Instructor's Manual

by Thomas H. Cormen

to Accompany

Introduction to Algorithms

Fourth Edition

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

The MIT Press
Cambridge, Massachusetts London, England

Instructor’s Manual to Accompany *Introduction to Algorithms*, Fourth Edition
by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Contents

Revision History

Revisions to the lecture notes and solutions are listed by date rather than being numbered.

- 14 March 2022. Initial release.

Preface

This document is an instructor's manual to accompany *Introduction to Algorithms*, Fourth 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.

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. Future revisions of this document may include additional material. There are two reasons that we have not included solutions to all exercises and problems. 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 much longer than the text itself.

We have numbered the pages using the format *CC-PP*, where *CC* is a chapter number of the text and *PP* is the page number within that chapter. The *PP* numbers restart from 1 at the beginning of each chapter. We chose this form of page numbering so that if we add or change material, the only pages whose numbering is affected are those 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; 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, COSC 31.
- Some are written just for this document.

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.

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 replicate slides containing this material so that you can easily reprise 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.

Pseudocode in this document omits line numbers, which are inconvenient to include when writing pseudocode on the board. 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 index lists all the exercises and problems for the included 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 document. We apologize for this inconvenience.

You can use the clrscode4e package for $\LaTeX\,2_\varepsilon$ to typeset pseudocode in the same way that we do. You can find it at https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/11599/clrscode4e.sty and its documentation at https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/11599/clrscode4e.pdf. Make sure to use the clrscode4e package, not the clrscode or clrscode3e packages, which are for earlier editions of the book.

Reporting errors and suggestions

Undoubtedly, this document contains errors. Please report errors by sending email to clrs-manual-bugs@mit.edu.

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, https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/11599/e4-bugs.html, before you submit it. You also can use the MIT Press web site for the text, https://mitpress.mit.edu/books/introduction-algorithms-fourth-edition, 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 document and the text.

How we produced this document

Like the fourth edition of *Introduction to Algorithms*, this document was produced in LaTeX 2_{ε}. We used the Times font with mathematics typeset using the MathTime Pro 2 fonts. As in all four editions of the textbook, we compiled the index using Windex, a C program that we wrote. We drew the illustrations using MacDraw Pro, with some of the mathematical expressions in illustrations laid in with the psfrag package for LaTeX 2_{ε}. We created the PDF files for this document on a MacBook Pro running OS 12.2.1.

Acknowledgments

This document borrows heavily from the manuals for the first three editions. Julie Sussman, P.P.A., wrote the first-edition manual. 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 subsequent editions of the book. Charles Leiserson also put in large amounts of time working with Julie on the first-edition manual.

The manual for the second edition was written by Tom Cormen, Clara Lee, and Erica Lin. Clara and Erica were undergraduate computer science majors at Dartmouth at the time, and they did a superb job.

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. Several of the solutions to new exercises and problems in the third edition were written by Sharath Gururaj and Priya Natarajan. Neerja Thakkar contributed many lecture notes and solutions for the fourth edition manual.

We also thank the MIT Press and our editors, Marie Lee and Elizabeth Swayze, for moral and financial support.

THOMAS H. CORMEN
Lebanon, New Hampshire
March 2022

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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

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++, Java, Python, JavaScript, and many other frequently used programming languages. 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

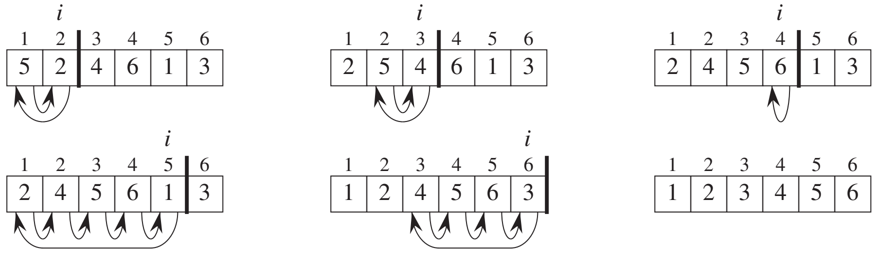
We use a procedure INSERTION-SORT.

- Takes as parameters an array $A[1:n]$ and the length n of the array.
- We use ":" to denote a range or subarray within an array. The notation $A[i:j]$ denotes the $j-i+1$ array elements $A[i]$ through and including $A[j]$. *[Note that the meaning of our subarray notation differs from its meaning in Python. In Python, $A[i:j]$ denotes the $j-i$ array elements $A[i]$ through $A[j-1]$, but does not include $A[j]$. Furthermore, in Python, negative indices count from the end. We do not use negative indices.]*
- *[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++, Java, Python, or JavaScript, which use 0-origin indexing, you need to be careful to get the indices right. One option is to adjust all index calculations 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. We are always clear about the bounds of an array, so that students know whether it's 0-origin or 1-origin indexed.]*
- 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, n)                                        cost    times
  for $i = 2$ to $n$                                        $c_1$   $n$
      $key = A[i]$                                          $c_2$   $n - 1$
      // Insert $A[i]$ into the sorted subarray $A[1:i-1]$. 0       $n - 1$
      $j = i - 1$                                           $c_4$   $n - 1$
      while $j > 0$ and $A[j] > key$                        $c_5$   $\sum_{i=2}^{n} t_i$
          $A[j+1] = A[j]$                                   $c_6$   $\sum_{i=2}^{n} (t_i - 1)$
          $j = j - 1$                                       $c_7$   $\sum_{i=2}^{n} (t_i - 1)$
      $A[j+1] = key$                                        $c_8$   $n - 1$
```

[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 i indicated. i indexes the "current card" being inserted into the hand. Elements to the left of $A[i]$ that are greater than $A[i]$ move one position to the right, and $A[i]$ moves into the evacuated position. The heavy vertical lines separate the part of the array in which an iteration works—$A[1:i]$—from the part of the array that is unaffected by this iteration—$A[i+1:n]$. 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 i—the subarray $A[1:i-1]$ consists of the elements originally in $A[1:i-1]$ 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: The loop terminates, and when it does, 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, $i = 2$. The subarray $A[1:i-1]$ 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 $A[i-1]$, $A[i-2]$, $A[i-3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[i]$) is found. At that point, the value of *key* is placed into this position.

Termination: The outer **for** loop starts with $i = 2$. Each iteration increases i by 1. The loop ends when $i > n$, which occurs when $i = n + 1$. Therefore, the loop terminates and $i - 1 = n$ at that time. Plugging n in for $i - 1$ 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

[See book pages 21–24 for more detail.]

- Indentation indicates block structure. Saves space and writing time. *[Readers are sometimes confused by how we indent **if**-**else** statements. We indent **else** at the same level as its matching **if**. The first executable line of an **else** clause appears on the same line as the keyword **else**. Multiway tests use **elseif** for tests after the first one. When an **if** statement is the first line in an **else** clause, it appears on the line following **else** to avoid it being misconstrued as **elseif**.]*
- Looping constructs are like in C, C++, Java, Python, and JavaScript. We assume that the loop variable in a **for** loop is still defined when the loop exits and has the value it had that caused the loop to terminate (such as $i = n + 1$ in INSERTION-SORT.)
- **//** indicates that the remainder of the line is a comment.
- Variables are local, unless otherwise specified.
- We often use ***objects***, which have ***attributes***. For an attribute *attr* of object x, we write $x.attr$. (This notation matches $x.attr$ in many object-oriented languages and is equivalent to x->*attr* in C++.) Attributes can cascade, so that if $x.y$ is an object and this object has attribute *attr*, then $x.y.attr$ indicates this object's attribute. That is, $x.y.attr$ is implicitly parenthesized as $(x.y).attr$.

- Objects are treated as references, like in most object-oriented languages. If x and y denote objects, then the assignment $y = x$ 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. 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.
- **return** statements are allowed to return multiple values to the caller (as Python can do with tuples).
- 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.)
- **error** means that conditions were wrong for the procedure to be called. The procedure immediately terminates. The caller is responsible for handling the error. This situation is somewhat like an exception in many programming languages, but we do not want to get into the details of handling exceptions.

Analyzing algorithms

We want to predict the resources that the algorithm requires. Usually, running time.

Why analyze?

Why not just code up the algorithm, run the code, and time it?

Because that would tell you how long the code takes to run

- on your particular computer,
- on that particular input,
- with your particular implementation,
- using your particular compiler or interpreter,
- with the particular libraries linked in,
- with the particular background tasks running at the time.

You wouldn't be able to predict how long the code would take on a different computer, with a different input, if implemented in a different programming language, etc.

Instead, devise a formula that characterizes the running time.

Random-access machine (RAM) model

In order to predict resource requirements, we need a computational 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. Ignore memory hierarchy (cache and virtual memory).

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 k takes the same amount of time c_k.
- 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 kth line takes time c_k, which is a constant. (Since the third line is a comment, it takes no time.)
- For $i = 2, 3, \ldots, n$, let t_i be the number of times that the **while** loop test is executed for that value of i.
- 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.

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

The running time depends on the values of t_i. 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 $i = i - 1$).
- All t_i 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_k) $\Rightarrow$ $T(n)$ 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 ith position ⇒ compare with $i - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $i - 1$ tests ⇒ $t_i = i$.
- $\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i$ and $\sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i - 1)$.
- $\sum_{i=1}^{n} i$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\dfrac{n(n+1)}{2}$.
- Since $\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1$, 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 $l = i - 1$, we see that $\sum_{i=2}^{n} (i - 1) = \sum_{l=1}^{n-1} l = \dfrac{n(n-1)}{2}$.
- 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) ⇒ $T(n)$ 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 $A[i]$ is less than half the elements in $A[1:i-1]$ and it's greater than the other half.
  $\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1:i-1]$ to decide where to drop *key*.
  $\Rightarrow t_i \approx i/2$.

  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 $A[1:i-1]$, place $A[i]$ correctly, so that $A[1:i]$ is sorted.

Divide and conquer

Another common approach.

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

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

[Are your students 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.

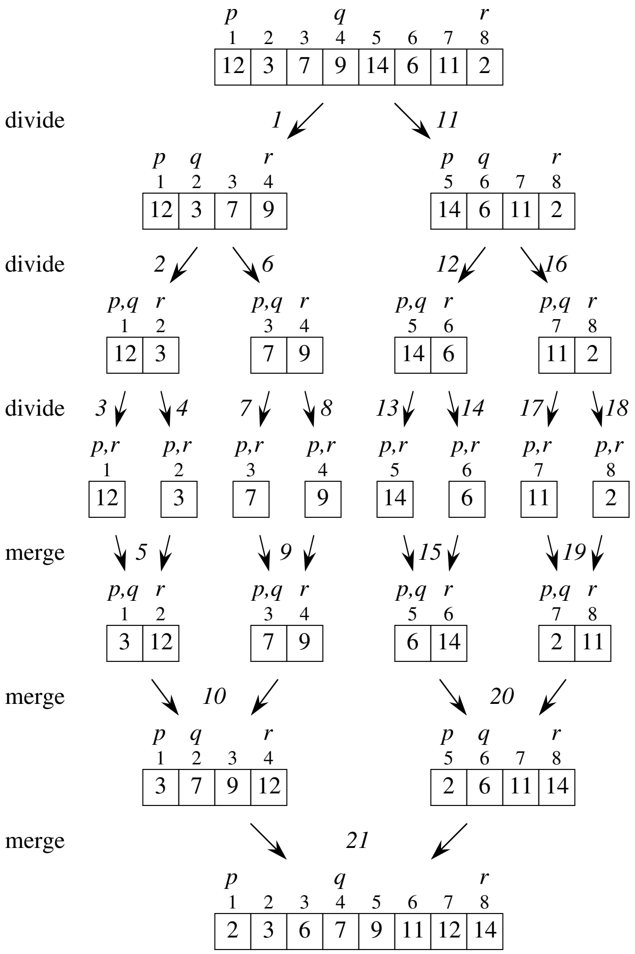
```
MERGE-SORT(A, p, r)
  if $p \geq r$                    // zero or one element?
      return
  $q = \lfloor (p+r)/2 \rfloor$              // midpoint of $A[p:r]$
  MERGE-SORT(A, p, q)          // recursively sort $A[p:q]$
  MERGE-SORT(A, q + 1, r)      // recursively sort $A[q+1:r]$
  // Merge $A[p:q]$ and $A[q+1:r]$ into $A[p:r]$.
  MERGE(A, p, q, r)
```

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.]

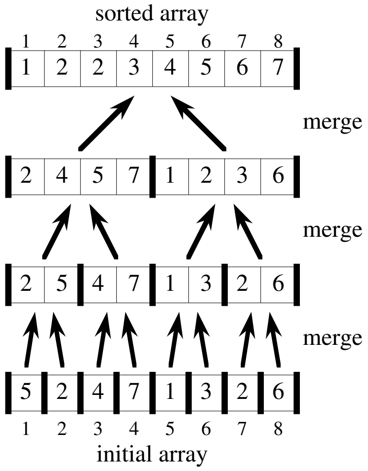
Example

MERGE-SORT on an array with $n = 8$: *[Indices p, q, r appear above their values. Numbers in italics indicate the order of calls of* MERGE *and* MERGE-SORT *after the initial call* MERGE-SORT$(A, 1, 8)$.*]*



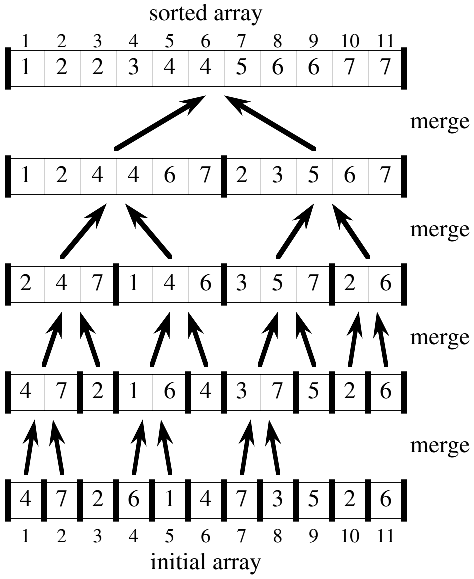
Example

Bottom-up view for $n = 8$: *[Heavy lines demarcate subarrays used in subproblems. Go through the following two figures bottom to top.]*



[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.
- Merge these two piles 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 checking just the two top cards.
- There are $\le n$ basic steps, since each basic step removes one card from the input piles, and started with n cards in the input piles.
- Therefore, this procedure should take $\Theta(n)$ time.

More details on the MERGE prodecure, which copies the two subarrays $A[p:q]$ and $A[q+1:r]$ into temporary arrays L and R ("left" and "right"), and then merges the values in L and R back into $A[p:r]$:

- First compute the lengths n_L and n_R of the subarrays $A[p:q]$ and $A[q+1:r]$, respectively.
- Then create arrays $L[0:n_L - 1]$ and $R[0:n_R - 1]$ with respective lengths n_L and n_R.
- The two **for** loops copy the subarrays $A[p:q]$ into L and $A[q+1:r]$ into R.
- The first **while** loop repeatedly identifies the smallest value in L and R that has yet to be copied back into $A[p:r]$ and copies it back in.

- As the comments indicate, the index k gives the position of A that is being filled in, and the indices i and j give the positions in L and R, respectively, of the smallest remaining values.
- Eventually, either all of L or all of R will be copied back into $A[p:r]$, and this loop terminates.
- If the loop terminated because all of R was copied back, that is, because $j = n_R$, then i is still less than n_L, so that some of L has yet to be copied back, and these values are the greatest in both L and R.
- In this case, the second **while** loop copies these remaining values of L into the last few positions of $A[p:r]$.
- Because $j = n_R$, the third **while** loop iterates zero times.
- If instead the first **while** loop terminated because $i = n_L$, then all of L has already been copied back into $A[p:r]$, and the third **while** loop copies the remaining values of R back into the end of $A[p:r]$.

*[The second and third editions of the book added a sentinel value of ∞ to the end of the L and R arrays. Doing so avoids one test in each iteration of the first **while** loop, and it eliminates the last two **while** loops. We removed the sentinel in the fourth edition because in practice, there might not be a clear choice for the sentinel value, depending on the types of the keys being compared.]*

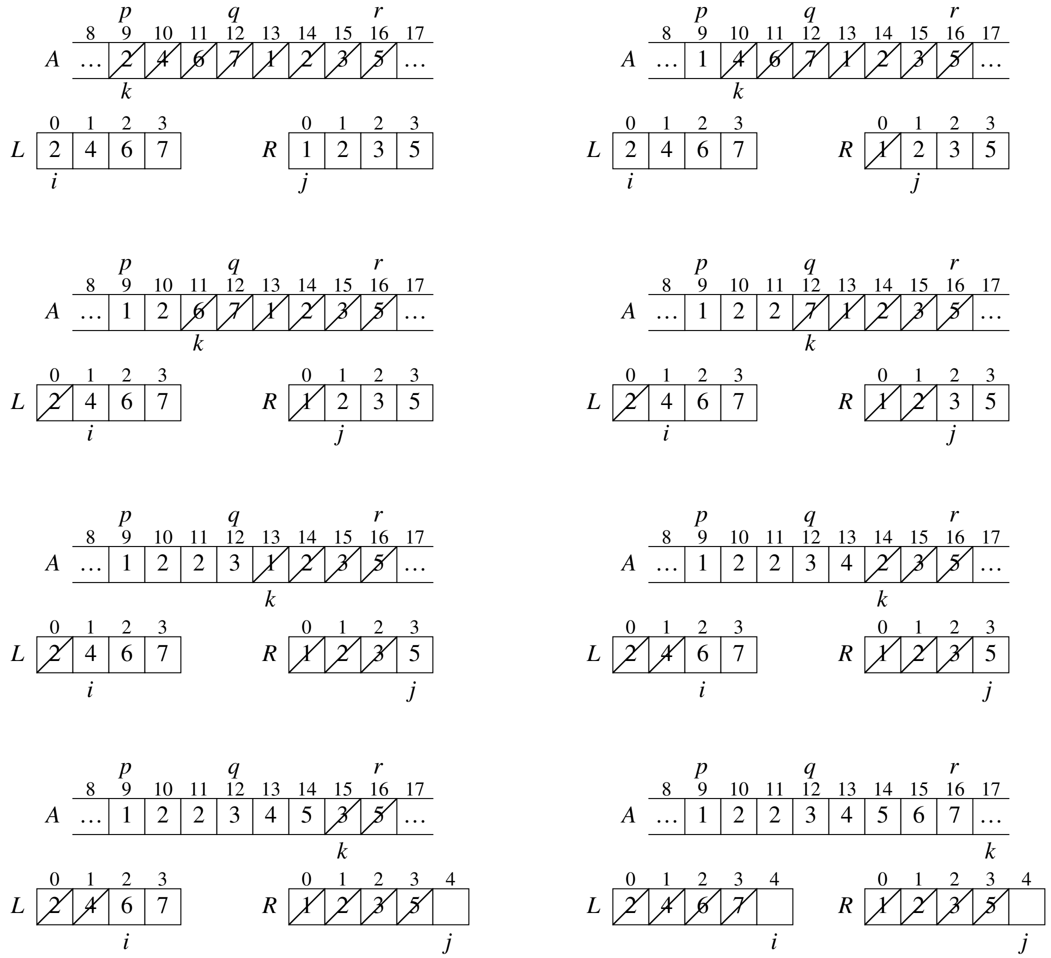
Pseudocode

```
MERGE($A, p, q, r$)
  $n_L = q - p + 1$          // length of $A[p:q]$
  $n_R = r - q$              // length of $A[q + 1:r]$
  let $L[0:n_L - 1]$ and $R[0:n_R - 1]$ be new arrays
  for $i = 0$ to $n_L - 1$   // copy $A[p:q]$ into $L[0:n_L - 1]$
      $L[i] = A[p + i]$
  for $j = 0$ to $n_R - 1$   // copy $A[q + 1:r]$ into $R[0:n_R - 1]$
      $R[j] = A[q + j + 1]$
  $i = 0$                    // $i$ indexes the smallest remaining element in $L$
  $j = 0$                    // $j$ indexes the smallest remaining element in $R$
  $k = p$                    // $k$ indexes the location in $A$ to fill
  // As long as each of the arrays $L$ and $R$ contains an unmerged element,
  //       copy the smallest unmerged element back into $A[p:r]$.
  while $i < n_L$ and $j < n_R$
      if $L[i] \le R[j]$
          $A[k] = L[i]$
          $i = i + 1$
      else $A[k] = R[j]$
          $j = j + 1$
      $k = k + 1$
  // Having gone through one of $L$ and $R$ entirely, copy the
  //       remainder of the other to the end of $A[p:r]$.
  while $i < n_L$
      $A[k] = L[i]$
      $i = i + 1$
      $k = k + 1$
  while $j < n_R$
      $A[k] = R[j]$
      $j = j + 1$
      $k = k + 1$
```

[Exercise 2.3-3 in the book asks the reader to use 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. Arrays L and R are indexed from 0, rather than from 1, to make the loop invariant simpler.]*

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** $k = p$ **to** r" loop, where $A[p:q]$ is copied into $L[0:n_L - 1]$ and $A[q+1:r]$ is copied into $R[0:n_R - 1]$. 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. In the last part, because all of R has been copied back into A, but the last two entries in L have not, the remainder of L is copied into the end of $A[p:r]$ without being compared with any entries in R. Now the subarrays are merged back into $A[p:r]$, which is now sorted.]*

Running time

The first two **for** loops take $\Theta(n_L + n_R) = \Theta(n)$ time. Each of the three lines before and after the **for** loops takes constant time.

Each iteration of the three **while** loops copies exactly one value from L or R into A, and every value is copied back into A exactly one time. Therefore, these three loops make a total of n iterations, each taking constant time, for $\Theta(n)$ time. Total running 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, $n \le n_0$ for some constant n_0), have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, 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 $D(n)$.
- Have a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- Get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le n_0 \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

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

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

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.

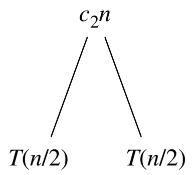
On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

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

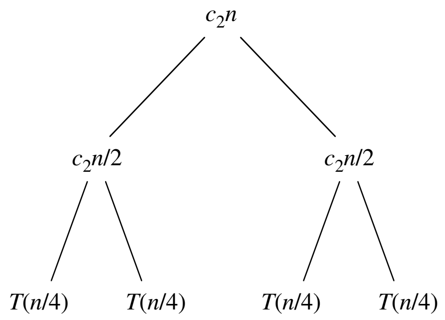
- Let c_1 be a constant that describes the running time for the base case and c_2 be a constant for the time per array element for the divide and conquer steps.
- Assume that the base case occurs for $n = 1$, so that $n_0 = 1$.
- Rewrite the recurrence as

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 . \end{cases}$$

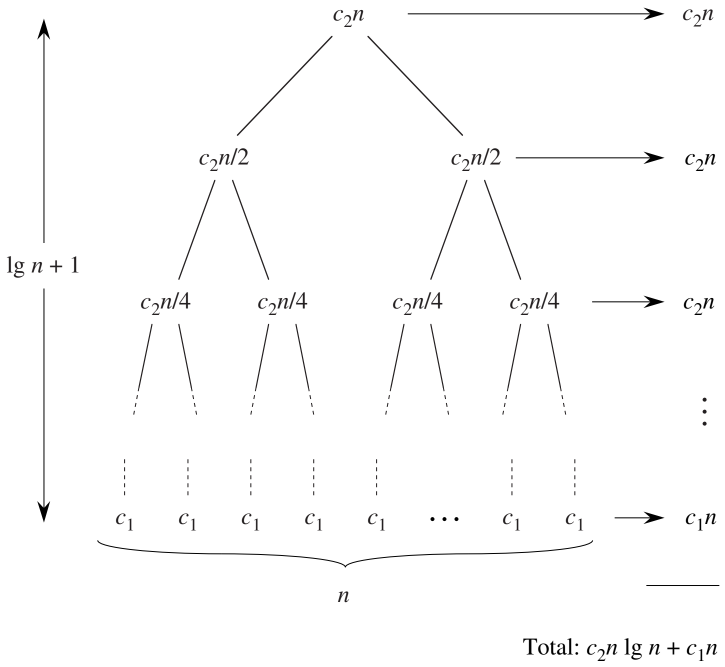
- Draw a ***recursion tree***, which shows successive expansions of the recurrence.
- For the original problem, have a cost of $c_2 n$, plus the two subproblems, each costing $T(n/2)$:



- For each of the size-$n/2$ subproblems, have a cost of $c_2 n/2$, plus two subproblems, each costing $T(n/4)$:



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



- Each level except the bottom has cost $c_2 n$.
  - The top level has cost $c_2 n$.
  - The next level down has 2 subproblems, each contributing cost $c_2 n/2$.
  - The next level has 4 subproblems, each contributing cost $c_2 n/4$.
  - 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 level except the bottom costs $c_2 n \Rightarrow c_2 n \lg n$. The bottom level has n leaves in the recursion tree, each costing $c_1 \Rightarrow c_1 n$.
- Total cost is $c_2 n \lg n + c_1 n$. Ignore low-order term of $c_1 n$ and constant coefficient $c_2 \Rightarrow \Theta(n \lg n)$.

Solutions for Chapter 2: Getting Started

Solution to Exercise 2.1-2

Loop invariant: At the start of each iteration i of the **for** loop, $sum = A[1] + A[2] + \cdots + A[i-1]$.

Now, we will use the initialization, maintenance, and termination properties of the loop invariant to show that SUM-ARRAY returns the sum of the numbers in $A[1:n]$.

Initialization: Upon entering the first iteration, $i = 1$. The sum $A[1] + A[2] + \cdots + A[i-1]$ is empty, since $i - 1 = 0$. The sum of no terms is the identity 0 for addition.

Maintenance: Assume that the loop invariant is true, so that upon entering an iteration for a value of i, $sum = A[1]+A[2]+\cdots+A[i-1]$. The iteration adds $A[i]$ into *sum* and then increments i, so that the loop invariant holds entering the next iteration.

Termination: The loop terminates once $i = n + 1$. According to the loop invariant, $sum = A[1] + A[2] + \cdots + A[n]$. Therefore, *sum*, which the procedure returns, is indeed the sum of $A[1:n]$.

Solution to Exercise 2.1-3

In line 5 of INSERTION-SORT, change the test $A[j] > key$ to $A[j] < key$.

Solution to Exercise 2.1-4

```
LINEAR-SEARCH(A, n, x)
  i = 1
  while i ≤ n and A[i] ≠ x
      i = i + 1
  if i > n
      return NIL
  else return i
```

The procedure checks each array element until either $i > n$ or the value x is found.

> **Loop invariant:** At the start of each iteration of the **while** loop, the value x does not appear in the subarray $A[1:i-1]$.

Initialization: Upon entering the first iteration, $i = 1$. The subarray $A[1:i-1]$ is empty, and the loop invariant is trivally true.

Maintenance: At the start of an iteration for index i, the loop invariant says that x does not appear in the subarray $A[1:i-1]$. If $i \leq n$, then either $A[i] = x$ or $A[i] \neq x$. In the former case, the test in the **for** loop header comes up FALSE, and there is no iteration for $i+1$. In the latter case, the test comes up TRUE. Since $A[i] \neq x$ and x does not appear in $A[1:i-1]$, we have that x does not appear in $A[1:i]$. Incrementing i for the next iteration preserves the loop invariant. If $i > n$, then the test in the **for** loop header comes up FALSE, and there is no iteration for $i+1$.

Termination: The **for** loop terminated for one of two reasons. If it terminated because $i > n$, then $i = n+1$ at that time. By the loop invariant, the value x does not appear in the the subarray $A[1:i-1]$, which is the entire array $A[1:n]$. The procedure properly returns NIL in this case. If the loop terminated because $i \leq n$ and $A[i] = x$, then the procedure properly returns the index i.

Solution to Exercise 2.1-5

```
ADD-BINARY-INTEGERS($A, B, n$)
  allocate array $C[0:n]$
  $carry = 0$
  for $i = 0$ to $n - 1$
      $sum = A[i] + B[i] + carry$
      $C[i] = sum \bmod 2$
      if $sum \leq 1$
          $carry = 0$
      else $carry = 1$
  $C[n] = carry$
  return $C$
```

Solution to Exercise 2.2-1

In terms of Θ-notation, the function $n^3/1000 + 100n^2 - 100n + 3$ is $\Theta(n^3)$.

Solution to Exercise 2.2-2
This solution is also posted publicly

```
SELECTION-SORT($A, n$)
  for $i = 1$ to $n - 1$
      $smallest = i$
      for $j = i + 1$ to $n$
          if $A[j] < A[smallest]$
              $smallest = j$
      exchange $A[i]$ with $A[smallest]$
```

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1:i-1]$ consists of the $i-1$ 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 2.2-3

On average, when searching for a value x, half of the elements will be less than or equal to x, and half will be greater than or equal to x, so that $n/2$ elements are checked on average. In the worst case, x appears only in the last position of the array, so that the entire array must be checked. Therefore, the running time of linear search is $\Theta(n)$ in both the average and worst cases.

Solution to Exercise 2.2-4
This solution is also posted publicly

Modify the algorithm so that it first checks the input array to see whether it is already sorted, taking $\Theta(n)$ time for an n-element array. If the array is already sorted, then the algorithm is done. Otherwise, sort the array as usual. The best-case running time is generally not a good measure of an algorithm's efficiency.

Solution to Exercise 2.3-2

In order for $p \neq r$ to suffice, we need to show that a call of MERGE-SORT$(A, 1, n)$ with $n \geq 1$ will never result in a recursive call that leads to $p > r$. In this initial call, $p \leq r$. We'll look at any call with $p \leq r$ and show that it cannot lead to a recursive call with $p > r$.

If $p = r$, then in the call MERGE-SORT(A, p, r), the test $p \neq r$ evaluates to FALSE, and the code makes no recursive calls.

If $p < r$, then we will show that $p \leq q$ in the first recursive call in line 4 and $q + 1 \leq r$ in the second recursive call in line 5. Because $p < r$, we have $p = 2p/2 = \lfloor (p + p)/2 \rfloor \leq \lfloor (p + r)/2 \rfloor = q$, so that $p \leq q$ in line 4. Because $r > \lfloor (r + p)/2 \rfloor$ and both r and $\lfloor (r + p)/2 \rfloor$ are integers, $r \geq \lfloor (r + p)/2 \rfloor + 1 = q + 1$. Therefore, we have $q + 1 \leq r$ in line 5.

Since an initial call of MERGE-SORT$(A, 1, n)$ with $n \geq 1$ will always result in calls to MERGE-SORT with $p \leq r$, changing the condition on line 1 to say $p \neq r$ will suffice.

Solution to Exercise 2.3-3

> **Loop invariant:** At the start of each iteration of the **while** loop of lines 12–18, the subarray $A[p:k-1]$ contains the $i + j$ smallest elements of $L[0:n_L - 1]$ and $R[0:n_R - 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **while** loop of lines 12–18, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates. In fact, we will consider as well the **while** loops of lines 20–23 and lines 24–27 to show that the MERGE procedure works correctly.

Initialization: Prior to the first iteration of the loop, we have $k = p$ so that the subarray $A[p:k-1]$ is empty. Since $i = j = 0$, this empty subarray contains the $i + j = 0$ smallest elements of L and R, and both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p:k-1]$ contains the $i + j$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p:k]$ will contain the $i + j + 1$ smallest elements. Incrementing i in line 15 and k in line 18 reestablishes the loop invariant for the next iteration. If it was instead the case that $L[i] > R[j]$, then lines 16–18 perform the appropriate action to maintain the loop invariant.

Termination: Each iteration of the loop increments either i or j. Eventually, either $i \geq n_L$, so that all elements in L have been copied back into A, or $j \geq n_R$, so that all elements in R have been copied back into A. By the loop invariant, when the loop terminates, the subarray $A[p:k-1]$ contains the $i + j$ smallest elements of $L[0:n_L-1]$ and $R[0:n_R-1]$, in sorted order. The subarray $A[p:r]$ consists of $r - p + 1$ positions, the last $r - p + 1 - (i + j)$ which have yet to be copied back into.

Suppose that the loop terminated because $i \geq n_L$. Then the **while** loop of lines 20–23 iterates 0 times, and the **while** loop of lines 24–27 copies the remaining $n_R - j$ elements of R into the rightmost $n_R - j$ positions of $A[p:r]$. These elements of R must be the $n_R - j$ greatest values in L and R. Thus, we just need to show that the correct number of elements in R are copied back into $A[p:r]$, that is, $r - p + 1 - (i + j) = n_R - j$. We use two facts to do so. First, because the number of positions in $A[p:r]$ equals the combined sizes of the L and R arrays, we have $r - p + 1 = n_L + n_R$, or $n_L = r - p + 1 - n_R$. Second, because $i \geq n_L$ and the **while** loop of lines 12–18 increases i by at most 1 in each iteration, we must have that $i = n_L$ when this loop terminated. Thus, we have

$$\begin{aligned} r - p + 1 - (i + j) &= r - p + 1 - n_L - j \\ &= r - p + 1 - (r - p + 1 - n_R) - j \\ &= n_R - j \; . \end{aligned}$$

If instead the loop terminated because $j \geq n_R$, then you can show that the remaining $n_L - i$ elements of L are the greatest values in L and R, and that the **while** loop of lines 20–23 copies them into the last $r - p + 1 - (i + j)$ positions of $A[p:r]$. In either case, we have shown that the MERGE procedure merges the two sorted subarrays $A[p:q]$ and $A[q + 1:r]$ correctly.

Solution to Exercise 2.3-4

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 2.3-5

The pseudocode for a recursive insertion sort is as follows:

```
RECURSIVE-INSERTION-SORT$(A, n)$
  if $n > 1$
      RECURSIVE-INSERTION-SORT$(A, n - 1)$
      $key = A[n]$
      $j = n - 1$
      while $j > 0$ and $A[j] > key$
          $A[j + 1] = A[j]$
          $j = j - 1$
      $A[j + 1] = key$
```

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

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

Although the exercise does not ask you to solve this recurrence, its solution is $T(n) = \Theta(n^2)$.

Solution to Exercise 2.3-6

This solution is also posted publicly

Procedure BINARY-SEARCH takes a sorted array A, a value x, and a range $[low : high]$ of the array, in which we search for the value x. The procedure compares x 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 $A[i] = x$, or NIL if no entry of $A[low : high]$ contains the value x. The initial call to either version should have the parameters $A, x, 1, n$.

```
ITERATIVE-BINARY-SEARCH($A, x, low, high$)
  while $low \le high$
      $mid = \lfloor (low + high)/2 \rfloor$
      if $x == A[mid]$
          return $mid$
      elseif $x > A[mid]$
          $low = mid + 1$
      else $high = mid - 1$
  return NIL
```

```
RECURSIVE-BINARY-SEARCH($A, x, low, high$)
  if $low > high$
      return NIL
  $mid = \lfloor (low + high)/2 \rfloor$
  if $x == A[mid]$
      return $mid$
  elseif $x > A[mid]$
      return RECURSIVE-BINARY-SEARCH($A, x, mid + 1, high$)
  else return RECURSIVE-BINARY-SEARCH($A, x, low, mid - 1$)
```

Both procedures terminate the search unsuccessfully when the range is empty (i.e., $low > high$) and terminate it successfully if the value x has been found. Based on the comparison of x 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 2.3-7

The **while** loop of lines 5–7 of INSERTION-SORT scans backward through the sorted array $A[1:i-1]$ to find the appropriate place for $A[i]$. The hitch is that the loop not only searches for the proper place for $A[i]$, but that it also moves each of the array elements that are greater than $A[i]$ one position to the right (line 6). These movements can take as much as $\Theta(j)$ time, which occurs when all the $i-1$ elements preceding $A[i]$ are greater than $A[i]$. Binary search can improve the running time of the search to $O(\lg i)$, 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 2.3-8

The following algorithm solves the problem. First, observe that since S is a set, if $x/2 \in S$, then $x/2$ appears just once, and so it cannot be a solution. Therefore, if $x/2 \in S$, first remove it from S. Then do the following:

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. Merge the two sorted sets S and S'.
5. 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 5, 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 $\Theta(n \lg n)$ steps. Steps 2, 4, and 5 require $\Theta(n)$ steps. Thus the overall running time is $\Theta(n \lg n)$.

A reader submitted a simpler solution that also runs in $\Theta(n \lg n)$ worst-case time. First, sort the elements in S (again, first removing $x/2$ if need be), taking $\Theta(n \lg n)$ time. Then, for each element $y \in S$, perform a binary search in S for $x - y$. Each binary search takes $\Theta(\lg n)$ time in the worst case, and there are are most n of them, so that the worst-case time for all the binary searches is $\Theta(n \lg n)$. The overall running time is $\Theta(n \lg n)$.

Here is yet another solution. Again, remove $x/2$ from S if need be and then sort S. Let's assume that S is now represented as a sorted array $S[1:n]$. Maintain two indices into S, *low* and *high*, with the loop invariant that if S contains two values that sum to x, then they are in the subarray $S[low:high]$. Here is pseudocode:

```
SUM-PAIR(S, x)
  sort S into S[1 : n]
  low = 1
  high = n
  while low < high
      sum = S[low] + S[high]
      if sum == x
          return (S[low], S[high])
      elseif sum < x
          low = low + 1
      else high = high − 1
  return "no pair sums to x"
```

Solution to Problem 2-1

[It may be better to assign this problem after covering asymptotic notation in Section 3.2; 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, 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 it is 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. You 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, $A[j]$ is the smallest value in the subarray $A[j:n]$, and $A[j:n]$ 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 the 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, $A[i]$ is the smallest value in the subarray $A[i:n]$, 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 $i-1$ smallest values in $A[1:n]$, in sorted order. Therefore, $A[i-1] \le A[i]$. 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 $A[1:i]$ now consists of 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 $i = n$, so that $i-1 = n-1$. By the statement of the loop invariant, $A[1:i-1]$ is the subarray $A[1:n-1]$, and it consists of the $n-1$ smallest values originally in $A[1:n]$, in sorted order. The remaining element must be the largest value in $A[1:n]$, and it is in $A[n]$. Therefore, the entire array $A[1:n]$ is sorted.

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 $1, 2, \ldots, n-1$. The total number of iterations, therefore, is

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

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 2-3

a. The procedure HORNER runs in $\Theta(n)$ time. The **for** loop iterates $n + 1$ times, and each iteration takes constant time.

b. The following procedure computes each term of the polynomial from scratch.

