes $O(h)$ time.

Since it allocates a new node (a constant amount of space) for each ancestor of the inserted node, it also needs $O(h)$ space.

d. If there were parent fields, then because of the new root, every node of the tree would have to be copied when a new node is inserted. To see why, observe that the children of the root would change to point to the new root, then their children would change to point to them, and so on. Since there are n nodes, this change would cause insertion to create $\Omega(n)$ new nodes and to take $\Omega(n)$ time.

e. From parts (a) and (c), we know that insertion into a persistent binary search tree of height h, like insertion into an ordinary binary search tree, takes worst-case time $O(h)$. A red-black tree has $h = O(\lg n)$, so insertion into an ordinary red-black tree takes $O(\lg n)$ time. We need to show that if the red-black tree is persistent, insertion can still be done in $O(\lg n)$ time. To do this, we will need to show two things:

- How to still find the parent pointers we need in $O(1)$ time without using a parent field. We cannot use a parent field because a persistent tree with parent fields uses $\Omega(n)$ time for insertion (by part (d)).
- That the additional node changes made during red-black tree operations (by rotation and recoloring) don't cause more than $O(\lg n)$ additional nodes to change.

Each parent pointer needed during insertion can be found in $O(1)$ time without having a parent field as follows:

To insert into a red-black tree, we call RB-INSERT, which in turn calls RB-INSERT-FIXUP. Make the same changes to RB-INSERT as we made to TREE-INSERT for persistence. Additionally, as RB-INSERT walks down the tree to find the place to insert the new node, have it build a stack of the nodes it traverses and pass this stack to RB-INSERT-FIXUP. RB-INSERT-FIXUP needs parent pointers to walk back up the same path, and at any given time it needs parent pointers only to find the parent and grandparent of the node it is working on. As RB-INSERT-FIXUP moves up the stack of parents, it needs only parent pointers that are at known locations a constant distance away in the stack. Thus, the parent information can be found in $O(1)$ time, just as if it were stored in a parent field.

Rotation and recoloring change nodes as follows:

- RB-INSERT-FIXUP performs at most 2 rotations, and each rotation changes the child pointers in 3 nodes (the node around which we rotate, that node's parent, and one of the children of the node around which we rotate). Thus, at most 6 nodes are directly modified by rotation during RB-INSERT-FIXUP. In a persistent tree, all ancestors of a changed node are copied, so RB-INSERT-FIXUP's rotations take $O(\lg n)$ time to change nodes due to rotation. (Actually, the changed nodes in this case share a single $O(\lg n)$-length path of ancestors.)
- RB-INSERT-FIXUP recolors some of the inserted node's ancestors, which are being changed anyway in persistent insertion, and some children of ancestors (the "uncles" referred to in the algorithm description). There are at most $O(\lg n)$ ancestors, hence at most $O(\lg n)$ color changes of uncles. Recoloring uncles doesn't cause any additional node changes due to persistence, because the ancestors of the uncles are the same nodes (ancestors of

the inserted node) that are being changed anyway due to persistence. Thus, recoloring does not affect the $O(\lg n)$ running time, even with persistence.

We could show similarly that deletion in a persistent tree also takes worst-case time $O(h)$.

- We already saw in part (a) that $O(h)$ nodes change.
- We could write a persistent RB-DELETE procedure that runs in $O(h)$ time, analogous to the changes we made for persistence in insertion. But to do so without using parent pointers we need to walk down the tree to the node to be deleted, to build up a stack of parents as discussed above for insertion. This is a little tricky if the set's keys are not distinct, because in order to find the path to the node to delete—a particular node with a given key—we have to make some changes to how we store things in the tree, so that duplicate keys can be distinguished. The easiest way is to have each key take a second part that is unique, and to use this second part as a tiebreaker when comparing keys.

Then the problem of showing that deletion needs only $O(\lg n)$ time in a persistent red-black tree is the same as for insertion.

- As for insertion, we can show that the parents needed by RB-DELETE-FIXUP can be found in $O(1)$ time (using the same technique as for insertion).
- Also, RB-DELETE-FIXUP performs at most 3 rotations, which as discussed above for insertion requires $O(\lg n)$ time to change nodes due to persistence. It also does $O(\lg n)$ color changes, which (as for insertion) take only $O(\lg n)$ time to change ancestors due to persistence, because the number of copied nodes is $O(\lg n)$.

Lecture Notes for Chapter 14: Augmenting Data Structures

Chapter 14 overview

We'll be looking at methods for *designing* algorithms. In some cases, the design will be intermixed with analysis. In other cases, the analysis is easy, and it's the design that's harder.

Augmenting data structures

- It's unusual to have to design an all-new data structure from scratch.
- It's more common to take a data structure that you know and store additional information in it.
- With the new information, the data structure can support new operations.
- But... you have to figure out how to *correctly maintain* the new information *without loss of efficiency*.

We'll look at a couple of situations in which we augment red-black trees.

Dynamic order statistics

We want to support the usual dynamic-set operations from R-B trees, plus:

- OS-SELECT(x, i): return pointer to node containing the ith smallest key of the subtree rooted at x.
- OS-RANK(T, x): return the rank of x in the linear order determined by an inorder walk of T.

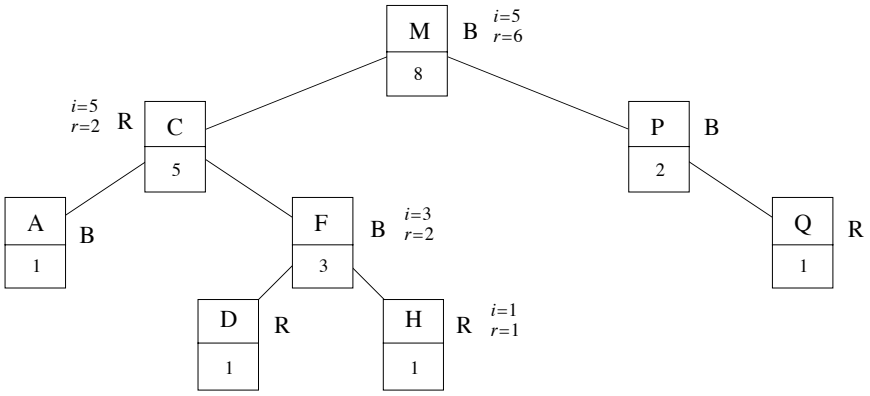
Augment by storing in each node x:

$size[x] =$ # of nodes in subtree rooted at x .

- Includes x itself.
- Does not include leaves (sentinels).

Define for sentinel $size[nil[T]] = 0$.

Then $size[x] = size[left[x]] + size[right[x]] + 1$.



*[**Example above:** Ignore colors, but legal coloring shown with "R" and "B" notations. Values of i and r are for the example below.]*

Note: OK for keys to not be distinct. Rank is defined with respect to position in inorder walk. So if we changed D to C, rank of original C is 2, rank of D changed to C is 3.

OS-SELECT(x, i)
$r \leftarrow size[left[x]]+1$
if $i = r$
  then return x
elseif $i < r$
  then return OS-SELECT$(left[x], i)$
else return OS-SELECT$(right[x], i - r)$

Initial call: OS-SELECT$(root[T], i)$

Try OS-SELECT$(root[T], 5)$. *[Values shown in figure above. Returns node whose key is H.]*

Correctness: $r =$ rank of x within subtree rooted at x.

- If $i = r$, then we want x.
- If $i < r$, then ith smallest element is in x's left subtree, and we want the ith smallest element in the subtree.
- If $i > r$, then ith smallest element is in x's right subtree, but subtract off the r elements in x's subtree that precede those in x's right subtree.
- Like the randomized SELECT algorithm!

Analysis: Each recursive call goes down one level. Since R-B tree has $O(\lg n)$ levels, have $O(\lg n)$ calls $\Rightarrow O(\lg n)$ time.

OS-RANK(T, x)
$r \leftarrow size[left[x]] + 1$
$y \leftarrow x$
while $y \neq root[T]$
  do if $y = right[p[y]]$
    then $r \leftarrow r + size[left[p[y]]] + 1$
  $y \leftarrow p[y]$
return r

Demo: Node D.

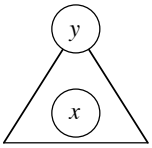
Why does this work?

> **Loop invariant:** At start of each iteration of **while** loop, r = rank of $key[x]$ in subtree rooted at y.

Initialization: Initially, r = rank of $key[x]$ in subtree rooted at x, and $y = x$.

Termination: Loop terminates when $y = root[T] \Rightarrow$ subtree rooted at y is entire tree. Therefore, r = rank of $key[x]$ in entire tree.

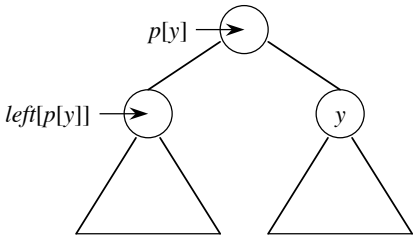
Maintenance: At end of each iteration, set $y \leftarrow p[y]$. So, show that if r = rank of $key[x]$ in subtree rooted at y at start of loop body, then r = rank of $key[x]$ in subtree rooted at $p[y]$ at end of loop body.



[r = # of nodes in subtree rooted at y preceding x in inorder walk]

Must add nodes in y's sibling's subtree.

- If y is a left child, its sibling's subtree follows all nodes in y's subtree $\Rightarrow$ don't change r.
- If y is a right child, all nodes in y's sibling's subtree precede all nodes in y's subtree $\Rightarrow$ add size of y's sibling's subtree, plus 1 for $p[y]$, into r.



Analysis: y goes up one level in each iteration $\Rightarrow O(\lg n)$ time.

Maintaining subtree sizes

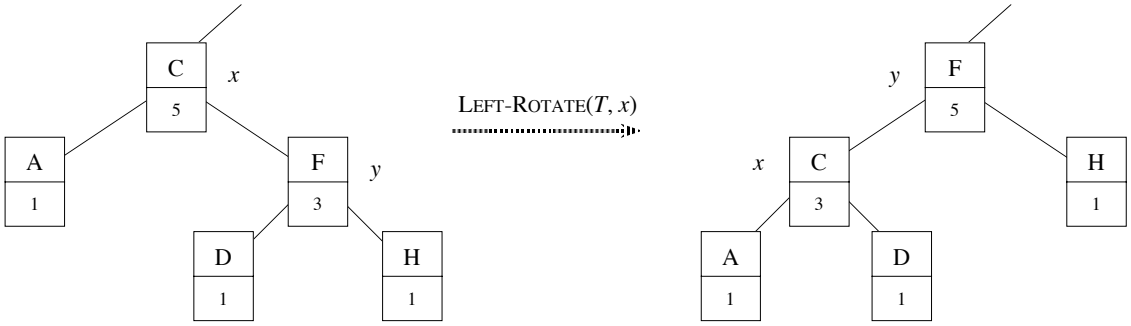
- Need to maintain $size[x]$ fields during insert and delete operations.
- Need to maintain them efficiently. Otherwise, might have to recompute them all, at a cost of $\Omega(n)$.

Will see how to maintain without increasing $O(\lg n)$ time for insert and delete.

Insert:

- During pass downward, we know that the new node will be a descendant of each node we visit, and only of these nodes. Therefore, increment *size* field of each node visited.

- Then there's the fixup pass:
  - Goes up the tree.
  - Changes colors $O(\lg n)$ times.
  - Performs ≤ 2 rotations.
- Color changes don't affect subtree sizes.
- Rotations do!
- But we can determine new sizes based on old sizes and sizes of children.



$$size[y] \leftarrow size[x]$$
$$size[x] \leftarrow size[left[x]] + size[right[x]] + 1$$

- Similar for right rotation.
- Therefore, can update in $O(1)$ time per rotation $\Rightarrow$ $O(1)$ time spent updating *size* fields during fixup.
- Therefore, $O(\lg n)$ to insert.

Delete: Also 2 phases:

1. Splice out some node y.
2. Fixup.

After splicing out y, traverse a path $y \rightarrow root$, decrementing *size* in each node on path. $O(\lg n)$ time.

During fixup, like insertion, only color changes and rotations.

- ≤ 3 rotations $\Rightarrow$ $O(1)$ time spent updating *size* fields during fixup.
- Therefore, $O(\lg n)$ to delete.

Done!

Methodology for augmenting a data structure

1. Choose an underlying data structure.
2. Determine additional information to maintain.

3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations.

Don't need to do these steps in strict order! Usually do a little of each, in parallel.

How did we do them for OS trees?

1. R-B tree.
2. $size[x]$.
3. Showed how to maintain *size* during insert and delete.
4. Developed OS-SELECT and OS-RANK.

Red-black trees are particularly amenable to augmentation.

Theorem

Augment a R-B tree with field f, where $f[x]$ depends only on information in x, $left[x]$, and $right[x]$ (including $f[left[x]]$ and $f[right[x]]$). Then can maintain values of f in all nodes during insert and delete without affecting $O(\lg n)$ performance.

Proof Since $f[x]$ depends only on x and its children, when we alter information in x, changes propagate only upward (to $p[x]$, $p[p[x]]$, . . . , *root*).

Height = $O(\lg n) \Rightarrow O(\lg n)$ updates, at $O(1)$ each.

Insertion: Insert a node as child of existing node. Even if can't update f on way down, can go up from inserted node to update f. During fixup, only changes come from color changes (no effect on f) and rotations. Each rotation affects f of ≤ 3 nodes (x,y, and parent), and can recompute each in $O(1)$ time. Then, if necessary, propagate changes up the tree. Therefore, $O(\lg n)$ time per rotation. Since ≤ 2 rotations, $O(\lg n)$ time to update f during fixup.

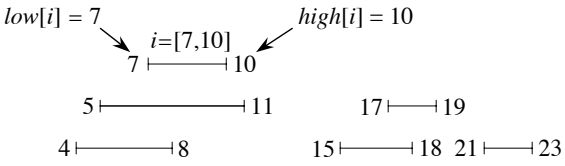
Delete: Same idea. After splicing out a node, go up from there to update f. Fixup has ≤ 3 rotations. $O(\lg n)$ per rotation $\Rightarrow O(\lg n)$ to update f during fixup.

■ (theorem)

For some attributes, can get away with $O(1)$ per rotation. Example: *size* field.

Interval trees

Maintain a set of intervals. For instance, time intervals.



[leave on board]

Operations

- INTERVAL-INSERT(T, x): $int[x]$ already filled in.
- INTERVAL-DELETE(T, x)
- INTERVAL-SEARCH(T, i): return pointer to a node x in T such that $int[x]$ overlaps interval i. Any overlapping node in T is OK. Return pointer to sentinel $nil[T]$ if no overlapping node in T.

Interval i has $low[i]$, $high[i]$.

i and j overlap if and only if $low[i] \le high[j]$ and $low[j] \le high[i]$.

(Go through examples of proper inclusion, overlap without proper inclusion, no overlap.)

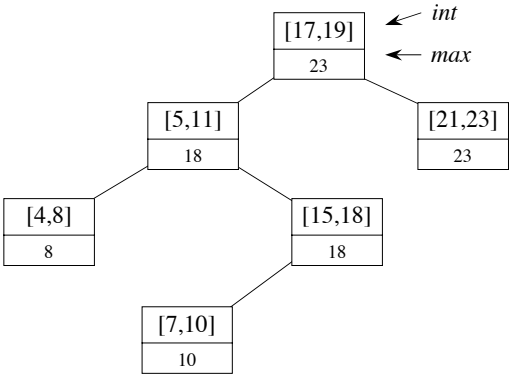
Another way: i and j *don't* overlap if and only if: $low[i] > high[j]$ or $low[j] > high[i]$. *[leave this on board]*

Recall the 4-part methodology.

For interval trees

1. Use R-B trees.
   - Each node x contains interval $int[x]$.
   - Key is low endpoint ($low[int[x]]$).
   - Inorder walk would list intervals sorted by low endpoint.
2. Each node x contains

   $max[x]$ = max endpoint value in subtree rooted at x .



   [leave on board]

$$max[x] = max \begin{cases} high[int[x]] , \\ max[left[x]] , \\ max[right[x]] \end{cases}$$

   Could $max[left[x]] > max[right[x]]$? Sure. Position in tree is determined only by low endpoints, not high endpoints.
3. Maintaining the information.

- This is easy—$max[x]$ depends only on:
  - information in x: $high[int[x]]$
  - information in $left[x]$: $max[left[x]]$
  - information in $right[x]$: $max[right[x]]$
- Apply the theorem.
- In fact, can update max on way down during insertion, and in $O(1)$ time per rotation.

4. Developing new operations.

INTERVAL-SEARCH(T, i)
$x \leftarrow root[T]$
while $x \neq nil[T]$ and i does not overlap $int[x]$
  do if $left[x] \neq nil[T]$ and $max[left[x]] \geq low[i]$
    then $x \leftarrow left[x]$
    else $x \leftarrow right[x]$
return x

Examples: Search for [14, 16] and [12, 14].

Time: $O(\lg n)$.

Correctness: Key idea: need check only 1 of node's 2 children.

Theorem
If search goes right, then either:

- There is an overlap in right subtree, or
- There is no overlap in either subtree.

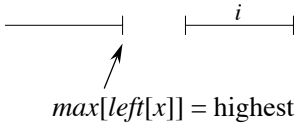
If search goes left, then either:

- There is an overlap in left subtree, or
- There is no overlap in either subtree.

Proof If search goes right:

- If there is an overlap in right subtree, done.
- If there is no overlap in right, show there is no overlap in left. Went right because
  - $left[x] = nil[T] \Rightarrow$ no overlap in left.

    OR
  - $max[left[x]] < low[i] \Rightarrow$ no overlap in left.



If search goes left:

- If there is an overlap in left subtree, done.
- If there is no overlap in left, show there is no overlap in right.
  - Went left because:
    $low[i] \leq max[left[x]]$
    $= high[j]$ for some j in left subtree .
  - Since there is no overlap in left, i and j don't overlap.
  - Refer back to: no overlap if
    $low[i] > high[j]$ or $low[j] > high[i]$.
  - Since $low[i] \leq high[j]$, must have $low[j] > high[i]$.
  - Now consider *any* interval k in *right* subtree.
  - Because keys are low endpoint,
    $\underbrace{low[j]}_{\text{in left}} \leq \underbrace{low[k]}_{\text{in right}}$.
  - Therefore, $high[i] < low[j] \leq low[k]$.
  - Therefore, $high[i] < low[k]$.
  - Therefore, i and k do not overlap. ■ (theorem)

Solutions for Chapter 14: Augmenting Data Structures

Solution to Exercise 14.1-5

Given an element x in an n-node order-statistic tree T and a natural number i, the following procedure retrieves the ith successor of x in the linear order of T:

OS-SUCCESSOR(T, x, i)
$r \leftarrow$ OS-RANK(T, x)
$s \leftarrow r + i$
return OS-SELECT$(root[T], s)$

Since OS-RANK and OS-SELECT each take $O(\lg n)$ time, so does the procedure OS-SUCCESSOR.

Solution to Exercise 14.1-6

When inserting node z, we search down the tree for the proper place for z. For each node x on this path, add 1 to $rank[x]$ if y is inserted within x's left subtree, and leave $rank[x]$ unchanged if y is inserted within x's right subtree. Similarly when deleting, subtract 1 from $rank[x]$ whenever the spliced-out node y had been in x's left subtree.

We also need to handle the rotations that occur during the fixup procedures for insertion and deletion. Consider a left rotation on node x, where the pre-rotation right child of x is y (so that x becomes y's left child after the left rotation). We leave $rank[x]$ unchanged, and letting $r = rank[y]$ before the rotation, we set $rank[y] \leftarrow r + rank[x]$. Right rotations are handled in an analogous manner.

Solution to Exercise 14.1-7

Let $A[1 \ldots n]$ be the array of n distinct numbers.

One way to count the inversions is to add up, for each element, the number of larger elements that precede it in the array:

$$\#\text{ of inversions} = \sum_{j=1}^{n} |Inv(j)| \ ,$$

where $Inv(j) = \{i : i < j \text{ and } A[i] > A[j]\}$.

Note that $|Inv(j)|$ is related to $A[j]$'s rank in the subarray $A[1..j]$ because the elements in $Inv(j)$ are the reason that $A[j]$ is not positioned according to its rank. Let $r(j)$ be the rank of $A[j]$ in $A[1..j]$. Then $j = r(j) + |Inv(j)|$, so we can compute

$$|Inv(j)| = j - r(j)$$

by inserting $A[1], \ldots, A[n]$ into an order-statistic tree and using OS-RANK to find the rank of each $A[j]$ in the tree immediately after it is inserted into the tree. (This OS-RANK value is $r(j)$.)

Insertion and OS-RANK each take $O(\lg n)$ time, and so the total time for n elements is $O(n \lg n)$.

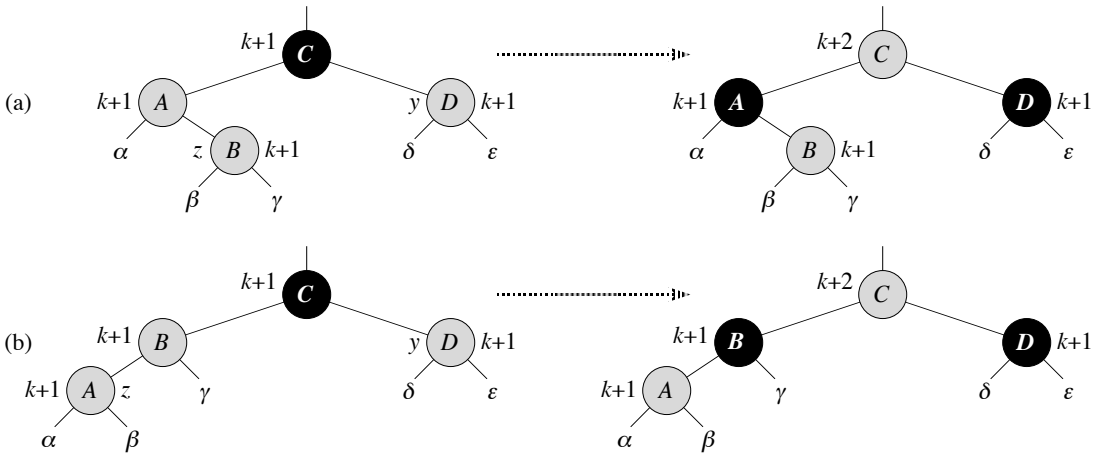
Solution to Exercise 14.2-2

Yes, by Theorem 14.1, because the black-height of a node can be computed from the information at the node and its two children. Actually, the black-height can be computed from just one child's information: the black-height of a node is the black-height of a red child, or the black height of a black child plus one. The second child does not need to be checked because of property 5 of red-black trees.

Within the RB-INSERT-FIXUP and RB-DELETE-FIXUP procedures are color changes, each of which potentially cause $O(\lg n)$ black-height changes. Let us show that the color changes of the fixup procedures cause only local black-height changes and thus are constant-time operations. Assume that the black-height of each node x is kept in the field $bh[x]$.

For RB-INSERT-FIXUP, there are 3 cases to examine.

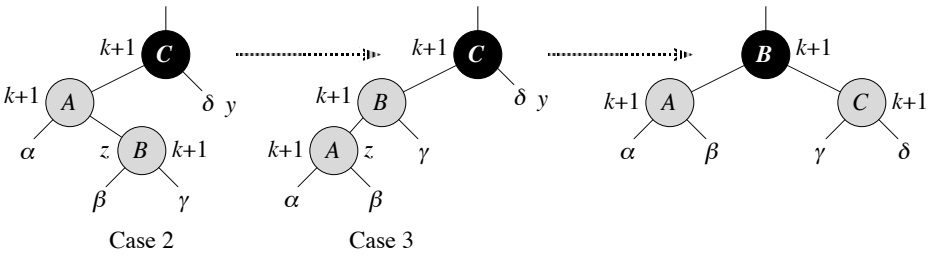
Case 1: z's uncle is red.



- Before color changes, suppose that all subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ have the same black-height k with a black root, so that nodes A, B, C, and D have black-heights of $k + 1$.
- After color changes, the only node whose black-height changed is node C. To fix that, add $bh[p[p[z]]] = bh[p[p[z]]] + 1$ after line 7 in RB-INSERT-FIXUP.
- Since the number of black nodes between $p[p[z]]$ and z remains the same, nodes above $p[p[z]]$ are not affected by the color change.

Case 2: z's uncle y is black, and z is a right child.

Case 3: z''s uncle y is black, and z is a left child.

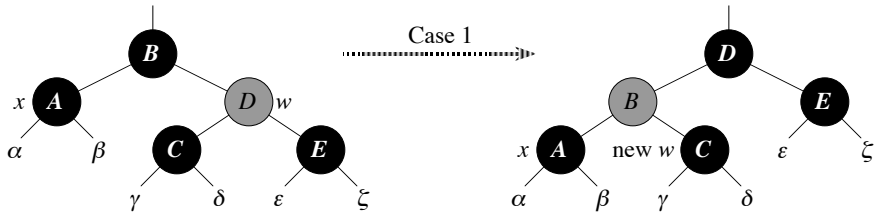


- With subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ of black-height k, we see that even with color changes and rotations, the black-heights of nodes A, B, and C remain the same ($k + 1$).

Thus, RB-INSERT-FIXUP maintains its original $O(\lg n)$ time.

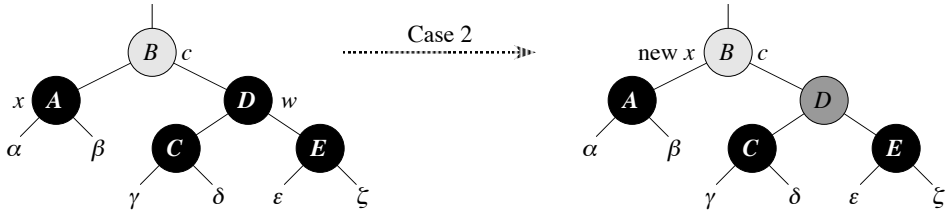
For RB-DELETE-FIXUP, there are 4 cases to examine.

Case 1: x's sibling w is red.



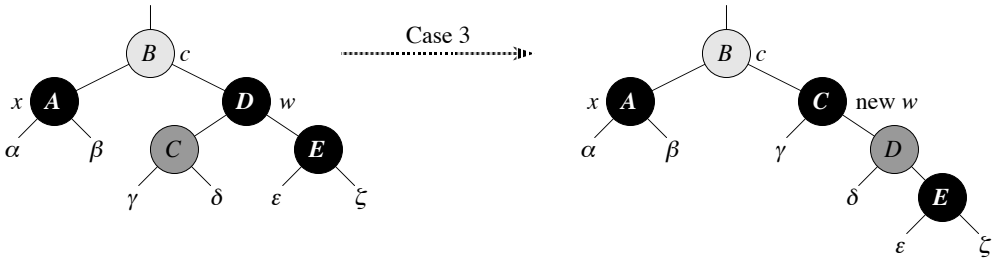
- Even though case 1 changes colors of nodes and does a rotation, black-heights are not changed.
- Case 1 changes the structure of the tree, but waits for cases 2, 3, and 4 to deal with the "extra black" on x.

Case 2: x's sibling w is black, and both of w's children are black.



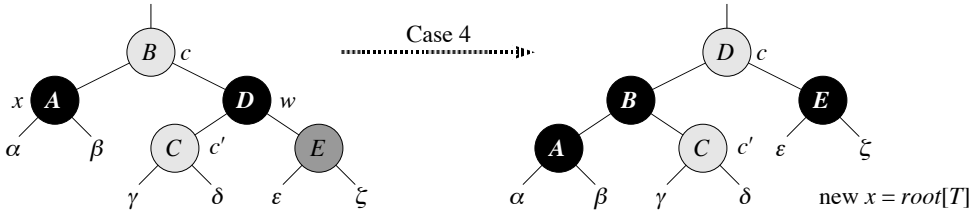
- w is colored red, and x's "extra" black is moved up to $p[x]$.
- Now we can add $bh[p[x]] = bh[x]$ after line 10 in RB-DELETE-FIXUP.
- This is a constant-time update. Then, keep looping to deal with the extra black on $p[x]$.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black.



- Regardless of the color changes and rotation of this case, the black-heights don't change.
- Case 3 just sets up the structure of the tree, so it can fall correctly into case 4.

Case 4: x's sibling w is black, and w's right child is red.



- Nodes A, C, and E keep the same subtrees, so their black-heights don't change.
- Add these two constant-time assignments in RB-DELETE-FIXUP after line 20:
  $bh[p[x]] = bh[x] + 1$.
  $bh[p[p[x]]] = bh[p[x]] + 1$.
- The extra black is taken care of. Loop terminates.

Thus, RB-DELETE-FIXUP maintains its original $O(\lg n)$ time.

Therefore, we conclude that black-heights of nodes can be maintained as fields in red-black trees without affecting the asymptotic performance of red-black tree operations.

Solution to Exercise 14.2-3

No, because the depth of a node depends on the depth of its parent. When the depth of a node changes, the depths of all nodes below it in the tree must be updated. Updating the root node causes $n - 1$ other nodes to be updated, which would mean that operations on the tree that change node depths might not run in $O(n \lg n)$ time.

Solution to Exercise 14.3-3

As it travels down the tree, INTERVAL-SEARCH first checks whether current node x overlaps the query interval i and, if it does not, goes down to either the left or right child. If node x overlaps i, and some node in the right subtree overlaps i, but no node in the left subtree overlaps i, then because the keys are low endpoints, this order of checking (first x, then one child) will return the overlapping interval with the minimum low endpoint. On the other hand, if there is an interval that overlaps i in the left subtree of x, then checking x before the left subtree might cause the procedure to return an interval whose low endpoint is not the minimum of those that overlap i. Therefore, if there is a possibility that the left subtree might contain an interval that overlaps i, we need to check the left subtree first. If there is no overlap in the left subtree but node x overlaps i, then we return x. We check the right subtree under the same conditions as in INTERVAL-SEARCH: the left subtree cannot contain an interval that overlaps i, and node x does not overlap i, either.

Because we might search the left subtree first, it is easier to write the pseudocode to use a recursive procedure MIN-INTERVAL-SEARCH-FROM(T, x, i), which returns the node overlapping i with the minimum low endpoint in the subtree rooted at x, or $nil[T]$ if there is no such node.

```
MIN-INTERVAL-SEARCH(T, i)
return MIN-INTERVAL-SEARCH-FROM(T, root[T], i)

MIN-INTERVAL-SEARCH-FROM(T, x, i)
if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
  then y ← MIN-INTERVAL-SEARCH-FROM(T, left[x], i)
       if y ≠ nil[T]
         then return y
       elseif i overlaps int[x]
         then return x
       else return nil[T]
elseif i overlaps int[x]
  then return x
else return MIN-INTERVAL-SEARCH-FROM(T, right[x], i)
```

The call MIN-INTERVAL-SEARCH(T, i) takes $O(\lg n)$ time, since each recursive call of MIN-INTERVAL-SEARCH-FROM goes one node lower in the tree, and the height of the tree is $O(\lg n)$.

Solution to Exercise 14.3-6

1. Underlying data structure:
   A red-black tree in which the numbers in the set are stored simply as the keys of the nodes.

SEARCH is then just the ordinary TREE-SEARCH for binary search trees, which runs in $O(\lg n)$ time on red-black trees.

2. Additional information:
   The red-black tree is augmented by the following fields in each node x:
   - *min-gap*$[x]$ contains the minimum gap in the subtree rooted at x. It has the magnitude of the difference of the two closest numbers in the subtree rooted at x. If x is a leaf (its children are all *nil*$[T]$), let *min-gap*$[x] = \infty$.
   - *min-val*$[x]$ contains the minimum value (key) in the subtree rooted at x.
   - *max-val*$[x]$ contains the maximum value (key) in the subtree rooted at x.
3. Maintaining the information:
   The three fields added to the tree can each be computed from information in the node and its children. Hence by Theorem 14.1, they can be maintained during insertion and deletion without affecting the $O(\lg n)$ running time:

$$\textit{min-val}[x] = \begin{cases} \textit{min-val}[\textit{left}[x]] & \text{if there's a left subtree ,} \\ \textit{key}[x] & \text{otherwise ,} \end{cases}$$

$$\textit{max-val}[x] = \begin{cases} \textit{max-val}[\textit{right}[x]] & \text{if there's a right subtree ,} \\ \textit{key}[x] & \text{otherwise ,} \end{cases}$$

$$\textit{min-gap}[x] = \min \begin{cases} \textit{min-gap}[\textit{left}[x]] & (\infty \text{ if no left subtree}) , \\ \textit{min-gap}[\textit{right}[x]] & (\infty \text{ if no right subtree}) , \\ \textit{key}[x] - \textit{max-val}[\textit{left}[x]] & (\infty \text{ if no left subtree}) , \\ \textit{min-val}[\textit{right}[x]] - \textit{key}[x] & (\infty \text{ if no right subtree}) . \end{cases}$$

   In fact, the reason for defining the *min-val* and *max-val* fields is to make it possible to compute *min-gap* from information at the node and its children.
4. New operation:
   MIN-GAP simply returns the *min-gap* stored at the tree root. Thus, its running time is $O(1)$.

   Note that in addition (not asked for in the exercise), it is possible to find the two closest numbers in $O(\lg n)$ time. Starting from the root, look for where the minimum gap (the one stored at the root) came from. At each node x, simulate the computation of *min-gap*$[x]$ to figure out where *min-gap*$[x]$ came from. If it came from a subtree's *min-gap* field, continue the search in that subtree. If it came from a computation with x's key, then x and that other number are the closest numbers.

Solution to Exercise 14.3-7

General idea: Move a sweep line from left to right, while maintaining the set of rectangles currently intersected by the line in an interval tree. The interval tree will organize all rectangles whose x interval includes the current position of the sweep line, and it will be based on the y intervals of the rectangles, so that any overlapping y intervals in the interval tree correspond to overlapping rectangles.

Details:

1. Sort the rectangles by their x-coordinates. (Actually, each rectangle must appear twice in the sorted list—once for its left x-coordinate and once for its right x-coordinate.)
2. Scan the sorted list (from lowest to highest x-coordinate).
   - When an x-coordinate of a left edge is found, check whether the rectangle's y-coordinate interval overlaps an interval in the tree, and insert the rectangle (keyed on its y-coordinate interval) into the tree.
   - When an x-coordinate of a right edge is found, delete the rectangle from the interval tree.

   The interval tree always contains the set of "open" rectangles intersected by the sweep line. If an overlap is ever found in the interval tree, there are overlapping rectangles.

Time: $O(n \lg n)$

- $O(n \lg n)$ to sort the rectangles (we can use merge sort or heap sort).
- $O(n \lg n)$ for interval-tree operations (insert, delete, and check for overlap).

Solution to Problem 14-1

a. Assume for the purpose of contradiction that there is no point of maximum overlap in an endpoint of a segment. The maximum overlap point p is in the interior of m segments. Actually, p is in the interior of the intersection of those m segments. Now look at one of the endpoints p' of the intersection of the m segments. Point p' has the same overlap as p because it is in the same intersection of m segments, and so p' is also a point of maximum overlap. Moreover, p' is in the endpoint of a segment (otherwise the intersection would not end there), which contradicts our assumption that there is no point of maximum overlap in an endpoint of a segment. Thus, there is always a point of maximum overlap which is an endpoint of one of the segments.

b. Keep a balanced binary tree of the endpoints. That is, to insert an interval, we insert its endpoints separately. With each left endpoint e, associate a value $p[e] = +1$ (increasing the overlap by 1). With each right endpoint e associate a value $p[e] = -1$ (decreasing the overlap by 1). When multiple endpoints have the same value, insert all the left endpoints with that value before inserting any of the right endpoints with that value.

Here's some intuition. Let $e_1, e_2, \ldots, e_n$ be the sorted sequence of endpoints corresponding to our intervals. Let $s(i, j)$ denote the sum $p[e_i] + p[e_{i+1}] + \cdots + p[e_j]$ for $1 \le i \le j \le n$. We wish to find an i maximizing $s(1, i)$.

Each node x stores three new attributes. Suppose that the subtree rooted at x includes the endpoints $e_{l[x]}, \ldots, e_{r[x]}$. We store $v[x] = s(l[x], r[x])$, the sum of the values of all nodes in x's subtree. We also store $m[x]$, the maximum value

obtained by the expression $s(l[x], i)$ for any i in $\{l[x], l[x]+1, \ldots, r[x]\}$. Finally, we store $o[x]$ as the value of i for which $m[x]$ achieves its maximum. For the sentinel, we define $v[nil[T]] = m[nil[T]] = 0$.

We can compute these attributes in a bottom-up fashion to satisfy the requirements of Theorem 14.1:

$$v[x] = v[left[x]] + p[x] + v[right[x]] ,$$

$$m[x] = \max \begin{cases} m[left[x]] & \text{(max is in } x\text{'s left subtree)} , \\ v[left[x]] + p[x] & \text{(max is at } x\text{)} , \\ v[left[x]] + p[x] + m[right[x]] & \text{(max is in } x\text{'s right subtree)} . \end{cases}$$

The computation of $v[x]$ is straightforward. The computation of $m[x]$ bears further explanation. Recall that it is the maximum value of the sum of the p values for the nodes in x's subtree, starting at $l[x]$, which is the leftmost endpoint in x's subtree and ending at any node i in x's subtree. The value of i that maximizes this sum is either a node in x's left subtree, x itself, or a node in x's right subtree. If i is a node in x's left subtree, then $m[left[x]]$ represents a sum starting at $l[x]$, and hence $m[x] = m[left[x]]$. If i is x itself, then $m[x]$ represents the sum of all p values in x's left subtree plus $p[x]$, so that $m[x] = v[left[x]] + p[x]$. Finally, if i is in x's right subtree, then $m[x]$ represents the sum of all p values in x's left subtree, plus $p[x]$, plus the sum of some set of p values in x's right subtree. Moreover, the values taken from x's right subtree must start from the leftmost endpoint in the right subtree. To maximize this sum, we need to maximize the sum from the right subtree, and that value is precisely $m[right[x]]$. Hence, in this case, $m[x] = v[left[x]] + p[x] + m[right[x]]$.

Once we understand how to compute $m[x]$, it is straightforward to compute $o[x]$ from the information in x and its two children. Thus, we can implement the operations as follows:

- INTERVAL-INSERT: insert two nodes, one for each endpoint of the interval.
- INTERVAL-DELETE: delete the two nodes representing the interval endpoints.
- FIND-POM: return the interval whose endpoint is represented by $o[root[T]]$.

Because of how we have defined the new attributes, Theorem 14.1 says that each operation runs in $O(\lg n)$ time. In fact, FIND-POM takes only $O(1)$ time.

Solution to Problem 14-2

a. We use a circular list in which each element has two fields, *key* and *next*. At the beginning, we initialize the list to contain the keys $1, 2, \ldots, n$ in that order. This initialization takes $O(n)$ time, since there is only a constant amount of work per element (i.e., setting its *key* and its *next* fields). We make the list circular by letting the *next* field of the last element point to the first element.

We then start scanning the list from the beginning. We output and then delete every mth element, until the list becomes empty. The output sequence is the

(n, m)-Josephus permutation. This process takes $O(m)$ time per element, for a total time of $O(mn)$. Since m is a constant, we get $O(mn) = O(n)$ time, as required.

b. We can use an order-statistic tree, straight out of Section 14.1. Why? Suppose that we are at a particular spot in the permutation, and let's say that it's the jth largest remaining person. Suppose that there are $k \le n$ people remaining. Then we will remove person j, decrement k to reflect having removed this person, and then go on to the $(j+m-1)$th largest remaining person (subtract 1 because we have just removed the jth largest). But that assumes that $j + m \le k$. If not, then we use a little modular arithmetic, as shown below.

In detail, we use an order-statistic tree T, and we call the procedures OS-INSERT, OS-DELETE, OS-RANK, and OS-SELECT:

```
JOSEPHUS(n, m)
initialize T to be empty
for j ← 1 to n
    do create a node x with key[x] = j
       OS-INSERT(T, x)
k ← n
j ← m
while k > 2
    do x ← OS-SELECT(root[T], j)
       print key[x]
       OS-DELETE(T, x)
       k ← k − 1
       j ← ((j + m − 2) mod k) + 1
print key[OS-SELECT(root[T], 1)]
```

The above procedure is easier to understand. Here's a streamlined version:

```
JOSEPHUS(n, m)
initialize T to be empty
for j ← 1 to n
    do create a node x with key[x] = j
       OS-INSERT(T, x)
j ← 1
for k ← n downto 1
    do j ← ((j + m − 2) mod k) + 1
       x ← OS-SELECT(root[T], j)
       print key[x]
       OS-DELETE(T, x)
```

Either way, it takes $O(n \lg n)$ time to build up the order-statistic tree T, and then we make $O(n)$ calls to the order-statistic-tree procedures, each of which takes $O(\lg n)$ time. Thus, the total time is $O(n \lg n)$.

Lecture Notes for Chapter 15: Dynamic Programming

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.
  - Minimization or maximization. (We'll see both.)

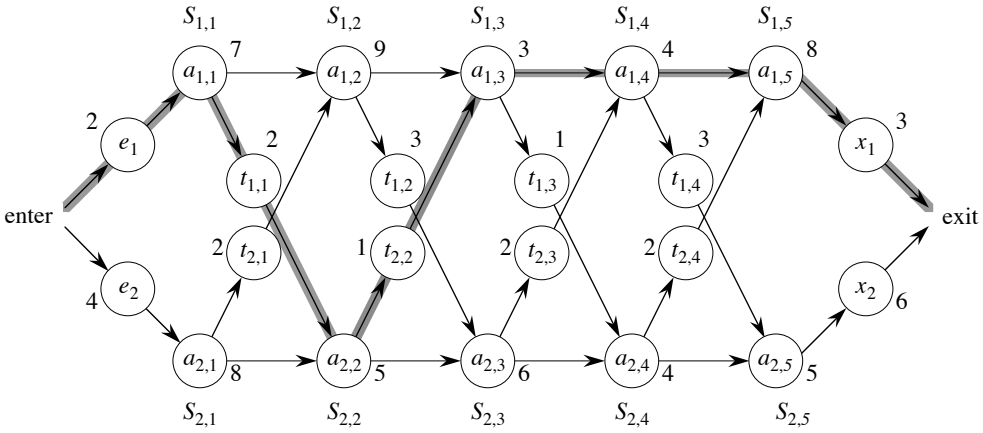
Four-step method

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Assembly-line scheduling

A simple dynamic-programming example. Actually, solvable by a graph algorithm that we'll see later in the course. But a good warm-up for dynamic programming.

[New in the second edition of the book.]



Automobile factory with two assembly lines.

- Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$.
- Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$.
- Entry times e_1 and e_2.
- Exit times x_1 and x_2.
- After going through a station, can either
  - stay on same line; no cost, or
  - transfer to other line; cost after $S_{i,j}$ is $t_{i,j}$. ($j = 1, \ldots, n-1$. No $t_{i,n}$, because the assembly line is done after $S_{i,n}$.)

Problem: Given all these costs (time = cost), what stations should be chosen from line 1 and from line 2 for fastest way through factory?

Try all possibilities?

- Each candidate is fully specified by which stations from line 1 are included. Looking for a subset of line 1 stations.
- Line 1 has n stations.
- 2^n subsets.
- Infeasible when n is large.

Structure of an optimal solution

Think about fastest way from entry through $S_{1,j}$.

- If $j = 1$, easy: just determine how long it takes to get through $S_{1,1}$.
- If $j \geq 2$, have two choices of how to get to $S_{1,j}$:
  - Through $S_{1,j-1}$, then directly to $S_{1,j}$.
  - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$.

Suppose fastest way is through $S_{1,j-1}$.

Key observation: We must have taken a fastest way from entry through $S_{1,j-1}$ in this solution. If there were a faster way through $S_{1,j-1}$, we would use it instead to come up with a faster way through $S_{1,j}$.

Now suppose a fastest way is through $S_{2,j-1}$. Again, we must have taken a fastest way through $S_{2,j-1}$. Otherwise use some faster way through $S_{2,j-1}$ to give a faster way through $S_{1,j}$

Generally: An optimal solution to a problem (fastest way through $S_{1,j}$) contains within it an optimal solution to subproblems (fastest way through $S_{1,j-1}$ or $S_{2,j-1}$).

This is ***optimal substructure***.

Use optimal substructure to construct optimal solution to problem from optimal solutions to subproblems.

Fastest way through $S_{1,j}$ is either

- fastest way through $S_{1,j-1}$ then directly through $S_{1,j}$, or
- fastest way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$.

Symmetrically:

Fastest way through $S_{2,j}$ is either

- fastest way through $S_{2,j-1}$ then directly through $S_{2,j}$, or
- fastest way through $S_{1,j-1}$, transfer from line 1 to line 2, then through $S_{2,j}$.

Therefore, to solve problems of finding a fastest way through $S_{1,j}$ and $S_{2,j}$, solve subproblems of finding a fastest way through $S_{1,j-1}$ and $S_{2,j-1}$.

Recursive solution

Let $f_i[j]$ = fastest time to get through $S_{i,j}$, $i = 1, 2$ and $j = 1, \ldots, n$.

Goal: fastest time to get all the way through = f^*.

$$\begin{aligned} f^* &= \min(f_1[n] + x_1,\ f_2[n] + x_2) \\ f_1[1] &= e_1 + a_{1,1} \\ f_2[1] &= e_2 + a_{2,1} \end{aligned}$$

For $j = 2, \ldots, n$:

$$\begin{aligned} f_1[j] &= \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) \\ f_2[j] &= \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) \end{aligned}$$

$f_i[j]$ gives the *value* of an optimal solution. What if we want to *construct* an optimal solution?

- $l_i[j]$ = line # (1 or 2) whose station $j-1$ is used in fastest way through $S_{i,j}$.
- In other words $S_{l_i[j],j-1}$ precedes $S_{i,j}$.
- Defined for $i = 1, 2$ and $j = 2, \ldots, n$.
- l^* = line # whose station n is used.

For example:

j	1	2	3	4	5
$f_1[j]$	9	18	20	24	32
$f_2[j]$	12	16	22	25	30

$f^* = 35$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

$l^* = 1$

Go through optimal way given by l values. (Shaded path in earlier figure.)

Compute an optimal solution

Could just write a recursive algorithm based on above recurrences.

- Let $r_i(j) =$ # of references made to $f_i[j]$.
- $r_1(n) = r_2(n) = 1$.
- $r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$ for $j = 1, \ldots, n-1$.

Claim
$r_i(j) = 2^{n-j}$.

Proof Induction on j, down from n.

Basis: $j = n$. $2^{n-j} = 2^0 = 1 = r_i(n)$.

Inductive step: Assume $r_i(j+1) = 2^{n-(j+1)}$.

$$\begin{aligned} \text{Then } r_i(j) &= r_i(j+1) + r_2(j+1) \\ &= 2^{n-(j+1)} + 2^{n-(j+1)} \\ &= 2^{n-(j+1)+1} \\ &= 2^{n-j} \ . \end{aligned}$$

■ (claim)

Therefore, $f_1[1]$ alone is referenced 2^{n-1} times!

So top down isn't a good way to compute $f_i[j]$.

Observation: $f_i[j]$ depends only on $f_1[j-1]$ and $f_2[j-1]$ (for $j \geq 2$).

So compute in order of *increasing* j.

FASTEST-WAY(a, t, e, x, n)
$f_1[1] \leftarrow e_1 + a_{1,1}$
$f_2[1] \leftarrow e_2 + a_{2,1}$
for $j \leftarrow 2$ **to** n
    do if $f_1[j-1] + a_{1,j} \le f_2[j-1] + t_{2,j-1} + a_{1,j}$
        then $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
            $l_1[j] \leftarrow 1$
        else $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
            $l_1[j] \leftarrow 2$
      if $f_2[j-1] + a_{2,j} \le f_1[j-1] + t_{1,j-1} + a_{2,j}$
        then $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
            $l_2[j] \leftarrow 2$
        else $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
            $l_2[j] \leftarrow 1$
if $f_1[n] + x_1 \le f_2[n] + x_2$
  then $f^* = f_1[n] + x_1$
       $l^* = 1$
  else $f^* = f_2[n] + x_2$
       $l^* = 2$

Go through example.

Constructing an optimal solution

PRINT-STATIONS(l, n)
$i \leftarrow l^*$
print "line " i ", station " n
for $j \leftarrow n$ **downto** 2
    do $i \leftarrow l_i[j]$
        print "line " i ", station " $j - 1$

Go through example.

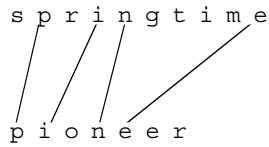
Time $= \Theta(n)$

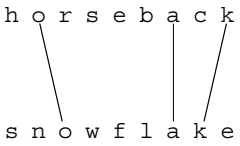
Longest common subsequence

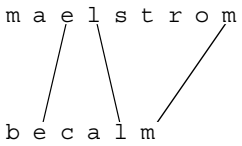
Problem: Given 2 sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

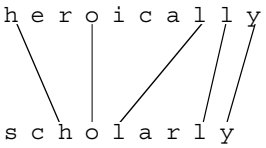
[To come up with examples of longest common subsequences, search the dictionary for all words that contain the word you are looking for as a subsequence. On a UNIX system, for example, to find all the words with `pine` *as a subsequence, use the command* `grep '.*p.*i.*n.*e.*' dict`*, where* `dict` *is your local dictionary. Then check if that word is actually a longest common subsequence. Working C code for finding a longest commmon subsequence of two strings appears at http://www.cs.dartmouth.edu/~thc/code/lcs.c]*

Examples: *[The examples are of different types of trees.]*









Brute-force algorithm:

For every subsequence of X, check whether it's a subsequence of Y.

Time: $\Theta(n2^m)$.

- 2^m subsequences of X to check.
- Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

Optimal substructure

Notation:

X_i = prefix $\langle x_1, \ldots, x_i \rangle$

Y_i = prefix $\langle y_1, \ldots, y_i \rangle$

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}.

Proof

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = \langle z_1, \ldots, z_k, x_m \rangle$. It's a common subsequence of X and Y and has length $k + 1$ $\Rightarrow Z'$ is a longer common subsequence than $Z \Rightarrow$ contradicts Z being an LCS.

   Now show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. Clearly, it's a common subsequence. Now suppose there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence W' by appending x_m to W. W' is common subsequence of X and Y, has length $\geq k + 1 \Rightarrow$ contradicts Z being an LCS.
2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$. Then W is a common subsequence of X and $Y \Rightarrow$ contradicts Z being an LCS.

3. Symmetric to 2. ■ (theorem)

Therefore, an LCS of two sequences contains as a prefix an LCS of prefixes of the sequences.

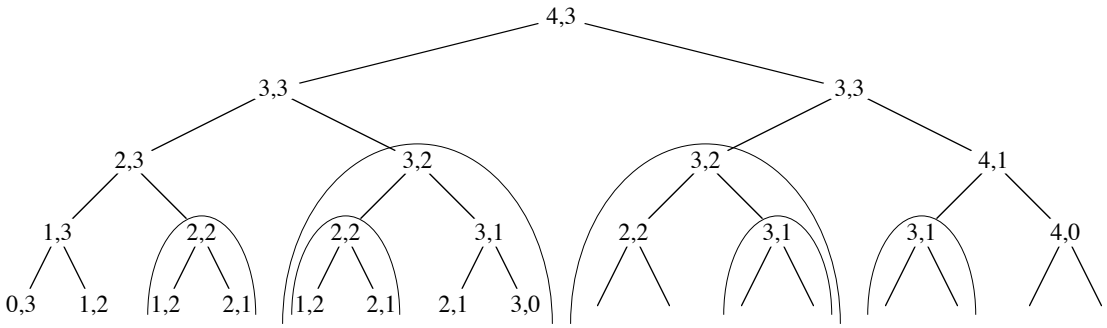
Recursive formulation

Define $c[i, j]$ = length of LCS of X_i and Y_j. We want $c[m, n]$.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

Again, we could write a recursive algorithm based on this formulation.