```
EVALUATE-POLYNOMIAL$(A, n, x)$
  $p = 0$
  for $i = 0$ to $n$
      $power = 1$
      for $k = 1$ to $i$
          $power = power \cdot x$
      $p = p + A[i] \cdot power$
  return $p$
```

Note that when $i = 0$, the **for** loop makes no iterations.

To determine the running time, observe that the outer **for** loop makes $n + 1$ iterations. For a given value of i, the inner **for** loop makes i iterations, each taking constant time. The total number of inner-loop iterations is $0 + 1 + 2 + \cdots + n = n(n+1)/2$, which is $\Theta(n^2)$. (Note that although the parameter n could equal 0, the input size in this case would be 1, so that we don't have to worry about evaluating $\Theta(n^2)$ when the input size is 0.) This method is slower than Horner's rule.

c. **Initialization:** At the start of the first iteration of the **for** loop, $i = n$ and so the summation in the loop invariant goes from $k = 0$ to $k = -1$. That is, it's an empty summation, equaling 0, which is the initial value of p.

Maintenance: Assume that the loop invariant holds entering an iteration of the **for** loop for a given value of i, so that $p = \sum_{k=0}^{n-(i+1)} A[k+i+1] \cdot x^k$. Denote by p' the new value of p computed in line 3. Then, we have

$$\begin{aligned} p' &= A[i] + x \cdot p \\ &= A[i] + x \cdot \sum_{k=0}^{n-(i+1)} A[k+i+1] \cdot x^k \\ &= A[i] + \sum_{k=0}^{n-(i+1)} A[k+i+1] \cdot x^{k+1} \\ &= A[i] + \sum_{k=1}^{n-i} A[k+i] \cdot x^k \\ &= \sum_{k=0}^{n-i} A[k+i] \cdot x^k \; . \end{aligned}$$

The next iteration decreases i by 1 so that the value i in this iteration becomes $i+1$ in the next iteration. Thus, entering the next iteration, $p = \sum_{k=0}^{n-(i+1)} A[k+i+1] \cdot x^k$ and the invariant is maintained.

Termination: The **for** loop terminates after $n+1$ iterations. When it terminates, $i = -1$. By the loop invariant, we have $p = \sum_{k=0}^{n} A[k] \cdot x^k$, which equals $P(x)$.

Solution to Problem 2-4
This solution is also posted publicly

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 drawn 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, i). Then $k < i$ and $A[k] > A[i]$. At the time that the outer **for** loop of lines 1–8 sets *key* $= A[i]$, the value that started in $A[k]$ is still somewhere to the left of $A[i]$. That is, it's in $A[j]$, where $1 \le j < i$, and so the inversion has become (j, i). Some iteration of the **while** loop of lines 5–7 moves $A[j]$ 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 greater 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 $L[i], L[i+1], L[i+2], \ldots, L[n_L - 1]$, and these $n_L - i$ 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 $n_L - i$ at that time to the total count of merge-inversions.

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

```
MERGE-INVERSIONS$(A, p, q, r)$
  $n_L = q - p + 1$
  $n_R = r - q$
  let $L[0:n_L - 1]$ and $R[0:n_R - 1]$ be new arrays
  for $i = 0$ to $n_L - 1$
      $L[i] = A[p + i - 1]$
  for $j = 0$ to $n_R - 1$
      $R[j] = A[q + j]$
  $i = 0$
  $j = 0$
  $k = p$
  $inversions = 0$
  while $i < n_L$ and $j < n_R$
      if $L[i] \leq R[j]$
          $inversions = inversions + n_L - i$
          $A[k] = L[i]$
          $i = i + 1$
      else $A[k] = R[j]$
          $j = j + 1$
      $k = k + 1$
  while $i < n_L$
      $A[k] = L[i]$
      $i = i + 1$
      $k = k + 1$
  while $j < n_R$
      $A[k] = R[j]$
      $j = j + 1$
      $k = k + 1$
  return $inversions$
```

```
COUNT-INVERSIONS$(A, p, r)$
  $inversions = 0$
  if $p < r$
      $q = \lfloor (p + r)/2 \rfloor$
      $inversions = inversions +$ COUNT-INVERSIONS$(A, p, q)$
      $inversions = inversions +$ COUNT-INVERSIONS$(A, q + 1, r)$
      $inversions = inversions +$ MERGE-INVERSIONS$(A, p, q, r)$
  return $inversions$
```

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

In MERGE-INVERSIONS, whenever $R[j]$ is exposed and a value greater than $R[j]$ becomes exposed in the L array, we increase *inversions* by the number of remaining elements in L. Then because $R[j + 1]$ becomes exposed, $R[j]$ can never be exposed again.

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: Characterizing Running Times

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:

$$\begin{array}{lcl} O & \approx & \le \\ \Omega & \approx & \ge \\ \Theta & \approx & = \\ o & \approx & < \\ \omega & \approx & > \end{array}$$

O-notation, Ω-notation, and Θ-notation

[Section 3.1 does not go over formal definitions of O-notation, Ω-notation, and Θ-notation, but is intended to informally introduce the three most commonly used types of asymptotic notation and show how to use these notations to reason about the worst-case running time of insertion sort.]

O-notation

O-notation characterizes an *upper bound* on the asymptotic behavior of a function: it says that a function grows *no faster* than a certain rate. This rate is based on the highest order term.

For example, $f(n) = 7n^3 + 100n^2 - 20n + 6$ is $O(n^3)$, since the highest order term is $7n^3$, and therefore the function grows no faster than n^3.

Te function $f(n)$ is also $O(n^5)$, $O(n^6)$, and $O(n^c)$ for any constant $c \ge 3$.

Ω-notation

Ω-notation characterizes a *lower bound* on the asymptotic behavior of a function: it says that a function grows *at least as fast* as a certain rate. This rate is again based on the highest-order term.

For example, $f(n) = 7n^3 + 100n^2 - 20n + 6$ is $\Omega(n^3)$, since the highest-order term, n^3, grows at least as fast as n^3.

The function $f(n)$ is also $\Omega(n^2)$, $\Omega(n)$, and $\Omega(n^c)$ for any constant $c \leq 3$.

Θ-notation

Θ-notation characterizes a *tight bound* on the asymptotic behavior of a function: it says that a function grows *precisely* at a certain rate, again based on the highest-order term.

If a function is both $O(f(n))$ and $\Omega(f(n))$, then a function is $\Theta(f(n))$.

Example: Insertion sort

We will characterize insertion sort's $\Theta(n^2)$ worst-case running time as an example of how to work with asymptotic notation.

Here is the INSERTION-SORT procedure, from Chapter 2:

```
INSERTION-SORT(A, n)
  for i = 2 to n
      key = A[i]
      // Insert A[i into the sorted subarray A[1 : i − 1].
      j = i − 1
      while j > 0 and A[j] > key
          A[j + 1] = A[j]
          j = j − 1
      A[j + 1] = key
```

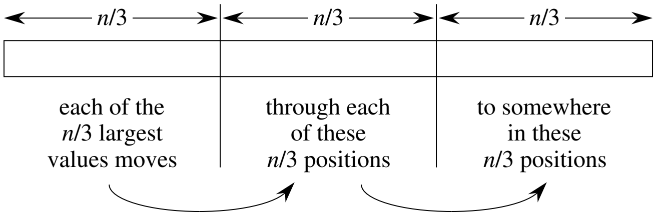
First, show that INSERTION-SORT is runs in$O(n^2)$ time, regardless of the input:

- The outer **for** loop runs $n - 1$ times regardless of the values being sorted.
- The inner **while** loop iterates at most $i - 1$ times.
- The exact number of iterations the **while** loop makes depends on the values it iterates over, but it will definitely iterate between 0 and $i - 1$ times.
- Since i is at most n, the total number of iterations of the inner loop is at most $(n-1)(n-1)$, which is less than n^2.

Since each iteration of the inner loop takes constant time, the total time spent in the inner loop is at most cn^2 for some constant c, or $O(n^2)$.

Now show that INSERTION-SORT has a worst-case running time of $\Omega(n^2)$ by demonstrating an input that makes the running time be at least some constant times n^2:

- Observe that for a value to end up k positions to the right of where it started, the line $A[j+1] = A[j]$ must have been executed k times.
- Assume that n is a multiple of 3 so that we can divide the array A into groups of $n/3$ positions.


n/3
n/3
n/3
each of the n/3 largest values moves
through each of these n/3 positions
to somewhere in these n/3 positions

- Suppose that the input to INSERTION-SORT has the $n/3$ largest values in the first $n/3$ array positions $A[1:n/3]$. The order within the first $n/3$ positions does not matter.
- Once the array has been sorted, each of these $n/3$ values will end up somewhere in the last $n/3$ positions $A[2n/3+1:n]$.
- For that to happen, each of these $n/3$ values must pass through each of the middle $n/3$ positions $A[n/3+1:2n/3]$.

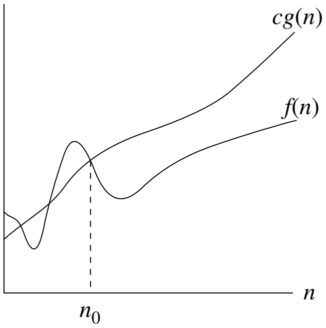
Because at least $n/3$ values must pass through at least $n/3$ positions, the line $A[j+1] = A[j]$ executes at least $(n/3)(n/3) = n^2/9$ times, which is $\Omega(n^2)$. For this input, INSERTION-SORT takes time $\Omega(n^2)$.

Since we have shown that INSERTION-SORT runs in $O(n^2)$ time in all cases and that there is an input that makes it take $\Omega(n^2)$ time, we can conclude that the worst-case running time of INSERTION-SORT is $\Theta(n^2)$. The constant factors for the upper and lower bounds may differ. That doesn't matter. The important point is to characterize the worst-case running time to within constant factors. We're focusing on just the worst-case running time here, since the best-case running time for insertion sort is $\Theta(n)$.

Asymptotic notation: formal definitions

O-notation

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



$g(n)$ 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

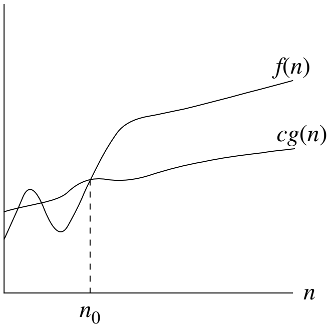
$2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

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

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



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

Example

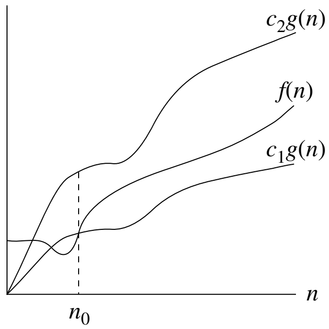
$\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

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

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



$g(n)$ 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.

Can express a constant factor as $O(1)$ or $\Theta(1)$, since it's within a constant factor of 1.

Asymptotic notation and running times

Need to be careful to use asymptotic notation correctly when characterizing a running time. Asymptotic notation describes functions, which in turn describe running times. Must be careful to specify *which* running time.

For example, the worst-case running time for insertion sort is $O(n^2)$, $\Omega(n^2)$, and $\Theta(n^2)$; all are correct. Prefer to use $\Theta(n^2)$ here, since it's the most precise. The best-case running time for insertion sort is $O(n)$, $\Omega(n)$, and $\Theta(n)$; prefer $\Theta(n)$.

But *cannot* say that the running time for insertion sort is $\Theta(n^2)$, with "worst-case" omitted. Omitting the case means making a blanket statement that covers *all* cases, and insertion sort does *not* run in $\Theta(n^2)$ time in all cases.

Can make the blanket statement that the running time for insertion sort is $O(n^2)$, or that it's $\Omega(n)$, because these asymptotic running times are true for all cases.

For merge sort, its running time is $\Theta(n \lg n)$ in all cases, so it's OK to omit which case.

Common error: conflating O-notation with Θ-notation by using O-notation to indicate an asymptotically tight bound. O-notation gives only an asymptotic upper bound. Saying "an $O(n \lg n)$-time algorithm runs faster than an $O(n^2)$-time algorithm" is not necessarily true. An algorithm that runs in $\Theta(n)$ time also runs in $O(n^2)$ time. If you really mean an asymptotically tight bound, then use Θ-notation.

Use the simplest and most precise asymptotic notation that applies. Suppose that an algorithm's running time is $3n^2 + 20n$. Best to say that it's $\Theta(n^2)$. Could say that it's $O(n^3)$, but that's less precise. Could say that it's $\Theta(3n^2 + 20n)$, but that obscures the order of growth.

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, $f(n) = 3n + 1$.

Interpret the number of anonymous functions as equaling the number 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 $2n^2 + g(n) = h(n)$.

Proper abuses of asymptotic notation

It's usually clear what variable in asymptotic notation is tending toward ∞: in $O(g(n))$, looking at the growth of $g(n)$ as n grows.

What about $O(1)$? There's no variable appearing in the asymptotic notation. Use the context to disambiguate: in $f(n) = O(1)$, the variable is n, even though it does not appear in the right-hand side of the equation.

Subtle point: asymptotic notation in recurrences

Often abuse asymptotic notation when writing recurrences: $T(n) = O(1)$ for $n < 3$. Strictly speaking, this statement is meaningless. Definition of O-notation says that $T(n)$ is bounded above by a constant $c > 0$ for $n \geq n_0$, for some $n_0 > 0$. The value of $T(n)$ for $n < n_0$ might not be bounded. So when we say $T(n) = O(1)$ for $n < 3$, cannot determine any constraint on $T(n)$ when $n < 3$ because could have $n_0 > 3$.

What we really mean is that there exists a constant $c > 0$ such that $T(n) \leq c$ for $n < 3$. This convention allows us to avoid naming the bounding constant so that we can focus on the more important part of the recurrence.

Asymptotic notation defined for only subsets

Suppose that an algorithm assumes that its input size is a power of 2. Can still use asymptotic notation to describe the growth of its running time. In general, can use asymptotic notation for $f(n)$ defined on only a subset of $\mathbb{N}$ or $\mathbb{R}$.

o-notation

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

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

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

ω-notation

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

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

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

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 3.3, 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:

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

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 $\log_b a$:

- Hold b constant ⇒ the expression is strictly increasing as a increases.
- Hold a constant ⇒ 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:

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

[For the last equality, can show by taking $\log_b$ *of both sides:*

$\log_b a^{\log_b c} = (\log_b c)(\log_b a)$,

$\log_b c^{\log_b a} = (\log_b a)(\log_b c)$. *]*

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: Characterizing Running Times

Solution to Exercise 3.1-1

If the input size n is not an exact multiple of 3, then divide the array A so that the leftmost and middle sections have $\lfloor n/3 \rfloor$ positions each, and the rightmost section has $n - 2\lfloor n/3 \rfloor > \lfloor n/3 \rfloor$ positions. Use the same argument as in the book, but for an input that has the $\lfloor n/3 \rfloor$ largest values starting in the leftmost section. Each such value must move through the middle $\lfloor n/3 \rfloor$ positions en route to its final position in the rightmost section, one position at a time. Therefore, the total number of executions of line 6 of INSERTION-SORT is at least

$$\begin{aligned}
\lfloor n/3 \rfloor \cdot \lfloor n/3 \rfloor &> (n/3 - 1)(n/3 - 1) \quad \text{(by equation (3.2))} \\
&= n^2/9 - 2n/3 + 1 \\
&= \Omega(n^2) \ .
\end{aligned}$$

Solution to Exercise 3.1-2

Recall the pseudocode for selection sort, from Exercise 2.2-2:

```
SELECTION-SORT(A, n)
  for i = 1 to n - 1
      smallest = i
      for j = i + 1 to n
          if A[j] < A[smallest]
              smallest = j
      exchange A[i] with A[smallest]
```

First, we show that SELECTION-SORT's running time is $O(n^2)$. The outer loop iterates $n - 1$ times, regardless of the values being sorted. The inner loop iterates at most $n - 1$ times per iteration of the outer loop, and so the inner loop iterates at most $(n - 1)(n - 1)$ times, which is less than n^2 times. Each iteration of the inner loop takes at most constant time, so that the total time spent in the inner loop is at most cn^2 for some constant c. Therefore, selection sort runs in $O(n^2)$ time in all cases.

Next, we show that SELECTION-SORT takes $\Omega(n^2)$ time. Consider what the procedure does to fill any position i, where $i \le n/2$. The procedure must examine each value to the right of position i. Since $i \le n/2$, that is, position i is in the leftmost half of the array, the procedure must examine at least every one of the $n/2$ positions in the rightmost half of the array. Thus, for each of the leftmost $n/2$ positions, it examines at least $n/2$ positions, so that at least $n^2/4$ positions are examined in total. Therefore, the time taken by SELECTION-SORT is at least proportional to $n^2/4$, which is $\Omega(n^2)$.

Since the running time of SELECTION-SORT is both $O(n^2)$ and $\Omega(n^2)$, it is $\Theta(n^2)$.

Solution to Exercise 3.1-3

The constant α must be any constant fraction in the range $0 < \alpha < 1/2$.

The lower-bound argument for insertion sort goes as follows. The αn largest values must pass through the middle $(1 - 2\alpha)n$ positions, requiring at least $\alpha n \cdot (1 - 2\alpha)n = (\alpha - 2\alpha^2)n^2$ executions of line 6. This function is $\Omega(n^2)$ because $\alpha - 2\alpha^2$ is a positive constant. It is constant because α is a constant, and it is positive because $\alpha < 1/2$ implies $1 - 2\alpha > 0$, and multiplying both sides by α gives $\alpha - 2\alpha^2 > 0$.

To determine the value of α that maximizes the number of times that the αn largest values must pass through the middle $1-2\alpha$ array positions, find the value of α that maximizes $\alpha - 2\alpha^2$. The derivative of this expression with respect to α is $1 - 4\alpha$. Setting this derivative to 0 and solving for α gives $\alpha = 1/4$.

Solution to Exercise 3.2-1

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

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

Since $f(n)$ and $g(n)$ are asymptotically nonnegative, there exists n_0 such that $f(n) \ge 0$ and $g(n) \ge 0$ for all $n \ge n_0$. Thus for $n \ge n_0$, $f(n) + g(n) \ge f(n) \ge 0$ and $f(n) + g(n) \ge g(n) \ge 0$. Since for any particular n, $h(n)$ is either $f(n)$ or $g(n)$, we have $f(n) + g(n) \ge h(n) \ge 0$, which shows that $h(n) = \max\{f(n), g(n)\} \le c_2(f(n) + g(n))$ for all $n \ge n_0$ (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 $n \ge n_0$, $0 \le f(n) \le h(n)$ and $0 \le g(n) \le h(n)$. Adding these two inequalities yields $0 \le f(n) + g(n) \le 2h(n)$, or equivalently $0 \le (f(n) + g(n))/2 \le h(n)$, which shows that $h(n) = \max\{f(n), g(n)\} \ge c_1(f(n) + g(n))$ for all $n \ge n_0$ (with $c_1 = 1/2$ in the definition of Θ).

Solution to Exercise 3.2-2
This solution is also posted publicly

Since O-notation provides only an upper bound, and not a tight bound, the statement is saying that the running of time of algorithm A is at least a function whose rate of growth is at most n^2.

Solution to Exercise 3.2-3
This solution is also posted publicly

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

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

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

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

$0 \leq 2^{2n} \leq c \cdot 2^n$ for all $n \geq n_0$.

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

Solution to Exercise 3.2-4

If $f(n) = \Omega(g(n))$ then there exist $c_1 > 0$ and n_1 such that $f(n) \geq c_1 g(n)$ for all $n > n_1$. Furthermore, if $f(n) = O(g(n))$ then there exist $c_2 > 0$ and n_2 such that $f(n) \leq c_2 g(n)$ for all $n > n_2$. Therefore, if we set $n_0 = \max\{n_1, n_2\}$, then for all $n > n_0$, we have $c_1 g(n) \leq f(n) \leq c_2 g(n)$, which shows that $f(n) = \Theta(g(n))$.

The other direction is even simpler. Suppose $f(n) = \Theta(g(n))$. Then there exist $c_1, c_2 > 0$ and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$, which immediately shows that $f(n) = \Omega(g(n))$, as well as $f(n) = O(g(n))$.

Solution to Exercise 3.2-5

If the worst-case running time is $O(g(n))$, then the running time is $O(g(n))$ in all cases. Likewise, if the best-case running time is $\Omega(g(n))$, then the running time is $\Omega(g(n))$ in all cases. By Theorem 3.1, therefore, the runnimg time is $\Theta(g(n))$.

Solution to Exercise 3.2-6

Suppose that $f(n) \in o(g(n))$. Then, for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$. Now suppose that $f(n) \in \omega(g(n))$ as well. Then, for any positive constant $c > 0$, there exists a constant $n'_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n'_0$. Fix some positive constant c, and let $n''_0 = \max\{n_0, n'_0\}$. Then we would have, for all $n \ge n''_0$, both $0 \le f(n) < cg(n)$ and $0 \le cg(n) < f(n)$, a contradiction.

Solution to Exercise 3.2-7

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

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

Solution to Exercise 3.3-2

Equation (3.3) gives $\lfloor \alpha n \rfloor = -\lceil -\alpha n \rceil$, so that $\lfloor \alpha n \rfloor + \lceil -\alpha n \rceil = 0$. Thus, we have

$$\begin{aligned}\lfloor \alpha n \rfloor + \lceil (1-\alpha)n \rceil &= \lfloor \alpha n \rfloor + \lceil n - \alpha n \rceil \\ &= \lfloor \alpha n \rfloor + \lceil -\alpha n \rceil + n && \text{(by equation (3.10))} \\ &= n && (\lfloor \alpha n \rfloor + \lceil -\alpha n \rceil = 0)\ .\end{aligned}$$

Solution to Exercise 3.3-5
This solution is also posted publicly

$\lceil \lg n \rceil!$ 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 $f(n)$ is polynomially bounded, then there exist positive constants c, k, and n_0 such that $0 \le f(n) \le cn^k$ for all $n \ge n_0$. Without loss of generality, assume that $c \ge 1$, since if $c < 1$, then $f(n) \le cn^k$ implies that $f(n) \le n^k$. Assume also that $n_0 \ge 2$, so that $n \ge n_0$ implies that $\lg c \le (\lg c)(\lg n)$. Then, we have

  $$\begin{aligned}\lg f(n) &\le \lg c + k \lg n \\ &\le (\lg c + k) \lg n\ ,\end{aligned}$$

  which, since c and k are constants, means that $\lg f(n) = O(\lg n)$.

- Now suppose that $\lg f(n) = O(\lg n)$. Then there exist positive constants c and n_0 such that $0 \le \lg f(n) \le c \lg n$ for all $n \ge n_0$. Then, we have

  $$0 \le f(n) = 2^{\lg f(n)} \le 2^{c \lg n} = (2^{\lg n})^c = n^c$$

  for all $n \ge n_0$, so that $f(n)$ 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 (3.28)).
2. $\lceil \lg n \rceil = \Theta(\lg n)$, because
   - $\lceil \lg n \rceil \ge \lg n$, and
   - $\lceil \lg n \rceil < \lg n + 1 \le 2 \lg n$ for all $n \ge 2$.

We have

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

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

We also have

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

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 Exercise 3.3-6

$\lg^*(\lg n)$ is asymptotically larger because $\lg^*(\lg n) = \lg^* n - 1$.

Solution to Exercise 3.3-7

Both ϕ^2 and $\phi + 1$ equal $(3 + \sqrt{5})/2$, and both $\widehat{\phi}^2$ and $\widehat{\phi} + 1$ equal $(3 - \sqrt{5})/2$.

Solution to Exercise 3.3-8

We have two base cases: $i = 0$ and $i = 1$. For $i = 0$, we have

$$\begin{aligned}
\frac{\phi^0 - \widehat{\phi}^0}{\sqrt{5}} &= \frac{1-1}{\sqrt{5}} \\
&= 0 \\
&= F_0 ,
\end{aligned}$$

and for $i = 1$, we have

$$\begin{aligned}
\frac{\phi^1 - \widehat{\phi}^1}{\sqrt{5}} &= \frac{(1+\sqrt{5}) - (1-\sqrt{5})}{2\sqrt{5}} \\
&= \frac{2\sqrt{5}}{2\sqrt{5}} \\
&= 1 \\
&= F_1 .
\end{aligned}$$

For the inductive case, the inductive hypothesis is that $F_{i-1} = (\phi^{i-1} - \widehat{\phi}^{i-1})/\sqrt{5}$ and $F_{i-2} = (\phi^{i-2} - \widehat{\phi}^{i-2})/\sqrt{5}$. We have

$$\begin{aligned}
F_i &= F_{i-1} + F_{i-2} && \text{(equation (3.31))} \\
&= \frac{\phi^{i-1} - \widehat{\phi}^{i-1}}{\sqrt{5}} + \frac{\phi^{i-2} - \widehat{\phi}^{i-2}}{\sqrt{5}} && \text{(inductive hypothesis)} \\
&= \frac{\phi^{i-2}(\phi+1) - \widehat{\phi}^{i-2}(\widehat{\phi}+1)}{\sqrt{5}} \\
&= \frac{\phi^{i-2}\phi^2 - \widehat{\phi}^{i-2}\widehat{\phi}^2}{\sqrt{5}} && \text{(Exercise 3.3-7)} \\
&= \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} .
\end{aligned}$$

Solution to Exercise 3.3-9

If $k \lg k = \Theta(n)$, then the symmetry property on page 61 implies that $n = \Theta(k \lg k)$. Taking the natural logarithm of both sides gives $\lg n = \Theta(\lg(k \lg k)) = \Theta(\lg k + \lg\lg k) = \Theta(\lg k)$ (dropping the low-order term $\lg\lg k$). Thus, we have

$$\frac{n}{\lg n} = \frac{\Theta(k \lg k)}{\Theta(\lg k)} = \Theta\left(\frac{k \lg k}{\lg k}\right) = \Theta(k) .$$

Applying the symmetry property again gives $k = \Theta(n/\lg n)$.

Solution to Problem 3-2

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ	yes	yes	no	no	no
b.	n^k	c^n	yes	yes	no	no	no
c.	$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no
d.	2^n	$2^{n/2}$	no	no	yes	yes	no
e.	$n^{\lg c}$	$c^{\lg n}$	yes	no	yes	no	yes
f.	$\lg(n!)$	$\lg(n^n)$	yes	no	yes	no	yes

Reasons:

a. Any polylogarithmic function is little-oh of any polynomial function with a positive exponent.

b. Any polynomial function is little-oh of any exponential function with a positive base.

c. The function $\sin n$ oscillates between -1 and 1. There is no value n_0 such that $\sin n$ is less than, greater than, or equal to $1/2$ for all $n \geq n_0$, and so there is no value n_0 such that $n^{\sin n}$ is less than, greater than, or equal to $cn^{1/2}$ for all $n \geq n_0$.

d. Take the limit of the quotient: $\lim_{n\to\infty} 2^n/2^{n/2} = \lim_{n\to\infty} 2^{n/2} = \infty$.

e. By equation (3.21), these quantities are equal.

f. By equation (3.28), $\lg(n!) = \Theta(n \lg n)$. Since $\lg(n^n) = n \lg n$, these functions are Θ of each other.

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 (3.28)).
7. $n! = \Theta(n^{n+1/2}e^{-n})$ by dropping constants and low-order terms in equation (3.25).
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 $\Omega(g_i(n))$.

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

Solution to Problem 3-4

a. The conjecture is false. For example, let $f(n) = n$ and $g(n) = n^2$. Then $f(n) = O(g(n))$, but $g(n)$ is not $O(f(n))$.

b. The conjecture is false. Again, let $f(n) = n$ and $g(n) = n^2$. Then the conjecture would be saying that $n + n^2 = \Theta(n)$, which is false.

c. The conjecture is true. Since $f(n) = O(g(n))$ and $f(n) \geq 1$ for sufficiently large n, there are some positive constants c and n_0 such that $1 \leq f(n) \leq cg(n)$ for all $n \geq n_0$, which implies $0 \leq \lg f(n) \leq \lg c + \lg g(n)$. Without loss of generality, assume that $c > 1/2$, so that $\lg c > -1$. Define the constant $d = 1 + \lg c > 0$. Then, we have

$$\begin{aligned} \lg f(n) &\leq \lg c + \lg g(n) \\ &= \left(1 + \frac{\lg c}{\lg g(n)}\right) \lg g(n) \\ &\leq (1 + \lg c) \lg g(n) \qquad \text{(because } \lg g(n) \geq 1\text{)} \\ &= d \lg g(n) \text{ ,} \end{aligned}$$

and so there exist positive constants d and n_0 such that $0 \leq \lg f(n) \leq d \lg g(n)$ for $n \geq n_0$. Thus, $\lg f(n) = O(\lg g(n))$.

d. The conjecture is false. For example, let $f(n) = 2n$ and $g(n) = n$. Then $f(n) = O(g(n))$, but $2^{f(n)} = 2^{2n}$ and $2^{g(n)} = 2^n$, so that $2^{f(n)}$ is not $O(2^{g(n)})$.

e. The conjecture is false. For example, let $f(n) = 1/n$, so that $f(n)^2 = 1/n^2$. It is not the case that $1/n = O(1/n^2)$.

f. The conjecture is true, by transpose symmetry on page 62.

g. The conjecture is false. Let $f(n) = 2^n$. It is not the case that 2^n is $\Theta(2^{n/2})$.

h. The conjecture is true. Let $g(n)$ be any function in $o(f(n))$. Then there exists a constant $n_0 > 0$ such that for any positive constant $c > 0$ and all $n \geq n_0$, we have $0 \leq g(n) < cf(n)$. Since $f(n) + g(n) \geq f(n)$, we have $f(n) + g(n) = \Omega(f(n))$. For the upper bound, choose the n_0 used for $g(n)$ and choose any constant $c > 0$. Then, we have

$$\begin{aligned} 0 &\leq f(n) + g(n) \\ &< f(n) + cf(n) \\ &= (1 + c)f(n) \\ &\leq c'f(n) \end{aligned}$$

for the constant $c' = 1 + c$. Therefore, $f(n) + g(n) = O(f(n))$, so that $f(n) + g(n) = \Theta(f(n))$.

Solution to Problem 3-7

a. $f_0(n) = n$. Since $f(n)$ just subtracts 1, the answer is how many times you subtract 1 from n before reaching 0, which is just n.

b. $f_1(\lg n) = \lg^* n$. This answer comes directly from the definition of the iterated logarithm function.

c. $f_1(n/2) = \lceil \lg n \rceil$. This result is easily shown by induction for n a power of 2. The ceiling function handles values of n between powers of 2.

d. $f_2(n/2) = \lceil \lg n \rceil - 1$. Take the answer for part (c), but halve one fewer time.

e. $f_2(\sqrt{n}) = \lceil \lg \lg n \rceil$. Define $m = \lg n$, so that $n = 2^m$. The problem then becomes determining $f_1((2^m)^{1/2}) = f_1(2^{m/2})$. (It's $f_1(2^{m/2})$ instead of $f_2(2^{m/2})$ because $n = 2$ implies $m = 1$.) By part (c), the answer is $\lceil \lg m \rceil = \lceil \lg \lg n \rceil$.

f. $f_1(\sqrt{n})$ is undefined. No matter how many times you take the square root of $n > 1$, you will never reach 1.

g. $\lceil \log_3 \log_3 n \rceil \leq f_2(n^{1/3}) \leq \lceil \log_3 \log_3 n \rceil + 1$. Similar to the solution to part (e), let $n = 3^m$ and $m = \log_3 n$, so that the problem becomes finding $f_{\log_3 2}(3^{m/3})$. As in part (c), the number of times you divide by 3 before reaching 1 is $\lceil \log_3 m \rceil = \lceil \log_3 \log_3 n \rceil$. Since $\log_3 2 < 1$, however, you might need to iterate one more time to reach $\log_3 2$.

Lecture Notes for Chapter 4: Divide-and-Conquer

Chapter 4 overview

Recall the divide-and-conquer paradigm, which we used for merge sort:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

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

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

We look at two algorithms for multiplying square matrices, based on divide-and-conquer.

Analyzing divide-and-conquer algorithms

Use a recurrence to characterize the running time of a divide-and-conquer algorithm. Solving the recurrence gives us the asymptotic running time.

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

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

A recurrence could have 0, 1, or more functions that satisfy it. ***Well defined*** if at least 1 function satisfies; otherwise, ***ill defined***.

Algorithmic recurrences

Interested in recurrences that describe running times of algorithms.

A recurrence $T(n)$ is ***algorithmic*** if for every sufficiently large ***threshold*** constant $n_0 > 0$:

- For all $n < n_0$, $T(n) = \Theta(1)$. *[Can consider the running time constant for small problem sizes.]*
- For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations. *[The recursive algorithm terminates.]*

Conventions

Will often state recurrences without base cases. When analyzing algorithms, assume that if no base case is given, the recurrence is algorithmic. Allows us to pick any sufficiently large threshold constant n_0 without changing the asymptotic behavior of the solution.

Ceilings and floors in divide-and-conquer recurrences don't change the asymptotic solution $\Rightarrow$ often state algorithmic recurrences without floors and ceilings, even though to be precise, they should be there. *[Example: recurrence for merge sort is really $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$.]*

Some recurrences are inequalities rather than equations.
Example: $T(n) \le 2T(n/2) + \Theta(n)$ gives only an upper bound on $T(n)$, so state the solution using O-notation rather than Θ-notation.

Examples of recurrences arising from divide-and-conquer algorithms

- $n \times n$ matrix multiplication by breaking into 8 subproblems of size $n/2 \times n/2$: $T(n) = 8T(n/2) + \Theta(1)$. Solution: $T(n) = \Theta(n^3)$. *[The first printing of the fourth edition says* 4 *subproblems. That is an error.]*
- Strassen's algorithm for $n \times n$ matrix multiplication by breaking into 7 subproblems of size $n/2 \times n/2$: $T(n) = 7T(n/2) + \Theta(1)$. Solution: $T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$.
- An algorithm that breaks a problem of size n into one subproblem of size $n/3$ and another of size $2n/3$, taking $\Theta(n)$ time to divide and combine: $T(n) = T(n/3) + T(2n/3) + \Theta(n)$. Solution: $T(n) = \Theta(n \lg n)$.
- An algorithm that breaks a problem of size n into one subproblem of size $n/5$ and another of size $7n/10$, taking $\Theta(n)$ time to divide and combine: $T(n) = T(n/5) + T(7n/10) + \Theta(n)$. Solution: $T(n) = \Theta(n)$. *[This is the recurrence for order-statistic algorithm in Chapter 9 that takes linear time in the worst case.]*
- Subproblems don't always have to be a constant fraction of the original problem size. ***Example:*** recursive linear search creates one subproblem and it has one element less than the original problem. Time to divide and combine is $\Theta(1)$, giving $T(n) = T(n-1) + \Theta(1)$. Solution: $T(n) = \Theta(n)$.

Methods for solving recurrences

The chapter contains four methods for solving recurrences. Each gives asymptotic bounds.

- Substitution method: Guess the solution, then use induction to prove that it's correct.
- Recursion-tree method: Draw out a recursion tree, determine the costs at each level, and sum them up. Useful for coming up with a guess for the substitution method.
- Master method: A cookbook method for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a > 0$ and $b > 1$ are constants, subject to certain

conditions. Requires memorizing three cases, but applies to many divide-and-conquer algorithms.

- Akra-Bazzi method: A general method for solving divide-and-conquer recurrences. Requires calculus, but applies to recurrences beyond those solved by the master method. *[These lecture notes do not cover the Akra-Bazzi method.]*

[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.]

Multiplying square matrices

Input: Three $n \times n$ (square) matrices, $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$.

Result: The matrix product $A \cdot B$ is added into C, so that

$$c_{ij} = c_{ij} + \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

for $i, j = 1, 2, \ldots, n$.

If only the product $A \cdot B$ is needed, then zero out all entries of C beforehand.

Straightforward method

```
MATRIX-MULTIPLY(A, B, C, n)
  for i = 1 to n                          // compute entries in each of n rows
      for j = 1 to n                      // compute n entries in row i
          for k = 1 to n
              $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$    // add in another term
```

Time: $\Theta(n^3)$ because of triply nested loops.

Simple divide-and-conquer algorithm

For simplicity, assume that C is initialized to 0, so computing $C = A \cdot B$.

If $n > 1$, partition each of A, B, C into four $n/2 \times n/2$ matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Rewrite $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} ,$$

giving the four equations

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} . \end{aligned}$$

Each of these equations multiplies two $n/2 \times n/2$ matrices and then adds their $n/2 \times n/2$ products. Assume that n is an exact power of 2, so that submatrix dimensions are always integer.

Use these equations to get a divide-and-conquer algorithm:

```
MATRIX-MULTIPLY-RECURSIVE$(A, B, C, n)$
  if $n == 1$
      // Base case.
      $c_{11} = c_{11} + a_{11} \cdot b_{11}$
      return
  // Divide.
  partition $A$, $B$, and $C$ into $n/2 \times n/2$ submatrices
      $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$;
      and $C_{11}, C_{12}, C_{21}, C_{22}$; respectively
  // Conquer.
  MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{11}, C_{11}, n/2)$
  MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{12}, C_{12}, n/2)$
  MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{11}, C_{21}, n/2)$
  MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{12}, C_{22}, n/2)$
  MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{21}, C_{11}, n/2)$
  MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{22}, C_{12}, n/2)$
  MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{21}, C_{21}, n/2)$
  MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{22}, C_{22}, n/2)$
```

[The book briefly discusses the question of how to avoid copying entries when partitioning matrices. Can partition matrices without copying entries by instead using index calculations. Identify a submatrix by ranges of row and column matrices from the original matrix. End up representing a submatrix differently from how we represent the original matrix. The advantage of avoiding copying is that partitioning would take only constant time, instead of $\Theta(n^2)$ time. The result of the asymptotic analysis won't change, but using index calculations to avoid copying gives better constant factors.]

Analysis

Let $T(n)$ be the time to multiply two $n \times n$ matrices.

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$.

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time, using index calculations. *[Otherwise,* $\Theta(n^2)$ *time.]*
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow$ $8T(n/2)$.
- No combine step, because C is updated in place.

Recurrence (omitting the base case) is $T(n) = 8T(n/2) + \Theta(1)$. Can use master method to show that it has solution $T(n) = \Theta(n^3)$.
Asymptotically, no better than the obvious method.

Bushiness of recursion trees: Compare this recurrence with the merge-sort recurrence $T(n) = 2T(n/2) + \Theta(n)$. If we draw out the recursion trees, the factor of 2 in the merge-sort recurrenece says that each non-leaf node has 2 children. But the factor of 8 in the recurrence for MATRIX-MULTIPLY-RECURSIVE says that each non-leaf node has 8 children. Get a bushier tree with many more leaves, even though internal nodes have a smaller cost.

Strassen's algorithm

Idea: Make the recursion tree less bushy. Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8. Will cost several additions/subtractions of $n/2 \times n/2$ matrices.

Since a subtraction is a "negative addition," just refer to all additions and subtractions as additions.

Example of reducing multiplications: Given x and y, compute $x^2 - y^2$. Obvious way uses 2 multiplications and one subtraction. But observe:

$$\begin{aligned} x^2 - y^2 &= x^2 - xy + xy - y^2 \\ &= x(x - y) + y(x - y) \\ &= (x + y)(x - y) , \end{aligned}$$

so at the expense of one extra addition, can get by with only 1 multiplication. Not a big deal if x, y are scalars, but can make a difference if they are matrices.

The algorithm:

1. Same base case as before, when $n = 1$.
2. When $n > 1$, then as in the recursive method, partition each of the matrices into four $n/2 \times n/2$ submatrices. Time: $\Theta(1)$, using index calculations.
3. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in previous step. Time: $\Theta(n^2)$ to create all 10 matrices.
4. Create and zero the entries of 7 matrices $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$. Time: $\Theta(n^2)$.
5. Using the submatrices of A and B and the matrices $S_1, S_2, \ldots, S_{10}$, recursively compute $P_1, P_2, \ldots, P_7$. Time: $7T(n/2)$.
6. Update the four $n/2 \times n/2$ submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of C by adding and subtracting various combinations of the P_i. Time: $\Theta(n^2)$.

Analysis

Recurrence will be $T(n) = 7T(n/2) + \Theta(n^2)$. By the master method, solution is $T(n) = \Theta(n^{\lg 7})$. Since $\lg 7 < 2.81$, the running time is $O(n^{2.81})$, beating the $\Theta(n^3)$-time methods.

Details

Step 2: Create the 10 matrices

$$\begin{aligned}
S_1 &= B_{12} - B_{22} \, , \\
S_2 &= A_{11} + A_{12} \, , \\
S_3 &= A_{21} + A_{22} \, , \\
S_4 &= B_{21} - B_{11} \, , \\
S_5 &= A_{11} + A_{22} \, , \\
S_6 &= B_{11} + B_{22} \, , \\
S_7 &= A_{12} - A_{22} \, , \\
S_8 &= B_{21} + B_{22} \, , \\
S_9 &= A_{11} - A_{21} \, , \\
S_{10} &= B_{11} + B_{12} \, .
\end{aligned}$$

Add or subtract $n/2 \times n/2$ matrices 10 times ⇒ time is $\Theta(n^2)$.

Step 4: Compute the 7 matrices

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \, , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \, , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \, , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \, , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \, , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \, , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \, .
\end{aligned}$$

The only multiplications needed are in the middle column; right-hand column just shows the products in terms of the original submatrices of A and B.

Step 5: Add and subtract the P_i to construct submatrices of C:

$$\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \, , \\
C_{12} &= P_1 + P_2 \, , \\
C_{21} &= P_3 + P_4 \, , \\
C_{22} &= P_5 + P_1 - P_3 - P_7 \, .
\end{aligned}$$

To see how these computations work, expand each right-hand side, replacing each P_i with the submatrices of A and B that form it, and cancel terms: *[We expand out all four right-hand sides here. You might want to do just one or two of them, to convince students that it works.]*

$$\begin{array}{lllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & & \\
 & - A_{11} \cdot B_{22} & & & & - A_{12} \cdot B_{22} & \\
 & & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & & & & + A_{12} \cdot B_{21}
\end{array}$$

$$\begin{array}{lll}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} & \\
 & + A_{11} \cdot B_{22} & + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22}
\end{array}$$

$$\begin{array}{lll}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} & \\
 & - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21}
\end{array}$$

$$\begin{array}{lllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12}
\end{array}$$

Theoretical and practical notes

Strassen's algorithm was the first to beat $\Theta(n^3)$ time, but it's not the asymptotically fastest known. A method by Coppersmith and Winograd runs in $O(n^{2.376})$ time. Current best asymptotic bound (not practical) is $O(n^{2.37286})$.

Practical issues against Strassen's algorithm:

- Higher constant factor than the obvious $\Theta(n^3)$-time method.
- Not good for sparse matrices.
- Not numerically stable: larger errors accumulate than in the obvious method.
- Submatrices consume space, especially if copying.

Numerical stability problem is not as bad as previously thought. And can use index calculations to reduce space requirement.

Various researchers have tried to find the crossover point, where Strassen's algorthm runs faster than the obvious $\Theta(n^3)$-time method. Answers vary.

Substitution method

1. Guess the solution.

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

Usually use the substitution method to establish either an upper bound (O-bound) or a lower bound (Ω-bound).

Example

Determine an asymptotic upper bound on $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$. Similar to the merge-sort recurrence except for the floor function. (Ensures that $T(n)$ is defined over integers.)

Guess same asymptotic upper bound as merge-sort recurrence: $T(n) = O(n \lg n)$.

Inductive hypothesis: $T(n) \leq cn \lg n$ for all $n \geq n_0$. Will choose constants $c, n_0 > 0$ later, once we know their constraints.

Inductive step: Assume that $T(n) \leq cn \lg n$ for all numbers $\geq n_0$ and $< n$. If $n \geq 2n_0$, holds for $\lfloor n/2 \rfloor \Rightarrow T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$. Substitute into the recurrence:

$$\begin{aligned}
T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + \Theta(n) \\
&\leq 2(c(n/2) \lg(n/2)) + \Theta(n) \\
&= cn \lg(n/2) + \Theta(n) \\
&= cn \lg n - cn \lg 2 + \Theta(n) \\
&= cn \lg n - cn + \Theta(n) \\
&\leq cn \lg n \ .
\end{aligned}$$

The last step holds if c, n_0 are sufficiently large that for $n \geq 2n_0$, cn dominates the $\Theta(n)$ term.

Base cases: Need to show that $T(n) \leq cn \lg n$ when $n_0 \leq n < 2n_0$. Add new constraint: $n_0 > 1 \Rightarrow \lg n > 0 \Rightarrow n \lg n > 0$. Pick $n_0 = 2$. Because no base case is given in the recurrence, it's algorithmic $\Rightarrow T(2), T(3)$ are constant. Choose $c = \max\{T(2), T(3)\} \Rightarrow T(2) \leq c < (2 \lg 2)c$ and $T(3) \leq c < (3 \lg 3)c \Rightarrow$ inductive hypothesis established for the base cases.

Wrap up: Have $T(n) \leq cn \lg n$ for all $n \geq 2 \Rightarrow T(n) = O(n \lg n)$.

In practice: Don't usually write out substitution proofs this detailed, especially regarding base cases. For most algorithmic recurrences, the base cases are handled the same way.

Making a good guess

No general way to make a good guess. Experience helps. Can also draw out a recursion tree.

If the recurrence is similar to one you've seen before, try guessing a similar solution. ***Example:*** $T(n) = 2T(n/2 + 17) + \Theta(n)$. This looks a lot like the merge-sort recurrence $(n) = 2T(n/2) + \Theta(n)$ except for the added 17. When n is large, the difference between $n/2$ and $n/2 + 17$ is small, since both cut n nearly in half. Guess that the solution to the merge-sort recurrence, $T(n) = O(n \lg n)$ works here. (It does.)

When the additive term uses asymptotic notation

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

[In the book, we don't show how to handle this situation until Section 4.4.]

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.

Important: We get to name the constant hidden in the asymptotic notation (c in this case), but we do *not* get to choose it, other than assume that it's enough to handle the base case of the recursion.

1. ***Upper bound:***

   Guess: $T(n) \le dn \lg n$ for some positive constant d. This is the inductive hypothesis.

   Important: We get to both name and choose the constant in the inductive hypothesis (d in this case). It OK for the constant in the inductive hypothesis (d) to depend on the constant hidden in the asymptotic notation (c).

   Substitution:

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

   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:

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

   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.]* ■

Subtracting a low-order term

Might guess the right asymptotic bound, but the math doesn't go through in the proof. Resolve by subtracting a lower-order term.

Example

$T(n) = 2T(n/2) + \Theta(1)$. Guess that $T(n) = O(n)$, and try to show $T(n) \le cn$ for $n \ge n_0$, where we choose c, n_0:

$$\begin{aligned} T(n) &\le 2(c(n/2)) + \Theta(1) \\ &= cn + \Theta(1) \,. \end{aligned}$$

But this doesn't say that $T(n) \le cn$ for *any* choice of c.

Could try a larger guess, such as $T(n) = O(n^2)$, but not necessary. We're off only by $\Theta(1)$, a lower-order term. Try subtracting a lower-order term in the guess: $T(n) \le cn - d$, where $d \ge 0$ is a constant:

$$\begin{aligned} T(n) &\le 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &\le cn - d - (d - \Theta(1)) \\ &\le cn - d \end{aligned}$$

as long as d is larger than the constant in $\Theta(1)$.

Why subtract off a lower-order term, rather than add it? Notice that it's subtracted twice. Adding a lower-order term twice would take us further away from the inductive hypothesis. Subtracting it twice gives us $T(n) \le cn - d - (d - \Theta(1))$, and it's easy to choose d to make that inequality hold.

Important: Once again, we get to name and choose the constant c in the inductive hypothesis. And we also get to name and choose the constant d that we subtract off.

Be careful when using asymptotic notation

A false proof for the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$, that $T(n) = O(n)$:

$$\begin{aligned} T(n) &\le 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n) \,. \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

This "proof" changes the constant in the Θ-notation. Can see this by using an explicit constant. Assume $T(n) \le cn$ for all $n \ge n_0$:

$$\begin{aligned} T(n) &\le 2(c\,\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) \,, \end{aligned}$$

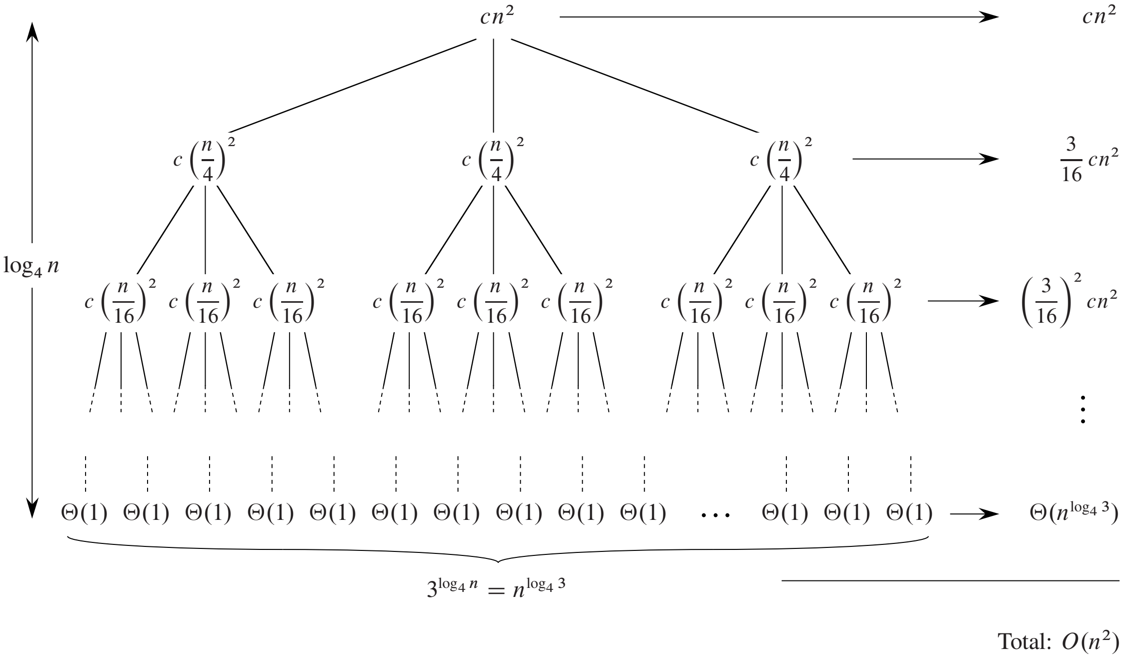
but $cn + \Theta(n) > cn$.

Recursion trees

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

Example

$T(n) = 3T(n/4) + \Theta(n^2)$.
Draw out a recursion tree for $T(n) = 3T(n/4) + cn^2$:



cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$\log_4 n$
$\Theta(1)$
$\cdots$
$3^{\log_4 n} = n^{\log_4 3}$
cn^2
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
Total: $O(n^2)$

[You might want to draw it out progressively, as in Figure 4.1 in the book.]

For simplicity, assume that n is a power of 4 and the base case is $T(1) = \Theta(1)$. Subproblem size for nodes at depth i is $n/4^i$. Get to base case when $n/4^i = 1 \Rightarrow n = 4^i \Rightarrow i = \log_4 n$.

Each level has 3 times as many nodes as the level above, so that depth i has 3^i nodes. Each internal node at depth i has cost $c(n/4^i)^2 \Rightarrow$ total cost at depth i (except for leaves) is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. Bottom level has depth $\log_4 n \Rightarrow$ number of leaves is $3^{\log_4 n} = n^{\log_4 3}$. Since each leaf contributes $\Theta(1)$, total cost of leaves is $\Theta(n^{\log_4 3})$.

Add up costs over all levels to determine cost for the entire tree:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) \, .
\end{aligned}$$

Idea: Coefficients of cn^2 form a decreasing geometric series. Bound it by an infinite series, and get a bound of 16/13 on the coefficients.

Use substitution method to verify $O(n^2)$ upper bound. Show that $T(n) \le dn^2$ for constant $d > 0$:

$$\begin{aligned} T(n) &\le 3T(n/4) + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2 , \end{aligned}$$

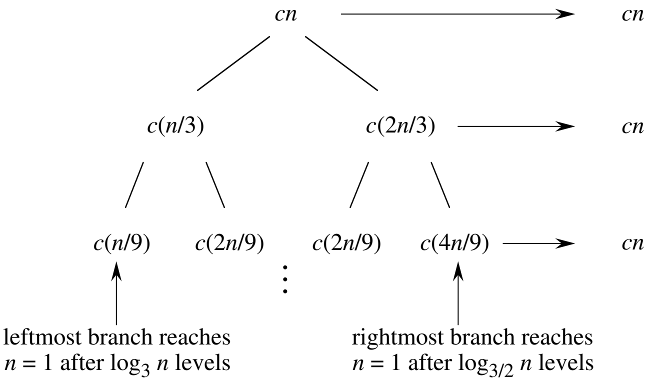
by choosing $d \ge (16/13)c$. *[Again, we get to name but not choose c, and we get to name and choose d.]*

That gives an upper bound of $O(n^2)$. The lower bound of $\Omega(n^2)$ is obvious because the recurrence contains a $\Theta(n^2)$ term. Hence, $T(n) = \Theta(n^2)$.

Irregular 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):



[This is a simpler way to draw the recursion tree than in Figure 4.2 in the book.]

- There are $\log_3 n$ full levels (going down the left side), and after $\log_{3/2} n$ levels, the problem size is down to 1 (going down the right side).
- 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:

   $$\begin{aligned} T(n) &\le T(n/3) + T(2n/3) + cn \\ &\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \end{aligned}$$

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

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

[As before, can name but not choose the constant c hidden in the additive term of $\Theta(n)$. Can both name and choose the constant d in the guess (inductive hypothesis).]

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 $T(n) = \Omega(n\lg n)$, conclude that $T(n) = \Theta(n\lg n)$.

[Omitting the analysis for the number of leaves.]

Master method

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

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

where $a \ge 1$, $b > 1$, and $f(n)$ is an asymptotically nonnegative function defined over all sufficiently large positive numbers.

Master recurrences describe recursive algorithms that divide a problem of size n into a subproblems, each of size n/b. Each recursive subproblem takes time $T(n/b)$ (unless it's a base case). Call $f(n)$ the ***driving function***.

In reality, subproblem sizes are integers, so that the real recurrence is more like

$$T(n) = a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil) + f(n)\,,$$

where $a', a'' \ge 0$ and $a' + a'' = a$. Ignoring floors and ceilings does not change the asymptotic solution to the recurrence.

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

Let $a, b, n_0 > 0$ be constants, $f(n)$ be a driving function defined and nonnegative on all sufficiently large reals. Define recurrence $T(n)$ on $n \in \mathbb{N}$ by

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

and where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a', a'' \geq 0$ satsifying $a = a' + a''$.

[The mathematics requires only that $a > 0$, but since in practice the number of subproblems is at least 1, the recurrences we see all have $a \geq 1$.]

Then you can solve the recurrence by comparing $n^{\log_b a}$ 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 $k \geq 0$ is a constant.
($f(n)$ 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: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.
[In the previous editions of the book, case 2 was stated for only $k = 0$.]

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.]

Call $n^{\log_b a}$ the ***watershed function***. Master method compares the driving function $f(n)$ with the watershed function $n^{\log_b a}$.

- If the watershed function grows *polynomially faster* than the driving function, then case 1 applies. Example (not likely to see in algorithm analysis): $T(n) = 4T(n/2) + n^{1.99}$. Watershed function is $n^{\log_2 4} = n^2$, which is polynomially larger than $n^{1.99}$ by a factor of $n^{0.01}$. Case 1 would apply $\Rightarrow T(n) = \Theta(n^2)$.
- If the driving function grows *polynomially faster* than the watershed function and the regularity condition holds, then case 3 applies. Example: $T(n) = 4T(n/2) + n^{2.01}$. Now the driving function is polynomially larger than the watershed function by a factor of $n^{0.01}$. Case 3 would apply $\Rightarrow T(n) = \Theta(n^{2.01})$.

- There are gaps between cases 1 and 2 and between cases 2 and 3. Example: $T(n) = 2T(n/2) + n/\lg n \Rightarrow$ watershed function is $n^{\log_2 2} = n$ and driving function is $f(n) = n/\lg n$. Have $f(n) = o(n)$, so that $f(n)$ grows more slowly than n, it doesn't grow *polynomially slower*. In terms of the master theorem, have $f(n) = n \lg^{-1} n$, so that $k = -1$. Master theorem holds only for $k \geq 0$, so case 2 does not apply.

Examples *[different from those in the book]*

- $T(n) = 5T(n/2) + \Theta(n^2)$
  $n^{\log_2 5}$ 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): $af(n/b) = 5(n/2)^3 = 5n^3/8 \leq cn^3$ 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. $n^3/\lg n = n^3 \lg^{-1} n \neq \Theta(n^3 \lg^k n)$ for any $k \geq 0$.
  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 4.6 of the text has the full proof of the continuous version of the master theorem, and Section 4.7 discusses the technicalities of floors and ceilings in recurrences. Section 4.7 also briefly covers the Akra-Bazzi method, which applies to divide-and-conquer recurrences such as $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.]

Solutions for Chapter 4: Divide-and-Conquer

Solution to Exercise 4.1-1

The easiest solution is to pad out the matrices with zeros so that their dimensions are the next higher power of 2. If the matrices are padded out to be $n' \times n'$, we have $n < n' < 2n$. Run MATRIX-MULTIPLY-RECURSIVE on the padded matrices and then take just the leading $n \times n$ submatrix of the result. Because $n' < 2n$, the padded matrices have less than $4n^2$ entries, and so we can create them in $\Theta(n^2)$ time. And because $n' < 2n$, the running time for MATRIX-MULTIPLY-RECURSIVE increases by at most a factor of 8, so that it still runs in $\Theta(n^3)$ time. Finally, extracting the leading $n \times n$ submatrix takes $\Theta(n^2)$ time, for a total running time of $\Theta(n^3)$.

Solution to Exercise 4.1-2

For both parts of the question, divide the matrices into k submatrices, each $n \times n$. A $kn \times n$ matrix consists of a column of k submatrices, and an $n \times kn$ matrix consists of a row of k submatrices.

Multiplying a $kn \times n$ matrix by an $n \times kn$ matrix produces a $kn \times kn$ matrix, which has k rows and k columns of $n \times n$ submatrices. Each submatrix is the result of mulitplying two $n \times n$ submatrices. Since there are k^2 submatrices to compute and each one takes $\Theta(n^3)$ time, the total running time is $\Theta(k^2 n^3)$.

Multiplying an $n \times kn$ matrix by a $kn \times n$ matrix produces an $n \times n$ matrix, which you can compute by multiplying the respective submatrices and adding the results together. Multiplying takes $\Theta(kn^3)$ time and adding takes $\Theta(kn^2)$ time, for a total time of $\Theta(kn^3)$.

Solution to Exercise 4.1-3

The recurrence becomes $T(n) = 8T(n/2) + \Theta(n^2)$. You can use the master method in Section 4.5 to show that the solution is $T(n) = \Theta(n^3)$.

Solution to Exercise 4.2-1

Assume that C is initialized to all zeros.

First, compute $S_1, \ldots, S_{10}$:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} &&= 8 - 2 &&= 6 , \\
S_2 &= A_{11} + A_{12} &&= 1 + 3 &&= 4 , \\
S_3 &= A_{21} + A_{22} &&= 7 + 5 &&= 12 , \\
S_4 &= B_{21} - B_{11} &&= 4 - 6 &&= -2 , \\
S_5 &= A_{11} + A_{22} &&= 1 + 5 &&= 6 , \\
S_6 &= B_{11} + B_{22} &&= 6 + 2 &&= 8 , \\
S_7 &= A_{12} - A_{22} &&= 3 - 5 &&= -2 , \\
S_8 &= B_{21} + B_{22} &&= 4 + 2 &&= 6 , \\
S_9 &= A_{11} - A_{21} &&= 1 - 7 &&= -6 , \\
S_{10} &= B_{11} + B_{12} &&= 6 + 8 &&= 14 .
\end{aligned}$$

Next, compute $P_1, \ldots, P_7$:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= 1 \cdot 6 &&= 6 , \\
P_2 &= S_2 \cdot B_{22} &&= 4 \cdot 2 &&= 8 , \\
P_3 &= S_3 \cdot B_{11} &&= 12 \cdot 6 &&= 72 , \\
P_4 &= A_{22} \cdot S_4 &&= 5 \cdot -2 &&= -10 , \\
P_5 &= S_5 \cdot S_6 &&= 6 \cdot 8 &&= 48 , \\
P_6 &= S_7 \cdot S_8 &&= -2 \cdot 6 &&= -12 , \\
P_7 &= S_9 \cdot S_{10} &&= -6 \cdot 14 &&= -84 .
\end{aligned}$$

Finally, compute $C_{11}, C_{12}, C_{21}, C_{22}$:

$$\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 &&= 48 + (-10) &&= 18 , \\
C_{12} &= P_1 + P_2 &&= 6 + 8 &&= 14 , \\
C_{21} &= P_3 + P_4 &&= 72 + (-10) &&= 62 , \\
C_{22} &= P_5 + P_1 - P_3 - P_7 &&= 48 + 6 - 72 - (-84) &&= 66 .
\end{aligned}$$

The result is $C = \begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$.

Solution to Exercise 4.2-2