Try with bozo, bat.



- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

Compute length of optimal solution

LCS-LENGTH(X, Y, m, n)

```
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
           do if x_i = y_j
                 then c[i, j] ← c[i − 1, j − 1] + 1
                      b[i, j] ← "↖"
                 else if c[i − 1, j] ≥ c[i, j − 1]
                         then c[i, j] ← c[i − 1, j]
                              b[i, j] ← "↑"
                         else c[i, j] ← c[i, j − 1]
                              b[i, j] ← "←"
return c and b
```

```
PRINT-LCS(b, X, i, j)
if i = 0 or j = 0
  then return
if b[i, j] = "↖"
  then PRINT-LCS(b, X, i − 1, j − 1)
       print x_i
elseif b[i, j] = "↑"
  then PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

- Initial call is PRINT-LCS(b, X, m, n).
- $b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.
- When $b[i, j] = \nwarrow$, we have extended LCS by one character. So longest common subsequence = entries with $\nwarrow$ in them.

Demonstration: show only $c[i, j]$:

		a	**m**	**p**	**u**	**t**	**a**	**t**	**i**	**o**	**n**
	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2
n	0	1	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	1	2	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	1	2	2	3	3	4
g	0	1	1	1	1	1	2	2	3	3	4
				p			**a**		**i**		**n**

Time: $\Theta(mn)$

Optimal binary search trees

[Also new in the second edition.]

- Given sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys, sorted ($k_1 < k_2 < \cdots < k_n$).
- Want to build a binary search tree from the keys.
- For k_i, have probability p_i that a search is for k_i.
- Want BST with minimum expected search cost.

- Actual cost = # of items examined.

  For key k_i, cost = $\text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T.

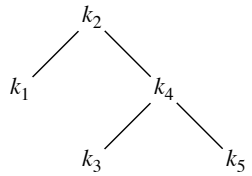
E [search cost in T]

$$\begin{aligned} &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i \\ &= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i \qquad \text{(since probabilities sum to 1)} \qquad (*) \end{aligned}$$

[Similar to optimal BST problem in the book, but simplified here: we assume that all searches are successful. Book has probabilities of searches between keys in tree.]

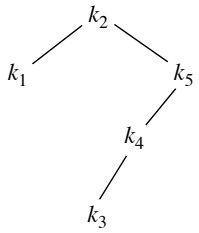
i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3

Example:



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	2	.1
4	1	.2
5	2	.6
		1.15

Therefore, E [search cost] = 2.15.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	3	.15
4	2	.4
5	1	.3
		1.10

Therefore, $\text{E}[\text{search cost}] = 2.10$, which turns out to be optimal.

Observations:

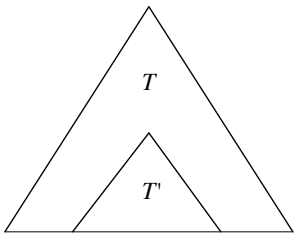
- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.

Build by exhaustive checking?

- Construct each n-node BST.
- For each, put in keys.
- Then compute expected search cost.
- But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.

Optimal substructure

Consider any subtree of a BST. It contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.

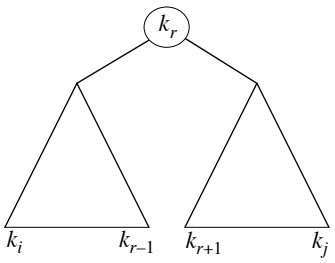


If T is an optimal BST and T contains subtree T' with keys $k_i, \ldots, k_j$, then T' must be an optimal BST for keys $k_i, \ldots, k_j$.

Proof Cut and paste. ■

Use optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblems:

- Given keys $k_i, \ldots, k_j$ (the problem).
- One of them, k_r, where $i \le r \le j$, must be the root.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.



- If
  - we examine all candidate roots k_r, for $i \le r \le j$, and
  - we determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$,

  then we're guaranteed to find an optimal BST for $k_i, \ldots, k_j$.

Recursive solution

Subproblem domain:

- Find optimal BST for $k_i, \ldots, k_j$, where $i \ge 1$, $j \le n$, $j \ge i - 1$.
- When $j = i - 1$, the tree is empty.

Define $e[i, j]$ = expected search cost of optimal BST for $k_i, \ldots, k_j$.

If $j = i - 1$, then $e[i, j] = 0$.

If $j \ge i$,

- Select a root k_r, for some $i \le r \le j$.
- Make an optimal BST with $k_i, \ldots, k_{r-1}$ as the left subtree.
- Make an optimal BST with $k_{r+1}, \ldots, k_j$ as the right subtree.
- Note: when $r = i$, left subtree is $k_i, \ldots, k_{i-1}$; when $r = j$, right subtree is $k_{j+1}, \ldots, k_j$.

When a subtree becomes a subtree of a node:

- Depth of every node in subtree goes up by 1.
- Expected search cost increases by

$$w(i, j) = \sum_{l=i}^{j} p_l \qquad \text{(refer to equation } (*)\text{)} .$$

If k_r is the root of an optimal BST for $k_i, \ldots, k_j$:

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) .$$

But $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$.

Therefore, $e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$.

This equation assumes that we already know which key is k_r.

We don't.

Try all candidates, and pick the best one:

$$e[i, j] = \begin{cases} 0 & \text{if } j = i-1 , \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j . \end{cases}$$

Could write a recursive algorithm...

Computing an optimal solution

As "usual," we'll store the values in a table:

$$e[\underbrace{1\,..\,n+1}_{\substack{\text{can store} \\ e[n+1, n]}} , \quad \underbrace{0\,..\,n}_{\substack{\text{can store} \\ e[1, 0]}} \ ]$$

- Will use only entries $e[i, j]$, where $j \ge i - 1$.
- Will also compute

  root$[i, j]$ = root of subtree with keys $k_i, \ldots, k_j$, for $1 \le i \le j \le n$.

One other table... don't recompute $w(i, j)$ from scratch every time we need it. (Would take $\Theta(j - i)$ additions.)

Instead:

- Table $w[1\,..\,n+1, 0\,..\,n]$
- $w[i, i-1] = 0$ for $1 \le i \le n$
- $w[i, j] = w[i, j-1] + p_j$ for $1 \le i \le j \le n$

Can compute all $\Theta(n^2)$ values in $O(1)$ time each.

```
OPTIMAL-BST(p, q, n)
for i ← 1 to n + 1
    do e[i, i − 1] ← 0
       w[i, i − 1] ← 0
for l ← 1 to n
    do for i ← 1 to n − l + 1
           do j ← i + l − 1
              e[i, j] ← ∞
              w[i, j] ← w[i, j − 1] + p_j
              for r ← i to j
                  do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
                     if t < e[i, j]
                       then e[i, j] ← t
                            root[i, j] ← r
return e and root
```

First **for** loop initializes e, w entries for subtrees with 0 keys.
Main **for** loop:

- Iteration for l works on subtrees with l keys.
- Idea: compute in order of subtree sizes, smaller (1 key) to larger (n keys).

For example at beginning:

i \ j	0	1	2	3	4	5
e						
1	0	.25	.65	.8	1.25	2.10
2		0	.2	.3	.75	1.35
3			0	.05	.3	.85
4				0	.2	.7
5					0	.3
6						0

(p_i marks the entry in row i, column $i-1$.)

i \ j	0	1	2	3	4	5
w						
1	0	.25	.45	.5	.7	1.0
2		0	.2	.25	.45	.75
3			0	.05	.25	.55
4				0	.2	.5
5					0	.3
6						0

i \ j	1	2	3	4	5
root					
1	1	1	1	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

Time: $O(n^3)$: for loops nested 3 deep, each loop index takes on $\leq n$ values. Can also show $\Omega(n^3)$. Therefore, $\Theta(n^3)$.

Construct an optimal solution

CONSTRUCT-OPTIMAL-BST(*root*)
$r \leftarrow root[1, n]$
print "k"$_r$ "is the root"
CONSTRUCT-OPT-SUBTREE$(1, r-1, r,$ "left", *root*)
CONSTRUCT-OPT-SUBTREE$(r+1, n, r,$ "right", *root*)

CONSTRUCT-OPT-SUBTREE$(i, j, r, dir, root)$
if $i \leq j$
  then $t \leftarrow root[i, j]$
    print "k"$_t$ "is" *dir* "child of k"$_r$
    CONSTRUCT-OPT-SUBTREE$(i, t-1, t,$ "left", *root*)
    CONSTRUCT-OPT-SUBTREE$(t+1, j, t,$ "right", *root*)

Elements of dynamic programming

Mentioned already:

- optimal substructure
- overlapping subproblems

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution. *[We find that students often have trouble understanding the relationship between optimal substructure and determining which choice is made in an optimal solution. One way that helps them understand optimal substructure is to imagine that "God" tells you what was the last choice made in an optimal solution.]*
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste:
  - Suppose that one of the subproblem solutions is not optimal.
  - *Cut* it out.
  - *Paste* in an optimal solution.
  - Get a better solution to the original problem. Contradicts optimality of problem solution.

That was optimal substructure.

Need to ensure that you consider a wide enough range of choices and subproblems that you get them all. *["God" is too busy to tell you what that last choice really was.]* Try all the choices, solve all the subproblems resulting from each choice, and pick the choice whose solution, along with subproblem solutions, is best.

How to characterize the space of subproblems?

- Keep the space as simple as possible.
- Expand it as necessary.

Examples:

Assembly-line scheduling

- Space of subproblems was fastest way from factory entry through stations $S_{1,j}$ and $S_{2,j}$.
- No need to try a more general space of subproblems.

Optimal binary search trees

- Suppose we had tried to constrain space of subproblems to subtrees with keys $k_1, k_2, \ldots, k_j$.

- An optimal BST would have root k_r, for some $1 \le r \le j$.
- Get subproblems $k_1, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$.
- Unless we could guarantee that $r = j$, so that subproblem with $k_{r+1}, \ldots, k_j$ is empty, then this subproblem is *not* of the form $k_1, k_2, \ldots, k_j$.
- Thus, needed to allow the subproblems to vary at "both ends," i.e., allow both i and j to vary.

Optimal substructure varies across problem domains:

1. *How many subproblems* are used in an optimal solution.
2. *How many choices* in determining which subproblem(s) to use.

- Assembly-line scheduling:
  - 1 subproblem
  - 2 choices (for $S_{i,j}$ use either $S_{1,j-1}$ or $S_{2,j-1}$)
- Longest common subsequence:
  - 1 subproblem
  - Either
    - 1 choice (if $x_i = y_j$, LCS of X_{i-1} and Y_{j-1}), or
    - 2 choices (if $x_i \ne y_j$, LCS of X_{i-1} and Y, and LCS of X and Y_{j-1})
- Optimal binary search tree:
  - 2 subproblems ($k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$)
  - $j - i + 1$ choices for k_r in $k_i, \ldots, k_j$. Once we determine optimal solutions to subproblems, we choose from among the $j - i + 1$ candidates for k_r.

Informally, running time depends on (# of subproblems overall) $\times$ (# of choices).

- Assembly-line scheduling: $\Theta(n)$ subproblems, 2 choices for each
  $\Rightarrow \Theta(n)$ running time.
- Longest common subsequence: $\Theta(mn)$ subproblems, ≤ 2 choices for each
  $\Rightarrow \Theta(mn)$ running time.
- Optimal binary search tree: $\Theta(n^2)$ subproblems, $O(n)$ choices for each
  $\Rightarrow O(n^3)$ running time.

Dynamic programming uses optimal substructure *bottom up*.

- *First* find optimal solutions to subproblems.
- *Then* choose which to use in optimal solution to the problem.

When we look at greedy algorithms, we'll see that they work *top down*: *first* make a choice that looks best, *then* solve the resulting subproblem.

Don't be fooled into thinking optimal substructure applies to all optimization problems. It doesn't.

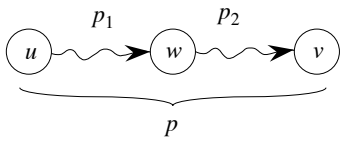
Here are two problems that look similar. In both, we're given an *unweighted, directed graph* $G = (V, E)$.

- V is a set of *vertices*.
- E is a set of *edges*.

And we ask about finding a ***path*** (sequence of connected edges) from vertex u to vertex v.

- ***Shortest path***: find path $u \rightsquigarrow v$ with fewest edges. Must be ***simple*** (no *cycles*), since removing a cycle from a path gives a path with fewer edges.
- ***Longest simple path***: find *simple* path $u \rightsquigarrow v$ with most edges. If didn't require simple, could repeatedly traverse a cycle to make an arbitrarily long path.

Shortest path has optimal substructure.



- Suppose p is shortest path $u \rightsquigarrow v$.
- Let w be any vertex on p.
- Let p_1 be the portion of p, $u \rightsquigarrow w$.
- Then p_1 is a shortest path $u \rightsquigarrow w$.

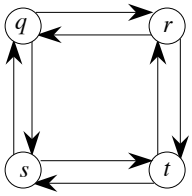
Proof Suppose there exists a shorter path p_1', $u \rightsquigarrow w$. Cut out p_1, replace it with p_1', get path $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ with fewer edges than p. ■

Therefore, can find shortest path $u \rightsquigarrow v$ by considering all intermediate vertices w, then finding shortest paths $u \rightsquigarrow w$ and $w \rightsquigarrow v$.

Same argument applies to p_2.

Does longest path have optimal substructure?

- It seems like it should.
- It does *not*.



Consider $q \to r \to t$ = longest path $q \rightsquigarrow t$. Are its subpaths longest paths?

No!

- Subpath $q \rightsquigarrow r$ is $q \to r$.
- Longest simple path $q \rightsquigarrow r$ is $q \to s \to t \to r$.
- Subpath $r \rightsquigarrow t$ is $r \to t$.
- Longest simple path $r \rightsquigarrow t$ is $r \to q \to s \to t$.

Not only isn't there optimal substructure, but we can't even assemble a legal solution from solutions to subproblems.

Combine longest simple paths:

$q \to s \to t \to r \to q \to s \to t$

Not simple!

In fact, this problem is NP-complete (so it probably has no optimal substructure to find.)

What's the big difference between shortest path and longest path?

- Shortest path has ***independent*** subproblems.
- Solution to one subproblem does not affect solution to another subproblem of the same problem.
- Longest simple path: subproblems are *not* independent.
- Consider subproblems of longest simple paths $q \leadsto r$ and $r \leadsto t$.
- Longest simple path $q \leadsto r$ uses s and t.
- Cannot use s and t to solve longest simple path $r \leadsto t$, since if we do, the path isn't simple.
- But we *have* to use t to find longest simple path $r \leadsto t$!
- Using resources (vertices) to solve one subproblem renders them unavailable to solve the other subproblem.

  [For shortest paths, if we look at a shortest path $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, no vertex other than w can appear in p_1 and p_2. Otherwise, we have a cycle.]

Independent subproblems in our examples:

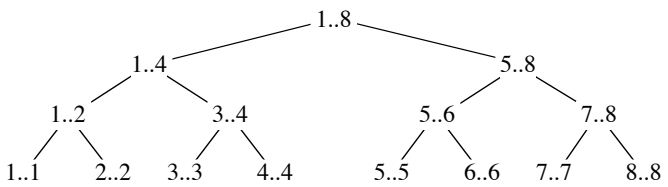
- Assembly line and longest common subsequence
  - 1 subproblem $\Rightarrow$ automatically independent.
- Optimal binary search tree
  - $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j \Rightarrow$ independent.

Overlapping subproblems

These occur when a recursive algorithm revisits the same problem over and over.

Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

Example: merge sort



Won't go through exercise of showing repeated subproblems.

Book has a good example for matrix-chain multiplication.

Alternative approach: ***memoization***

- "Store, don't recompute."
- Make a table indexed by subproblem.
- When solving a subproblem:
  - Lookup in table.
  - If answer is there, use it.
  - Else, compute answer, then store it.
- In dynamic programming, we go one step further. We determine in what order we'd want to access the table, and fill it in that way.

Solutions for Chapter 15: Dynamic Programming

Solution to Exercise 15.1-5

If $l_1[j] = 2$, then the fastest way to go through station j on line 1 is by changing lines from station $j-1$ on line 2. This means that $f_2[j-1] + t_{1,j-1} + a_{1,j} < f_1[j-1] + a_{1,j}$. Dropping $a_{1,j}$ from both sides of the equation yields $f_2[j-1] + t_{1,j-1} < f_1[j-1]$.

If $l_2[j] = 1$, then the fastest way to go through station j on line 2 is by changing lines from station $j-1$ on line 1. This means that $f_1[j-1] + t_{2,j-1} + a_{2,j} < f_2[j-1] + a_{2,j}$. Dropping $a_{2,j}$ from both sides of the equation yields $f_1[j-1] + t_{2,j-1} < f_2[j-1]$.

We can derive a contradiction by combining the two equations as follows: $f_2[j-1] + t_{1,j-1} < f_1[j-1]$ and $f_1[j-1] + t_{2,j-1} < f_2[j-1]$ yields $f_2[j-1] + t_{1,j-1} + t_{2,j-1} < f_2[j-1]$. Since all transfer costs are nonnegative, the resulting inequality cannot hold. We conclude that we cannot have the situation where $l_1[j] = 2$ and $l_2[j] = 1$.

Solution to Exercise 15.2-4

Each time the l-loop executes, the i-loop executes $n - l + 1$ times. Each time the i-loop executes, the k-loop executes $j - i = l - 1$ times, each time referencing m twice. Thus the total number of times that an entry of m is referenced while computing other entries is $\sum_{l=2}^{n}(n-l+1)(l-1)2$. Thus,

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) &= \sum_{l=2}^{n}(n-l+1)(l-1)2 \\
&= 2\sum_{l=1}^{n-1}(n-l)l \\
&= 2\sum_{l=1}^{n-1}nl - 2\sum_{l=1}^{n-1}l^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6}
\end{aligned}$$

$$
\begin{aligned}
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3} .
\end{aligned}
$$

Solution to Exercise 15.3-1

Running RECURSIVE-MATRIX-CHAIN is asymptotically more efficient than enumerating all the ways of parenthesizing the product and computing the number of multiplications for each.

Consider the treatment of subproblems by the two approaches.

- For each possible place to split the matrix chain, the enumeration approach finds all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left- and right-half subproblem results is thus the product of the number of ways to do the left half and the number of ways to do the right half.
- For each possible place to split the matrix chain, RECURSIVE-MATRIX-CHAIN finds the best way to parenthesize the left half, finds the best way to parenthesize the right half, and combines just those two results. Thus the amount of work to combine the left- and right-half subproblem results is $O(1)$.

Section 15.2 argued that the running time for enumeration is $\Omega(4^n/n^{3/2})$. We will show that the running time for RECURSIVE-MATRIX-CHAIN is $O(n3^{n-1})$.

To get an upper bound on the running time of RECURSIVE-MATRIX-CHAIN, we'll use the same approach used in Section 15.2 to get a lower bound: Derive a recurrence of the form $T(n) \le \ldots$ and solve it by substitution. For the lower-bound recurrence, the book assumed that the execution of lines 1–2 and 6–7 each take at least unit time. For the upper-bound recurrence, we'll assume those pairs of lines each take at most constant time c. Thus, we have the recurrence

$$
T(n) \le \begin{cases} c & \text{if } n = 1 , \\ c + \displaystyle\sum_{k=1}^{n-1} (T(k) + T(n-k) + c) & \text{if } n \ge 2 . \end{cases}
$$

This is just like the book's $\ge$ recurrence except that it has c instead of 1, and so we can be rewrite it as

$$
T(n) \le 2 \sum_{i=1}^{n-1} T(i) + cn .
$$

We shall prove that $T(n) = O(n3^{n-1})$ using the substitution method. (Note: Any upper bound on $T(n)$ that is $o(4^n/n^{3/2})$ will suffice. You might prefer to prove one that is easier to think up, such as $T(n) = O(3.5^n)$.) Specifically, we shall show that $T(n) \le cn3^{n-1}$ for all $n \ge 1$. The basis is easy, since $T(1) \le c = c \cdot 1 \cdot 3^{1-1}$.

Inductively, for $n \geq 2$ we have

$$\begin{aligned}
T(n) &\leq 2\sum_{i=1}^{n-1} T(i) + cn \\
&\leq 2\sum_{i=1}^{n-1} ci3^{i-1} + cn \\
&\leq c \cdot \left(2\sum_{i=1}^{n-1} i3^{i-1} + n\right) \\
&= c \cdot \left(2 \cdot \left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2}\right) + n\right) \qquad \text{(see below)} \\
&= cn3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n\right) \\
&= cn3^{n-1} + \frac{c}{2}(2n + 1 - 3^n) \\
&\leq cn3^{n-1} \text{ for all } c > 0, n \geq 1 .
\end{aligned}$$

Running RECURSIVE-MATRIX-CHAIN takes $O(n3^{n-1})$ time, and enumerating all parenthesizations takes $\Omega(4^n/n^{3/2})$ time, and so RECURSIVE-MATRIX-CHAIN is more efficient than enumeration.

Note: The above substitution uses the fact that

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

This equation can be derived from equation (A.5) by taking the derivative. Let

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n - 1}{x-1} - 1 .$$

Then

$$\sum_{i=1}^{n-1} ix^{i-1} = f'(x) = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

Solution to Exercise 15.4-4

When computing a particular row of the c table, no rows before the previous row are needed. Thus only two rows—$2 \cdot length[Y]$ entries—need to be kept in memory at a time. (Note: Each row of c actually has $length[Y] + 1$ entries, but we don't need to store the column of 0's—instead we can make the program "know" that those entries are 0.) With this idea, we need only $2 \cdot \min(m, n)$ entries if we always call LCS-LENGTH with the shorter sequence as the Y argument.

We can thus do away with the c table as follows:

- Use two arrays of length $\min(m, n)$, *previous-row* and *current-row*, to hold the appropriate rows of c.
- Initialize *previous-row* to all 0 and compute *current-row* from left to right.

- When *current-row* is filled, if there are still more rows to compute, copy *current-row* into *previous-row* and compute the new *current-row*.

Actually only a little more than one row's worth of c entries—$\min(m, n) + 1$ entries—are needed during the computation. The only entries needed in the table when it is time to compute $c[i, j]$ are $c[i, k]$ for $k \le j - 1$ (i.e., earlier entries in the current row, which will be needed to compute the next row); and $c[i - 1, k]$ for $k \ge j - 1$ (i.e., entries in the previous row that are still needed to compute the rest of the current row). This is one entry for each k from 1 to $\min(m, n)$ except that there are two entries with $k = j - 1$, hence the additional entry needed besides the one row's worth of entries.

We can thus do away with the c table as follows:

- Use an array a of length $\min(m, n) + 1$ to hold the appropriate entries of c. At the time $c[i, j]$ is to be computed, a will hold the following entries:
  - $a[k] = c[i, k]$ for $1 \le k < j - 1$ (i.e., earlier entries in the current "row"),
  - $a[k] = c[i - 1, k]$ for $k \ge j - 1$ (i.e., entries in the previous "row"),
  - $a[0] = c[i, j - 1]$ (i.e., the previous entry computed, which couldn't be put into the "right" place in a without erasing the still-needed $c[i - 1, j - 1]$).
- Initialize a to all 0 and compute the entries from left to right.
  - Note that the 3 values needed to compute $c[i, j]$ for $j > 1$ are in $a[0] = c[i, j - 1]$, $a[j - 1] = c[i - 1, j - 1]$, and $a[j] = c[i - 1, j]$.
  - When $c[i, j]$ has been computed, move $a[0]$ ($c[i, j - 1]$) to its "correct" place, $a[j - 1]$, and put $c[i, j]$ in $a[0]$.

Solution to Problem 15-1

Taking the book's hint, we sort the points by x-coordinate, left to right, in $O(n \lg n)$ time. Let the sorted points be, left to right, $\langle p_1, p_2, p_3, \ldots, p_n \rangle$. Therefore, p_1 is the leftmost point, and p_n is the rightmost.

We define as our subproblems paths of the following form, which we call bitonic paths. A ***bitonic path*** $P_{i,j}$, where $i \le j$, includes all points $p_1, p_2, \ldots, p_j$; it starts at some point p_i, goes strictly left to point p_1, and then goes strictly right to point p_j. By "going strictly left," we mean that each point in the path has a lower x-coordinate than the previous point. Looked at another way, the indices of the sorted points form a strictly decreasing sequence. Likewise, "going strictly right" means that the indices of the sorted points form a strictly increasing sequence. Moreover, $P_{i,j}$ contains all the points $p_1, p_2, p_3, \ldots, p_j$. Note that p_j is the rightmost point in $P_{i,j}$ and is on the rightgoing subpath. The leftgoing subpath may be degenerate, consisting of just p_1.

Let us denote the euclidean distance between any two points p_i and p_j by $|p_i p_j|$. And let us denote by $b[i, j]$, for $1 \le i \le j \le n$, the length of the shortest bitonic path $P_{i,j}$. Since the leftgoing subpath may be degenerate, we can easily compute all values $b[1, j]$. The only value of $b[i, i]$ that we will need is $b[n, n]$, which is

the length of the shortest bitonic tour. We have the following formulation of $b[i, j]$ for $1 \le i \le j \le n$:

$$\begin{aligned} b[1,2] &= |p_1 p_2| \ , \\ b[i,j] &= b[i,j-1] + |p_{j-1} p_j| \quad \text{for } i < j-1 \ , \\ b[j-1,j] &= \min_{1 \le k < j-1} \{b[k,j-1] + |p_k p_j|\} \ . \end{aligned}$$

Why are these formulas correct? Any bitonic path ending at p_2 has p_2 as its rightmost point, so it consists only of p_1 and p_2. Its length, therefore, is $|p_1 p_2|$.

Now consider a shortest bitonic path $P_{i,j}$. The point p_{j-1} is somewhere on this path. If it is on the rightgoing subpath, then it immediately preceeds p_j on this subpath. Otherwise, it is on the leftgoing subpath, and it must be the rightmost point on this subpath, so $i = j - 1$. In the first case, the subpath from p_i to p_{j-1} must be a shortest bitonic path $P_{i,j-1}$, for otherwise we could use a cut-and-paste argument to come up with a shorter bitonic path than $P_{i,j}$. (This is part of our optimal substructure.) The length of $P_{i,j}$, therefore, is given by $b[i, j-1] + |p_{j-1} p_j|$. In the second case, p_j has an immediate predecessor p_k, where $k < j - 1$, on the rightgoing subpath. Optimal substructure again applies: the subpath from p_k to p_{j-1} must be a shortest bitonic path $P_{k,j-1}$, for otherwise we could use cut-and-paste to come up with a shorter bitonic path than $P_{i,j}$. (We have implicitly relied on paths having the same length regardless of which direction we traverse them.) The length of $P_{i,j}$, therefore, is given by $\min_{1 \le k \le j-1} \{b[k, j-1] + |p_k p_j|\}$.

We need to compute $b[n, n]$. In an optimal bitonic tour, one of the points adjacent to p_n must be p_{n-1}, and so we have

$$b[n,n] = b[n-1,n] + |p_{n-1} p_n| \ .$$

To reconstruct the points on the shortest bitonic tour, we define $r[i, j]$ to be the immediate predecessor of p_j on the shortest bitonic path $P_{i,j}$. The pseudocode below shows how we compute $b[i, j]$ and $r[i, j]$:

```
EUCLIDEAN-TSP(p)
sort the points so that ⟨p_1, p_2, p_3, ..., p_n⟩ are in order of increasing x-coordinate
b[1,2] ← |p_1 p_2|
for j ← 3 to n
    do for i ← 1 to j − 2
           do b[i, j] ← b[i, j − 1] + |p_{j−1} p_j|
              r[i, j] ← j − 1
       b[j − 1, j] ← ∞
       for k ← 1 to j − 2
           do q ← b[k, j − 1] + |p_k p_j|
              if q < b[j − 1, j]
                 then b[j − 1, j] ← q
                      r[j − 1, j] ← k
b[n, n] ← b[n − 1, n] + |p_{n−1} p_n|
return b and r
```

We print out the tour we found by starting at p_n, then a leftgoing subpath that includes p_{n-1}, from right to left, until we hit p_1. Then we print right-to-left the remaining subpath, which does not include p_{n-1}. For the example in Figure 15.9(b)

on page 365, we wish to print the sequence $p_7, p_6, p_4, p_3, p_1, p_2, p_5$. Our code is recursive. The right-to-left subpath is printed as we go deeper into the recursion, and the left-to-right subpath is printed as we back out.

```
PRINT-TOUR(r, n)
print p_n
print p_{n-1}
k ← r[n − 1, n]
PRINT-PATH(r, k, n − 1)
print p_k

PRINT-PATH(r, i, j)
if i < j
  then k ← r[i, j]
       print p_k
       if k > 1
         then PRINT-PATH(r, i, k)
  else k ← r[j, i]
       if k > 1
         then PRINT-PATH(r, k, j)
              print p_k
```

The relative values of the parameters i and j in each call of PRINT-PATH indicate which subpath we're working on. If $i < j$, we're on the right-to-left subpath, and if $i > j$, we're on the left-to-right subpath.

The time to run EUCLIDEAN-TSP is $O(n^2)$ since the outer loop on j iterates $n-2$ times and the inner loops on i and k each run at most $n-2$ times. The sorting step at the beginning takes $O(n \lg n)$ time, which the loop times dominate. The time to run PRINT-TOUR is $O(n)$, since each point is printed just once.

Solution to Problem 15-2

Note: we will assume that no word is longer than will fit into a line, i.e., $l_i \le M$ for all i.

First, we'll make some definitions so that we can state the problem more uniformly. Special cases about the last line and worries about whether a sequence of words fits in a line will be handled in these definitions, so that we can forget about them when framing our overall strategy.

- Define $extras[i, j] = M - j + i - \sum_{k=i}^{j} l_k$ to be the number of extra spaces at the end of a line containing words i through j. Note that *extras* may be negative.
- Now define the cost of including a line containing words i through j in the sum we want to minimize:

$$lc[i, j] = \begin{cases} \infty & \text{if } extras[i, j] < 0 \text{ (i.e., words } i, \ldots, j \text{ don't fit)}, \\ 0 & \text{if } j = n \text{ and } extras[i, j] \ge 0 \text{ (last line costs 0)}, \\ (extras[i, j])^3 & \text{otherwise}. \end{cases}$$

By making the line cost infinite when the words don't fit on it, we prevent such an arrangement from being part of a minimal sum, and by making the cost 0 for the last line (if the words fit), we prevent the arrangement of the last line from influencing the sum being minimized.

We want to minimize the sum of lc over all lines of the paragraph.

Our subproblems are how to optimally arrange words $1, \ldots, j$, where $j = 1, \ldots, n$.

Consider an optimal arrangement of words $1, \ldots, j$. Suppose we know that the last line, which ends in word j, begins with word i. The preceding lines, therefore, contain words $1, \ldots, i-1$. In fact, they must contain an optimal arrangement of words $1, \ldots, i-1$. (Insert your favorite cut-and-paste argument here.)

Let $c[j]$ be the cost of an optimal arrangement of words $1, \ldots, j$. If we know that the last line contains words $i, \ldots, j$, then $c[j] = c[i-1] + lc[i, j]$. As a base case, when we're computing $c[1]$, we need $c[0]$. If we set $c[0] = 0$, then $c[1] = lc[1, 1]$, which is what we want.

But of course we have to figure out which word begins the last line for the subproblem of words $1, \ldots, j$. So we try all possibilities for word i, and we pick the one that gives the lowest cost. Here, i ranges from 1 to j. Thus, we can define $c[j]$ recursively by

$$c[j] = \begin{cases} 0 & \text{if } j = 0\,, \\ \min\limits_{1 \le i \le j} (c[i-1] + lc[i, j]) & \text{if } j > 0\,. \end{cases}$$

Note that the way we defined lc ensures that

- all choices made will fit on the line (since an arrangement with $lc = \infty$ cannot be chosen as the minimum), and
- the cost of putting words $i, \ldots, j$ on the last line will not be 0 unless this really is the last line of the paragraph ($j = n$) or words $i \ldots j$ fill the entire line.

We can compute a table of c values from left to right, since each value depends only on earlier values.

To keep track of what words go on what lines, we can keep a parallel p table that points to where each c value came from. When $c[j]$ is computed, if $c[j]$ is based on the value of $c[k-1]$, set $p[j] = k$. Then after $c[n]$ is computed, we can trace the pointers to see where to break the lines. The last line starts at word $p[n]$ and goes through word n. The previous line starts at word $p[p[n]]$ and goes through word $p[n] - 1$, etc.

In pseudocode, here's how we construct the tables:

```
PRINT-NEATLY(l, n, M)
▷ Compute extras[i, j] for 1 ≤ i ≤ j ≤ n.
for i ← 1 to n
    do extras[i, i] ← M − l_i
       for j ← i + 1 to n
           do extras[i, j] ← extras[i, j − 1] − l_j − 1
▷ Compute lc[i, j] for 1 ≤ i ≤ j ≤ n.
for i ← 1 to n
    do for j ← i to n
           do if extras[i, j] < 0
                 then lc[i, j] ← ∞
              elseif j = n and extras[i, j] ≥ 0
                 then lc[i, j] ← 0
              else lc[i, j] ← (extras[i, j])^3
▷ Compute c[j] and p[j] for 1 ≤ j ≤ n.
c[0] ← 0
for j ← 1 to n
    do c[j] ← ∞
       for i ← 1 to j
           do if c[i − 1] + lc[i, j] < c[j]
                 then c[j] ← c[i − 1] + lc[i, j]
                      p[j] ← i
return c and p
```

Quite clearly, both the time and space are $\Theta(n^2)$.

In fact, we can do a bit better: we can get both the time and space down to $\Theta(nM)$. The key observation is that at most $\lceil M/2 \rceil$ words can fit on a line. (Each word is at least one character long, and there's a space between words.) Since a line with words $i, \ldots, j$ contains $j - i + 1$ words, if $j - i + 1 > \lceil M/2 \rceil$ then we know that $lc[i, j] = \infty$. We need only compute and store $extras[i, j]$ and $lc[i, j]$ for $j - i + 1 \le \lceil M/2 \rceil$. And the inner **for** loop header in the computation of $c[j]$ and $p[j]$ can run from $\max(1, j - \lceil M/2 \rceil + 1)$ to j.

We can reduce the space even further to $\Theta(n)$. We do so by not storing the lc and *extras* tables, and instead computing the value of $lc[i, j]$ as needed in the last loop. The idea is that we could compute $lc[i, j]$ in $O(1)$ time if we knew the value of $extras[i, j]$. And if we scan for the minimum value in *descending* order of i, we can compute that as $extras[i, j] = extras[i + 1, j] - l_i - 1$. (Initially, $extras[j, j] = M - l_j$.) This improvement reduces the space to $\Theta(n)$, since now the only tables we store are c and p.

Here's how we print which words are on which line. The printed output of GIVE-LINES(p, j) is a sequence of triples (k, i, j), indicating that words $i, \ldots, j$ are printed on line k. The return value is the line number k.

```
GIVE-LINES(p, j)
i ← p[j]
if i = 1
  then k ← 1
  else k ← GIVE-LINES(p, i − 1) + 1
print (k, i, j)
return k
```

The initial call is GIVE-LINES(p, n). Since the value of j decreases in each recursive call, GIVE-LINES takes a total of $O(n)$ time.

Solution to Problem 15-3

a. Dynamic programming is the ticket. This problem is slightly similar to the longest-common-subsequence problem. In fact, we'll define the notational conveniences X_i and Y_j in the similar manner as we did for the LCS problem: $X_i = x[1\,..\,i]$ and $Y_j = y[1\,..\,j]$.

Our subproblems will be determining an optimal sequence of operations that converts X_i to Y_j, for $0 \le i \le m$ and $0 \le j \le n$. We'll call this the "$X_i \to Y_j$ problem." The original problem is the $X_m \to Y_n$ problem.

Let's suppose for the moment that we know what was the last operation used to convert X_i to Y_j. There are six possibilities. We denote by $c[i, j]$ the cost of an optimal solution to the $X_i \to Y_j$ problem.

- If the last operation was a copy, then we must have had $x[i] = y[j]$. The subproblem that remains is converting X_{i-1} to Y_{j-1}. And an optimal solution to the $X_i \to Y_j$ problem must include an optimal solution to the $X_{i-1} \to Y_{j-1}$ problem. The cut-and-paste argument applies. Thus, assuming that the last operation was a copy, we have $c[i, j] = c[i - 1, j - 1] + \text{cost(copy)}$.
- If it was a replace, then we must have had $x[i] \ne y[j]$. (Here, we assume that we cannot replace a character with itself. It is a straightforward modification if we allow replacement of a character with itself.) We have the same optimal substructure argument as for copy, and assuming that the last operation was a replace, we have $c[i, j] = c[i - 1, j - 1] + \text{cost(replace)}$.
- If it was a twiddle, then we must have had $x[i] = y[j - 1]$ and $x[i - 1] = y[j]$, along with the implicit assumption that $i, j \ge 2$. Now our subproblem is $X_{i-2} \to Y_{j-2}$ and, assuming that the last operation was a twiddle, we have $c[i, j] = c[i - 2, j - 2] + \text{cost(twiddle)}$.
- If it was a delete, then we have no restrictions on x or y. Since we can view delete as removing a character from X_i and leaving Y_j alone, our subproblem is $X_{i-1} \to Y_j$. Assuming that the last operation was a delete, we have $c[i, j] = c[i - 1, j] + \text{cost(delete)}$.
- If it was an insert, then we have no restrictions on x or y. Our subproblem is $X_i \to Y_{j-1}$. Assuming that the last operation was an insert, we have $c[i, j] = c[i, j - 1] + \text{cost(insert)}$.

- If it was a kill, then we had to have completed converting X_m to Y_n, so that the current problem must be the $X_m \to Y_n$ problem. In other words, we must have $i = m$ and $j = n$. If we think of a kill as a multiple delete, we can get any $X_i \to Y_n$, where $0 \le i < m$, as a subproblem. We pick the best one, and so assuming that the last operation was a kill, we have

  $$c[m, n] = \min_{0 \le i < m} \{c[i, n]\} + \text{cost(kill)} \ .$$

We have not handled the base cases, in which $i = 0$ or $j = 0$. These are easy. X_0 and Y_0 are the empty strings. We convert an empty string into Y_j by a sequence of j inserts, so that $c[0, j] = j \cdot \text{cost(insert)}$. Similarly, we convert X_i into Y_0 by a sequence of i deletes, so that $c[i, 0] = i \cdot \text{cost(delete)}$. When $i = j = 0$, either formula gives us $c[0, 0] = 0$, which makes sense, since there's no cost to convert the empty string to the empty string.

For $i, j > 0$, our recursive formulation for $c[i, j]$ applies the above formulas in the situations in which they hold:

$$c[i, j] = \min \begin{cases} c[i-1, j-1] + \text{cost(copy)} & \text{if } x[i] = y[j] \ , \\ c[i-1, j-1] + \text{cost(replace)} & \text{if } x[i] \ne y[j] \ , \\ c[i-2, j-2] + \text{cost(twiddle)} & \text{if } i, j \ge 2, \\ & x[i] = y[j-1], \\ & \text{and } x[i-1] = y[j] \ , \\ c[i-1, j] + \text{cost(delete)} & \text{always} \ , \\ c[i, j] = c[i, j-1] + \text{cost(insert)} & \text{always} \ , \\ \min\limits_{0 \le i < m} \{c[i, n]\} + \text{cost(kill)} & \text{if } i = m \text{ and } j = n \ . \end{cases}$$

Like we did for LCS, our pseudocode fills in the table in row-major order, i.e., row-by-row from top to bottom, and left to right within each row. Column-major order (column-by-column from left to right, and top to bottom within each column) would also work. Along with the $c[i, j]$ table, we fill in the table $op[i, j]$, holding which operation was used.

```
EDIT-DISTANCE(x, y, m, n)
for i ← 0 to m
    do c[i, 0] ← i · cost(delete)
       op[i, 0] ← DELETE
for j ← 0 to n
    do c[0, j] ← j · cost(insert)
       op[0, j] ← INSERT
for i ← 1 to m
    do for j ← 1 to n
          do c[i, j] ← ∞
             if x[i] = y[j]
               then c[i, j] ← c[i − 1, j − 1] + cost(copy)
                    op[i, j] ← COPY
             if x[i] ≠ y[j] and c[i − 1, j − 1] + cost(replace) < c[i, j]
               then c[i, j] ← c[i − 1, j − 1] + cost(replace)
                    op[i, j] ← REPLACE(by y[j])
             if i ≥ 2 and j ≥ 2 and x[i] = y[j − 1] and
                       x[i − 1] = y[j] and
                       c[i − 2, j − 2] + cost(twiddle) < c[i, j]
               then c[i, j] ← c[i − 2, j − 2] + cost(twiddle)
                    op[i, j] ← TWIDDLE
             if c[i − 1, j] + cost(delete) < c[i, j]
               then c[i, j] ← c[i − 1, j] + cost(delete)
                    op[i, j] ← DELETE
             if c[i, j − 1] + cost(insert) < c[i, j]
               then c[i, j] ← c[i, j − 1] + cost(insert)
                    op[i, j] ← INSERT(y[j])
for i ← 0 to m − 1
    do if c[i, n] + cost(kill) < c[m, n]
         then c[m, n] ← c[i, n] + cost(kill)
              op[m, n] ← KILL i
return c and op
```

The time and space are both $\Theta(mn)$. If we store a KILL operation in $op[m, n]$, we also include the index i after which we killed, to help us reconstruct the optimal sequence of operations. (We don't need to store $y[i]$ in the op table for replace or insert operations.)

To reconstruct this sequence, we use the op table returned by EDIT-DISTANCE. The procedure OP-SEQUENCE(op, i, j) reconstructs the optimal operation sequence that we found to transform X_i into Y_j. The base case is when $i = j = 0$. The first call is OP-SEQUENCE(op, m, n).

```
OP-SEQUENCE(op, i, j)
if i = 0 and j = 0
  then return
if op[i, j] = COPY or op[i, j] = REPLACE
  then i' ← i − 1
       j' ← j − 1
elseif op[i, j] = TWIDDLE
  then i' ← i − 2
       j' ← j − 2
elseif op[i, j] = DELETE
  then i' ← i − 1
       j' ← j
elseif op[i, j] = INSERT        ▷ Don't care yet what character is inserted.
  then i' ← i
       j' ← j − 1
else                  ▷ Must be KILL, and must have i = m and j = n.
       let op[i, j] = KILLk
       i' ← k
       j' ← j
OP-SEQUENCE(op, i', j')
print op[i, j]
```

This procedure determines which subproblem we used, recurses on it, and then prints its own last operation.

b. The DNA-alignment problem is just the edit-distance problem, with

$$\begin{aligned}\text{cost(copy)} &= -1\,,\\ \text{cost(replace)} &= +1\,,\\ \text{cost(delete)} &= +2\,,\\ \text{cost(insert)} &= +2\,,\end{aligned}$$

and the twiddle and kill operations are not permitted.

The score that we are trying to maximize in the DNA-alignment problem is precisely the negative of the cost we are trying to minimize in the edit-distance problem. The negative cost of copy is not an impediment, since we can only apply the copy operation when the characters are equal.

Solution to Problem 15-6

Denote each square by the pair (i, j), where i is the row number, j is the column number, and $1 \le i, j \le n$. Our goal is to find a most profitable way from any square in row 1 to any square in row n. Once we do so, we can look up all the most profitable ways to get to any square in row n and pick the best one.

A subproblem is the most profitable way to get from some square in row 1 to a particular square (i, j). We have optimal substructure as follows. Consider a subproblem for (i, j), where $i > 1$, and consider the most profitable way to (i, j).

Because of how we define legal moves, it must be through square $(i-1, j')$, where $j' = j-1$, j, or $j+1$. Then the way that we got to $(i-1, j')$ within the most profitable way to (i, j) must itself be a most profitable way to $(i-1, j')$. The usual cut-and-paste argument applies. Suppose that in our most profitable way to (i, j), which goes through $(i-1, j')$, we earn a profit of d dollars to get to $(i-1, j')$, and then earn $p((i-1, j'), (i, j))$ dollars getting from $(i-1, j')$ to (i, j); thus, we earn $d + p((i-1, j'), (i, j))$ dollars getting to (i, j). Now suppose that there's a way to $(i-1, j')$ that earns d' dollars, where $d' > d$. Then we would use that to get to $(i-1, j')$ on our way to (i, j), earning $d' + p((i-1, j'), (i, j)) > d + p((i-1, j'), (i, j))$, and thus contradicting the optimality of our way to (i, j).

We also have overlapping subproblems. We need the most profitable way to (i, j) to find the most profitable way to $(i+1, j-1)$, to $(i+1, j)$, and to $(i+1, j+1)$. So we'll need to directly refer to the most profitable way to (i, j) up to three times, and if we were to implement this algorithm recursively, we'd be solving each subproblem many times.

Let $d[i, j]$ be the profit we earn in the most profitable way to (i, j). Then we have that $d[1, j] = 0$ for all $j = 1, 2, \ldots, n$. For $i = 2, 3, \ldots, n$, we have

$$d[i, j] = \max \begin{cases} d[i-1, j-1] + p((i-1, j-1), (i, j)) & \text{if } j > 1\,, \\ d[i-1, j] + p((i-1, j), (i, j)) & \text{always}\,, \\ d[i-1, j+1] + p((i-1, j+1), (i, j)) & \text{if } j < n\,. \end{cases}$$

To keep track of how we got to (i, j) most profitably, we let $w[i, j]$ be the value of j used to achieve the maximum value of $d[i, j]$. These values are defined for $2 \le i \le n$ and $1 \le j \le n$.

Thus, we can run the following procedure:

```
CHECKERBOARD(n, p)
for j ← 1 to n
    do d[1, j] ← 0
for i ← 2 to n
    do for j ← 1 to n
           do d[i, j] ← −∞
              if j > 1
                then d[i, j] ← d[i − 1, j − 1] + p((i − 1, j − 1), (i, j))
                     w[i, j] ← j − 1
              if d[i − 1, j] + p((i − 1, j), (i, j)) > d[i, j]
                then d[i, j] ← d[i − 1, j] + p((i − 1, j), (i, j))
                     w[i, j] ← j
              if j < n and d[i − 1, j + 1] + p((i − 1, j + 1), (i, j)) > d[i, j]
                then d[i, j] ← d[i − 1, j + 1] + p((i − 1, j + 1), (i, j))
                     w[i, j] ← j + 1
return d and w
```

Once we fill in the $d[i, j]$ table, the profit earned by the most profitable way to any square along the top row is $\max_{1 \le j \le n} \{d[n, j]\}$.

To actually compute the set of moves, we use the usual recursive backtracking method. This procedure prints the squares visited, from row 1 to row n:

```
PRINT-MOVES(w, i, j)
if i > 1
  then PRINT-MOVES(w, i − 1, w[i, j])
print "(" i "," j ")"
```

Letting $t = \max_{1 \le j \le n} \{d[n, j]\}$, the initial call is PRINT-MOVES(w, n, t).

The time to run CHECKERBOARD is clearly $\Theta(n^2)$. Once we have computed the d and w tables, PRINT-MOVES runs in $\Theta(n)$ time, which we can see by observing that $i = n$ in the initial call and i decreases by 1 in each recursive call.

Lecture Notes for Chapter 16: Greedy Algorithms

Chapter 16 Introduction

Similar to dynamic programming.

Used for optimization problems.

Idea: When we have a choice to make, make the one that looks best *right now*. Make a *locally optimal choice* in hope of getting a *globally optimal solution*.

Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions.

[We do not cover Huffman codes or matroids in these notes.]

Activity selection

n ***activities*** require *exclusive* use of a common resource. For example, scheduling the use of a classroom.

Set of activities $S = \{a_1, \ldots, a_n\}$.

a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where $s_i =$ start time and $f_i =$ finish time.

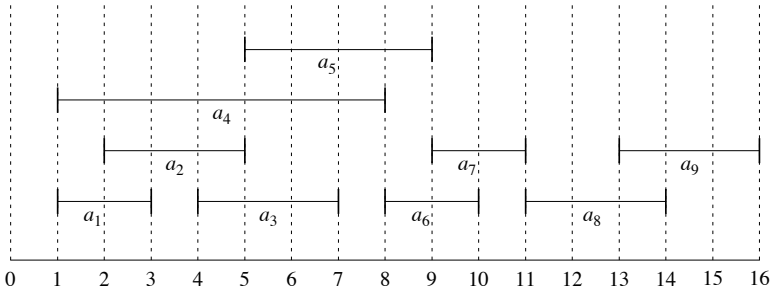
Goal: Select the largest possible set of nonoverlapping (***mutually compatible***) activities.

Note: Could have many other objectives:

- Schedule room for longest time.
- Maximize income rental fees.

Example: S sorted by finish time: *[Leave on board]*

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



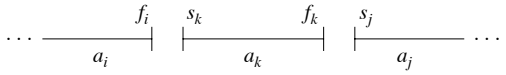
Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_2, a_5, a_7, a_9\}$.

Optimal substructure of activity selection

S_{ij} = $\{a_k \in S : f_i \le s_k < f_k \le s_j\}$ *[Leave on board]*

= activities that start after a_i finishes and finish before a_j starts .



Activities in S_{ij} are compatible with

- all activities that finish by f_i, and
- all activities that start no earlier than s_j.

To represent the entire problem, add fictitious activities:

$a_0 = [-\infty, 0)$

$a_{n+1} = [\infty, \text{“}\infty + 1\text{”})$

We don't care about $-\infty$ in a_0 or “$\infty + 1$” in a_{n+1}.

Then $S = S_{0,n+1}$.

Range for S_{ij} is $0 \le i, j \le n + 1$.

Assume that activities are sorted by monotonically increasing finish time:

$f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$.

Then $i \ge j \Rightarrow S_{ij} = \emptyset$. *[Leave on board]*

- If there exists $a_k \in S_{ij}$:

  $f_i \le s_k < f_k \le s_j < f_j \Rightarrow f_i < f_j$.
- But $i \ge j \Rightarrow f_i \ge f_j$. Contradiction.

So only need to worry about S_{ij} with $0 \le i < j \le n + 1$.

All other S_{ij} are $\emptyset$.

Suppose that a solution to S_{ij} includes a_k. Have 2 subproblems:

- S_{ik} (start after a_i finishes, finish before a_k starts)
- S_{kj} (start after a_k finishes, finish before a_j starts)

Solution to S_{ij} is (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj}).

Since a_k is in neither subproblem, and the subproblems are disjoint,

$|\text{solution to } S| = |\text{solution to } S_{ik}| + 1 + |\text{solution to } S_{kj}|$.

If an optimal solution to S_{ij} includes a_k, then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well. Use the usual cut-and-paste argument.

Let A_{ij} = optimal solution to S_{ij}.

So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ *[leave on board]* , assuming:

- S_{ij} is nonempty, and
- we know a_k.

Recursive solution to activity selection

$c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

- $i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i, j] = 0$.

If $S_{ij} \neq \emptyset$, suppose we know that a_k is in the subset. Then

$c[i, j] = c[i, k] + 1 + c[k, j]$.

But of course we don't know which k to use, and so

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \text{ ,} \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \text{ .} \end{cases}$$

[The first two printings of the book omit the requirement that $a_k \in S_{ij}$ from this max *computation. This error was corrected in the third printing.]*

Why this range of k? Because $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\} \Rightarrow a_k$ can't be a_i or a_j. Also need to ensure that a_k is actually in S_{ij}, since $i < k < j$ is not sufficient on its own to ensure this.

From here, we could continue treating this like a dynamic-programming problem.

We can simplify our lives, however.

Theorem

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min\{f_k : a_k \in S_{ij}\}$. Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

Proof

2. Suppose there is some $a_k \in S_{im}$. Then $f_i \le s_k < f_k \le s_m < f_m \Rightarrow f_k < f_m$. Then $a_k \in S_{ij}$ and it has an earlier finish time than f_m, which contradicts our choice of a_m. Therefore, there is no $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$.
1. Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij}.

   Order activites in A_{ij} in monotonically increasing order of finish time.

   Let a_k be the first activity in A_{ij}.

   If $a_k = a_m$, done (a_m is used in a maximum-size subset).

   Otherwise, construct $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m).

   Claim
   Activities in A'_{ij} are disjoint.

   Proof Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish, $f_m \le f_k$ (so a_m doesn't overlap anything else in A'_{ij}). ■ (claim)

   Since $\left|A'_{ij}\right| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is A'_{ij}. ■ (theorem)

This is great:

	before theorem	after theorem
# of subproblems in optimal solution	2	1
# of choices to consider	$j - i - 1$	1

Now we can solve *top down*:

- To solve a problem S_{ij},
  - Choose $a_m \in S_{ij}$ with earliest finish time: the ***greedy choice***.
  - Then solve S_{mj}.

What are the subproblems?

- Original problem is $S_{0,n+1}$.
- Suppose our first choice is a_{m_1}.
- Then next subproblem is $S_{m_1,n+1}$.
- Suppose next choice is a_{m_2}.
- Next subproblem is $S_{m_2,n+1}$.
- And so on.

Each subproblem is $S_{m_i,n+1}$, i.e., the last activities to finish.

And the subproblems chosen have finish times that increase.

Therefore, we can consider each activity just once, in monotonically increasing order of finish time.

Easy recursive algorithm: Assumes activites already sorted by monotonically increasing finish time. (If not, then sort in $O(n \lg n)$ time.) Return an optimal solution for $S_{i,n+1}$:

[The first two printings had a procedure that purported to return an optimal solution for S_{ij}, where $j > i$. This procedure had an error: it worked only when $j = n + 1$. It turns out that it was called only with $j = n + 1$, however. To avoid this problem altogether, the procedure was changed to the following in the third printing.]