```
STRASSEN($A, B, C, n$)
  if $n == 1$
      $c_{11} = c_{11} + a_{11} \cdot b_{11}$
  else partition $A$, $B$, and $C$ as in equations (4.2)
      create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$ and $P_1, P_2, \ldots, P_7$
      initialize $P_1, P_2, \ldots, P_7$ to all zeros
      $S_1 = B_{12} - B_{22}$
      $S_2 = A_{11} + A_{12}$
      $S_3 = A_{12} + A_{22}$
      $S_4 = B_{21} - B_{11}$
      $S_5 = A_{11} + A_{22}$
      $S_6 = B_{11} + B_{22}$
      $S_7 = A_{12} - A_{22}$
      $S_8 = B_{21} + B_{22}$
      $S_9 = A_{11} - A_{21}$
      $S_{10} = B_{11} + B_{12}$
      STRASSEN($A_{11}, S_1, P_1, n/2$)
      STRASSEN($S_2, B_{22}, P_2, n/2$)
      STRASSEN($S_3, B_{11}, P_3, n/2$)
      STRASSEN($A_{22}, S_4, P_4, n/2$)
      STRASSEN($S_5, S_6, P_5, n/2$)
      STRASSEN($S_7, S_8, P_6, n/2$)
      STRASSEN($S_9, S_{10}, P_7, n/2$)
      $C_{11} = C_{11} + P_5 + P_4 - P_2 + P_6$
      $C_{12} = C_{12} + P_1 + P_2$
      $C_{21} = C_{21} + P_3 + P_4$
      $C_{22} = C_{22} + P_5 + P_1 - P_3 - P_7$
      combine $C_{11}$, $C_{12}$, $C_{21}$, and $C_{22}$ into $C$
```

Solution to Exercise 4.2-3

This solution is also posted publicly

If you can multiply 3×3 matrices using k multiplications, then you can multiply $n \times n$ matrices by recursively multiplying $n/3 \times n/3$ matrices, in time $T(n) = kT(n/3) + \Theta(n^2)$.

Using the master method to solve this recurrence, consider the ratio of $n^{\log_3 k}$ and n^2:

- If $\log_3 k = 2$, case 2 applies and $T(n) = \Theta(n^2 \lg n)$. In this case, $k = 9$ and $T(n) = o(n^{\lg 7})$.
- If $\log_3 k < 2$, case 3 applies and $T(n) = \Theta(n^2)$. In this case, $k < 9$ and $T(n) = o(n^{\lg 7})$.

- If $\log_3 k > 2$, case 1 applies and $T(n) = \Theta(n^{\log_3 k})$. In this case, $k > 9$. $T(n) = o(n^{\lg 7})$ when $\log_3 k < \lg 7$, i.e., when $k < 3^{\lg 7} \approx 21.85$. The largest such integer k is 21.

Thus, $k = 21$ and the running time is $\Theta(n^{\log_3 k}) = \Theta(n^{\log_3 21}) = O(n^{2.80})$ (since $\log_3 21 \approx 2.77$).

Solution to Exercise 4.2-4

Because Strassen's algorithm has subproblems of size $n/2$ and requires 7 recursive multiplications, the recurrence for analyzing it is $T(n) = 7T(n/2) + \Theta(n^2)$. To generalize, if the subproblems have size n/b and require a recursive multiplications, the recurrence is $T(n) = aT(n/b) + \Theta(n^2)$. Using the master method in Section 4.5 gives the following running times:

- $a = 132{,}464$, $b = 68$: $\Theta(n^{\log_{68} 132{,}464}) = O(n^{2.795129})$.
- $a = 143{,}640$, $b = 70$: $\Theta(n^{\log_{70} 143{,}640}) = O(n^{2.795123})$.
- $a = 155{,}424$, $b = 72$: $\Theta(n^{\log_{72} 155{,}424}) = O(n^{2.795148})$.

Of the three methods that Pan discovered, the middle one—multiplying 70 × 70 matrices using 143,640 multiplications—has the best asymptotic running time. All three are asymptotically faster than Strassen's method, because $\Theta(n^{\lg 7}) = \Omega(n^{2.8})$.

Solution to Exercise 4.2-5

The three multiplications needed are ac, bd, and $(a + b)(c + d) = ac + ad + bc + bd$. With ac and bd, compute the real component $ac - bd$. With ac, bd, and $(a + b)(c + d)$, compute the imaginary component $(a + b)(c + d) - ac - bd = ad + bc$.

Solution to Exercise 4.2-6

Create the $2n \times 2n$ matrix $M = \begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}$, so that $M^2 = \begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix}$. It takes $\Theta(n^2)$ time to create M and extract the product AB from M. The time to square M is $\Theta((2n)^\alpha)$, which is $\Theta(n^\alpha)$ since 2^α is a constant.

Solution to Exercise 4.3-1

a. We guess that $T(n) \le cn^2$ for some constant $c > 0$. We have

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

$$\begin{aligned} &\leq c(n-1)^2 + n \\ &= cn^2 - 2cn + c + n \\ &= cn^2 + c(1-2n) + n \; . \end{aligned}$$

This last quantity is less than or equal to cn^2 if $c(1-2n) + n \leq 0$ or, equivalently, $c \geq n/(2n-1)$. This last condition holds for all $n \geq 1$ and $c \geq 1$.

For the boundary condition, we set $T(1) = 1$, and so $T(1) = 1 \leq c \cdot 1^2$. Thus, we can choose $n_0 = 1$ and $c = 1$.

b. We guess that $T(n) = c \lg n$, where c is the constant in the $\Theta(1)$ term. We have

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= c \lg(n/2) + c \\ &= c \lg n - c + c \\ &= c \lg n \; . \end{aligned}$$

For the boundary condition, choose $T(2) = c$.

c. We guess that $T(n) = n \lg n$. We have

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

For the boundary condition, choose $T(2) = 2$.

d. We will show that $T(n) \leq cn \lg n$ for $c = 20$ and $n \geq 917$. (Different combinations of c and n_0 work. We just happen to choose this combination.) First, observe that $n/2 + 17 \leq 3n/4 < n$ for all $n \geq 68$. We have

$$\begin{aligned} T(n) &= 2T(n/2 + 17) + n \\ &= 2(c(n/2 + 17)\lg(n/2 + 17)) + n \\ &= cn \lg(n/2 + 17) + 34c \lg(n/2 + 17) + n \\ &< cn \lg(3n/4) + 34c \lg n + n \qquad \text{(because } n \geq 68\text{)} \\ &= cn \lg n - cn \lg(4/3) + 34c \lg n + n \\ &= cn \lg n + (34c \lg n - n(c \lg(4/3) - 1)) \\ &\leq cn \lg n \end{aligned}$$

if $34c \lg n \leq n(c \lg(4/3) - 1)$. If we choose $c = 20$, then this inequality holds for all $n \geq 917$. (Notice that for there to be an n_0 such that the inequality holds for all $n \geq n_0$, we must choose c such that $c \lg(4/3) - 1 > 0$, or $c > 1/\lg(4/3) \approx 3.476$.)

e. Let c be the constant in the $\Theta(n)$ term. We need to show only the upper bound of $O(n)$, since the lower bound of $\Omega(n)$ follows immediately from the $\Theta(n)$ term in the recurrence. We guess that $T(n) \leq dn$, where d is a constant that we will choose. We have

$$\begin{aligned} T(n) &= 2T(n/3) + cn \\ &\leq 2dn/3 + cn \end{aligned}$$

$$\begin{aligned} &= n(2d/3 + c) \\ &\le dn \end{aligned}$$

if $2d/3 + c \le d$ or, equivalently, $d \ge 3c$.

f. Let c be the constant in the $\Theta(n)$ term. We guess that $T(n) = dn^2 - d'n$ for constants d and d' that we will choose. We will show the upper (O) and lower (Ω) bounds separately.

For the upper bound, we have

$$\begin{aligned} T(n) &\le 4T(n/2) + cn \\ &= 4(d(n/2)^2 - d'n/2) + cn \\ &= dn^2 - 2d'n + cn \\ &\le dn^2 - d'n \end{aligned}$$

if $-2d'n + cn \le -d'n$ or, equivalently, $d' \ge c$. For the lower bound, we just need $d' \le c$. Thus, setting $d' = c$ works for both the upper and lower bounds.

Solution to Exercise 4.3-2

We want to solve the recurrence $T(n) = 4T(n/2) + n$. Using the substitution method while assuming that $T(n) \le cn^2$ will fail:

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4c\left(\frac{n}{2}\right)^2 + n \\ &= cn^2 + n \,, \end{aligned}$$

which is greater than cn^2. In order to make the substitution proof work, subtract off a lower-order term and assume that $T(n) \le cn^2 - dn$, where we get to choose d. Now,

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4\left(c\left(\frac{n}{2}\right)^2 - \frac{dn}{2}\right) + n \\ &= cn^2 - 2dn + n \,, \end{aligned}$$

which is less than or equal to $cn^2 - dn$ if $d \ge 1$.

Solution to Exercise 4.3-3

For the recurrence $T(n) = 2T(n-1) + 1$, if we use the guess that $T(n) \le c2^n$, the proof will fail:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &\le 2(c2^{n-1}) + 1 \\ &= c2^n + 1 \,, \end{aligned}$$

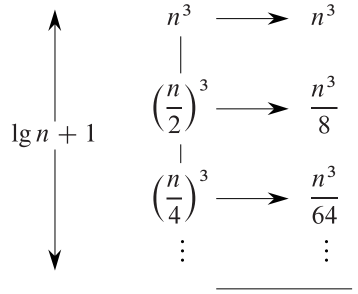
which is greater than $c2^n$. Instead, subtract off a constant d, which we get to choose: $T(n) \le c2^n - d$. Now, we have

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &\le 2(c2^{n-1} - d) + 1 \\ &= c2^n - 2d + 1 \,, \end{aligned}$$

which is less than or equal to $c2^n - d$ if $d \ge 1$.

Solution to Exercise 4.4-1

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



$$\text{Total: } n^3 \sum_{k=0}^{\lg n} \frac{1}{8^k} < n^3 \sum_{k=0}^{\infty} \frac{1}{8^k} = O(n^3)$$

The recursion tree has a single node at each level, contributing $(n/2^k)^3 = n^3/8^k$ at each depth k. The total is

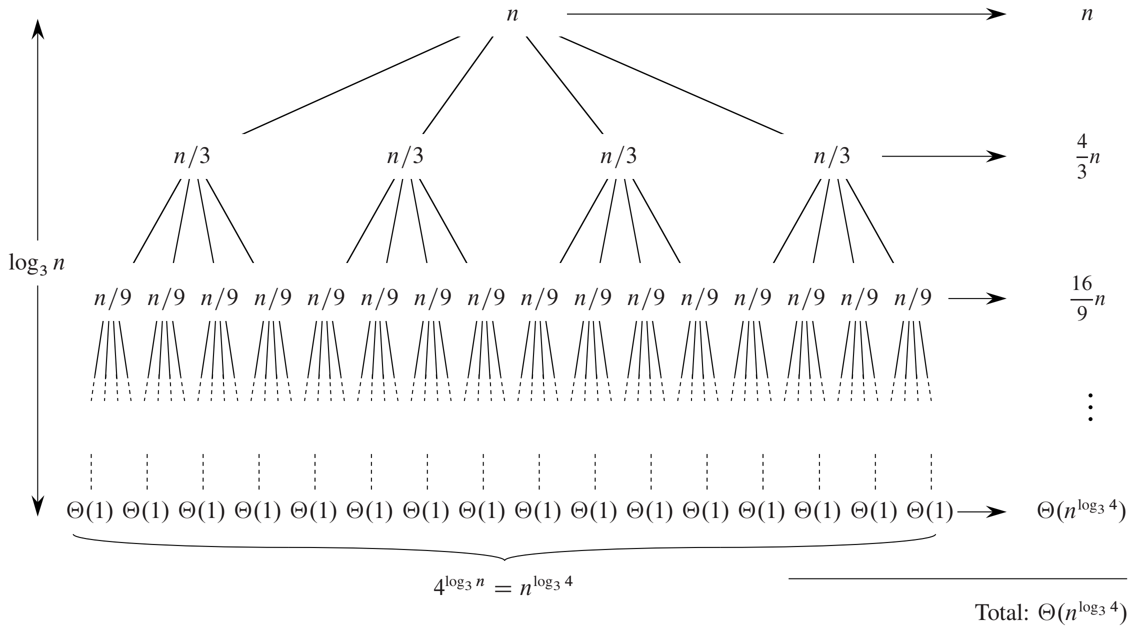
$$\begin{aligned} \sum_{k=0}^{\lg n} \frac{n^3}{8^k} &< n^3 \sum_{k=0}^{\infty} (1/8)^k \\ &= n^3 \cdot \frac{1}{1 - 1/8} \\ &= (8/7)n^3 \\ &= O(n^3) \,. \end{aligned}$$

Now, we prove that $T(n) = O(n^3)$ with the substitution method. We need to show that $T(n) \le cn^3$ for some constant c. We have

$$\begin{aligned} T(n) &= T(n/2) + n^3 \\ &\le c(n/2)^3 + n^3 \\ &= cn^3/8 + n^3 \\ &= n^3(c/8 + 1) \,, \end{aligned}$$

which is less than or equal to cn^3 if $c \ge 8/7$.

b. $T(n) = 4T(n/3) + n$.



n
n
n/3
n/3
n/3
n/3
$\frac{4}{3}n$
$\log_3 n$
n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9 n/9
$\frac{16}{9}n$
Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1)
$\Theta(n^{\log_3 4})$
$4^{\log_3 n} = n^{\log_3 4}$
Total: $\Theta(n^{\log_3 4})$

The number of nodes increases by a factor of 4 as we go down each level in the recursion tree, and the size of each subproblem decreases by a factor of 3. Thus, at each depth k above the leaves, there are 4^k nodes, each with cost $n/3^k$, so that the total cost at depth k is $(4/3)^k n$. The subproblem size reduces to 1 after $\log_3 n$ levels, so that there are $\log_3 n - 1$ levels above the leaves. The number of leaves is $4^{\log_3 n} = n^{\log_3 4}$, each costing $\Theta(1)$. The total cost is then

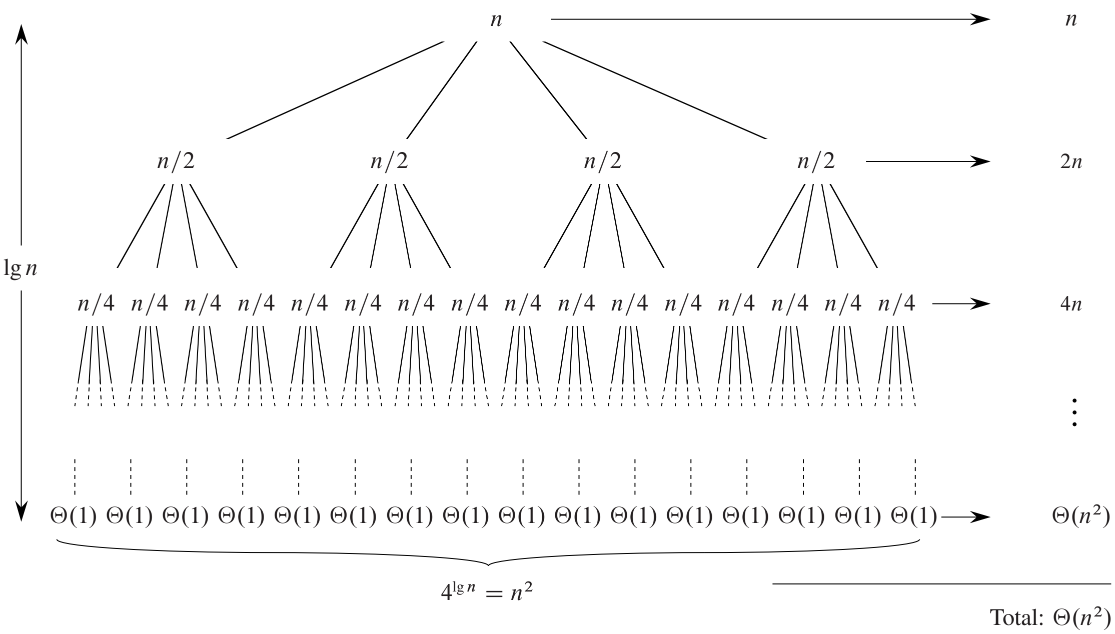
$$\begin{aligned}
\sum_{k=0}^{\log_3 n-1} \left(\frac{4}{3}\right)^k n + \Theta(n^{\log_3 4}) &= n \cdot \frac{(4/3)^{\log_3 n} - 1}{(4/3) - 1} + \Theta(n^{\log_3 4}) \\
&< 3n(4/3)^{\log_3 n} + \Theta(n^{\log_3 4}) \\
&= 3n(4^{\log_3 n})(1/3)^{\log_3 n} + \Theta(n^{\log_3 4}) \\
&= 3n(n^{\log_3 4})(3^{-\log_3 n}) + \Theta(n^{\log_3 4}) \\
&= 3n^{\log_3 4+1} n^{-\log_3 3} + \Theta(n^{\log_3 4}) \\
&= 3n^{\log_3 4+1} n^{-1} + \Theta(n^{\log_3 4}) \\
&= 3n^{\log_3 4} + \Theta(n^{\log_3 4}) \\
&= \Theta(n^{\log_3 4}) .
\end{aligned}$$

Now, we prove that $T(n) = O(n^{\log_3 4})$ with the substitution method. We guess that $T(n) \le n^{\log_3 4} - cn$, where $c > 0$ is a constant that we will choose. We have

$$\begin{aligned}
T(n) &= 4T(n/3) + n \\
&\le 4((n/3)^{\log_3 4} - cn/3) + n \\
&= 4n^{\log_3 4}(1/3)^{\log_3 4} - (4/3)cn + n \\
&= 4n^{\log_3 4}(3^{-\log_3 4}) - (4/3)cn + n \\
&= 4n^{\log_3 4}(4^{-\log_3 3}) - (4/3)cn + n \\
&= 4n^{\log_3 4}(4^{-1}) - (4/3)cn + n \\
&\le n^{\log_3 4} - cn
\end{aligned}$$

if $-(4/3)cn + n \le -cn$ or, equivalently, $c \ge 3$. Hence, $T(n) = O(n^{\log_3 4})$.

c. $T(n) = 4(n/2) + n$.


n
n
n/2 n/2 n/2 n/2
2n
lg n
n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4 n/4
4n
Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1)
Θ(n²)
4^{lg n} = n²
Total: Θ(n²)

The number of nodes increases by a factor of 4 as we go down each level in the recursion tree, and the size of each subproblem decreases by a factor of 2. Thus, at each depth k above the leaves, there are 4^k nodes, each with cost $n/2^k$, so that the total cost at depth k is $(4/2)^k n = 2^k n$. The subproblem size reduces to 1 after $\lg n$ levels, so that there are $\lg n - 1$ levels above the leaves. The number of leaves is $4^{\lg n} = n^2$, each costing $\Theta(1)$. The total cost is then

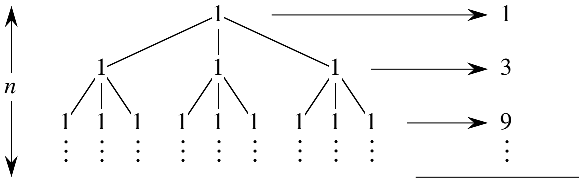
$$\begin{aligned}
\sum_{k=0}^{\lg n - 1} 2^k n + \Theta(n^2) &= n \cdot \frac{2^{\lg n - 1} - 1}{2 - 1} + \Theta(n^2) \\
&< n^2 + \Theta(n^2) \\
&= \Theta(n^2) \,.
\end{aligned}$$

Now, we prove that $T(n) = O(n^2)$ with the substitution method. We guess that $T(n) \le n^2 - cn$, where $c > 0$ is a constant that we will choose. We have

$$\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4((n/2)^2 - cn/2) + n \\
&= n^2 - 2cn + n \\
&\le n^2 - cn
\end{aligned}$$

if $2cn + n \le -cn$ or, equivalently, $c \ge 1$. Hence, $T(n) = O(n^2)$.

d. $T(n) = 3T(n-1) + 1$



Total: $1 + 3 + 9 + \cdots + 3^{n-1} = O(3^n)$

The recursion tree is full, with each depth i contributing 3^i. The total contribution is $\sum_{i=0}^{n-1} 3^i = (3^n - 1)/(3 - 1) = O(3^n)$.

Now we prove that $T(n) = O(3^n)$ by the substitution method. We guess that $T(n) = 3^n - c$, where $c > 0$ is a constant that we will choose. We have

$$\begin{aligned} T(n) &= 3T(n-1) + 1 \\ &\le 3(3^{n-1} - c) + 1 \\ &= 3^n - 3c + 1 \\ &\le 3^n - c \end{aligned}$$

if $-3c + 1 \le -c$ or, equivalently, $c \ge 1/2$. Hence, $T(n) = O(3^n)$.

Solution to Exercise 4.4-2

We guess that $L(n) \ge dn$, where $d > 0$ is a constant that we will choose. We have

$$\begin{aligned} L(n) &= L(n/3) + L(2n/3) \\ &\ge dn/3 + 2dn/3 \\ &= dn\ . \end{aligned}$$

Choosing $d > 0$ so that $L(n) \ge dn$ for all $n < n_0$ finishes the proof.

Solution to Exercise 4.4-3

To show by substitution that $T(n) = \Omega(n \lg n)$, we guess that $T(n) \ge dn \lg n$, where $d > 0$ is a constant that we will choose. Let $c > 0$ be the constant in the $\Theta(n)$ term of the recurrence, so that we have

$$\begin{aligned} T(n) &= T(n/3) + T(2n/3) + cn \\ &\ge (dn/3)\lg(n/3) + (2dn/3)\lg(2n/3) + cn \\ &= (dn/3)\lg n - (dn/3)\lg 3 + (2dn/3)\lg n + (2dn/3)\lg(2/3) + cn \\ &= dn \lg n - (dn/3)\lg 3 + (2dn/3)\lg(2/3) + cn \\ &\ge dn \lg n \end{aligned}$$

if $-(dn/3)\lg 3 + (2dn/3)\lg(2/3) + cn \ge 0$. This requirement is equivalent to $d \le c/((1/3)\lg 3 - (2/3)\lg(2/3))$. Since the denominator on the right-hand side is positive, we can choose such a $d > 0$. Therefore, $T(n) = \Omega(n \lg n)$. Since the text showed that $T(n) = O(n \lg n)$, we have that $T(n) = \Theta(n \lg n)$.

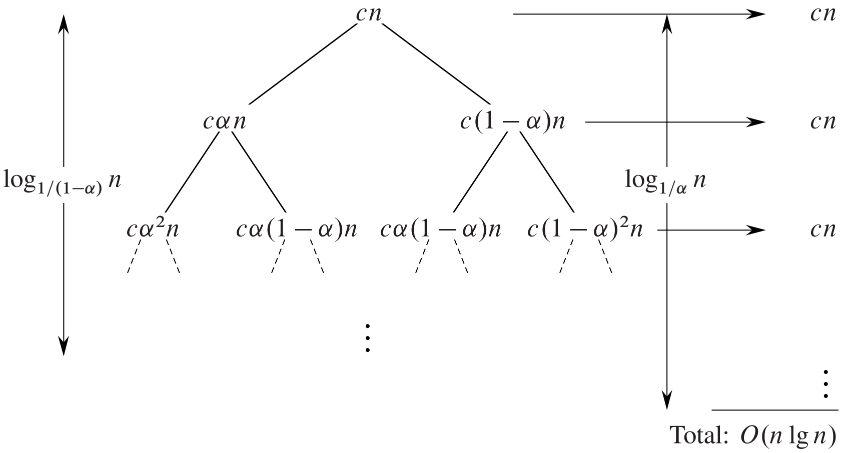
Solution to Exercise 4.4-4

This solution is also posted publicly

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

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 $T(n) \leq dn \lg n$ for a suitable constant $d > 0$:

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

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

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

Since $1/2 \leq \alpha < 1$ and $0 < 1-\alpha \leq 1/2$, 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:

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

or

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

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 $T(n) \geq dn \lg n$ for a suitable constant $d > 0$. We can use the same proof as for the upper bound, substituting $\geq$ for $\leq$, and we get the requirement that

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

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

Solution to Exercise 4.5-1

In all parts of this problem, we have $a = 2$ and $b = 4$, and thus $n^{\log_b a} = n^{\log_4 2} = n^{1/2} = \sqrt{n}$.

a. $T(n) = \Theta(\sqrt{n})$. Here, $f(n) = O(n^{1/2-\epsilon})$ for $\epsilon = 1/2$. Case 1 applies, and $T(n) = \Theta(n^{1/2}) = \Theta(\sqrt{n})$.

b. $T(n) = \Theta(\sqrt{n} \lg n)$. Now $f(n) = \sqrt{n} = \Theta(n^{\log_b a})$. Case 2 applies, with $k = 0$.

c. $T(n) = \Theta(\sqrt{n} \lg^3 n)$. Now $f(n) = \sqrt{n} \lg^2 n = \Theta(n^{\log_b a} \lg^2 n)$. Case 2 applies, with $k = 2$.

d. $T(n) = \Theta(n)$. This time, $f(n) = n^1$, and so $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon = 1/2$. In order for case 3 to apply, we have to check the regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$. Here, $af(n/b) = n/2$, and so the regularity condition holds for $c = 1/2$. Therefore, case 3 applies.

e. $T(n) = \Theta(n^2)$. Now, $f(n) = n^2$, and so $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon = 3/2$. In order for case 3 to apply, we again have to check the regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$. Here, $af(n/b) = n^2/8$, and so the regularity condition holds for $c = 1/8$. Therefore, case 3 applies.

Solution to Exercise 4.5-2

We need to find the largest integer a such that $\log_4 a < \lg 7$. The answer is $a = 48$.

Solution to Exercise 4.5-3

Here, we have $n^{\log_b a} = n^{\lg 1} = n^0$. Since the driving function is $\Theta(n^0)$, case 2 of the master theorem applies with $k = 0$. The solution is $T(n) = \Theta(\lg n)$.

Solution to Exercise 4.5-4

In order for $af(n/b) \leq cf(n)$ to hold with $a = 1$, $b = 2$, and $f(n) = \lg n$, we would need to have $(\lg(n/2))/\lg n < c$. Since $\lg(n/2) = \lg n - 1$, we would need $(\lg n - 1)/\lg n < c$, and for any constant $c < 1$, there exist an infinite number of values for n for which this inequality does not hold.

Furthermore, since $n^{\log_b a} = n^0$, there is no constant $\epsilon > 0$ such that $\lg n = \Omega(n^\epsilon)$.

Solution to Exercise 4.5-5

Choose $a = 1$, $b = \sqrt{2}$, and $\epsilon = 1$, in which case we have $n^{\log_b a + \epsilon} = n^0 \cdot n^1 = n$. Since $f(n) = 2^{\lceil \lg n \rceil} \geq 2^{\lg n} = n = \Omega(n)$, the condition that $f(n) = \Omega(n^{\log_b a + \epsilon})$ is satisfied. For all $n = 2^k$, where $k > 0$ is integer, we have $f(n) = 2^{\lceil \lg n \rceil} = 2^k = n$ and $af(n/b) = f(n/\sqrt{2}) = 2^{\left\lceil \lg n - \lg(\sqrt{2}) \right\rceil} = 2^{\lceil \lg n - 1/2 \rceil} = 2^{\lceil k - 1/2 \rceil} = 2^k = n = f(n)$, and thus for $n = 2^k$, we have $af(n/b) = f(n)$. Consequently, no $c < 1$ can exist for which $af(n/b) \leq cf(n)$ for all sufficiently large n.

Solution to Problem 4-1

Note: In parts (a), (b), and (e) below, we are applying case 3 of the master theorem, which requires the regularity condition that $af(n/b) \leq cf(n)$ 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. $T(n) = T(8n/11) + n = \Theta(n)$. This is a divide-and-conquer recurrence with $a = 1$, $b = 11/8$, $f(n) = n$, and $n^{\log_b a} = n^{\log_{11/8} 1} = n^0 = 1$. Since $n = \Omega(n^{\log_{11/8} 1 + 1})$ and $a/b^k = 1/(11/8)^1 = 8/11 < 1$, 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 a 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 with $k = 0$, and $T(n) = \Theta(n^2 \lg n)$.

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

e. $T(n) = 8T(n/3) + n^2 = \Theta(n^2)$. This is a divide-and-conquer recurrence with $a = 8$, $b = 3$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_3 8}$. Since $1 < \log_3 8 < 2$, we have that $n^2 = \Omega(n^{\log_3 8+\epsilon})$ for some constant $\epsilon > 0$. We also have $a/b^k = 8/3^2 = 8/9 < 1$, so that case 3 of the master theorem applies, and $T(n) = \Theta(n^2)$.

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

g. $T(n) = 2T(n/4) + \sqrt{n} = \Theta(\sqrt{n} \lg n)$. 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 with $k = 0$, and $T(n) = \Theta(\sqrt{n} \lg n)$.

h. $T(n) = T(n-2) + n^2$. To guess a bound, assume that n is even. Then if the recurrence were iterated out, it would contain $n/2$ terms before getting down to $n = 0$. Each of these terms is at most n^2, so that the sum is at most $(n/2)n^2 = n^3/2$, giving an upper bound of $O(n^3)$. To get a lower bound, observe that of the $n/2$ terms in the summation, half of them (that is, $n/4$ of them) are at least $n/2$, giving a sum that is at least $(n/4)(n/2)^2 = n^3/16$, or $\Omega(n^3)$. Therefore, our guess is that $T(n) = \Theta(n^3)$.

First, we prove the $T(n) = \Omega(n^3)$ part by induction. The inductive hypothesis is $T(n) \geq cn^3$ for some constant $c > 0$.

$$\begin{aligned} T(n) &= T(n-2) + n^2 \\ &\geq c(n-2)^3 + n^2 \\ &= cn^3 - 6cn^2 + 12cn - 8c + n^2 \\ &\geq cn^3 \end{aligned}$$

if $-6cn^2 + 12cn - 8c + n^2 \geq 0$. This condition holds when $n \geq 1$ and $0 < c \leq 1/6$.

For the upper bound, $T(n) = O(n^3)$, we use the inductive hypothesis that $T(n) \leq cn^3$ for some constant $c > 0$. By a similar derivation, we get that $T(n) \leq cn^3$ if $-6cn^2 + 12cn - 8c + n^2 \leq 0$. This condition holds for $c \geq 1/2$ and $n \geq 1$.

Thus, $T(n) = \Omega(n^3)$ and $T(n) = O(n^3)$, so we conclude that $T(n) = \Theta(n^3)$.

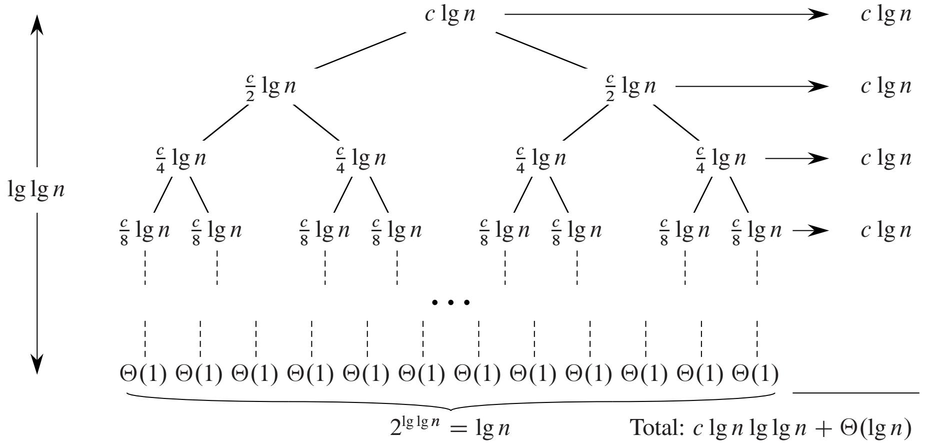
Solution to Problem 4-3

a. Since $m = \lg n$, we also have $n = 2^m$. The recurrence becomes $T(2^m) = 2T(2^{m/2}) + \Theta(m)$. As a recurrence for $S(m)$, it becomes $S(m) = 2S(m/2) + \Theta(m)$.

b. The recurrence for $S(m)$ falls into case 2 of the master theorem, with $a = b = 2$ and $k = 1$, so that its solution is $S(m) = \Theta(m \lg m)$.

c. Substituting back into the recurrence for $T(n)$, we get that $T(n) = T(2^m) = S(m) = \Theta(m \lg m) = \Theta(\lg n \lg\lg n)$.

d. Here is the recursion tree:



The constant c stands for the constant in the $\Theta(\lg n)$ term. The root contributes a cost of $c \lg n$. Each child of the root has a subproblem size of $\sqrt{n} = n^{1/2}$, contributing a cost of $c \lg n^{1/2} = (c \lg n)/2$, for a combined cost of $c \lg n$ for both children. At the next level down, the subproblem sizes are $n^{1/4}$, each contributing a cost of $(c \lg n)/4$; with 4 such subproblems, the combined cost at this level is also $c \lg n$. At each level above the leaves, the cost per level is $c \lg n$.

How many levels until we get down to a subproblem size of 1? We never do, because if $n > 1$, repeatedly taking square roots will never get down to 1. We can use a subproblem size of 2 as a base case, however. How many levels until getting down to a subproblem of size 2? Think of how many bits we need to represent the subproblem size. At each level, the number of bits is halved, until we get down to 1 bit to represent a subproblem of size 2. That is, the number of levels is (lg the number of bits for the original problem). Since the original problem requires $\lg n$ bits to represent n, the number of levels is $\lg \lg n$. The total cost of all levels above the leaves is then $c \lg n \lg \lg n$.

How many leaves are there? The number of leaves doubles in each level, and there are $\lg \lg n$ levelsl, so that the total number of leaves becomes $2^{\lg \lg n} = \lg n$. Since each leaf costs $\Theta(1)$, the total cost of the leaves is $\Theta(\lg n)$.

Adding up the costs at all levels, we get a total cost of $c \lg n \lg \lg n + \Theta(\lg n) = \Theta(\lg n \lg \lg n)$.

e. $T(n) = 2T(\sqrt{n}) + \Theta(1)$.

Again, we let $m = \lg n$ so that $n = 2^m$. The recurrence is then $T(2^m) = T(2^{m/2}) + \Theta(1)$. As a recurrence for $S(m)$, it becomes $S(m) = 2S(m/2) + \Theta(1)$. This recurrence falls into case 2 of the master theorem with $a = 2$ and $b = 2$ so that its solution is $S(m) = \Theta(m)$. Substituting back into the recurrence for $T(n)$, we get $T(n) = \Theta(\lg n)$.

f. $T(n) = 3T(\sqrt[3]{n}) + \Theta(n)$.

This time, we let $m = \log_3 n$ so that $n = 3^m$. The recurrence is $T(3^m) = 3T(3^{m/3}) + \Theta(3^m)$. As a recurrence for $S(m)$, it becomes $S(m) = 3S(m/3) +$

$\Theta(3^m)$. If we were to draw out recursion tree for this recurrence, it would have $\log_3 m$ levels with the costs summing as $3^m + 3 \cdot 3^{m/3} + 3^2 \cdot 3^{m/9} + \cdots$ (ignoring for the moment the constant in the $\Theta(3^m)$ term). The 3^m term dominates this summation, so that $S(m) = \Theta(3^m)$. Substituting back into the recurrence for $T(n)$, we get $T(n) = \Theta(n)$.

Solution to Problem 4-4

a. $T(n) = 5T(n/3) + n \lg n$. We have $f(n) = n \lg n$ and $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465}$. Since $n \lg n = O(n^{\log_3 4 - \epsilon})$ for any $0 < \epsilon \le 0.46$, by case 1 of the master theorem, we have $T(n) = \Theta(n^{\log_3 5})$.

b. $T(n) = 3T(n/3) + n/\lg n$. If we were to draw the recursion tree, depth i of the tree would have 3^i nodes. Each node at depth i incurs a cost of $n/(3^i \lg(n/3^i))$, for a total cost at depth i of $n/\lg(n/3^i)$. Using equation (3.19), we have

$$\lg(n/3^i) = \frac{\log_3(n/3^i)}{\log_3 2} = \frac{\log_3 n - i}{\log_3 2} .$$

The number of leaves is $3^{\log_3 n} = n$, each contributing $\Theta(1)$, for a total contribution from the leaves of $\Theta(n)$. Thus, the total cost of the recursion tree is

$$\begin{aligned}\Theta(n) + \sum_{i=0}^{\log_3 n - 1} \frac{n}{\lg 3^i} &= \Theta(n) + n \log_3 2 \sum_{i=0}^{\log_3 n - 1} \frac{1}{\log_3 n - i} \\ &= \Theta(n) + n \log_3 2 \sum_{i=0}^{\log_3 n - 1} \frac{1}{i} \\ &= \Theta(n) + n \log_3 2 \cdot H_{\log_3 n - 1} \\ &= \Theta(n) + n \log_3 2 \cdot \Theta(\ln \log_3 n - 1) \\ &= \Theta(n \lg \lg n) .\end{aligned}$$

c. $T(n) = 8T(n/2) + n^3\sqrt{n}$. We have $f(n) = n^3\sqrt{n} = n^{7/2}$ and $n^{\log_b a} = n^{\log_2 8} = n^3$. Since $n^{7/2} = \Omega(n^{3+\epsilon})$ for $\epsilon = 1/2$, we look at the regularity condition in case 3 of the master theorem. We have $af(n/b) = 8(n/2)^3\sqrt{n/2} = n^{7/2}/\sqrt{2} \le cn^{7/2}$ for $1/\sqrt{2} \le c < 1$. Case 3 applies, and we have $T(n) = \Theta(n^3\sqrt{n})$.

d. $T(n) = 2T(n/2 - 2) + n/2$. Subtracting 2 in the argument shouldn't make much difference, and so this recurrence looks like $T(n) = 2T(n/2) + \Theta(n)$, which falls into case 2 of the master theorem with $k = 0$. Therefore, we guess that $T(n) = \Theta(n \lg n)$. We'll prove the upper and lower bounds separately.

The upper bound is easy. We assume that $T(n)$ monotonically increases, and so $T(n) = 2T(n/2 - 2) + n/2 \le 2T(n/2) + n/2$. We can use case 2 of the master theorem with $k = 0$ for the upper bound, getting $T(n) = O(n \lg n)$.

For the lower bound, we use a substitution proof, which relies on the inequality $n/2 - 2 \ge n/4$ for $n \ge 8$. We assume that $T(n) \ge cn \lg n$ for some positive constant c that we will choose. We have

$$\begin{aligned}
T(n) &= 2T(n/2-2)+n/2 \\
&\ge 2c(n/2-2)\lg(n/2-2)+n/2 \\
&= cn\lg(n/2-2)-4c\lg(n/2-2)+n/2 \\
&\ge cn\lg(n/4)-4c\lg(n/2)+n/2 \qquad \text{for } n\ge 8 \\
&= cn\lg n-2cn-4c\lg n+4c+n/2 \\
&\ge cn\lg n
\end{aligned}$$

if $-2cn-6c\lg n+4c+n/2\ge 0$, which is equivalent to $c\le n/(4n+8\lg n-4)$. Choosing $c\le 1/10$ satisfies this inequality.

e. $T(n)=2T(n/2)+n/\lg n$. This part is similar to part (b) and, in fact, a little simpler. If we were to draw the recursion tree, depth i of the tree would have 2^i nodes. Each node at depth i incurs a cost of $n/(2^i\lg(n/2^i))$, for a total cost at depth i of $n/\lg(n/2^i)$. The number of leaves is $2^{\lg n}=n$, each contributing $\Theta(1)$, for a total contribution from the leaves of $\Theta(n)$. Thus, the total cost of the recursion tree is

$$\begin{aligned}
\Theta(n)+\sum_{i=0}^{\lg n-1}\frac{n}{\lg(n/2^i)} &= \Theta(n)+n\sum_{i=0}^{\lg n-1}\frac{1}{\lg n-i} \\
&= \Theta(n)+n\sum_{i=0}^{\lg n-1}\frac{1}{i} \\
&= \Theta(n)+n\cdot H_{\lg n-1} \\
&= \Theta(n)+n\cdot\Theta(\ln\lg n-1) \\
&= \Theta(n\lg\lg n)\ .
\end{aligned}$$

We did a careful accounting in our recursion tree, but 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

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

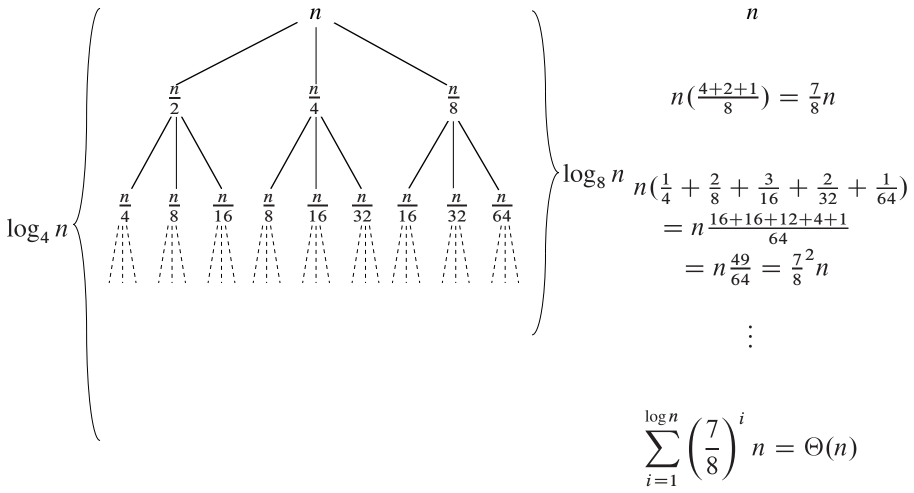
where the last line follows from the identity $H_k=H_{k-1}+1/k$.

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 $T(n) = \Theta(n)$.


n
$\frac{n}{2}$
$\frac{n}{4}$
$\frac{n}{8}$
$\frac{n}{4}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{64}$
$\log_4 n$
$\log_8 n$
n
$n(\frac{4+2+1}{8}) = \frac{7}{8}n$
$n(\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \frac{2}{32} + \frac{1}{64})$
$= n\frac{16+16+12+4+1}{64}$
$= n\frac{49}{64} = \frac{7}{8}^2 n$
$\vdots$
$\sum_{i=1}^{\log n} \left(\frac{7}{8}\right)^i n = \Theta(n)$

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 $T(n) = \Theta(n)$.

In fact, $T(n) = 8n$ is an exact solution, as we can see by substitution:

$$\begin{aligned}
T(n) &= T(n/2) + T(n/4) + T(n/8) + n \\
&= 4n + 2n + n + n \\
&= 8n .
\end{aligned}$$

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.9), we have that $T(n) = \Theta(\lg n)$.

h. $T(n) = T(n-1) + \lg n$. We guess that $T(n) = \Theta(n \lg n)$. Observe that with base case of $n = 1$, we have $T(n) = \sum_{i=1}^{n} \lg i$. We'll bound this summation from above and below to obtain bounds of $O(n \lg n)$ and $\Omega(n \lg n)$.

For the upper bound, we have

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} \lg i \\ &\le \sum_{i=1}^{n} \lg n \\ &= n \lg n . \end{aligned}$$

To obtain a lower bound, we use just the upper half of the summation:

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

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

i. $T(n) = T(n-2) + 1/\lg n$. The solution is $T(n) = \Theta(n/\lg n)$. To see why, expand out the sum:

$$\begin{aligned} T(n) &= T(n-2) + \frac{1}{\lg n} \\ &= T(n-4) + \frac{1}{\lg(n-2)} + \frac{1}{\lg n} \\ &= T(n-6) + \frac{1}{\lg(n-4)} + \frac{1}{\lg(n-2)} + \frac{1}{\lg n} \\ &\ \ \vdots \\ &= T(2) + \frac{1}{\lg 4} + \cdots + \frac{1}{\lg(n-4)} + \frac{1}{\lg(n-2)} + \frac{1}{\lg n} \end{aligned}$$

$$= \frac{1}{\lg 2} + \frac{1}{\lg 4} + \cdots + \frac{1}{\lg(n-4)} + \frac{1}{\lg(n-2)} + \frac{1}{\lg n} ,$$

where the summation on the last line has $n/2$ terms.

The lower bound of $\Omega(n/\lg n)$ is easily seen. The smallest term is $1/\lg n$, and there are $n/2$ terms. Thus, the sum is at least $n/(2\lg n) = \Omega(n/\lg n)$.

For the upper bound, break the summation into the first $\sqrt{n}/2$ terms and the last $(n - \sqrt{n})/2$ terms. Of the first $\sqrt{n}/2$ terms, the largest is the first one, $1/\lg 2 = 1$, and so the first $\sqrt{n}/2$ terms sum to at most $\sqrt{n}/2$. Each of the last $(n - \sqrt{n})/2$ terms is at most $1/\lg\sqrt{n} = 2/\lg n$, and so the last $(n - \sqrt{n})/2$ terms sum to at most

$$\frac{n - \sqrt{n}}{2} \cdot \frac{2}{\lg n} = \frac{n - \sqrt{n}}{\lg n} .$$

Therefore, the sum of all $n/2$ terms is at most

$$\frac{\sqrt{n}}{2} + \frac{n - \sqrt{n}}{\lg n} .$$

The $(n - \sqrt{n})/\lg n$ term dominates for $n \geq 4$, and the sum is $O(n/\lg n)$.

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$. If we draw the recursion tree, we see that at depth 0, there is one node with a cost of n; at depth 1, there are $n^{1/2}$ nodes, each with a cost of $n^{1/2}$, for total cost of n at depth 1; at depth 2, there are $n^{1/2} \cdot n^{1/4} = n^{3/4}$ nodes, each with a cost of $n^{1/4}$, for total cost of n at depth 2; at depth 3, there are $n^{3/4} \cdot n^{1/8} = n^{7/8}$ nodes, each with a cost of $n^{1/8}$, for total cost of n at depth 3; and so on. In general, at depth i, there are $n^{1-1/2^i}$ nodes, each with a cost of $n^{1/2^i}$, for a total cost of n at each level. Because the subproblem sizes decrease by a square root for each increase in the depth, the recursion tree has $\lg\lg n$ levels. Therefore, we guess that $T(n) = \Theta(n \lg\lg n)$. In fact, this recurrence has the exact solution $T(n) = n \lg\lg n$, which we show by substitution:

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

Solution to Problem 4-5

a. The identity can be shown by expanding $\mathcal{F}(z)$ and using the definition of the Fibonacci series.

$$z + z\mathcal{F}(z) + z^2\mathcal{F}(z) = z + z\left(\sum_{i=0}^{\infty} F_i z^i\right) + z^2\left(\sum_{i=0}^{\infty} F_i z^i\right)$$

$$
\begin{aligned}
&= z + \sum_{i=0}^{\infty} F_i z^{i+1} + \sum_{i=0}^{\infty} F_i z^{i+2} \\
&= z + \sum_{i=1}^{\infty} F_{i-1} z^i + \sum_{i=2}^{\infty} F_{i-2} z^i \\
&= z + F_0 z + \sum_{i=2}^{\infty} (F_{i-1} + F_{i-2}) z^i \\
&= F_1 z + F_0 z + \sum_{i=2}^{\infty} F_i z^i \\
&= \mathcal{F}(z) .
\end{aligned}
$$

b. The equation in part (a) gives $z = \mathcal{F}(z) - z\mathcal{F}(z) - z^2\mathcal{F}(z) = \mathcal{F}(z)(1-z-z^2)$, so that $\mathcal{F}(z) = z/(1-z-z^2)$. For the next step, start by observing that $\phi + \widehat{\phi} = 1$ and $\phi\widehat{\phi} = -1$. Then, we have

$$
\begin{aligned}
(1-\phi z)(1-\widehat{\phi} z) &= 1 - (\phi + \widehat{\phi})z + \phi\widehat{\phi} z^2 \\
&= 1 - z - z^2 .
\end{aligned}
$$

Finally, by observing that $\phi - \widehat{\phi} = \sqrt{5}$ and making a common denominator, we have

$$
\begin{aligned}
\frac{1}{\sqrt{5}} \left(\frac{1}{1-\phi z} - \frac{1}{1-\widehat{\phi} z} \right) &= \frac{1}{\sqrt{5}} \cdot \frac{(1-\widehat{\phi} z) - (1-\phi z)}{(1-\phi z)(1-\widehat{\phi} z)} \\
&= \frac{1}{\sqrt{5}} \cdot \frac{\sqrt{5} z}{(1-\phi z)(1-\widehat{\phi} z)} \\
&= \frac{z}{(1-\phi z)(1-\widehat{\phi} z)} .
\end{aligned}
$$

c. Using the hint, apply equation (A.7) to produce

$$
\begin{aligned}
\mathcal{F}(z) &= \frac{1}{\sqrt{5}} \left(\frac{1}{1-\phi z} - \frac{1}{1-\widehat{\phi} z} \right) \\
&= \frac{1}{\sqrt{5}} \left(\sum_{i=0}^{\infty} (\phi z)^i - \sum_{i=0}^{\infty} (\widehat{\phi} z)^i \right) \\
&= \frac{1}{\sqrt{5}} \sum_{i=0}^{\infty} (\phi^i - \widehat{\phi}^i) z^i .
\end{aligned}
$$

d. The definition of $\mathcal{F}(z)$ in the problem and part (c) give

$$
\sum_{i=0}^{\infty} F_i z^i = \frac{1}{\sqrt{5}} \sum_{i=0}^{\infty} (\phi^i - \widehat{\phi}^i) z^i .
$$

Since these summations are formal power series, we have

$$
F_i = \frac{1}{\sqrt{5}} (\phi^i - \widehat{\phi}^i)
$$

for all $i \geq 0$. Equivalently, we have

$$
F_i - \frac{1}{\sqrt{5}} \phi^i = -\frac{1}{\sqrt{5}} \widehat{\phi}^i .
$$

Because $\widehat{\phi} = -0.61803\ldots < 0$, the factor $\widehat{\phi}^i$ is positive for odd $i > 0$ and negative for even $i > 0$, but $\left|\widehat{\phi}\right|^i < 1$ for all $i > 0$. Therefore,

$$\left|F_i - \frac{\phi^i}{\sqrt{5}}\right| = \left|-\frac{\widehat{\phi}^i}{\sqrt{5}}\right| < \frac{1}{\sqrt{5}} < \frac{1}{2}$$

for all $i > 0$. Thus, rounding $\phi^i/\sqrt{5}$ to the nearest integer gives F_i.

e. We start by showing that $F_{i+2} \geq \phi^i$ for $i = 0, 1, 2$: $F_2 = 1 = \phi^0$, $F_3 = 2 > \phi = 1.61803\ldots$, and $F_4 = 3 > (3+\sqrt{5})/2 = \phi^2$ (because $\phi^2 = (3+\sqrt{5})/2$ and $2 < \sqrt{5} < 3$).

Now we'll show that $F_{i+2} > \phi^i$ for $i \geq 3$. By part (d), because $F_i = \phi^i/\sqrt{5}$, rounded to the nearest integer, we have

$$\begin{aligned} F_{i+2} &\geq \frac{\phi^{i+2}}{\sqrt{5}} - \frac{1}{2} \\ &= \phi^i\left(\frac{\phi^2}{\sqrt{5}} - \frac{1}{2\phi^i}\right) . \end{aligned}$$

Observe that

$$\frac{\phi^2}{\sqrt{5}} - \frac{1}{2\phi^i} \geq \frac{\phi^2}{\sqrt{5}} - \frac{1}{2\phi^3}$$

for $i \geq 3$. If we can prove that

$$\frac{\phi^2}{\sqrt{5}} - \frac{1}{2\phi^3} > 1 ,$$

then for $i \geq 3$, we have

$$\begin{aligned} F_{i+2} &\geq \phi^i\left(\frac{\phi^2}{\sqrt{5}} - \frac{1}{2\phi^i}\right) \\ &\geq \phi^i\left(\frac{\phi^2}{\sqrt{5}} - \frac{1}{2\phi^3}\right) \\ &> \phi^i , \end{aligned}$$

as desired. Noting that $\phi^3 = 2 + \sqrt{5}$, we have

$$\begin{aligned} \frac{\phi^2}{\sqrt{5}} - \frac{1}{2\phi^3} &= \frac{3+\sqrt{5}}{2\sqrt{5}} - \frac{1}{2(2+\sqrt{5})} \\ &= \frac{1}{2}\left(\frac{3+\sqrt{5}}{\sqrt{5}} - \frac{1}{2+\sqrt{5}}\right) \\ &= \frac{1}{2}\left(1 + \frac{3}{\sqrt{5}} - \frac{1}{2+\sqrt{5}}\right) \\ &= \frac{1}{2}\left(1 + \frac{6+3\sqrt{5}}{5+2\sqrt{5}} - \frac{\sqrt{5}}{5+2\sqrt{5}}\right) \\ &= \frac{1}{2}\left(1 + \frac{6+2\sqrt{5}}{5+2\sqrt{5}}\right) \\ &> \frac{1}{2}(1+1) \\ &= 1 , \end{aligned}$$

which completes the proof.

Solution to Problem 4-6

a. We will prove by contradiction that if at least $n/2$ chips are bad, there is no algorithm A that can determine which chips are good using a strategy based on pairwise tests.

Assume for sake of contradiction that there exists an algorithm A that, if at least $n/2$ chips are bad, can output a good chip, using pairwise tests.

Since we can assign arbitrary behavior to bad chips, a bad chip can always mislead about the other chip, so that pairwise tests could always go as follows:

- If both chips are good, then both are reported as good.
- If both chips are bad, then both are reported as good.
- If one chip is good and one chip is bad, then both are reported as bad.

Partition the n chips into sets X and Y, where $|X| = \lfloor n/2 \rfloor$ and $|Y| = \lceil n/2 \rceil$. Assume that the results above happen when pairwise tests are carried out.

First, consider the case where all the chips in X are good and all the chips in Y are bad. In this case, A will use the results of the pairwise tests to output some chip x from X. Now, consider the case where all the chips in X are bad and all the chips in Y are good. In this case, A will again use the results of the pairwise tests to output some chip y from Y.

Since these two cases will have the exact same results from pairwise comparisons, the chips x and y would be the same chip. However, a single chip cannot be from both X and Y, so that A is not a correct algorithm. Therefore, the assumption that a correct algorithm that can output a good chip exists is wrong, and no algorithm exists.

b. Execute the following procedure. If n is an odd number, set one arbitrary chip c aside to give an even number of remaining chips. If n is even, do not set a chip aside. Call the chips that remain (either n or $n-1$ of them) set C. Start with a set R of chips, initially empty. Repeatedly take two chips at a time from C and conduct a pairwise test. If a test reports that both chips are bad or that one is good and one is bad, do not put either chip into set R. If a test reports that both chips are good, add one of the two chips to set R. At the end of all of the pairwise tests, if $|R|$ is even and a chip c had been set aside at the beginning, add c to R. We will show that the set R has at most $\lceil n/2 \rceil$ chips and that more than half of the chips in R are good.

We first prove that $|R| \le \lceil n/2 \rceil$. Since $|C|$ is equal to n if n is even and to $n-1$ if n is odd, $|C| \le n$. There are $|C|/2$ pairwise tests, and each test contributes either 0 or 1 chip to R, so that $|R| \le |C|/2$. If n is even, then $|R| \le |C|/2 = n/2 = \lceil n/2 \rceil$. If n is odd, $|C|/2$ with the addition of a single chip that might be added back in equals $(n-1)/2 + 1 = \lceil n/2 \rceil$, so that $|R| \le |C|/2 + 1 = (n-1)/2 + 1 = \lceil n/2 \rceil$.

Now, we show that R will always have more good chips than bad chips. Let GG be the set of chips in pairs where both chips are good, BB be the set of chips in pairs where both chips are bad, and $BG = C - (GG \cup BB)$ be the set of chips in pairs with one bad and one good. The number of good chips in C is $|GG| + |BG|/2$, and the number of bad chips in C is $|BB| + |BG|/2$. We know that at the start of the procedure, there are more good chips than bad chips. If n is even, so that no chip was set aside, then $|GG| + |BG|/2 > |BB| + |BG|/2$, so that $|GG| > |BB|$. If n is odd, the chip c set aside at the beginning could have been either good or bad. If it was good, then we have $|GG| + |BG|/2 + 1 > |BB| + |BG|/2$, so that $|GG| + |BG|/2 \geq |BB| + |BG|/2$ and thus $|GG| \geq |BB|$. If chip c was bad, then we have $|GG| + |BG|/2 > |BB| + |BG|/2 + 1$, so that $|GG| + |BG|/2 > |BB| + |BG|/2$ and thus $|GG| > |BB|$.

We will proceed through several cases to show that we will always maintain R having strictly more good than bad chips. To show this, let $GG' = GG \cap R$, the chips in GG that go into R, and $BB' = BB \cap R$, the chips in BB that go into R. No chips from BG go into R, since the good chip in a pair always reports that the other chip is bad. Thus, R contains only chips from GG and BB, so that $R = GG' \cup BB'$ before adding in the chip set aside.

First, consider the case where n is even and no chip is set aside, so that $|GG| > |BB|$. Because there are strictly more good than bad chips and n is even, there are at least two more good chips than bad chips. Whenever the test reports that both are bad or one is good and one is bad, at least one chip in the pair is bad. Therefore, if a test reports at least one chip is bad, it does not add a bad chip to R. If the test reports that both are good, either both chips are good (from GG) or both chips are bad (from BB). In this case, one chip from the pair is added to R, so that $|GG'| = |GG|/2$ and $|BB'| = |BB|/2$. Since $R = GG' \cup BB'$ and $|GG'| = |GG|/2 > |BB|/2 = |BB'|$, we know that R ends up with strictly more good chips than bad chips in this case.

Now, let us consider the case where n is odd and a chip c was put aside. Chip c could be good or bad.

If c is a good chip, then we know that $|GG| \geq |BB|$ from above. All of the pairs of chips that are reported as both bad or one good and one bad will not be added to R, and R will contain only half of the chips from the pairs of chips reported as both good. As before, this will result in $|GG'| = |GG|/2$ and $|BB'| = |BB|/2$, so that $|GG'| \geq |BB'|$. Since $R = GG' \cup BB'$, at minimum half of the chips in R will be good. If R ends up with an equal number of good and bad chips, then adding c into R gives R more good chips than bad chips. Of course, if R contains more good chips than bad chips, then since c is a good chip, R still contains more good chips than bad chips after adding in c.

Finally, consider the case in which c is a bad chip. In this case, $n - 1$ is even, and we have that $|GG| > |BB|$ from above. As in the case where n is even, R ends up with more good chips than bad chips after all the pairwise tests. Now, if $|R|$ is even, then since R contains more good chips than bad chips, it must contain at least two more good chips than bad chips. Therefore, after R has c added in at the end, R still contains more good chips than bad chips. If $|R|$ is odd, it might have just one more good chip than bad chip, but since c is not added into R in this case, R still has more good chips than bad chips.

Therefore, in all cases, we are able to use $\lfloor n/2 \rfloor$ pairwise tests to obtain a set R with at most $\lceil n/2 \rceil$ chips, maintaining the property that more than half of the chips are good.

c. In order to identify one good chip, start with all n chips, and execute the procedure from part (b) to get a set R with at most $\lceil n/2 \rceil$ chips, more than half of which will be good. Then, recursively repeat the procedure on set R, and continue to do so until a set of one chip remains. We know that this procedure will terminate with one chip, since every recursive call will end with a set of size at most $\lceil m/2 \rceil$, where m is the size of the set at the beginning of the recursive call. Since this set will retain the property that more chips are good than bad, as proven in part (b), the one remaining chip must be good.

The recurrence that describes the number of tests needed is $T(n) = T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor$, since $\lfloor n/2 \rfloor$ pairwise tests occur at each execution of the procedure, leaving at most $\lceil n/2 \rceil$ for the next recursive call. Since floors and ceilings usually do not matter when solving recurrences asymptotically, we can write the recurrence as $T(n) = T(n/2) + \Theta(n)$. Then, by the master method, we have $a = 1$, $b = 2$, and $f(n) = \Theta(n)$. Plugging into $n^{\log_b a}$, we get $n^{\log_2 1} = 1$. Since $f(n) = \Omega(n^{0+\epsilon})$ for $\epsilon = 1$, and $f(n/2) \leq cf(n)$ for all $c \geq 1/2$, case 3 of the master theorem applies, and $T(n) = \Theta(n)$.

d. From part (c), we know how to identify one good chip g. Since a good chip always accurately reports whether the other chip in a pairwise test is good or bad, just test g against each of the other $n - 1$ chips, requiring an additional $n - 1 = \Theta(n)$ pairwise tests.

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 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, you can determine if they're 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
      interview candidate i
      if candidate i is better than candidate best
          best = i
          hire candidate i
```