```
REC-ACTIVITY-SELECTOR(s, f, i, n)
m ← i + 1
while m ≤ n and s_m < f_i                ▷ Find first activity in S_{i,n+1}.
    do m ← m + 1
if m ≤ n
  then return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
  else return ∅
```

Initial call: REC-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Idea: The **while** loop checks $a_{i+1}, a_{i+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

Go through example given earlier. Should get $\{a_1, a_4, a_8, a_{11}\}$.

Time: $\Theta(n)$—each activity examined exactly once.

Can make this *iterative*. It's already almost tail recursive.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
A ← {a_1}
i ← 1
for m ← 2 to n
    do if s_m ≥ f_i
         then A ← A ∪ {a_m}
              i ← m              ▷ a_i is most recent addition to A
return A
```

Go through example given earlier. Should again get $\{a_1, a_4, a_8, a_{11}\}$.

Time: $\Theta(n)$.

Greedy strategy

The choice that seems best at the moment is the one we go with.

What did we do for activity selection?

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Prove that at any stage of recursion, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
4. Show that all but one of the subproblems resulting from the greedy choice are empty.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.

At first, it looked like dynamic programming.

Typically, we streamline these steps.

Develop the substructure with an eye toward

- making the greedy choice,
- leaving just one subproblem.

For activity selection, we showed that the greedy choice implied that in S_j, only i varied, and j was fixed at $n + 1$.

We could have started out with a greedy algorithm in mind:

- Define $S_i = \{a_k \in S : f_i \leq s_k\}$.
- Then show that the greedy choice—first a_m to finish in S_i—combined with optimal solution to $S_m \Rightarrow$ optimal solution to S_i.

Typical streamlined steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.

No general way to tell if a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.

Greedy-choice property

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Dynamic programming:

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Solve *bottom-up*.

Greedy:

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*.

Typically show the greedy-choice property by what we did for activity selection:

- Look at a globally optimal solution.
- If it includes the greedy choice, done.
- Else, modify it to include the greedy choice, yielding another solution that's just as good.

Can get efficiency gains from greedy-choice property.

- Preprocess input to put it into greedy order.
- Or, if dynamic data, use a priority queue.

Optimal substructure

Just show that optimal solution to subproblem and greedy choice $\Rightarrow$ optimal solution to problem.

Greedy vs. dynamic programming

The knapsack problem is a good example of the difference.

0-1 knapsack problem:

- n items.
- Item i is worth $\$v_i$, weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- Have to either take an item or not take it—can't take part of it.

Fractional knapsack problem: Like the 0-1 knapsack problem, but can take fraction of an item.

Both have optimal substructure.

But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.

To solve the fractional problem, rank items by value/weight: v_i/w_i.

Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i.

FRACTIONAL-KNAPSACK(v, w, W)
$load \leftarrow 0$
$i \leftarrow 1$
while $load < W$ and $i \leq n$
  do if $w_i \leq W - load$
    then take all of item i
    else take $(W - load)/w_i$ of item i
    add what was taken to $load$
    $i \leftarrow i + 1$

Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

Greedy doesn't work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50$.

Greedy solution:

- Take items 1 and 2.
- value $= 160$, weight $= 30$.

Have 20 pounds of capacity left over.

Optimal solution:

- Take items 2 and 3.
- value $= 220$, weight $= 50$.

No leftover capacity.

Solutions for Chapter 16: Greedy Algorithms

Solution to Exercise 16.1-2

The proposed approach—selecting the last activity to start that is compatible with all previously selected activities—is really the greedy algorithm but starting from the end rather than the beginning.

Another way to look at it is as follows. We are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of activities, where $a_i = [s_i, f_i)$, and we propose to find an optimal solution by selecting the last activity to start that is compatible with all previously selected activities. Instead, let us create a set $S' = \{a'_1, a'_2, \ldots, a'_n\}$, where $a'_i = [f_i, s_i)$. That is, a'_i is a_i in reverse. Clearly, a subset of $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\} \subseteq S$ is mutually compatible if and only if the corresponding subset $\{a'_{i_1}, a'_{i_2}, \ldots, a'_{i_k}\} \subseteq S'$ is also mutually compatible. Thus, an optimal solution for S maps directly to an optimal solution for S' and vice versa.

The proposed approach of selecting the last activity to start that is compatible with all previously selected activities, when run on S, gives the same answer as the greedy algorithm from the text—selecting the first activity to finish that is compatible with all previously selected activities—when run on S. The solution that the proposed approach finds for S corresponds to the solution that the text's greedy algorithm finds for S', and so it is optimal.

Solution to Exercise 16.1-3

Let S be the set of n activities.

The "obvious" solution of using GREEDY-ACTIVITY-SELECTOR to find a maximum-size set S_1 of compatible activities from S for the first lecture hall, then using it again to find a maximum-size set S_2 of compatible activities from $S - S_1$ for the second hall, (and so on until all the activities are assigned), requires $\Theta(n^2)$ time in the worst case.

There is a better algorithm, however, whose asymptotic time is just the time needed to sort the activities by time—$O(n \lg n)$ time for arbitrary times, or possibly as fast as $O(n)$ if the times are small integers.

The general idea is to go through the activities in order of start time, assigning each to any hall that is available at that time. To do this, move through the set

of events consisting of activities starting and activities finishing, in order of event time. Maintain two lists of lecture halls: Halls that are busy at the current event-time t (because they have been assigned an activity i that started at $s_i \le t$ but won't finish until $f_i > t$) and halls that are free at time t. (As in the activity-selection problem in Section 16.1, we are assuming that activity time intervals are half open—i.e., that if $s_i \ge f_j$, then activities i and j are compatible.) When t is the start time of some activity, assign that activity to a free hall and move the hall from the free list to the busy list. When t is the finish time of some activity, move the activity's hall from the busy list to the free list. (The activity is certainly in some hall, because the event times are processed in order and the activity must have started before its finish time t, hence must have been assigned to a hall.)

To avoid using more halls than necessary, always pick a hall that has already had an activity assigned to it, if possible, before picking a never-used hall. (This can be done by always working at the front of the free-halls list—putting freed halls onto the front of the list and taking halls from the front of the list—so that a new hall doesn't come to the front and get chosen if there are previously-used halls.)

This guarantees that the algorithm uses as few lecture halls as possible: The algorithm will terminate with a schedule requiring $m \le n$ lecture halls. Let activity i be the first activity scheduled in lecture hall m. The reason that i was put in the mth lecture hall is that the first $m - 1$ lecture halls were busy at time s_i. So at this time there are m activities occurring simultaneously. Therefore any schedule must use at least m lecture halls, so the schedule returned by the algorithm is optimal.

Run time:

- Sort the $2n$ activity-starts/activity-ends events. (In the sorted order, an activity-ending event should precede an activity-starting event that is at the same time.) $O(n \lg n)$ time for arbitrary times, possibly $O(n)$ if the times are restricted (e.g., to small integers).
- Process the events in $O(n)$ time: Scan the $2n$ events, doing $O(1)$ work for each (moving a hall from one list to the other and possibly associating an activity with it).

Total: $O(n + \text{time to sort})$

[The idea of this algorithm is related to the rectangle-overlap algorithm in Exercise 14.3-7.]

Solution to Exercise 16.1-4

- For the approach of selecting the activity of least duration from those that are compatible with previously selected activities:

i	1	2	3
s_i	0	2	3
f_i	3	4	6
duration	3	2	3

  This approach selects just $\{a_2\}$, but the optimal solution selects $\{a_1, a_3\}$.

- For the approach of always selecting the compatible activity that overlaps the fewest other remaining activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	0	1	1	1	2	3	4	5	5	5	6
f_i	2	3	3	3	4	5	6	7	7	7	8
# of overlapping activities	3	4	4	4	4	2	4	4	4	4	3

  This approach first selects a_6, and after that choice it can select only two other activities (one of a_1, a_2, a_3, a_4 and one of a_8, a_9, a_{10}, a_{11}). An optimal solution is $\{a_1, a_5, a_7, a_{11}\}$.
- For the approach of always selecting the compatible remaining activity with the earliest start time, just add one more activity with the interval $[0, 14)$ to the example in Section 16.1. It will be the first activity selected, and no other activities are compatible with it.

Solution to Exercise 16.2-2

The solution is based on the optimal-substructure observation in the text: Let i be the highest-numbered item in an optimal solution S for W pounds and items $1, \ldots, n$. Then $S' = S - \{i\}$ must be an optimal solution for $W - w_i$ pounds and items $1, \ldots, i-1$, and the value of the solution S is v_i plus the value of the subproblem solution S'.

We can express this relationship in the following formula: Define $c[i, w]$ to be the value of the solution for items $1, \ldots, i$ and maximum weight w. Then

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0\ , \\ c[i-1, w] & \text{if } w_i > w\ , \\ \max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w \geq w_i\ . \end{cases}$$

The last case says that the value of a solution for i items either includes item i, in which case it is v_i plus a subproblem solution for $i-1$ items and the weight excluding w_i, or doesn't include item i, in which case it is a subproblem solution for $i-1$ items and the same weight. That is, if the thief picks item i, he takes v_i value, and he can choose from items $1, \ldots, i-1$ up to the weight limit $w - w_i$, and get $c[i-1, w-w_i]$ additional value. On the other hand, if he decides not to take item i, he can choose from items $1, \ldots, i-1$ up to the weight limit w, and get $c[i-1, w]$ value. The better of these two choices should be made.

The algorithm takes as inputs the maximum weight W, the number of items n, and the two sequences $v = \langle v_1, v_2, \ldots, v_n \rangle$ and $w = \langle w_1, w_2, \ldots, w_n \rangle$. It stores the $c[i, j]$ values in a table $c[0\,..\,n, 0\,..\,W]$ whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row, and so on.) At the end of the computation, $c[n, W]$ contains the maximum value the thief can take.

```
DYNAMIC-0-1-KNAPSACK(v, w, n, W)
for w ← 0 to W
    do c[0, w] ← 0
for i ← 1 to n
    do c[i, 0] ← 0
       for w ← 1 to W
           do if w_i ≤ w
                 then if v_i + c[i − 1, w − w_i] > c[i − 1, w]
                         then c[i, w] ← v_i + c[i − 1, w − w_i]
                         else c[i, w] ← c[i − 1, w]
                 else c[i, w] ← c[i − 1, w]
```

The set of items to take can be deduced from the c table by starting at $c[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i-1, w]$, then item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.

The above algorithm takes $\Theta(nW)$ time total:

- $\Theta(nW)$ to fill in the c table: $(n+1) \cdot (W+1)$ entries, each requiring $\Theta(1)$ time to compute.
- $O(n)$ time to trace the solution (since it starts in row n of the table and moves up one row at each step).

Solution to Exercise 16.2-4

The optimal strategy is the obvious greedy one. Starting will a full tank of gas, Professor Midas should go to the farthest gas station he can get to within n miles of Newark. Fill up there. Then go to the farthest gas station he can get to within n miles of where he filled up, and fill up there, and so on.

Looked at another way, at each gas station, Professor Midas should check whether he can make it to the next gas station without stopping at this one. If he can, skip this one. If he cannot, then fill up. Professor Midas doesn't need to know how much gas he has or how far the next station is to implement this approach, since at each fillup, he can determine which is the next station at which he'll need to stop.

This problem has optimal substructure. Suppose there are m possible gas stations. Consider an optimal solution with s stations and whose first stop is at the kth gas station. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining $m - k$ stations. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer than $s - 1$ stops, we could use it to come up with a solution with fewer than s stops for the full problem, contradicting our supposition of optimality.

This problem also has the greedy-choice property. Suppose there are k gas stations beyond the start that are within n miles of the start. The greedy solution chooses the kth station as its first stop. No station beyond the kth works as a first stop, since Professor Midas runs out of gas first. If a solution chooses a station $j < k$ as

its first stop, then Professor Midas could choose the kth station instead, having at least as much gas when he leaves the kth station as if he'd chosen the jth station. Therefore, he would get at least as far without filling up again if he had chosen the kth station.

If there are m gas stations on the map, Midas needs to inspect each one just once. The running time is $O(m)$.

Solution to Exercise 16.2-6

Use a linear-time median algorithm to calculate the median m of the v_i/w_i ratios. Next, partition the items into three sets: $G = \{i : v_i/w_i > m\}$, $E = \{i : v_i/w_i = m\}$, and $L = \{i : v_i/w_i < m\}$; this step takes linear time. Compute $W_G = \sum_{i \in G} w_i$ and $W_E = \sum_{i \in E} w_i$, the total weight of the items in sets G and E, respectively.

- If $W_G > W$, then do not yet take any items in set G, and instead recurse on the set of items G and knapsack capacity W.
- Otherwise ($W_G \le W$), take all items in set G, and take as much of the items in set E as will fit in the remaining capacity $W - W_G$.
- If $W_G + W_E \ge W$ (i.e., there is no capacity left after taking all the items in set G and all the items in set E that fit in the remaining capacity $W - W_G$), then we are done.
- Otherwise ($W_G + W_E < W$), then after taking all the items in sets G and E, recurse on the set of items L and knapsack capacity $W - W_G - W_E$.

To analyze this algorithm, note that each recursive call takes linear time, exclusive of the time for a recursive call that it may make. When there is a recursive call, there is just one, and it's for a problem of at most half the size. Thus, the running time is given by the recurrence $T(n) \le T(n/2) + \Theta(n)$, whose solution is $T(n) = O(n)$.

Solution to Exercise 16.2-7

Sort A and B into monotonically decreasing order.

Here's a proof that this method yields an optimal solution. Consider any indices i and j such that $i < j$, and consider the terms $a_i^{b_i}$ and $a_j^{b_j}$. We want to show that it is no worse to include these terms in the payoff than to include $a_i^{b_j}$ and $a_j^{b_i}$, i.e., that $a_i^{b_i} a_j^{b_j} \ge a_i^{b_j} a_j^{b_i}$. Since A and B are sorted into monotonically decreasing order and $i < j$, we have $a_i \ge a_j$ and $b_i \ge b_j$. Since a_i and a_j are positive and $b_i - b_j$ is nonnegative, we have $a_i^{b_i - b_j} \ge a_j^{b_i - b_j}$. Multiplying both sides by $a_i^{b_j} a_j^{b_j}$ yields $a_i^{b_i} a_j^{b_j} \ge a_i^{b_j} a_j^{b_i}$.

Since the order of multiplication doesn't matter, sorting A and B into monotonically increasing order works as well.

Solution to Exercise 16.4-2

We need to show three things to prove that $(S, \mathcal{I})$ is a matroid:

1. S is finite. That's because S is the set of of m columns of matrix T.
2. $\mathcal{I}$ is hereditary. That's because if $B \in \mathcal{I}$, then the columns in B are linearly independent. If $A \subseteq B$, then the columns of A must also be linearly independent, and so $A \in \mathcal{I}$.
3. $(S, \mathcal{I})$ satisfies the exchange property. To see why, let us suppose that $A, B \in \mathcal{I}$ and $|A| < |B|$.

   We will use the following properties of matrices:

   - The rank of a matrix is the number of columns in a maximal set of linearly independent columns (see page 731 of the text). The rank is also equal to the dimension of the column space of the matrix.
   - If the column space of matrix B is a subspace of the column space of matrix A, then $\text{rank}(B) \leq \text{rank}(A)$.

   Because the columns in A are linearly independent, if we take just these columns as a matrix A, we have that $\text{rank}(A) = |A|$. Similarly, if we take the columns of B as a matrix B, we have $\text{rank}(B) = |B|$. Since $|A| < |B|$, we have $\text{rank}(A) < \text{rank}(B)$.

   We shall show that there is some column $b \in B$ that is not a linear combination of the columns in A, and so $A \cup \{b\}$ is linearly independent. The proof proceeds by contradiction. Assume that each column in B is a linear combination of the columns of A. That means that any vector that is a linear combination of the columns of B is also a linear combination of the columns of A, and so, treating the columns of A and B as matrices, the column space of B is a subspace of the column space of A. By the second property above, we have $\text{rank}(B) \leq \text{rank}(A)$. But we have already shown that $\text{rank}(A) < \text{rank}(B)$, a contradiction. Therefore, some column in B is not a linear combination of the columns of A, and $(S, \mathcal{I})$ satisfies the exchange property.

Solution to Exercise 16.4-3

We need to show three things to prove that $(S, \mathcal{I}')$ is a matroid:

1. S is finite. We are given that.
2. $\mathcal{I}'$ is hereditary. Suppose that $B' \in \mathcal{I}'$ and $A' \subseteq B'$. Since $B' \in \mathcal{I}'$, there is some maximal set $B \in \mathcal{I}$ such that $B \subseteq S - B'$. But $A' \subseteq B'$ implies that $S - B' \subseteq S - A'$, and so $B \subseteq S - B' \subseteq S - A'$. Thus, there exists a maximal set $B \in \mathcal{I}$ such that $B \subseteq S - A'$, proving that $A' \in \mathcal{I}'$.
3. $(S, \mathcal{I}')$ satisfies the exchange property. We start with two preliminary facts about sets. The proofs of these facts are omitted.

Fact 1: $|X - Y| = |X| - |X \cap Y|$.

Fact 2: Let S be the universe of elements. If $X - Y \subseteq Z$ and $Z \subseteq S - Y$, then $|X \cap Z| = |X| - |X \cap Y|$.

To show that $(S, \mathcal{I}')$ satisfies the exchange property, let us assume that $A' \in \mathcal{I}'$, $B' \in \mathcal{I}'$, and that $|A'| < |B'|$. We need to show that there exists some $x \in B' - A'$ such that $A \cup \{x\} \in \mathcal{I}'$. Because $A' \in \mathcal{I}'$ and $B' \in \mathcal{I}'$, there are maximal sets $A \subseteq S - A'$ and $B \subseteq S - B'$ such that $A \in \mathcal{I}$ and $B \in \mathcal{I}$.

Define the set $X = B' - A' - A$, so that X consists of elements in B but not in A' or A.

If X is nonempty, then let x be any element of X. By how we defined set X, we know that $x \in B'$ and $x \notin A'$, so that $x \in B' - A'$. Since $x \notin A$, we also have that $A \subseteq S - A' - \{x\} = S - (A \cup \{x\})$, and so $A \cup \{x\} \in \mathcal{I}'$.

If X is empty, the situation is more complicated. Because $|A'| < |B'|$, we have that $B' - A' \neq \emptyset$, and so X being empty means that $B' - A' \subseteq A$.

Claim

There is an element $y \in B - A'$ such that $(A - B') \cup \{y\} \in \mathcal{I}$.

Proof First, observe that because $A - B' \subseteq A$ and $A \in \mathcal{I}$, we have that $A - B' \in \mathcal{I}$. Similarly, $B - A' \subseteq B$ and $B \in \mathcal{I}$, and so $B - A' \in \mathcal{I}$. If we show that $|A - B'| < |B - A'|$, the assumption that $(S, \mathcal{I})$ is a matroid proves the existence of y.

Because $B' - A' \subseteq A$ and $A \subseteq S - A'$, we can apply Fact 2 to conclude that $|B' \cap A| = |B'| - |B' \cap A'|$. We claim that $|B \cap A'| \leq |A' - B'|$. To see why, observe that $A' - B' = A' \cap (S - B')$ and $B \subseteq S - B'$, and so $B \cap A' \subseteq (S - B') \cap A' = A' \cap (S - B') = A' - B'$. Applying Fact 1, we see that $|A' - B'| = |A'| - |A' \cap B'| = |A'| - |B' \cap A'|$, and hence $|B \cap A'| \leq |A'| - |B' \cap A'|$.

Now, we have

$$\begin{array}{rcll}
|A'| & < & |B'| & \text{(by assumption)} \\
|A'| - |B' \cap A'| & < & |B'| - |B' \cap A'| & \text{(subtracting same quantity)} \\
|B \cap A'| & < & |B'| - |B' \cap A'| & (|B \cap A'| \leq |A'| - |B' \cap A'|) \\
|B \cap A'| & < & |B' \cap A| & (|B' \cap A| = |B'| - |B' \cap A'|) \\
|B| - |B \cap A'| & > & |A| - |B' \cap A| & (|A| = |B|) \\
|B - A'| & > & |A - B'| & \text{(Fact 1)}
\end{array}$$

■ (claim)

Now we know there is an element $y \in B - A'$ such that $(A - B') \cup \{y\} \in \mathcal{I}$. Moreover, we claim that $y \notin A$. To see why, we know that by the exchange property, $y \notin A - B'$. In order for y to be in A, it would have to be in $A \cap B'$. But $y \in B$, which means that $y \notin B'$, and hence $y \notin A \cap B'$. Therefore $y \notin A$.

We keep applying the exchange property, adding elements in $B - A'$ to $A - B'$, maintaining that the set we get is in $\mathcal{I}$. Continue adding these elements until we get a set, say C, such that $|C| = |A|$. Once $|C| = |A|$, there is some element $x \in A$ that we have not added into C. We know this because the element y that we first added into C was not in A, and so some element of A must be left over.

The set C is maximal, because it has the same cardinality as A, which is maximal, and $C \in \mathcal{I}$. Since C started with all elements in $A - B'$ and we added only elements in $B - A'$, at no time did C receive an element in A'. Because we also never added x to C, we have that $C \subseteq S - A' - \{x\} = S - (A' \cup \{x\})$, which proves that $A' \cup \{x\} \in \mathcal{I}'$, as we needed to show.

Solution to Problem 16-1

Before we go into the various parts of this problem, let us first prove once and for all that the coin-changing problem has optimal substructure.

Suppose we have an optimal solution for a problem of making change for n cents, and we know that this optimal solution uses a coin whose value is c cents; let this optimal solution use k coins. We claim that this optimal solution for the problem of n cents must contain within it an optimal solution for the problem of $n - c$ cents. We use the usual cut-and-paste argument. Clearly, there are $k - 1$ coins in the solution to the $n - c$ cents problem used within our optimal solution to the n cents problem. If we had a solution to the $n - c$ cents problem that used fewer than $k - 1$ coins, then we could use this solution to produce a solution to the n cents problem that uses fewer than k coins, which contradicts the optimality of our solution.

a. A greedy algorithm to make change using quarters, dimes, nickels, and pennies works as follows:

- Give $q = \lfloor n/25 \rfloor$ quarters. That leaves $n_q = n \bmod 25$ cents to make change.
- Then give $d = \lfloor n_q/10 \rfloor$ dimes. That leaves $n_d = n_q \bmod 10$ cents to make change.
- Then give $k = \lfloor n_d/5 \rfloor$ nickels. That leaves $n_k = n_d \bmod 5$ cents to make change.
- Finally, give $p = n_k$ pennies.

An equivalent formulation is the following. The problem we wish to solve is making change for n cents. If $n = 0$, the optimal solution is to give no coins. If $n > 0$, determine the largest coin whose value is less than or equal to n. Let this coin have value c. Give one such coin, and then recursively solve the subproblem of making change for $n - c$ cents.

To prove that this algorithm yields an optimal solution, we first need to show that the greedy-choice property holds, that is, that some optimal solution to making change for n cents includes one coin of value c, where c is the largest coin value such that $c \leq n$. Consider some optimal solution. If this optimal solution includes a coin of value c, then we are done. Otherwise, this optimal solution does not include a coin of value c. We have four cases to consider:

- If $1 \leq n < 5$, then $c = 1$. A solution may consist only of pennies, and so it must contain the greedy choice.
- If $5 \leq n < 10$, then $c = 5$. By supposition, this optimal solution does not contain a nickel, and so it consists of only pennies. Replace five pennies by one nickel to give a solution with four fewer coins.

- If $10 \le n < 25$, then $c = 10$. By supposition, this optimal solution does not contain a dime, and so it contains only nickels and pennies. Some subset of the nickels and pennies in this solution adds up to 10 cents, and so we can replace these nickels and pennies by a dime to give a solution with (between 1 and 9) fewer coins.
- If $25 \le n$, then $c = 25$. By supposition, this optimal solution does not contain a quarter, and so it contains only dimes, nickels, and pennies. If it contains three dimes, we can replace these three dimes by a quarter and a nickel, giving a solution with one fewer coin. If it contains at most two dimes, then some subset of the dimes, nickels, and pennies adds up to 25 cents, and so we can replace these coins by one quarter to give a solution with fewer coins.

Thus, we have shown that there is always an optimal solution that includes the greedy choice, and that we can combine the greedy choice with an optimal solution to the remaining subproblem to produce an optimal solution to our original problem. Therefore, the greedy algorithm produces an optimal solution.

For the algorithm that chooses one coin at a time and then recurses on subproblems, the running time is $\Theta(k)$, where k is the number of coins used in an optimal solution. Since $k \le n$, the running time is $O(n)$. For our first description of the algorithm, we perform a constant number of calculations (since there are only 4 coin types), and the running time is $O(1)$.

b. When the coin denominations are $c^0, c^1, \ldots, c^k$, the greedy algorithm to make change for n cents works by finding the denomination c^j such that $j = \max\{0 \le i \le k : c^i \le n\}$, giving one coin of denomination c^j, and recursing on the subproblem of making change for $n - c^j$ cents. (An equivalent, but more efficient, algorithm is to give $\lfloor n/c^k \rfloor$ coins of denomination c^k and $\lfloor (n \bmod c^{i+1})/c^i \rfloor$ coins of denomination c^i for $i = 0, 1, \ldots, k-1$.)

To show that the greedy algorithm produces an optimal solution, we start by proving the following lemma:

Lemma

For $i = 0, 1, \ldots, k$, let a_i be the number of coins of denomination c^i used in an optimal solution to the problem of making change for n cents. Then for $i = 0, 1, \ldots, k-1$, we have $a_i < c$.

Proof If $a_i \ge c$ for some $0 \le i < k$, then we can improve the solution by using one more coin of denomination c^{i+1} and c fewer coins of denomination c^i. The amount for which we make change remains the same, but we use $c - 1 > 0$ fewer coins. ■ (lemma)

To show that the greedy solution is optimal, we show that any non-greedy solution is not optimal. As above, let $j = \max\{0 \le i \le k : c^i \le n\}$, so that the greedy solution uses at least one coin of denomination c^j. Consider a non-greedy solution, which must use no coins of denomination c^j or higher. Let the non-greedy solution use a_i coins of denomination c^i, for $i = 0, 1, \ldots, j-1$; thus we have $\sum_{i=0}^{j-1} a_i c^i = n$. Since $n \ge c^j$, we have that $\sum_{i=0}^{j-1} a_i c^i \ge c^j$.

Now suppose that the non-greedy solution is optimal. By the above lemma, $a_i \le c-1$ for $i = 0, 1, \ldots, j-1$. Thus,

$$\begin{aligned}
\sum_{i=0}^{j-1} a_i c^i &\le \sum_{i=0}^{j-1} (c-1)c^i \\
&= (c-1)\sum_{i=0}^{j-1} c^i \\
&= (c-1)\,\frac{c^j - 1}{c-1} \\
&= c^j - 1 \\
&< c^j \,,
\end{aligned}$$

which contradicts our earlier assertion that $\sum_{i=0}^{j-1} a_i c^i \ge c^j$. We conclude that the non-greedy solution is not optimal.

Since any algorithm that does not produce the greedy solution fails to be optimal, only the greedy algorithm produces the optimal solution.

The problem did not ask for the running time, but for the more efficient greedy-algorithm formulation, it is easy to see that the running time is $O(k)$, since we have to perform at most k each of the division, floor, and mod operations.

c. With actual U.S. coins, we can use coins of denomination 1, 10, and 25. When $n = 30$ cents, the greedy solution gives one quarter and five pennies, for a total of six coins. The non-greedy solution of three dimes is better.

The smallest integer numbers we can use are 1, 3, and 4. When $n = 6$ cents, the greedy solution gives one 4-cent coin and two 1-cent coins, for a total of three coins. The non-greedy solution of two 3-cent coins is better.

d. Since we have optimal substructure, dynamic programming might apply. And indeed it does.

Let us define $c[j]$ to be the minimum number of coins we need to make change for j cents. Let the coin denominations be $d_1, d_2, \ldots, d_k$. Since one of the coins is a penny, there is a way to make change for any amount $j \ge 1$.

Because of the optimal substructure, if we knew that an optimal solution for the problem of making change for j cents used a coin of denomination d_i, we would have $c[j] = 1 + c[j - d_i]$. As base cases, we have that $c[j] = 0$ for all $j \le 0$.

To develop a recursive formulation, we have to check all denominations, giving

$$c[j] = \begin{cases} 0 & \text{if } j \le 0 \,, \\ 1 + \min\limits_{1 \le i \le k} \{c[j - d_i]\} & \text{if } j > 1 \,. \end{cases}$$

We can compute the $c[j]$ values in order of increasing j by using a table. The following procedure does so, producing a table $c[1..n]$. It avoids even examining $c[j]$ for $j \le 0$ by ensuring that $j \ge d_i$ before looking up $c[j - d_i]$. The procedure also produces a table $denom[1..n]$, where $denom[j]$ is the denomination of a coin used in an optimal solution to the problem of making change for j cents.

```
COMPUTE-CHANGE(n, d, k)
for j ← 1 to n
    do c[j] ← ∞
       for i ← 1 to k
           do if j ≥ d_i and 1 + c[j − d_i] < c[j]
                 then c[j] ← 1 + c[j − d_i]
                      denom[j] ← d_i
return c and denom
```

This procedure obviously runs in $O(nk)$ time.

We use the following procedure to output the coins used in the optimal solution computed by COMPUTE-CHANGE:

```
GIVE-CHANGE(j, denom)
if j > 0
  then give one coin of denomination denom[j]
       GIVE-CHANGE(j − denom[j], denom)
```

The initial call is GIVE-CHANGE$(n, denom)$. Since the value of the first parameter decreases in each recursive call, this procedure runs in $O(n)$ time.

Lecture Notes for Chapter 17: Amortized Analysis

Chapter 17 overview

Amortized analysis

- Analyze a *sequence* of operations on a data structure.
- ***Goal:*** Show that although some individual operations may be expensive, *on average* the cost per operation is small.

Average in this context does not mean that we're averaging over a distribution of inputs.

- No probability is involved.
- We're talking about *average cost in the worst case*.

Organization

We'll look at 3 methods:

- aggregate analysis
- accounting method
- potential method

Using 3 examples:

- stack with multipop operation
- binary counter
- dynamic tables (later on)

Aggregate analysis

Stack operations

- PUSH(S, x): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.
- POP(S): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.

- MULTIPOP(S, k)
  while S is not empty and $k > 0$
  do POP(S)
  $k \leftarrow k - 1$

Running time of MULTIPOP:

- Linear in # of POP operations.
- Let each PUSH/POP cost 1.
- # of iterations of **while** loop is $\min(s, k)$, where $s =$ # of objects on stack.
- Therefore, total cost $= \min(s, k)$.

Sequence of n PUSH, POP, MULTIPOP operations:

- Worst-case cost of MULTIPOP is $O(n)$.
- Have n operations.
- Therefore, worst-case cost of sequence is $O(n^2)$.

Observation

- Each object can be popped only once per time that it's pushed.
- Have $\leq n$ PUSHes $\Rightarrow \leq n$ POPs, including those in MULTIPOP.
- Therefore, total cost $= O(n)$.
- Average over the n operations $\Rightarrow O(1)$ per operation on average.

Again, notice no probability.

- Showed *worst-case* $O(n)$ cost for sequence.
- Therefore, $O(1)$ per operation on average.

This technique is called ***aggregate analysis***.

Binary counter

- k-bit binary counter $A[0\,..\,k-1]$ of bits, where $A[0]$ is the least significant bit and $A[k-1]$ is the most significant bit.
- Counts upward from 0.
- Value of counter is $\sum_{i=0}^{k-1} A[i] \cdot 2^i$.
- Initially, counter value is 0, so $A[0\,..\,k-1] = 0$.
- To increment, add 1 $\pmod{2^k}$:

```
INCREMENT(A, k)
i ← 0
while i < k and A[i] = 1
    do A[i] ← 0
       i ← i + 1
if i < k
  then A[i] ← 1
```

Example: $k = 3$

[Underlined bits flip. Show costs later.]

counter value	A 2 1 0	cost
0	0 0 $\underline{0}$	0
1	0 $\underline{0\ 1}$	1
2	0 1 $\underline{0}$	3
3	$\underline{0\ 1\ 1}$	4
4	1 0 $\underline{0}$	7
5	1 $\underline{0\ 1}$	8
6	1 1 $\underline{0}$	10
7	$\underline{1\ 1\ 1}$	11
0	0 0 $\underline{0}$	14
$\vdots$	$\vdots$	15

Cost of INCREMENT $= \Theta($# of bits flipped$)$.

Analysis: Each call could flip k bits, so n INCREMENTs takes $O(nk)$ time.

Observation

Not every bit flips every time.

[Show costs from above.]

bit	flips how often	times in n INCREMENTs
0	every time	n
1	1/2 the time	$\lfloor n/2 \rfloor$
2	1/4 the time	$\lfloor n/4 \rfloor$
	$\vdots$	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	$\vdots$	
$i \geq k$	never	0

$$\begin{aligned}
\text{Therefore, total \# of flips} &= \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor \\
&< n \sum_{i=0}^{\infty} 1/2^i \\
&= n \left(\frac{1}{1 - 1/2} \right) \\
&= 2n \ .
\end{aligned}$$

Therefore, n INCREMENTs costs $O(n)$.

Average cost per operation $= O(1)$.

Accounting method

Assign different charges to different operations.

- Some are charged more than actual cost.
- Some are charged less.

Amortized cost = amount we charge.

When amortized cost > actual cost, store the difference *on specific objects* in the data structure as ***credit***.

Use credit later to pay for operations whose actual cost > amortized cost.

Differs from aggregate analysis:

- In the accounting method, different operations can have different costs.
- In aggregate analysis, all operations have same cost.

Need credit to never go negative.

- Otherwise, have a sequence of operations for which the amortized cost is not an upper bound on actual cost.
- Amortized cost would tell us *nothing*.

Let c_i = actual cost of ith operation ,
$\widehat{c_i}$ = amortized cost of ith operation .

Then require $\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

Total credit stored $= \sum_{i=1}^{n} \widehat{c_i} - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0$.

Stack

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	$\min(k, s)$	0

Intuition: When pushing an object, pay \$2.

- \$1 pays for the PUSH.
- \$1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has \$1, which is credit, the credit can never go negative.
- Therefore, total amortized cost, $= O(n)$, is an upper bound on total actual cost.

Binary counter

Charge \$2 to set a bit to 1.

- \$1 pays for setting a bit to 1.
- \$1 is prepayment for flipping it back to 0.
- Have \$1 of credit for every 1 in the counter.
- Therefore, credit ≥ 0.

Amortized cost of INCREMENT:

- Cost of resetting bits to 0 is paid by credit.
- At most 1 bit is set to 1.
- Therefore, amortized cost $\leq$ \$2.
- For n operations, amortized cost $= O(n)$.

Potential method

Like the accounting method, but think of the credit as *potential* stored with the entire data structure.

- Accounting method stores credit with specific objects.
- Potential method stores potential in the data structure as a whole.
- Can release potential to pay for future operations.
- Most flexible of the amortized analysis methods.

Let D_i = data structure after ith operation ,
D_0 = initial data structure ,
c_i = actual cost of ith operation ,
$\widehat{c}_i$ = amortized cost of ith operation .

Potential function $\Phi : D_i \to \mathbf{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i.

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} .\end{aligned}$$

$$\begin{aligned}\text{Total amortized cost} &= \sum_{i=1}^{n} \widehat{c}_i \\ &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &\quad \text{(telescoping sum: every term other than } D_0 \text{ and } D_n \\ &\qquad \text{is added once and subtracted once)} \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .\end{aligned}$$

If we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then the amortized cost is always an upper bound on actual cost.

In practice: $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ for all i.

Stack

Φ = # of objects in stack
(= # of \$1 bills in accounting method)

D_0 = empty stack $\Rightarrow \Phi(D_0) = 0$.

Since # of objects in stack is always ≥ 0, $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all i.

operation	actual cost	$\Delta\Phi$	amortized cost
PUSH	1	$(s+1) - s = 1$ where s = # of objects initially	$1 + 1 = 2$
POP	1	$(s-1) - s = -1$	$1 - 1 = 0$
MULTIPOP	$k' = \min(k, s)$	$(s - k') - s = -k'$	$k' - k' = 0$

Therefore, amortized cost of a sequence of n operations $= O(n)$.

Binary counter

$\Phi = b_i$ = # of 1's after ith INCREMENT

Suppose ith operation resets t_i bits to 0.

$c_i \leq t_i + 1$ (resets t_i bits, sets ≤ 1 bit to 1)

- If $b_i = 0$, the ith operation reset all k bits and didn't set one, so $b_{i-1} = t_i = k \Rightarrow b_i = b_{i-1} - t_i$.
- If $b_i > 0$, the ith operation reset t_i bits, set one, so $b_i = b_{i-1} - t_i + 1$.
- Either way, $b_i \leq b_{i-1} - t_i + 1$.
- Therefore,

$$\begin{aligned} \Delta\Phi(D_i) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i \ . \end{aligned}$$

$$\begin{aligned} \widehat{c_i} &= c_i + \Delta\Phi(D_i) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 \ . \end{aligned}$$

If counter starts at 0, $\Phi(D_0) = 0$.

Therefore, amortized cost of n operations $= O(n)$.

Dynamic tables

A nice use of amortized analysis.

Scenario

- Have a table—maybe a hash table.
- Don't know in advance how many objects will be stored in it.
- When it fills, must reallocate with a larger size, copying all objects into the new, larger table.
- When it gets sufficiently small, *might* want to reallocate with a smaller size.

Details of table organization not important.

Goals

1. $O(1)$ amortized time per operation.
2. Unused space always $\leq$ constant fraction of allocated space.

Load factor $\alpha = num/size$, where num = # items stored, $size$ = allocated size.

If $size = 0$, then $num = 0$. Call $\alpha = 1$.

Never allow $\alpha > 1$.

Keep $\alpha >$ a constant fraction $\Rightarrow$ goal (2).

Table expansion

Consider only insertion.

- When the table becomes full, double its size and reinsert all existing items.
- Guarantees that $\alpha \geq 1/2$.
- Each time we actually insert an item into the table, it's an ***elementary insertion***.

```
TABLE-INSERT(T, x)
if size[T] = 0
  then allocate table[T] with 1 slot
       size[T] ← 1
if num[T] = size[T]                ▷ expand?
  then allocate new-table with 2 · size[T] slots
       insert all items in table[T] into new-table      ▷ num[T] elem insertions
       free table[T]
       table[T] ← new-table
       size[T] ← 2 · size[T]
insert x into table[T]                                  ▷ 1 elem insertion
num[T] ← num[T] + 1
```

Initially, $num[T] = size[T] = 0$.

Running time: Charge 1 per elementary insertion. Count only elementary insertions, since all other costs together are constant per call.

c_i = actual cost of ith operation

- If not full, $c_i = 1$.
- If full, have $i - 1$ items in the table at the start of the ith operation. Have to copy all $i - 1$ existing items, then insert ith item $\Rightarrow c_i = i$.

n operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for n operations.

Of course, we don't always expand:

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

$$\begin{aligned} \text{Total cost} &= \sum_{i=1}^{n} c_i \\ &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} \\ &< n + 2n \\ &= 3n \end{aligned}$$

Therefore, **aggregate analysis** says amortized cost per operation = 3.

Accounting method

Charge \$3 per insertion of x.

- \$1 pays for x's insertion.
- \$1 pays for x to be moved in the future.
- \$1 pays for some other item to be moved.

Suppose we've just expanded, $size = m$ before next expansion, $size = 2m$ after next expansion.

- Assume that the expansion used up all the credit, so that there's no credit stored after the expansion.
- Will expand again after another m insertions.
- Each insertion will put \$1 on one of the m items that were in the table just after expansion and will put \$1 on the item inserted.
- Have $\$2m$ of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion, with no credit left over!

Potential method

$\Phi(T) = 2 \cdot num[T] - size[T]$

- Initially, $num = size = 0 \Rightarrow \Phi = 0$.
- Just after expansion, $size = 2 \cdot num \Rightarrow \Phi = 0$.
- Just before expansion, $size = num \Rightarrow \Phi = num \Rightarrow$ have enough potential to pay for moving all items.
- Need $\Phi \geq 0$, always.

  Always have

$$\begin{array}{lllll} size & \geq & num & \geq & \frac{1}{2} \cdot size \quad \Rightarrow \\ & & 2 \cdot num & \geq & size \quad \Rightarrow \\ & & \Phi & \geq & 0 \, . \end{array}$$

Amortized cost of ith operation:

$$\begin{aligned} num_i &= num \text{ after } i\text{th operation} , \\ size_i &= size \text{ after } i\text{th operation} , \\ \Phi_i &= \Phi \text{ after } i\text{th operation} . \end{aligned}$$

- If no expansion:

$$\begin{aligned} size_i &= size_{i-1} , \\ num_i &= num_{i-1} + 1 , \\ c_i &= 1 . \end{aligned}$$

  Then we have

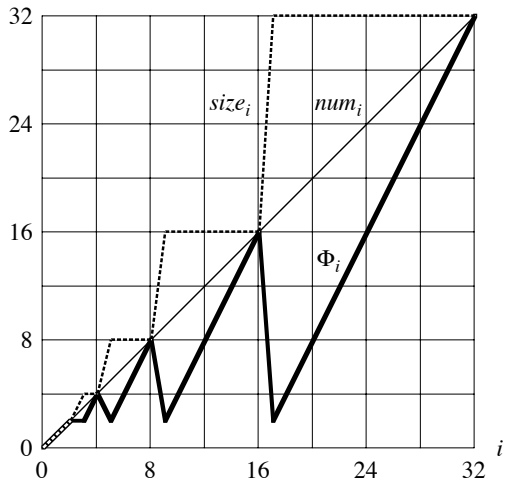
$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 1 + 2 \\ &= 3 . \end{aligned}$$

- If expansion:

$$\begin{aligned} size_i &= 2 \cdot size_{i-1} , \\ size_{i-1} &= num_{i-1} = num_i - 1 , \\ c_i &= num_{i-1} + 1 = num_i . \end{aligned}$$

  Then we have

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i + \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3 . \end{aligned}$$



Expansion and contraction

When α drops too low, contract the table.

- Allocate a new, smaller one.
- Copy all items.

Still want

- α bounded from below by a constant,
- amortized cost per operation $= O(1)$.

Measure cost in terms of elementary insertions and deletions.

"Obvious strategy":

- Double size when inserting into a full table (when $\alpha = 1$, so that after insertion α would become > 1).
- Halve size when deletion would make table less than half full (when $\alpha = 1/2$, so that after deletion α would become $< 1/2$).
- Then always have $1/2 \le \alpha \le 1$.
- Suppose we fill table.
  Then insert $\Rightarrow$ double
  2 deletes $\Rightarrow$ halve
  2 inserts $\Rightarrow$ double
  2 deletes $\Rightarrow$ halve
  . . .

Not performing enough operations after expansion or contraction to pay for the next one.

Simple solution:

- Double as before: when inserting with $\alpha = 1 \Rightarrow$ after doubling, $\alpha = 1/2$.
- Halve size when deleting with $\alpha = 1/4 \Rightarrow$ after halving, $\alpha = 1/2$.
- Thus, immediately after either expansion or contraction, have $\alpha = 1/2$.
- Always have $1/4 \le \alpha \le 1$.

Intuition:

- Want to make sure that we perform enough operations between consecutive expansions/contractions to pay for the change in table size.
- Need to delete half the items before contraction.
- Need to double number of items before expansion.
- Either way, number of operations between expansions/contractions is at least a constant fraction of number of items copied.

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha \geq 1/2 \ , \\ size[T]/2 - num[T] & \text{if } \alpha < 1/2 \ . \end{cases}$$

T empty $\Rightarrow \Phi = 0$.

$\alpha \geq 1/2 \Rightarrow num \geq \frac{1}{2} \cdot size \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi \geq 0$.

$\alpha < 1/2 \Rightarrow num < \frac{1}{2} \cdot size \Rightarrow \Phi \geq 0$.

Intuition: Φ measures how far from $\alpha = 1/2$ we are.

- $\alpha = 1/2 \Rightarrow \Phi = 2 \cdot num - 2 \cdot num = 0$.
- $\alpha = 1 \Rightarrow \Phi = 2 \cdot num - num = num$.
- $\alpha = 1/4 \Rightarrow \Phi = size/2 - num = 4 \cdot num/2 - num = num$.
- Therefore, when we double or halve, have enough potential to pay for moving all *num* items.
- Potential increases linearly between $\alpha = 1/2$ and $\alpha = 1$, and it also increases linearly between $\alpha = 1/2$ and $\alpha = 1/4$.
- Since α has different distances to go to get to 1 or $1/4$, starting from $1/2$, rate of increase of Φ differs.
  - For α to go from $1/2$ to 1, *num* increases from *size*$/2$ to *size*, for a total increase of *size*$/2$. Φ increases from 0 to *size*. Thus, Φ needs to increase by 2 for each item inserted. That's why there's a coefficient of 2 on the $num[T]$ term in the formula for Φ when $\alpha \geq 1/2$.
  - For α to go from $1/2$ to $1/4$, *num* decreases from *size*$/2$ to *size*$/4$, for a total decrease of *size*$/4$. Φ increases from 0 to *size*$/4$. Thus, Φ needs to increase by 1 for each item deleted. That's why there's a coefficient of -1 on the $num[T]$ term in the formula for Φ when $\alpha < 1/2$.

Amortized costs: more cases

- insert, delete
- $\alpha \geq 1/2$, $\alpha < 1/2$ (use α_i, since α can vary a lot)
- *size* does/doesn't change

Insert:

- $\alpha_{i-1} \geq 1/2$, same analysis as before $\Rightarrow \widehat{c_i} = 3$.
- $\alpha_{i-1} < 1/2 \Rightarrow$ *no expansion* (only occurs when $\alpha_{i-1} = 1$).

- If $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2$:

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi_i + \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0 .\end{aligned}$$

- If $\alpha_{i-1} < 1/2$ and $\alpha_i \geq 1/2$:

$$\begin{aligned}\widehat{c}_i &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\ &= 3 \cdot \alpha_{i-1}\, size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\ &< \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\ &= 3 .\end{aligned}$$

  Therefore, amortized cost of insert is < 3.

Delete:

- If $\alpha_{i-1} < 1/2$, then $\alpha_i < 1/2$.
  - If no contraction:

$$\begin{aligned}\widehat{c}_i &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2 .\end{aligned}$$

  - If contraction:

$$\begin{aligned}\widehat{c}_i &= \underbrace{(num_i + 1)}_{\text{move + delete}} + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &\quad [size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1] \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1 .\end{aligned}$$

- If $\alpha_{i-1} \geq 1/2$, then no contraction.
  - If $\alpha_i \geq 1/2$:

$$\begin{aligned}\widehat{c}_i &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_i + 2 - size_i) \\ &= -1 .\end{aligned}$$

  - If $\alpha_i < 1/2$, since $\alpha_{i-1} \geq 1/2$, have

$$num_i = num_{i-1} - 1 \geq \frac{1}{2} \cdot size_{i-1} - 1 = \frac{1}{2} \cdot size_i - 1 .$$

Thus,

$$\begin{aligned}
\widehat{c}_i &= 1 + (size_i/2 - num_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (2 \cdot num_i + 2 - size_i) \\
&= -1 + \frac{3}{2} \cdot size_i - 3 \cdot num_i \\
&\le -1 + \frac{3}{2} \cdot size_i - 3\left(\frac{1}{2} \cdot size_i - 1\right) \\
&= 2\,.
\end{aligned}$$

Therefore, amortized cost of delete is ≤ 2.

Solutions for Chapter 17: Amortized Analysis

Solution to Exercise 17.1-3

Let c_i = cost of ith operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

Operation	Cost
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1
⋮	⋮

n operations cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n \ .$$

(Note: Ignoring floor in upper bound of $\sum 2^j$.)

Average cost of operation $= \frac{\text{Total cost}}{\#\text{ operations}} < 3$

By aggregate analysis, the amortized cost per operation $= O(1)$.

Solution to Exercise 17.2-1

[We assume that the only way in which COPY *is invoked is automatically, after every sequence of k* PUSH *and* POP *operations.]*

Charge \$2 for each PUSH and POP operation and \$0 for each COPY. When we call PUSH, we use \$1 to pay for the operation, and we store the other \$1 on the item pushed. When we call POP, we again use \$1 to pay for the operation, and we store the other \$1 in the stack itself. Because the stack size never exceeds k, the actual cost of a COPY operation is at most \$$k$, which is paid by the \$$k$ found in the items in the stack and the stack itself. Since there are k PUSH and POP operations between two consecutive COPY operations, there are \$$k$ of credit stored, either on individual items (from PUSH operations) or in the stack itself (from POP operations) by the time a COPY occurs. Since the amortized cost of each operation is $O(1)$ and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

Solution to Exercise 17.2-2

Let c_i = cost of ith operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

Charge each operation \$3 (amortized cost $\widehat{c_i}$).

- If i is not an exact power of 2, pay \$1, and store \$2 as credit.
- If i is an exact power of 2, pay \$$i$, using stored credit.

Operation	Cost	Actual cost	Credit remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
⋮	⋮	⋮	⋮

Since the amortized cost is \$3 per operation, $\sum_{i=1}^{n} \widehat{c_i} = 3n$.

We know from Exercise 17.1-3 that $\sum_{i=1}^{n} c_i < 3n$.

Then we have $\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i \Rightarrow$ credit = amortized cost − actual cost ≥ 0.

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

Solution to Exercise 17.2-3

We introduce a new field $max[A]$ to hold the index of the high-order 1 in A. Initially, $max[A]$ is set to -1, since the low-order bit of A is at index 0, and there are initially no 1's in A. The value of $max[A]$ is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of A must be looked at to reset it. By controlling the cost of RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTs.

```
INCREMENT(A)
i ← 0
while i < length[A] and A[i] = 1
    do A[i] ← 0
       i ← i + 1
if i < length[A]
  then A[i] ← 1
       ▷ Additions to book's INCREMENT start here
       if i > max[A]
         then max[A] ← i
  else max[A] ← −1

RESET(A)
for i ← 0 to max[A]
    do A[i] ← 0
max[A] ← −1
```

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update $max[A]$.

Setting and resetting of bits by INCREMENT will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing.