Cost

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

- Have to pay nc_i to interview, no matter how many you hire.
- So we focus on analyzing the hiring cost mc_h.
- mc_h varies with each run—it depends on the order in which you interview the candidates.
- This is a model of a common paradigm: 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, you 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 you hire all n, then the cost is $O(c_i n + c_h n) = O(c_h n)$ (since $c_h > c_i$).

Probabilistic analysis

In general, you 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: $\mathit{rank}(i)$ is a unique integer in the range 1 to n.
- The ordered list $\langle \mathit{rank}(1), \mathit{rank}(2), \ldots, \mathit{rank}(n) \rangle$ 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

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

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

Instead, 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 you a list of all n candidates in advance.
- On each day, you randomly choose a candidate from the list to interview (but considering only those not yet interviewed).
- Instead of relying on the candidates being presented in a random order, 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***.

- $\text{R}\small{\text{ANDOM}}(a, b)$ 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. Provides an easy way to convert a probability to an expectation.

Helpful in situations in which there may be dependence.

Given a sample space and an event A, 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 from one flip of a fair coin.

- 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.41).
- Instead, 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 (equation (C.24)) 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 5.2-5 is a problem with lots of dependence. See the solution on page 5-12 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 you 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 ⇒ $\mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\}$.

Need to determine $\Pr\{\text{candidate } i \text{ is hired}\}$.

- 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 ⇒ candidates $1, 2, \ldots, i$ arrive in random order ⇒ 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, by the lemma, implies $\mathrm{E}[X_i] = 1/i$.

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

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/i \\
&= \ln n + O(1) \quad \text{(equation (A.9): the sum is a harmonic series)} .
\end{aligned}$$

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

Randomized algorithms

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

The hiring problem

For the hiring problem, the algorithm is deterministic:

- For any given input, the number of times you hire a new office assistant will always be the same.

- The number of times you 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 you 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 run of the algorithm can result in a different hiring cost.
- In other words, in each run of the algorithm, the execution depends on the random choices made.
- No particular input always elicits worst-case behavior.
- Bad behavior occurs only if you 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

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 5-15 of this manual.)

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

```
RANDOMLY-PERMUTE(A, n)
  for $i = 1$ to $n$
      swap $A[i]$ with $A[\text{RANDOM}(i, n)]$
```

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. (On page 1180 in Appendix C.)

Lemma

RANDOMLY-PERMUTE 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.16) $\Rightarrow \Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\}$.

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: The loop terminates, since it's a **for** loop iterating n times. 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
      $x$ = BIASED-RANDOM()
      $y$ = BIASED-RANDOM()
      if $x \neq y$
          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.36), 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
This solution is also posted publicly

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 $i \leq n-1$ 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.16),

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

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 5.2-3

For $i = 1, 2, \ldots, 6$, define the indicator random variable

$$\begin{aligned}
X_i &= \mathrm{I}\{i \text{ is rolled}\} \\
&= \begin{cases} 1 & \text{if } i \text{ is rolled} , \\ 0 & \text{if } i \text{ is not rolled} . \end{cases}
\end{aligned}$$

Since each face value has a probability of $1/6$ of being rolled, $\mathrm{E}[X_i] = 1/6$, for $i = 1, 2, \ldots, 6$.

Now define a random variable Y_j that is equal to the value rolled on die j, where $j = 1, 2, \ldots, n$. Then,

$$\begin{aligned}
\mathrm{E}[Y_j] &= \sum_{i=1}^{6} i \, \mathrm{E}[X_i] \\
&= \sum_{i=1}^{6} i \frac{1}{6} \\
&= \frac{1}{6} \sum_{i=1}^{6} i \\
&= \frac{7}{2} .
\end{aligned}$$

Finally, define a random variable Z equal to the sum of the n dice rolls. We want to compute $E[Z]$. Therefore, we have

$$\begin{aligned}
\mathrm{E}[Z] &= \sum_{j=1}^{n} \mathrm{E}[Y_j] \\
&= \sum_{j=1}^{n} \frac{7}{2} \\
&= \frac{7n}{2} ,
\end{aligned}$$

and so $7n/2$ is the expected sum of n dice rolls.

Solution to Exercise 5.2-4

From Exercise 5.2-3, the expected value of the sum of two dice is 7 ($7n/2$ for $n = 2$).

For the case in which the first die is rolled normally and the second die is set equal to the first die's value, define the indicator random variable

$$X_i = \mathrm{I}\{i \text{ is rolled on the first die}\} \ .$$

We have $\mathrm{E}[X_i] = 1/6$ for $i = 1, \ldots, 6$. Define a random variable Y that is equal to the sum of the two dice, so that

$$\begin{aligned}
\mathrm{E}[Y] &= \sum_{i=1}^{6} 2i\,\mathrm{E}[X_i] \\
&= \sum_{i=1}^{6} 2i\,\frac{1}{6} \\
&= \frac{1}{3}\sum_{i=1}^{6} i \\
&= 7 \ .
\end{aligned}$$

For the case in which the first die is rolled normally and the second die is set to 7 minus the value of the first die, define the indicator random variables X_i and Y as above. This time, we have

$$\begin{aligned}
\mathrm{E}[Y] &= \sum_{i=1}^{6} (i + (7 - i))\mathrm{E}[X_i] \\
&= \sum_{i=1}^{6} 7 \cdot \frac{1}{6} \\
&= 7 \ .
\end{aligned}$$

Solution to Exercise 5.2-5
This solution is also posted publicly

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 $\pi(i) = i$.) We 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 their 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 their 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 5.2-6
This solution is also posted publicly

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

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

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

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

Solution to Exercise 5.3-1

Here's the rewritten procedure:

```
RANDOMLY-PERMUTE($A, n$)
  swap $A[1]$ with $A[$RANDOM$(1, n)]$
  for $i = 2$ to $n$
      swap $A[i]$ with $A[$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
This solution is also posted publicly

Along with the identity permutation, there are other permutations that PERMUTE-WITHOUT-IDENTITY 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, PERMUTE-WITHOUT-IDENTITY can produce only $2 \cdot 1 = 2$ possible permutations, 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 $n = 3$. The procedure makes 3 calls to RANDOM, each of which returns one of 3 values, and so calling PERMUTE-WITH-ALL has 27 possible outcomes. 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 sum to 1, none are equal to $1/6$.

Solution to Exercise 5.3-4
This solution is also posted publicly

PERMUTE-BY-CYCLIC chooses *offset* as a random integer in the range $1 \leq \textit{offset} \leq n$, and then it performs a cyclic rotation of the array. That is, $B[((i + \textit{offset} - 1) \bmod n) + 1] = A[i]$ 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 $B[(i + \textit{offset}) \bmod n] = A[i]$ 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 5.3-5

First, we show that the set S returned by RANDOM-SAMPLE contains m elements. Each iteration of the **for** loop adds exactly one element into S. The number of iterations is $n - (n - m + 1) + 1 = m$, and so $|S| = m$ at completion.

Because the elements of set S are chosen independently of each other, it suffices to show that each of the n values appears in S with probability m/n. We use an inductive proof. The inductive hypothesis is that after an iteration of the **for** loop for a specific value of k, the set S contains $|S| = k - (n - m)$ elements, each appearing with probability $|S|/k$. During the first iteration, i is equally likely to be any integer in $\{1, 2, \ldots, k\}$ and is added to S, which is initially empty; therefore the inductive hypothesis holds after the first iteration.

For the inductive step, denote by S' the set S after the iteration for $k - 1$. By the inductive hypothesis, $|S'| = |S| - 1 = k - 1 - (n - m)$, and each value in $\{1, 2, \ldots, k - 1\}$ appears with probability $|S'|/(k - 1)$. For the iteration with value k, we consider separately the probabilities that S contains $j < k$ and that S contains k. Let R_j be the event that the call RANDOM$(1, k)$ returns j, so that $\Pr\{R_j\} = 1/k$.

For $j < k$, the event that $j \in S$ is the union of two disjoint events:

- $j \in S'$, and
- $j \notin S'$ and R_j (these events are independent),

Thus,

$$\begin{aligned}
\Pr\{j \in S\} &= \Pr\{j \in S'\} + \Pr\{j \notin S' \text{ and } R_j\} && \text{(the events are disjoint)} \\
&= \frac{|S'|}{k-1} + \left(1 - \frac{|S'|}{k-1}\right) \cdot \frac{1}{k} && \text{(by the inductive hypothesis)} \\
&= \frac{|S| - 1}{k-1} + \left(\frac{k-1}{k-1} - \frac{|S| - 1}{k-1}\right) \cdot \frac{1}{k} \\
&= \frac{|S| - 1}{k-1} \cdot \frac{k}{k} + \frac{k - |S|}{k-1} \cdot \frac{1}{k} \\
&= \frac{(|S| - 1)k + (k - |S|)}{(k-1)k} \\
&= \frac{|S|\,k - k + k - |S|}{(k-1)k} \\
&= \frac{|S|\,(k-1)}{(k-1)k} \\
&= \frac{|S|}{k} .
\end{aligned}$$

The event that $k \in S$ is also the union of two disjoint events:

- R_k, and
- R_j and $j \in S'$ for some $j < k$ (these events are independent).

Thus,

$$
\begin{aligned}
&\Pr\{k \in S\} \\
&= \Pr\{R_k\} + \Pr\{R_j \text{ and } j \in S' \text{ for some } j < k\} && \text{(the events are disjoint)} \\
&= \frac{1}{k} + \left(1 - \frac{1}{k}\right) \cdot \frac{|S'|}{k-1} && \text{(by the inductive hypothesis)} \\
&= \frac{1}{k} + \frac{k-1}{k} \cdot \frac{|S'|}{k-1} \\
&= \frac{1}{k} \cdot \frac{k-1}{k-1} + \frac{k-1}{k} \cdot \frac{|S|-1}{k-1} \\
&= \frac{k-1+k\,|S|-k-|S|+1}{k(k-1)} \\
&= \frac{k\,|S|-|S|}{k(k-1)} \\
&= \frac{|S|\,(k-1)}{k(k-1)} \\
&= \frac{|S|}{k} \,.
\end{aligned}
$$

Therefore, after the last iteration, in which $k = n$, each of the n values appears in S with probability $|S|/n = m/n$.

Solution to Exercise 5.4-7

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

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

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

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 (3.16), 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

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

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.33), $\mathrm{Var}[V_n] = \mathrm{Var}[X_1] + \mathrm{Var}[X_2] + \cdots + \mathrm{Var}[X_n]$.

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.31),

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

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

Solution to Problem 5-2

a. Here is pseudocode for RANDOM-SEARCH:

```
RANDOM-SEARCH($A, n, x$)
  allocate an array $checked[1:n]$
  for $i = 1$ to $n$
      $checked[i] = $ FALSE
  $count = 0$          // number of TRUE entries in $checked$
  while $count < n$
      $i = $ RANDOM$(1, n)$
      if $A[i] == x$
          return $i$
      elseif $checked[i] ==$ FALSE
          $checked[i] = $ TRUE
          $count = count + 1$
  return NIL
```

b. We can model this question with a geometric distribution, where success means finding x. The probability of success is $1/n$, and so by equation (C.36), the expected number of indices into A that must be picked equals n.

c. Using the same idea as in part (b), if k positions in A contain x, then the probability of success becomes k/n. Equation (C.36) says that the expected number of indices into A that must be picked equals n/k.

d. We can model this question using by the answer to "How many balls must you toss until every bin contains at least one ball?" on page 143. Here, a ball landing in a particular bin corresponds to picking a particular array index. There are n "bins", so that using the analysis on page 143, the expected number of indices into A that must be picked is $n(\ln n + O(1))$.

e. Intuitively, we know that the average-case running time of DETERMINISTIC-SEARCH is $\Theta(n)$ when x appears in the array exactly once, because on average x appears halfway into the array. Let's prove it rigorously. Let $X_i = \mathrm{I}\{A[i] \text{ is examined}\}$ and $X = \sum_{i=1}^{n} X_i$ equal the number of indices of A that are examined. Suppose that $A[j] = x$. Then $A[i]$ is examined if it occurs before $A[j]$. Taking positions i and j together, the probability that $i \neq j$ occurs before j for random i and j is 1/2, so that $\mathrm{E}[X_i] = 1/2$ if $i \neq j$. Position j is always examined, so that $\mathrm{E}[X_j] = 1$. Thus,

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \mathrm{E}\left[\sum_{1\le i\le n, i\neq j} X_i + X_j\right] \\
&= \sum_{1\le i\le n, i\neq j} \mathrm{E}[X_i] + \mathrm{E}[X_j] \quad \text{(by linearity of expectation)} \\
&= (n-1)\cdot\frac{1}{2} + 1 \\
&= \frac{n+1}{2}\ .
\end{aligned}$$

In the worst case, $A[n] = x$, and the running time is also $\Theta(n)$.

f. As in part (e), let $X_i = \mathrm{I}\{A[i] \text{ is examined}\}$ and $X = \sum_{i=1}^{n} X_i$ equal the number of indices of A that are examined. Let $S = \{i : A[i] = x\}$ and $\overline{S} = \{i : A[i] \neq x\}$, so that $|S| = k$ and $|\overline{S}| = n - k$. A position $i \in S$ is examined only if it's the first position in S, which occurs with probability $1/k$. A position $i \notin S$ is examined only if, out of position i and all k positions in S, i is the first position, which occurs with probability $1/(k+1)$. Thus,

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i\in S} \mathrm{E}[X_i] + \sum_{i\notin S} \mathrm{E}[X_i] \quad \text{(by linearity of expectation)} \\
&= k\cdot\frac{1}{k} + (n-k)\cdot\frac{1}{k+1} \\
&= 1 + \frac{n-k}{k+1} \\
&= \frac{k+1}{k+1} + \frac{n-k}{k+1}
\end{aligned}$$

$$= \frac{n+1}{k+1} .$$

Observe that when $k = 1$, we get the same result as in part (e).

In the worst case, the k positions of A containing x are the last k positions, so that the running time is $\Theta(n - k + 1)$.

g. If x does not appear in A, then all positions of A are examined in all cases, so that the running time is $\Theta(n)$.

h. SCRAMBLE-SEARCH runs in the time to randomly permute the array, plus the time for DETERMINISTIC-SEARCH. Assuming that randomly permuting the array takes $\Theta(n)$ time (for example, by calling RANDOMLY-PERMUTE on page 136), SCRAMBLE-SEARCH runs in $\Theta(n)$ time in all cases.

i. Of the three searching algorithms, DETERMINISTIC-SEARCH has the best expected and worst-case running times.

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

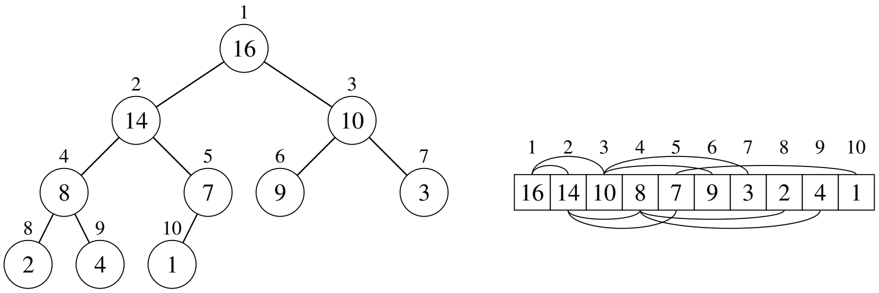
- A heap (*not* garbage-collected storage) is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap $=$ height of root $= \Theta(\lg n)$.
- A heap can be stored as an array A.
  - Root of tree is $A[1]$.
 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

```
PARENT(i)
  return $\lfloor i/2 \rfloor$

LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$
```

- Computing is fast with binary representation implementation.
- Attribute $A.\mathit{heap\text{-}size}$ says how many elements are stored in A. Only the elements in $A[1 : A.\mathit{heap\text{-}size}]$ are in the heap.

Example

Of a max-heap in array with $\mathit{heap\text{-}size} = 10$. *[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.

[In general, heaps can be k-ary trees instead of binary trees.]

Maintaining the heap property

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

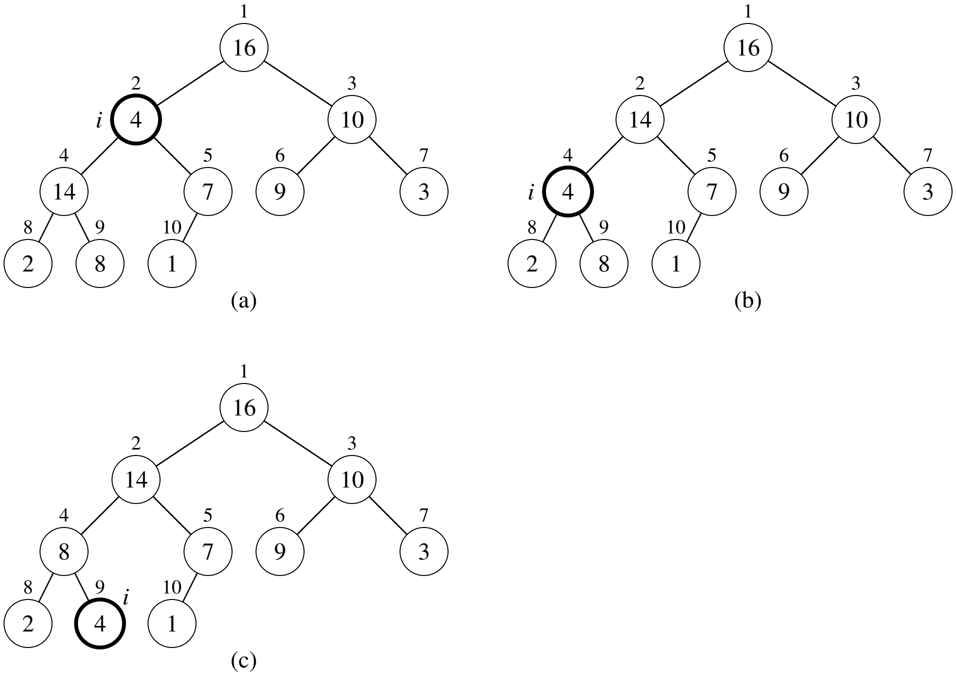
- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume that left and right subtrees of i are max-heaps. (No violations of max-heap property within the left and right subtrees. The only violation within the subtree rooted at i could be between i and its children.)
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

```
MAX-HEAPIFY$(A, i)$
  $l = $ LEFT$(i)$
  $r = $ RIGHT$(i)$
  **if** $l \leq A.heap\text{-}size$ and $A[l] > A[i]$
      $largest = l$
  **else** $largest = i$
  **if** $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
      $largest = r$
  **if** $largest \neq i$
      exchange $A[i]$ with $A[largest]$
      MAX-HEAPIFY$(A, largest)$
```

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.



(a)

(b)

(c)

- 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

$O(\lg n)$.

Analysis

[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 $A[1:n]$, will produce a max-heap of the n elements in A.

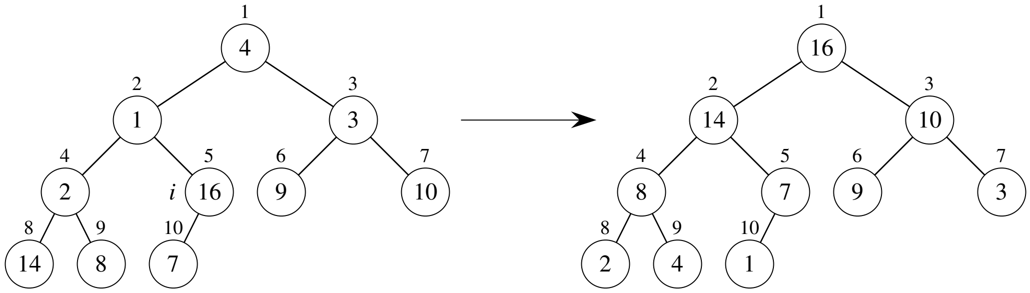
```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

Example

Building a max-heap by calling BUILD-MAX-HEAP$(A, 10)$ on the following unsorted array $A[1:10]$ results in the first heap example.

- $A.heap\text{-}size$ is set to 10.
- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): $A[5]$, $A[4]$, $A[3]$, $A[2]$, $A[1]$.

	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 6.1-8, 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 $\Rightarrow O(n \lg n)$. *[A good approach to analysis in general is to start by proving an 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 $\le \lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 6.3-4), 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.11) $\left(\sum_{k=0}^{\infty} kx^k\right)$, which yields

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

  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:

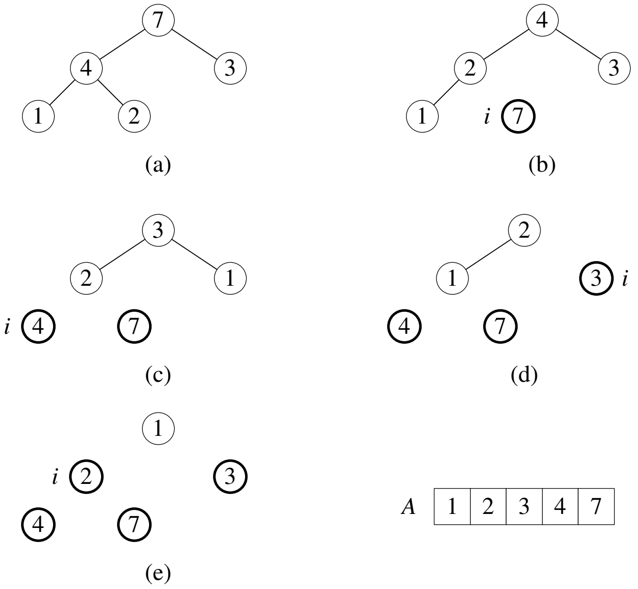
- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.

- "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 $i = n$ downto 2
      exchange $A[1]$ with $A[i]$
      $A.heap\text{-}size = A.heap\text{-}size - 1$
      MAX-HEAPIFY$(A, 1)$
```

Example

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



(a) (b) (c) (d) (e)

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.

Priority queues

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, k): inserts element x with key k 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. Assumes $k \geq x$'s current key value.
- Example max-priority queue application: schedule jobs on shared computer. Scheduler adds new jobs to run by calling INSERT and runs the job with the highest priority among those pending by calling EXTRACT-MAX.
- Min-priority queue supports similar operations:
  - INSERT(S, x, k): inserts element x with key k 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. Assumes $k \leq x$'s current key value.
- Example min-priority queue application: event-driven simulator. Events are simulated in order of time of occurrence by calling EXTRACT-MIN.

Elements in the priority queue correspond to objects in the application that uses the priority queue. Each object contains a key. Need to be able to map between application objects and array indices in the heap. Two suggested ways:

- Each heap element has a ***handle*** that allows access to an object in the application, and each object in the application has a handle (likely an array index) to access the heap element. Good software engineering practice is to make the handles opaque to the surrounding code.
- Store within the priority queue a mapping from application objects to array indices in the heap.

  Advantage: application objects don't need to use handles.

  Disadvantage: need to establish and maintain the mapping.

  One option would be to use a hash table (see Chapter 11) *[which is how Python dictionaries are implemented]* .

Will examine how to implement max-priority queue operations.

Finding the maximum element

Getting the maximum element is easy: it's the root. First, check that the heap is not empty.

```
MAX-HEAP-MAXIMUM(A)
  if A.heap-size < 1
      error "heap underflow"
  return the element in A[1]
```

Time

$\Theta(1)$.

Extracting the maximum element

Given the array A:

- Identify the maximum element by calling MAX-HEAP-MAXIMUM.
- Make the last node in the tree the new root.
- Remove the last node from the heap by decrementing *heap-size*.
- Re-heapify the heap by calling MAX-HEAPIFY. Implicitly assume that MAX-HEAPIFY compares objects based on keys, and also that it updates the mapping between objects and array indices as necessary.
- Return the copy of the maximum element.

```
MAX-HEAP-EXTRACT-MAX(A)
  max = MAX-HEAP-MAXIMUM(A)
  A[1] = A[A.heap-size]
  A.heap-size = A.heap-size - 1
  MAX-HEAPIFY(A, 1)        // remakes heap
  return max
```

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 the value of an object's key

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

- Make sure $k \geq x$'s current key.
- Update x's key value to k.
- Find where in the array where x occurs.
- Traverse the tree upward comparing the key to its parent's key and swapping keys if necessary, until the node's key is smaller than its parent's key or reach the root. Update the mapping between objects and array indices as necessary.

```
MAX-HEAP-INCREASE-KEY$(A, x, k)$
  if $k < x.key$
      error "new key is smaller than current key"
  $x.key = k$
  find the index $i$ in array $A$ where object $x$ occurs
  while $i > 1$ and $A[\text{PARENT}(i)].key < A[i].key$
      exchange $A[i]$ with $A[\text{PARENT}(i)]$, updating the information that maps
          priority queue objects to array indices
      $i = \text{PARENT}(i)$
```

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 an object x to insert into the heap:

- Check that the heap has space for a new object.
- Add a new node to the heap by incrementing *heap-size*.
- Insert a new node in the last position in the heap, with key $-\infty$.
- Save the value of x's key in the variable k, and set x's key to $-\infty$.
- Make x be the last node in the heap, updating the mapping between objects and array indices as necessary.
- Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure defined above.

```
MAX-HEAP-INSERT$(A, x, n)$
  if $A.heap\text{-}size == n$
      error "heap overflow"
  $A.heap\text{-}size = A.heap\text{-}size + 1$
  $k = x.key$
  $x.key = -\infty$
  $A[A.heap\text{-}size] = x$
  map $x$ to index $heap\text{-}size$ in the array
  MAX-HEAP-INCREASE-KEY$(A, x, k)$
```

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

This solution is also posted publicly

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

This solution is also posted publicly

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

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

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

Solution to Exercise 6.1-3

Assume that 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 $A[\text{PARENT}(m)] \geq A[m]$. So our assumption is false, and the claim is true.

Solution to Exercise 6.1-4

The smallest element must reside in a leaf.

Solution to Exercise 6.1-5

For $2 \le k \le \lfloor n/2 \rfloor$, the kth largest element could be at any level except the root. Consider a max-heap that is a full binary tree with the $\lfloor n/2 \rfloor$ largest elements other than the root in the left subtree of the root and the $\lfloor n/2 \rfloor$ smallest elements in the right subtree.

Solution to Exercise 6.1-6

Yes, an array in sorted order is a min-heap, since $A[i] \le A[2i]$ where $2i \le n$, and $A[i] \le A[2i + 1]$ where $2i + 1 \le n$.

Solution to Exercise 6.1-7

No, this array is not a max-heap. Element 15 has 13 and 16 as children, and $16 > 15$.

Solution to Exercise 6.1-8

To show that the leaves of an n-element heap stored in the array representation are indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$, it suffices to show that node $\lfloor n/2 \rfloor$ is not a leaf and that node $\lfloor n/2 \rfloor + 1$ is a leaf.

To show that node $\lfloor n/2 \rfloor$ is not leaf, we just need to show that it has a left child, i.e., that the index of its left child is at most n. The left child has index $2 \lfloor n/2 \rfloor \le 2(n/2) = n$.

To show that node $\lfloor n/2 \rfloor + 1$ is a leaf, we need to show that it has no left child, i.e., that the index of the node that would be its left child is greater than n. The index of the would-be left child is $2(\lfloor n/2 \rfloor + 1) > 2((n/2 - 1) + 1) = n$.

Thus, the leaves are indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

Solution to Exercise 6.2-2

The greatest imbalance occurs when the left subtree of the root is a full binary tree with k levels and the right subtree of the root is a full binary tree with $k-1$ levels. The left subtree then contains $2^k - 1$ nodes, and the right subtree contains $2^{k-1} - 1$ nodes. With the root, the total number of nodes is $n = (2^k - 1) + (2^{k-1} - 1) + 1 = 2^k + 2^{k-1} - 1$. Therefore, the ratio of nodes in the left subtree to the total number of nodes is

$$\begin{aligned}
\frac{2^k - 1}{2^k + 2^{k-1} - 1} &= \frac{2 \cdot 2^{k-1} - 1}{3 \cdot 2^{k-1} - 1} \\
&< \frac{2 \cdot 2^{k-1}}{3 \cdot 2^{k-1}} \quad \text{(the numerator is smaller than the denominator)} \\
&= 2/3 \; .
\end{aligned}$$

The smallest constant α such that each subtree has at most αn nodes is $3/5$, and it occurs in a heap with 5 nodes. As long as α is a constant strictly less than 1, the recurrence $T(n) \le T(\alpha n) + \Theta(1)$ has the same solution of $O(\lg n)$.

Solution to Exercise 6.2-4

The heap doesn't change because line 8 of MAX-HEAPIFY finds that *largest* $= i$.

Solution to Exercise 6.2-5

The heap doesn't change because by Exercise 6.1-8, node i is a leaf for $i > A.\mathit{heap\text{-}size}/2$.

Solution to Exercise 6.2-6