In addition, we'll use \$1 to pay to update max, and if max increases, we'll place an additional \$1 of credit on the new high-order 1. (If max doesn't increase, we can just waste that \$1—it won't be needed.) Since RESET manipulates bits at positions only up to $max[A]$, and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to $max[A]$, every bit seen by RESET has \$1 of credit on it. So the zeroing of bits of A by RESET can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting max.

Thus charging \$4 for each INCREMENT and \$1 for each RESET is sufficient, so the sequence of n INCREMENT and RESET operations takes $O(n)$ time.

Solution to Exercise 17.3-3

Let D_i be the heap after the ith operation, and let D_i consist of n_i elements. Also, let k be a constant such that each INSERT or EXTRACT-MIN operation takes at most $k \ln n$ time, where $n = \max(n_{i-1}, n_i)$. (We don't want to worry about taking the log of 0, and at least one of n_{i-1} and n_i is at least 1. We'll see later why we use the natural log.)

Define

$$\Phi(D_i) = \begin{cases} 0 & \text{if } n_i = 0 \,, \\ k n_i \ln n_i & \text{if } n_i > 0 \,. \end{cases}$$

This function exhibits the characteristics we like in a potential function: if we start with an empty heap, then $\Phi(D_0) = 0$, and we always maintain that $\Phi(D_i) \ge 0$.

Before proving that we achieve the desired amortized times, we show that if $n \ge 2$, then $n \ln \frac{n}{n-1} \le 2$. We have

$$\begin{aligned} n \ln \frac{n}{n-1} &= n \ln \left(1 + \frac{1}{n-1}\right) \\ &= \ln \left(1 + \frac{1}{n-1}\right)^n \\ &\le \ln \left(e^{\frac{1}{n-1}}\right)^n && \text{(since } 1 + x \le e^x \text{ for all real } x\text{)} \\ &= \ln e^{\frac{n}{n-1}} \\ &= \frac{n}{n-1} \\ &\le 2 \,, \end{aligned}$$

assuming that $n \ge 2$. (The equation $\ln e^{\frac{n}{n-1}} = \frac{n}{n-1}$ is why we use the natural log.)

If the ith operation is an INSERT, then $n_i = n_{i-1} + 1$. If the ith operation inserts into an empty heap, then $n_i = 1$, $n_{i-1} = 0$, and the amortized cost is

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln 1 + k \cdot 1 \ln 1 - 0 \\ &= 0 \,. \end{aligned}$$

If the ith operation inserts into a nonempty heap, then $n_i = n_{i-1} + 1$, and the amortized cost is

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln n_i + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_i + k n_i \ln n_i - k(n_i - 1) \ln(n_i - 1) \\ &= k \ln n_i + k n_i \ln n_i - k n_i \ln(n_i - 1) + k \ln(n_i - 1) \\ &< 2k \ln n_i + k n_i \ln \frac{n_i}{n_i - 1} \\ &\le 2k \ln n_i + 2k \\ &= O(\lg n_i) \,. \end{aligned}$$

If the ith operation is an EXTRACT-MIN, then $n_i = n_{i-1} - 1$. If the ith operation extracts the one and only heap item, then $n_i = 0$, $n_{i-1} = 1$, and the amortized cost

is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln 1 + 0 - k \cdot 1 \ln 1 \\ &= 0 \, .\end{aligned}$$

If the ith operation extracts from a heap with more than 1 item, then $n_i = n_{i-1} - 1$ and $n_{i-1} \ge 2$, and the amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln n_{i-1} + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_{i-1} + k(n_{i-1} - 1) \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_{i-1} + k n_{i-1} \ln(n_{i-1} - 1) - k \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\ &= k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln \frac{n_{i-1} - 1}{n_{i-1}} \\ &< k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln 1 \\ &= k \ln \frac{n_{i-1}}{n_{i-1} - 1} \\ &\le k \ln 2 \qquad (\text{since } n_{i-1} \ge 2) \\ &= O(1) \, .\end{aligned}$$

A slightly different potential function—which may be easier to work with—is as follows. For each node x in the heap, let $d_i(x)$ be the depth of x in D_i. Define

$$\begin{aligned}\Phi(D_i) &= \sum_{x \in D_i} k(d_i(x) + 1) \\ &= k \left(n_i + \sum_{x \in D_i} d_i(x) \right) \, ,\end{aligned}$$

where k is defined as before.

Initially, the heap has no items, which means that the sum is over an empty set, and so $\Phi(D_0) = 0$. We always have $\Phi(D_i) \ge 0$, as required.

Observe that after an INSERT, the sum changes only by an amount equal to the depth of the new last node of the heap, which is $\lfloor \lg n_i \rfloor$. Thus, the change in potential due to an INSERT is $k(1 + \lfloor \lg n_i \rfloor)$, and so the amortized cost is $O(\lg n_i) + O(\lg n_i) = O(\lg n_i) = O(\lg n)$.

After an EXTRACT-MIN, the sum changes by the negative of the depth of the old last node in the heap, and so the potential *decreases* by $k(1 + \lfloor \lg n_{i-1} \rfloor)$. The amortized cost is at most $k \lg n_{i-1} - k(1 + \lfloor \lg n_{i-1} \rfloor) = O(1)$.

Solution to Problem 17-2

a. The SEARCH operation can be performed by searching each of the individually sorted arrays. Since all the individual arrays are sorted, searching one of them using a binary search algorithm takes $O(\lg m)$ time, where m is the size of the array. In an unsuccessful search, the time is $\Theta(\lg m)$. In the worst case, we may

assume that all the arrays $A_0, A_1, \ldots, A_{k-1}$ are full, $k = \lceil \lg(n+1) \rceil$, and we perform an unsuccessful search. The total time taken is

$$\begin{aligned} T(n) &= \Theta(\lg 2^{k-1} + \lg 2^{k-2} + \cdots + \lg 2^1 + \lg 2^0) \\ &= \Theta((k-1) + (k-2) + \cdots + 1 + 0) \\ &= \Theta(k(k-1)/2) \\ &= \Theta(\lceil \lg(n+1) \rceil \, (\lceil \lg(n+1) \rceil - 1)/2) \\ &= \Theta\left(\lg^2 n\right) . \end{aligned}$$

Thus, the worst-case running time is $\Theta(\lg^2 n)$.

b. We create a new sorted array of size 1 containing the new element to be inserted. If array A_0 (which has size 1) is empty, then we replace A_0 with the new sorted array. Otherwise, we merge sort the two arrays into another sorted array of size 2. If A_1 is empty, then we replace A_1 with the new array; otherwise we merge sort the arrays as before and continue. Since array A_i is of size 2^i, if we merge sort two arrays of size 2^i each, we obtain one of size 2^{i+1}, which is the size of A_{i+1}. Thus, this method will result in another list of arrays in the same structure that we had before.

Let us analyze its worst-case running time. We will assume that merge sort takes $2m$ time to merge two sorted lists of size m each. If all the arrays $A_0, A_1, \ldots, A_{k-2}$ are full, then the running time to fill array A_{k-1} would be

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

Therefore, the worst-case time to insert an element into this data structure is $\Theta(n)$.

However, let us now analyze the amortized running time. Using the aggregate method, we compute the total cost of a sequence of n inserts, starting with the empty data structure. Let r be the position of the rightmost 0 in the binary representation $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$ of n, so that $n_j = 1$ for $j = 0, 1, \ldots, r-1$. The cost of an insertion when n items have already been inserted is

$$\sum_{j=0}^{r-1} 2 \cdot 2^j = O(2^r) .$$

Furthermore, $r = 0$ half the time, $r = 1$ a quarter of the time, and so on. There are at most $\lceil n/2^r \rceil$ insertions for each value of r. The total cost of the n operations is therefore bounded by

$$O\left(\sum_{r=0}^{\lceil \lg(n+1) \rceil} \left(\left\lceil \frac{n}{2^r} \right\rceil \right) 2^r \right) = O(n \lg n) .$$

The amortized cost per INSERT operation, therefore is $O(\lg n)$.

We can also use the accounting method to analyze the running time. We can charge $\$k$ to insert an element. \$1 pays for the insertion, and we put $\$(k-1)$ on the inserted item to pay for it being involved in merges later on. Each time it is merged, it moves to a higher-indexed array, i.e., from A_i to A_{i+1}. It can

move to a higher-indexed array at most $k - 1$ times, and so the $\$(k - 1)$ on the item suffices to pay for all the times it will ever be involved in merges. Since $k = \Theta(\lg n)$, we have an amortized cost of $\Theta(\lg n)$ per insertion.

c. DELETE(x) will be implemented as follows:

1. Find the smallest j for which the array A_j with 2^j elements is full. Let y be the last element of A_j.
2. Let x be in the array A_i. If necessary, find which array this is by using the search procedure.
3. Remove x from A_i and put y into A_i. Then move y to its correct place in A_i.
4. Divide A_j (which now has $2^j - 1$ elements left): The first element goes into array A_0, the next 2 elements go into array A_1, the next 4 elements go into array A_2, and so forth. Mark array A_j as empty. The new arrays are created already sorted.

The cost of DELETE is $\Theta(n)$ in the worst case, where $i = k - 1$ and $j = k - 2$: $\Theta(\lg n)$ to find A_j, $\Theta(\lg^2 n)$ to find A_i, $\Theta(2^i) = \Theta(n)$ to put y in its correct place in array A_i, and $\Theta(2^j) = \Theta(n)$ to divide array A_j. The following sequence of n operations, where $n/3$ is a power of 2, yields an amortized cost that is no better: perform $n/3$ INSERT operations, followed by $n/3$ pairs of DELETE and INSERT. It costs $O(n \lg n)$ to do the first $n/3$ INSERT operations. This creates a single full array. Each subsequent DELETE/INSERT pair costs $\Theta(n)$ for the DELETE to divide the full array and another $\Theta(n)$ for the INSERT to recombine it. The total is then $\Theta(n^2)$, or $\Theta(n)$ per operation.

Solution to Problem 17-4

a. For RB-INSERT, consider a complete red-black tree in which the colors alternate between levels. That is, the root is black, the children of the root are red, the grandchildren of the root are black, the great-grandchildren of the root are red, and so on. When a node is inserted as a red child of one of the red leaves, then case 1 of RB-INSERT-FIXUP occurs $(\lg(n + 1))/2$ times, so that there are $\Omega(\lg n)$ color changes to fix the colors of nodes on the path from the inserted node to the root.

For RB-DELETE, consider a complete red-black tree in which all nodes are black. If a leaf is deleted, then the double blackness will be pushed all the way up to the root, with a color change at each level (case 2 of RB-DELETE-FIXUP), for a total of $\Omega(\lg n)$ color changes.

b. All cases except for case 1 of RB-INSERT-FIXUP and case 2 of RB-DELETE-FIXUP are terminating.

c. Case 1 of RB-INSERT-FIXUP reduces the number of red nodes by 1. As Figure 13.5 shows, node z's parent and uncle change from red to black, and z's grandparent changes from black to red. Hence, $\Phi(T') = \Phi(T) - 1$.

d. Lines 1–16 of RB-INSERT cause one node insertion and a unit increase in potential. The nonterminating case of RB-INSERT-FIXUP (Case 1) makes three

color changes and decreases the potential by 1. The terminating cases of RB-INSERT-FIXUP (cases 2 and 3) cause one rotation each and do not affect the potential. (Although case 3 makes color changes, the potential does not change. As Figure 13.6 shows, node z's parent changes from red to black, and z's grandparent changes from black to red.)

e. The number of structural modifications and amount of potential change resulting from lines 1–16 of RB-INSERT and from the terminating cases of RB-INSERT-FIXUP are $O(1)$, and so the amortized number of structural modifications of these parts is $O(1)$. The nonterminating case of RB-INSERT-FIXUP may repeat $O(\lg n)$ times, but its amortized number of structural modifications is 0, since by our assumption the unit decrease in the potential pays for the structural modifications needed. Therefore, the amortized number of structural modifications performed by RB-INSERT is $O(1)$.

f. From Figure 13.5, we see that case 1 of RB-INSERT-FIXUP makes the following changes to the tree:

- Changes a black node with two red children (node C) to a red node, resulting in a potential change of -2.
- Changes a red node (node A in part (a) and node B in part (b)) to a black node with one red child, resulting in no potential change.
- Changes a red node (node D) to a black node with no red children, resulting in a potential change of 1.

The total change in potential is -1, which pays for the structural modifications performed, and thus the amortized number of structural modifications in case 1 (the nonterminating case) is 0. The terminating cases of RB-INSERT-FIXUP cause $O(1)$ structural changes. Because $w(v)$ is based solely on node colors and the number of color changes caused by terminating cases is $O(1)$, the change in potential in terminating cases is $O(1)$. Hence, the amortized number of structural modifications in the terminating cases is $O(1)$. The overall amortized number of structural modifications in RB-INSERT, therefore, is $O(1)$.

g. Figure 13.7 shows that case 2 of RB-DELETE-FIXUP makes the following changes to the tree:

- Changes a black node with no red children (node D) to a red node, resulting in a potential change of -1.
- If B is red, then it loses a black child, with no effect on potential.
- If B is black, then it goes from having no red children to having one red child, resulting in a potential change of -1.

The total change in potential is either -1 or -2, depending on the color of B. In either case, one unit of potential pays for the structural modifications performed, and thus the amortized number of structural modifications in case 2 (the nonterminating case) is at most 0. The terminating cases of RB-DELETE cause $O(1)$ structural changes. Because $w(v)$ is based solely on node colors and the number of color changes caused by terminating cases is $O(1)$, the change in potential in terminating cases is $O(1)$. Hence, the amortized number of structural changes in the terminating cases is $O(1)$. The overall amortized number of structural modifications in RB-DELETE-FIXUP, therefore, is $O(1)$.

h. Since the amortized number structural modification in each operation is $O(1)$, the actual number of structural modifications for any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree is $O(m)$ in the worst case.

Lecture Notes for Chapter 21: Data Structures for Disjoint Sets

Chapter 21 overview

Disjoint-set data structures

- Also known as "union find."
- Maintain collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of disjoint dynamic (changing over time) sets.
- Each set is identified by a *representative*, which is some member of the set.

  Doesn't matter which member is the representative, as long as if we ask for the representative twice without modifying the set, we get the same answer both times.

[We do not include notes for the proof of running time of the disjoint-set forest implementation, which is covered in Section 21.4.]

Operations

- MAKE-SET(x): make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$.
- UNION(x, y): if $x \in S_x$, $y \in S_y$, then $\mathcal{S} \leftarrow \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$.
  - Representative of new set is any member of $S_x \cup S_y$, often the representative of one of S_x and S_y.
  - Destroys S_x and S_y (since sets must be disjoint).
- FIND-SET(x): return representative of set containing x.

Analysis in terms of:

- n = # of elements = # of MAKE-SET operations,
- m = total # of operations.

Analysis:

- Since MAKE-SET counts toward total # of operations, $m \geq n$.
- Can have at most $n - 1$ UNION operations, since after $n - 1$ UNIONs, only 1 set remains.
- Assume that the first n operations are MAKE-SET (helpful for analysis, usually not really necessary).

Application: dynamic connected components.

For a graph $G = (V, E)$, vertices u, v are in same connected component if and only if there's a path between them.

- Connected components partition vertices into equivalence classes.

```
CONNECTED-COMPONENTS(V, E)
for each vertex v ∈ V
    do MAKE-SET(v)
for each edge (u, v) ∈ E
    do if FIND-SET(u) ≠ FIND-SET(v)
          then UNION(u, v)
```

```
SAME-COMPONENT(u, v)
if FIND-SET(u) = FIND-SET(v)
  then return TRUE
  else return FALSE
```

Note: If actually implementing connected components,

- each vertex needs a handle to its object in the disjoint-set data structure,
- each object in the disjoint-set data structure needs a handle to its vertex.

Linked list representation

- Each set is a singly linked list.
- Each list node has fields for
  - the set member
  - pointer to the representative
  - next
- List has *head* (pointer to representative) and *tail*.

MAKE-SET: create a singleton list.

FIND-SET: return pointer to representative.

UNION: a couple of ways to do it.

1. UNION(x, y): append x's list onto end of y's list. Use y's tail pointer to find the end.

- Need to update the representative pointer for every node on x's list.
- If appending a large list onto a small list, it can take a while.

Operation	# objects updated
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
UNION(x_4, x_5)	4
$\vdots$	$\vdots$
UNION(x_{n-1}, x_n)	$n-1$
	$\Theta(n^2)$ total

Amortized time per operation $= \Theta(n)$.

2. ***Weighted-union heuristic:*** Always append the smaller list to the larger list.

   A single union can still take $\Omega(n)$ time, e.g., if both sets have $n/2$ members.

   Theorem
   With weighted union, a sequence of m operations on n elements takes $O(m + n \lg n)$ time.

   Sketch of proof Each MAKE-SET and FIND-SET still takes $O(1)$. How many times can each object's representative pointer be updated? It must be in the smaller set each time.

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\lg n$	$\geq n$

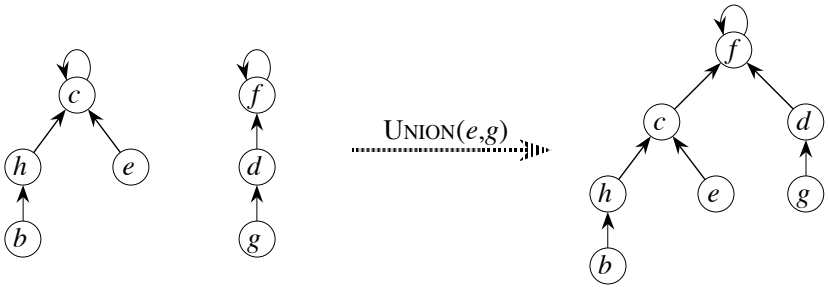
   Therefore, each representative is updated $\leq \lg n$ times. ■ (theorem)

   Seems pretty good, but we can do much better.

Disjoint-set forest

Forest of trees.

- 1 tree per set. Root is representative.
- Each node points only to its parent.

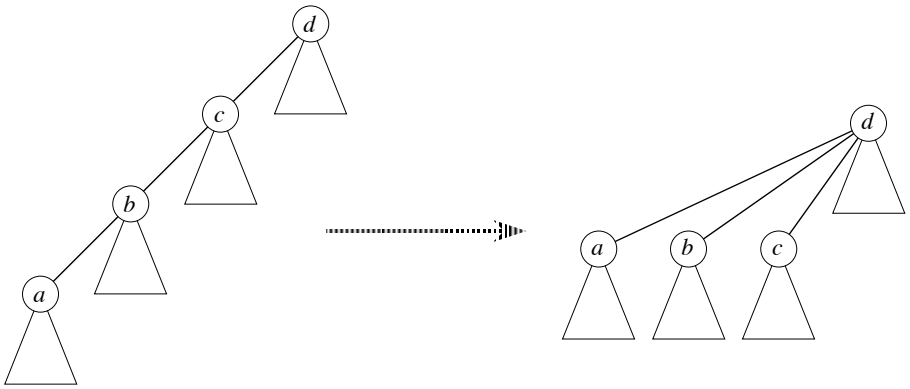


- MAKE-SET: make a single-node tree.
- UNION: make one root a child of the other.
- FIND-SET: follow pointers to the root.

Not so good—could get a linear chain of nodes.

Great heuristics

- ***Union by rank:*** make the root of the smaller tree (fewer nodes) a child of the root of the larger tree.
  - Don't actually use *size*.
  - Use *rank*, which is an upper bound on height of node.
  - Make the root with the smaller rank into a child of the root with the larger rank.
- ***Path compression: Find path*** = nodes visited during FIND-SET on the trip to the root. Make all nodes on the find path direct children of root.



MAKE-SET(x)
$p[x] \leftarrow x$
$rank[x] \leftarrow 0$

UNION(x, y)
LINK(FIND-SET(x), FIND-SET(y))

```
LINK(x, y)
if rank[x] > rank[y]
  then p[y] ← x
  else p[x] ← y
       ▷ If equal ranks, choose y as parent and increment its rank.
       if rank[x] = rank[y]
         then rank[y] ← rank[y] + 1
```

```
FIND-SET(x)
if x ≠ p[x]
  then p[x] ← FIND-SET(p[x])
return p[x]
```

FIND-SET makes a pass up to find the root, and a pass down as recursion unwinds to update each node on find path to point directly to root.

Running time

If use both union by rank and path compression, $O(m\,\alpha(n))$.

n	$\alpha(n)$
0–2	0
3	1
4–7	2
8–2047	3
2048–$A_4(1)$	4

What's $A_4(1)$? See Section 21.4, if you dare. It's $\gg 10^{80} \approx$ # of atoms in observable universe.

This bound is tight—there is a sequence of operations that takes $\Omega(m\,\alpha(n))$ time.

Solutions for Chapter 21: Data Structures for Disjoint Sets

Solution to Exercise 21.2-3

We want to show that we can assign $O(1)$ charges to MAKE-SET and FIND-SET and an $O(\lg n)$ charge to UNION such that the charges for a sequence of these operations are enough to cover the cost of the sequence — $O(m + n \lg n)$, according to the theorem. When talking about the charge for each kind of operation, it is helpful to also be able to talk about the number of each kind of operation.

Consider the usual sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, and let $l < n$ be the number of UNION operations. (Recall the discussion in Section 21.1 about there being at most $n - 1$ UNION operations.) Then there are n MAKE-SET operations, l UNION operations, and $m - n - l$ FIND-SET operations.

The theorem didn't separately name the number l of UNIONs; rather, it bounded the number by n. If you go through the proof of the theorem with l UNIONs, you get the time bound $O(m - l + l \lg l) = O(m + l \lg l)$ for the sequence of operations. That is, the actual time taken by the sequence of operations is at most $c(m + l \lg l)$, for some constant c.

Thus, we want to assign operation charges such that

$$\begin{array}{lcl} (\text{MAKE-SET charge}) & \cdot & n \\ +(\text{FIND-SET charge}) & \cdot & (m - n - l) \\ +(\text{UNION charge}) & \cdot & l \\ \hline \geq c(m + l \lg l) \ , & & \end{array}$$

so that the amortized costs give an upper bound on the actual costs.

The following assignments work, where c' is some constant $\geq c$:

- MAKE-SET: c'
- FIND-SET: c'
- UNION: $c'(\lg n + 1)$

Substituting into the above sum, we get

$$\begin{aligned} c'n + c'(m - n - l) + c'(\lg n + 1)l &= c'm + c'l \lg n \\ &= c'(m + l \lg n) \\ &> c(m + l \lg l) \ . \end{aligned}$$

Solution to Exercise 21.2-5

Let's call the two lists A and B, and suppose that the representative of the new list will be the representative of A. Rather than appending B to the end of A, instead splice B into A right after the first element of A. We have to traverse B to update representative pointers anyway, so we can just make the last element of B point to the second element of A.

Solution to Exercise 21.3-3

You need to find a sequence of m operations on n elements that takes $\Omega(m \lg n)$ time. Start with n MAKE-SETs to create singleton sets $\{x_1\}, \{x_2\}, \ldots, \{x_n\}$. Next perform the $n-1$ UNION operations shown below to create a single set whose tree has depth $\lg n$.

$\text{UNION}(x_1, x_2)$	$n/2$ of these
$\text{UNION}(x_3, x_4)$	
$\text{UNION}(x_5, x_6)$	
$\vdots$	
$\text{UNION}(x_{n-1}, x_n)$	
$\text{UNION}(x_2, x_4)$	$n/4$ of these
$\text{UNION}(x_6, x_8)$	
$\text{UNION}(x_{10}, x_{12})$	
$\vdots$	
$\text{UNION}(x_{n-2}, x_n)$	
$\text{UNION}(x_4, x_8)$	$n/8$ of these
$\text{UNION}(x_{12}, x_{16})$	
$\text{UNION}(x_{20}, x_{24})$	
$\vdots$	
$\text{UNION}(x_{n-4}, x_n)$	
$\vdots$	
$\text{UNION}(x_{n/2}, x_n)$	1 of these

Finally, perform $m - 2n + 1$ FIND-SET operations on the deepest element in the tree. Each of these FIND-SET operations takes $\Omega(\lg n)$ time. Letting $m \geq 3n$, we have more than $m/3$ FIND-SET operations, so that the total cost is $\Omega(m \lg n)$.

Solution to Exercise 21.3-4

With the path-compression heuristic, the sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations take place before any of the

FIND-SET operations, runs in $O(m)$ time. The key observation is that once a node x appears on a find path, x will be either a root or a child of a root at all times thereafter.

We use the accounting method to obtain the $O(m)$ time bound. We charge a MAKE-SET operation two dollars. One dollar pays for the MAKE-SET, and one dollar remains on the node x that is created. The latter pays for the first time that x appears on a find path and is turned into a child of a root.

We charge one dollar for a LINK operation. This dollar pays for the actual linking of one node to another.

We charge one dollar for a FIND-SET. This dollar pays for visiting the root and its child, and for the path compression of these two nodes, during the FIND-SET. All other nodes on the find path use their stored dollar to pay for their visitation and path compression. As mentioned, after the FIND-SET, all nodes on the find path become children of a root (except for the root itself), and so whenever they are visited during a subsequent FIND-SET, the FIND-SET operation itself will pay for them.

Since we charge each operation either one or two dollars, a sequence of m operations is charged at most $2m$ dollars, and so the total time is $O(m)$.

Observe that nothing in the above argument requires union by rank. Therefore, we get an $O(m)$ time bound regardless of whether we use union by rank.

Solution to Exercise 21.4-4

Clearly, each MAKE-SET and LINK operation takes $O(1)$ time. Because the rank of a node is an upper bound on its height, each find path has length $O(\lg n)$, which in turn implies that each FIND-SET takes $O(\lg n)$ time. Thus, any sequence of m MAKE-SET, LINK, and FIND-SET operations on n elements takes $O(m \lg n)$ time. It is easy to prove an analogue of Lemma 21.7 to show that if we convert a sequence of m' MAKE-SET, UNION, and FIND-SET operations into a sequence of m MAKE-SET, LINK, and FIND-SET operations that take $O(m \lg n)$ time, then the sequence of m' MAKE-SET, UNION, and FIND-SET operations takes $O(m' \lg n)$ time.

Solution to Exercise 21.4-5

Professor Dante is mistaken. Take the following scenario. Let $n = 16$, and make 16 separate singleton sets using MAKE-SET. Then do 8 UNION operations to link the sets into 8 pairs, where each pair has a root with rank 0 and a child with rank 1. Now do 4 UNIONs to link pairs of these trees, so that there are 4 trees, each with a root of rank 2, children of the root of ranks 1 and 0, and a node of rank 0 that is the child of the rank-1 node. Now link pairs of these trees together, so that there are two resulting trees, each with a root of rank 3 and each containing a path from a leaf to the root with ranks 0, 1, and 3. Finally, link these two trees together, so that

there is a path from a leaf to the root with ranks 0, 1, 3, and 4. Let x and y be the nodes on this path with ranks 1 and 3, respectively. Since $A_1(1) = 3$, level$(x) = 1$, and since $A_0(3) = 4$, level$(y) = 0$. Yet y follows x on the find path.

Solution to Exercise 21.4-6

First, $\alpha'(2^{2047} - 1) = \min\{k : A_k(1) \geq 2047\} = 3$, and $2^{2047} - 1 \gg 10^{80}$.

Second, we need that $0 \leq \text{level}(x) \leq \alpha'(n)$ for all nonroots x with $rank[x] \geq 1$. With this definition of $\alpha'(n)$, we have $A_{\alpha'(n)}(\text{rank}[x]) \geq A_{\alpha'(n)}(1) \geq \lg(n+1) > \lg n \geq rank(p[x])$. The rest of the proof goes through with $\alpha'(n)$ replacing $\alpha(n)$.

Solution to Problem 21-1

a. For the input sequence

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

the values in the *extracted* array would be 4, 3, 2, 6, 8, 1.

The following table shows the situation after the ith iteration of the **for** loop when we use OFF-LINE-MINIMUM on the same input. (For this input, $n = 9$ and m—the number of extractions—is 6).

i	K_1	K_2	K_3	K_4	K_5	K_6	K_7	*extracted*					
								1	2	3	4	5	6
0	{4, 8}	{3}	{9, 2, 6}	{}	{}	{1, 7}	{5}						
1	{4, 8}	{3}	{9, 2, 6}	{}	{}		{5, 1, 7}						1
2	{4, 8}	{3}		{9, 2, 6}	{}		{5, 1, 7}			2			1
3	{4, 8}			{9, 2, 6, 3}	{}		{5, 1, 7}		3	2			1
4				{9, 2, 6, 3, 4, 8}	{}		{5, 1, 7}	4	3	2			1
5				{9, 2, 6, 3, 4, 8}	{}		{5, 1, 7}	4	3	2			1
6					{9, 2, 6, 3, 4, 8}		{5, 1, 7}	4	3	2	6		1
7					{9, 2, 6, 3, 4, 8}		{5, 1, 7}	4	3	2	6		1
8							{5, 1, 7, 9, 2, 6, 3, 4, 8}	4	3	2	6	8	1

Because $j = m + 1$ in the iterations for $i = 5$ and $i = 7$, no changes occur in these iterations.

b. We want to show that the array *extracted* returned by OFF-LINE-MINIMUM is correct, meaning that for $i = 1, 2, \ldots, m$, $extracted[j]$ is the key returned by the jth EXTRACT-MIN call.

We start with n INSERT operations and m EXTRACT-MIN operations. The smallest of all the elements will be extracted in the first EXTRACT-MIN after its insertion. So we find j such that the minimum element is in K_j, and put the minimum element in $extracted[j]$, which corresponds to the EXTRACT-MIN after the minimum element insertion.

Now we reduce to a similar problem with $n - 1$ INSERT operations and $m - 1$ EXTRACT-MIN operations in the following way: the INSERT operations are

the same but without the insertion of the smallest that was extracted, and the EXTRACT-MIN operations are the same but without the extraction that extracted the smallest element.

Conceptually, we unite I_j and I_{j+1}, removing the extraction between them and also removing the insertion of the minimum element from $I_j \cup I_{j+1}$. Uniting I_j and I_{j+1} is accomplished by line 6. We need to determine which set is K_l, rather than just using K_{j+1} unconditionally, because K_{j+1} may have been destroyed when it was united into a higher-indexed set by a previous execution of line 6.

Because we process extractions in increasing order of the minimum value found, the remaining iterations of the **for** loop correspond to solving the reduced problem.

There are two other points worth making. First, if the smallest remaining element had been inserted after the last EXTRACT-MIN (i.e., $j = m + 1$), then no changes occur, because this element is not extracted. Second, there may be smaller elements within the K_j sets than the the one we are currently looking for. These elements do not affect the result, because they correspond to elements that were already extracted, and their effect on the algorithm's execution is over.

c. To implement this algorithm, we place each element in a disjoint-set forest. Each root has a pointer to its K_i set, and each K_i set has a pointer to the root of the tree representing it. All the valid sets K_i are in a linked list.

Before OFF-LINE-MINIMUM, there is initialization that builds the initial sets K_i according to the I_i sequences.

- Line 2 ("determine j such that $i \in K_j$") turns into $j \leftarrow$ FIND-SET(i).
- Line 5 ("let l be the smallest value greater than j for which set K_l exists") turns into $K_l \leftarrow next[K_j]$.
- Line 6 ("$K_l \leftarrow K_j \cup K_l$, destroying K_j") turns into $l \leftarrow$ LINK(j, l) and remove K_j from the linked list.

To analyze the running time, we note that there are n elements and that we have the following disjoint-set operations:

- n MAKE-SET operations
- at most $n - 1$ UNION operations before starting
- n FIND-SET operations
- at most n LINK operations

Thus the number m of overall operations is $O(n)$. The total running time is $O(m\,\alpha(n)) = O(n\,\alpha(n))$.

[The "tight bound" wording that this question uses does not refer to an "asymptotically tight" bound. Instead, the question is merely asking for a bound that is not too "loose."]

Solution to Problem 21-2

a. Denote the number of nodes by n, and let $n = (m + 1)/3$, so that $m = 3n - 1$. First, perform the n operations MAKE-TREE(v_1), MAKE-TREE(v_2), ..., MAKE-TREE(v_n). Then perform the sequence of $n - 1$ GRAFT operations GRAFT(v_1, v_2), GRAFT(v_2, v_3), ..., GRAFT(v_{n-1}, v_n); this sequence produces a single disjoint-set tree that is a linear chain of n nodes with v_n at the root and v_1 as the only leaf. Then perform FIND-DEPTH(v_1) repeatedly, n times. The total number of operations is $n + (n - 1) + n = 3n - 1 = m$.

Each MAKE-TREE and GRAFT operation takes $O(1)$ time. Each FIND-DEPTH operation has to follow an n-node find path, and so each of the n FIND-DEPTH operations takes $\Theta(n)$ time. The total time is $n \cdot \Theta(n) + (2n - 1) \cdot O(1) = \Theta(n^2) = \Theta(m^2)$.

b. MAKE-TREE is like MAKE-SET, except that it also sets the d value to 0:

```
MAKE-TREE(v)
p[v] ← v
rank[v] ← 0
d[v] ← 0
```

It is correct to set $d[v]$ to 0, because the depth of the node in the single-node disjoint-set tree is 0, and the sum of the depths on the find path for v consists only of $d[v]$.

c. FIND-DEPTH will call a procedure FIND-ROOT:

```
FIND-ROOT(v)
if p[v] ≠ p[p[v]]
  then y ← p[v]
       p[v] ← FIND-ROOT(y)
       d[v] ← d[v] + d[y]
return p[v]

FIND-DEPTH(v)
FIND-ROOT(v)          ▷ No need to save the return value.
if v = p[v]
  then return d[v]
  else return d[v] + d[p[v]]
```

FIND-ROOT performs path compression and updates pseudodistances along the find path from v. It is similar to FIND-SET on page 508, but with three changes. First, when v is either the root or a child of a root (one of these conditions holds if and only if $p[v] = p[p[v]]$) in the disjoint-set forest, we don't have to recurse; instead, we just return $p[v]$. Second, when we do recurse, we save the pointer $p[v]$ into a new variable y. Third, when we recurse, we update $d[v]$ by adding into it the d values of all nodes on the find path that are no longer proper

ancestors of v after path compression; these nodes are precisely the proper ancestors of v other than the root. Thus, as long as v does not start out the FIND-ROOT call as either the root or a child of the root, we add $d[y]$ into $d[v]$. Note that $d[y]$ has been updated prior to updating $d[v]$, if y is also neither the root nor a child of the root.

FIND-DEPTH first calls FIND-ROOT to perform path compression and update pseudodistances. Afterward, the find path from v consists of either just v (if v is a root) or just v and $p[v]$ (if v is not a root, in which case it is a child of the root after path compression). In the former case, the depth of v is just $d[v]$, and in the latter case, the depth is $d[v] + d[p[v]]$.

d. Our procedure for GRAFT is a combination of UNION and LINK:

```
GRAFT(r, v)
r' ← FIND-ROOT(r)
v' ← FIND-ROOT(v)
z ← FIND-DEPTH(v)
if rank[r'] > rank[v']
  then p[v'] ← r'
       d[r'] ← d[r'] + z + 1
       d[v'] ← d[v'] − d[r']
  else p[r'] ← v'
       d[r'] ← d[r'] + z + 1 − d[v']
       if rank[r'] = rank[v']
         then rank[v'] ← rank[v'] + 1
```

This procedure works as follows. First, we call FIND-ROOT on r and v in order to find the roots r' and v', respectively, of their trees in the disjoint-set forest. As we saw in part (c), these FIND-ROOT calls also perform path compression and update pseudodistances on the find paths from r and v. We then call FIND-DEPTH(v), saving the depth of v in the variable z. (Since we have just compressed v's find path, this call of FIND-DEPTH takes $O(1)$ time.) Next, we emulate the action of LINK, by making the root (r' or v') of smaller rank a child of the root of larger rank; in case of a tie, we make r' a child of v'.

If v' has the smaller rank, then all nodes in r's tree will have their depths increased by the depth of v plus 1 (because r is to become a child of v). Altering the psuedodistance of the root of a disjoint-set tree changes the computed depth of all nodes in that tree, and so adding $z + 1$ to $d[r']$ accomplishes this update for all nodes in r's disjoint-set tree. Since v' will become a child of r' in the disjoint-set forest, we have just increased the computed depth of all nodes in the disjoint-set tree rooted at v' by $d[r']$. These computed depths should not have changed, however. Thus, we subtract off $d[r']$ from $d[v']$, so that the sum $d[v'] + d[r']$ after making v' a child of r' equals $d[v']$ before making v' a child of r'.

On the other hand, if r' has the smaller rank, or if the ranks are equal, then r' becomes a child of v' in the disjoint-set forest. In this case, v' remains a root in the disjoint-set forest afterward, and we can leave $d[v']$ alone. We have to update $d[r']$, however, so that after making r' a child of v', the depth of each node in r's disjoint-set tree is increased by $z + 1$. We add $z + 1$ to $d[r']$, but we

also subtract out $d[v']$, since we have just made r' a child of v'. Finally, if the ranks of r' and v' are equal, we increment the rank of v', as is done in the LINK procedure.

e. The asymptotic running times of MAKE-TREE, FIND-DEPTH, and GRAFT are equivalent to those of MAKE-SET, FIND-SET, and UNION, respectively. Thus, a sequence of m operations, n of which are MAKE-TREE operations, takes $\Theta(m\,\alpha(n))$ time in the worst case.

Lecture Notes for Chapter 22: Elementary Graph Algorithms

Graph representation

Given graph $G = (V, E)$.

- May be either directed or undirected.
- Two common ways to represent for algorithms:
  1. Adjacency lists.
  2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$. In asymptotic notation—and *only* in asymptotic notation—we'll drop the cardinality. Example: $O(V + E)$.

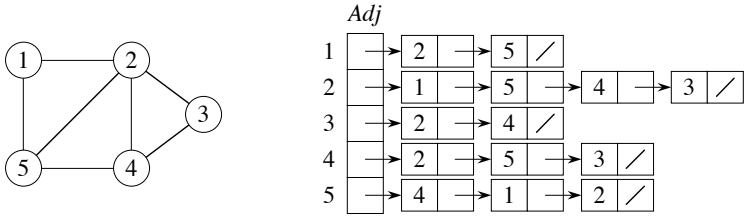
[The introduction to Part VI talks more about this.]

Adjacency lists

Array *Adj* of $|V|$ lists, one per vertex.

Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

Example: For an undirected graph:



If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbf{R}$

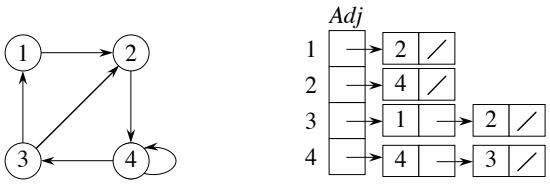
We'll use weights later on for spanning trees and shortest paths.

Space: $\Theta(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine if $(u, v) \in E$: $O(\text{degree}(u))$.

Example: For a directed graph:



Same asymptotic space and time.

Adjacency matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine if $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

We'll use both representations in these lecture notes.

Breadth-first search

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.

Output: $d[v]$ = distance (smallest # of edges) from s to v, for all $v \in V$.
In book, also $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.

- u is v's ***predecessor***.
- set of edges $\{(\pi[v], v) : v \neq s\}$ forms a tree.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple.

- Compute only $d[v]$, not $\pi[v]$. *[See book for $\pi[v]$.]*
- Omitting colors of vertices. *[Used in book to reason about the algorithm. We'll skip them here.]*

Idea: Send a wave out from s.

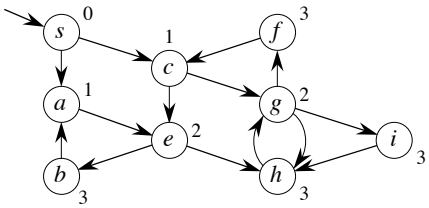
- First hits all vertices 1 edge from s.
- From there, hits all vertices 2 edges from s.
- Etc.

Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has hit v but has not come out of v yet.

```
BFS(V, E, s)
for each u ∈ V − {s}
    do d[u] ← ∞
d[s] ← 0
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if d[v] = ∞
                 then d[v] ← d[u] + 1
                      ENQUEUE(Q, v)
```

Example: directed graph *[undirected example in book]* .



Can show that Q consists of vertices with d values.

$i \quad i \quad i \quad \ldots \quad i \quad i+1 \quad i+1 \quad \ldots \quad i+1$

- Only 1 or 2 values.
- If 2, differ by 1 and all smallest are first.

Since each vertex gets a finite d value at most once, values assigned to vertices are monotonically increasing over time.

Actual proof of correctness is a bit trickier. See book.

BFS may not reach all vertices.

Time $= O(V + E)$.

- $O(V)$ because every vertex enqueued at most once.
- $O(E)$ because every vertex dequeued at most once and we examine (u, v) only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

Depth-first search

Input: $G = (V, E)$, directed or undirected. No source vertex given!

Output: 2 ***timestamps*** on each vertex:

- $d[v] =$ ***discovery time***
- $f[v] =$ ***finishing time***

These will be useful for other algorithms later on.

Can also compute $\pi[v]$. *[See book.]*

Will methodically explore *every* edge.

- Start over from different vertices as necessary.

As soon as we discover a vertex, explore from it.

- Unlike BFS, which puts a vertex on a queue so that we explore from it later.

As DFS progresses, every vertex has a ***color***:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

Discovery and finish times:

- Unique integers from 1 to $2|V|$.
- For all v, $d[v] < f[v]$.

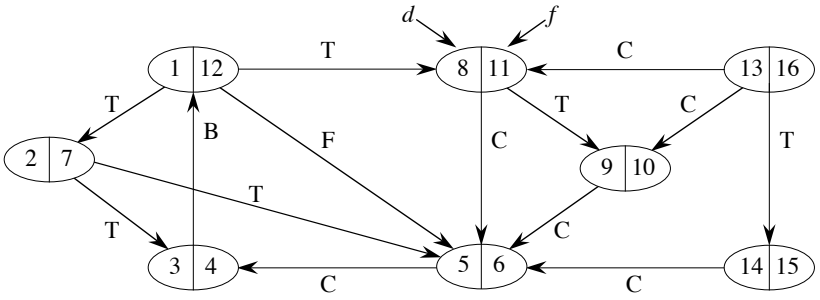
In other words, $1 \le d[v] < f[v] \le 2|V|$.

Pseudocode: Uses a global timestamp *time*.

```
DFS(V, E)
for each u ∈ V
    do color[u] ← WHITE
time ← 0
for each u ∈ V
    do if color[u] = WHITE
          then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY          ▷ discover u
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]      ▷ explore (u, v)
    do if color[v] = WHITE
          then DFS-VISIT(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time              ▷ finish u
```

Example: *[Go through this example, adding in the d and f values as they're computed. Show colors as they change. Don't put in the edge types yet.]*



Time $= \Theta(V + E)$.

- Similar to BFS analysis.
- Θ, not just O, since guaranteed to examine every vertex and edge.

DFS forms a ***depth-first forest*** comprised of > 1 ***depth-first trees***. Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

Theorem (Parenthesis theorem)
[Proof omitted.]

For all u, v, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither of u and v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.

Like parentheses:

- OK: ( ) [ ] ([ ]) [( )]
- Not OK: ([ )] [( ])

Corollary
v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.

Theorem (White-path theorem)
[Proof omitted.]

v is a descendant of u if and only if at time $d[u]$, there is a path $u \rightsquigarrow v$ consisting of only white vertices. (Except for u, which was *just* colored gray.)

Classification of edges

- ***Tree edge:*** in the depth-first forest. Found by exploring (u, v).
- ***Back edge:*** (u, v), where u is a descendant of v.
- ***Forward edge:*** (u, v), where v is a descendant of u, but not a tree edge.
- ***Cross edge:*** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

[Now label the example from above with edge types.]

In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Classify by the first type above that matches.

Theorem

[Proof omitted.]

In DFS of an *un*directed graph, we get only tree and back edges. No forward or cross edges.

Topological sort

Directed acyclic graph (dag)

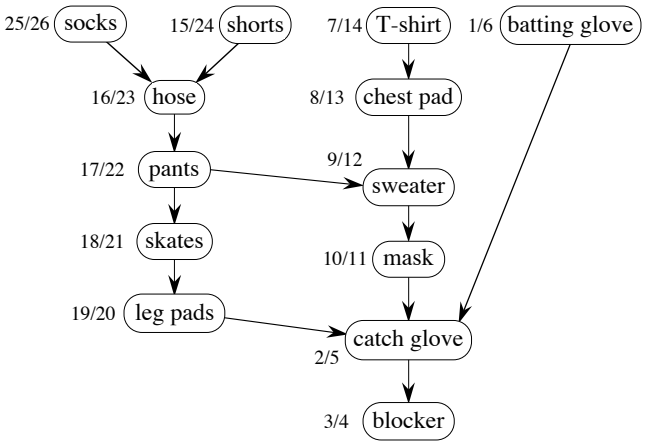
A directed graph with no cycles.

Good for modeling processes and structures that have a ***partial order:***

- $a > b$ and $b > c \Rightarrow a > c$.
- But may have a and b such that neither $a > b$ nor $b > c$.

Can always make a ***total order*** (either $a > b$ or $b > a$ for all $a \neq b$) from a partial order. In fact, that's what a topological sort will do.

Example: dag of dependencies for putting on goalie equipment: *[Leave on board, but show without discovery and finish times. Will put them in later.]*

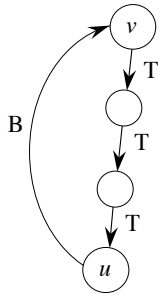


Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Proof $\Rightarrow$: Show that back edge $\Rightarrow$ cycle.

Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest.



Therefore, there is a path $v \rightsquigarrow u$, so $v \rightsquigarrow u \rightarrow v$ is a cycle.

$\Leftarrow$: Show that cycle $\Rightarrow$ back edge.

Suppose G contains cycle c. Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in c. At time $d[v]$, vertices of c form a white path $v \rightsquigarrow u$ (since v is the first vertex discovered in c). By white-path theorem, u is descendant of v in depth-first forest. Therefore, (u, v) is a back edge. ■ (lemma)

Topological sort of a dag: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v. (Not like sorting numbers.)

TOPOLOGICAL-SORT(V, E)
call DFS(V, E) to compute finishing times $f[v]$ for all $v \in V$
output vertices in order of *decreasing* finish times

Don't need to sort by finish times.

- Can just output vertices as they're finished and understand that we want the *reverse* of this list.
- Or put them onto the *front* of a linked list as they're finished. When done, the list contains vertices in topologically sorted order.

Time: $\Theta(V + E)$.

Do example. *[Now write discovery and finish times in goalie equipment example.]*

Order:

26 socks
24 shorts
23 hose
22 pants
21 skates
20 leg pads
14 t-shirt
13 chest pad
12 sweater
11 mask
6 batting glove
5 catch glove
4 blocker

Correctness: Just need to show if $(u, v) \in E$, then $f[v] < f[u]$.
When we explore (u, v), what are the colors of u and v?

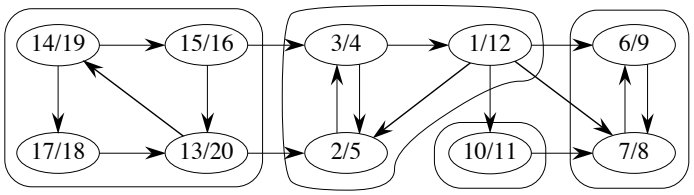
- u is gray.
- Is v gray, too?
  - *No*, because then v would be ancestor of u.
    $\Rightarrow (u, v)$ is a back edge.
    $\Rightarrow$ contradiction of previous lemma (dag has no back edges).
- Is v white?
  - Then becomes descendant of u.
    By parenthesis theorem, $d[u] < d[v] < \underline{f[v] < f[u]}$.
- Is v black?
  - Then v is already finished.
    Since we're exploring (u, v), we have not yet finished u.
    Therefore, $f[v] < f[u]$. ■

Strongly connected components

Given directed graph $G = (V, E)$.

A ***strongly connected component*** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example: *[Just show SCC's at first. Do DFS a little later.]*



Algorithm uses G^T = ***transpose*** of G.

- $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
- G^T is G with all edges reversed.

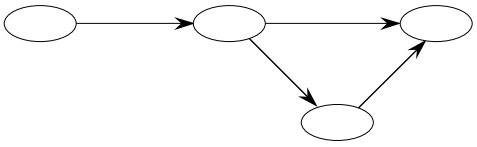
Can create G^T in $\Theta(V + E)$ time if using adjacency lists.

Observation: G and G^T have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^T.)

Component graph

- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.

For our example:



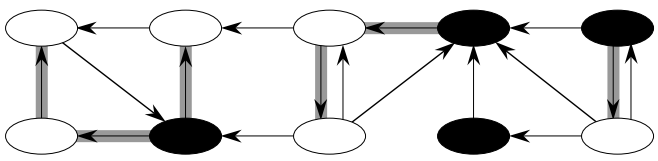
Lemma
G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof Suppose there is a path $v' \rightsquigarrow v$ in G. Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G. Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ■ (lemma)

SCC(G)
call DFS(G) to compute finishing times $f[u]$ for all u
compute G^T
call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$
    (as computed in first DFS)
output the vertices in each tree of the depth-first forest formed in second DFS
    as a separate SCC

Example:

1. Do DFS
2. G^T
3. DFS (roots blackened)



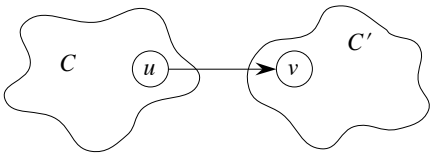
Time: $\Theta(V + E)$.
How can this possibly work?

Idea: By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the component graph in topological sort order.

To prove that it works, first deal with 2 notational issues:

- Will be discussing $d[u]$ and $f[u]$. These always refer to *first* DFS.
- Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(U) = \min_{u \in U} \{d[u]\}$ (earliest discovery time)
  - $f(U) = \max_{u \in U} \{f[u]\}$ (latest finishing time)

Lemma
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$.



Then $f(C) > f(C')$.

Proof Two cases, depending on which SCC had the first discovered vertex during the first DFS.

- If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.

  By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.

  By the parenthesis theorem, $f[x] = f(C) > f(C')$.
- If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in $C' \Rightarrow$ all vertices in C' become descendants of y. Again, $f[y] = f(C')$.

  At time $d[y]$, all vertices in C are white.

  By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C.

  So no vertex in C is reachable from y.

  Therefore, at time $f[y]$, all vertices in C are still white.

  Therefore, for all $w \in C$, $f[w] > f[y]$, which implies that $f(C) > f(C')$.

  ■ (lemma)

Corollary
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^{\mathrm{T}}$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof $(u, v) \in E^{\mathrm{T}} \Rightarrow (v, u) \in E$. Since SCC's of G and G^{T} are the same, $f(C') > f(C)$. ■ (corollary)

Corollary
Let C and C' be distinct SCC's in $G = (V, E)$, and suppose that $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^{T}.

Proof It's the contrapositive of the previous corollary. ■

Now we have the intuition to understand why the SCC procedure works.

When we do the second DFS, on G^{T}, start with SCC C such that $f(C)$ is maximum. The second DFS starts from some $x \in C$, and it visits all vertices in C. Corollary says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^{T}.

Therefore, DFS will visit *only* vertices in C.

Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCC's other than C. DFS visits all vertices in C', but the only edges out of C' go to C, *which we've already visited.*

Therefore, the only tree edges will be to vertices in C'.

We can continue the process.

Each time we choose a root for the second DFS, it can reach only

- vertices in its SCC—get tree edges to these,
- vertices in SCC's *already visited* in second DFS—get *no* tree edges to these.

We are visiting vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in reverse of topologically sorted order.

[The book has a formal proof.]