```
ITERATIVE-HEAPIFY$(A, i)$
  $heapified = $ FALSE
  while $heapified ==$ FALSE
      $l = $ LEFT$(i)$
      $r = $ RIGHT$(i)$
      if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$
          $largest = l$
      else $largest = i$
      if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
          $largest = r$
      if $i \neq largest$
          exchange $A[i]$ with $A[largest]$
          $i = largest$
      else $heapified = $ TRUE
```

Solution to Exercise 6.2-7
This solution is also posted publicly

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 greater than or equal to 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 6.3-2

For $0 \leq h \leq \lceil \lg n \rceil$, we have

$$\begin{aligned}\left\lceil \frac{n}{2^{h+1}} \right\rceil &\geq \left\lceil \frac{n}{2^{\lg n+1}} \right\rceil \\ &= \left\lceil \frac{n}{2n} \right\rceil \\ &\geq 1/2 \,.\end{aligned}$$

Solution to Exercise 6.3-3

We want to proceed from the leaves to the root because MAX-HEAPIFY assumes that both subtrees of node i have the max-heap property.

Solution to Exercise 6.3-4

This solution relies on five facts:

1. Every node *not* on the unique simple path from the last leaf to the root is the root of a complete binary subtree.
2. A node that is the root of a complete binary subtree and has height h is the ancestor of 2^h leaves. (If $h = 0$, then the node is a leaf and its own ancestor.)
3. By Exercise 6.1-8, an n-element heap has $\lceil n/2 \rceil$ leaves.
4. For nonnegative reals a and b, we have $\lceil a \rceil \cdot b \ge \lceil ab \rceil$.
5. Subtrees whose roots have equal heights are disjoint.

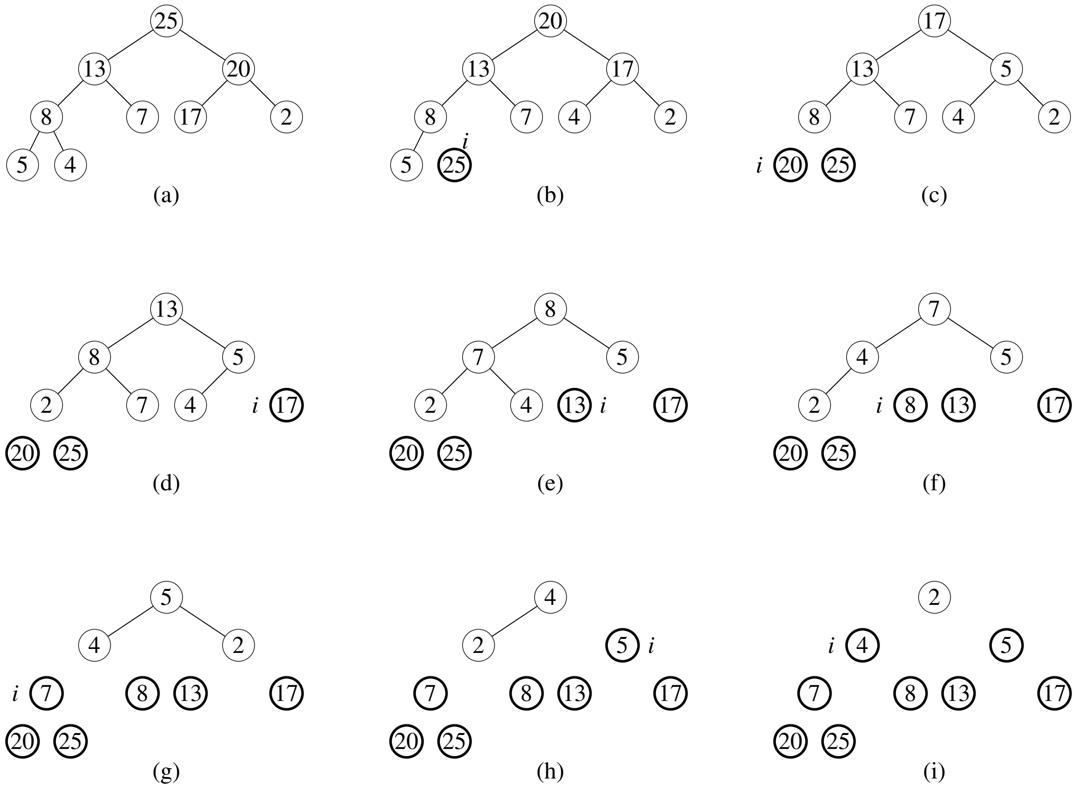
The proof is by contradiction. Assume that for some height h, an n-element heap contains at least $\lceil n/2^{h+1} \rceil + 1$ nodes of height h. Exactly one node of height h is on the unique simple path from the last leaf to the root, and the subtree rooted at this node has at least one leaf (that being the last leaf). All other nodes of height h, of which the heap contains at least $\lceil n/2^{h+1} \rceil$, are the roots of complete binary subtrees, and each such node is the root of a subtree with 2^h leaves. Since each subtree whose root is at height h is disjoint, the number of leaves in the entire heap is at least

$$\begin{aligned}
\left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot 2^h + 1 &\ge \left\lceil \frac{n}{2^{h+1}} \cdot 2^h \right\rceil + 1 \\
&= \left\lceil \frac{n}{2} \right\rceil + 1 ,
\end{aligned}$$

which contradicts the property that an n-element heap has $\lceil n/2 \rceil$ leaves.

Solution to Exercise 6.4-1

This solution is also posted publicly



(a) (b) (c) (d) (e) (f) (g) (h) (i)

A	2	4	5	7	8	13	17	20	25

Solution to Exercise 6.4-2

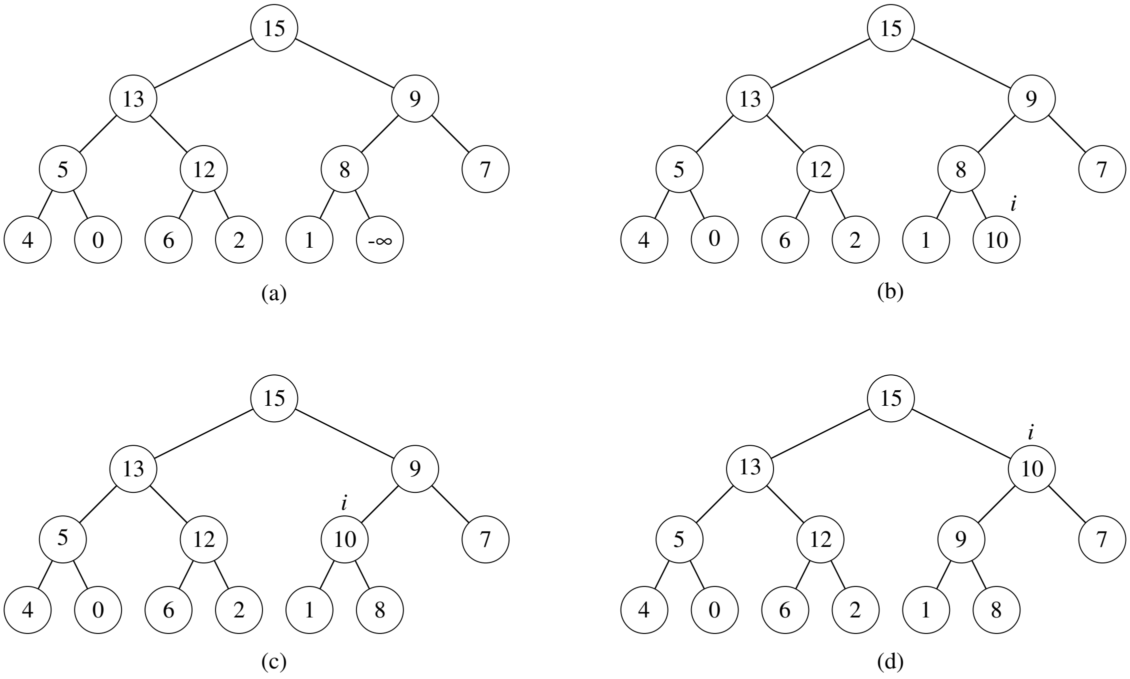
Initialization: Prior to the first iteration, $i = n$. The subarray $A[1:i]$ is the entire array, which is a heap from having already called BUILD-MAX-HEAP, and the subarray $A[i+1:n]$ is empty.

Maintenance: Because $A[1:i]$ is a max-heap, the largest element out of the i smallest is in $A[1]$. Because $A[i+1:n]$ contains the $n-i$ largest elements, sorted, moving $A[1]$ to position i makes $A[i:n]$ the $n-i+1$ largest elements, sorted. Moving $A[i]$ to $A[1]$, decrementing $A.heap\text{-}size$, and calling MAX-HEAPIFY makes $A[1:i-1]$ a max-heap. Decrementing i reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 1$. The subarray $A[1:i]$ is just one node, which is the smallest element in $A[1:n]$. The subarray $A[i+1:n]$ is $A[2:n]$, which contains the $n-1$ largest elements in $A[1:n]$, sorted. Thus, $A[1:n]$ is sorted.

Solution to Exercise 6.5-2

This solution is also posted publicly



Solution to Exercise 6.5-4

```
MAX-HEAP-DECREASE-KEY(A, x, k)
  if k > x.key
      error "new key is greater than current key"
  x.key = k
  find the index i in array A where object x occurs
  while i > 1 and A[PARENT(i)].key > A[i].key
      exchange A[i] with A[PARENT(i)], updating the information that maps
          priority queue objects to array indices
      i = PARENT(i)
```

The running time is $O(\lg n)$ plus the overhead for mapping priority queue objects to array indices.

Solution to Exercise 6.5-5

Setting the key of the inserted object to $-\infty$ avoids the "new key is smaller than current key" error in MAX-HEAP-INCREASE-KEY.

Solution to Exercise 6.5-7

Initialization: The subarray $A[1:A.\mathit{heap\text{-}size}]$ satisfies the max-heap property at the time MAX-HEAP-INCREASE-KEY is called, so at that moment all three parts of the loop invariant hold. When $A[i].\mathit{key}$ increases, that does not change the relationship between $A[\text{PARENT}(i)].\mathit{key}$ and $A[\text{LEFT}(i)].\mathit{key}$ or between $A[\text{PARENT}(i)].\mathit{key}$ and $A[\text{RIGHT}(i)].\mathit{key}$, assuming that these nodes exist. Increasing $A[i].\mathit{key}$ could cause $A[i].\mathit{key}$ to become greater than $A[\text{PARENT}(i)].\mathit{key}$ entering the first iteration of the loop.

Maintenance: Entering a loop iteration, $A[\text{PARENT}(i)].\mathit{key} \ge A[\text{LEFT}(i)].\mathit{key}$ and $A[\text{PARENT}(i)].\mathit{key} \ge A[\text{RIGHT}(i)].\mathit{key}$, if these nodes exist. The loop iteration swaps $A[i].\mathit{key}$ and $A[\text{PARENT}(i)].\mathit{key}$, so that after the swap, $A[i].\mathit{key} \ge A[\text{LEFT}(i)].\mathit{key}$ and $A[i].\mathit{key} \ge A[\text{LEFT}(i)].\mathit{key}$; thus, there is no violation of the max-heap property among node i and its children. Before the swap, the only possible violation of the max-heap property was between node i and its parent, so that $A[i].\mathit{key} \ge A[\text{LEFT}(i)].\mathit{key}$ and $A[i].\mathit{key} \ge A[\text{LEFT}(i)].\mathit{key}$ before the swap. After the swap, i's key moves to its parent, so that $A[\text{PARENT}(i)].\mathit{key} \ge A[\text{LEFT}(i)].\mathit{key}$ and $A[\text{PARENT}(i)].\mathit{key} \ge A[\text{RIGHT}(i)].\mathit{key}$ after the swap. If there is a violation of the max-heap property after the swap, it is between $\text{PARENT}(i)$ and $\text{PARENT}(\text{PARENT}(i))$. Setting $i = \text{PARENT}(i)$ restores the loop invariant for the next iteration.

Termination: The loop terminates because either $i > 1$, so that node i is the root and has no parent, or $A[\text{PARENT}(i)].\mathit{key} \ge A[i].\mathit{key}$. If the second condition never occurs, the first one will because each iteration moves i one level closer to the root. Whether node i is the root or node i's key is no larger than its parent's key, there is no violation of the max-heap property between node i and its parent. By the loop invariant, that would have been the only possible vioation of the max-heap property, and so terminating the loop at that time results in no violations of the max-heap property anywhere in the heap.

Solution to Exercise 6.5-8

Change the procedure to the following:

```
MAX-HEAP-INCREASE-KEY$(A, x, k)$
  if $k < x.key$
      error “new key is smaller than current key”
  $x.key = k$
  find the index $i$ in array $A$ where object $x$ occurs
  while $i > 1$ and $A[\text{PARENT}(i)].key < A[i].key$
      $A[i] = A[\text{PARENT}(i)]$, updating the information that maps
          priority queue objects to array indices
      $i = \text{PARENT}(i)$
  $A[i] = x$, mapping $x$ to index $i$
```

Solution to Exercise 6.5-9

For a stack, use a max-priority queue and keep a counter c of how many objects have been pushed onto the stack, initially 0. To push, call INSERT with the current value of c as the key, and then increment c. To pop, just call MAXIMUM.

For a queue, use a min-priority queue, again keeping the counter c, initially 0. To enqueue, do the same as pushing: call INSERT with the current value of c as the key, and then increment c. To dequeue, call MINIMUM.

Solution to Exercise 6.5-10

```
MAX-HEAP-DELETE$(A, x)$
  find the index $i$ in array $A$ where object $x$ occurs
  $A[i] = A[A.\mathit{heap\text{-}size}]$
  update the mapping information
  $A.\mathit{heap\text{-}size} = A.\mathit{heap\text{-}size} - 1$
  MAX-HEAPIFY$(A, i)$, updating the mapping information
```

Solution to Exercise 6.5-11

Maintain a min-heap that always contains at most k elements, one from each list. The lists are all merged together at once by repeatedly extracting the minimum element from the heap and placing it into the sorted outout. If the element placed into the output was from the ith list, then the next element from the ith list is read in and inserted into the min-heap. Each such heap operation takes $O(\lg k)$ time, for a total of $O(n \lg k)$ time.

Solution to Problem 6-1

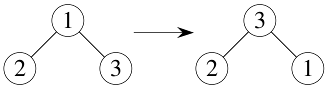
This solution is also posted publicly

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:

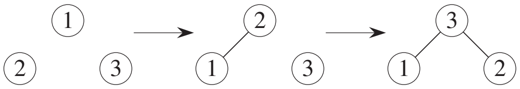
A	1	2	3

BUILD-MAX-HEAP(A):



A	3	2	1

BUILD-MAX-HEAP′(A):



A	3	1	2

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. Represent a d-ary heap in a 1-dimensional array as follows. The root resides in $A[1]$, its d children reside in order in $A[2]$ through $A[d + 1]$, their children reside in order in $A[d + 2]$ through $A[d^2 + d + 1]$, and so on. Nodes at depth k start at index $\left(\sum_{i=0}^{k-1} d^i\right) + 1$ and end at index $\sum_{i=0}^{k} d^i$. The following two procedures map a node with index i to its parent and to its jth child (for $1 \le j \le d$), 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 $1 \le j \le d$. 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 MAX-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. Here is an updated version of MAX-HEAPIFY for a d-ary heap (ignoring the mapping between objects and heap elements). It assumes that the degree d of the d-ary heap is global.

```
MAX-HEAPIFY(A, i)
  rightmost-child = min{D-ARY-CHILD(i, d), A.heap-size}
  largest = i
  j = D-ARY-CHILD(i, 1)
  while j ≤ rightmost-child
      if A[j] > A[largest]
          largest = j
      j = j + 1
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

The running time of MAX-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-INCREASE-KEY given in the text for binary heaps works fine for d-ary heaps too, with calls to PARENT changed to calls to D-ARY-PARENT. The worst-case running time is still $\Theta(h)$, where h is the height of the heap. For a d-ary heap, this running time is $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

e. The MAX-HEAP-INSERT procedure needs no changes for a d-ary heap. The worst-case running time is the same as for MAX-HEAP-INCREASE-KEY: $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

Solution to Problem 6-3

a. There are many ways to arrange these elements in a Young tableau. Here are three of them:

$$\begin{array}{cccc} 2 & 3 & 9 & 12 \\ 4 & 8 & 14 & \infty \\ 5 & \infty & \infty & \infty \\ 16 & \infty & \infty & \infty \end{array} \qquad \begin{array}{cccc} 2 & 3 & 4 & 5 \\ 8 & 9 & \infty & \infty \\ 12 & 14 & \infty & \infty \\ 16 & \infty & \infty & \infty \end{array} \qquad \begin{array}{cccc} 2 & 4 & 12 & \infty \\ 3 & 5 & 16 & \infty \\ 8 & 14 & \infty & \infty \\ 9 & \infty & \infty & \infty \end{array}$$

b. In a Young tableau, each row and each column is in nondecreasing order. If an entry of a row is ∞, then all entries to its right must also be ∞. Likewise, if an entry of a column is ∞, then all entries below it must also be ∞. Therefore, if $Y[1,1] = \infty$, then the rest of row 1 must be ∞. Since columns 1 through n in row 1 are ∞, columns 1 through n must be ∞ in all rows. Hence, the Young tableau Y is emmpty.

The argument goes the other way if $Y[m,n] < \infty$. Each entry in row m must be finite. Because each entry in every column of row m is finite, each entry in every column of every row is finite, so that Young tableau Y is full.

c. MAX-HEAPIFY compares a heap element with its children and, if either of the children is greater than the element, swaps the greater of the children with the heap element and recurses on that child's position. The procedure SINK follows the same idea, but instead of children of a node, it looks at the neighboring elements to the right and below of element $Y[i,j]$ and uses the smaller of the two.

```
SINK($Y, i, j, m, n$)
  if $i < m$
      $below = Y[i+1, j]$
  else $below = \infty$
  if $j < n$
      $right = Y[i, j+1]$
  else $right = \infty$
  if $\min\{below, right\} < \infty$
      if $below < right$
          exchange $Y[i,j]$ with $Y[i+1,j]$
          SINK($A, i+1, j, m, n$)
      else exchange $Y[i,j]$ with $Y[i,j+1]$
          SINK($A, i, j+1, m, n$)
```

EXTRACT-MIN saves the element in $Y[1,1]$ in a local variable, places ∞ into $Y[1,1]$, calls SINK to let this ∞ "sink down" into the Young tableau, and returns the saved value. Before making any changes, however, it checks for the error condition of an empty Young tableau.

```
EXTRACT-MIN(Y, m, n)
  min = Y[1, 1]
  if min == ∞
      error "Young tableau is empty"
  Y[1, 1] = ∞
  SINK(Y, 1, 1)
  return min
```

Clearly, EXTRACT-MIN runs in $O(1)$ time plus the time for SINK. To see that SINK runs in $O(m + n)$ time, observe that each recursive call moves either down by one row or to the right by one column. It must get to $Y[m, n]$ after at most $(m - 1) + (n - 1)$ recursive calls. Since each call takes $O(1)$ time plus the time for the recursive calls, the total time for SINK is $O(m + n)$. In general, the call SINK(Y, i, j, m, n) takes $O(m + n - (i + j))$ time.

d. If the Young tableau Y is not full, then $Y[m, n] = \infty$. INSERT works by inserting the new element k in $Y[m, n]$ and then letting it float up and/or to the left. The procedure FLOAT is analogous to SINK, replacing ∞ by $-\infty$ and going up instead of down and left instead of right.

```
INSERT(Y, k, m, n)
  if Y[m.n] ≠ ∞
      error "Young tableau is full"
  Y[m, n] = k
  FLOAT(Y, m, n)
```

```
FLOAT(Y, i, j)
  if i > 1
      above = Y[i - 1, j]
  else above = -∞
  if j > 1
      left = Y[i, j - 1]
  else left = -∞
  if max {above, left} > -∞
      if above > left
          exchange Y[i, j] with Y[i - 1, j]
          FLOAT(Y, i - 1, j)
      else exchange Y[i, j] with Y[i, j - 1]
          FLOAT(Y, i, j - 1)
```

Just as SINK runs in $O(m + n)$ time, so does FLOAT, since each recursive call decrements either i or j.

e. To sort n^2 numbers in $O(n^3)$ with a Young tableau:

1. Create an empty $n \times n$ Young tableau. Time: $\Theta(n^2)$.
2. Call INSERT for each of the n^2 numbers. Time: $O(n^3)$.
3. Create an index l into the output array, and initialize it to 1. Time: $\Theta(1)$, or $O(n)$ if the output array needs to be created.

4. Call EXTRACT-MIN n^2 times, placing each returned number into index l of the output array, then incrementing l. Time: $O(n^3)$.

Total time: $O(n^3)$.

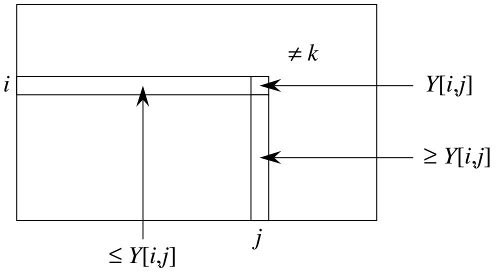
f. The following procedure returns either the position (i, j) of the number k in Young tableau Y, or NIL if k does not appear in Y.

```
SEARCH($Y, k, m, n$)
 $i = 1$
 $j = n$
 while $i \le m$ and $j \ge 1$
     if $k == Y[i, j]$
         return $(i, j)$
     elseif $k < Y[i, j]$
         $j = j - 1$
     else $i = i + 1$
 return NIL
```

The SEARCH procedure maintains the following loop invariant:

Loop invariant: At the start of each iteration of the **while** loop, $Y[i', j'] \ne k$ for all $i' < i$ or $j' > j$.

Pictorially, the loop invariant looks like this:



Initialization: Initially, $i = 1$ and $j = n$, so that no row numbers are less than i and no column numbers are greater than j. That is, $\{Y[i', j'] : i' < i$ and $j' > j\}$ is an empty set.

Maintenance: If $k < Y[i, j]$, then as the figure shows, $k < Y[i', j]$ for all $i' > i$. The section labeled "$\ge Y[i, j]$" contains only numbers that are greater than k. Therefore, this section can be ruled out. Decrementing j does so and maintains the loop invariant for the next iteration. If instead, $k > Y[i, j]$, then $k > Y[i, j']$ for all $j' < j$, so that the section labeled "$\le Y[i, j]$" contains only numbers that are less than k. Thus, this section can be ruled out, and incrementing i does so, maintaining the loop invariant for the next iteration.

Termination: The loop terminates for one of three reasons. If $k = Y[i, j]$, then SEARCH returns (i, j), having found an element equal to k. If $i > m$, then all rows have been determined to contain only numbers not equal to k. In other words, no row contains a number equal to k. The loop terminates, and the procedure returns NIL. Likewise, if $j < 1$, then all columns have been determined to contain only numbers not equal to k, i.e., no column contains a number equal to k. As in the case for $i > m$, the loop terminates, and the procedure returns NIL. The loop is guaranteed to terminate, since each iteration either increments i or decrements j.

The SEARCH procedure runs in $O(m + n)$ time, since in the worst case, the **while** loop increments i m times and decrements j n times, with each iteration taking $O(1)$ time.

Lecture Notes for Chapter 7: Quicksort

Chapter 7 overview

[The treatment in the second and later editions differs from that of the first edition. We use a different partitioning method—known as "Lomuto partitioning"—in the second and third editions, 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 is $\Theta(n^2)$.
- Randomized version has expected running time $\Theta(n \lg n)$, assuming that all elements to be sorted are distinct.
- 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 $\leq A[q]$ and $A[q]$ is $\leq$ 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
      // Partition the subarray around the pivot, which ends up in A[q].
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)  // recursively sort the low side
      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

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

Partitioning

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

```
PARTITION(A, p, r)
  x = A[r]                          // the pivot
  i = p − 1                         // highest index into the low side
  for j = p to r − 1                // process each element other than the pivot
      if A[j] ≤ x                   // does this element belong on the low side?
          i = i + 1                 // index of a new slot in the low side
          exchange A[i] with A[j]   // put this element there
  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
  return i + 1                      // new index of the pivot
```

- 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:

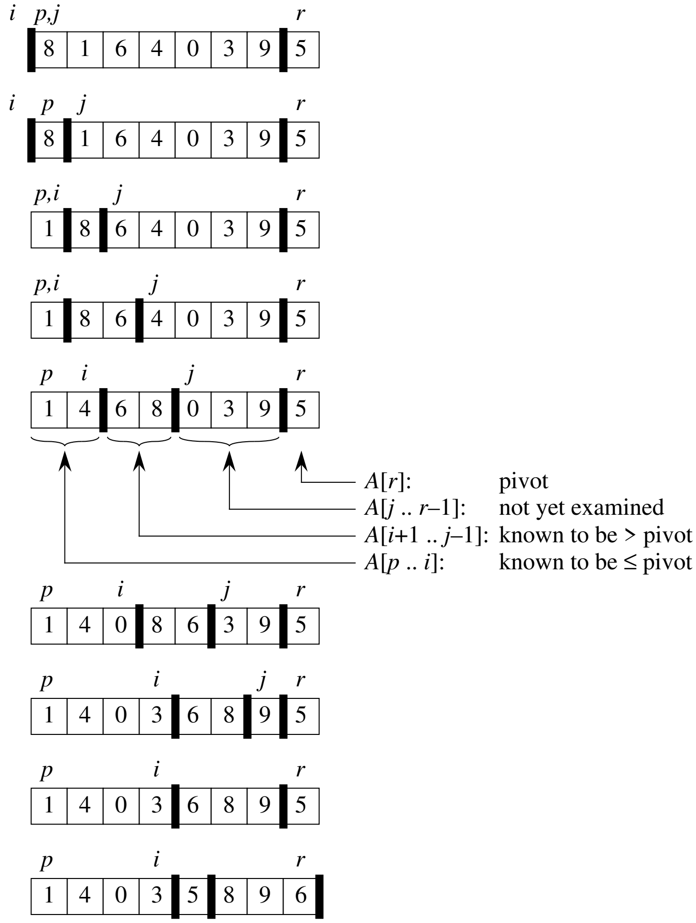
  Loop invariant:

  1. All entries in $A[p:i]$ are $\leq$ 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. *[Differs from the example on page 185 in the book.]*



[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 $A[j] \leq$ 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 that all elements in A are partitioned into one of the three cases: $A[p:i] \leq$ 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 worst-case 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

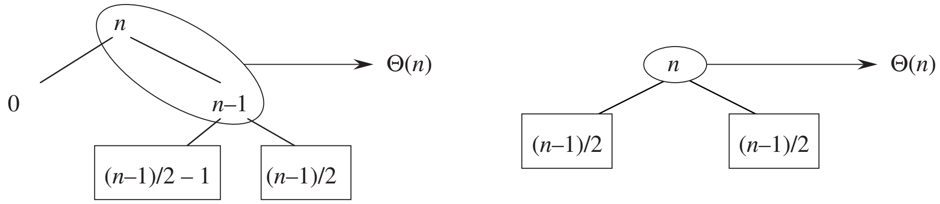
- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\le n/2$ elements.
- For an upper bound, get the recurrence
$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

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
$$\begin{aligned} T(n) &\le T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n) . \end{aligned}$$
- Intuition: look at the recursion tree.
  - It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$ in Section 4.4.
  - Except that here the constants are different: $\log_{10} n$ 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.

```
RANDOMIZED-PARTITION$(A, p, r)$
  $i = $ RANDOM$(p, r)$
  exchange $A[r]$ with $A[i]$
  **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** $p < r$
      $q = $ 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 is highly unlikely to 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:

  $$T(n) = \max\{T(q) + T(n-q-1)) : 0 \le q \le n-1\} + \Theta(n) .$$

- 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:

  $$\begin{aligned} T(n) &\le \max\left\{cq^2 + c(n-q-1)^2 : 0 \le q \le n-1\right\} + \Theta(n) \\ &= c \cdot \max\left\{q^2 + (n-q-1)^2 : 0 \le q \le n-1\right\} + \Theta(n) . \end{aligned}$$

- 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.) Therefore,

  $$\begin{aligned} \max\left\{q^2 + (n-q-1)^2 : 0 \le q \le n-1\right\} &\le (n-1)^2 \\ &= n^2 - 2n + 1 . \end{aligned}$$

- And thus,

  $$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 \qquad \text{if } c(2n-1) \ge \Theta(n) . \end{aligned}$$

- 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 $\Omega(n^2)$. Don't really need to, since we saw that when partitioning is unbalanced, quicksort takes $\Theta(n^2)$ time. Thus, the worst-case running time is $\Theta(n^2)$.

Average-case analysis

- Assume that all values being sorted are distinct. (No repeated values.)
- The dominant cost of the algorithm is partitioning.
- Assume that RANDOMIZED-PARTITION makes it so that the pivot selected by PARTITION is selected randomly from the subarray passed to these procedures.
- 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 $O(n + X)$.

Need to 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. (So that the output order is $z_1, z_2, \ldots, z_n$.)
- 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 with the pivot element, and then the pivot element is never in any later call to PARTITION.

Let $X_{ij} = \mathrm{I}\{z_i \text{ is compared with } z_j\}$.
(Considering whether z_i is compared with 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:

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

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 with 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 with all the elements of Z_{ij}, except itself.
- The probability that z_i is compared with 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,

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

[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.9):

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

So the expected running time of quicksort, using RANDOMIZED-PARTITION, is $O(n \lg n)$ if all values being sorted are distinct.

Solutions for Chapter 7: Quicksort

Solution to Exercise 7.1-2

If all the elements in subarray $A[p:r]$ have the same value, then the test in line 4 of PARTITION evaluates to true every time. When the **for** loop of lines 3–6 terminates, all elements other than the pivot will be in the subarray $A[p:i]$, where $i = r - 1$. Line 7 leaves the pivot in $A[r]$, and line 8 returns r as the pivot index. The result is unbalanced partitioning.

One way to make the partition balanced when all elements in $A[p:r]$ are equal is to check specifically for this case before moving elements around, and just return the index $\lfloor (p+r)/2 \rfloor$ if all elements are equal.

Another way, which does not require a check beforehand, is to keep a flag saying which partition elements equal to the pivot go into, flipping the value of the flag each time the procedure finds an element equal to the pivot. Here is pseudocode:

```
PARTITION($A, p, r$)
  $x = A[r]$
  $i = p - 1$
  $\mathit{flag} = \text{LEFT}$
  for $j = p$ to $r - 1$
      if $A[j] < x$ or ($A[j] == x$ and $\mathit{flag} == \text{LEFT}$)
          if $A[j] == x$
              $\mathit{flag} = \text{RIGHT}$
          $i = i + 1$
          exchange $A[i]$ with $A[j]$
      elseif $A[j] == x$
          $\mathit{flag} = \mathit{left}$
  exchange $A[i + 1]$ with $A[r]$
  return $i + 1$
```

Solution to Exercise 7.1-3

The **for** loop of lines 3–6 iterates $n - 1$ times, and each iteration takes $\Theta(1)$ time. The parts of the procedure outside the loop take $\Theta(1)$ time, for a total of $\Theta(n)$ time.

Solution to Exercise 7.1-4

Just change the test in line 4 to $A[j] \geq x$.

Solution to Exercise 7.2-1

To show that $T(n) = O(n^2)$, denote by c the constant hidden in the $\Theta(n)$ term. Guess that $T(n) \leq dn^2$ for a constant d to be chosen. We have

$$\begin{aligned} T(n) &\leq T(n-1) + cn \\ &\leq d(n-1)^2 + cn \\ &= dn^2 - 2dn + d + cn \ . \end{aligned}$$

This last quantity is less than or equal to dn^2 if $-2dn + d + cn \leq 0$, which is equivalent to $d \geq cn/(2n-1)$. This last inequality holds for all $n \geq 1$ and $d \geq c$.

For the lower bound, use the same c as for the upper bound, and guess that $T(n) \geq dn^2$ for a constant d to be chosen. Changing $\leq$ to $\geq$ above yields $T(n) \geq dn^2$ if $d \leq cn/(2n-1)$, which holds for all $n \geq 1$ and $d \leq c/2$.

Thus, $T(n) = \Theta(n^2)$.

Solution to Exercise 7.2-2

When all elements of array A have the same value, every split of partition yields a maximally unbalanced partition. The recurrence for the running time is then $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

Solution to Exercise 7.2-3
This solution is also posted publicly

Suppose that PARTITION is called on a subarray $A[p:r]$ whose elements are distinct and in decreasing order. PARTITION chooses the smallest element, in $A[r]$, as the pivot. Every test in line 4 comes up false, so that no elements are exchanged during the execution of the **for** loop. Before PARTITION returns, line 6 finds that $i = p - 1$, and so it swaps the elements in $A[p]$ and $A[r]$. PARTITION returns p as the position of the pivot. The subarray containing elements less than or equal to the pivot is empty. The subarray containing elements greater than the pivot, $A[p+1:r]$, has all but the pivot and is in decreasing order except that the maximum element of this subarray is in $A[r]$.

When QUICKSORT calls PARTITION on $A[p:q-1]$, nothing changes, as this subarray is empty. When QUICKSORT calls PARTITION on $A[q+1:r]$, now the pivot is the greatest element in the subarray. Although every test in line 4 comes up true, the indices i and j are always equal in line 6, so that just as in the case where the pivot is the smallest element, no elements are exchanged during the execution of the **for** loop. Before PARTITION returns, line 6 finds that $i = r-1$, so that the swap in line 6 leaves the pivot in $A[r]$. PARTITION returns r as the position of the pivot. Now the subarray containing elements less than or equal to the pivot has all but the pivot and is in decreasing order, and the subarray containing elements greater than the pivot is empty. The next call to PARTITION, therefore, is on a subarray that is in decreasing order, so that it goes back to the first case above.

Therefore, each recursive call is on a subarray only one element smaller, giving a recurrence for the running time of $T(n) = T(n-1) + \Theta(n)$, whose solution is $\Theta(n^2)$.

Solution to Exercise 7.2-4

In the best case, QUICKSORT runs in $\Theta(n \lg n)$ time. In the situation with bank checks, suppose that on average, each check is within k positions of where it belongs in the sorted order, where k is a constant. Then in INSERTION-SORT, array elements move a total of at most kn times overall. Each iteration of the **while** loop of lines 5–7 of INSERTION-SORT moves one element by one position, so that total number of iterations of this loop is at most kn. Since k is a constant, INSERTION-SORT runs in $\Theta(n)$ time in this case, beating QUICKSORT.

Solution to Exercise 7.2-5
This solution is also posted publicly

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 level of recursion reduces the number of elements from n to αn, and i levels of recursion reduce 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 logarithms, we get $m \lg \alpha = -\lg n$, or $m = -\lg n / \lg \alpha$. (This quantity is positive because $0 < \alpha < 1$ implies that $\lg \alpha < 0$.)

Similarly, the maximum-depth path corresponds to always taking the larger part of the partition, i.e., keeping a fraction β of the elements each time. The maximum depth M is reached when there is one element left, that is, when $\beta^M n = 1$. Thus, $M = -\lg n / \lg \beta$. (Again, this quantity is positive because $0 < \beta < 1$ implies that $\lg \beta < 0$.)

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

Solution to Exercise 7.2-6

Let the array have n elements. The split is less balanced than $1 - \alpha$ to α if the pivot occurs in the smallest αn elements or in the largest αn elements. The split is balanced if the pivot occurs anywhere else, i.e., in the middle $n - 2\alpha n$ elements, taken by size. The probability of that happening—any of the middle $n - 2\alpha n$ elements being in the last position, where the pivot is selected—is $(n - 2\alpha n)/n = 1 - 2\alpha$.

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 7.3-2

In the best case, the recursion tree is as balanced as possible at every level. Thinking of a full binary tree with n leaves, each internal node represents a call of RANDOMIZED-QUICKSORT that calls RANDOMIZED-PARTITION. Since a full binary tree with n leaves has $\Theta(n)$ internal nodes, there are $\Theta(n)$ calls to RANDOM in the best case.

The worst-case recursion tree always has a split of $n-1$ to 0, so that each recursive call is on a subproblem only one element smaller. This recursion tree has $n - 1$ internal nodes, so that again there are $\Theta(n)$ calls to RANDOM.

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

$$T(n) = \min\{T(q) + T(n-q-1) : 0 \le q \le n-1\} + \Theta(n) .$$

We guess that $T(n) \ge cn \lg n$ for some constant c. Substituting this guess into the recurrence, we obtain

$$\begin{aligned} T(n) &\ge \min\{cq \lg q + c(n-q-1)\lg(n-q-1) : 0 \le q \le n-1\} + \Theta(n) \\ &= c \cdot \min\{q \lg q + (n-q-1)\lg(n-q-1) : 0 \le q \le n-1\} + \Theta(n) . \end{aligned}$$

As we'll show below, the expression $q \lg q + (n - q - 1)\lg(n - q - 1)$ achieves a minimum over the range $0 \le q \le n-1$ 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 $n - q - 1 = (n-1)/2$, and thus the bound

$$\begin{aligned}
&\min\{q \lg q + (n - q - 1)\lg(n - q - 1) : 0 \le q \le n - 1\} \\
&\qquad \ge \frac{n-1}{2}\lg\frac{n-1}{2} + \left(n - \frac{n-1}{2} - 1\right)\lg\left(n - \frac{n-1}{2} - 1\right) \\
&\qquad = (n-1)\lg\frac{n-1}{2} .
\end{aligned}$$

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 $0 \le q \le n - 1$. 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 $q = (n-1)/2$:

$$\begin{aligned}
f''(q) &= \frac{d}{dq}\left(\frac{\ln q - \ln(n - q - 1)}{\ln 2}\right) \\
&= \frac{1}{\ln 2}\left(\frac{1}{q} + \frac{1}{n - q - 1}\right)
\end{aligned}$$

and

$$
\begin{aligned}
f''\left(\frac{n-1}{2}\right) &= \frac{1}{\ln 2}\left(\frac{2}{n-1}+\frac{2}{n-1}\right) \\
&= \frac{1}{\ln 2}\cdot\frac{4}{n-1} \\
&> 0 \qquad \text{(since } n \geq 2\text{)} .
\end{aligned}
$$

Solution to Problem 7-2

a. If all elements are equal, then when PARTITION returns, $q = r$ and all elements in $A[p:q-1]$ are equal. We get the recurrence $T(n) = T(n-1) + \Theta(n)$ for the running time, and so $T(n) = \Theta(n^2)$.

b. The PARTITION$'$ procedure here chooses $A[p]$ as the pivot, instead of $A[r]$:

```
PARTITION′(A, p, r)
  x = A[p]
  i = p
  h = p
  for j = p + 1 to r
      // Invariant: A[p : i − 1] < x, A[i : h] = x,
          A[h + 1 : j − 1] > x, A[j : r] unknown.
      if A[j] < x
          y = A[j]
          A[j] = A[h + 1]
          A[h + 1] = A[i]
          A[i] = y
          i = i + 1
          h = h + 1
      elseif A[j] == x
          exchange A[h + 1] with A[j]
          h = h + 1
  return (i, h)
```

c. RANDOMIZED-PARTITION$'$ is the same as RANDOMIZED-PARTITION, but with the call to PARTITION replaced by a call to PARTITION$'$.

```
QUICKSORT′(A, p, r)
  if p < r
      (q, t) = RANDOMIZED-PARTITION′(A, p, r)
      QUICKSORT′(A, p, q − 1)
      QUICKSORT′(A, t + 1, r)
```

d. Putting elements equal to the pivot in the same partition as the pivot can only help, because QUICKSORT$'$ does not recurse on elements equal to the pivot. Thus, the subproblem sizes with QUICKSORT$'$, even with equal elements, are no larger than the subproblem sizes with QUICKSORT when all elements are distinct.

Solution to Problem 7-3

a. For any given element, RANDOMIZED-PARTITION has a $1/n$ probability of placing it into the pivot position, $A[r]$. Since we assume that the elements are distinct, the probability that the ith smallest element is chosen as the pivot is $1/n$. Therefore, $\Pr\{X_i\} = \mathrm{E}[X_i] = 1/n$.

b. If RANDOMIZED-PARTITION selects the qth smallest element as the pivot, then one recursive call of RANDOMIZED-QUICKSORT will be on a subarray of size $q-1$, and the other recursive call will be on a subarray of size $n-q$. RANDOMIZED-PARTITION takes $\Theta(n)$ time, no matter what. In the recursive case of RANDOMIZED-QUICKSORT, if RANDOMIZED-PARTITION returns the index of the qth smallest element, then the running time of RANDOMIZED-QUICKSORT is given by $T(q-1) + T(n-q) + \Theta(n)$.

The indicator random variable X_q equals 1 only if RANDOMIZED-PARTITION returns the index of the qth smallest element, and it equals 0 otherwise. Taking into account all possible values of q, we get that

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

Taking expectations of both sides gives

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right] \ .$$

c.

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right] \\
&= \sum_{q=1}^{n} \mathrm{E}[X_q(T(q-1) + T(n-q) + \Theta(n))] \\
&\qquad\qquad\qquad \text{(linearity of expectation)} \\
&= \sum_{q=1}^{n} \mathrm{E}[X_q]\,\mathrm{E}[(T(q-1) + T(n-q) + \Theta(n))] \\
&\qquad\qquad\qquad \text{(independence)} \\
&= \sum_{q=1}^{n} \frac{1}{n}\,\mathrm{E}[(T(q-1) + T(n-q) + \Theta(n))] \\
&= \frac{1}{n}\mathrm{E}\left[\sum_{q=1}^{n} T(q-1)\right] + \frac{1}{n}\mathrm{E}\left[\sum_{q=1}^{n} T(n-q)\right] + \frac{1}{n}\mathrm{E}\left[\sum_{q=1}^{n} \Theta(n)\right] \\
&\qquad\qquad\qquad \text{(linearity of expectation)} \\
&= \frac{1}{n}\mathrm{E}\left[\sum_{q=0}^{n-1} T(q)\right] + \frac{1}{n}\mathrm{E}\left[\sum_{q=0}^{n-1} T(q)\right] + \Theta(n) \qquad \text{(reindexing)} \\
&= \frac{2}{n}\mathrm{E}\left[\sum_{q=0}^{n-1} T(q)\right] + \Theta(n)
\end{aligned}$$

$$
\begin{aligned}
&= \frac{2}{n}\sum_{q=0}^{n-1} \mathrm{E}[T(q)] + \Theta(n) && \text{(linearity of expectation)} \\
&= \frac{2}{n}\sum_{q=1}^{n-1} \mathrm{E}[T(q)] + \Theta(n) && (T(0) = \Theta(1)) \ .
\end{aligned}
$$

d. Splitting the summation as given in the hint yields

$$
\begin{aligned}
\sum_{q=1}^{n-1} q \lg q &= \sum_{q=1}^{\lceil n/2\rceil -1} q \lg q + \sum_{q=\lceil n/2\rceil}^{n-1} q \lg q \\
&\le \lg(n/2) \sum_{q=1}^{\lceil n/2\rceil -1} q + \lg n \sum_{q=\lceil n/2\rceil}^{n-1} q \\
&= (\lg n - 1) \sum_{q=1}^{\lceil n/2\rceil -1} q + \lg n \sum_{q=\lceil n/2\rceil}^{n-1} q \\
&= \lg n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil n/2\rceil -1} q \\
&\le \frac{1}{2} n(n-1)\lg n - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2} \\
&\le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}
$$

if $n \ge 2$.

e. Assume that $\mathrm{E}[T(n)] \le an \lg n + b$ for some constants $a, b > 0$ that we get to choose. Choose b such that $\mathrm{E}[T(1)] \le b$, so that $\mathrm{E}[T(n)] \le an \lg n + b$ for $n = 1$. Then, for $n \ge 2$, by substitution we have

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \frac{2}{n}\sum_{q=1}^{n-1} \mathrm{E}[T(q)] + \Theta(n) \\
&\le \frac{2}{n}\sum_{q=1}^{n-1} (aq \lg q + b) + \Theta(n) \\
&= \frac{2a}{n}\sum_{q=1}^{n-1} q \lg q + \frac{2b(n-1)}{n} + \Theta(n) \\
&\le \frac{2a}{n}\left(\frac{n^2}{2}\lg n - \frac{n^2}{8}\right) + \frac{2b(n-1)}{n} + \Theta(n) \quad \text{(by part (d))} \\
&< an \lg n - \frac{an}{4} + 2b + \Theta(n) \\
&= an \lg n + b + \left(\Theta(n) + b - \frac{an}{4}\right) \\
&\le an \lg n + b \ ,
\end{aligned}
$$

since we can choose a large enough that $an/4$ dominates $\Theta(n) + b$. We conclude that $\mathrm{E}[T(n)] = O(n \lg n)$.

Solution to Problem 7-4

a. We first demonstrate that we always have $p \leq r$ in line 1, so that if line 2 executes, then the greater element moves to a higher index. In the initial call, $n \geq 1$, so that $p = 1 \leq n = r$. Now, consider any recursive call. Recursive calls occur only if line 3 finds that $p + 1 < r$, so that the subarray $A[p:r]$ has $r - p + 1 \geq 3$ elements. The value of k computed in line 4 is at least 1 and at most $(r - p + 1)/3$ and so we have that $p < r - k$ in lines 5 and 7 and that $p + k < r$ in line 6. Therefore, all three recursive calls have the second parameter strictly less than the third parameter.

Note that by computing k as the floor of $(r - p + 1)/3$, rather than the ceiling, if the size of the subarray $A[p:r]$ is not an exact multiple of 3, then the subarray sizes in the recursive calls are for subarrays of size $\lceil 2(r - p + 1)/3 \rceil$, so that lines 5–7 are calls on subarrays at least $2/3$ as large.

So now we need merely argue that sorting the first two-thirds, sorting the last two-thirds, and sorting the first two-thirds again suffices to sort the entire subarray. Denote the subarray size by n. Where can the $\lfloor n/3 \rfloor$ largest elements be after the first recursive call? They were either in the rightmost $\lfloor n/3 \rfloor$ positions, in which case they have not moved, or they were in the leftmost $\lceil 2n/3 \rceil$ positions, in which case they are in the rightmost positions within the leftmost $\lceil 2n/3 \rceil$. That is, they are in the middle $\lfloor n/3 \rfloor$ positions. The second recursive call guarantees that the largest $\lfloor n/3 \rfloor$ elements are in the rightmost $\lfloor n/3 \rfloor$ positions, and that they are sorted. All that remains is to sort the smallest $\lceil 2n/3 \rceil$ elements, which is taken care of by the third recursive call.

b. The recurrence is $T(n) = 3T(2n/3) + \Theta(1)$. We solve this recurrence by the master theorem with $a = 3$, $b = 3/2$, and $f(n) = \Theta(1) = \Theta(n^0)$. We need to determine $\log_{3/2} 3$, and it is approximately 2.71. Since $f(n) = O(n^{\log_{3/2} 3 - \epsilon})$ for $\epsilon = 2.7$, this recurrence falls into case 1, with the solution $T(n) = \Theta(n^{\log_{3/2} 3})$.

c. The running time of STOOGE-SORT is asymptotically greater than the worst-case running times of each of the four sorting methods in the question. No tenure for Professors Howard, Fine, and Howard. They should go back to teaching "Swingin' the Alphabet."[1]

Solution to Problem 7-5

a. TRE-QUICKSORT does exactly what QUICKSORT does, so that it sorts correctly.

QUICKSORT and TRE-QUICKSORT do the same partitioning, and then each calls itself with arguments $A, p, q - 1$. QUICKSORT then calls itself again, with

[1] And Curly's a dope.

arguments $A, q+1, r$. TRE-QUICKSORT instead sets $p = q+1$ 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 TRE-QUICKSORT will be $\Theta(n)$ on an n-element input array if there are $\Theta(n)$ recursive calls to TRE-QUICKSORT. This happens if every call to PARTITION(A, p, r) returns $q = r$. The sequence of recursive calls in this scenario is

$$
\begin{array}{l}
\text{TRE-QUICKSORT}(A, 1, n)\ , \\
\text{TRE-QUICKSORT}(A, 1, n-1)\ , \\
\text{TRE-QUICKSORT}(A, 1, n-2)\ , \\
\qquad\qquad\vdots \\
\text{TRE-QUICKSORT}(A, 1, 1)\ .
\end{array}
$$

Any array that is already sorted in increasing order will cause TRE-QUICKSORT to behave this way.

c. The problem demonstrated by the scenario in part (b) is that each invocation of TRE-QUICKSORT calls TRE-QUICKSORT again with almost the same range. To avoid such behavior, we must change TRE-QUICKSORT so that the recursive call is on a smaller interval of the array. The following variation of TRE-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).

```
TRE-QUICKSORT'(A, p, r)
  while $p < r$
      // Partition and sort the small subarray first.
      $q = $ PARTITION$(A, p, r)$
      if $q - p < r - q$
          TRE-QUICKSORT'$(A, p, q - 1)$
          $p = q + 1$
      else TRE-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

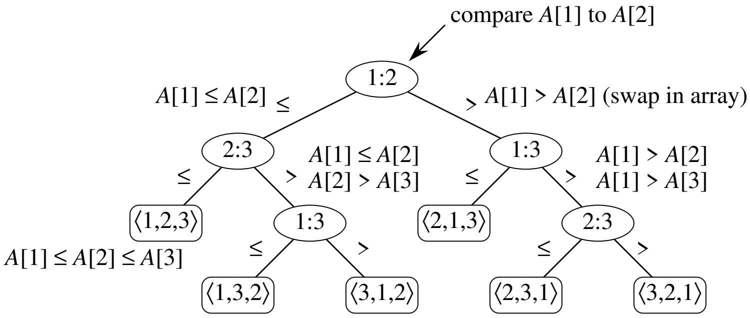
Lower bounds

- $\Omega(n)$ 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: $\Theta(n \lg n)$

Theorem
Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

Proof Let the decision tree have height h and l reachable leaves.

- $l \geq n!$
- $l \leq 2^h$ (see Section B.5.3: in a complete k-ary tree, there are k^h leaves)
- $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Take logarithms: $h \geq \lg(n!)$
- Use Stirling's approximation: $n! > (n/e)^n$ (by equation (3.23))

$$\begin{aligned} h &\geq \lg(n/e)^n \\ &= n \lg(n/e) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) . \end{aligned}$$

■ (theorem)

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.

Auxiliary storage: $C[0:k]$

```
COUNTING-SORT(A, n, k)
  let B[1:n] and C[0:k] be new arrays
  for i = 0 to k
      C[i] = 0
  for j = 1 to n
      C[A[j]] = C[A[j]] + 1
  // C[i] now contains the number of elements equal to i.
  for i = 1 to k
      C[i] = C[i] + C[i − 1]
  // C[i] now contains the number of elements less than or equal to i.
  // Copy A to B, starting from the end of A.
  for j = n downto 1
      B[C[A[j]]] = A[j]
      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
  return B
```

Do an example for $A = \langle 2_1, 5_1, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3 \rangle$. *[Subscripts show original order of equal keys in order to demonstrate stability.]*

i	0	1	2	3	4	5
$C[i]$ after second **for** loop	2	0	2	3	0	1
$C[i]$ after third **for** loop	2	2	4	7	7	8

Sorted output is $\langle 0_1, 0_2, 2_1, 2_2, 3_1, 3_2, 3_3, 5_1 \rangle$.

Idea: After the third **for** loop, $C[i]$ counts how many keys are less than or equal to i. If all elements are distinct, then an element with value i should go into $B[i]$. But if elements are not distinct, by examining values in A in reverse order, the last **for** loop puts $A[j]$ into $B[C[A[j]]]$ and then decrements $C[A[j]]$ so that the next time it finds element with the same value as $A[j]$, that element goes into the position of B just before $A[j]$.

Exercise 8.2-4 is to prove this loop invariant:

> **Loop invariant:** At the start of each iteration of the last **for** loop, the last element in A with value i that has not yet been copied into B belongs in $B[C[i]]$.

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

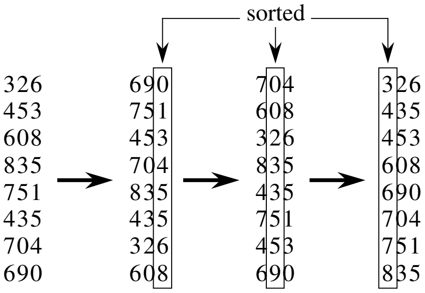
How IBM made its money. IBM made punch card readers for census tabulation in early 1900's. Card sorters worked on one column at a time. It's the algorithm for using the machine that extends the technique to multi-column sorting. The human operator was part of the algorithm!

Key idea: Sort *least* significant digits first.

To sort d digits:

```
RADIX-SORT(A, n, d)
  for i = 1 to d
      use a stable sort to sort array A[1 : n] 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 two digits in position i are different, ordering by position i is correct, and positions $1, \ldots, i - 1$ are irrelevant.

- If two digits in position i are equal, the 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 $= \Theta\left((b/r)(n + 2^r)\right)$.

How to choose r? Balance b/r and $n + 2^r$: decreasing r causes b/r to increase, but increasing r causes 2^r to increase.

If $b < \lfloor \lg n \rfloor$, then choose $r = b \Rightarrow (b/r)(n + 2^r) = \Theta(n)$, which is optimal.

If $b \geq \lfloor \lg n \rfloor$, then choosing $r \approx \lg n$ gives $\Theta\left((b/\lg n)(n + n)\right) = \Theta(bn/\lg n)$.

- Choosing $r < \lg n \Rightarrow b/r > b/\lg n$, and $n + 2^r$ term doesn't improve.
- Choosing $r > \lg n \Rightarrow n + 2^r$ 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 two keys.
- Used keys as array indices.

Bucket sort

Assumes that the input is generated by a random process that distributes elements uniformly and independently over $[0, 1)$.

Idea

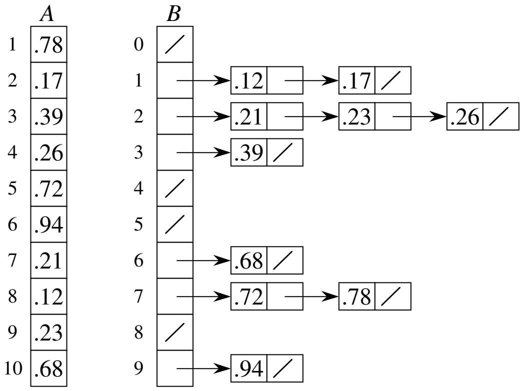
- Divide $[0, 1)$ into n equal-sized *buckets*.
- Distribute the n input values into the buckets. *[Can implement the buckets with linked lists; see Section 10.2.]*
- Sort each bucket.
- Then go through buckets in order, listing elements in each one.

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)
  let B[0 : n − 1] be a new array
  for i = 0 to n − 1
      make B[i] an empty list
  for i = 1 to n
      insert A[i] into list B[⌊n · A[i]⌋]
  for i = 0 to n − 1
      sort list B[i] with insertion sort
  concatenate lists B[0], B[1], ..., B[n − 1] together in order
  return the concatenated lists
```

Example



[The buckets are shown after each has been sorted. Slashes indicate the end of each bucket.]

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:

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}\left[n_i^2\right]) \quad (\mathrm{E}[aX] = a\mathrm{E}[X]) \end{aligned}$$

Claim

$\mathrm{E}[n_i^2] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim *[The proof of this claim is new in the fourth edition.]*

View each n_i as number of successes in n Bernoulli trials (see Section C.4).
Success occurs when an element goes into bucket $B[i]$.

- Probability p of success: $p = 1/n$.
- Probability q of failure: $q = 1 - 1/n$.

Binomial distribution counts number of successes in n trials: $\mathrm{E}[n_i] = np = n(1/n) = 1$ and $\mathrm{Var}[n_i] = npq = 1 - 1/n$ (see equations (C.41) and (C.44)). By equation (C.31):

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\ &= (1 - 1/n) + 1^2 \\ &= 2 - 1/n \end{aligned}$$

■ (claim)

Therefore:

$$\begin{aligned}
\mathrm{E}\left[T(n)\right] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}$$

- 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 $[0, 1)$, the algorithm is still correct, but might not run in $\Theta(n)$ time. It runs in linear time as long as the sum of squares of bucket sizes is $\Theta(n)$.

Solutions for Chapter 8: Sorting in Linear Time

Solution to Exercise 8.1-2

For either the upper bound or lower bound, start by observing that

$$\begin{aligned}
\lg(n!) &= \lg\left(\prod_{k=1}^{n} k\right) \\
&= \sum_{k=1}^{n} \lg k \; .
\end{aligned}$$

For the lower bound of $\Omega(n \lg n)$:

$$\begin{aligned}
\lg(n!) &= \sum_{k=1}^{n} \lg k \\
&\geq \sum_{k=\lceil n/2 \rceil}^{n} \lg k \\
&\geq \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lceil \frac{n}{2} \right\rceil \\
&\geq \left(\frac{n}{2} - 1\right) \lg \frac{n}{2} \\
&= \left(\frac{n}{2} - 1\right) (\lg n - 1) \\
&= \frac{n \lg n}{2} - \frac{n}{2} - \lg n + 1 \\
&= \Omega(n \lg n) \; .
\end{aligned}$$

For the upper bound of $O(n \lg n)$:

$$\begin{aligned}
\lg(n!) &= \sum_{k=1}^{n} \lg k \\
&\leq \sum_{k=1}^{n} \lg n \\
&= n \lg n \; .
\end{aligned}$$

Solution to Exercise 8.1-3
This solution is also posted publicly

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 values of m given here, $\lg m = \Omega(n \lg n)$, hence $h = \Omega(n \lg n)$.

In particular, using equation (3.25):

$$\begin{aligned}
\lg \frac{n!}{2} &= \lg n! - 1 \geq n \lg n - n \lg e - 1 \,, \\
\lg \frac{n!}{n} &= \lg n! - \lg n \geq n \lg n - n \lg e - \lg n \,, \\
\lg \frac{n!}{2^n} &= \lg n! - n \geq n \lg n - n \lg e - n \,.
\end{aligned}$$

Solution to Exercise 8.1-4

To get a permutation, place each of the $i \bmod 4 = 0$ elements; there are $3^{n/4}$ ways to do so. Now you can place each of the remaining $3n/4$ items in any order in the remaining places, so that there are $3^{n/4}(3n/4)!$ possible sorted orders and $3^{n/4}(3n/4)!$ leaves in the decision tree. The height of this decision tree is at least $\lg(3^{n/4}(3n/4)!)$, which is $\Omega(n \lg n)$.

Solution to Exercise 8.2-3
This solution is also posted publicly

[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 whether A is processed front to back or back to front.

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.

Rewrite of COUNTING-SORT that writes elements with the same value into the output array in order of increasing index and is stable:

```
COUNTING-SORT($A, n, k$)
  let $B[1:n]$, $C[0:k]$, and $L[0:k]$ be new arrays
  for $i = 0$ to $k$
      $C[i] = 0$
  for $j = 1$ to $n$
      $C[A[j]] = C[A[j]] + 1$
  // $C[i]$ now contains the number of elements equal to $i$.
  $L[0] = 1$
  for $i = 1$ to $k$
      $L[i] = L[i-1] + C[i-1]$
  // $L[i]$ now contains the index of the first element of $A$ with value $i$
  for $j = 1$ to $n$
      $B[L[A[j]]] = A[j]$
      $L[A[j]] = L[A[j]] + 1$
  return $B$
```

Solution to Exercise 8.2-4

> **Loop invariant:** At the start of each iteration of the **for** loop of lines 11–13, the last element in A with value i that has not yet been copied into B belongs in $B[C[i]]$.

Initialization: Initially, no elements in A have been copied into B, so that the last element in A with value i that has not been copied into B is just the last element in A with value i. Since there are $C[i]$ elements in A with value less than or equal to i, this last element in A with value i belongs in $B[C[i]]$.

Maintenance: Let $A[j] = i$ in a given iteration of the **for** loop of lines 11–13. By the loop invariant, $A[j]$ belongs in $B[C[i]]$. Let $m = \max\{l : l < j \text{ and } A[l] = i\}$ be the index of the rightmost element of A with value i that occurs before $A[j]$. Then $A[m]$ should go into the position of B immediately before where $A[j]$ goes, that is, $A[m]$ should go into position $C[i] - 1$. Decrementing $C[A[j]]$ in line 13 causes that to happen in the later iteration when $j = m$.

Termination: The loop terminates after n iterations. At that time, each element of A has been copied into its correct location in B.

Solution to Exercise 8.2-5

Count how many of each key there are and then just refill A with the correct number of each key.

```
COUNTING-SORT-KEYS-ONLY$(A, n, k)$
  let $C[0:k]$ be a new array
  for $i = 0$ to $k$
      $C[i] = 0$
  for $j = 1$ to $n$
      $C[A[j]] = C[A[j]] + 1$
  // $C[i]$ now contains the number of elements equal to $i$.
  $j = 1$
  for $i = 0$ to $k$
      for $l = 1$ to $C[i]$
          $A[j] = i$
          $j = j + 1$
  return $A$
```

Solution to Exercise 8.2-6

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 8.2-7

Modify the counting array C to be $C[0:10^d k + 1]$, initializing all entries to 0. Everywhere that $C[A[j]]$ appears in COUNTING-SORT, change it to $C[A[j] \cdot 10^d]$.

Solution to Exercise 8.3-2

Insertion sort is stable. When inserting $A[j]$ into the sorted sequence $A[1:j-1]$, the procedure compares $A[j]$ with each $A[i]$, starting with $i = j - 1$ and going down to $i = 1$. It continues 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 $\lg n + 1$ 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 each comparison takes an additional constant amount of time.

Solution to Exercise 8.3-3
This solution is also posted publicly

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

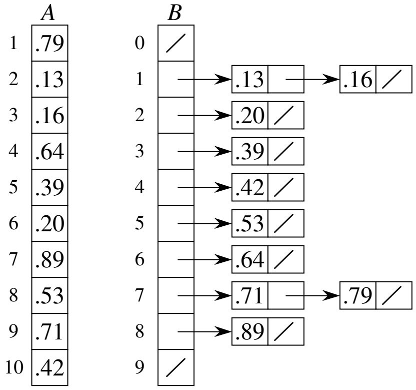
Do all the census in a single pass, storing the result in a 2D array indexed by which digit (i in RADIX-SORT) and digit value.

Solution to Exercise 8.3-5
This solution is also posted publicly

Treat the numbers as 3-digit numbers in radix n. Each digit ranges from 0 to $n - 1$. Sort these 3-digit numbers with radix sort.

There are 3 calls to counting sort, each taking $\Theta(n + n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

Solution to Exercise 8.4-1



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 $\Theta(n^2)$ time in the worst case.

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, such as merge sort or heapsort, instead of insertion sort when sorting the buckets.

Solution to Exercise 8.4-3

In two flips of a fair coin, $\Pr\{X = 0\} = \Pr\{X = 2\} = 1/4$ and $\Pr\{X = 1\} = 1/2$. Therefore, $\mathrm{E}[X] = (1/4 \cdot 0) + (1/2 \cdot 1) + (1/4 \cdot 2) = 1$, so that $\mathrm{E}^2[X] = 1$. However, $\mathrm{E}[X^2] = (1/4 \cdot 0) + (1/2 \cdot 1) + (1/4 \cdot 4) = 3/2$.

Solution to Exercise 8.4-5

Define $n + 1$ concentric disks with radius $\sqrt{i/n}$ for $i = 0, 1, \ldots, n$. Define ring i to contain the area inside the disk with radius $\sqrt{i/n}$ but not the area inside the disk with radius $\sqrt{(i-1)/n}$. Disk 0 and ring 0 are just the origin. Ring i has area $\pi(i/n - (i-1)/n) = \pi/n$. Because the area of ring i does not depend on i, each ring has the same area. Make $n + 1$ buckets, and put the points that fall into ring i into bucket i. Then use BUCKET-SORT.

Solution to Problem 8-1

This solution is also posted publicly

a. For a comparison algorithm A to sort, no two input permutations can reach the same leaf of the decision tree, so that 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. All of T's leaves must be 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,

$$\begin{aligned}
D(T) &= \sum_{x \in \text{leaves}(T)} d_T(x) \\
&= \sum_{x \in \text{leaves}(LT)} d_T(x) + \sum_{x \in \text{leaves}(RT)} d_T(x) \\
&= \sum_{x \in \text{leaves}(LT)} (d_{LT}(x) + 1) + \sum_{x \in \text{leaves}(RT)} (d_{RT}(x) + 1) \\
&= \sum_{x \in \text{leaves}(LT)} d_{LT}(x) + \sum_{x \in \text{leaves}(RT)} d_{RT}(x) + \sum_{x \in \text{leaves}(T)} 1 \\
&= D(LT) + D(RT) + k \ .
\end{aligned}$$

c. To show that $d(k) = \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$, we will show separately that $d(k) \le \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$ and $d(k) \ge \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$.

- We show that $d(k) \le \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$ by showing that $d(k) \le d(i)+d(k-i)+k$ for $i = 1, 2, \ldots, k-1$. By Exercise B.5-4, there are full binary trees with i leaves for any i from 1 to $k-1$. Therefore, we can create decision trees LT with i leaves and RT with $k-i$ leaves such that $D(LT) = d(i)$ and $D(RT) = d(k-i)$. Construct T such that LT and RT are the left and right subtrees of T's root, respectively. Then

$$\begin{aligned}
&d(k) \\
&\le D(T) && \text{(by definition of } d \text{ as minimum } D(T) \text{ value)} \\
&= D(LT) + D(RT) + k && \text{(by part (b))} \\
&= d(i) + d(k-i) + k && \text{(by choice of } LT \text{ and } RT) \ .
\end{aligned}$$

- We show that $d(k) \geq \min\{d(i) + d(k-i) + k : 1 \leq i \leq k-1\}$ by showing that $d(k) \geq d(i) + d(k-i) + k$, for some i in $\{1, 2, \ldots, k-1\}$. Take the tree T with k leaves such that $D(T) = d(k)$, let LT and RT be T's left and right subtree, respectively, and let i be the number of leaves in LT. Then $k - i$ is the number of leaves in RT and

  $$\begin{aligned} &d(k) \\ &= D(T) && \text{(by choice of } T\text{)} \\ &= D(LT) + D(RT) + k && \text{(by part (b))} \\ &\geq d(i) + d(k-i) + k && \text{(by definition of } d \text{ as minimum } D(T) \text{ value)} . \end{aligned}$$

  Neither i nor $k - i$ can be 0 (and hence $1 \leq i \leq k-1$), since if one of these were 0, either LT or RT would contain all k leaves of T. The root of T would have only one child, so that T would not be a full binary tree and hence not a decision tree.

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 that this is indeed a minimum (not a maximum), check that the second derivative of f_k is positive at $i = k/2$:

$$\begin{aligned} f_k''(i) &= \frac{d}{di}\left(\frac{\ln i - \ln(k-i)}{\ln 2}\right) \\ &= \frac{1}{\ln 2}\left(\frac{1}{i} + \frac{1}{k-i}\right) . \end{aligned}$$

$$\begin{aligned} f_k''(k/2) &= \frac{1}{\ln 2}\left(\frac{2}{k} + \frac{2}{k}\right) \\ &= \frac{1}{\ln 2} \cdot \frac{4}{k} \\ &> 0 \quad \text{(since } k > 1\text{)} . \end{aligned}$$

Now we use substitution to prove $d(k) = \Omega(kb \lg k)$. 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, assume that $d(i) \geq ci \lg i$ for $1 \leq i \leq k-1$, where c is some constant to be determined:

$$\begin{aligned} d(k) &= \min\{d(i) + d(k-i) + k : 1 \leq i \leq k-1\} \\ &\geq \min\{c(i \lg i + (k-i)\lg(k-i)) + k : 1 \leq i \leq k-1\} \\ &= c\left(\frac{k}{2}\lg\frac{k}{2} + \left(k - \frac{k}{2}\right)\lg\left(k - \frac{k}{2}\right)\right) + k \\ &= ck\lg\left(\frac{k}{2}\right) + k \\ &= c(k\lg k - k) + k \end{aligned}$$

$$\begin{aligned} &= ck \lg k + (k - ck) \\ &\geq ck \lg k \qquad \text{if } c \leq 1 , \end{aligned}$$

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 (one 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 average-case 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 8-2

a. COUNTING-SORT sorts records with keys 0 and 1 in $O(n)$ time and is stable.

b. To sort in $O(n)$ time in place, use a variation of the PARTITION method. Require that the pivot value be 1, and change the test in line 4 from "**if** $A[j] \leq x$" to ""**if** $A[j] < x$" so that all records with key 0 go left and all records with key 1 go right. To ensure that the pivot equals 1, if the record in position n of the array does not have a key of 1, perform a linear search for a record with key 0 and swap it with the record in position n. If no record has key 0, then the array is already sorted.

c. INSERTION-SORT is stable and sorts in place.

d. COUNTING-SORT can be used as the sorting method in RADIX-SORT so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time.

e. The procedure IN-PLACE-COUNTING-SORT uses $O(k)$ storage outside the input array and sortes in $O(n + k)$ time. It works because sorting is permuting, and any permutation is the union of disjoint cyclic permutations. The procedure must guard against processing an element that it has already moved within a cyclic permutation, as processing an element more than once can wreak havoc with the C array. The trick is to take advantage of the keys being 1 to k and negating each key when it is processed. When looking to start a cyclic permutation, the procedure checks the element's sign. If positive, it starts the cycle. If negative, it skips over the element. At the end, all elements have been processed and therefore have negative keys. One additional $\Theta(n)$-time pass over the elements restores the original positive key values. (If the keys were 0 to k, then instead of negating, the procedure could subtract $k + 1$ from each key to ensure negativity and then add back $k + 1$ at the end.)

```
IN-PLACE-COUNTING-SORT$(A, n, k)$
  let $C[1:k]$ be a new array
  for $i = 1$ to $k$
      $C[i] = 0$
  for $j = 1$ to $n$
      $C[A[j]] = C[A[j]] + 1$
  // $C[i]$ now contains the number of elements equal to $i$.
  for $i = 2$ to $k$
      $C[i] = C[i] + C[i-1]$
  // $C[i]$ now contains the number of elements less than or equal to $i$.
  for $j = n$ downto 1
      $x = A[j]$
      if $x > 0$                    // new element, start a cycle
          $pos = C[x]$
          while $pos \neq j$        // cycle continues until it comes back to $j$
              $y = A[pos]$          // next element in cycle
              $A[pos] = -x$         // put this element where it belongs, negated
              $C[x] = C[x] - 1$
              $x = y$               // prepare for next element in cycle
              $pos = C[x]$
          $A[pos] = -x$             // process last element in cycle
          $C[x] = C[x] - 1$
  for $j = 1$ to $n$
      $A[j] = |A[j]|$               // restore positive values
  return $A$
```

This algorithm is not stable. If the input is $\langle 2_1, 2_2, 2_3, 1_1 \rangle$, where subscripts show the order of equal keys in the input, this algorithm produces the output $\langle 1_1, 2_2, 2_3, 2_1 \rangle$.

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 $m \leq n$. 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 $m \leq n$.

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

$$\{(i, \pi(i)) : 1 \leq i \leq n \text{ and } \pi \text{ is a permutation on } \{1, 2, \ldots, n\}\} ,$$

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

$$3^h \geq n! \geq (n/e)^n \Rightarrow h \geq n \log_3 n - n \log_3 e = \Omega(n \lg n) .$$

Therefore, 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: $R \subseteq \{1, 2, \ldots, n\}$, representing red jugs, and $B \subseteq \{1, 2, \ldots, n\}$, 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
      return
  if $|R| == 1$                // sets contain just one jug each
      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 $r_1, r_2, \ldots, r_n$ and $b_1, b_2, \ldots, b_n$ where $r_i < r_{i+1}$ and $b_i < b_{i+1}$ for $i = 1, 2, \ldots, n$, 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 X of comparisons is given by

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

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 $R_>$, so that 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 expected number of comparisons being $O(n \lg n)$.

Just as 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 $T(n) = \Theta(n^2)$.

Solution to Problem 8-7

a. $A[q]$ must go the wrong place, because it goes where $A[p]$ should go. Since $A[p]$ is the smallest value in array A that goes to the wrong array location, $A[p]$ must be smaller than $A[q]$.

b. From how we have defined the array B, we have that if $A[i] \le A[j]$ then $B[i] \le B[j]$. Therefore, algorithm X performs the same sequence of exchanges on array B as it does on array A. The output produced on array A is of the form $\ldots A[q] \ldots A[p] \ldots$, and so the output produced on array B is of the form $\ldots B[q] \ldots B[p] \ldots$, or $\ldots 1 \ldots 0 \ldots$. Hence algorithm X fails to sort array B correctly.

c. The even steps perform fixed permutations. The odd steps sort each column by some sorting algorithm, which might not be an oblivious compare-exchange algorithm. But the result of sorting each column would be the same as if we did use an oblivious compare-exchange algorithm.

d. After step 1, each column has 0s on top and 1s on the bottom, with at most one transition between 0s and 1s, and it is a $0 \rightarrow 1$ transition. (As we read the array in column-major order, all $1 \rightarrow 0$ transitions occur between adjacent columns.) After step 2, therefore, each consecutive group of r/s rows, read in row-major order, has at most one transition, and again it is a $0 \rightarrow 1$ transition. All $1 \rightarrow 0$ transitions occur at the end of a group of r/s rows. Since there are s groups of r/s rows, there are at most s dirty rows, and the rest of the rows are clean. Step 3 moves the 0s to the top rows and the 1s to the bottom rows. The s dirty rows are somewhere in the middle.

e. The dirty area after step 3 is at most s rows high and s columns wide, and so its area is at most s^2. Step 4 turns the clean 0s in the top rows into a clean area on the left, the clean 1s in the bottom rows into a clean area on the right, and the dirty area of size s^2 is between the two clean areas.

f. First, we argue that if the dirty area after step 4 has size at most $r/2$, then steps 5–8 complete the sorting. If the dirty area has size at most $r/2$ (half a column), then it either resides entirely in one column or it resides in the bottom half of one column and the top half of the next column. In the former case, step 5 sorts the column containing the dirty area, and steps 6–8 maintain that the array is sorted. In the latter case, step 5 cannot increase the size of the dirty area, step 6 moves the entire dirty area into the same column, step 7 sorts it, and step 8 moves it back.

Second, we argue that the dirty area after step 4 has size at most $r/2$. But that follows immediately from the requirement that $r \geq 2s^2$ and the property that after step 4, the dirty area has size at most s^2.

g. If s does not divide r, then after step 2, we can see up to s $0 \to 1$ transitions and $s - 1$ $1 \to 0$ transitions in the rows. After step 3, we would have up to $2s - 1$ dirty rows, for a dirty area size of at most $2s^2 - s$. To push the correctness proof through, we need $2s^2 - s \leq r/2$, or $r \geq 4s^2 - 2s$.

h. We can reduce the number of transitions in the rows after step 2 back down to at most s by sorting every other column in reverse order in step 1. Now if we have a transition (either $1 \to 0$ or $0 \to 1$) between columns after step 1, then either one of the columns had all 1s or the other had all 0s, in which case we would not have a transition within one of the columns.

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 $1 \leq i \leq n$.

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.

Easy to solve the selection problem 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
      if min > A[i]
          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 suffice 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, 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, 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 $p == r$
      return $A[p]$        // $1 \leq i \leq r - p + 1$ when $p == r$ means that $i = 1$
  $q = $ RANDOMIZED-PARTITION($A, p, r$)
  $k = q - p + 1$
  if $i == k$
      return $A[q]$        // the pivot value is the answer
  elseif $i < k$
      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 one 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.

Analysis assumes that the recursion goes as deep as possible: until only one element remains.

Intuition: Suppose that each pivot is in the second or third quartiles if the elements were sorted—in the "middle half." Then at least $1/4$ of the remaining elements are ignored in all future recursive calls $\Rightarrow$ at most $3/4$ of the elements are still ***in play***: somewhere within $A[p:r]$. RANDOMIZE-PARTITION takes $\Theta(n)$ time to partition n elements $\Rightarrow$ recurrence would be $T(n) = T(3n/4) + \Theta(n) = \Theta(n)$ by case 3 of the master method.

What if the pivot is not always in the middle half? Probability that it is in the middle half is $1/2$. View selecting a pivot in the middle half as a Bernoulli trial with probability of success $1/2$. Then the number of trials before a success is a geometric distribution with expected value 2. So that half the time, $1/4$ of the elements go out of play, and the other half of the time, as few as one element (the pivot) goes out of play. But that just doubles the running time, so still expect $\Theta(n)$.

Rigorous analysis:

- Define $A^{(j)}$ as the set of elements still in play (within $A[p:r]$) after j recursive calls (i.e., after jth partitioning). $A^{(0)}$ is all the elements in A.
- $|A^{(j)}|$ is a random variable that depends on A and order statistic i, but not on the order of elements in A.
- Each partitioning removes at least one element (the pivot) $\Rightarrow$ sizes of $A^{(j)}$ strictly decrease.
- jth partitioning takes set $A^{(j-1)}$ and produces $A^{(j)}$.
- Assume a 0th "dummy" partitioning that produces $A^{(0)}$.
- jth partitioning is ***helpful*** if $|A^{(j)}| \le (3/4)|A^{(j-1)}|$. Not all partitionings are necessarily helpful. Think of a helpful partitioning as a successful Bernoulli trial.

Lemma

A partitioning is helpful with probability $\ge 1/2$.

Proof

- Whether or not a partitioning is helpful depends on the randomly chosen pivot.
- Define "middle half" of an n-element subarray as all but the smallest $\lceil n/4 \rceil - 1$ and greatest $\lceil n/4 \rceil - 1$ elements. That is, all but the first and last $\lceil n/4 \rceil - 1$ if the subarray were sorted.

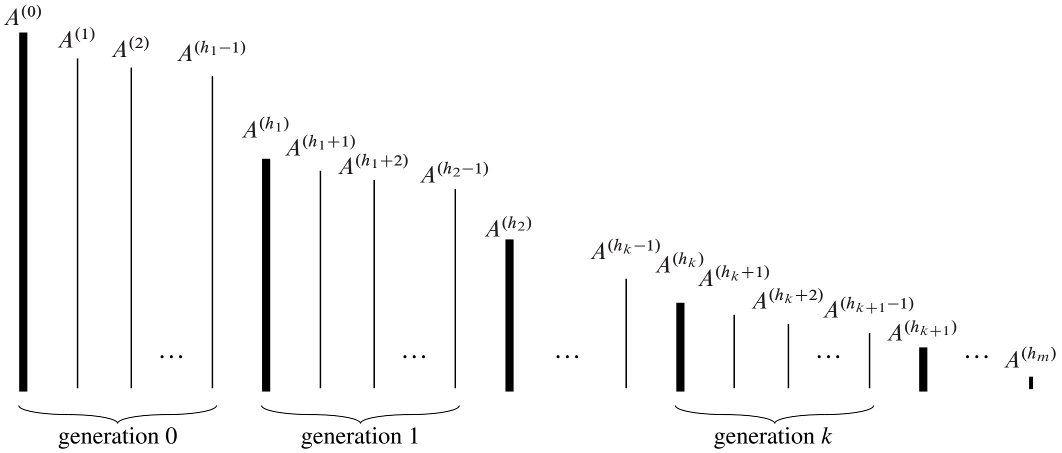
- Will show that if the pivot is in the middle half, then that pivot leads to a helpful partitioning and that the probability that the pivot is in the middle half is $\geq 1/2$.
- No matter where the pivot lies, either all elements $>$ pivot or all elements $<$ pivot, and the pivot itself, are not in play after partitioning $\Rightarrow$ if the pivot is in the middle half, at least the smallest $\lceil n/4 \rceil - 1$ or greatest $\lceil n/4 \rceil - 1$ elements, plus the pivot, will not be in play after partitioning $\Rightarrow \geq \lceil n/4 \rceil$ elements not in play.
- Then, at most $n - \lceil n/4 \rceil = \lfloor 3n/4 \rfloor < 3n/4$ elements in play $\Rightarrow$ partitioning is helpful. ($n - \lceil n/4 \rceil = \lfloor 3n/4 \rfloor$ is from Exercise 3.3-2.)
- To find a lower bound on the probability that a randomly chosen pivot is in the middle half, find an upper bound on the probability that it is not:
$$\begin{aligned} \frac{2(\lceil n/4 \rceil - 1)}{n} &\leq \frac{2((n/4+1)-1)}{n} \quad \text{(inequailty (3.2))} \\ &= \frac{n/2}{n} \\ &= 1/2 \,. \end{aligned}$$
- Since the pivot has probability $\geq 1/2$ of falling into the middle half, a partitioning is helpful with probability $\geq 1/2$. ■ (lemma)

Theorem

The expected running time of RANDOMIZED-SELECT is $\Theta(n)$.

Proof

- Let the sequence of helpful partitionings be $\langle h_0, h_1, \ldots, h_m \rangle$. Consider the 0th partitioning as helpful $\Rightarrow h_0 = 0$. Can bound m, since after at most $\lceil \log_{4/3} n \rceil$ helpful partitionings, only one element remains in play.
- Define $n_k = |A^{(h_k)}|$ and $n_0 = |A^{(0)}|$, the original problem size. $n_k = |A^{(h_k)}| \leq (3/4)|A^{(h_k - 1)}| = (3/4)\, n_{k-1}$ for $k = 1, 2, \ldots, m$.
- Iterating gives $n_k \leq (3/4)^k n_0$.
- Break up sets into m "generations." The sets in generation k are $A^{(h_k)}$, $A^{(h_k+1)}$, $\ldots$, $A^{(h_{k+1}-1)}$, where $A^{(h_k)}$ is the result of a helpful partitioning and $A^{(h_{k+1}-1)}$ is the last set before the next helpful partitioning.



[Height of each line indicates the set of the set (number of elements in play). Heavy lines are sets $A^{(h_k)}$, resulting from helpful partitionings and are first within their generation. Other lines are not first within their generation. A generation may contain just one set.]