Solutions for Chapter 22: Elementary Graph Algorithms

Solution to Exercise 22.1-6

We start by observing that if $a_{ij} = 1$, so that $(i, j) \in E$, then vertex i cannot be a universal sink, for it has an outgoing edge. Thus, if row i contains a 1, then vertex i cannot be a universal sink. This observation also means that if there is a self-loop (i, i), then vertex i is not a universal sink. Now suppose that $a_{ij} = 0$, so that $(i, j) \notin E$, and also that $i \neq j$. Then vertex j cannot be a universal sink, for either its in-degree must be strictly less than $|V| - 1$ or it has a self-loop. Thus if column j contains a 0 in any position other than the diagonal entry (j, j), then vertex j cannot be a universal sink.

Using the above observations, the following procedure returns TRUE if vertex k is a universal sink, and FALSE otherwise. It takes as input a $|V| \times |V|$ adjacency matrix $A = (a_{ij})$.

```
IS-SINK(A, k)
let A be |V| × |V|
for j ← 1 to |V|                    ▷ Check for a 1 in row k
    do if a_kj = 1
          then return FALSE
for i ← 1 to |V|                    ▷ Check for an off-diagonal 0 in column k
    do if a_ik = 0 and i ≠ k
          then return FALSE
return TRUE
```

Because this procedure runs in $O(V)$ time, we may call it only $O(1)$ times in order to achieve our $O(V)$-time bound for determining whether directed graph G contains a universal sink.

Observe also that a directed graph can have at most one universal sink. This property holds because if vertex j is a universal sink, then we would have $(i, j) \in E$ for all $i \neq j$ and so no other vertex i could be a universal sink.

The following procedure takes an adjacency matrix A as input and returns either a message that there is no universal sink or a message containing the identity of the universal sink. It works by eliminating all but one vertex as a potential universal sink and then checking the remaining candidate vertex by a single call to IS-SINK.

```
UNIVERSAL-SINK(A)
let A be |V| × |V|
i ← j ← 1
while i ≤ |V| and j ≤ |V|
    do if a_ij = 1
          then i ← i + 1
          else  j ← j + 1
s ← 0
if i > |V|
  then return "there is no universal sink"
elseif IS-SINK(A, i) = FALSE
  then return "there is no universal sink"
else return i "is a universal sink"
```

UNIVERSAL-SINK walks through the adjacency matrix, starting at the upper left corner and always moving either right or down by one position, depending on whether the current entry a_{ij} it is examining is 0 or 1. It stops once either i or j exceeds $|V|$.

To understand why UNIVERSAL-SINK works, we need to show that after the **while** loop terminates, the only vertex that might be a universal sink is vertex i. The call to IS-SINK then determines whether vertex i is indeed a universal sink.

Let us fix i and j to be values of these variables at the termination of the **while** loop. We claim that every vertex k such that $1 \le k < i$ cannot be a universal sink. That is because the way that i achieved its final value at loop termination was by finding a 1 in each row k for which $1 \le k < i$. As we observed above, any vertex k whose row contains a 1 cannot be a universal sink.

If $i > |V|$ at loop termination, then we have eliminated all vertices from consideration, and so there is no universal sink. If, on the other hand, $i \le |V|$ at loop termination, we need to show that every vertex k such that $i < k \le |V|$ cannot be a universal sink. If $i \le |V|$ at loop termination, then the **while** loop terminated because $j > |V|$. That means that we found a 0 in every column. Recall our earlier observation that if column k contains a 0 in an off-diagonal position, then vertex k cannot be a universal sink. Since we found a 0 in every column, we found a 0 in every column k such that $i < k \le |V|$. Moreover, we never examined any matrix entries in rows greater than i, and so we never examined the diagonal entry in any column k such that $i < k \le |V|$. Therefore, all the 0s that we found in columns k such that $i < k \le |V|$ were off-diagonal. We conclude that every vertex k such that $i < k \le |V|$ cannot be a universal sink.

Thus, we have shown that every vertex less than i and every vertex greater than i cannot be a universal sink. The only remaining possibility is that vertex i might be a universal sink, and the call to IS-SINK checks whether it is.

To see that UNIVERSAL-SINK runs in $O(V)$ time, observe that either i or j is incremented in each iteration of the **while** loop. Thus, the **while** loop makes at most $2|V| - 1$ iterations. Each iteration takes $O(1)$ time, for a total **while** loop time of $O(V)$ and, combined with the $O(V)$-time call to IS-SINK, we get a total running time of $O(V)$.

Solution to Exercise 22.1-7

$$BB^T(i, j) = \sum_{e \in E} b_{ie} b_{ej}^T = \sum_{e \in E} b_{ie} b_{je}$$

- If $i = j$, then $b_{ie} b_{je} = 1$ (it is $1 \cdot 1$ or $(-1) \cdot (-1)$) whenever e enters or leaves vertex i, and 0 otherwise.
- If $i \neq j$, then $b_{ie} b_{je} = -1$ when $e = (i, j)$ or $e = (j, i)$, and 0 otherwise.

Thus,

$$BB^T(i, j) = \begin{cases} \text{degree of } i = \text{in-degree} + \text{out-degree} & \text{if } i = j \ , \\ -(\#\text{ of edges connecting } i \text{ and } j) & \text{if } i \neq j \ . \end{cases}$$

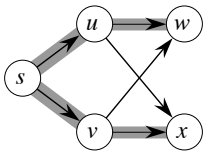
Solution to Exercise 22.2-4

The correctness proof for the BFS algorithm shows that $d[u] = \delta(s, u)$, and the algorithm doesn't assume that the adjacency lists are in any particular order.

In Figure 22.3, if t precedes x in $Adj[w]$, we can get the breadth-first tree shown in the figure. But if x precedes t in $Adj[w]$ and u precedes y in $Adj[x]$, we can get edge (x, u) in the breadth-first tree.

Solution to Exercise 22.2-5

The edges in E_π are shaded in the following graph:



To see that E_π cannot be a breadth-first tree, let's suppose that $Adj[s]$ contains u before v. BFS adds edges (s, u) and (s, v) to the breadth-first tree. Since u is enqueued before v, BFS then adds edges (u, w) and (u, x). (The order of w and x in $Adj[u]$ doesn't matter.) Symmetrically, if $Adj[s]$ contains v before u, then BFS adds edges (s, v) and (s, u) to the breadth-first tree, v is enqueued before u, and BFS adds edges (v, w) and (v, x). (Again, the order of w and x in $Adj[v]$ doesn't matter.) BFS will never put both edges (u, w) and (v, x) into the breadth-first tree. In fact, it will also never put both edges (u, x) and (v, w) into the breadth-first tree.

Solution to Exercise 22.2-6

Create a graph G where each vertex represents a wrestler and each edge represents a rivalry. The graph will contain n vertices and r edges.

Perform as many BFS's as needed to visit all vertices. Assign all wrestlers whose distance is even to be good guys and all wrestlers whose distance is odd to be bad guys. Then check each edge to verify that it goes between a good guy and a bad guy. This solution would take $O(n+r)$ time for the BFS, $O(n)$ time to designate each wrestler as a good guy or bad guy, and $O(r)$ time to check edges, which is $O(n+r)$ time overall.

Solution to Exercise 22.3-4

a. Edge (u, v) is a tree edge or forward edge if and only if v is a descendant of u in the depth-first forest. (If (u, v) is a back edge, then u is a descendant of v, and if (u, v) is a cross edge, then neither of u or v is a descendant of the other.) By Corollary 22.8, therefore, (u, v) is a tree edge or forward edge if and only if $d[u] < d[v] < f[v] < f[u]$.

b. First, suppose that (u, v) is a back edge. A self-loop is by definition a back edge. If (u, v) is a self-loop, then clearly $d[v] = d[u] < f[u] = f[v]$. If (u, v) is not a self-loop, then u is a descendant of v in the depth-first forest, and by Corollary 22.8, $d[v] < d[u] < f[u] < f[v]$.

Now, suppose that $d[v] \le d[u] < f[u] \le f[v]$. If u and v are the same vertex, then $d[v] = d[u] < f[u] = f[v]$, and (u, v) is a self-loop and hence a back edge. If u and v are distinct, then $d[v] < d[u] < f[u] < f[v]$. By Theorem 22.7, interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree. Thus, (u, v) is a back edge.

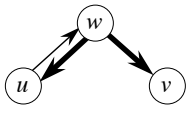
c. First, suppose that (u, v) is a cross edge. Since neither u nor v is an ancestor of the other, Theorem 22.7 says that the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint. Thus, we must have either $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$. We claim that we cannot have $d[u] < d[v]$ if (u, v) is a cross edge. Why? If $d[u] < d[v]$, then v is white at time $d[u]$. By Theorem 22.9, v is a descendant of u, which contradicts (u, v) being a cross edge. Thus, we must have $d[v] < f[v] < d[u] < f[u]$.

Now suppose that $d[v] < f[v] < d[u] < f[u]$. By Theorem 22.7, neither u nor v is a descendant of the other, which means that (u, v) must be a cross edge.

Solution to Exercise 22.3-7

Let us consider the example graph and depth-first search below.

	d	f
w	1	6
u	2	3
v	4	5

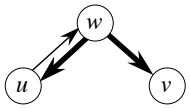


Clearly, there is a path from u to v in G. The bold edges are in the depth-first forest produced. We can see that $d[u] < d[v]$ in the depth-first search but v is not a descendant of u in the forest.

Solution to Exercise 22.3-8

Let us consider the example graph and depth-first search below.

	d	f
w	1	6
u	2	3
v	4	5

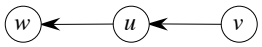


Clearly, there is a path from u to v in G. The bold edges of G are in the depth-first forest produced by the search. However, $d[v] > f[u]$ and the conjecture is false.

Solution to Exercise 22.3-10

Let us consider the example graph and depth-first search below.

	d	f
w	1	2
u	3	4
v	5	6



Clearly u has both incoming and outgoing edges in G but a depth-first search of G produced a depth-first forest where u is in a tree by itself.

Solution to Exercise 22.3-11

Compare the following pseudocode to the pseudocode of DFS on page 541 of the book. Changes were made in order to assign the desired *cc* label to vertices.

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
counter ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then counter ← counter + 1
               DFS-VISIT(u, counter)
```

```
DFS-VISIT(u, counter)
color[u] ← GRAY
cc[u] ← counter        ▷ Label the vertex.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v, counter)
color[u] ← BLACK
f[u] ← time ← time + 1
```

This DFS increments a counter each time DFS-VISIT is called to grow a new tree in the DFS forest. Every vertex visited (and added to the tree) by DFS-VISIT is labeled with that same counter value. Thus $cc[u] = cc[v]$ if and only if u and v are visited in the same call to DFS-VISIT from DFS, and the final value of the counter is the number of calls that were made to DFS-VISIT by DFS. Also, since every vertex is visited eventually, every vertex is labeled.

Thus all we need to show is that the vertices visited by each call to DFS-VISIT from DFS are exactly the vertices in one connected component of G.

- All vertices in a connected component are visited by one call to DFS-VISIT from DFS:

  Let u be the first vertex in component C visited by DFS-VISIT. Since a vertex becomes non-white only when it is visited, all vertices in C are white when DFS-VISIT is called for u. Thus, by the white-path theorem, all vertices in C become descendants of u in the forest, which means that all vertices in C are visited (by recursive calls to DFS-VISIT) before DFS-VISIT returns to DFS.
- All vertices visited by one call to DFS-VISIT from DFS are in the same connected component:

  If two vertices are visited in the same call to DFS-VISIT from DFS, they in the same connected component, because vertices are visited only by following paths in G (by following edges found in adjacency lists, starting from some vertex).

Solution to Exercise 22.4-3

An undirected graph is acyclic (i.e., a forest) if and only if a DFS yields no back edges.

- If there's a back edge, there's a cycle.
- If there's no back edge, then by Theorem 22.10, there are only tree edges. Hence, the graph is acyclic.

Thus, we can run DFS: if we find a back edge, there's a cycle.

- Time: $O(V)$. (Not $O(V + E)$!)
  If we ever see $|V|$ distinct edges, we must have seen a back edge because (by Theorem B.2 on p. 1085) in an acyclic (undirected) forest, $|E| \leq |V| - 1$.

Solution to Exercise 22.4-5

```
TOPOLOGICAL-SORT(G)
▷ Initialize in-degree, Θ(V) time
for each vertex u ∈ V
    do in-degree[u] ← 0
▷ Compute in-degree, Θ(V + E) time
for each vertex u ∈ V
    do for each v ∈ Adj[u]
          do in-degree[v] ← in-degree[v] + 1
▷ Initialize Queue, Θ(V) time
Q ← ∅
for each vertex u ∈ V
    do if in-degree[u] = 0
          then ENQUEUE(Q, u)
▷ while loop takes O(V + E) time
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       output u
       ▷ for loop executes O(E) times total
       for each v ∈ Adj[u]
           do in-degree[v] ← in-degree[v] − 1
              if in-degree[v] = 0
                 then ENQUEUE(Q, v)
▷ Check for cycles, O(V) time
for each vertex u ∈ V
    do if in-degree[u] ≠ 0
          then report that there's a cycle
▷ Another way to check for cycles would be to count the vertices
▷ that are output and report a cycle if that number is < |V|.
```

To find and output vertices of in-degree 0, we first compute all vertices' in-degrees by making a pass through all the edges (by scanning the adjacency lists of all the vertices) and incrementing the in-degree of each vertex an edge enters.

- This takes $\Theta(V + E)$ time ($|V|$ adjacency lists accessed, $|E|$ edges total found in those lists, $\Theta(1)$ work for each edge).

We keep the vertices with in-degree 0 in a FIFO queue, so that they can be enqueued and dequeued in $O(1)$ time. (The order in which vertices in the queue are processed doesn't matter, so any kind of queue works.)

- Initializing the queue takes one pass over the vertices doing $\Theta(1)$ work, for total time $\Theta(V)$.

As we process each vertex from the queue, we effectively remove its outgoing edges from the graph by decrementing the in-degree of each vertex one of those edges enters, and we enqueue any vertex whose in-degree goes to 0. There's

no need to actually remove the edges from the adjacency list, because that adjacency list will never be processed again by the algorithm: Each vertex is enqueued/dequeued at most once because it is enqueued only if it starts out with in-degree 0 or if its in-degree becomes 0 after being decremented (and never incremented) some number of times.

- The processing of a vertex from the queue happens $O(V)$ times because no vertex can be enqueued more than once. The per-vertex work (dequeue and output) takes $O(1)$ time, for a total of $O(V)$ time.
- Because the adjacency list of each vertex is scanned only when the vertex is dequeued, the adjacency list of each vertex is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, at most $O(E)$ time is spent in total scanning adjacency lists. For each edge in an adjacency list, $\Theta(1)$ work is done, for a total of $O(E)$ time.

Thus the total time taken by the algorithm is $O(V + E)$.

The algorithm outputs vertices in the right order (u before v for every edge (u, v)) because v will not be output until its in-degree becomes 0, which happens only when every edge (u, v) leading into v has been "removed" due to the processing (including output) of u.

If there are no cycles, all vertices are output.

- Proof: Assume that some vertex v_0 is not output. v_0 cannot start out with in-degree 0 (or it would be output), so there are edges into v_0. Since v_0's in-degree never becomes 0, at least one edge (v_1, v_0) is never removed, which means that at least one other vertex v_1 was not output. Similarly, v_1 not output means that some vertex v_2 such that $(v_2, v_1) \in E$ was not output, and so on. Since the number of vertices is finite, this path $(\cdots \to v_2 \to v_1 \to v_0)$ is finite, so we must have $v_i = v_j$ for some i and j in this sequence, which means there is a cycle.

If there are cycles, not all vertices will be output, because some in-degrees never become 0.

- Proof: Assume that a vertex in a cycle is output (its in-degree becomes 0). Let v be the first vertex in its cycle to be output, and let u be v's predecessor in the cycle. In order for v's in-degree to become 0, the edge (u, v) must have been "removed," which happens only when u is processed. But this cannot have happened, because v is the first vertex in its cycle to be processed. Thus no vertices in cycles are output.

Solution to Exercise 22.5-5

We have at our disposal an $O(V + E)$-time algorithm that computes strongly connected components. Let us assume that the output of this algorithm is a mapping $scc[u]$, giving the number of the strongly connected component containing vertex u, for each vertex u. Without loss of generality, assume that $scc[u]$ is an integer in the set $\{1, 2, \ldots, |V|\}$.

Construct the multiset (a set that can contain the same object more than once) $T = \{scc[u] : u \in V\}$, and sort it by using counting sort. Since the values we are sorting are integers in the range 1 to $|V|$, the time to sort is $O(V)$. Go through the sorted multiset T and every time we find an element x that is distinct from the one before it, add x to V^{SCC}. (Consider the first element of the sorted set as "distinct from the one before it.") It takes $O(V)$ time to construct V^{SCC}.

Construct the set of ordered pairs

$$S = \{(x, y) : \text{there is an edge } (u, v) \in E, x = scc[u], \text{ and } y = scc[v]\} .$$

We can easily construct this set in $\Theta(E)$ time by going through all edges in E and looking up $scc[u]$ and $scc[v]$ for each edge $(u, v) \in E$.

Having constructed S, remove all elements of the form (x, x). Alternatively, when we construct S, do not put an element in S when we find an edge (u, v) for which $scc[u] = scc[v]$. S now has at most $|E|$ elements.

Now sort the elements of S using radix sort. Sort on one component at a time. The order does not matter. In other words, we are performing two passes of counting sort. The time to do so is $O(V + E)$, since the values we are sorting on are integers in the range 1 to $|V|$.

Finally, go through the sorted set S, and every time we find an element (x, y) that is distinct from the element before it (again considering the first element of the sorted set as distinct from the one before it), add (x, y) to E^{SCC}. Sorting and then adding (x, y) only if it is distinct from the element before it ensures that we add (x, y) at most once. It takes $O(E)$ time to go through S in this way, once S has been sorted.

The total time is $O(V + E)$.

Solution to Exercise 22.5-6

The basic idea is to replace the edges within each SCC by one simple, directed cycle and then remove redundant edges between SCC's. Since there must be at least k edges within an SCC that has k vertices, a single directed cycle of k edges gives the k-vertex SCC with the fewest possible edges.

The algorithm works as follows:

1. Identify all SCC's of G. Time: $\Theta(V + E)$, using the SCC algorithm in Section 22.5.
2. Form the component graph G^{SCC}. Time: $O(V + E)$, by Exercise 22.5-5.
3. Start with $E' = \emptyset$. Time: $O(1)$.
4. For each SCC of G, let the vertices in the SCC be $v_1, v_2, \ldots, v_k$, and add to E' the directed edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)$. These edges form a simple, directed cycle that includes all vertices of the SCC. Time for all SCC's: $O(V)$.
5. For each edge (u, v) in the component graph G^{SCC}, select any vertex x in u's SCC and any vertex y in v's SCC, and add the directed edge (x, y) to E'. Time: $O(E)$.

Thus, the total time is $\Theta(V + E)$.

Solution to Exercise 22.5-7

To determine if $G = (V, E)$ is semiconnected, do the following:

1. Call STRONGLY-CONNECTED-COMPONENTS.
2. Form the component graph. (By Exercise 22.5-5, you may assume that this takes $O(V + E)$ time.)
3. Topologically sort the component graph. (Recall that it's a dag.) Assuming that there are k SCC's, the topological sort gives a linear ordering $\langle v_1, v_2, \ldots, v_k \rangle$ of the vertices.
4. Verify that the sequence of vertices $\langle v_1, v_2, \ldots, v_k \rangle$ given by topological sort forms a linear chain in the component graph. That is, verify that the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ exist in the component graph. If the vertices form a linear chain, then the original graph is semiconnected; otherwise it is not.

Because we know that all vertices in each SCC are mutually reachable from each other, it suffices to show that the component graph is semiconnected if and only if it contains a linear chain. We must also show that if there's a linear chain in the component graph, it's the one returned by topological sort.

We'll first show that if there's a linear chain in the component graph, then it's the one returned by topological sort. In fact, this is trivial. A topological sort has to respect every edge in the graph. So if there's a linear chain, a topological sort *must* give us the vertices in order.

Now we'll show that the component graph is semiconnected if and only if it contains a linear chain.

First, suppose that the component graph contains a linear chain. Then for every pair of vertices u, v in the component graph, there is a path between them. If u precedes v in the linear chain, then there's a path $u \rightsquigarrow v$. Otherwise, v precedes u, and there's a path $v \rightsquigarrow u$.

Conversely, suppose that the component graph does not contain a linear chain. Then in the list returned by topological sort, there are two consecutive vertices v_i and v_{i+1}, but the edge (v_i, v_{i+1}) is not in the component graph. Any edges out of v_i are to vertices v_j, where $j > i + 1$, and so there is no path from v_i to v_{i+1} in the component graph. And since v_{i+1} follows v_i in the topological sort, there cannot be any paths at all from v_{i+1} to v_i. Thus, the component graph is not semiconnected.

Running time of each step:

1. $\Theta(V + E)$.
2. $O(V + E)$.
3. Since the component graph has at most $|V|$ vertices and at most $|E|$ edges, $O(V + E)$.
4. Also $O(V + E)$. We just check the adjacency list of each vertex v_i in the component graph to verify that there's an edge (v_i, v_{i+1}). We'll go through each adjacency list once.

Thus, the total running time is $\Theta(V + E)$.

Solution to Problem 22-1

a. 1. Suppose (u, v) is a back edge or a forward edge in a BFS of an undirected graph. Then one of u and v, say u, is a proper ancestor of the other (v) in the breadth-first tree. Since we explore all edges of u before exploring any edges of any of u's descendants, we must explore the edge (u, v) at the time we explore u. But then (u, v) must be a tree edge.

2. In BFS, an edge (u, v) is a tree edge when we set $\pi[v] \leftarrow u$. But we only do so when we set $d[v] \leftarrow d[u] + 1$. Since neither $d[u]$ nor $d[v]$ ever changes thereafter, we have $d[v] = d[u] + 1$ when BFS completes.

3. Consider a cross edge (u, v) where, without loss of generality, u is visited before v. At the time we visit u, vertex v must already be on the queue, for otherwise (u, v) would be a tree edge. Because v is on the queue, we have $d[v] \leq d[u] + 1$ by Lemma 22.3. By Corollary 22.4, we have $d[v] \geq d[u]$. Thus, either $d[v] = d[u]$ or $d[v] = d[u] + 1$.

b. 1. Suppose (u, v) is a forward edge. Then we would have explored it while visiting u, and it would have been a tree edge.

2. Same as for undirected graphs.

3. For any edge (u, v), whether or not it's a cross edge, we cannot have $d[v] > d[u] + 1$, since we visit v at the latest when we explore edge (u, v). Thus, $d[v] \leq d[u] + 1$.

4. Clearly, $d[v] \geq 0$ for all vertices v. For a back edge (u, v), v is an ancestor of u in the breadth-first tree, which means that $d[v] \leq d[u]$. (Note that since self-loops are considered to be back edges, we could have $u = v$.)

Solution to Problem 22-3

a. An Euler tour is a single cycle that traverses each edge of G exactly once, but it might not be a simple cycle. An Euler tour can be decomposed into a set of edge-disjoint simple cycles, however.

If G has an Euler tour, therefore, we can look at the simple cycles that, together, form the tour. In each simple cycle, each vertex in the cycle has one entering edge and one leaving edge. In each simple cycle, therefore, each vertex v has in-degree(v) = out-degree(v), where the degrees are either 1 (if v is on the simple cycle) or 0 (if v is not on the simple cycle). Adding the in- and out-degrees over all edges proves that if G has an Euler tour, then in-degree(v) = out-degree(v) for all vertices v.

We prove the converse—that if in-degree(v) = out-degree(v) for all vertices v, then G has an Euler tour—in two different ways. One proof is nonconstructive, and the other proof will help us design the algorithm for part (b).

First, we claim that if in-degree(v) = out-degree(v) for all vertices v, then we can pick any vertex u for which in-degree(u) = out-degree$(u) \geq 1$ and create a cycle (not necessarily simple) that contains u. To prove this claim, let us start

by placing vertex u on the cycle, and choose any leaving edge of u, say (u, v). Now we put v on the cycle. Since in-degree(v) = out-degree$(v) \geq 1$, we can pick some leaving edge of v and continue visiting edges and vertices. Each time we pick an edge, we can remove it from further consideration. At each vertex other than u, at the time we visit an entering edge, there must be an unvisited leaving edge, since in-degree(v) = out-degree(v) for all vertices v. The only vertex for which there might not be an unvisited leaving edge is u, since we started the cycle by visiting one of u's leaving edges. Since there's always a leaving edge we can visit from all vertices other than u, eventually the cycle must return to u, thus proving the claim.

The nonconstructive proof proves the contrapositive—that if G does not have an Euler tour, then in-degree$(v) \neq$ out-degree(v) for some vertex v—by contradiction. Choose a graph $G = (V, E)$ that does not have an Euler tour but has at least one edge and for which in-degree(v) = out-degree(v) for all vertices v, and let G have the fewest edges of any such graph. By the above claim, G contains a cycle. Let C be a cycle of G with the greatest number of edges, and let V_C be the set of vertices visited by cycle C. By our assumption, C is not an Euler tour, and so the set of edges $E' = E - C$ is nonempty. If we use the set V of vertices and the set E' of edges, we get the graph $G' = (V, E')$; this graph has in-degree(v) = out-degree(v) for all vertices v, since we have removed one entering edge and one leaving edge for each vertex on cycle C. Consider any component $G'' = (V'', E'')$ of G', and observe that G'' also has in-degree(v) = out-degree(v) for all vertices v. Since $E'' \subseteq E' \subsetneq E$, it follows from how we chose G that G'' must have an Euler tour, say C'. Because the original graph G is connected, there must be some vertex $x \in V'' \cup V_C$ and, without loss of generality, consider x to be the first and last vertex on both C and C'. But then the cycle C'' formed by first traversing C and then traversing C' is a cycle of G with more edges than C, contradicting our choice of C. We conclude that C must have been an Euler tour.

The constructive proof uses the same ideas. Let us start at a vertex u and, via random traversal of edges, create a cycle. We know that once we take any edge entering a vertex $v \neq u$, we can find an edge leaving v that we have not yet taken. Eventually, we get back to vertex u, and if there are still edges leaving u that we have not taken, we can continue the cycle. Eventually, we get back to vertex u and there are no untaken edges leaving u. If we have visited every edge in the graph G, we are done. Otherwise, since G is connected, there must be some unvisited edge leaving a vertex, say v, on the cycle. We can traverse a new cycle starting at v, visiting only previously unvisited edges, and we can splice this cycle into the cycle we already know. That is, if the original cycle is $\langle u, \ldots, v, w, \ldots, u\rangle$, and the new cycle is $\langle v, x, \ldots, v\rangle$, then we can create the cycle $\langle u, \ldots, v, x, \ldots, v, w, \ldots, u\rangle$. We continue this process of finding a vertex with an unvisited leaving edge on a visited cycle, visiting a cycle starting and ending at this vertex, and splicing in the newly visited cycle, until we have visited every edge.

b. The algorithm is based on the idea in the constructive proof above.

We assume that G is represented by adjacency lists, and we work with a copy of the adjacency lists, so that as we visit each edge, we can remove it from

its adjacency list. The singly linked form of adjacency list will suffice. The output of this algorithm is a doubly linked list T of vertices which, read in list order, will give an Euler tour. The algorithm constructs T by finding cycles (also represented by doubly linked lists) and splicing them into T. By using doubly linked lists for cycles and the Euler tour, splicing a cycle into the Euler tour takes constant time.

We also maintain a singly linked list L in which each list element consists of two parts:

1. a vertex v, and
2. a pointer to some appearance of v in T.

Initially, L contains one vertex, which may be any vertex of G.

Here is the algorithm:

```
EULER-TOUR(G)
T ← empty list
L ← (any vertex v, NIL)
while L is not empty
    do remove (v, location-in-T) from L
       C ← VISIT(v)
       if location-in-T = NIL
         then T ← C
         else splice C into T just before location-in-T
return T

VISIT(v)
C ← empty sequence of vertices
u ← v
while out-degree(u) > 0
    do let w be the first vertex in Adj[u]
       remove w from Adj[u], decrementing out-degree(u)
       add u onto the end of C
       if out-degree(u) > 0
         then add (u, u's location in C) to L
       u ← w
return C
```

The use of NIL in the initial assignment to L ensures that the first cycle C returned by VISIT becomes the current version of the Euler tour T. All cycles returned by VISIT thereafter are spliced into T. We assume that whenever an empty cycle is returned by VISIT, splicing it into T leaves T unchanged.

Each time EULER-TOUR removes a vertex v from the list L, it calls VISIT(v) to find a cycle C, possibly empty and possibly not simple, that starts and ends at v; the cycle C is represented by a list that starts with v and ends with the last vertex on the cycle before the cycle ends at v. EULER-TOUR then splices this cycle C into the Euler tour T just before some appearance of v in T.

When VISIT is at a vertex u, it looks for some vertex w such that the edge (u, w) has not yet been visited. Removing w from $Adj[u]$ ensures that we will never

visit (u, w) again. VISIT adds u onto the cycle C that it constructs. If, after removing edge (u, w), vertex u still has any leaving edges, then u, along with its location in C, is added to L. The cycle construction continues from w, and it ceases once a vertex with no unvisited leaving edges is found. Using the argument from part (a), at that point, this vertex must close up a cycle. At that point, therefore, the cycle C is returned.

It is possible that a vertex u has unvisited leaving edges at the time it is added to list L in VISIT, but that by the time that u is removed from L in EULER-TOUR, all of its leaving edges have been visited. In this case, the **while** loop of VISIT executes 0 iterations, and VISIT returns an empty cycle.

Once the list L is empty, every edge has been visited. The resulting cycle T is then an Euler tour.

To see that EULER-TOUR takes $O(E)$ time, observe that because we remove each edge from its adjacency list as it is visited, no edge is visited more than once. Since each edge is visited at some time, the number of times that a vertex is added to L, and thus removed from L, is at most $|E|$. Thus, the **while** loop in EULER-TOUR executes at most E iterations. The **while** loop in VISIT executes one iteration per edge in the graph, and so it executes at most E iterations as well. Since adding vertex u to the doubly linked list C takes constant time and splicing C into T takes constant time, the entire algorithm takes $O(E)$ time.

Here is a variation on EULER-TOUR, which may be a bit simpler to reason about. It maintains a pointer u to a vertex on the Euler tour, with the invariant that all vertices on the Euler tour behind u have already had all entering and leaving edges added to the tour. This variation calls the same procedure VISIT as above.

```
EULER-TOUR′(G)
v ← any vertex
T ← VISIT(v)
mark v's position as the starting vertex in T
u ← next[v]
while u's position in T ≠ v's position in T
    do C ← VISIT(u)
       splice C into T, just before u's position
       ▷ If C was empty, T has not changed.
       ▷ If C was nonempty, then it began with u
       u ← next[next[prev[u]]]
       ▷ If C was empty, u now points to the next vertex on T
       ▷ If C was nonempty, u now points to the next vertex on C
                (which has been spliced into T)
return T
```

Whenever we return from calling VISIT(u), we know that out-degree$(u) = 0$, which means that we have visited all edges entering or leaving vertex u. Since VISIT adds each edge it visits to the cycle C, which is then added to the Euler tour T, when we return from a call to VISIT(u), all edges entering or leaving vertex u have been added to the tour. When we advance the pointer u in the

while loop, we need to ensure that it is advanced according to the current tour T, which may have just had a cycle C spliced into it. That's why we advance u by the expression $next[next[prev[u]]]$, rather than just simply $next[u]$.

Since the graph G is connected, every edge will eventually be visited and added to the tour T. As before, each edge is visited exactly once, so that at completion, T will consist of exactly $|E|$ edges. Once a vertex u has had VISIT called on it, any future call of VISIT(u) will take $O(1)$ time, and so the total time for all calls to VISIT is $O(E)$.

Solution to Problem 22-4

Compute G^{T} in the usual way, so that G^{T} is G with its edges reversed. Then do a depth-first search on G^{T}, but in the main loop of DFS, consider the vertices in order of increasing values of $L(v)$. If vertex u is in the depth-first tree with root v, then $\min(u) = v$. Clearly, this algorithm takes $O(V + E)$ time.

To show correctness, first note that if u is in the depth-first tree rooted at v in G^{T}, then there is a path $v \rightsquigarrow u$ in G^{T}, and so there is a path $u \rightsquigarrow v$ in G. Thus, the minimum vertex label of all vertices reachable from u is at most $L(v)$, or in other words, $L(v) \geq \min\{L(w) : w \in R(u)\}$.

Now suppose that $L(v) > \min\{L(w) : w \in R(u)\}$, so that there is a vertex $w \in R(u)$ such that $L(w) < L(v)$. At the time $d[v]$ that we started the depth-first search from v, we would have already discovered w, so that $d[w] < d[v]$. By the parenthesis theorem, either the intervals $[d[v], f[v]]$, and $[d[w], f[w]]$ are disjoint and neither v nor w is a descendant of the other, or we have the ordering $d[w] < d[v] < f[v] < f[w]$ and v is a descendant of w. The latter case cannot occur, since v is a root in the depth-first forest (which means that v cannot be a descendant of any other vertex). In the former case, since $d[w] < d[v]$, we must have $d[w] < f[w] < d[v] < f[v]$. In this case, since u is reachable from w in G^{T}, we would have discovered u by the time $f[w]$, so that $d[u] < f[w]$. Since we discovered u during a search that started at v, we have $d[v] \leq d[u]$. Thus, $d[v] \leq d[u] < f[w] < d[v]$, which is a contradiction. We conclude that no such vertex w can exist.

Lecture Notes for Chapter 23: Minimum Spanning Trees

Chapter 23 overview

Problem

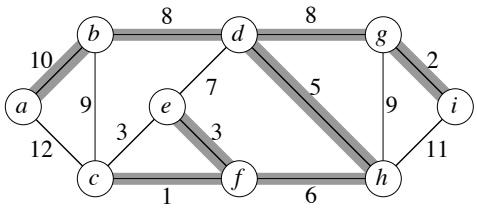
- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that
  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

Model as a graph:

- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that
  1. T connects all vertices (T is a ***spanning tree***), and
  2. $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.

A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.

Example of such a graph *[edges in MST are shaded]* :



In this example, there is more than one MST. Replace edge (e, f) by (c, e). Get a different spanning tree with the same weight.

Growing a minimum spanning tree

Some properties of an MST:

- It has $|V| - 1$ edges.
- It has no cycles.
- It might not be unique.

Building up the solution

- We will build a set A of edges.
- Initially, A has no edges.
- As we add edges to A, maintain a loop invariant:

  Loop invariant: A is a subset of some MST.

- Add only edges that maintain the invariant. If A is a subset of some MST, an edge (u, v) is ***safe*** for A if and only if $A \cup \{(u, v)\}$ is also a subset of some MST. So we will add only safe edges.

Generic MST algorithm

GENERIC-MST(G, w)
$A \leftarrow \emptyset$
while A is not a spanning tree
  do find an edge (u, v) that is safe for A
    $A \leftarrow A \cup \{(u, v)\}$
return A

Use the loop invariant to show that this generic algorithm works.

Initialization: The empty set trivially satisfies the loop invariant.

Maintenance: Since we add only safe edges, A remains a subset of some MST.

Termination: All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST.

Finding a safe edge

How do we find safe edges?

Let's look at the example. Edge (c, f) has the lowest weight of any edge in the graph. Is it safe for $A = \emptyset$?

Intuitively: Let $S \subset V$ be any set of vertices that includes c but not f (so that f is in $V - S$). In any MST, there has to be one edge (at least) that connects S with $V - S$. Why not choose the edge with minimum weight? (Which would be (c, f) in this case.)

Some definitions: Let $S \subset V$ and $A \subseteq E$.

- A ***cut*** $(S, V - S)$ is a partition of vertices into disjoint sets V and $S - V$.
- Edge $(u, v) \in E$ ***crosses*** cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
- A cut ***respects*** A if and only if no edge in A crosses the cut.
- An edge is a ***light edge*** crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be > 1 light edge crossing it.

Theorem

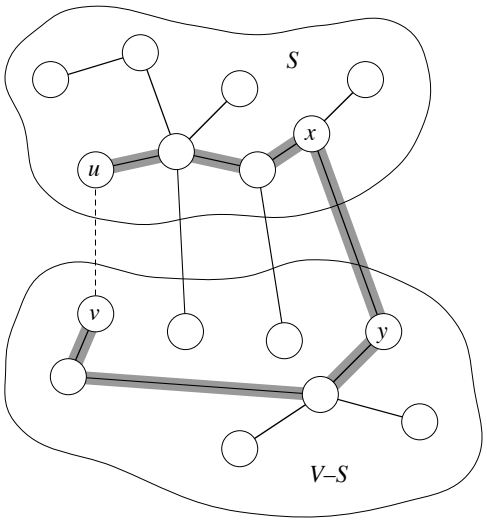
Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Proof Let T be an MST that includes A.

If T contains (u, v), done.

So now assume that T does not contain (u, v). We'll construct a different MST T' that includes $A \cup \{(u, v)\}$.

Recall: a tree has unique path between each pair of vertices. Since T is an MST, it contains a unique path p between u and v. Path p must cross the cut $(S, V - S)$ at least once. Let (x, y) be an edge of p that crosses the cut. From how we chose (u, v), must have $w(u, v) \leq w(x, y)$.



[Except for the dashed edge (u, v), all edges shown are in T. A is some subset of the edges of T, but A cannot contain any edges that cross the cut $(S, V - S)$, since this cut respects A. Shaded edges are the path p.]

Since the cut respects A, edge (x, y) is not in A.

To form T' from T:

- Remove (x, y). Breaks T into two components.
- Add (u, v). Reconnects.

So $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

T' is a spanning tree.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) , \end{aligned}$$

since $w(u, v) \leq w(x, y)$. Since T' is a spanning tree, $w(T') \leq w(T)$, and T is an MST, then T' must be an MST.

Need to show that (u, v) is safe for A:

- $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$.
- $A \cup \{(u, v)\} \subseteq T'$.
- Since T' is an MST, (u, v) is safe for A. ■ (theorem)

So, in GENERIC-MST:

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the **for** loop iterates $|V| - 1$ times. Equivalently, after adding $|V|-1$ safe edges, we're down to just one component.

Corollary
If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component in G_A (i.e., (u, v) is a light edge crossing the cut $(V_C, V - V_C)$), then (u, v) is safe for A.

Proof Set $S = V_C$ in the theorem. ■ (corollary)

This naturally leads to the algorithm called Kruskal's algorithm to solve the minimum-spanning-tree problem.

Kruskal's algorithm

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \rightarrow \mathbf{R}$.

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

```
KRUSKAL(V, E, w)
A ← ∅
for each vertex v ∈ V
    do MAKE-SET(v)
sort E into nondecreasing order by weight w
for each (u, v) taken from the sorted list
    do if FIND-SET(u) ≠ FIND-SET(v)
          then A ← A ∪ {(u, v)}
               UNION(u, v)
return A
```

Run through the above example to see how Kruskal's algorithm works on it:

(c, f) : safe
(g, i) : safe
(e, f) : safe
(c, e) : reject
(d, h) : safe
(f, h) : safe
(e, d) : reject
(b, d) : safe
(d, g) : safe
(b, c) : reject
(g, h) : reject
(a, b) : safe

At this point, we have only one component, so all other edges will be rejected. *[We could add a test to the main loop of* KRUSKAL *to stop once* $|V| - 1$ *edges have been added to A.]*

Get the shaded edges shown in the figure.

Suppose we had examined (c, e) *before* (e, f). Then would have found (c, e) safe and would have rejected (e, f).

Analysis

Initialize A:	$O(1)$
First **for** loop:	$\|V\|$ MAKE-SETs
Sort E:	$O(E \lg E)$
Second **for** loop:	$O(E)$ FIND-SETs and UNIONs

- Assuming the implementation of disjoint-set data structure, already seen in Chapter 21, that uses union by rank and path compression:

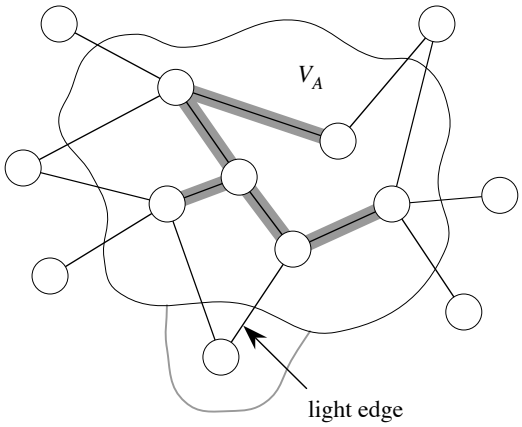
  $$O((V + E)\,\alpha(V)) + O(E \lg E)\ .$$

- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|) = O(\lg V) = O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.
- $|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.

- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

Prim's algorithm

- Builds one tree, so A is always a tree.
- Starts from an arbitrary "root" r.
- At each step, find a light edge crossing cut $(V_A, V - V_A)$, where V_A = vertices that A is incident on. Add this edge to A.



[Edges of A are shaded.]

How to find the light edge quickly?

Use a priority queue Q:

- Each object is a vertex in $V - V_A$.
- Key of v is minimum weight of any edge (u, v), where $u \in V_A$.
- Then the vertex returned by EXTRACT-MIN is v such that there exists $u \in V_A$ and (u, v) is light edge crossing $(V_A, V - V_A)$.
- Key of v is ∞ if v is not adjacent to any vertices in V_A.

The edges of A will form a rooted tree with root r:

- r is given as an input to the algorithm, but it can be any vertex.
- Each vertex knows its parent in the tree by the attribute $\pi[v]$ = parent of v. $\pi[v] =$ NIL if $v = r$ or v has no parent.
- As algorithm progresses, $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
- At termination, $V_A = V \Rightarrow Q = \emptyset$, so MST is $A = \{(v, \pi[v]) : v \in V - \{r\}\}$.

```
PRIM(V, E, w, r)
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
       π[u] ← NIL
       INSERT(Q, u)
DECREASE-KEY(Q, r, 0)      ▷ key[r] ← 0
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
                 then π[v] ← u
                      DECREASE-KEY(Q, v, w(u, v))
```

Do example from previous graph. *[Let a student pick the root.]*

Analysis

Depends on how the priority queue is implemented:

- Suppose Q is a binary heap.

Initialize Q and first **for** loop:	$O(V \lg V)$
Decrease key of r:	$O(\lg V)$
while loop:	$\lvert V\rvert$ EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
	$\leq \lvert E\rvert$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$
Total:	$O(E \lg V)$

- Suppose we could do DECREASE-KEY in $O(1)$ *amortized* time.

  Then $\leq \lvert E\rvert$ DECREASE-KEY calls take $O(E)$ time altogether $\Rightarrow$ total time becomes $O(V \lg V + E)$.

  In fact, there is a way to do DECREASE-KEY in $O(1)$ amortized time: Fibonacci heaps, in Chapter 20.

Solutions for Chapter 23: Minimum Spanning Trees

Solution to Exercise 23.1-1

Theorem 23.1 shows this.

Let A be the empty set and S be any set containing u but not v.

Solution to Exercise 23.1-4

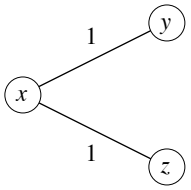
A triangle whose edge weights are all equal is a graph in which every edge is a light edge crossing some cut. But the triangle is cyclic, so it is not a minimum spanning tree.

Solution to Exercise 23.1-6

Suppose that for every cut of G, there is a unique light edge crossing the cut. Let us consider two minimum spanning trees, T and T', of G. We will show that every edge of T is also in T', which means that T and T' are the same tree and hence there is a unique minimum spanning tree.

Consider any edge $(u, v) \in T$. If we remove (u, v) from T, then T becomes disconnected, resulting in a cut $(S, V - S)$. The edge (u, v) is a light edge crossing the cut $(S, V - S)$ (by Exercise 23.1-3). Now consider the edge $(x, y) \in T'$ that crosses $(S, V - S)$. It, too, is a light edge crossing this cut. Since the light edge crossing $(S, V - S)$ is unique, the edges (u, v) and (x, y) are the same edge. Thus, $(u, v) \in T'$. Since we chose (u, v) arbitrarily, every edge in T is also in T'.

Here's a counterexample for the converse:



Here, the graph is its own minimum spanning tree, and so the minimum spanning tree is unique. Consider the cut $(\{x\}, \{y, z\})$. Both of the edges (x, y) and (x, z) are light edges crossing the cut, and they are both light edges.

Solution to Exercise 23.1-10

Let $w(T) = \sum_{(x,y)\in T} w(x, y)$. We have $w'(T) = w(T) - k$. Consider any other spanning tree T', so that $w(T) \le w(T')$.

If $(x, y) \notin T'$, then $w'(T') = w(T') \ge w(T) > w'(T)$.

If $(x, y) \in T'$, then $w'(T') = w(T') - k \ge w(T) - k = w'(T)$.

Either way, $w'(T) \le w'(T')$, and so T is a minimum spanning tree for weight function w'.

Solution to Exercise 23.2-4

We know that Kruskal's algorithm takes $O(V)$ time for initialization, $O(E \lg E)$ time to sort the edges, and $O(E\,\alpha(V))$ time for the disjoint-set operations, for a total running time of $O(V + E \lg E + E\,\alpha(V)) = O(E \lg E)$.

If we knew that all of the edge weights in the graph were integers in the range from 1 to $|V|$, then we could sort the edges in $O(V + E)$ time using counting sort. Since the graph is connected, $V = O(E)$, and so the sorting time is reduced to $O(E)$. This would yield a total running time of $O(V + E + E\,\alpha(V)) = O(E\,\alpha(V))$, again since $V = O(E)$, and since $E = O(E\,\alpha(V))$. The time to process the edges, not the time to sort them, is now the dominant term. Knowledge about the weights won't help speed up any other part of the algorithm, since nothing besides the sort uses the weight values.

If the edge weights were integers in the range from 1 to W for some constant W, then we could again use counting sort to sort the edges more quickly. This time, sorting would take $O(E + W) = O(E)$ time, since W is a constant. As in the first part, we get a total running time of $O(E\,\alpha(V))$.

Solution to Exercise 23.2-5

The time taken by Prim's algorithm is determined by the speed of the queue operations. With the queue implemented as a Fibonacci heap, it takes $O(E + V \lg V)$ time.

Since the keys in the priority queue are edge weights, it might be possible to implement the queue even more efficiently when there are restrictions on the possible edge weights.

We can improve the running time of Prim's algorithm if W is a constant by implementing the queue as an array $Q[0\,..\,W + 1]$ (using the $W + 1$ slot for key$= \infty$),

where each slot holds a doubly linked list of vertices with that weight as their key. Then EXTRACT-MIN takes only $O(W) = O(1)$ time (just scan for the first nonempty slot), and DECREASE-KEY takes only $O(1)$ time (just remove the vertex from the list it's in and insert it at the front of the list indexed by the new key). This gives a total running time of $O(E)$, which is the best possible asymptotic time (since $\Omega(E)$ edges must be processed).

However, if the range of edge weights is 1 to $|V|$, then EXTRACT-MIN takes $\Theta(V)$ time with this data structure. So the total time spent doing EXTRACT-MIN is $\Theta(V^2)$, slowing the algorithm to $\Theta(E + V^2) = \Theta(V^2)$. In this case, it is better to keep the Fibonacci-heap priority queue, which gave the $\Theta(E + V \lg V)$ time.

There are data structures not in the text that yield better running times:

- The van Emde Boas data structure (mentioned in the chapter notes for Chapter 6 and the introduction to Part V) gives an upper bound of $O(E + V \lg\lg V)$ time for Prim's algorithm.
- A redistributive heap (used in the single-source shortest-paths algorithm of Ahuja, Mehlhorn, Orlin, and Tarjan, and mentioned in the chapter notes for Chapter 24) gives an upper bound of $O\left(E + V\sqrt{\lg V}\right)$ for Prim's algorithm.

Solution to Exercise 23.2-7

We start with the following lemma.

Lemma

Let T be a minimum spanning tree of $G = (V, E)$, and consider a graph $G' = (V', E')$ for which G is a subgraph, i.e., $V \subseteq V'$ and $E \subseteq E'$. Let $\overline{T} = E - T$ be the edges of G that are not in T. Then there is a minimum spanning tree of G' that includes no edges in $\overline{T}$.

Proof By Exercise 23.2-1, there is a way to order the edges of E so that Kruskal's algorithm, when run on G, produces the minimum spanning tree T. We will show that Kruskal's algorithm, run on G', produces a minimum spanning tree T' that includes no edges in $\overline{T}$. We assume that the edges in E are considered in the same relative order when Kruskal's algorithm is run on G and on G'. We first state and prove the following claim.

Claim

For any pair of vertices $u, v \in V$, if these vertices are in the same set after Kruskal's algorithm run on G considers any edge $(x, y) \in E$, then they are in the same set after Kruskal's algorithm run on G' considers (x, y).

Proof of claim Let us order the edges of E by nondecreasing weight as $\langle(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\rangle$, where $k = |E|$. This sequence gives the order in which the edges of E are considered by Kruskal's algorithm, whether it is run on G or on G'. We will use induction, with the inductive hypothesis that if u and v are in the same set after Kruskal's algorithm run on G considers an edge (x_i, y_i), then they are in

the same set after Kruskal's algorithm run on G' considers the same edge. We use induction on i.

Basis: For the basis, $i = 0$. Kruskal's algorithm run on G has not considered any edges, and so all vertices are in different sets. The inductive hypothesis holds trivially.

Inductive step: We assume that any vertices that are in the same set after Kruskal's algorithm run on G has considered edges $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_{i-1}, y_{i-1}) \rangle$ are in the same set after Kruskal's algorithm run on G' has considered the same edges. When Kruskal's algorithm runs on G', after it considers (x_{i-1}, y_{i-1}), it may consider some edges in $E' - E$ before considering (x_i, y_i). The edges in $E' - E$ may cause UNION operations to occur, but sets are never divided. Hence, any vertices that are in the same set after Kruskal's algorithm run on G' considers (x_{i-1}, y_{i-1}) are still in the same set when (x_i, y_i) is considered.

When Kruskal's algorithm run on G considers (x_i, y_i), either x_i and y_i are found to be in the same set or they are not.

- If Kruskal's algorithm run on G finds x_i and y_i to be in the same set, then no UNION operation occurs. The sets of vertices remain the same, and so the inductive hypothesis continues to hold after considering (x_i, y_i).
- If Kruskal's algorithm run on G finds x_i and y_i to be in different sets, then the operation UNION(x_i, y_i) will occur. Kruskal's algorithm run on G' will find that either x_i and y_i are in the same set or they are not. By the inductive hypothesis, when edge (x_i, y_i) is considered, all vertices in x_i's set when Kruskal's algorithm runs on G are in x_i's set when Kruskal's algorithm runs on G', and the same holds for y_i. Regardless of whether Kruskal's algorithm run on G' finds x_i and y_i to already be in the same set, their sets are united after considering (x_i, y_i), and so the inductive hypothesis continues to hold after considering (x_i, y_i). ■ (claim)

With the claim in hand, we suppose that some edge $(u, v) \in \overline{T}$ is placed into T'. That means that Kruskal's algorithm run on G found u and v to be in the same set (since $(u, v) \in \overline{T}$) but Kruskal's algorithm run on G' found u and v to be in different sets (since (u, v) is placed into T'). This fact contradicts the claim, and we conclude that no edge in $\overline{T}$ is placed into T'. Thus, by running Kruskal's algorithm on G and G', we demonstrate that there exists a minimum spanning tree of G' that includes no edges in $\overline{T}$. ■ (lemma)

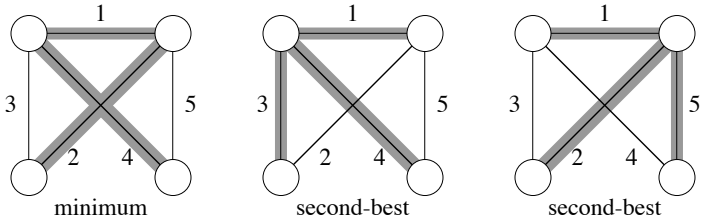
We use this lemma as follows. Let $G' = (V', E')$ be the graph $G = (V, E)$ with the one new vertex and its incident edges added. Suppose that we have a minimum spanning tree T for G. We compute a minimum spanning tree for G' by creating the graph $G'' = (V', E'')$, where E'' consists of the edges of T and the edges in $E' - E$ (i.e., the edges added to G that made G'), and then finding a minimum spanning tree T' for G''. By the lemma, there is a minimum spanning tree for G' that includes no edges of $E - T$. In other words, G' has a minimum spanning tree that includes only edges in T and $E' - E$; these edges comprise exactly the set E''. Thus, the the minimum spanning tree T' of G'' is also a minimum spanning tree of G'.

Even though the proof of the lemma uses Kruskal's algorithm, we are not required to use this algorithm to find T'. We can find a minimum spanning tree by any means we choose. Let us use Prim's algorithm with a Fibonacci-heap priority queue. Since $|V'| = |V| + 1$ and $|E''| \le 2|V| - 1$ (E'' contains the $|V| - 1$ edges of T and at most $|V|$ edges in $E' - E$), it takes $O(V)$ time to construct G'', and the run of Prim's algorithm with a Fibonacci-heap priority queue takes time $O(E'' + V' \lg V') = O(V \lg V)$. Thus, if we are given a minimum spanning tree of G, we can compute a minimum spanning tree of G' in $O(V \lg V)$ time.

Solution to Problem 23-1

a. To see that the minimum spanning tree is unique, observe that since the graph is connected and all edge weights are distinct, then there is a unique light edge crossing every cut. By Exercise 23.1-6, the minimum spanning tree is unique.

To see that the second-best minimum spanning tree need not be unique, here is a weighted, undirected graph with a unique minimum spanning tree of weight 7 and two second-best minimum spanning trees of weight 8:



b. Since any spanning tree has exactly $|V| - 1$ edges, any second-best minimum spanning tree must have at least one edge that is not in the (best) minimum spanning tree. If a second-best minimum spanning tree has exactly one edge, say (x, y), that is not in the minimum spanning tree, then it has the same set of edges as the minimum spanning tree, except that (x, y) replaces some edge, say (u, v), of the minimum spanning tree. In this case, $T' = T - \{(u, v)\} \cup \{(x, y)\}$, as we wished to show.

Thus, all we need to show is that by replacing two or more edges of the minimum spanning tree, we cannot obtain a second-best minimum spanning tree. Let T be the minimum spanning tree of G, and suppose that there exists a second-best minimum spanning tree T' that differs from T by two or more edges. There are at least two edges in $T - T'$, and let (u, v) be the edge in $T - T'$ with minimum weight. If we were to add (u, v) to T', we would get a cycle c. This cycle contains some edge (x, y) in $T' - T$ (since otherwise, T would contain a cycle).

We claim that $w(x, y) > w(u, v)$. We prove this claim by contradiction, so let us assume that $w(x, y) < w(u, v)$. (Recall the assumption that edge weights are distinct, so that we do not have to concern ourselves with $w(x, y) = w(u, v)$.) If we add (x, y) to T, we get a cycle c', which contains

some edge (u', v') in $T - T'$ (since otherwise, T' would contain a cycle). Therefore, the set of edges $T'' = T - \{(u', v')\} \cup \{(x, y)\}$ forms a spanning tree, and we must also have $w(u', v') < w(x, y)$, since otherwise T'' would be a spanning tree with weight less than $w(T)$. Thus, $w(u', v') < w(x, y) < w(u, v)$, which contradicts our choice of (u, v) as the edge in $T - T'$ of minimum weight.

Since the edges (u, v) and (x, y) would be on a common cycle c if we were to add (u, v) to T', the set of edges $T' - \{(x, y)\} \cup \{(u, v)\}$ is a spanning tree, and its weight is less than $w(T')$. Moreover, it differs from T (because it differs from T' by only one edge). Thus, we have formed a spanning tree whose weight is less than $w(T')$ but is not T. Hence, T' was not a second-best minimum spanning tree.

c. We can fill in $max[u, v]$ for all $u, v \in V$ in $O(V^2)$ time by simply doing a search from each vertex u, having restricted the edges visited to those of the spanning tree T. It doesn't matter what kind of search we do: breadth-first, depth-first, or any other kind.

We'll give pseudocode for both breadth-first and depth-first approaches. Each approach differs from the pseudocode given in Chapter 22 in that we don't need to compute d or f values, and we'll use the max table itself to record whether a vertex has been visited in a given search. In particular, $max[u, v] = \text{NIL}$ if and only if $u = v$ or we have not yet visited vertex v in a search from vertex u. Note also that since we're visiting via edges in a spanning tree of an undirected graph, we are guaranteed that the search from each vertex u—whether breadth-first or depth-first—will visit all vertices. There will be no need to "restart" the search as is done in the DFS procedure of Section 22.3. Our pseudocode assumes that the adjacency list of each vertex consists only of edges in the spanning tree T.

Here's the breadth-first search approach:

```
BFS-FILL-MAX(T, w)
for each vertex u ∈ V
    do for each vertex v ∈ V
           do max[u, v] ← NIL
       Q ← ∅
       ENQUEUE(Q, u)
       while Q ≠ ∅
           do x ← DEQUEUE(Q)
              for each v ∈ Adj[x]
                  do if max[u, v] = NIL and v ≠ u
                        then if x = u or w(x, v) > max[u, x]
                                then max[u, v] ← (x, v)
                                else max[u, v] ← max[u, x]
                             ENQUEUE(Q, v)
return max
```

Here's the depth-first search approach:

```
DFS-FILL-MAX(T, w)
for each vertex u ∈ V
    do for each vertex v ∈ V
           do max[u, v] ← NIL
       DFS-FILL-MAX-VISIT(u, u, max)
return max

DFS-FILL-MAX-VISIT(u, x, max)
for each vertex v ∈ Adj[x]
    do if max[u, v] = NIL and v ≠ u
          then if x = u or w(x, v) > max[u, x]
                  then max[u, v] ← (x, v)
                  else  max[u, v] ← max[u, x]
               DFS-FILL-MAX-VISIT(u, v, max)
```

For either approach, we are filling in $|V|$ rows of the *max* table. Since the number of edges in the spanning tree is $|V| - 1$, each row takes $O(V)$ time to fill in. Thus, the total time to fill in the *max* table is $O(V^2)$.

d. In part (b), we established that we can find a second-best minimum spanning tree by replacing just one edge of the minimum spanning tree T by some edge (u, v) not in T. As we know, if we create spanning tree T' by replacing edge $(x, y) \in T$ by edge $(u, v) \notin T$, then $w(T') = w(T) - w(x, y) + w(u, v)$. For a given edge (u, v), the edge $(x, y) \in T$ that minimizes $w(T')$ is the edge of maximum weight on the unique path between u and v in T. If we have already computed the *max* table from part (c) based on T, then the identity of this edge is precisely what is stored in $max[u, v]$. All we have to do is determine an edge $(u, v) \notin T$ for which $w(max[u, v]) - w(u, v)$ is minimum.

Thus, our algorithm to find a second-best minimum spanning tree goes as follows:

1. Compute the minimum spanning tree T. Time: $O(E + V \lg V)$, using Prim's algorithm with a Fibonacci-heap implementation of the priority queue. Since $|E| < |V|^2$, this running time is $O(V^2)$.
2. Given the minimum spanning tree T, compute the *max* table, as in part (c). Time: $O(V^2)$.
3. Find an edge $(u, v) \notin T$ that minimizes $w(max[u, v]) - w(u, v)$. Time: $O(E)$, which is $O(V^2)$.
4. Having found an edge (u, v) in step 3, return $T' = T - \{max[u, v]\} \cup \{(u, v)\}$ as a second-best minimum spanning tree.

The total time is $O(V^2)$.

Lecture Notes for Chapter 24: Single-Source Shortest Paths

Shortest paths

How to find the shortest route between two points on a map.

Input:

- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbf{R}$

Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$= \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

= sum of edge weights on path p .

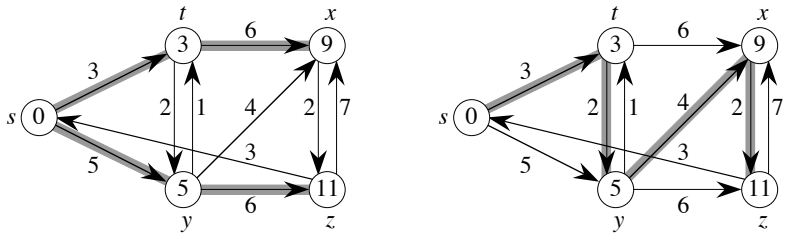
Shortest-path weight u to v:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there exists a path } u \leadsto v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Example: shortest paths from s

[d values appear inside vertices. Shaded edges show shortest paths.]



This example shows that the shortest path might not be unique.

It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as a tree.

Can think of weights as representing any measure that

- accumulates linearly along a path,
- we want to minimize.

Examples: time, cost, penalties, loss.

Generalization of breadth-first search to weighted graphs.

Variants

- ***Single-source:*** Find shortest paths from a given ***source*** vertex $s \in V$ to every vertex $v \in V$.
- ***Single-destination:*** Find shortest paths to a given destination vertex.
- ***Single-pair:*** Find shortest path from u to v. No way known that's better in worst case than solving single-source.
- ***All-pairs:*** Find shortest path from u to v for all $u, v \in V$. We'll see algorithms for all-pairs in the next chapter.

Negative-weight edges

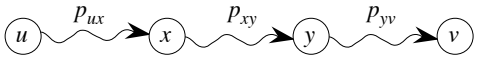
OK, as long as no negative-weight cycles are reachable from the source.

- If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
- But OK if the negative-weight cycle is not reachable from the source.
- Some algorithms work only if there are no negative-weight edges in the graph. We'll be clear when they're allowed and not allowed.

Optimal substructure

Lemma
Any subpath of a shortest path is a shortest path.

Proof Cut-and-paste.



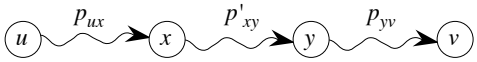
Suppose this path p is a shortest path from u to v.

Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \overset{p'_{xy}}{\rightsquigarrow} y$.

Then $w(p'_{xy}) < w(p_{xy})$.

Construct p':



Then

$$\begin{aligned} w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\ &< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\ &= w(p) \, . \end{aligned}$$

So p wasn't a shortest path after all! ■ (lemma)

Cycles

Shortest paths can't contain cycles:

- Already ruled out negative-weight cycles.
- Positive-weight $\Rightarrow$ we can get a shorter path by omitting the cycle.
- Zero-weight: no reason to use them $\Rightarrow$ assume that our solutions won't use them.

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

- $d[v] = \delta(s, v)$.
  - Initially, $d[v] = \infty$.
  - Reduces as algorithms progress. But always maintain $d[v] \geq \delta(s, v)$.
  - Call $d[v]$ a ***shortest-path estimate***.
- $\pi[v]$ = predecessor of v on a shortest path from s.
  - If no predecessor, $\pi[v] = \text{NIL}$.
  - π induces a tree—***shortest-path tree***.
  - We won't prove properties of π in lecture—see text.

Initialization

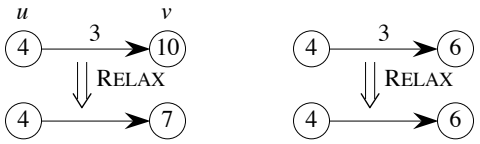
All the shortest-paths algorithms start with INIT-SINGLE-SOURCE.

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

Relaxing an edge (u, v)

Can we improve the shortest-path estimate for v by going through u and taking (u, v)?

RELAX(u, v, w)
if $d[v] > d[u] + w(u, v)$
  then $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$



For all the single-source shortest-paths algorithms we'll look at,

- start by calling INIT-SINGLE-SOURCE,
- then relax edges.

The algorithms differ in the order and how many times they relax each edge.

Shortest-paths properties

Based on calling INIT-SINGLE-SOURCE once and then calling RELAX zero or more times.

Triangle inequality

For all $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Proof Weight of shortest path $s \rightsquigarrow v$ is $\leq$ weight of any path $s \rightsquigarrow v$. Path $s \rightsquigarrow u \rightarrow v$ is a path $s \rightsquigarrow v$, and if we use a shortest path $s \rightsquigarrow u$, its weight is $\delta(s, u) + w(u, v)$. ■

Upper-bound property

Always have $d[v] \geq \delta(s, v)$ for all v. Once $d[v] = \delta(s, v)$, it never changes.

Proof Initially true.

Suppose there exists a vertex such that $d[v] < \delta(s, v)$.

Without loss of generality, v is first vertex for which this happens.

Let u be the vertex that causes $d[v]$ to change.

Then $d[v] = d[u] + w(u, v)$.

So,

$$\begin{aligned} d[v] &< \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) && \text{(triangle inequality)} \\ &\leq d[u] + w(u, v) && (v \text{ is first violation}) \\ \Rightarrow d[v] &< d[u] + w(u, v)\ . \end{aligned}$$

Contradicts $d[v] = d[u] + w(u, v)$.

Once $d[v]$ reaches $\delta(s, v)$, it never goes lower. It never goes up, since relaxations only lower shortest-path estimates. ■

No-path property

If $\delta(s, v) = \infty$, then $d[v] = \infty$ always.

Proof $d[v] \geq \delta(s, v) = \infty \Rightarrow d[v] = \infty$. ■

Convergence property

If $s \rightsquigarrow u \rightarrow v$ is a shortest path, $d[u] = \delta(s, u)$, and we call RELAX(u, v, w), then $d[v] = \delta(s, v)$ afterward.

Proof After relaxation:

$$\begin{aligned} d[v] &\leq d[u] + w(u, v) && \text{(RELAX code)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{(lemma—optimal substructure)} \end{aligned}$$

Since $d[v] \geq \delta(s, v)$, must have $d[v] = \delta(s, v)$. ■

Path relaxation property

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If we relax, *in order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $d[v_k] = \delta(s, v_k)$.

Proof Induction to show that $d[v_i] = \delta(s, v_i)$ after (v_{i-1}, v_i) is relaxed.

Basis: $i = 0$. Initially, $d[v_0] = 0 = \delta(s, v_0) = \delta(s, s)$.

Inductive step: Assume $d[v_{i-1}] = \delta(s, v_{i-1})$. Relax (v_{i-1}, v_i). By convergence property, $d[v_i] = \delta(s, v_i)$ afterward and $d[v_i]$ never changes. ■

The Bellman-Ford algorithm

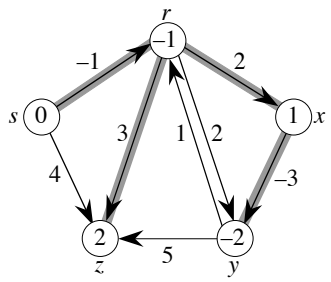
- Allows negative-weight edges.
- Computes $d[v]$ and $\pi[v]$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

```
BELLMAN-FORD(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
for i ← 1 to |V| − 1
    do for each edge (u, v) ∈ E
           do RELAX(u, v, w)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
          then return FALSE
return TRUE
```

Core: The first **for** loop relaxes all edges $|V| - 1$ times.

Time: $\Theta(VE)$.

Example:



Values you get on each pass and how quickly it converges depends on order of relaxation.

But guaranteed to converge after $|V| - 1$ passes, assuming no negative-weight cycles.

Proof Use path-relaxation property.

Let v be reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v, where $v_0 = s$ and $v_k = v$. Since p is acyclic, it has $\leq |V| - 1$ edges, so $k \leq |V| - 1$.

Each iteration of the **for** loop relaxes all edges:

- First iteration relaxes (v_0, v_1).
- Second iteration relaxes (v_1, v_2).
- kth iteration relaxes (v_{k-1}, v_k).

By the path-relaxation property, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. ■

How about the TRUE/FALSE return value?

- Suppose there is no negative-weight cycle reachable from s.

  At termination, for all $(u, v) \in E$,

  $$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \quad \text{(triangle inequality)} \\ &= d[u] + w(u, v) \,. \end{aligned}$$

  So BELLMAN-FORD returns TRUE.

- Now suppose there exists negative-weight cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, reachable from s.

  Then $\sum_{i=1}^{k} (v_{i-1}, v_i) < 0$.

  Suppose (for contradiction) that BELLMAN-FORD returns TRUE.

  Then $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.

  Sum around c:

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &\le \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

  Each vertex appears once in each summation $\sum_{i=1}^{k} d[v_i]$ and $\sum_{i=1}^{k} d[v_{i-1}] \Rightarrow$

  $0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

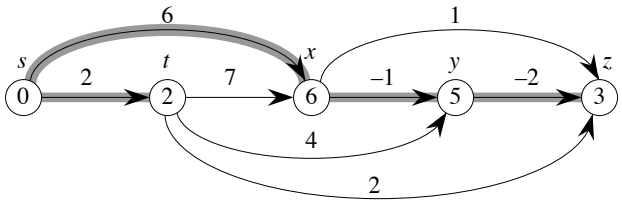
  This contradicts c being a negative-weight cycle! ∎

Single-source shortest paths in a directed acyclic graph

Since a dag, we're guaranteed no negative-weight cycles.

DAG-SHORTEST-PATHS(V, E, w, s)
topologically sort the vertices
INIT-SINGLE-SOURCE(V, s)
for each vertex u, taken in topologically sorted order
  do for each vertex $v \in Adj[u]$
    do RELAX(u, v, w)

Example:



Time: $\Theta(V + E)$.

Correctness: Because we process vertices in topologically sorted order, edges of *any* path must be relaxed in order of appearance in the path.
$\Rightarrow$ Edges on any shortest path are relaxed in order.
$\Rightarrow$ By path-relaxation property, correct. ∎

Dijkstra's algorithm

No negative-weight *edges*.

Essentially a weighted version of breadth-first search.

- Instead of a FIFO queue, uses a priority queue.
- Keys are shortest-path weights ($d[v]$).

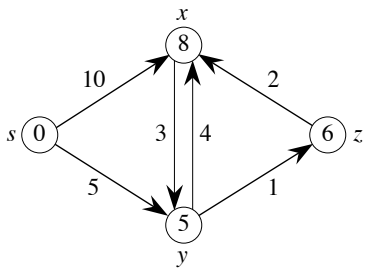
Have two sets of vertices:

- S = vertices whose final shortest-path weights are determined,
- Q = priority queue = $V - S$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

- Looks a lot like Prim's algorithm, but computing $d[v]$, and using shortest-path weights as keys.
- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" ("closest"?) vertex in $V - S$ to add to S.

Example:



Order of adding to S: s, y, z, x.

Correctness:

Loop invariant: At the start of each iteration of the **while** loop, $d[v] = \delta(s, v)$ for all $v \in S$.

Initialization: Initially, $S = \emptyset$, so trivially true.

Termination: At end, $Q = \emptyset \Rightarrow S = V \Rightarrow d[v] = \delta(s, v)$ for all $v \in V$.

Maintenance: Need to show that $d[u] = \delta(s, u)$ when u is added to S in each iteration.

Suppose there exists u such that $d[u] \neq \delta(s, u)$. Without loss of generality, let u be the first vertex for which $d[u] \neq \delta(s, u)$ when u is added to S.

Observations:

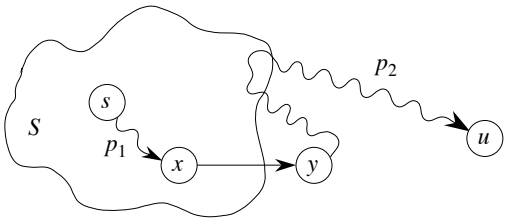
- $u \neq s$, since $d[s] = \delta(s, s) = 0$.
- Therefore, $s \in S$, so $S \neq \emptyset$.
- There must be some path $s \rightsquigarrow u$, since otherwise $d[u] = \delta(s, u) = \infty$ by no-path property.

So, there's a path $s \rightsquigarrow u$.

This means there's a shortest path $s \overset{p}{\rightsquigarrow} u$.

Just before u is added to S, path p connects a vertex in S (i.e., s) to a vertex in $V - S$ (i.e., u).

Let y be first vertex along p that's in $V - S$, and let $x \in S$ be y's predecessor.



Decompose p into $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$. (Could have $x = s$ or $y = u$, so that p_1 or p_2 may have no edges.)

Claim

$d[y] = \delta(s, y)$ when u is added to S.

Proof $x \in S$ and u is the first vertex such that $d[u] \neq \delta(s, u)$ when u is added to $S \Rightarrow d[x] = \delta(s, x)$ when x is added to S. Relaxed (x, y) at that time, so by the convergence property, $d[y] = \delta(s, y)$. ■ (claim)

Now can get a contradiction to $d[u] \neq \delta(s, u)$:

y is on shortest path $s \rightsquigarrow u$, and all edge weights are nonnegative
$\Rightarrow \delta(s, y) \leq \delta(s, u) \Rightarrow$

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq d[u] \qquad \text{(upper-bound property)} . \end{aligned}$$

Also, both y and u were in Q when we chose u, so

$d[u] \leq d[y] \Rightarrow d[u] = d[y]$.

Therefore, $d[y] = \delta(s, y) = \delta(s, u) = d[u]$.

Contradicts assumption that $d[u] \neq \delta(s, u)$. Hence, Dijkstra's algorithm is correct. ■

Analysis: Like Prim's algorithm, depends on implementation of priority queue.

- If binary heap, each operation takes $O(\lg V)$ time $\Rightarrow O(E \lg V)$.
- If a Fibonacci heap:
  - Each EXTRACT-MIN takes $O(1)$ amortized time.
  - There are $O(V)$ other operations, taking $O(\lg V)$ amortized time each.
  - Therefore, time is $O(V \lg V + E)$.

Difference constraints

Given a set of inequalities of the form $x_j - x_i \le b_k$.

- x's are variables, $1 \le i, j \le n$,
- b's are constants, $1 \le k \le m$.

Want to find a set of values for the x's that satisfy all m inequalities, or determine that no such values exist. Call such a set of values a ***feasible solution***.

Example:

$$\begin{aligned} x_1 - x_2 &\le 5 \\ x_1 - x_3 &\le 6 \\ x_2 - x_4 &\le -1 \\ x_3 - x_4 &\le -2 \\ x_4 - x_1 &\le -3 \end{aligned}$$

Solution: $x = (0, -4, -5, -3)$

Also: $x = (5, 1, 0, 2) =$ [above solution] $+ 5$

Lemma

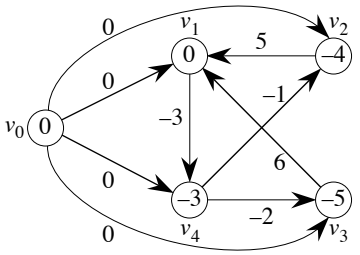
If x is a feasible solution, then so is $x + d$ for any constant d.

Proof x is a feasible solution $\Rightarrow x_j - x_i \le b_k$ for all i, j, k
$\Rightarrow (x_j + d) - (x_i + d) \le b_k$. ■ (lemma)

Constraint graph

$G = (V, E)$, weighted, directed.

- $V = \{v_0, v_1, v_2, \ldots, v_n\}$: one vertex per variable + v_0
- $E = \{(v_i, v_j) : x_j - x_i \le b_k$ is a constraint$\} \cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$
- $w(v_0, v_j) = 0$ for all j
- $w(v_i, v_j) = b_k$ if $x_j - x_i \le b_k$



Theorem

Given a system of difference constraints, let $G = (V, E)$ be the corresponding constraint graph.

1. If G has no negative-weight cycles, then

   $x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$

   is a feasible solution.
2. If G has a negative-weight cycle, then there is no feasible solution.

Proof

1. Show no negative-weight cycles $\Rightarrow$ feasible solution.

   Need to show that $x_j - x_i \le b_k$ for all constraints. Use

   $$\begin{aligned} x_j &= \delta(v_0, v_j) \\ x_i &= \delta(v_0, v_i) \\ b_k &= w(v_i, v_j) \ . \end{aligned}$$

   By the triangle inequality,

   $$\begin{aligned} \delta(v_0, v_j) &\le \delta(v_0, v_i) + w(v_i, v_j) \\ x_j &\le x_i + b_k \\ x_j - x_i &\le b_k \ . \end{aligned}$$

   Therefore, feasible.
2. Show negative-weight cycles $\Rightarrow$ no feasible solution.

   Without loss of generality, let a negative-weight cycle be $c = \langle v_1, v_2, \ldots, v_k \rangle$, where $v_1 = v_k$. (v_0 can't be on c, since v_0 has no entering edges.) c corresponds to the constraints

   $$\begin{aligned} x_2 - x_1 &\le w(v_1, v_2) \ , \\ x_3 - x_2 &\le w(v_2, v_3) \ , \\ &\vdots \\ x_{k-1} - x_{k-2} &\le w(v_{k-2}, v_{k-1}) \ , \\ x_k - x_{k-1} &\le w(v_{k-1}, v_k) \ . \end{aligned}$$

   [The last two inequalities above are incorrect in the first three printings of the book. They were corrected in the fourth printing.]

   If x is a solution satisfying these inequalities, it must satisfy their sum.

   So add them up.

Each x_i is added once and subtracted once. ($v_1 = v_k \Rightarrow x_1 = x_k$.)

We get $0 \leq w(c)$.

But $w(c) < 0$, since c is a negative-weight cycle.

Contradiction $\Rightarrow$ no such feasible solution x exists. ■ (theorem)

How to find a feasible solution

1. Form constraint graph.
   - $n + 1$ vertices.
   - $m + n$ edges.
   - $\Theta(m + n)$ time.
2. Run BELLMAN-FORD from v_0.
   - $O((n + 1)(m + n)) = O(n^2 + nm)$ time.
3. If BELLMAN-FORD returns FALSE $\Rightarrow$ no feasible solution.
   If BELLMAN-FORD returns TRUE $\Rightarrow$ set $x_i = \delta(v_0, v_i)$ for all i.

Solutions for Chapter 24: Single-Source Shortest Paths

Solution to Exercise 24.1-3

The proof of the convergence property shows that for every vertex v, the shortest-path estimate $d[v]$ has attained its final value after *length* (any shortest-weight path to v) iterations of BELLMAN-FORD. Thus after m passes, BELLMAN-FORD can terminate. We don't know m in advance, so we can't make the algorithm loop exactly m times and then terminate. But if we just make the algorithm stop when nothing changes any more, it will stop after $m+1$ iterations (i.e., after one iteration that makes no changes).

```
BELLMAN-FORD-(M+1)(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
changes ← TRUE
while changes = TRUE
    do changes ← FALSE
       for each edge (u, v) ∈ E[G]
           do RELAX-M(u, v, w)

RELAX-M(u, v, w)
if d[v] > d[u] + w(u, v)
  then d[v] ← d[u] + w(u, v)
       π[v] ← u
       changes ← TRUE
```

The test for a negative-weight cycle (based on there being a d that would change if another relaxation step was done) has been removed above, because this version of the algorithm will never get out of the **while** loop unless all d's stop changing.

Solution to Exercise 24.2-3

We'll give two ways to transform a PERT chart $G = (V, E)$ with weights on vertices to a PERT chart $G' = (V', E')$ with weights on edges. In each way, we'll have that $|V'| \leq 2|V|$ and $|E'| \leq |V| + |E|$. We can then run on G' the same

algorithm to find a longest path through a dag as is given in Section 24.2 of the text.

In the first way, we transform each vertex $v \in V$ into two vertices v' and v'' in V'. All edges in E that enter v will enter v' in E', and all edges in E that leave v will leave v'' in E'. In other words, if $(u, v) \in E$, then $(u'', v') \in E'$. All such edges have weight 0. We also put edges (v', v'') into E' for all vertices $v \in V$, and these edges are given the weight of the corresponding vertex v in G. Thus, $|V'| = 2|V|$, $|E'| = |V| + |E|$, and the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

In the second way, we leave vertices in V alone, but we add one new source vertex s to V', so that $V' = V \cup \{s\}$. All edges of E are in E', and E' also includes an edge (s, v) for every vertex $v \in V$ that has in-degree 0 in G. Thus, the only vertex with in-degree 0 in G' is the new source s. The weight of edge $(u, v) \in E'$ is the weight of vertex v in G. In other words, the weight of each entering edge in G is the weight of the vertex it enters in G. In effect, we have "pushed back" the weight of each vertex onto the edges that enter it. Here, $|V'| = |V| + 1$, $|E'| \le |V| + |E|$ (since no more than $|V|$ vertices have in-degree 0 in G), and again the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

Solution to Exercise 24.3-3

Yes, the algorithm still works. Let u be the leftover vertex that does not get extracted from the priority queue Q. If u is not reachable from s, then $d[u] = \delta(s, u) = \infty$. If u is reachable from s, there is a shortest path $p = s \rightsquigarrow x \rightarrow u$. When the node x was extracted, $d[x] = \delta(s, x)$ and then the edge (x, u) was relaxed; thus, $d[u] = \delta(s, u)$.

Solution to Exercise 24.3-4

To find the most reliable path between s and t, run Dijkstra's algorithm with edge weights $w(u, v) = -\lg r(u, v)$ to find shortest paths from s in $O(E+V \lg V)$ time. The most reliable path is the shortest path from s to t, and that path's reliability is the product of the reliabilities of its edges.

Here's why this method works. Because the probabilities are independent, the probability that a path will not fail is the product of the probabilities that its edges will not fail. We want to find a path $s \overset{p}{\rightsquigarrow} t$ such that $\prod_{(u,v)\in p} r(u, v)$ is maximized. This is equivalent to maximizing $\lg(\prod_{(u,v)\in p} r(u, v)) = \sum_{(u,v)\in p} \lg r(u, v)$, which is in turn equivalent to minimizing $\sum_{(u,v)\in p} -\lg r(u, v)$. (Note: $r(u, v)$ can be 0, and $\lg 0$ is undefined. So in this algorithm, define $\lg 0 = -\infty$.) Thus if we assign weights $w(u, v) = -\lg r(u, v)$, we have a shortest-path problem.

Since $\lg 1 = 0$, $\lg x < 0$ for $0 < x < 1$, and we have defined $\lg 0 = -\infty$, all the weights w are nonnegative, and we can use Dijkstra's algorithm to find the shortest paths from s in $O(E + V \lg V)$ time.

Alternate answer

You can also work with the original probabilities by running a modified version of Dijkstra's algorithm that maximizes the product of reliabilities along a path instead of minimizing the sum of weights along a path.

In Dijkstra's algorithm, use the reliabilities as edge weights and substitute

- max (and EXTRACT-MAX) for min (and EXTRACT-MIN) in relaxation and the queue,
- $\times$ for $+$ in relaxation,
- 1 (identity for $\times$) for 0 (identity for $+$) and $-\infty$ (identity for min) for ∞ (identity for max).

For example, the following is used instead of the usual RELAX procedure:

RELAX-RELIABILITY(u, v, r)
if $d[v] < d[u] \cdot r(u, v)$
  then $d[v] \leftarrow d[u] \cdot r(u, v)$
    $\pi[v] \leftarrow u$

This algorithm is isomorphic to the one above: It performs the same operations except that it is working with the original probabilities instead of the transformed ones.

Solution to Exercise 24.3-6

Observe that if a shortest-path estimate is not ∞, then it's at most $(|V| - 1)W$. Why? In order to have $d[v] < \infty$, we must have relaxed an edge (u, v) with $d[u] < \infty$. By induction, we can show that if we relax (u, v), then $d[v]$ is at most the number of edges on a path from s to v times the maximum edge weight. Since any acyclic path has at most $|V| - 1$ edges and the maximum edge weight is W, we see that $d[v] \leq (|V| - 1)W$. Note also that $d[v]$ must also be an integer, unless it is ∞.

We also observe that in Dijkstra's algorithm, the values returned by the EXTRACT-MIN calls are monotonically increasing over time. Why? After we do our initial $|V|$ INSERT operations, we never do another. The only other way that a key value can change is by a DECREASE-KEY operation. Since edge weights are nonnegative, when we relax an edge (u, v), we have that $d[u] \leq d[v]$. Since u is the minimum vertex that we just extracted, we know that any other vertex we extract later has a key value that is at least $d[u]$.

When keys are known to be integers in the range 0 to k and the key values extracted are monotonically increasing over time, we can implement a min-priority queue so that any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations takes $O(m + k)$ time. Here's how. We use an array, say $A[0\,..\,k]$, where $A[j]$ is a linked list of each element whose key is j. Think of $A[j]$ as a bucket for all elements with key j. We implement each bucket by a circular, doubly linked list with a sentinel, so that we can insert into or delete from each bucket in $O(1)$ time. We perform the min-priority queue operations as follows:

- INSERT: To insert an element with key j, just insert it into the linked list in $A[j]$. Time: $O(1)$ per INSERT.
- EXTRACT-MIN: We maintain an index *min* of the value of the smallest key extracted. Initially, *min* is 0. To find the smallest key, look in $A[min]$ and, if this list is nonempty, use any element in it, removing the element from the list and returning it to the caller. Otherwise, we rely on the monotonicity property (and that there is no INCREASE-KEY operation) and increment *min* until we either find a list $A[min]$ that is nonempty (using any element in $A[min]$ as before) or we run off the end of the array A (in which case the min-priority queue is empty).

  Since there are at most m INSERT operations, there are at most m elements in the min-priority queue. We increment *min* at most k times, and we remove and return some element at most m times. Thus, the total time over all EXTRACT-MIN operations is $O(m + k)$.
- DECREASE-KEY: To decrease the key of an element from j to i, first check whether $i \le j$, flagging an error if not. Otherwise, we remove the element from its list $A[j]$ in $O(1)$ time and insert it into the list $A[i]$ in $O(1)$ time. Time: $O(1)$ per DECREASE-KEY.

To apply this kind of min-priority queue to Dijkstra's algorithm, we need to let $k = (|V| - 1)W$, and we also need a separate list for keys with value ∞. The number of operations m is $O(V + E)$ (since there are $|V|$ INSERT and $|V|$ EXTRACT-MIN operations and at most $|E|$ DECREASE-KEY operations), and so the total time is $O(V + E + VW) = O(VW + E)$.

Solution to Exercise 24.3-7

First, observe that at any time, there are at most $W + 2$ distinct key values in the priority queue. Why? A key value is either ∞ or it is not. Consider what happens whenever a key value $d[v]$ becomes finite. It must have occurred due to the relaxation of an edge (u, v). At that time, u was being placed into S, and $d[u] \le d[y]$ for all vertices $y \in V - S$. After relaxing edge (u, v), we have $d[v] \le d[u] + W$. Since any other vertex $y \in V - S$ with $d[y] < \infty$ also had its estimate changed by a relaxation of some edge x with $d[x] \le d[u]$, we must have $d[y] \le d[x] + W \le d[u] + W$. Thus, at the time that we are relaxing edges from a vertex u, we must have, for all vertices $v \in V - S$, that $d[u] \le d[v] \le d[u] + W$ or $d[v] = \infty$. Since shortest-path estimates are integer values (except for ∞), at any given moment we have at most $W + 2$ different ones: $d[u], d[u] + 1, d[u] + 2, \ldots, d[u] + W$ and ∞.

Therefore, we can maintain the min-priorty queue as a binary min-heap in which each node points to a doubly linked list of all vertices with a given key value. There are at most $W + 2$ nodes in the heap, and so EXTRACT-MIN runs in $O(\lg W)$ time. To perform DECREASE-KEY, we need to be able to find the heap node corresponding to a given key in $O(\lg W)$ time. We can do so in $O(1)$ time as follows. First, keep a pointer *inf* to the node containing all the ∞ keys. Second, maintain an array $loc[0 \,..\, W]$, where $loc[i]$ points to the unique heap entry whose

key value is congruent to $i \pmod{(W+1)}$. As keys move around in the heap, we can update this array in $O(1)$ time per movement.

Alternatively, instead of using a binary min-heap, we could use a red-black tree. Now INSERT, DELETE, MINIMUM, and SEARCH—from which we can construct the priority-queue operations—each run in $O(\lg W)$ time.

Solution to Exercise 24.4-4

Let $\delta(u)$ be the shortest-path weight from s to u. Then we want to find $\delta(t)$.

δ must satisfy

$$\begin{aligned} \delta(s) &= 0 \\ \delta(v) - \delta(u) &\le w(u,v) \text{ for all } (u,v) \in E \qquad \text{(Lemma 24.10)} \end{aligned}$$

where $w(u, v)$ is the weight of edge (u, v).

Thus $x_v = \delta(v)$ is a solution to

$$\begin{aligned} x_s &= 0 \\ x_v - x_u &\le w(u,v) \ . \end{aligned}$$

To turn this into a set of inequalities of the required form, replace $x_s = 0$ by $x_s \le 0$ and $-x_s \le 0$ (i.e., $x_s \ge 0$). The constraints are now

$$\begin{aligned} x_s &\le 0 \ , \\ -x_s &\le 0 \ , \\ x_v - x_u &\le w(u,v) \ , \end{aligned}$$

which still has $x_v = \delta(v)$ as a solution.

However, δ isn't the only solution to this set of inequalities. (For example, if all edge weights are nonnegative, all $x_i = 0$ is a solution.) To force $x_t = \delta(t)$ as required by the shortest-path problem, add the requirement to maximize (the objective function) x_t. This is correct because

- $\max(x_t) \ge \delta(t)$ because $x_t = \delta(t)$ is part of one solution to the set of inequalities,
- $\max(x_t) \le \delta(t)$ can be demonstrated by a technique similar to the proof of Theorem 24.9:

  Let p be a shortest path from s to t. Then by definition,

  $$\delta(t) = \sum_{(u,v) \in p} w(u,v) \ .$$

  But for each edge (u, v) we have the inequality $x_v - x_u \le w(u, v)$, so

  $$\delta(t) = \sum_{(u,v) \in p} w(u,v) \ge \sum_{(u,v) \in p} (x_v - x_u) = x_t - x_s \ .$$

  But $x_s = 0$, so $x_t \le \delta(t)$.

Note: Maximizing x_t subject to the above inequalities solves the single-pair shortest-path problem when t is reachable from s and there are no negative-weight cycles. But if there's a negative-weight cycle, the inequalities have no feasible solution (as demonstrated in the proof of Theorem 24.9); and if t is not reachable from s, then x_t is unbounded.

Solution to Exercise 24.4-7

Observe that after the first pass, all d values are at most 0, and that relaxing edges (v_0, v_i) will never again change a d value. Therefore, we can eliminate v_0 by running the Bellman-Ford algorithm on the constraint graph without the v_0 node but initializing all shortest path estimates to 0 instead of ∞.

Solution to Exercise 24.4-10

To allow for single-variable constraints, we add the variable x_0 and let it correspond to the source vertex v_0 of the constraint graph. The idea is that, if there are no negative-weight cycles containing v_0, we will find that $\delta(v_0, v_0) = 0$. In this case, we set $x_0 = 0$, and so we can treat any single-variable constraint using x_i as if it were a 2-variable constraint with x_0 as the other variable.

Specifically, we treat the constraint $x_i \le b_k$ as if it were $x_i - x_0 \le b_k$, and we add the edge (v_0, v_i) with weight b_k to the constraint graph. We treat the constraint $-x_i \le b_k$ as if it were $x_0 - x_i \le b_k$, and we add the edge (v_i, v_0) with weight b_k to the constraint graph.

Once we find shortest-path weights from v_0, we set $x_i = \delta(v_0, v_i)$ for all $i = 0, 1, \ldots, n$; that is, we do as before but also include x_0 as one of the variables that we set to a shortest-path weight. Since v_0 is the source vertex, either $x_0 = 0$ or $x_0 < 0$.

If $\delta(v_0, v_0) = 0$, so that $x_0 = 0$, then setting $x_i = \delta(v_0, v_i)$ for all $i = 0, 1, \ldots, n$ gives a feasible solution for the system. The only new constraints beyond those in the text are those involving x_0. For constraints $x_i \le b_k$, we use $x_i - x_0 \le b_k$. By the triangle inequality, $\delta(v_0, v_i) \le \delta(v_0, v_0) + w(v_0, v_i) = b_k$, and so $x_i \le b_k$. For constraints $-x_i \le b_k$, we use $x_0 - x_i \le b_k$. By the triangle inequality, $0 = \delta(v_0, v_0) \le \delta(v_0, v_i) + w(v_i, v_0)$; thus, $0 \le x_i + b_k$ or, equivalently, $-x_i \le b_k$.

If $\delta(v_0, v_0) < 0$, so that $x_0 < 0$, then there is a negative-weight cycle containing v_0. The portion of the proof of Theorem 24.9 that deals with negative-weight cycles carries through but with v_0 on the negative-weight cycle, and we see that there is no feasible solution.

Solution to Exercise 24.5-4

Whenever RELAX sets π for some vertex, it also reduces the vertex's d value. Thus if $\pi[s]$ gets set to a non-NIL value, $d[s]$ is reduced from its initial value of 0 to a negative number. But $d[s]$ is the weight of some path from s to s, which is a cycle including s. Thus, there is a negative-weight cycle.

Solution to Exercise 24.5-7

Suppose we have a shortest-paths tree G_π. Relax edges in G_π according to the order in which a BFS would visit them. Then we are guaranteed that the edges along each shortest path are relaxed in order. By the path-relaxation property, we would then have $d[v] = \delta(s, v)$ for all $v \in V$. Since G_π contains at most $|V| - 1$ edges, we need to relax only $|V| - 1$ edges to get $d[v] = \delta(s, v)$ for all $v \in V$.

Solution to Exercise 24.5-8

Suppose that there is a negative-weight cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, that is reachable from the source vertex s; thus, $w(c) < 0$. Without loss of generality, c is simple. There must be an acyclic path from s to some vertex of c that uses no other vertices in c. Without loss of generality let this vertex of c be v_0, and let this path from s to v_0 be $p = \langle u_0, u_1, \ldots, u_l \rangle$, where $u_0 = s$ and $u_l = v_0 = v_k$. (It may be the case that $u_l = s$, in which case path p has no edges.) After the call to INITIALIZE-SINGLE-SOURCE sets $d[v] = \infty$ for all $v \in V - \{s\}$, perform the following sequence of relaxations. First, relax every edge in path p, in order. Then relax every edge in cycle c, in order, and repeatedly relax the cycle. That is, we relax the edges $(u_0, u_1), (u_1, u_2), \ldots, (u_{l-1}, v_0), (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_0), (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_0), (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_0), \ldots$.

We claim that every edge relaxation in this sequence reduces a shortest-path estimate. Clearly, the first time we relax an edge (u_{i-1}, u_i) or (v_{j-1}, v_j), for $i = 1, 2, \ldots, l$ and $j = 1, 2, \ldots, k - 1$ (note that we have not yet relaxed the last edge of cycle c), we reduce $d[u_i]$ or $d[v_j]$ from ∞ to a finite value. Now consider the relaxation of any edge (v_{j-1}, v_j) after this opening sequence of relaxations. We use induction on the number of edge relaxations to show that this relaxation reduces $d[v_j]$.

Basis: The next edge relaxed after the opening sequence is (v_{k-1}, v_k). Before relaxation, $d[v_k] = w(p)$, and after relaxation, $d[v_k] = w(p) + w(c) < w(p)$, since $w(c) < 0$.

Inductive step: Consider the relaxation of edge (v_{j-1}, v_j). Since c is a simple cycle, the last time $d[v_j]$ was updated was by a relaxation of this same edge. By the inductive hypothesis, $d[v_{j-1}]$ has just been reduced. Thus, $d[v_{j-1}] + w(v_{j-1}, v_j) < d[v_j]$, and so the relaxation will reduce the value of $d[v_j]$.

Solution to Problem 24-1

a. Assume for contradiction that G_f is not acyclic; thus G_f has a cycle. A cycle must have at least one edge (u, v) in which u has higher index than v. This

edge is not in E_f (by the definition of E_f), in contradition to the assumption that G_f has a cycle. Thus G_f is acyclic.

$\langle v_1, v_2, \ldots, v_{|V|} \rangle$ is a topological sort for G_f, because from the definition of E_f we know that all edges are directed from smaller indices to larger indices.

The proof for E_b is similar.

b. For all vertices $v \in V$, we know that either $\delta(s, v) = \infty$ or $\delta(s, v)$ is finite. If $\delta(s, v) = \infty$, then $d[v]$ will be ∞. Thus, we need to consider only the case where $d[v]$ is finite. There must be some shortest path from s to v. Let $p = \langle v_0, v_1, \ldots, v_{k-1}, v_k \rangle$ be that path, where $v_0 = s$ and $v_k = v$. Let us now consider how many times there is a change in direction in p, that is, a situation in which $(v_{i-1}, v_i) \in E_f$ and $(v_i, v_{i+1}) \in E_b$ or vice versa. There can be at most $|V| - 1$ edges in p, so there can be at most $|V| - 2$ changes in direction. Any portion of the path where there is no change in direction is computed with the correct d values in the first or second half of a single pass once the node that begins the no-change-in-direction sequence has the correct d value, because the edges are relaxed in the order of the direction of the sequence. Each change in direction requires a half pass in the new direction of the path. The following table shows the maximum number of passes needed depending on the parity of $|V| - 1$ and the direction of the first edge:

$\|V\| - 1$	first edge direction	passes
even	forward	$(\|V\| - 1)/2$
even	backward	$(\|V\| - 1)/2 + 1$
odd	forward	$\|V\|/2$
odd	backward	$\|V\|/2$

In any case, the maximum number of passes that we will need is $\lceil |V|/2 \rceil$.

c. This scheme does not affect the asymptotic running time of the algorithm because even though we perform only $\lceil |V|/2 \rceil$ passes instead of $|V| - 1$ passes, it is still $O(V)$ passes. Each pass still takes $\Theta(E)$ time, so the running time remains $O(VE)$.

Solution to Problem 24-2

a. Consider boxes with dimensions $x = (x_1, \ldots, x_d)$, $y = (y_1, \ldots, y_d)$, and $z = (z_1, \ldots, z_d)$. Suppose there exists a permutation π such that $x_{\pi(i)} < y_i$ for $i = 1, \ldots, d$ and there exists a permutation π' such that $y_{\pi'(i)} < z_i$ for $i = 1, \ldots, d$, so that x nests inside y and y nests inside z. Construct a permutation π'', where $\pi''(i) = \pi'(\pi(i))$. Then for $i = 1, \ldots, d$, we have $x_{\pi''(i)} = x_{\pi'(\pi(i))} < y_{\pi'(i)} < z_i$, and so x nests inside z.

b. Sort the dimensions of each box from longest to shortest. A box X with sorted dimensions $(x_1, x_2, \ldots, x_d)$ nests inside a box Y with sorted dimensions $(y_1, y_2, \ldots, y_d)$ if and only if $x_i < y_i$ for $i = 1, 2, \ldots, d$. The sorting can be done in $O(d \lg d)$ time, and the test for nesting can be done in $O(d)$ time, and so the algorithm runs in $O(d \lg d)$ time. This algorithm works because a

d-dimensional box can be oriented so that every permutation of its dimensions is possible. (Experiment with a 3-dimensional box if you are unsure of this).

c. Construct a dag $G = (V, E)$, where each vertex v_i corresponds to box B_i, and $(v_i, v_j) \in E$ if and only if box B_i nests inside box B_j. Graph G is indeed a dag, because nesting is transitive and antireflexive. The time to construct the dag is $O(dn^2 + dn \lg d)$, from comparing each of the $\binom{n}{2}$ pairs of boxes after sorting the dimensions of each.

Add a supersource vertex s and a supersink vertex t to G, and add edges (s, v_i) for all vertices v_i with in-degree 0 and (v_j, t) for all vertices v_j with out-degree 0. Call the resulting dag G'. The time to do so is $O(n)$.

Find a longest path from s to t in G'. (Section 24.2 discusses how to find a longest path in a dag.) This path corresponds to a longest sequence of nesting boxes. The time to find a longest path is $O(n^2)$, since G' has $n + 2$ vertices and $O(n^2)$ edges.

Overall, this algorithm runs in $O(dn^2 + dn \lg d)$ time.

Solution to Problem 24-3

a. We can use the Bellman-Ford algorithm on a suitable weighted, directed graph $G = (V, E)$, which we form as follows. There is one vertex in V for each currency, and for each pair of currencies c_i and c_j, there are directed edges (v_i, v_j) and (v_j, v_i). (Thus, $|V| = n$ and $|E| = \binom{n}{2}$.)

To determine edge weights, we start by observing that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

if and only if

$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} < 1 \, .$$

Taking logs of both sides of the inequality above, we express this condition as

$$\lg \frac{1}{R[i_1, i_2]} + \lg \frac{1}{R[i_2, i_3]} + \lg \frac{1}{R[i_{k-1}, i_k]} + \cdots + \lg \frac{1}{R[i_k, i_1]} < 0 \, .$$

Therefore, if we define the weight of edge (v_i, v_j) as

$$\begin{aligned} w(v_i, v_j) &= \lg \frac{1}{R[i, j]} \\ &= -\lg R[i, j] \, , \end{aligned}$$

then we want to find whether there exists a negative-weight cycle in G with these edge weights.

We can determine whether there exists a negative-weight cycle in G by adding an extra vertex v_0 with 0-weight edges (v_0, v_i) for all $v_i \in V$, running BELLMAN-FORD from v_0, and using the boolean result of BELLMAN-FORD (which is TRUE if there are no negative-weight cycles and FALSE if there is a

negative-weight cycle) to guide our answer. That is, we invert the boolean result of BELLMAN-FORD.

This method works because adding the new vertex v_0 with 0-weight edges from v_0 to all other vertices cannot introduce any new cycles, yet it ensures that all negative-weight cycles are reachable from v_0.

It takes $\Theta(n^2)$ time to create G, which has $\Theta(n^2)$ edges. Then it takes $O(n^3)$ time to run BELLMAN-FORD. Thus, the total time is $O(n^3)$.

Another way to determine whether a negative-weight cycle exists is to create G and, without adding v_0 and its incident edges, run either of the all-pairs shortest-paths algorithms. If the resulting shortest-path distance matrix has any negative values on the diagonal, then there is a negative-weight cycle.

b. Assuming that we ran BELLMAN-FORD to solve part (a), we only need to find the vertices of a negative-weight cycle. We can do so as follows. First, relax all the edges once more. Since there is a negative-weight cycle, the d value of some vertex u will change. We just need to repeatedly follow the π values until we get back to u. In other words, we can use the recursive method given by the PRINT-PATH procedure of Section 22.2, but stop it when it returns to vertex u.

The running time is $O(n^3)$ to run BELLMAN-FORD, plus $O(n)$ to print the vertices of the cycle, for a total of $O(n^3)$ time.

Solution to Problem 24-4

a. Since all weights are nonnegative, use Dijkstra's algorithm. Implement the priority queue as an array $Q[0\,..\,|E|+1]$, where $Q[i]$ is a list of vertices v for which $d[v] = i$. Initialize $d[v]$ for $v \neq s$ to $|E|+1$ instead of to ∞, so that all vertices have a place in Q. (Any initial $d[v] > \delta(s, v)$ works in the algorithm, since $d[v]$ decreases until it reaches $\delta(s, v)$.)

The $|V|$ EXTRACT-MINs can be done in $O(E)$ total time, and decreasing a d value during relaxation can be done in $O(1)$ time, for a total running time of $O(E)$.

- When $d[v]$ decreases, just add v to the front of the list in $Q[d[v]]$.
- EXTRACT-MIN removes the head of the list in the first nonempty slot of Q. To do EXTRACT-MIN without scanning all of Q, keep track of the smallest i for which $Q[i]$ is not empty. The key point is that when $d[v]$ decreases due to relaxation of edge (u, v), $d[v]$ remains $\geq d[u]$, so it never moves to an earlier slot of Q than the one that had u, the previous minimum. Thus EXTRACT-MIN can always scan upward in the array, taking a total of $O(E)$ time for all EXTRACT-MINs.

b. For all $(u, v) \in E$, we have $w_1(u, v) \in \{0, 1\}$, so $\delta_1(s, v) \leq |V| - 1 \leq |E|$. Use part (a) to get the $O(E)$ time bound.

c. To show that $w_i(u, v) = 2w_{i-1}(u, v)$ or $w_i(u, v) = 2w_{i-1}(u, v) + 1$, observe that the i bits of $w_i(u, v)$ consist of the $i-1$ bits of $w_{i-1}(u, v)$ followed by one

more bit. If that low-order bit is 0, then $w_i(u, v) = 2w_{i-1}(u, v)$; if it is 1, then $w_i(u, v) = 2w_{i-1}(u, v) + 1$.

Notice the following two properties of shortest paths:

1. If all edge weights are multiplied by a factor of c, then all shortest-path weights are multiplied by c.
2. If all edge weights are increased by at most c, then all shortest-path weights are increased by at most $c(|V| - 1)$, since all shortest paths have at most $|V| - 1$ edges.

The lowest possible value for $w_i(u, v)$ is $2w_{i-1}(u, v)$, so by the first observation, the lowest possible value for $\delta_i(s, v)$ is $2\delta_{i-1}(s, v)$.

The highest possible value for $w_i(u, v)$ is $2w_{i-1}(u, v) + 1$. Therefore, using the two observations together, the highest possible value for $\delta_i(s, v)$ is $2\delta_{i-1}(s, v) + |V| - 1$.

d. We have

$$\begin{aligned}\widehat{w}_i(u, v) &= w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) \\ &\geq 2w_{i-1}(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) \\ &\geq 0 \ .\end{aligned}$$

The second line follows from part (c). The third line follows from Lemma 24.10: $\delta_{i-1}(s, v) \leq \delta_{i-1}(s, u) + w_{i-1}(u, v)$.

e. Observe that if we compute $\widehat{w}_i(p)$ for any path $p : u \rightsquigarrow v$, the terms $\delta_{i-1}(s, t)$ cancel for every intermediate vertex t on the path. Thus,

$$\widehat{w}_i(p) = w_i(p) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) \ .$$

(This will be shown in detail in equation (25.10) within the proof of Lemma 25.1.) The δ_{i-1} terms depend only on u, v, and s, but not on the path p; therefore the same paths will be of minimum w_i weight and of minimum $\widehat{w}_i$ weight between u and v. Letting $u = s$, we get

$$\widehat{\delta}_i(s, v) = \delta_i(s, v) + 2\delta_{i-1}(s, s) - 2\delta_{i-1}(s, v) = \delta_i(s, v) - 2\delta_{i-1}(s, v) \ .$$

Rewriting this result as $\delta_i(s, v) = \widehat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$ and combining it with $\delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$ (from part (c)) gives us $\widehat{\delta}_i(s, v) \leq |V| - 1 \leq |E|$.

f. To compute $\delta_i(s, v)$ from $\delta_{i-1}(s, v)$ for all $v \in V$ in $O(E)$ time:

1. Compute the weights $\widehat{w}_i(u, v)$ in $O(E)$ time, as shown in part (d).
2. By part (e), $\widehat{\delta}_i(s, v) \leq |E|$, so use part (a) to compute all $\widehat{\delta}_i(s, v)$ in $O(E)$ time.
3. Compute all $\delta_i(s, v)$ from $\widehat{\delta}_i(s, v)$ and $\delta_{i-1}(s, v)$ as shown in part (e), in $O(V)$ time.

To compute all $\delta(s, v)$ in $O(E \lg W)$ time:

1. Compute $\delta_1(s, v)$ for all $v \in V$. As shown in part (b), this takes $O(E)$ time.
2. For each $i = 2, 3, \ldots, k$, compute all $\delta_i(s, v)$ from $\delta_{i-1}(s, v)$ in $O(E)$ time as shown above. This procedure computes $\delta(s, v) = \delta_k(u, v)$ in time $O(Ek) = O(E \lg W)$.

Solution to Problem 24-6

Observe that a bitonic sequence can increase, then decrease, then increase, or it can decrease, then increase, then decrease. That is, there can be at most two changes of direction in a bitonic sequence. Any sequence that increases, then decreases, then increases, then decreases has a bitonic sequence as a subsequence.

Now, let us suppose that we had an even stronger condition than the bitonic property given in the problem: for each vertex $v \in V$, the weights of the edges along any shortest path from s to v are increasing. Then we could call INITIALIZE-SINGLE-SOURCE and then just relax all edges one time, going in increasing order of weight. Then the edges along every shortest path would be relaxed in order of their appearance on the path. (We rely on the uniqueness of edge weights to ensure that the ordering is correct. *[Note that the uniqueness assumption was added in the fifth printing of the text.]*) The path-relaxation property (Lemma 24.15) would guarantee that we would have computed correct shortest paths from s to each vertex.

If we weaken the condition so that the weights of the edges along any shortest path increase and then decrease, we could relax all edges one time, in increasing order of weight, and then one more time, in decreasing order of weight. That order, along with uniqueness of edge weights, would ensure that we had relaxed the edges of every shortest path in order, and again the path-relaxation property would guarantee that we would have computed correct shortest paths.

To make sure that we handle all bitonic sequences, we do as suggested above. That is, we perform four passes, relaxing each edge once in each pass. The first and third passes relax edges in increasing order of weight, and the second and fourth passes in decreasing order. Again, by the path-relaxation property and the uniqueness of edge weights, we have computed correct shortest paths.

The total time is $O(V + E \lg V)$, as follows. The time to sort $|E|$ edges by weight is $O(E \lg E) = O(E \lg V)$ (since $|E| = O(V^2)$). INITIALIZE-SINGLE-SOURCE takes $O(V)$ time. Each of the four passes takes $O(E)$ time. Thus, the total time is $O(E \lg V + V + E) = O(V + E \lg V)$.

Lecture Notes for Chapter 25: All-Pairs Shortest Paths

Chapter 25 overview

Given a directed graph $G = (V, E)$, weight function $w : E \to \mathbf{R}$, $|V| = n$.

Goal: create an $n \times n$ matrix of shortest-path distances $\delta(u, v)$.

Could run BELLMAN-FORD once from each vertex:

- $O(V^2E)$—which is $O(V^4)$ if the graph is ***dense*** ($E = \Theta(V^2)$).

If no negative-weight edges, could run Dijkstra's algorithm once from each vertex:

- $O(VE \lg V)$ with binary heap—$O(V^3 \lg V)$ if dense,
- $O(V^2 \lg V + VE)$ with Fibonacci heap—$O(V^3)$ if dense.

We'll see how to do in $O(V^3)$ in all cases, with no fancy data structure.

Shortest paths and matrix multiplication

Assume that G is given as adjacency matrix of weights: $W = (w_{ij})$, with vertices numbered 1 to n.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j\ , \\ \text{weight of } (i, j) & \text{if } i \neq j, (i, j) \in E\ , \\ \infty & \text{if } i \neq j, (i, j) \notin E\ . \end{cases}$$

Output is matrix $D = (d_{ij})$, where $d_{ij} = \delta(i, j)$. Won't worry about predecessors—see book.

Will use dynamic programming at first.

Optimal substructure: Recall: subpaths of shortest paths are shortest paths.

Recursive solution: Let $l_{ij}^{(m)}$ = weight of shortest path $i \rightsquigarrow j$ that contains $\leq m$ edges.

- $m = 0$
  $\Rightarrow$ there is a shortest path $i \rightsquigarrow j$ with $\leq m$ edges if and only if $i = j$
  $\Rightarrow l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j\ , \\ \infty & \text{if } i \neq j\ . \end{cases}$

- $m \geq 1$

$$\begin{aligned} \Rightarrow l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &\quad (k \text{ is all possible predecessors of } j) \\ &= \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} \\ &\quad \text{since } w_{jj} = 0 \text{ for all } j \ . \end{aligned}$$

- Observe that when $m = 1$, must have $l_{ij}^{(1)} = w_{ij}$.

  Conceptually, when the path is restricted to at most 1 edge, the weight of the shortest path $i \rightsquigarrow j$ must be w_{ij}.

  And the math works out, too:

$$\begin{aligned} l_{ij}^{(1)} &= \min_{1 \leq k \leq n}\left\{l_{ik}^{(0)} + w_{kj}\right\} \\ &= l_{ii}^{(0)} + w_{ij} \qquad (l_{ii}^{(0)} \text{ is the only non-}\infty \text{ among } l_{ik}^{(0)}) \\ &= w_{ij} \ . \end{aligned}$$

All simple shortest paths contain $\leq n - 1$ edges
$\Rightarrow \delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots$

Compute a solution bottom-up: Compute $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$.

Start with $L^{(1)} = W$, since $l_{ij}^{(1)} = w_{ij}$.

Go from $L^{(m-1)}$ to $L^{(m)}$:

```
EXTEND(L, W, n)
create L', an n × n matrix
for i ← 1 to n
    do for j ← 1 to n
          do l'_ij ← ∞
             for k ← 1 to n
                do l'_ij ← min(l'_ij, l_ik + w_kj)
return L'
```

Compute each $L^{(m)}$:

```
SLOW-APSP(W, n)
L^(1) ← W
for m ← 2 to n − 1
    do L^(m) ← EXTEND(L^(m−1), W, n)
return L^(n−1)
```

Time:

- EXTEND: $\Theta(n^3)$.
- SLOW-APSP: $\Theta(n^4)$.

Observation: EXTEND is like matrix multiplication:

L	$\rightarrow$	A
W	$\rightarrow$	B
L'	$\rightarrow$	C
min	$\rightarrow$	$+$
$+$	$\rightarrow$	$\cdot$
∞	$\rightarrow$	0

create C, an $n \times n$ matrix
for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do $c_{ij} \leftarrow 0$
      for $k \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

So, we can view EXTEND as just like matrix multiplication!

Why do we care?

Because our goal is to compute $L^{(n-1)}$ as fast as we can. Don't need to compute *all* the intermediate $L^{(1)}, L^{(2)}, L^{(3)}, \ldots, L^{(n-2)}$.

Suppose we had a matrix A and we wanted to compute A^{n-1} (like calling EXTEND $n-1$ times).

Could compute $A, A^2, A^4, A^8, \ldots$

If we knew $A^m = A^{n-1}$ for all $m \geq n-1$, could just finish with A^r, where r is the smallest power of 2 that's $\geq n-1$. ($r = 2^{\lceil \lg(n-1) \rceil}$)

FASTER-APSP(W, n)
$L^{(1)} \leftarrow W$
$m \leftarrow 1$
while $m < n-1$
  do $L^{(2m)} \leftarrow$ EXTEND$(L^{(m)}, L^{(m)}, n)$
    $m \leftarrow 2m$
return $L^{(m)}$

OK to overshoot, since products don't change after $L^{(n-1)}$.

Time: $\Theta(n^3 \lg n)$.

Floyd-Warshall algorithm

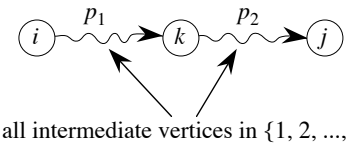
A different dynamic-programming approach.

For path $p = \langle v_1, v_2, \ldots, v_l \rangle$, an ***intermediate vertex*** is any vertex of p other than v_1 or v_l.

Let $d_{ij}^{(k)}$ = shortest-path weight of any path $i \rightsquigarrow j$ with all intermediate vertices in $\{1, 2, \ldots, k\}$.

Consider a shortest path $i \overset{p}{\rightsquigarrow} j$ with all intermediate vertices in $\{1, 2, \ldots, k\}$:

- If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, 2, \ldots, k-1\}$.
- If k is an intermediate vertex:



Recursive formulation

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \,. \end{cases}$$

(Have $d_{ij}^{(0)} = w_{ij}$ because can't have intermediate vertices $\Rightarrow \leq 1$ edge.)

Want $D^{(n)} = \left(d_{ij}^{(n)}\right)$, since all vertices numbered $\leq n$.

Compute bottom-up

Compute in increasing order of k:

FLOYD-WARSHALL(W, n)
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n
  do for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
      do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Can drop superscripts. (See Exercise 25.2-4 in text.)

Time: $\Theta(n^3)$.

Transitive closure

Given $G = (V, E)$, directed.
Compute $G^* = (V, E^*)$.

- $E^* = \{(i, j) :$ there is a path $i \rightsquigarrow j$ in $G\}$.

Could assign weight of 1 to each edge, then run FLOYD-WARSHALL.

- If $d_{ij} < n$, then there is a path $i \rightsquigarrow j$.
- Otherwise, $d_{ij} = \infty$ and there is no path.

Simpler way: Substitute other values and operators in FLOYD-WARSHALL.

- Use unweighted adjacency matrix
- min $\rightarrow \vee$ (OR)
- $+ \rightarrow \wedge$ (AND)
- $t_{ij}^{(k)} = \begin{cases} 1 & \text{if there is path } i \rightsquigarrow j \text{ with all intermediate vertices in } \{1, 2, \ldots, k\}\,, \\ 0 & \text{otherwise}\,. \end{cases}$
- $t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E\,, \\ 1 & \text{if } i = j \text{ or } (i, j) \in E\,. \end{cases}$
- $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$.

TRANSITIVE-CLOSURE(E, n)
for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do if $i = j$ or $(i, j) \in E[G]$
      then $t_{ij}^{(0)} \leftarrow 1$
      else $t_{ij}^{(0)} \leftarrow 0$
for $k \leftarrow 1$ **to** n
  do for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
      do $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
return $T^{(n)}$

Time: $\Theta(n^3)$, but simpler operations than FLOYD-WARSHALL.

Johnson's algorithm

Idea: If the graph is sparse, it pays to run Dijkstra's algorithm once from each vertex.

If we use a Fibonacci heap for the priority queue, the running time is down to $O(V^2 \lg V + VE)$, which is better than FLOYD-WARSHALL's $\Theta(V^3)$ time if $E = o(V^2)$.

But Dijkstra's algorithm requires that all edge weights be nonnegative.

Donald Johnson figured out how to make an equivalent graph that *does* have all edge weights ≥ 0.

Reweighting

Compute a new weight function $\widehat{w}$ such that

1. For all $u, v \in V$, p is a shortest path $u \rightsquigarrow v$ using w if and only if p is a shortest path $u \rightsquigarrow v$ using $\widehat{w}$.
2. For all $(u, v) \in E$, $\widehat{w}(u, v) \geq 0$.

Property (1) says that it suffices to find shortest paths with $\widehat{w}$. Property (2) says we can do so by running Dijkstra's algorithm from each vertex.

How to come up with $\widehat{w}$?

Lemma shows it's easy to get property (1):

Lemma (Reweighting doesn't change shortest paths)
Given a directed, weighted graph $G = (V, E)$, $w : E \to \mathbf{R}$. Let h be any function such that $h : V \to \mathbf{R}$. For all $(u, v) \in E$, define

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) .$$

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be any path $v_0 \rightsquigarrow v_k$.

Then, p is a shortest path $v_0 \rightsquigarrow v_k$ with w if and only if p is a shortest path $v_0 \rightsquigarrow v_k$ with $\widehat{w}$. (Formally, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w} = \widehat{\delta}(v_0, v_k)$, where $\widehat{\delta}$ is the shortest-path weight with $\widehat{w}$.)

Also, G has a negative-weight cycle with w if and only if G has a negative-weight cycle with $\widehat{w}$.

Proof First, we'll show that $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$:

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(sum telescopes)} \\
&= w(p) + h(v_0) - h(v_k) .
\end{aligned}$$

Therefore, any path $v_0 \overset{p}{\rightsquigarrow} v_k$ has $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$. Since $h(v_0)$ and $h(v_k)$ don't depend on the path from v_0 to v_k, if one path $v_0 \rightsquigarrow v_k$ is shorter than another with w, it's also shorter with $\widehat{w}$.

Now show there exists a negative-weight cycle with w if and only if there exists a negative-weight cycle with $\widehat{w}$:

- Let cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$.
- Then

$$\begin{aligned}
\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\
&= w(c) \qquad \text{(since } v_0 = v_k) .
\end{aligned}$$

Therefore, c has a negative-weight cycle with w if and only if it has a negative-weight cycle with $\widehat{w}$. ■ (lemma)

So, now to get property (2), we just need to come up with a function $h : V \to \mathbf{R}$ such that when we compute $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$, it's ≥ 0.

Do what we did for difference constraints:

- $G' = (V', E')$

- $V' = V \cup \{s\}$, where s is a new vertex.
- $E' = E \cup \{(s, v) : v \in V\}$.
- $w(s, v) = 0$ for all $v \in V$.

- Since no edges enter s, G' has the same set of cycles as G. In particular, G' has a negative-weight cycle if and only if G does.

Define $h(v) = \delta(s, v)$ for all $v \in V$.

Claim
$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$.

Proof By the triangle inequality,

$$\begin{aligned} \delta(s, v) &\leq \delta(s, u) + w(u, v) \\ h(v) &\leq h(u) + w(u, v) \ . \end{aligned}$$

Therefore, $w(u, v) + h(u) - h(v) \geq 0$. ■ (claim)

Johnson's algorithm

form G'
run BELLMAN-FORD on G' to compute $\delta(s, v)$ for all $v \in V$
if BELLMAN-FORD returns FALSE
  then G has a negative-weight cycle
  else
    compute $\widehat{w}(u, v) = w(u, v) + \delta(s, u) - \delta(s, v)$ for all $(u, v) \in E$
    for each vertex $u \in V$
      do run Dijkstra's algorithm from u using weight function $\widehat{w}$
        to compute $\widehat{\delta}(u, v)$ for all $v \in V$
      for each vertex $v \in V$
        do ▷ Compute entry d_{uv} in matrix D
          $d_{uv} = \underbrace{\widehat{\delta}(u, v) + \delta(s, v) - \delta(s, u)}$
            because if p is a path $u \rightsquigarrow v$,
            then $\widehat{w}(p) = w(p) + h(u) - h(v)$

Time:

- $\Theta(V + E)$ to compute G'.
- $O(VE)$ to run BELLMAN-FORD.
- $\Theta(E)$ to compute $\widehat{w}$.
- $O(V^2 \lg V + VE)$ to run Dijkstra's algorithm $|V|$ times (using Fibonacci heap).
- $\Theta(V^2)$ to compute D matrix.

Total: $O(V^2 \lg V + VE)$.

Solutions for Chapter 25: All-Pairs Shortest Paths

Solution to Exercise 25.1-3

The matrix $L^{(0)}$ corresponds to the identity matrix

$$I = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

of regular matrix multiplication. Substitute 0 (the identity for $+$) for ∞ (the identity for min), and 1 (the identity for $\cdot$) for 0 (the identity for $+$).

Solution to Exercise 25.1-5

The all-pairs shortest-paths algorithm in Section 25.1 computes

$L^{(n-1)} = W^{n-1} = L^{(0)} \cdot W^{n-1}$

where $l_{ij}^{(n-1)} = \delta(i, j)$ and $L^{(0)}$ is the identity matrix. That is, the entry in the ith row and jth column of the matrix "product" is the shortest-path distance from vertex i to vertex j, and row i of the product is the solution to the single-source shortest-paths problem for vertex i.

Notice that in a matrix "product" $C = A \cdot B$, the ith row of C is the ith row of A "multiplied" by B. Since all we want is the ith row of C, we never need more than the ith row of A.

Thus the solution to the single-source shortest-paths from vertex i is $L_i^{(0)} \cdot W^{n-1}$, where $L_i^{(0)}$ is the ith row of $L^{(0)}$—a vector whose ith entry is 0 and whose other entries are ∞.

Doing the above "multiplications" starting from the left is essentially the same as the BELLMAN-FORD algorithm. The vector corresponds to the d values in BELLMAN-FORD—the shortest-path estimates from the source to each vertex.

- The vector is initially 0 for the source and ∞ for all other vertices, the same as the values set up for d by INITIALIZE-SINGLE-SOURCE.

- Each "multiplication" of the current vector by W relaxes all edges just as BELLMAN-FORD does. That is, a distance estimate in the row, say the distance to v, is updated to a smaller estimate, if any, formed by adding some $w(u, v)$ to the current estimate of the distance to u.
- The relaxation/multiplication is done $n - 1$ times.

Solution to Exercise 25.1-10

Run SLOW-ALL-PAIRS-SHORTEST-PATHS on the graph. Look at the diagonal elements of $L^{(m)}$. Return the first value of m for which one (or more) of the diagonal elements ($l_{ii}^{(m)}$) is negative. If m reaches $n + 1$, then stop and declare that there are no negative-weight cycles.

Let the number of edges in a minimum-length negative-weight cycle be m^*, where $m^* = \infty$ if the graph has no negative-weight cycles.

Correctness: Let's assume that for some value $m^* \le n$ and some value of i, we find that $l_{ii}^{(m^*)} < 0$. Then the graph has a cycle with m^* edges that goes from vertex i to itself, and this cycle has negative weight (stored in $l_{ii}^{(m^*)}$). This is the minimum-length negative-weight cycle because SLOW-ALL-PAIRS-SHORTEST-PATHS computes all paths of 1 edge, then all paths of 2 edges, and so on, and all cycles shorter than m^* edges were checked before and did not have negative weight. Now assume that for all $m \le n$, there is no negative $l_{ii}^{(m)}$ element. This means there is no negative-weight cycle in the graph, because all cycles have length at most n.

Time: $O(n^4)$. More precisely, $\Theta(n^3 \cdot \min(n, m^*))$.

Faster solution

Run FASTER-ALL-PAIRS-SHORTEST-PATHS on the graph until the first time that the matrix $L^{(m)}$ has one or more negative values on the diagonal, or until we have computed $L^{(m)}$ for some $m > n$. If we find any negative entries on the diagonal, we know that the minimum-length negative-weight cycle has more than $m/2$ edges and at most m edges. We just need to binary search for the value of m^* in the range $m/2 < m^* \le m$. The key observation is that on our way to computing $L^{(m)}$, we computed $L^{(1)}$, $L^{(2)}$, $L^{(4)}$, $L^{(8)}$, ..., $L^{(m/2)}$, and these matrices suffice to compute every matrix we'll need. Here's pseudocode:

```
FIND-MIN-LENGTH-NEG-WEIGHT-CYCLE(W)
n ← rows[W]
L^(1) ← W
m ← 1
while m ≤ n and no diagonal entries of L^(m) are negative
    do L^(2m) ← EXTEND-SHORTEST-PATHS(L^(m), L^(m))
       m ← 2m
if m > n and no diagonal entries of L^(m) are negative
  then return "no negative-weight cycles"
elseif m ≤ 2
  then return m
else
       low ← m/2
       high ← m
       d ← m/4
       while d ≥ 1
           do s ← low + d
              L^(s) ← EXTEND-SHORTEST-PATHS(L^(low), L^(d))
              if L^(s) has any negative entries on the diagonal
                then high ← s
                else low ← s
              d ← d/2
       return high
```

Correctness: If, after the first **while** loop, $m > n$ and no diagonal entries of $L^{(m)}$ are negative, then there is no negative-weight cycle. Otherwise, if $m \le 2$, then either $m = 1$ or $m = 2$, and $L^{(m)}$ is the first matrix with a negative entry on the diagonal. Thus, the correct value to return is m.

If $m > 2$, then we maintain an interval bracketed by the values *low* and *high*, such that the correct value m^* is in the range $low < m^* \le high$. We use the following loop invariant:

Loop invariant: At the start of each iteration of the "**while** $d \ge 1$" loop,

1. $d = 2^p$ for some integer $p \ge -1$,
2. $d = (high - low)/2$,
3. $low < m^* \le high$.

Initialization: Initially, m is an integer power of 2 and $m > 2$. Since $d = m/4$, we have that d is an integer power of 2 and $d > 1/2$, so that $d = 2^p$ for some integer $p \ge 0$. We also have $(high - low)/2 = (m - (m/2))/2 = m/4 = d$. Finally, $L^{(m)}$ has a negative entry on the diagonal and $L^{(m/2)}$ does not. Since $low = m/2$ and $high = m$, we have that $low < m^* \le high$.

Maintenance: We use *high*, *low*, and d to denote variable values in a given iteration, and $high'$, low', and d' to denote the same variable values in the next iteration. Thus, we wish to show that $d = 2^p$ for some integer $p \ge -1$ implies $d' = 2^{p'}$ for some integer $p' \ge -1$, that $d = (high - low)/2$ implies $d' = (high' - low')/2$, and that $low < m^* \le high$ implies $low' < m^* \le high'$.

To see that $d' = 2^{p'}$, note that $d' = d/2$, and so $d = 2^{p-1}$. The condition that $d \geq 1$ implies that $p \geq 0$, and so $p' \geq -1$.

Within each iteration, s is set to $low + d$, and one of the following actions occurs:

- If $L^{(s)}$ has any negative entries on the diagonal, then $high'$ is set to s and d' is set to $d/2$. Upon entering the next iteration, $(high' - low')/2 = (s - low')/2 = ((low + d) - low)/2 = d/2 = d'$. Since $L^{(s)}$ has a negative diagonal entry, we know that $m^* \leq s$. Because $high' = s$ and $low' = low$, we have that $low' < m^* \leq high'$.
- If $L^{(s)}$ has no negative entries on the diagonal, then low' is set to s, and d' is set to $d/2$. Upon entering the next iteration, $(high' - low')/2 = (high' - s)/2 = (high - (low + d))/2 = (high - low)/2 - d/2 = d - d/2 = d/2 = d'$. Since $L^{(s)}$ has no negative diagonal entries, we know that $m^* > s$. Because $low' = s$ and $high' = high$, we have that $low' < m^* \leq high'$.

Termination: At termination, $d < 1$. Since $d = 2^p$ for some integer $p \geq -1$, we must have $p = -1$, so that $d = 1/2$. By the second part of the loop invariant, if we multiply both sides by 2, we get that $high - low = 2d = 1$. By the third part of the loop invariant, we know that $low < m^* \leq high$. Since $high - low = 2d = 1$ and $m^* > low$, the only possible value for m^* is $high$, which the procedure returns.

Time: If there is no negative-weight cycle, the first **while** loop iterates $\Theta(\lg n)$ times, and the total time is $\Theta(n^3 \lg n)$.

Now suppose that there is a negative-weight cycle. We claim that each time we call EXTEND-SHORTEST-PATHS$(L^{(low)}, L^{(d)})$, we have already computed $L^{(low)}$ and $L^{(d)}$. Initially, since $low = m/2$, we had already computed $L^{(low)}$ in the first **while** loop. In succeeding iterations of the second **while** loop, the only way that low changes is when it gets the value of s, and we have just computed $L^{(s)}$. As for $L^{(d)}$, observe that d takes on the values $m/4, m/8, m/16, \ldots, 1$, and again, we computed all of these L matrices in the first **while** loop. Thus, the claim is proven. Each of the two **while** loops iterates $\Theta(\lg m^*)$ times. Since we have already computed the parameters to each call of EXTEND-SHORTEST-PATHS, each iteration is dominated by the $\Theta(n^3)$-time call to EXTEND-SHORTEST-PATHS. Thus, the total time is $\Theta(n^3 \lg m^*)$.

In general, therefore, the running time is $\Theta(n^3 \lg \min(n, m^*))$.

Space: The slower algorithm needs to keep only three matrices at any time, and so its space requirement is $\Theta(n^3)$. This faster algorithm needs to maintain $\Theta(\lg \min(n, m^*))$ matrices, and so the space requirement increases to $\Theta(n^3 \lg \min(n, m^*))$.

Solution to Exercise 25.2-4

With the superscripts, the computation is $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$. If, having dropped the superscripts, we were to compute and store d_{ik} or d_{kj} before

using these values to compute d_{ij}, we might be computing one of the following:

$$\begin{aligned} d_{ij}^{(k)} &\leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}\right) \\ d_{ij}^{(k)} &\leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)}\right) \\ d_{ij}^{(k)} &\leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)}\right) \end{aligned}$$

In any of these scenarios, we're computing the weight of a shortest path from i to j with all intermediate vertices in $\{1, 2, \ldots, k\}$. If we use $d_{ik}^{(k)}$, rather than $d_{ik}^{(k-1)}$, in the computation, then we're using a subpath from i to k with all intermediate vertices in $\{1, 2, \ldots, k\}$. But k cannot be an *intermediate* vertex on a shortest path from i to k, since otherwise there would be a cycle on this shortest path. Thus, $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. A similar argument applies to show that $d_{kj}^{(k)} = d_{kj}^{(k-1)}$. Hence, we can drop the superscripts in the computation.

Solution to Exercise 25.2-6

Here are two ways to detect negative-weight cycles:

1. Check the main-diagonal entries of the result matrix for a negative value. There is a negative weight cycle if and only if $d_{ii}^{(n)} < 0$ for some vertex i:

   - $d_{ii}^{(n)}$ is a path weight from i to itself; so if it is negative, there is a path from i to itself (i.e., a cycle), with negative weight.
   - If there is a negative-weight cycle, consider the one with the fewest vertices.
     - If it has just one vertex, then some $w_{ii} < 0$, so d_{ii} starts out negative, and since d values are never increased, it is also negative when the algorithm terminates.
     - If it has at least two vertices, let k be the highest-numbered vertex in the cycle, and let i be some other vertex in the cycle. $d_{ik}^{(k-1)}$ and $d_{ki}^{(k-1)}$ have correct shortest-path weights, because they are not based on negative-weight cycles. (Neither $d_{ik}^{(k-1)}$ nor $d_{ki}^{(k-1)}$ can include k as an intermediate vertex, and i and k are on the negative-weight cycle with the fewest vertices.) Since $i \rightsquigarrow k \rightsquigarrow i$ is a negative-weight cycle, the sum of those two weights is negative, so $d_{ii}^{(k)}$ will be set to a negative value. Since d values are never increased, it is also negative when the algorithm terminates.

   In fact, it suffices to check whether $d_{ii}^{(n-1)} < 0$ for some vertex i. Here's why. A negative-weight cycle containing vertex i either contains vertex n or it does not. If it does not, then clearly $d_{ii}^{(n-1)} < 0$. If the negative-weight cycle contains vertex n, then consider $d_{nn}^{(n-1)}$. This value must be negative, since the cycle, starting and ending at vertex n, does not include vertex n as an intermediate vertex.
2. Alternatively, one could just run the normal FLOYD-WARSHALL algorithm one extra iteration to see if any of the d values change. If there are negative cycles, then some shortest-path cost will be cheaper. If there are no such cycles, then no d values will change because the algorithm gives the correct shortest paths.

Solution to Exercise 25.3-4

It changes shortest paths. Consider the following graph. $V = \{s, x, y, z\}$, and there are 4 edges: $w(s, x) = 2$, $w(x, y) = 2$, $w(s, y) = 5$, and $w(s, z) = -10$. So we'd add 10 to every weight to make $\widehat{w}$. With w, the shortest path from s to y is $s \to x \to y$, with weight 4. With $\widehat{w}$, the shortest path from s to y is $s \to y$, with weight 15. (The path $s \to x \to y$ has weight 24.) The problem is that by just adding the same amount to every edge, you penalize paths with more edges, even if their weights are low.

Solution to Exercise 25.3-6

In this solution, we assume that $\infty - \infty$ is undefined; in particular, it's not 0.

Let $G = (V, E)$, where $V = \{s, u\}$, $E = \{(u, s)\}$, and $w(u, s) = 0$. There is only one edge, and it enters s. When we run Bellman-Ford from s, we get $h(s) = \delta(s, s) = 0$ and $h(u) = \delta(s, u) = \infty$. When we reweight, we get $\widehat{w}(u, s) = 0 + \infty - 0 = \infty$. We compute $\widehat{\delta}(u, s) = \infty$, and so we compute $d_{us} = \infty + 0 - \infty \neq 0$. Since $\delta(u, s) = 0$, we get an incorrect answer.

If the graph G is strongly connected, then we get $h(v) = \delta(s, v) < \infty$ for all vertices $v \in V$. Thus, the triangle inequality says that $h(v) \leq h(u) + w(u, v)$ for all edges $(u, v) \in E$, and so $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$. Moreover, all edge weights $\widehat{w}(u, v)$ used in Lemma 25.1 are finite, and so the lemma holds. Therefore, the conditions we need in order to use Johnson's algorithm hold: that reweighting does not change shortest paths, and that all edge weights $\widehat{w}(u, v)$ are nonnegative. Again relying on G being strongly connected, we get that $\widehat{\delta}(u, v) < \infty$ for all edges $(u, v) \in E$, which means that $d_{uv} = \widehat{\delta}(u, v) + h(v) - h(u)$ is finite and correct.

Solution to Problem 25-1

a. Let T be the $|V| \times |V|$ matrix representing the transitive closure, such that $T[i, j]$ is 1 if there is a path from i to j, and 0 if not.

Initialize T (when there are no edges in G) as follows:

$$T[i, j] = \begin{cases} 1 & \text{if } i = j \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

T can be updated as follows when an edge (u, v) is added to G:

TRANSITIVE-CLOSURE-UPDATE(u, v)
for $i \leftarrow 1$ **to** $|V|$
  do for $j \leftarrow 1$ **to** $|V|$
    do if $T[i, u] = 1$ and $T[v, j] = 1$
      then $T[i, j] \leftarrow 1$

- This says that the effect of adding edge (u, v) is to create a path (via the new edge) from every vertex that could already reach u to every vertex that could already be reached from v.
- Note that the procedure sets $T[u, v] \leftarrow 1$, because of the initial values $T[u, u] = T[v, v] = 1$.
- This takes $\Theta(V^2)$ time because of the two nested loops.

b. Consider inserting the edge (v_n, v_1) into the straight-line graph $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$, where $n = |V|$.

Before this edge is inserted, only $n(n + 1)/2$ entries in T are 1 (the entries on and above the main diagonal). After the edge is inserted, the graph is a cycle in which every vertex can reach every other vertex, so all n^2 entries in T are 1. Hence $n^2 - (n(n + 1)/2) = \Theta(n^2) = \Theta(V^2)$ entries must be changed in T, so any algorithm to update the transitive closure must take $\Omega(V^2)$ time on this graph.

c. The algorithm in part (a) would take $\Theta(V^4)$ time to insert all possible $\Theta(V^2)$ edges, so we need a more efficient algorithm in order for any sequence of insertions to take only $O(V^3)$ total time.

To improve the algorithm, notice that the loop over j is pointless when $T[i, v] = 1$. That is, if there is already a path $i \rightsquigarrow v$, then adding the edge (u, v) can't make any new vertices reachable from i. The loop to set $T[i, j]$ to 1 for j such that there's a path $v \rightsquigarrow j$ is just setting entries that are already 1. Eliminate this redundant processing as follows:

```
TRANSITIVE-CLOSURE-UPDATE(u, v)
for i ← 1 to |V|
    do if T[i, u] = 1 and T[i, v] = 0
          then for j ← 1 to |V|
                  do if T[v, j] = 1
                        then T[i, j] ← 1
```

We show that this procedure takes $O(V^3)$ time to update the transitive closure for any sequence of n insertions:

- There can't be more than $|V|^2$ edges in G, so $n \leq |V|^2$.
- Summed over n insertions, time for the first two lines is $O(nV) = O(V^3)$.
- The last three lines, which take $\Theta(V)$ time, are executed only $O(V^2)$ times for n insertions. To see this, notice that the last three lines are executed only when $T[i, v] = 0$, and in that case, the last line sets $T[i, v] \leftarrow 1$. Thus, the number of 0 entries in T is reduced by at least 1 each time the last three lines run. Since there are only $|V|^2$ entries in T, these lines can run at most $|V|^2$ times.
- Hence the total running time over n insertions is $O(V^3)$.

Lecture Notes for Chapter 26: Maximum Flow

Chapter 26 overview

Network flow

Use a graph to model material that flows through conduits.

Each edge represents one conduit, and has a ***capacity***, which is an upper bound on the ***flow rate*** = units/time.

Can think of edges as pipes of different sizes. But flows don't have to be of liquids. Book has an example where a flow is how many trucks per day can ship hockey pucks between cities.

Want to compute max rate that we can ship material from a designated ***source*** to a designated ***sink***.

Flow networks

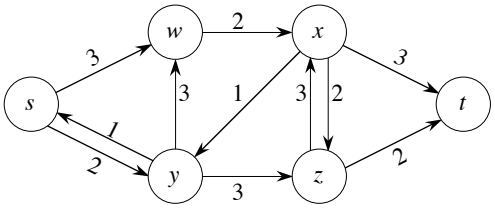
$G = (V, E)$ directed.

Each edge (u, v) has a ***capacity*** $c(u, v) \geq 0$.

If $(u, v) \notin E$, then $c(u, v) = 0$.

Source vertex s, ***sink*** vertex t, assume $s \rightsquigarrow v \rightsquigarrow t$ for all $v \in V$.

Example: *[Edges are labeled with capacities.]*



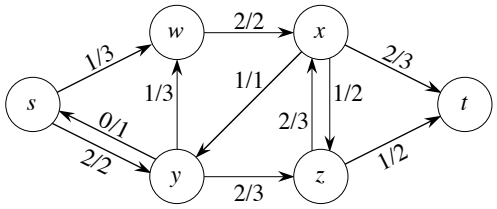
[In these notes, we define positive flow first because it's more intuitive to students than the flow formulation in the book. We'll migrate towards flow in the book soon. We'll call it "net flow" at first in the lecture notes. Net flow tends to be mathematically neater to work with than positive flow.]

Positive flow: A function $p : V \times V \to \mathbf{R}$ satisfying

- ***Capacity constraint:*** For all $u, v \in V, 0 \le p(u, v) \le c(u, v)$,
- ***Flow conservation:*** For all $u \in V - \{s, t\}$, $\underbrace{\sum_{v \in V} p(v, u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} p(u, v)}_{\text{flow out of } u}$.

  Equivalently, $\sum_{v \in V} p(u, v) - \sum_{v \in V} p(v, u) = 0$.

[Add flows to previous example. Edges here are labeled as flow/capacity.]



- Note that all positive flows are $\le$ capacities.
- Verify flow conservation by adding up flows at a couple of vertices.
- Note that all positive flows $= 0$ is legitimate.

Cancellation with positive flows

- Without loss of generality, can say positive flow goes either from u to v or from v to u, but not both. (Because if not true, can transform by cancellation to be true.)
- In the above example, we can "cancel" 1 unit of flow in each direction between x and z.

  1 unit $x \to z$ and 2 units $z \to x$ $\Rightarrow$ 0 units $x \to z$ and 1 unit $z \to x$
- In both cases, "net flow" is 1 unit $z \to x$.
- Capacity constraint is still satisfied (because flows only decrease).
- Flow conservation is still satisfied (flow in and flow out are both reduced by the same amount).

Here's a concept similar to positive flow:

Net flow: A function $f : V \times V \to \mathbf{R}$ satisfying

- ***Capacity constraint:*** For all $u, v \in V$, $f(u, v) \le c(u, v)$,
- ***Skew symmetry:*** For all $u, v \in V$, $f(u, v) = -f(v, u)$,
- ***Flow conservation:*** For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$.

Another way to think of flow conservation:

$$\underbrace{\sum_{v \in V : f(v,u) > 0} f(v, u)}_{\substack{\text{total positive} \\ \text{flow entering } u}} = \underbrace{\sum_{v \in V : f(u,v) > 0} f(u, v)}_{\substack{\text{total positive} \\ \text{flow leaving } u}}.$$

"flow in = flow out"

The differences between positive flow p and net flow f:

- $p(u, v) \geq 0$,
- f satisfies skew symmetry.

Equivalence of positive flow and net flow definitions

Define net flow in terms of positive flow:

- Define $f(u, v) = p(u, v) - p(v, u)$.
- Argue, given definition of p, that this definition of f satisfies capacity constraint and flow conservation.

  Capacity constraint:

  $p(u, v) \leq c(u, v)$ and $p(v, u) \geq 0 \Rightarrow p(u, v) - p(v, u) \leq c(u, v)$.

  Flow conservation:

$$\begin{aligned}\sum_{v \in V} f(u, v) &= \sum_{v \in V} (p(u, v) - p(v, u)) \\ &= \sum_{v \in V} p(u, v) - \sum_{v \in V} p(v, u) \\ &= 0 \, .\end{aligned}$$

- Skew symmetry is trivially satisfied by this definition of $f(u, v)$:

$$\begin{aligned}f(u, v) &= p(u, v) - p(v, u) \\ &= -(p(v, u) - p(u, v)) \\ &= -f(v, u) \, .\end{aligned}$$

Define positive flow in terms of net flow:

- Define

$$p(u, v) = \begin{cases} f(u, v) & \text{if } f(u, v) > 0 \, , \\ 0 & \text{if } f(u, v) \leq 0 \, . \end{cases}$$

- Argue, given definition of f, that this definition of p satisfies capacity constraint and flow conservation.

  Capacity constraint:

  - If $f(u, v) > 0$:
    $f(u, v) \leq c(u, v) \Rightarrow 0 \leq p(u, v) \leq c(u, v)$.
  - If $f(u, v) \leq 0$:
    $0 = p(u, v) \leq c(u, v)$.

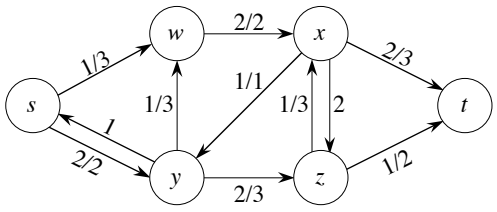
Flow conservation:

$$\begin{aligned}
&\sum_{v\in V} p(u,v) - \sum_{v\in V} p(v,u) \\
&\quad = \left(\sum_{v\in V:f(u,v)>0} p(u,v) + \sum_{v\in V:f(u,v)\le 0} p(u,v)\right) \\
&\qquad\quad - \left(\sum_{v\in V:f(u,v)>0} p(v,u) + \sum_{v\in V:f(u,v)\le 0} p(v,u)\right) \\
&\quad = \sum_{v\in V:f(u,v)>0} p(u,v) + \sum_{v\in V:f(u,v)\le 0} p(u,v) \\
&\qquad\quad - \sum_{v\in V:f(v,u)<0} p(v,u) - \sum_{v\in V:f(v,u)\ge 0} p(v,u) \\
&\quad = \sum_{v\in V:f(u,v)>0} p(u,v) + 0 - 0 - \sum_{v\in V:f(v,u)\ge 0} p(v,u) \\
&\quad = \sum_{v\in V:f(u,v)>0} f(u,v) - \sum_{v\in V:f(v,u)\ge 0} f(v,u) \\
&\quad = \sum_{v\in V:f(u,v)>0} f(u,v) - \sum_{v\in V:f(u,v)\le 0} (-f(u,v)) \\
&\quad = \sum_{v\in V} f(u,v) \\
&\quad = 0\,.
\end{aligned}$$

[We'll use net flow, instead of positive flow, for the rest of our discussion, in order to cut the number of summations down by half. From now on, we'll just call it "flow" rather than "net flow."]

Value of flow $f = |f| = \sum_{v\in V} f(s,v) =$ total flow out of source.

Consider the example below. *[The cancellation possible in the previous example has been made here. Also, showing only flows that are positive.]*



Value of flow $f = |f| = 3$.

Cancellation with flow

If we have edges (u,v) and (v,u), skew symmetry makes it so that *at most* one of these edges has positive flow.

Say $f(u,v) = 5$. If we "ship" 2 units $v \to u$, we lower $f(u,v)$ to 3. The 2 units $v \to u$ ***cancel*** 2 of the $u \to v$ units.

Due to cancellation, a flow only gives us this "net" effect. We cannot reconstruct actual shipments from a flow.

5 units $u \to v$	same as	8 units $u \to v$
0 units $v \to u$		3 units $v \to u$

We could add another 3 units of flow $u \to v$ and another 3 units $v \to u$, maintaining flow conservation.

The flow from u to v would remain $f(u, v) = 5$, and $f(v, u) = -5$.

Maximum-flow problem: Given G, s, t, and c, find a flow whose value is maximum.

Implicit summation notation

We work with functions, like f, that take pairs of vertices as arguments.

Extend to take sets of vertices, with interpretation of summing over all vertices in the set.

Example: If X and Y are sets of vertices,

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y) \ .$$

Therefore, can express flow conservation as $f(u, V) = 0$, for all $u \in V - \{s, t\}$.

Notation: Omit braces in sets with implicit summations.

Example: $f(s, V - s) = f(s, V)$. Here, $f(s, V - s)$ really means $f(s, V - \{s\})$.

Lemma

For any flow f in $G = (V, E)$:

1. For all $X \subseteq V$, $f(X, X) = 0$,
2. For all $X, Y \subseteq V$, $f(X, Y) = -f(Y, X)$,
3. For all $X, Y, Z \subseteq V$ such that $X \cap Y = \emptyset$,
   $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ and
   $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.

[Leave on board.]

Proof

2.
$$\begin{aligned} f(X, Y) &= \sum_{x \in X} \sum_{y \in Y} f(x, y) \\ &= \sum_{x \in X} \sum_{y \in Y} -f(y, x) \quad \text{(skew symmetry)} \\ &= \sum_{y \in Y} \sum_{x \in X} -f(y, x) \\ &= -f(Y, X) \end{aligned}$$

1.
$$\begin{aligned} f(X, X) &= -f(X, X) \quad \text{(part (2))} \\ \Rightarrow f(X, X) &= 0 \end{aligned}$$

3. $$\begin{aligned} f(X \cup Y, Z) &= \sum_{v \in X \cup Y} \sum_{z \in Z} f(v, z) \\ &= \sum_{v \in X} \left(\sum_{z \in Z} f(v, z) \right) + \sum_{v \in Y} \left(\sum_{z \in Z} f(v, z) \right) \quad (X \cap Y = \emptyset) \\ &= f(X, Z) + f(Y, Z) \end{aligned}$$

Other part is symmetric. ■ (lemma)

Example of using this lemma:

Lemma
$|f| = f(V, t)$.

Proof First, show that $f(V, V - s - t) = 0$:
$f(u, V) = 0$ for all $u \in V - \{s, t\}$
$\Rightarrow f(V - s - t, V) = 0$ (add up $f(u, V)$ for all $u \in V - \{s, t\}$)
$\Rightarrow f(V, V - s - t) = 0$ (by lemma, part (2)).

Thus,

$$\begin{aligned} |f| &= f(s, V) && \text{(definition of } |f|) \\ &= f(V, V) - f(V - s, V) && \text{(lemma, part (3))} \\ &= -f(V - s, V) && \text{(lemma, part (1))} \\ &= f(V, V - s) && \text{(lemma, part (2))} \\ &= f(V, t) + f(V, V - s - t) && \text{(lemma, part (3))} \\ &= f(V, t) && \text{(from above)} \end{aligned}$$

■ (lemma)

Cuts

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

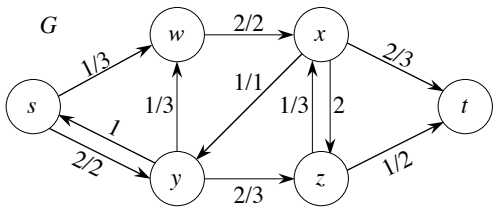
- Similar to cut used in minimum spanning trees, except that here the graph is directed, and we require $s \in S$ and $t \in T$.

For flow f, the ***net flow*** across cut (S, T) is $f(S, T)$.

Capacity of cut (S, T) is $c(S, T)$.

A ***minimum cut*** of G is a cut whose capacity is minimum over all cuts of G.

For our example: *[Leave on board.]*



Consider the cut $S = \{s, w, y\}$, $T = \{x, z, t\}$.

$$\begin{aligned} f(S,T) &= f(w,x) + f(y,x) + f(y,z) \\ &= 2 + -1 + 2 \\ &= 3 . \end{aligned}$$

$$\begin{aligned} c(S,T) &= c(w,x) + c(y,z) \\ &= 2 + 3 \\ &= 5 . \end{aligned}$$

Note the difference between capacity and flow.

- Flow obeys skew symmetry, so $f(y,x) = -f(x,y) = -1$.
- Capacity does not: $c(y,x) = 0$, but $c(x,y) = 1$.

So include flows going both ways across the cut, but capacity going only S to T.

Now consider the cut $S = \{s, w, x, y\}$, $T = \{z, t\}$.

$$\begin{aligned} f(S,T) &= f(x,z) + f(x,t) + f(y,z) \\ &= -1 + 2 + 2 \\ &= 3 . \end{aligned}$$

$$\begin{aligned} c(S,T) &= c(x,z) + c(x,t) + c(y,z) \\ &= 2 + 3 + 3 \\ &= 8 . \end{aligned}$$

Same flow as previous cut, higher capacity.

Lemma

For any cut (S, T), $f(S,T) = |f|$.

[Leave on board.]

Proof First, show that $f(S - s, V) = 0$:

$$S - \{s\} \subseteq V - \{s, t\} .$$

Therefore,

$$\begin{aligned} f(S-s,V) &= \sum_{u \in S - \{s\}} f(u,V) \\ &= \sum_{u \in S - \{s\}} 0 \qquad \text{(flow conservation and } S - \{s\} \subseteq V - \{s,t\}) \\ &= 0 . \end{aligned}$$

So,

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) && \text{(lemma, part (3), } S \cup T = V, S \cap T = \emptyset) \\ &= f(S,V) && \text{(lemma, part (1))} \\ &= f(s,V) + f(S-s,V) && \text{(lemma, part (3))} \\ &= f(s,V) && (f(S-s,V) = 0) \\ &= |f| && \blacksquare \text{ (lemma)} \end{aligned}$$

Corollary
The value of any flow ≤ capacity of any cut.
[Leave on board.]

Proof Let (S, T) be any cut, f be any flow.

$$\begin{aligned} |f| &= f(S, T) && \text{(lemma)} \\ &= \sum_{u \in S} \sum_{v \in T} f(u, v) \\ &\le \sum_{u \in S} \sum_{v \in T} c(u, v) && \text{(capacity constraints)} \\ &= c(S, T) \ . \end{aligned}$$

■ (corollary)

Therefore, maximum flow ≤ capacity of minimum cut.

Will see a little later that this is in fact an equality.

The Ford-Fulkerson method

Residual network

Given a flow f in network $G = (V, E)$.

Consider a pair of vertices $u, v \in V$.

How much additional flow can we push directly from u to v?
That's the ***residual capacity***,

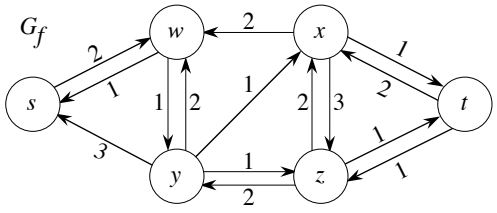
$$\begin{aligned} c_f(u, v) &= c(u, v) - f(u, v) \\ &\ge 0 && \text{(since } f(u, v) \le c(u, v)) \ . \end{aligned}$$

Residual network: $G_f = (V, E_f)$,

$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.

Each edge of the residual network can admit a positive flow.

For our example:



Every edge $(u, v) \in E_f$ corresponds to an edge $(u, v) \in E$ or $(v, u) \in E$ (or both).

Therefore, $|E_f| \le 2\,|E|$.

Given flows f_1 and f_2, the ***flow sum*** $f_1 + f_2$ is defined by

$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$

for all $u, v \in V$.

Lemma

Given a flow network G, a flow f in G, and the residual network G_f, let f' be any flow in G_f. Then the flow sum $f + f'$ is a flow in G with value $|f + f'| = |f| + |f'|$.

[See book for proof.]

Augmenting path

A path $s \rightsquigarrow t$ in G_f.

- Admits more flow along each edge.
- Like a sequence of pipes through which we can squirt more flow from s to t.

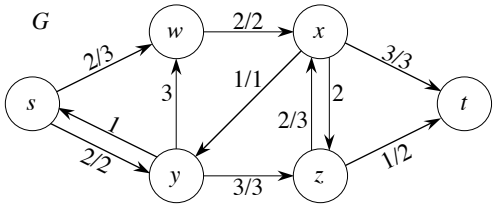
How much more flow can we push from s to t along augmenting path p?

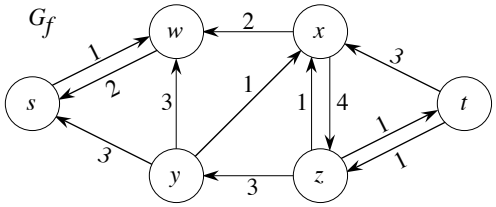
$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$.

For our example, consider the augmenting path $p = \langle s, w, y, z, x, t\rangle$.

Minimum residual capacity is 1.

After we push 1 additional unit along p: *[Continue from G left on board from before.]*





Observe that G_f now has no augmenting path. Why? No edges cross the cut $(\{s, w\}, \{x, y, z, t\})$ in the forward direction in G_f. So no path can get from s to t.

Claim that the flow shown in G is a maximum flow.

Lemma

Given flow network G, flow f in G, residual network G_f. Let p be an augmenting path in G_f. Define $f_p : V \times V \to \mathbf{R}$:

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p\ , \\ -c_f(p) & \text{if } (v, u) \text{ is on } p\ , \\ 0 & \text{otherwise .} \end{cases}$$

Then f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Corollary

Given flow network G, flow f in G, and an augmenting path p in G_f, define f_p as in lemma, and define $f' : V \times V \to \mathbf{R}$ by $f' = f + f_p$. Then f' is a flow in G with value $|f'| = |f| + c_f(p) > |f|$.

Theorem (Max-flow min-cut theorem)

The following are equivalent:

1. f is a maximum flow.
2. f admits no augmenting path.
3. $|f| = c(S, T)$ for some cut (S, T).

Proof

(1) $\Rightarrow$ (2): If f admits an augmenting path p, then (by above corollary) would get a flow with value $|f| + c_f(p) > |f|$, so f wasn't a max flow to start with.

(2) $\Rightarrow$ (3): Suppose f admits no augmenting path. Define

$$S = \{v \in V : \text{there exists a path } s \rightsquigarrow v \text{ in } G_f\} ,$$
$$T = V - S .$$

Must have $t \in T$; otherwise there is an augmenting path.
Therefore, (S, T) is a cut.
For each $u \in S$ and $v \in T$, must have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$ and then $v \in S$.
By lemma ($f(S, T) = |f|$), $|f| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): By corollary, $|f| \le c(S, T)$.
$|f| = c(S, T) \Rightarrow f$ is a max flow. ■ (theorem)

Ford-Fulkerson algorithm

FORD-FULKERSON(V, E, s, t)
for all $(u, v) \in E$
  do $f[u, v] \leftarrow f[v, u] \leftarrow 0$
while there is an augmenting path p in G_f
  do augment f by $c_f(p)$

Subtle difference between $f[u, v]$ and $f(u, v)$:

- $f(u, v)$ is a function, defined on all $u, v \in V$.
- $f[u, v]$ is a value computed by algorithm.
  - $f[u, v] = f(u, v)$ where $(u, v) \in E$ or $(v, u) \in E$.
  - $f[u, v]$ is undefined if neither $(u, v) \in E$ nor $(v, u) \in E$.

Analysis: If capacities are all integer, then each augmenting path raises $|f|$ by ≥ 1. If max flow is f^*, then need $\leq |f^*|$ iterations $\Rightarrow$ time is $O(E\,|f^*|)$.
[Handwaving—see book for better explanation.]

Note that this running time is *not* polynomial in input size. It depends on $|f^*|$, which is not a function of $|V|$ and $|E|$.

If capacities are rational, can scale them to integers.

If irrational, FORD-FULKERSON might never terminate!

Edmonds-Karp algorithm

Do FORD-FULKERSON, but compute augmenting paths by BFS of G_f. Augmenting paths are shortest paths $s \rightsquigarrow t$ in G_f, with all edge weights $= 1$.

Edmonds-Karp runs in $O(VE^2)$ time.

To prove, need to look at distances to vertices in G_f.

Let $\delta_f(u, v) =$ shortest path distance u to v in G_f, with unit edge weights.

Lemma
For all $v \in V - \{s, t\}$, $\delta_f(s, v)$ increases monotonically with each flow augmentation.

Proof Suppose there exists $v \in V - \{s, t\}$ such that there is a flow augmentation that causes $\delta_f(s, v)$ to decrease. Will derive a contradiction.

Let f be the flow before the first augmentation that causes a shortest-path distance to decrease, f' be the flow afterward.

Let v be a vertex with minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so $\delta_{f'}(s, v) < \delta_f(s, v)$.

Let a shortest path s to v in $G_{f'}$ be $s \rightsquigarrow u \rightarrow v$, so $(u, v) \in E_{f'}$ and $\delta_{f'}(s, v) = \delta_{f'}(s, u) + 1$. (Or $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$.)

Since $\delta_{f'}(s, u) < \delta_{f'}(s, v)$ and how we chose v, we have $\delta_{f'}(s, u) \geq \delta_f(s, u)$.

Claim
$(u, v) \notin E_f$.

Proof If $(u, v) \in E_f$, then

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 \quad \text{(triangle inequality)} \\ &\leq \delta_{f'}(s, u) + 1 \\ &= \delta_{f'}(s, v)\ , \end{aligned}$$

contradicting $\delta_{f'}(s, v) < \delta_f(s, v)$. ■ (claim)

How can $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$?

The augmentation must increase flow v to u.

Since Edmonds-Karp augments along shortest paths, the shortest path s to u in G_f has $v \rightarrow u$ as its last edge.

Therefore,

$$\begin{aligned}\delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 \\ &= \delta_{f'}(s, v) - 2 ,\end{aligned}$$

contradicting $\delta_{f'}(s, v) < \delta_f(s, v)$.

Therefore, v cannot exist. ■ (lemma)

Theorem

Edmonds-Karp performs $O(VE)$ augmentations.

Proof Suppose p is an augmenting path and $c_f(u, v) = c_f(p)$. Then call (u, v) a ***critical*** edge in G_f, and it disappears from the residual network after an augmentation along p.

≥ 1 edge on any augmenting path is critical.

Will show that each of the $|E|$ edges can become critical $\leq |V|/2 - 1$ times.

Consider $u, v \in V$ such that either $(u, v) \in E$ or $(v, u) \in E$ or both. Since augmenting paths are shortest paths, when (u, v) becomes critical first time, $\delta_f(s, v) = \delta_f(s, u) + 1$.

Augment flow, so that (u, v) disppears from the residual network. This edge cannot reappear in the residual network until flow from u to v decreases, which happens only if (v, u) is on an augmenting path in $G_{f'}$: $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$. ($f'$ is flow when this occurs.)

By lemma, $\delta_f(s, v) \leq \delta_{f'}(s, v) \Rightarrow$

$$\begin{aligned}\delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2 .\end{aligned}$$

Therefore, from the time (u, v) becomes critical to the next time, distance of u from s increases by ≥ 2. Initially, distance to u is ≥ 0, and augmenting path can't have s, u, and t as intermediate vertices.

Therefore, until u becomes unreachable from source, its distance is $\leq |V| - 2 \Rightarrow$ u can become critical $\leq (|V| - 2)/2 = |V|/2 - 1$ times.

Since $O(E)$ pairs of vertices can have an edge between them in residual graph, total # of critical edges is $O(VE)$. Since each augmenting path has ≥ 1 critical edge, have $O(VE)$ augmentations. ■ (theorem)

Use BFS to find each augmenting path in $O(E)$ time $\Rightarrow O(VE^2)$ time.

Can get better bounds.

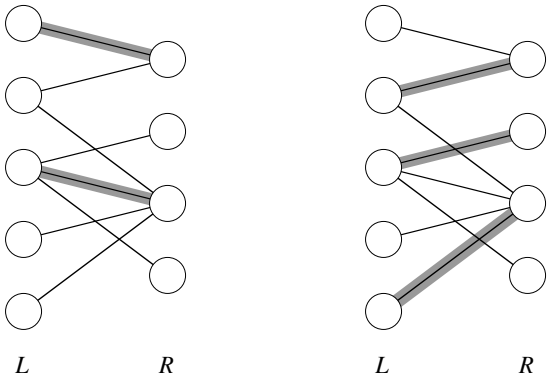
Push-relabel algorithms in Sections 26.4–26.5 give $O(V^3)$.

Can do even better.

Maximum bipartite matching

Example of a problem that can be solved by turning it into a flow problem.

$G = (V, E)$ (undirected) is ***bipartite*** if we can partition $V = L \cup R$ such that all edges in E go between L and R.



matching maximum matching

A ***matching*** is a subset of edges $M \subseteq E$ such that for all $v \in V$, ≤ 1 edge of M is incident on v. (Vertex v is ***matched*** if an edge of M is incident on it; otherwise ***unmatched***).

Maximum matching: a matching of maximum cardinality. (M is a maximum matching if $|M| \geq |M'|$ for all matchings M'.)

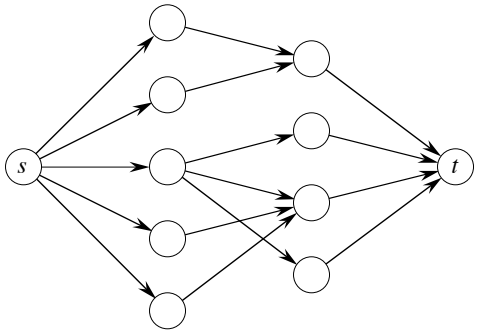
Problem: Given a bipartite graph (with the partition), find a maximum matching.

Application: Matching planes to routes.

- $L =$ set of planes.
- $R =$ set of routes.
- $(u, v) \in E$ if plane u can fly route v.
- Want maximum # of routes to be served by planes.

Given G, define flow network $G' = (V', E')$.

- $V' = V \cup \{s, t\}$.
- $E' = \{(s, u) : u \in L\}$
  $\cup \{(u, v) : u \in L, v \in R, (u, v) \in E\}$
  $\cup \{(v, t) : v \in R\}$.
- $c(u, v) = 1$ for all $(u, v) \in E'$.



Each vertex in V has ≥ 1 incident edge $\Rightarrow |E| \geq |V|/2$.

Therefore, $|E| \leq |E'| = |E| + |V| \leq 3|E|$.

Therefore, $|E'| = \Theta(E)$.

Find a max flow in G'. Book shows that it will have integer values for all (u, v).

Use edges that carry flow of 1 in matching.

Book proves this gives maximum matching.

Solutions for Chapter 26: Maximum Flow

Solution to Exercise 26.1-4

We want to prove the following lemma.

Lemma
For any flow f in $G = (V, E)$:

1. For all $X \subseteq V$, $f(X, X) = 0$,
2. For all $X, Y \subseteq V$, $f(X, Y) = -f(Y, X)$,
3. For all $X, Y, Z \subseteq V$ such that $X \cap Y = \emptyset$,
   $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ and
   $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.

Proof

2.

$$\begin{aligned} f(X, Y) &= \sum_{x \in X} \sum_{y \in Y} f(x, y) \\ &= \sum_{x \in X} \sum_{y \in Y} -f(y, x) \quad \text{(skew symmetry)} \\ &= \sum_{y \in Y} \sum_{x \in X} -f(y, x) \\ &= -f(Y, X) \end{aligned}$$

1.

$$\begin{aligned} f(X, X) &= -f(X, X) \\ \Rightarrow f(X, X) &= 0 \end{aligned}$$

3.

$$\begin{aligned} f(X \cup Y, Z) &= \sum_{v \in X \cup Y} \sum_{z \in Z} f(v, z) \\ &= \sum_{v \in X} \left(\sum_{z \in Z} f(v, z) \right) + \sum_{v \in Y} \left(\sum_{z \in Z} f(v, z) \right) \quad (X \cap Y = \emptyset) \\ &= f(X, Z) + f(Y, Z) \end{aligned}$$

The other part is symmetric. ■ (lemma)

Solution to Exercise 26.1-6

The flow sum $f_1 + f_2$ satisfies skew symmetry and flow conservation, but might violate the capacity constraint.

We give proofs for skew symmetry and flow conservation and an example that shows a violation of the capacity constraint. Let $f(u, v) = (f_1 + f_2)(u, v)$.

For skew symmetry:

$$\begin{aligned} f(u, v) &= f_1(u, v) + f_2(u, v) \\ &= -f_1(v, u) - f_2(v, u) \qquad \text{(skew symmetry)} \\ &= -(f_1(v, u) + f_2(v, u)) \\ &= -f(v, u) \ . \end{aligned}$$

For flow conservation, let $u \in V - \{s, t\}$:

$$\begin{aligned} \sum_{v \in V} f(u, v) &= \sum_{v \in V} (f_1(u, v) + f_2(u, v)) \\ &= \sum_{v \in V} f_1(u, v) + \sum_{v \in V} f_2(u, v) \\ &= 0 + 0 \qquad \text{(flow conservation)} \\ &= 0 \ . \end{aligned}$$

For the capacity constraint, let $V = \{s, t\}$, $E = \{(s, t)\}$, and $c(s, t) = 1$. Let $f_1(s, t) = f_2(s, t) = 1$. Then f_1 and f_2 obey the capacity constraint, but $(f_1 + f_2)(u, v) = 2$, which violates the capacity constraint.

Solution to Exercise 26.1-7

To see that the flows form a convex set, we show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α such that $0 \le \alpha \le 1$.

For the capacity constraint, first observe that $\alpha \le 1$ implies that $1 - \alpha \ge 0$. Thus, for any $u, v \in V$, we have

$$\begin{aligned} \alpha f_1(u, v) + (1 - \alpha) f_2(u, v) &\ge 0 \cdot f_1(u, v) + 0 \cdot (1 - \alpha) f_2(u, v) \\ &= 0 \ . \end{aligned}$$

Since $f_1(u, v) \le c(u, v)$ and $f_2(u, v) \le c(u, v)$, we also have

$$\begin{aligned} \alpha f_1(u, v) + (1 - \alpha) f_2(u, v) &\le \alpha c(u, v) + (1 - \alpha) c(u, v) \\ &= (\alpha + (1 - \alpha)) c(u, v) \\ &= c(u, v) \ . \end{aligned}$$

For skew symmetry, we have $f_1(u, v) = -f_1(v, u)$ and $f_2(u, v) = -f_2(v, u)$ for any $u, v \in V$. Thus, we have

$$\begin{aligned} \alpha f_1(u, v) + (1 - \alpha) f_2(u, v) &= -\alpha f_1(v, u) - (1 - \alpha) f_2(v, u) \\ &= -(\alpha f_1(v, u) + (1 - \alpha) f_2(v, u)) \ . \end{aligned}$$

For flow conservation, observe that since f_1 and f_2 obey flow conservation, we have $\sum_{v \in V} f_1(u, v) = 0$ and $\sum_{v \in V} f_2(u, v) = 0$ for any $u \in V - \{s, t\}$. Thus,

$$\begin{aligned}\sum_{v \in V}(\alpha f_1(u, v) + (1 - \alpha) f_2(u, v)) &= \alpha \sum_{v \in V} f_1(u, v) + (1 - \alpha) \sum_{v \in V} f_2(u, v) \\ &= \alpha \cdot 0 + (1 - \alpha) \cdot 0 \\ &= 0 .\end{aligned}$$

Solution to Exercise 26.1-9

Create a vertex for each corner, and if there is a street between corners u and v, create directed edges (u, v) and (v, u). Set the capacity of each edge to 1. Let the source be corner on which the professor's house sits, and let the sink be the corner on which the school is located. We wish to find a flow of value 2 that also has the property that $f(u, v)$ is an integer for all vertices u and v. Such a flow represents two edge-disjoint paths from the house to the school.

Solution to Exercise 26.2-4

$$\begin{aligned}c_f(u, v) + c_f(v, u) &= c(u, v) - f(u, v) + c(v, u) - f(v, u) \\ &\qquad \text{(by definition)} \\ &= c(u, v) + c(v, u) \\ &\qquad \text{(by skew symmetry: } f(u, v) = -f(v, u))\end{aligned}$$

Solution to Exercise 26.2-9

For any two vertices u and v in G, you can define a flow network G_{uv} consisting of the directed version of G with all edge capacities set to 1, $s = u$, and $t = v$. (G_{uv} has $O(V)$ vertices—actually, $|V|$—and $O(E)$ edges, as required. We want all capacities to be 1 so that the number of edges crossing a cut equals the capacity of the cut.) Let f_{uv} denote a maximum flow in G_{uv}.

We claim that for any $u \in V$, the edge connectivity k equals $\min_{v \in V - \{u\}} |f_{uv}|$. We'll show below that this claim holds. Assuming that it holds, we can find k as follows:

EDGE-CONNECTIVITY(G)
select any vertex $u \in V$
for each vertex $v \in V - \{u\}$ $\quad \triangleright$ $|V| - 1$ iterations
  do set up the flow network G_{uv} as described above
    find the maximum flow f_{uv} on G_{uv}
return the minimum of the $|V| - 1$ max-flow values: $\min_{v \in V - \{u\}} |f_{uv}|$

The claim follows from the max-flow min-cut theorem and how we chose capacities so that the capacity of a cut is the number of edges crossing it. We prove that $k = \min_{v \in V-\{u\}} |f_{uv}|$, for any $u \in V$ by showing separately that k is at least this minimum and that k is at most this minimum.

- Proof that $k \geq \min_{v \in V-\{u\}} |f_{uv}|$:

  Let $m = \min_{v \in V-\{u\}} |f_{uv}|$. Suppose we remove only $m - 1$ edges from G. For any vertex v, by the max-flow min-cut theorem, u and v are still connected. (The max flow from u to v is at least m, hence any cut separating u from v has capacity at least m, which means at least m edges cross any such cut. Thus at least one edge is left crossing the cut when we remove $m-1$ edges.) Thus every node is connected to u, which implies that the graph is still connected. So at least m edges must be removed to disconnect the graph—i.e., $k \geq \min_{v \in V-\{u\}} |f_{uv}|$.

- Proof that $k \leq \min_{v \in V-\{u\}} |f_{uv}|$:

  Consider a vertex v with the minimum $|f_{uv}|$. By the max-flow min-cut theorem, there is a cut of capacity $|f_{uv}|$ separating u and v. Since all edge capacities are 1, exactly $|f_{uv}|$ edges cross this cut. If these edges are removed, there is no path from u to v, and so our graph becomes disconnected. Hence $k \leq \min_{v \in V-\{u\}} |f_{uv}|$.

- Thus, the claim that $k = \min_{v \in V-\{u\}} |f_{uv}|$, for any $u \in V$ is true.

Solution to Exercise 26.2-10

From the time (u, v) is a critical edge until it is again a critical edge, $\delta(s, u)$ increases by at least 2, as shown in Theorem 26.9. Similarly, you can show that $\delta(v, t)$ also increases by at least 2. Thus the length of the augmenting path $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ increases by at least 4 between times (u, v) is critical. Since the length of an augmenting path cannot exceed $|V| - 1$, (u, v) can be critical $< |V|/4$ times.

Edmonds-Karp terminates when there are no more augmenting paths, so it must terminate when there are no more critical edges, which takes at most (# edges)·(max # times each edge critical) $< |E_f|\,(|V|/4)$ iterations. In general, $|E_f| \leq 2\,|E|$, so the number of iterations is at most (actually, fewer than) $|E|\,|V|/2$. But if we assume that G always has edges in both directions (i.e., $(u, v) \in E$ if and only if $(v, u) \in E$), then $|E_f| \leq |E|$, and the number of iterations is at most $|E|\,|V|/4$.

Solution to Exercise 26.3-3

By definition, an augmenting path is a simple path $s \rightsquigarrow t$ in the residual graph G_f. Since G has no edges between vertices in L and no edges between vertices in R, neither does the flow network G' and hence neither does G'_f. Also, the only edges

involving s or t connect s to L and R to t. Note that although edges in G can go only from L to R, edges in G'_f can also go from R to L.

Thus any augmenting path must go

$$s \to L \to R \to \cdots \to L \to R \to t \ ,$$

crossing back and forth between L and R at most as many times as it can do so without using a vertex twice. It contains s, t, and equal numbers of distinct vertices from L and R—at most $2 + 2 \cdot \min(|L|, |R|)$ vertices in all. The length of an augmenting path (i.e., its number of edges) is thus bounded above by $2 \cdot \min(|L|, |R|) + 1$.

Solution to Exercise 26.4-2

Each time we call RELABEL(u), we examine all edges $(u, v) \in E_f$. Since the number of relabel operations is at most $2|V| - 1$ per vertex, edge (u, v) will be examined during relabel operations at most $4|V| - 2 = O(V)$ times (at most $2|V| - 1$ times during calls to RELABEL(u) and at most $2|V| - 1$ times during calls to RELABEL(v)). Summing up over all the possible residual edges, of which there are at most $2|E| = O(E)$, we see that the total time spent relabeling vertices is $O(VE)$.

Solution to Exercise 26.4-3

We can find a minimum cut, given a maximum flow found in $G = (V, E)$ by a push-relabel algorithm, in $O(V)$ time. First, find a height $\widehat{h}$ such that $0 < \widehat{h} < |V|$ and there is no vertex whose height equals $\widehat{h}$ at termination of the algorithm. Since $h[s] = |V|$ and $h[t] = 0$, we need consider only $|V| - 2$ vertices. Since there are $|V| - 1$ possible values for $\widehat{h}$, we know that for at least one number in $1, 2, \ldots, |V| - 1$, there will be no vertex of that height. Hence, $\widehat{h}$ is well defined, and it is easy to find in $O(V)$ time by using a simple boolean array indexed by heights $1, 2, \ldots, |V| - 1$.

Let $S = \{u \in V : h[u] > \widehat{h}\}$ and $T = \{v \in V : h[v] < \widehat{h}\}$. Because $h[s] = |V| > \widehat{h}$, we have $s \in S$, and because $h[t] = 0 < \widehat{h}$, we have $t \in T$, as required for a cut.

We need to show that $f(u, v) = c(u, v)$, i.e., that $(u, v) \notin E_f$ for all $u \in S$ and $v \in T$. Once we do that, we have that $f(S, T) = c(S, T)$, and by Corollary 26.6, (S, T) is a minimum cut.

Suppose for the purpose of contradiction that there exist vertices $u \in S$ and $v \in T$ such that $(u, v) \in E_f$. Because h is always maintained as a height function (Lemma 26.17), we have that $h[u] \le h[v] + 1$. But we also have $h[v] < \widehat{h} < h[u]$, and because all values are integer, $h[v] \le h[u] - 2$. Thus, we have $h[u] \le h[v] + 1 \le h[u] - 2 + 1 = h[u] - 1$, which gives the contradiction that $0 \le -1$. Thus, (S, T) is a minimum cut.

Solution to Exercise 26.4-6

If we set $h[s] = |V| - 2$, we have to change our definition of a height function to allow $h[s] = |V| - 2$, rather than $h[s] = |V|$. The only change we need to make to the proof of correctness is to update the proof of Lemma 26.18. The original proof derives the contradiction that $h[s] \le k < |V|$, which is at odds with $h[s] = |V|$. When $h[s] = |V| - 2$, there is no contradiction.

As in the original proof, let us suppose that we have a simple augmenting path $\langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = t$, so that $k < |V|$. How could (s, v_1) be a residual edge? It had been saturated in INITIALIZE-PREFLOW, which means that we had to have pushed some flow from v_1 to s. In order for that to have happened, we must have had $h[v_1] = h[s] + 1$. If we set $h[s] = |V| - 2$, that means that $h[v_1]$ was $|V| - 1$ at the time. Since then, $h[v_1]$ did not decrease, and so we have $h[v_1] \ge |V| - 1$. Working backwards over our augmenting path, we have $h[v_{k-i}] \le h[t] + i$ for $i = 0, 1, \ldots, k$. As before, because the augmenting path is simple, $k < |V|$. Letting $i = k - 1$, we have $h[v_1] \le h[t] + k - 1 < 0 + |V| - 1$. We now have the contradiction that $h[v_1] \ge |V| - 1$ and $h[v_1] < |V| - 1$, which shows that Lemma 26.18 still holds.

Nothing in the analysis changes asymptotically.

Solution to Problem 26-2

a. The idea is to use a maximum-flow algorithm to find a maximum bipartite matching that selects the edges to use in a minimum path cover. We must show how to formulate the max-flow problem and how to construct the path cover from the resulting matching, and we must prove that the algorithm indeed finds a minimum path cover.

Define G' as suggested, with directed edges. Make G' into a flow network with source x_0 and sink y_0 by defining all edge capacities to be 1. G' is the flow network corresponding to a bipartite graph G'' in which $L = \{x_1, \ldots x_n\}$, $R = \{y_1, \ldots y_n\}$, and the edges are the (undirected version of the) subset of E that doesn't involve x_0 or y_0.

The relationship of G to the bipartite graph G'' is that every vertex i in G is represented by two vertices, x_i and y_i, in G''. Edge (i, j) in G corresponds to edge (x_i, y_j) in G''. That is, an edge (x_i, y_j) in G'' means that an edge in G leaves i and enters j. x_i tells us about edges leaving i and y_i tells us about edges entering i.

The edges in a bipartite matching in G'' can be used in a path cover of G, because:

- In a bipartite matching, no vertex is used more than once. In a bipartite matching in G'', the fact that no x_i is used more than once means that at most one edge in the matching leaves any vertex i in G, and similarly the fact that

no y_i is used more than once means that at most one edge in the matching enters any vertex i in G.

- In a path cover, no vertex appears in more than one path, hence at most one path edge enters each vertex and at most one path edge leaves each vertex.

We can construct a path cover P from any bipartite matching M (not just a maximum matching) by moving from some x_i to the matching y_j (if any), then from x_j to its matching y_k, and so on, as follows:

1. Start a new path containing a vertex i that has not yet been placed in a path.
2. If x_i is unmatched, the path can't go any farther; just add it to P.
3. If x_i is matched to some y_j, add j to the current path. If j has already been placed in a path (i.e., though we've just entered j by processing y_j, we've already built a path that leaves j by processing x_j), combine this path with that one and go back to step 1. Otherwise go to step 2 to process x_j.

This algorithm constructs a path cover because:

- Every vertex is put into some path, because we keep picking an unused vertex from which to start a path until there are no unused vertices.
- No vertex is put into two paths, because every x_i is matched to at most one y_j, and vice versa. That is, at most one candidate edge leaves each vertex and at most one candidate edge enters each vertex. The normal path-building starts at or enters a vertex and then leaves it, building a single path. If we ever enter a vertex that was left earlier, it must have been the start of another path, since there are no cycles, and we combine those paths so that the vertex is entered and left on a single path.

Every edge in M is used in some path because we visit every x_i, and we incorporate the single edge, if any, from each visited x_i. Thus there is a one-to-one correspondence between edges in the matching and edges in the constructed path cover.

We now show that the path cover P constructed above has the fewest possible paths when the matching is maximum.

Let f be the flow corresonding to the bipartite matching M.

$$\begin{aligned}
|V| &= \sum_{p \in P} (\#\text{ vertices in } p) && \text{(every vertex is on exactly 1 path)} \\
&= \sum_{p \in P} (1 + \#\text{ edges in } p) \\
&= \sum_{p \in P} 1 + \sum_{p \in P} (\#\text{ edges in } p) \\
&= |P| + (\#\text{ edges in } M) && \text{(by 1-to-1 correspondence)} \\
&= |P| + |f| && \text{(Lemma 26.10)} \ .
\end{aligned}$$

Thus for the fixed set V in our graph G, $|P|$ (the number of paths) is minimized when the flow f is maximized.

Thus the overall algorithm is as follows:

- Use FORD-FULKERSON to find a maximum flow in G', hence a maximum bipartite matching M in G''.
- Construct the path cover as described above.

Time: $O(VE)$ total

- $O(V + E)$ to set up G'
- $O(VE)$ to find the maximum bipartite matching
- $O(E)$ to trace the paths, because each edge $\in M$ is traversed only once and there are $O(E)$ edges in M.

b. The algorithm does not work if there are cycles.

Consider a graph G with 4 vertices, consisting of a directed triangle and an edge pointing to the triangle:

$E = \{(1, 2), (2, 3), (3, 1), (4, 1)\}$

G can be covered with a single path: $4 \to 1 \to 2 \to 3$, but our algorithm might find only a 2-path cover.

In the bipartite graph G', the edges (x_i, y_j) are

$(x_1, y_2), (x_2, y_3), (x_3, y_1), (x_4, y_1)$.

There are 4 edges from an x_i to a y_j, but 2 of them lead to y_1, so a maximum bipartite matching can have only 3 edges (and the maximum flow in G is 3). In fact, there are 2 possible maximum matchings. It is always possible to match $x_1 \to y_2$ and $x_2 \to y_3$, but then either $x_3 \to y_1$ or $x_4 \to y_1$ can be chosen, but not both.

The maximum-flow found by one of our max-flow algorithms could find the flow corresponding to either of these matchings, since both are maximal. But one of the matchings doesn't contain an edge to or from vertex 4, so given that matching, our path algorithm is forced to produce 2 paths, one of which contains just the vertex 4.

Solution to Problem 26-4

a. Just execute one iteration of the Ford-Fulkerson algorithm. The edge (u, v) in E with increased capacity ensures that the edge (u, v) is in the residual graph. So look for an augmenting path and update the flow if a path is found.

Time: $O(V + E) = O(E)$ if we find the augmenting path with either depth-first or breadth-first search.

To see that only one iteration is needed, consider separately the cases in which (u, v) is or is not an edge that crosses a minimum cut. If (u, v) does not cross a minimum cut, then increasing its capacity does not change the capacity of any minimum cut, and hence the value of the maximum flow does not change. If (u, v) does cross a minimum cut, then increasing its capacity by 1 increases the capacity of that minimum cut by 1, and hence possibly the value of the maximum flow by 1. In this case, there is either no augmenting path (in which case there was some other minimum cut that (u, v) does not cross), or the augmenting path increases flow by 1. No matter what, one iteration of Ford-Fulkerson suffices.

b. Let f be the maximum flow before reducing $c(u, v)$.

If $f(u, v) = 0$, we don't need to do anything.

If $f(u, v) > 0$, we will need to update the maximum flow. Assume from now on that $f(u, v) > 0$, which in turn implies that $f(u, v) \geq 1$.

Define $f'(x, y) = f(x, y)$ for all $x, y \in V$, except that $f'(u, v) = f(u, v) - 1$. Although f' obeys all capacity contraints, even after $c(u, v)$ has been reduced, it is not a legal flow, as it violates skew symmetry and flow conservation at u and v. f' has one more unit of flow entering u than leaving u, and it has one more unit of flow leaving v than entering v.

The idea is to try to reroute this unit of flow so that it goes out of u and into v via some other path. If that is not possible, we must reduce the flow from s to u and from v to t by one unit.

Look for an augmenting path from u to v (note: *not* from s to t).

- If there is such a path, augment the flow along that path.
- If there is no such path, reduce the flow from s to u by augmenting the flow from u to s. That is, find an augmenting path $u \rightsquigarrow s$ and augment the flow along that path. (There definitely is such a path, because there is flow from s to u.) Similarly, reduce the flow from v to t by finding an augmenting path $t \rightsquigarrow v$ and augmenting the flow along that path.

Time: $O(V + E) = O(E)$ if we find the paths with either DFS or BFS.

Solution to Problem 26-5

a. The capacity of a cut is defined to be the sum of the capacities of the edges crossing it. Since the number of such edges is at most $|E|$, and the capacity of each edge is at most C, the capacity of *any* cut of G is at most $C|E|$.

b. The capacity of an augmenting path is the minimum capacity of any edge on the path, so we are looking for an augmenting path whose edges *all* have capacity at least K. Do a breadth-first search or depth-first-search as usual to find the path, considering only edges with residual capacity at least K. (Treat lower-capacity edges as though they don't exist.) This search takes $O(V + E) = O(E)$ time. (Note that $|V| = O(E)$ in a flow network.)

c. MAX-FLOW-BY-SCALING uses the Ford-Fulkerson method. It repeatedly augments the flow along an augmenting path until there are no augmenting paths of capacity greater ≥ 1. Since all the capacities are integers, and the capacity of an augmenting path is positive, this means that there are no augmenting paths whatsoever in the residual graph. Thus, by the max-flow min-cut theorem, MAX-FLOW-BY-SCALING returns a maximum flow.

d. • The first time line 4 is executed, the capacity of any edge in G_f equals its capacity in G, and by part (a) the capacity of a minimum cut of G is at most $C|E|$. Initially $K = 2^{\lfloor \lg C \rfloor}$, hence $2K = 2 \cdot 2^{\lfloor \lg C \rfloor} = 2^{\lfloor \lg C \rfloor + 1} > 2^{\lg C} = C$. So the capacity of a minimum cut of G_f is initially less than $2K|E|$.

- The other times line 4 is executed, K has just been halved, so the capacity of a cut of G_f is at most $2K\,|E|$ at line 4 if and only if that capacity was at most $K\,|E|$ when the **while** loop of lines 5–6 last terminated. So we want to show that when line 7 is reached, the capacity of a minimum cut of G_f is most $K\,|E|$.

  Let G_f be the residual network when line 7 is reached.

  There is no augmenting path of capacity $\geq K$ in G_f
  $\Rightarrow$ max flow f' in G_f has value $|f'| < K\,|E|$
  $\Rightarrow$ min cut in G_f has capacity $< K\,|E|$

e. By part (d), when line 4 is reached, the capacity of a minimum cut of G_f is at most $2K\,|E|$, and thus the maximum flow in G_f is at most $2K\,|E|$.

By an extension of Lemma 26.2, the value of the maximum flow in G equals the value of the current flow in G plus the value of the maximum flow in G_f. (Lemma 26.2 shows that, given a flow f in G, every flow f' in G_f induces a flow $f + f'$ in G; the reverse claim, that every flow $f + f'$ in G induces a flow f' in G_f, is proved in a similar manner. Together these claims provide the necessary correspondence between a maximum flow in G and a maximum flow in G_f.) Therefore, the maximum flow in G is at most $2K\,|E|$ more than the current flow in G. Every time the inner **while** loop finds an augmenting path of capacity at least K, the flow in G increases by $\geq K$. Since the flow cannot increase by more than $2K\,|E|$, the loop executes at most $(2K\,|E|)/K = 2\,|E|$ times.

f. The time complexity is dominated by the loop of lines 4–7. (The lines outside the loop take $O(E)$ time.) The outer **while** loop executes $O(\lg C)$ times, since K is initially $O(C)$ and is halved on each iteration, until $K < 1$. By part (e), the inner **while** loop executes $O(E)$ times for each value of K; and by part (b), each iteration takes $O(E)$ time. Thus, the total time is $O(E^2 \lg C)$.

Lecture Notes for Chapter 27: Sorting Networks

Chapter 27 overview

Sorting networks

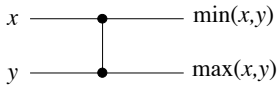
An example of parallel algorithms.

We'll see how, if we allow a certain kind of parallelism, we can sort in $O(\lg^2 n)$ "time."

Along the way, we'll see the 0-1 principle, which is a great way to prove the correctness of any comparison-based sorting algorithm.

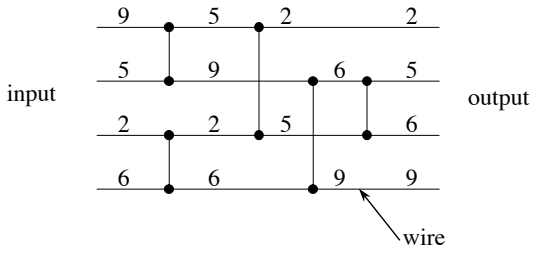
Comparison networks

Comparator



Works in $O(1)$ time.

Comparison network



Wires go straight, left to right.

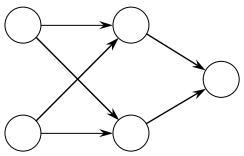
Each comparator has inputs/outputs on some pair of wires.

Claim that this comparison network will sort any set of 4 input values:

- After leftmost comparators, minimum is on either wire 1 (from top) or 3, maximum is on either wire 2 or 4.
- After next 2 comparators, minimum is on wire 1, maximum on wire 4.
- Last comparator gets correct values onto wires 2 and 3.

Running time = ***depth*** = longest path of comparators. (3 in previous example.)

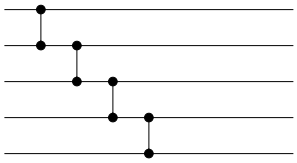
- Think of dag of comparators that depend on each other. Depth = longest path through dag (counting vertices, not edges).



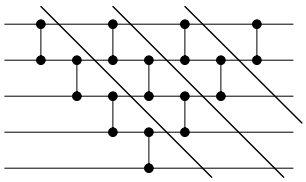
- Depth $\neq$ max # of comparators attached to a single wire.
  - In the above example, that is 2.

Selection sorter

To find max of 5 values:



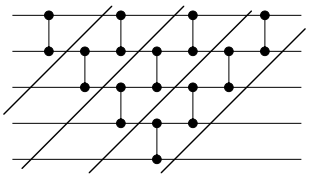
Can repeat, decreasing # of values:



$$\begin{aligned}
\text{Depth: } D(n) &= D(n-1)+2 \\
D(2) &= 1 \\
\Rightarrow D(n) &= 2n-3 \\
&= \Theta(n)\ .
\end{aligned}$$

If view depth as "time," parallelism gets us a faster method than any sequential comparison sort!

Can view the same network as insertion sort:



[This material answers Exercise 27.1-6, showing that the network in Figure 27.3 does correctly sort and showing its relationship to insertion sort.]

Zero-one principle

How can we test if a comparison network sorts?

- We could try all $n!$ permutations of input.
- But we need to test only 2^n permutations. This is *many* fewer than all $n!$ permutations.

Theorem (0-1 principle)
If a comparison network with n inputs sorts all 2^n sequences of 0's and 1's, then it sorts all sequences of arbitrary numbers.

Note: In practice, we don't even have to reason about "all 2^n sequences"—instead, we look at the patterns of 0's and 1's—we'll see later how.

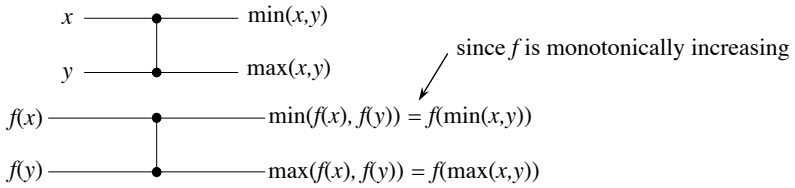
Lemma
If a comparison network transforms

$a = \langle a_1, a_2, \ldots, a_n \rangle$ into $b = \langle b_1, b_2, \ldots, b_n \rangle$,

then for any monotonically increasing function f, it transforms

$f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ into $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_m) \rangle$.

Sketch of proof

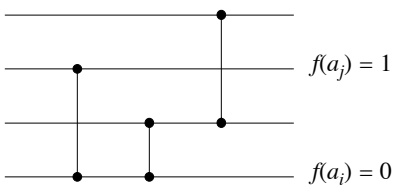


Then use induction on comparator depth. ■ (lemma)

Proof (of 0-1 principle)
Suppose that the principle is not true, so that an n-input comparison network sorts all 0-1 sequences, but there is a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ such that $a_i < a_j$ but a_i comes *after* a_j in the output.
Define the monotonically increasing function

$$f(x) = \begin{cases} 0 & \text{if } x \le a_i \ , \\ 1 & \text{if } x > a_i \ . \end{cases}$$

By the lemma, if we give the input $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$, then the output will have $f(a_i)$ after $f(a_j)$:



But that's a 0-1 sequence that is sorted incorrectly, a contradiction. ■ (theorem)

A bitonic sorting network

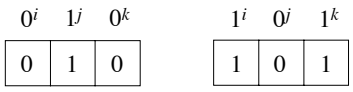
Constructing a sorting network

Step 1: Construct a "bitonic sorter." It sorts any bitonic sequence.

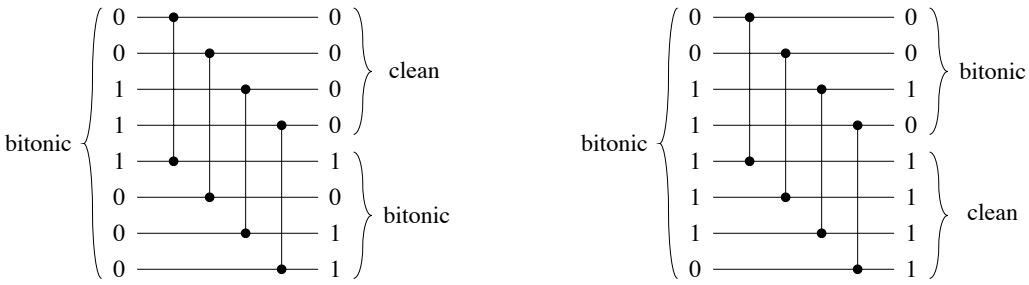
A sequence is ***bitonic*** if it monotonically increases, then monotonically decreases, or it can be circularly shifted to become so.

Examples: $\langle 1, 3, 7, 4, 2 \rangle$
$\langle 6, 8, 3, 1, 2, 4 \rangle$
$\langle 8, 7, 2, 1, 3, 5 \rangle$
Any sequence of 1 or 2 numbers

For 0-1 sequences—which we can focus on—bitonic sequences have the form



Half-cleaner:



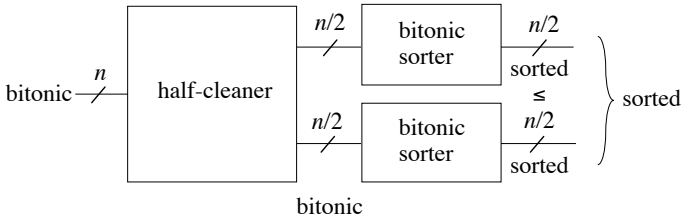
Depth = 1.

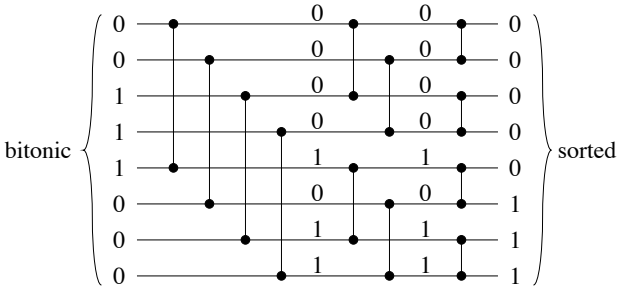
Lemma
If the input to a half-cleaner is a bitonic 0-1 sequence, then for the output:

- both the top and bottom half are bitonic,
- every element in the top half is $\leq$ every element in the bottom half, and
- at least one of the halves is ***clean***—all 0's or all 1's.

Skipping proof—see book (not difficult at all).

Bitonic sorter:





Depth: $D(n) = D(n/2) + 1$

$D(2) = 1$

$\Rightarrow D(n) = \lg n$.

Step 2: Construct a merging network.

It merges 2 sorted sequences.
Adapt a half-cleaner.

Idea: Given 2 sorted sequences, reverse the second one, then concatenate with the first one $\Rightarrow$ get a bitonic sequence.

Example:

$X = 0011 \qquad Y = 0111$

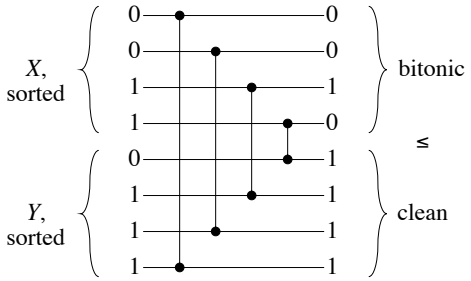
$Y^R = 1110$

$XY^R = 00111110$ (bitonic)

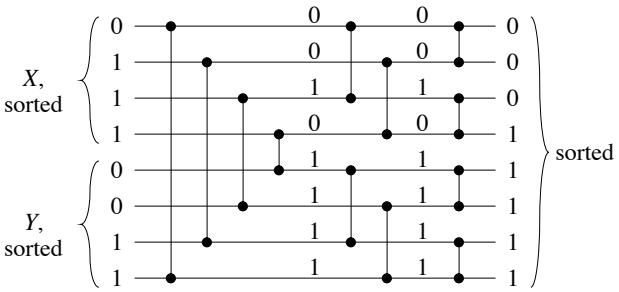
So, we can merge X and Y by doing a bitonic sort on X and Y^R.

How to reverse Y? Don't!

Instead, reverse the bottom half of the connections of the first half-cleaner:



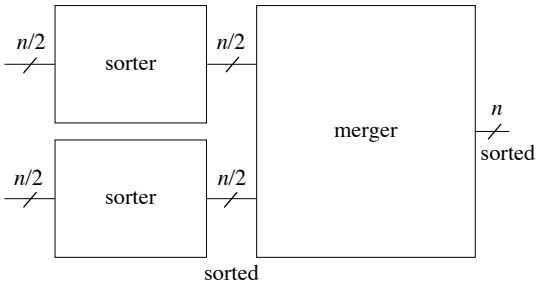
Full merging network:

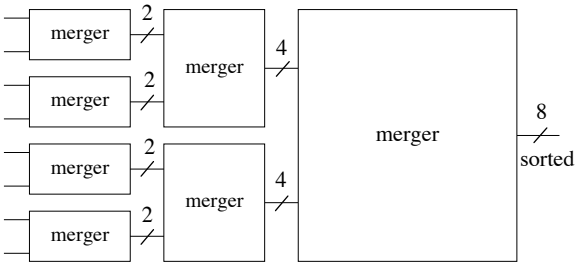


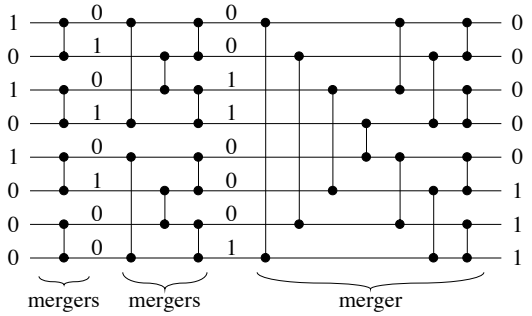
Depth is same as bitonic sorter: $\lg n$.

Step 3: Construct a sorting network.

Recursive merging—like merge sort, bottom-up:







Depth: $D(n) = D(n/2) + \lg n$

$D(2) = 1$

$\Rightarrow D(n) = \Theta(\lg^2 n)$ (Exercise 4.4-2) .

Use 0-1 principle to prove that this sorts all inputs.

Can we do better?

Yes—the AKS network has depth $O(\lg n)$.

- Huge constant—over 1000.
- Really hard to construct.
- Highly impractical—of theoretical interest only.

Solutions for Chapter 27: Sorting Networks

Solution to Exercise 27.1-4

Consider any input element x. After 1 level of the network, x can be in at most 2 different places, in at most 4 places after 2 levels, and so forth. Thus we need at least $\lg n$ depth to be able to move x to the right place, which could be any of the n ($= 2^{\lg n}$) outputs.

Solution to Exercise 27.1-5

Simulation of any sorting network on a serial machine is a comparison sort, hence there are $\Omega(n \lg n)$ comparisons/comparators. Intuitively, since the depth is $\Omega(\lg n)$ and we can perform at most $n/2$ comparisons at each depth of the network, this $\Omega(n \lg n)$ bound makes sense.

Solution to Exercise 27.1-7

We take advantage of the comparators appearing in sorted order within the network in the following pseudocode.

for $i \leftarrow 1$ **to** n
    do $d[i] \leftarrow 0$
for each comparator (i, j) in the list of comparators
    do $d[i] \leftarrow d[j] \leftarrow \max(d[i], d[j]) + 1$
return $\max_{1 \le i \le n} d[i]$

This algorithm implicitly finds the longest path in a dag of the comparators (in which an edge connects each comparator to the comparators that need its outputs). Even though we don't explicitly construct the dag, the above sort produces a topological sort of the dag.

The first **for** loop takes $\Theta(n)$ time, the second **for** loop takes $\Theta(c)$ time, and computing the maximum $d[i]$ value in the **return** statement takes $\Theta(n)$ time, for a total of $\Theta(n + c) = O(n + c)$ time.

Solution to Exercise 27.2-2

In both parts of the proof, we will be using a set $\{f_1, f_2, \ldots, f_{n-1}\}$ of monotonically increasing functions, where

$$f_k(x) = \begin{cases} 0 & \text{if } x \le k\ , \\ 1 & \text{if } x > k\ . \end{cases}$$

For convenience, let us also define the sequences $s_1, s_2, \ldots, s_{n-1}$, where s_i is the sequence consisting of $n - i$ 1's followed by i 0's.

$\Rightarrow$: Assume that the sequence $\langle n, n-1, \ldots, 1\rangle$ is correctly sorted by the given comparison network. Then by Lemma 27.1, we know that applying any monotonically increasing function to the sequence $s = \langle n, n-1, \ldots, 1\rangle$ produces a sequence that is also correctly sorted by the given comparison network. For $k = 1, 2, \ldots, n-1$, when we apply the monotonically increasing function f_k to the sequence s, the resulting sequence is s_k, which is correctly sorted by the comparison network.

$\Leftarrow$: Now assume that the comparison network fails to correctly sort the input sequence $\langle n, n-1, \ldots, 1\rangle$. Then there are elements i and j in this sequence for which $i < j$ but i appears after j in the output sequence. Consider the input sequence $\langle f_i(n), f_i(n-1), \ldots, f_i(1)\rangle$, which is the same as the sequence s_i. By Lemma 27.1, the network produces an output sequence in which $f_i(i)$ appears after $f_i(j)$. But $f_i(i) = 0$ and $f_i(j) = 1$, and so the network fails to sort the input sequence s_i.

Solution to Exercise 27.5-1

SORTER$[n]$ consists of $(n/4)\lg^2 n + (n/4)\lg n = \Theta(n \lg^2 n)$ comparators. To see this result, we first note that MERGER$[n]$ consists of $(n/2)\lg n$ comparators, since it has $\lg n$ levels, each with $n/2$ comparators.

If we denote the number of comparators in SORTER$[n]$ by $C(n)$, we have the recurrence

$$C(n) = \begin{cases} 0 & \text{if } n = 1\ , \\ 2C(n/2) + \dfrac{n}{2}\lg n & \text{if } n = 2^k \text{ and } k \ge 1\ . \end{cases}$$

We prove that $C(n) = (n/4)\lg^2 n + (n/4)\lg n$ by induction on k.

Basis: When $k = 0$, we have $n = 1$. Then $(n/4)\lg^2 n + (n/4)\lg n = 0 = C(n)$.

Inductive step: Assume that the inductive hypothesis holds for $k - 1$, so that $C(n/2) = (n/8)\lg^2(n/2) + (n/8)\lg(n/2) = (n/8)(\lg n - 1)^2 + (n/8)(\lg n - 1)$. We have

$$\begin{aligned}
C(n) &= 2C(n/2) + \frac{n}{2}\lg n \\
&= 2\left(\frac{n}{8}(\lg n - 1)^2 + \frac{n}{8}(\lg n - 1)\right) + \frac{n}{2}\lg n \\
&= \frac{n}{4}\lg^2 n - \frac{n}{2}\lg n + \frac{n}{4} + \frac{n}{4}\lg n - \frac{n}{4} + \frac{n}{2}\lg n \\
&= \frac{n}{4}\lg^2 n + \frac{n}{4}\lg n \; .
\end{aligned}$$

Solution to Exercise 27.5-2

We show by substitution that the recurrence for the depth of SORTER$[n]$,

$$D(n) = \begin{cases} 0 & \text{if } n = 1 \; , \\ D(n/2) + \lg n & \text{if } n = 2^k \text{ and } k \geq 1 \; , \end{cases}$$

has the solution $D(n) = (\lg n)(\lg n + 1)/2$.

Basis: When $k = 0$, we have $n = 1$. Then $(\lg n)(\lg n + 1)/2 = 0 = D(1)$.

Inductive step: Assume that the inductive hypothesis holds for $k - 1$, so that $D(n/2) = (\lg(n/2))(\lg(n/2) + 1)/2 = (\lg n - 1)(\lg n)/2$. We have

$$\begin{aligned}
D(n) &= D(n/2) + \lg n \\
&= \frac{(\lg n - 1)(\lg n)}{2} + \lg n \\
&= \frac{\lg^2 n - \lg n}{2} + \lg n \\
&= \frac{\lg^2 n + \lg n}{2} \\
&= \frac{(\lg n)(\lg n + 1)}{2} \; .
\end{aligned}$$

Index

This index covers exercises and problems from the textbook that are solved in this manual. The first page in the manual that has the solution is listed here.