- If $A^{(j)}$ is in the kth generation, then $|A^{(j)}| \le |A^{(h_k)}| = n_k \le (3/4)^k n_0$.
- Define random variable $X_k = h_{k+1} - h_k$ as the number of sets in the kth generation $\Rightarrow$ kth generation includes sets $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_k+X_k-1)}$.
- By previous lemma, a partitioning is helpful with probability $\ge 1/2$. The probability is even higher, since a partitioning is helpful even if the pivot doesn't fall into middle half, but the ith smallest element lies in the smaller side. Just use the $1/2$ lower bound $\Rightarrow \mathrm{E}[X_k] \le 2$ for $k = 0, 1, \ldots, m-1$ (by equation (C.36), expectation of a geometric distribution).
- The total running time is dominated by the comparisons during partitioning. The jth partitioning takes $A^{(j-1)}$ and compares the pivot with all the other $|A^{(j-1)}| - 1$ elements $\Rightarrow$ jth partitioning makes $< |A^{(j-1)}|$ comparisons.
- The total number of comparisons is less than

$$\begin{aligned}
\sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}| &\le \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(h_k)}| \\
&= \sum_{k=0}^{m-1} X_k \, |A^{(h_k)}| \\
&\le \sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0 \ .
\end{aligned}$$

- Since $\mathrm{E}[X_k] \le 2$, the expected total number of comparisons is less than

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0\right] &= \sum_{k=0}^{m-1} \mathrm{E}\left[X_k \left(\frac{3}{4}\right)^k n_0\right] && \text{(linearity of expectation)} \\
&= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \mathrm{E}[X_k] \\
&\le 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\
&< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 8n_0 && \text{(infinite geometric series)} \ .
\end{aligned}$$

- n_0 is the size of the original array A $\Rightarrow$ an $O(n)$ upper bound on the expected running time. For the lower bound, the first call of RANDOMIZED-PARTITION examines all n elements $\Rightarrow \Theta(n)$. ■ (theorem)

Therefore, we can determine any order statistic in linear time on average, assuming that all elements are distinct.

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. It's not terribly practical—primarily of theoretical interest.

Idea: Like RANDOMIZED-SELECT, recursively partition the input array. But instead of picking a pivot randomly, guarantee a good split by picking a provably good pivot. How? Recursively!

SELECT uses a simple variant of the PARTITION algorithm that takes as an additional parameter the value of the pivot. Call it PARTITION-AROUND.

Input to SELECT is the same as for RANDOMIZED-SELECT.

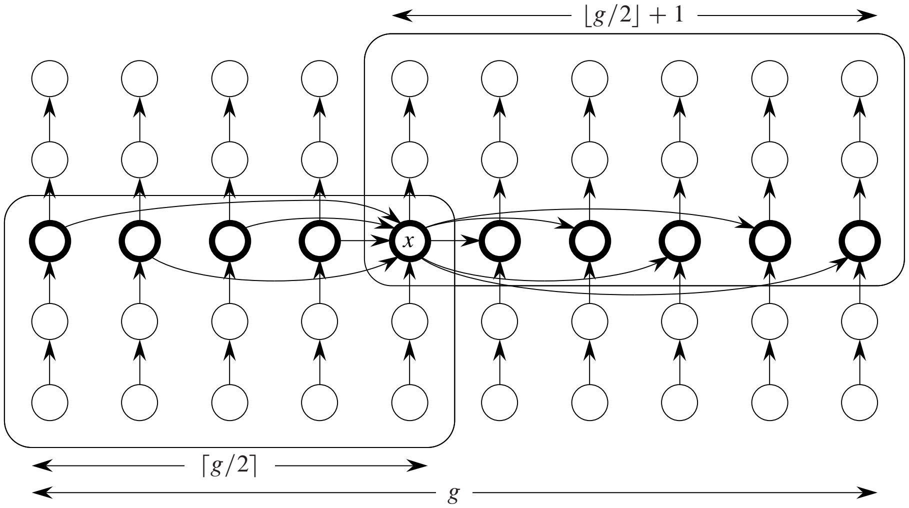
```
SELECT$(A, p, r, i)$
  while $(r - p + 1) \bmod 5 \neq 0$
      for $j = p + 1$ to $r$                    // put the minimum into $A[p]$
          if $A[p] > A[j]$
              exchange $A[p]$ with $A[j]$
      // If we want the minimum of $A[p:r]$, we're done.
      if $i == 1$
          return $A[p]$
      // Otherwise, we want the $(i-1)$st element of $A[p+1:r]$.
      $p = p + 1$
      $i = i - 1$
  $g = (r - p + 1)/5$                            // the number of 5-element groups
  for $j = p$ to $p + g - 1$                      // sort each group
      sort $\langle A[j], A[j+g], A[j+2g], A[j+3g], A[j+4g] \rangle$ in place
  // All group medians now lie in the middle fifth of $A[p:r]$.
  // Find the pivot $x$ recursively as the median of the group medians.
  $x =$ SELECT$(A, p+2g, p+3g-1, \lceil g/2 \rceil)$
  $q =$ PARTITION-AROUND$(A, p, r, x)$   // partition around the pivot
  // The rest is just like the end of RANDOMIZED-SELECT.
  $k = q - p + 1$
  if $i == k$
      return $A[q]$                              // the pivot value is the answer
  elseif $i < k$
      return SELECT$(A, p, q-1, i)$
  else return SELECT$(A, q+1, r, i-k)$
```

The algorithm works with groups of 5 elements. If n is not a multiple of 5, the beginning **while** loop removes n mod 5 elements from consideration. If $i \leq n$ mod 5, then the ith iteration of the **while** loop identifes and returns the ith smallest element. Each iteration puts the smallest element into $A[p]$. If $i = 1$, then done. Otherwise, increment p, so that $A[p]$ is no longer in play, and decrement i, so that this minimum element doesn't matter any longer.

After the **while** loop, size of the subarray $A[p:r]$ is divisible by 5. From there:

- Compute $g = (r - p + 1)/5 =$ the number of groups of 5 elements.

- Compose the groups of 5 elements by taking every 5th one:
  - First group is $\langle A[p], A[p+g], A[p+2g], A[p+3g], A[3+4g] \rangle$.
  - Second group is $\langle A[p+1], A[p+g+1], A[p+2g+1], A[p+3g+1], A[p+4g+1] \rangle$.
  - And so on. Last group ($(g-1)$st group) is $\langle A[p+g-1], A[p+2g-1], A[p+3g-1], A[p+4g-1], A[r] \rangle$. ($r = p+5g-1$.)



[Each column is a group of 5 elements. Arrows point from smaller elements to larger elements. The group medians are the middle fifth of the array, shown in the figure with heavy outlines in the middle row.]

- Sort each group of 5 to find its median.
- Recursively call SELECT on the medians to find the median of the group medians. That value will be the pivot x. In the middle row, all medians to the left of x are $\leq x$, and all medians to the right of x are $\geq x$. We don't know the ordering of the medians to the left of x relative to each other, and same for to the right of x.
- From there, it's the same as RANDOMIZED-SELECT:
  - Partition the entire subarray $A[p:r]$ around x. The call of PARTITION-AROUND returns the index q of where the pivot x ends up. Compute the relative index k of q within the subarray $A[p:r]$.
  - If $i = k$, done. $A[q]$ is the ith smallest element in $A[p:r]$.
  - Otherwise, recurse on either the elements preceding $A[q]$ or the elements following $A[q]$. In the latter case, want the $(i-k)$th smallest element.

Analysis

Will show that SELECT runs in worst-case time $\Theta(n)$.

The lower bound of $\Omega(n)$ comes from each iteration of the **while** loop and also sorting the g groups of 5 elements. (Note: $g \geq (n-4)/5$.)

For the upper bound of $O(n)$:

- Define $T(n)$ as the worst-case time for SELECT on a subarray of size at most n. $T(n)$ monotonically increases.
- The **while** loop executes at most 4 times, each iteration taking $O(n)$ time $\Rightarrow$ $O(n)$ time for the **while** loop.
- Sorting the g groups of 5 takes $O(n)$ time because sorting 5 elements takes constant time (even using insertion sort) and $g \le n/5$.
- Time for PARTITION-AROUND to partition around the pivot x is $\Theta(n)$.
- The code contains three recursive calls, of which at most two execute. The first recursive call to find the median of the medians always executes, taking time $T(g) \le T(n/5)$. At most one of the two other recursive calls executes.
- Claim: Whichever of the latter two calls of SELECT executes, it is on a subarray of at most $7n/10$ elements.
- Proof of claim: *[Refer to the previous figure.]*
  - There are $g \le n/5$ groups of 5 elements, each group shown as a column, sorted bottom to top. Arrows go from smaller to larger elements.
  - Groups are ordered left to right, with all groups to the left of the pivot x having a median smaller than x and all groups to the right of x having a median greater than x.
  - The upper-right region contains elements known to be $\ge x$. The lower-left region contains elements known to be $\le x$. Pivot x is in both regions.
  - The upper-right region contains $\lfloor g/2 \rfloor + 1$ groups $\Rightarrow$ as least $3(\lfloor g/2 \rfloor + 1) \ge 3g/2$ elements are $\ge x$.
  - The lower-left region contains $\lceil g/2 \rceil$ groups $\Rightarrow$ at least $3\lceil g/2 \rceil$ elements are $\le x$.
  - Either way, the recursive call excludes $\ge 3g/2$ elements, leaving at most $5g - 3g/2 = 7g/2 \le 7n/10$ elements.
- Get the recurrence $T(n) \le T(n/5) + T(7n/10) + \Theta(n)$.
- Prove that $T(n) \le cn$ by substitution for suitably large constant c.
- Assuming that $n \ge 5$ gives

  $$\begin{aligned} T(n) &\le c(n/5) + c(7n/10) + \Theta(n) \\ &\le 9cn/10 + \Theta(n) \\ &= cn - cn/10 + \Theta(n) \\ &\le cn \end{aligned}$$

  if $cn/10$ dominates the constant in the $\Theta(n)$ term. Also need to pick c large enough so that $T(n) \le cn$ for $n \le 4$ (base case).
- Therefore, $T(n) = O(n)$. Conclude that $T(n) = \Theta(n)$.

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 that 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 9.1-2

It takes 3 comparisons. Let the numbers be $x_1, x_2, \ldots, x_n$. If $n = 3$, compare x_1 with x_2, then compare the smaller one with x_3. If x_3 is less than the smaller number from the first comparison, that smaller number from the first comparison is the middle one. Otherwise, compare x_3 with the greater number from the first comparison, and the smaller number is the middle one. If $n \geq 4$, compare x_1 with x_2, then compare x_3 with x_4, and then compare the two numbers found to be smaller in the comparisons. The larger of the two is neither the minimum nor the maximum.

Solution to Exercise 9.2-1

For RANDOMIZED-SELECT to make a recursive call on a 0-length array, either $p = q$ in line 8 or $q = r$ in line 9. In the former case, line 4 computes $k = 1$, so that i would have to be less than 1. In the latter case, line 4 computes $k = r - p + 1 = n$, so that for $i - k$ to be positive, i would have to be greater than n.

Solution to Exercise 9.2-2

```
ITERATIVE-RANDOMIZED-SELECT($A, p, r, i$)
 while $p < r$
      $q$ = RANDOMIZED-PARTITION($A, p, r$)
      $k = q - p + 1$
      if $i == k$
           return $A[q]$
      elseif $i < k$
           $r = q - 1$
      else $p = q + 1$
           $i = i - k$
 return $A[p]$
```

Solution to Exercise 9.2-4

As suggested, we provide an argument by induction on the length of the input array. The base cases ($n \leq 1$) are trivial.

We first observe that for a given set of n input elements, the randomly chosen pivot element x is equally likely to be any one of the n input elements. This is true by construction, independent of the order in which the input elements appear in the input array $A[p:r]$. The running time for choosing x is constant, independent of the input order.

Second, the algorithm partitions the input array into three parts:

- The set L containing all elements less than x.
- The set $\{x\}$ containing just the pivot element x.
- The set G containing all elements greater than x.

The running time of the partition routine is a fixed linear function of n, independent of the input order. The sizes of the sets L and G do not depend on the input order, but only on the number of input elements that are respectively smaller than or greater than x.

The algorithm then either stops (if x is the desired answer), or recurses on L or G (as stored in the array now as $A[p:q-1]$ or as $A[q+1:r]$). The decision as to whether to stop or to recurse (and on which side to recurse) does not depend on the input order, but only on the sizes of L and G and on the input i saying which element is to be selected. (Otherwise, changing the input order could change the answer.)

By induction, then, for any given input array $A[p:r]$, RANDOMIZED-SELECT has the same probability (independent of the input order) for any sequence of choices of pivot elements and recursive calls on subarrays. Thus, the expected running time is unchanged by a change of the order of the input elements.

Solution to Exercise 9.3-1
This solution is also posted publicly

For groups of 7, the algorithm still works in linear time. The number g of groups is at most $n/7$. There are at least $4(\lfloor g/2 \rfloor + 1) \geq 2g$ elements greater than or equal to the pivot, and at least $4\lceil g/2 \rceil \geq 2g$ elements less than or equal to the pivot. That leaves at most $7g - 2g = 5g \leq 5n/7$ elements in the recursive call. The recurrence becomes $T(n) \leq T(n/7) + T(5n/7) + O(n)$, which you can show by substitution has the solution $T(n) = O(n)$.

[In fact, any odd group size ≥ 5 works in linear time.]

Solution to Exercise 9.3-3
This solution is also posted publicly

A modification to quicksort that allows it to run in $O(n \lg n)$ time in the worst case uses the deterministic PARTITION-AROUND procedure that takes 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 $p < r$
      $i = \lfloor (r - p + 1)/2 \rfloor$
      $x =$ SELECT($A, p, r, i$)
      $q =$ PARTITION-AROUND($A, p, r, x$)
      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 9.3-6
This solution is also posted publicly

Let the procedure MEDIAN take as parameters an array A and subarray indices p and r and return 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-AROUND procedure that takes an element to partition around as an input parameter.

```
SELECT′(A, p, r, i)
  if p == r
      return A[p]
  x = MEDIAN(A, p, r)
  q = PARTITION-AROUND(A, p, r, x)
  k = q − p + 1
  if i == k
      return A[q]
  elseif i < k
      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 subarray $A[p:q-1]$ 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 9.3-7

The main pipeline should have a y-coordinate that is the median of the y-coordinates of the n wells. That is, the number of wells north of the pipeline should equal the number of wells south of the pipeline. If n is odd, then the pipeline should run through some well with the median y-coordinate. If n is even, the pipeline should run between the two middle wells or through one of them.

To see why, let m be the median y-coordinate, and let N and S be the sets of wells north and south of m, respectively. If n is even and m runs through a well, consider that well to be in N if it's the more northern of the two middle wells and in S if it's the more southern. Because m is the median y-coordinate, we have $|N| = |S|$. For each well w, denote its y-coordinate by w_y. Since wells in N have y-coordinates at least m and wells in S have y-coordinates at most m, the total length l of the spurs when the main pipeline has y-coordinate m is given by

$$l = \sum_{w \in N} (w_y - m) + \sum_{w \in S} (m - w_y) \; .$$

Now let $m' > m$ be the y-coordinate of some other location for the main pipeline for which some nonempty subset $N' \subseteq N$ of wells is south of m' and the remaining wells in $N - N'$ are all north of m'. We can rewrite the formula for l as

$$\begin{aligned} l &= \sum_{w \in N} (w_y - m) + \sum_{w \in S} (m - w_y) \\ &= \sum_{w \in N - N'} (w_y - m) + \sum_{w \in N'} (w_y - m) + \sum_{w \in S} (m - w_y) \\ &= \sum_{w \in N - N'} w_y - \sum_{w \in S} w_y + (|S| - (|N - N'| + |N'|))m + \sum_{w \in N'} w_y \\ &= \sum_{w \in N - N'} w_y - \sum_{w \in S} w_y + \sum_{w \in N'} w_y \end{aligned}$$

because $|N - N'| + |N'| = |N| = |S|$.

Now suppose that the main pipeline moves to y-coordinate m' so that the spur length for each well $w \in N'$ equals $m' - w_y > 0$. For each well $w \in S$, we have $m' - w_y > m - w_y > 0$, and for each well $w \in N - N'$, we have $w_y - m' > 0$. The total length l' of the spurs with the main pipeline at y-coordinate m' is given by

$$\begin{aligned} l' &= \sum_{w \in N - N'} (w_y - m') + \sum_{w \in N'} (m' - w_y) + \sum_{w \in S} (m' - w_y) \\ &= \sum_{w \in N - N'} w_y - \sum_{w \in S} w_y + (|S| + |N'| - |N - N'|)m' - \sum_{w \in N'} w_y \\ &= \sum_{w \in N - N'} w_y - \sum_{w \in S} w_y + 2\,|N'|\,m' - \sum_{w \in N'} w_y \\ &> \sum_{w \in N - N'} w_y - \sum_{w \in S} w_y + 2 \sum_{w \in N'} w_y - \sum_{w \in N'} w_y \quad (m' > w_y \text{ for all } w_y \in N') \\ &= \sum_{w \in N - N'} w_y - \sum_{w \in S} w_y + \sum_{w \in N'} w_y \\ &= l \; . \end{aligned}$$

Thus, by moving the y-coordinate of the main pipeline up so that some wells that were north of the main pipeline become south of it, the total spur length increases. A symmetric argument shows that moving the main pipeline south so that some wells that were south of it become north of it also increases the total spur length. Hence, the median of the y-coordinates minimizes the total spur length.

To find the optimal location in linear time, use SELECT on the y-coordinates of the n wells.

Solution to Exercise 9.3-8

To find the k quantiles of an n-element set, take advantage of the SELECT procedure's property that it partitions the elements into those less than and those greater than the order statistic it finds.

- If $k = 1$, then there are no quantiles, so just return.
- If k is even, find the $(k/2)$th order statistic using the SELECT procedure. Then recursively find the $k/2$ quantiles of the elements less than and of the elements greater than the $(k/2)$th order statistic.
- If k is odd, find the $\lfloor k/2 \rfloor$th order statistic using the SELECT procedure. Then recursively find the $\lfloor k/2 \rfloor$ quantiles of the elements less than the $(k/2)$th order statistic and the $\lceil k/2 \rceil$ quantiles of the elements greater than the $(k/2)$th order statistic.

The recurrence for the running time is $T(n, k) \le T(n/2, \lceil k/2 \rceil) + O(n)$. Assuming that $k < n$, this recurrence reaches the base case after $O(\lg k)$ levels of recursion, and each level of recursion totals to $O(n)$ time, making the total running time $O(n \lg k)$.

Solution to Exercise 9.3-9

First, use the SELECT procedure to find the median x. Then create a new array A with values $\{|y - x| : y \in S\}$, and maintain a correspondence between values in A and elements in S. Use SELECT to find the kth smallest element z in A. Each value in A that is less than or equal to z corresponds to one of the k closest elements in S to the median.

Solution to Exercise 9.3-10

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($X, Y, n$)
  $median$ = FIND-MEDIAN($X, Y, n, 1, n$)
  if $median$ == NOT-FOUND
      $median$ = FIND-MEDIAN($Y, X, n, 1, n$)
  return $median$

FIND-MEDIAN($A, B, n, low, high$)
  if $low > high$
      return NOT-FOUND
  else $k = \lfloor (low + high)/2 \rfloor$
      if $k == n$ and $A[n] \le B[1]$
          return $A[n]$
      elseif $k < n$ and $B[n-k] \le A[k] \le B[n-k+1]$
          return $A[k]$
      elseif $A[k] > B[n-k+1]$
          return FIND-MEDIAN($A, B, n, low, k-1$)
      else return FIND-MEDIAN($A, B, n, k+1, high$)
```

Solution to Problem 9-1

This solution is also posted publicly

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 $\geq n/2$ 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).

Solution to Problem 9-2

a. If RANDOMIZED-PARTITION chooses the greatest element as the pivot, then it returns $q = r$, and the recursive call in line 6 has the parameters A, p, r, i, which are the same parameters as in the original call. In the worst case, therefore, RANDOMIZED-PARTITION always chooses the greatest element as the pivot, and the procedure infinitely recurses.

b. The same technique as in Section 9.2 works here. The difference is that it is possible that $|A_j| = |A_{j-1}|$, but the probability of this occuring is only $1/n$ in each call of RANDOMIZED-PARTITION. We can lower the probability that a partition is helpful from $1/2$ to, say, $1/3$ to accommodate the possibility of the greatest element being chosen as the pivot. That would just change $\mathrm{E}[X_k]$ from 2 to 3, so that upper bound on the expected number of comparisons goes from $8n$ to $12n$, which is still $O(n)$.

Solution to Problem 9-3

a. The median x of the elements $x_1, x_2, \ldots, x_n$, is an element $x = x_k$ satisfying $|\{x_i : 1 \leq i \leq n \text{ and } x_i < x\}| \leq n/2$ and $|\{x_i : 1 \leq i \leq n \text{ and } x_i > x\}| \leq n/2$. If each element x_i is assigned a weight $w_i = 1/n$, then we get

$$
\begin{aligned}
\sum_{x_i < x} w_i &= \sum_{x_i < x} \frac{1}{n} \\
&= \frac{1}{n} \cdot \sum_{x_i < x} 1 \\
&= \frac{1}{n} \cdot |\{x_i : 1 \leq i \leq n \text{ and } x_i < x\}| \\
&\leq \frac{1}{n} \cdot \frac{n}{2} \\
&= \frac{1}{2} ,
\end{aligned}
$$

and

$$\begin{aligned}\sum_{x_i > x} w_i &= \sum_{x_i > x} \frac{1}{n} \\ &= \frac{1}{n} \cdot \sum_{x_i > x} 1 \\ &= \frac{1}{n} \cdot |\{x_i : 1 \le i \le n \text{ and } x_i > x\}| \\ &\le \frac{1}{n} \cdot \frac{n}{2} \\ &= \frac{1}{2} ,\end{aligned}$$

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. First, sort the n elements into increasing order by x_i values. Then, scan the array of sorted x_i's, starting with the smallest element and accumulating weights as you 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 elements up to and including x_k 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. To find the weighted median in $\Theta(n)$ worst-case time, use 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 $n \le 2$, just return the brute-force solution. Otherwise, proceed as follows. Find the actual median x_k of the n elements and then partition around it. 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
    return x_1
elseif n == 2
    if $w_1 \geq w_2$
        return $x_1$
    else return $x_2$
else find the median $x_k$ of $X = \{x_1, x_2, \ldots, x_n\}$
    partition the set $X$ around $x_k$
    compute $W_L = \sum_{x_i < x_k} w_i$ and $W_G = \sum_{x_i > x_k} w_i$
    if $W_L < 1/2$ and $W_G < 1/2$
        return $x_k$
    elseif $W_L > 1/2$
        $w_k = w_k + W_G$
        $X' = \{x_i \in X : x_i \leq x_k\}$
        return WEIGHTED-MEDIAN($X'$)
    else $w_k = w_k + W_L$
        $X' = \{x_i \in X : x_i \geq x_k\}$
        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 $x \ge x_i$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^{n} w_i(|y - x_i| - |x - x_i|) \\ &\ge \sum_{x<x_i} w_i(x - y) + \sum_{x \ge x_i} w_i(y - x) \\ &= (y - x)\left(\sum_{x \ge x_i} w_i - \sum_{x<x_i} w_i\right) . \end{aligned}$$

The property that $\sum_{x_i<x} w_i < 1/2$ implies that $\sum_{x \ge x_i} w_i \ge 1/2$. This fact, combined with $y - x > 0$, yields that $f(y) - f(x) \ge 0$. In the third case, where $x \ge x_i$ and $|y - x_i| - |x - x_i| = |y - x| = y - x > 0$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^{n} w_i(|y - x_i| - |x - x_i|) \\ &= \sum_{i=1}^{n} w_i(y - x) \\ &\ge 0 . \end{aligned}$$

When $x > y$, we again bound the quantity $|y - x_i| - |x - x_i|$ from below by examining three cases:

1. $x_i \le y < x$: 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. $y \le x_i < x$: Here, $|y - x_i| \ge 0$ and $|x - x_i| \le x - y$, which imply that $|y - x_i| - |x - x_i| \ge -(x - y) = y - x$.
3. $y < x \le x_i$. 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 $x \le x_i$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^{n} w_i(|y - x_i| - |x - x_i|) \\ &\ge \sum_{x>x_i} w_i(y - x) + \sum_{x \le x_i} w_i(x - y) \\ &= (x - y)\left(\sum_{x \le x_i} w_i - \sum_{x>x_i} w_i\right) . \end{aligned}$$

The property that $\sum_{x_i>x} w_i \le 1/2$ implies that $\sum_{x \le x_i} w_i \ge 1/2$. This fact, combined with $x - y > 0$, yields that $f(y) - f(x) > 0$. In the third case, where $x \le x_i$ and $|y - x_i| - |x - x_i| = |x - y| = x - y > 0$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^{n} w_i(|y - x_i| - |x - x_i|) \\ &= \sum_{i=1}^{n} w_i(x - y) \\ &\ge 0 . \end{aligned}$$

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,

$$\begin{aligned}
\min_{x,y} \{f(x, y)\} &= \min_{x,y} \{g(x) + h(y)\} \\
&= \min_{x} \left\{ \min_{y} \{g(x) + h(y)\} \right\} \\
&= \min_{x} \left\{ g(x) + \min_{y} \{h(y)\} \right\} \\
&= \min_{x} \{g(x)\} + \min_{y} \{h(y)\} .
\end{aligned}$$

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 9-4

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: in lines 7 and 21. If SELECT returns in line 7, it has placed the $i - 1$ smallest elements into $A[1 : i - 1]$, in order, and then placed the ith smallest element into $A[i]$. If SELECT returns in line 21, then lines 1–10 placed the n mod 5 smallest elements into the beginning positions of A, and the prior calls to PARTITION-AROUND have placed the remaining elements smaller than $A[q]$ into positions before $A[q]$.

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, $U_i(n) = S(n)$.

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, $p = 0$.

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 $p+m+1 \le j, k \le p+2m$), it also exchanges $A[j-m]$ and $A[k-m]$. (Note that if n is odd, then $p+2m < n$, so that no exchange involving $A[n-m]$ occurs because of an exchange involving $A[n]$. Including $A[n]$ in recursively finding the ith smallest element in $A[p+m+1:p+n]$ explains why the comparisons in step 2 put the smaller element in the higher-indexed positions: if $A[n]$ is one of the i smallest elements out of itself and the smaller elements found in step 2, the recursive step treats it correctly.) 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 the procedure recurses 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 the procedure calls SELECT on an array with $2i$ elements, the number of comparisons is $S(2i)$.

Thus, when $i < n/2$, the total number of comparisons is $\lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + S(2i)$.

b. We show by substitution that if $i < n/2$, then $U_i(n) = n + O(S(2i)\lg(n/i))$. In particular, we show that $U_i(n) \le n + cS(2i)\lg(n/i) - d(\lg\lg n)S(2i) = n + cS(2i)\lg n - cS(2i)\lg i - d(\lg\lg n)S(2i)$, where $c > 0$ is the constant hidden in the O-notation, d is a positive constant to be chosen later, and $n \ge 4$. We have

$$\begin{aligned}
U_i(n) &= \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + S(2i) \\
&\le \lfloor n/2 \rfloor + \lceil n/2 \rceil + cS(2i)\lg\lceil n/2 \rceil - cS(2i)\lg i \\
&\qquad - d(\lg\lg\lceil n/2 \rceil)S(2i) \\
&= n + cS(2i)\lg\lceil n/2 \rceil - cS(2i)\lg i - d(\lg\lg\lceil n/2 \rceil)S(2i) \\
&\le n + cS(2i)\lg(n/2+1) - cS(2i)\lg i - d(\lg\lg(n/2))S(2i) \\
&= n + cS(2i)\lg(n/2+1) - cS(2i)\lg i - d(\lg(\lg n - 1))S(2i) \\
&\le n + cS(2i)\lg n - cS(2i)\lg i - d(\lg\lg n)S(2i)
\end{aligned}$$

if $cS(2i)\lg(n/2+1) - d(\lg(\lg n - 1))S(2i) \le cS(2i)\lg n - d(\lg\lg n)S(2i)$. Algebraic manipulations give the following sequence of equivalent conditions:

$$
\begin{aligned}
&cS(2i)\lg(n/2+1) - d(\lg(\lg n - 1))S(2i) \le cS(2i)\lg n - d(\lg\lg n)S(2i) \\
&c\lg(n/2+1) - d(\lg(\lg n - 1)) \le c\lg n - d(\lg\lg n) \\
&c(\lg(n/2+1) - \lg n) \le d(\lg(\lg n - 1) - \lg\lg n) \\
&c\left(\lg\frac{n/2+1}{n}\right) \le d\lg\frac{\lg n - 1}{\lg n} \\
&c\left(\lg\left(\frac{1}{2}+\frac{1}{n}\right)\right) \le d\lg\frac{\lg n - 1}{\lg n}
\end{aligned}
$$

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, $S(2i) = O(1)$ and $\lg(n/i) = \lg n - \lg i = O(\lg n)$. Thus, when i is a constant less than $n/2$, we have that

$$
\begin{aligned}
U_i(n) &= n + O(S(2i)\lg(n/i)) \\
&= n + O(O(1)\cdot O(\lg n)) \\
&= n + O(\lg n) .
\end{aligned}
$$

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

$$
\begin{aligned}
U_i(n) &= n + O(S(2i)\lg(n/i)) \\
&= n + O(S(2n/k)\lg(n/(n/k))) \\
&= n + O(S(2n/k)\lg k) .
\end{aligned}
$$

If $k = 2$, then $n = 2i$ and $\lg k = 1$. We have

$$
\begin{aligned}
U_i(n) &= S(n) \\
&= n + (S(n) - n) \\
&\le n + (S(2i) - n) \\
&= n + (S(2n/k) - n) \\
&= n + (S(2n/k)\lg k - n) \\
&= n + O(S(2n/k)\lg k) .
\end{aligned}
$$

Solution to Problem 9-5

a. As in the analysis of Section 7.4.2, elements z_j and z_k are never compared with each other if any element in $z_{j+1}, \ldots, z_{k-1}$ is chosen as a pivot element before either z_j or z_k, since z_j and z_k would then lie in different partitions. There is another reason that z_j and z_k might not be compared with each other. Suppose that $i < j$, so that $z_i < z_j$, and suppose further that z_l is chosen as a pivot, where $i \le l < j$. In this case, because $i \le l$, all elements $z_{l+1}, \ldots, z_n$ are not even examined in any future recursive calls. That is, after partitioning with z_l as the pivot, no future recursive call will ever examine z_j or z_k, and they will never be compared with each other. Likewise, if $k < i$ and the pivot element is z_l, where $k < l \le i$, then after partitioning with z_l as the pivot, z_j and z_k will not be compared with each other in any future recursive call. The remaining case is when $j \le i \le k$, with at least one of these inequalities being strict, where the analysis is the same as for quicksort: z_j and z_k are compared with each other only if one of them becomes the pivot element.

Returning to the case where $i < j$, we know that z_j and z_k are compared with each other only if one of them is chosen as a pivot element. They are never compared with each other if the pivot is between them or if the pivot is z_l for $l < j$. Similarly, when $k < i$, elements z_j and z_k are compared with each other only if one of them is chosen as a pivot element, and they are never compared with each other if the pivot is between them or if the pivot is z_l for $k < l$.

In order to compute the probability that z_j and z_k are compared with each other, define

$$Z_{ijk} = \begin{cases} \{z_j, z_{j+1}, \ldots, z_k\} & \text{if } j \le i \le k\ , \\ \{z_i, z_{i+1}, \ldots, z_k\} & \text{if } i < j < k\ , \\ \{z_j, z_{j+1}, \ldots, z_i\} & \text{if } j < k < i\ . \end{cases}$$

That is, Z_{ijk} is the set of elements $z_j, \ldots, z_k$ along with $z_i, \ldots, z_{j-1}$ if $i < j$ or $z_{k+1}, \ldots, z_i$ if $k < i$. With this definition of Z_{ijk}, we have

$$|Z_{ijk}| = \begin{cases} k - j + 1 & \text{if } j \le i \le k\ , \\ k - i + 1 & \text{if } i < j < k\ , \\ i - j + 1 & \text{if } j < k < i\ . \end{cases}$$

Until an element from Z_{ijk} is chosen as the pivot, the entire set Z_{ijk} remains together in the same partition, so that each each element of Z_{ijk} is equally likely to be the first one chosen as the pivot. We can now compute $\mathrm{E}[X_{ijk}]$. By Lemma 5.1, we have

$$\begin{aligned} \mathrm{E}[X_{ijk}] &= \Pr\{z_j \text{ is compared with } z_k \text{ sometime during the execution} \\ &\qquad\quad \text{of the algorithm to find } z_i\} \\ &= \Pr\{z_j \text{ or } z_k \text{ is the first pivot chosen from } Z_{ijk}\} \\ &= \Pr\{z_j \text{ is the first pivot chosen from } Z_{ijk}\} \\ &\qquad + \Pr\{z_k \text{ is the first pivot chosen from } Z_{ijk}\} \\ &= \frac{1}{|Z_{ijk}|} + \frac{1}{|Z_{ijk}|} \\ &= \begin{cases} 2/(k - j + 1) & \text{if } j \le i \le k\ , \\ 2/(k - i + 1) & \text{if } i < j < k\ , \\ 2/(i - j + 1) & \text{if } j < k < i\ . \end{cases} \end{aligned}$$

b. Letting X_i denote the total number of comparisons performed when selecting the ith smallest element, we add up all the possible pairs of elements that might be compared to get

$$X_i = \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} X_{ijk} \ ,$$

so that

$$\begin{aligned} \mathrm{E}[X_i] &= \mathrm{E}\left[\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} X_{ijk}\right] \\ &= \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \mathrm{E}[X_{ijk}] \qquad \text{(by linearity of expectation)} \ . \end{aligned}$$

Depending on i, j, and k, one of the three cases holds, so that $\mathrm{E}[X]$ is bounded from above by the sum of all three. With i fixed, we vary j and k. For the case $j \le i \le k$, we further loosen the upper bound by allowing both j and k to equal i. For the case $i < j < k$, we further loosen the upper bound by allowing both j and k to start at $i + 1$. For the case $j < k < i$, we further loosen the upper bound by allowing j to go up to $i - 2$, regardless of k. Using the bounds on $\mathrm{E}[X_{ijk}]$ from above, we get

$$\begin{aligned} \mathrm{E}[X_i] &\le \sum_{j=1}^{i} \sum_{k=i}^{n} \frac{2}{k-j+1} + \sum_{j=i+1}^{k-1} \sum_{k=i+1}^{n} \frac{2}{k-i+1} + \sum_{j=1}^{i-2} \sum_{k=j+1}^{i-1} \frac{2}{i-j+1} \\ &= 2\left(\sum_{j=1}^{i} \sum_{k=i}^{n} \frac{1}{k-j+1} + \sum_{k=i+1}^{n} \frac{k-i-1}{k-i+1} + \sum_{j=1}^{i-2} \frac{i-j-1}{i-j+1}\right) \ . \end{aligned}$$

c. Observe that each of the latter two summations sums fractions that are strictly less than 1. Together, they encompass $(n - i) + (i - 2) = n - 2$ terms, totaling less than n. For the first summation, for a fixed value of i, let $m = k - j$. The only way that m can equal 0 is for $j = i = k$ (which isn't even really allowed, but remember that we're just deriving an upper bound). There are two ways for m to equal 1: either $j = i - 1$ and $k = i$ or $j = i$ and $k = i + 1$. There are three ways for m to equal 2: $j = i - 2, k = i$; $j = i - 1, k = i + 1$; or $j = i$, $k = i + 2$. And so on, so that for each value of m, there are $m + 1$ ways for $k - j$ to equal m. Since $k - j \le n - 1$, we can rewrite the first summation as

$$\begin{aligned} \sum_{j=1}^{i} \sum_{k=i}^{n} \frac{2}{k-j+1} &= \sum_{m=0}^{n-1} \frac{m+1}{m+1} \\ &= n \ . \end{aligned}$$

Thus, we have

$$\begin{aligned} \mathrm{E}[X] &\le 2(n + n) \\ &= 4n \ . \end{aligned}$$

d. We can repurpose Lemma 7.1 for RANDOMIZED-SELECT:

Lemma

Let X_i be the number of comparisons performed in line 4 of PARTITION over the entire execution of RANDOMIZED-SELECT on an n-element array when selecting the ith smallest element. Then the running time of RANDOMIZED-SELECT is $O(n + X_i)$. ■

By this lemma and part (c), assuming that all elements of the array are distinct, the expected running time of RANDOMIZED-SELECT is $O(n)$.

Lecture Notes for Chapter 10: Elementary Data Structures

Chapter 10 overview

This chapter examines representations of dynamic sets by simple data structures which use pointers. We will look at rudimentary data structures: arrays, matrices, stacks, queues, linked lists, rooted trees.

Simple array-based data structures: arrays, matrices, stacks, and queues

Arrays

Arrays store elements contiguously in memory.

- If
  - the first element of an array has index s,
  - the array starts at memory address a, and
  - each element occupies b bytes,

  then the ith element occupies bytes $a + b(i - s)$ through $a + b(i + 1 - s) - 1$.

  The most common values for s are 0 and 1.

  - $s = 0 \Rightarrow a + bi$ through $a + b(i + 1) - 1$.
  - $s = 1 \Rightarrow a + b(i - 1)$ through $a + bi - 1$.
- The computer can access any array element in constant time (assuming that the computer can access all memory locations in same amount of time).
- If elements of an array occupy different numbers of bytes, elements might be accessed incorrectly or not in constant time. Therefore, most programming languages require that each element of an array must be the same size. Sometimes, pointers to objects are stored instead of objects themselves in order to meet this requirement.

Matrices

Notation: an $m \times n$ matrix has m rows and n columns.

We represent a matrix with one or more arrays.

Two common ways to store a matrix:

- ***Row-major***: matrix is stored row by row.
- ***Column-major***: matrix is stored column by column.

Example: Consider the 2×3 matrix

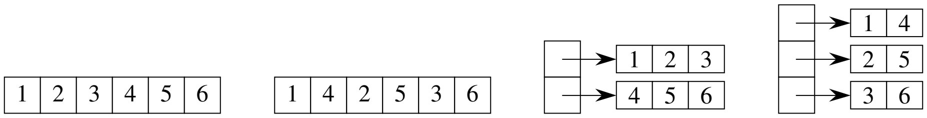
$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} . \qquad (*)$$

Row-major order: store two rows 1 2 3 and 4 5 6

Column-major order: store three columns 1 4; 2 5; and 3 6.

There are many ways to store M. Shown below, from left to right, are four possible ways:

1. In row-major order, single array.
2. In column-major order, single array.
3. In row-major order, one array per row with a single array of pointers to the row arrays.
4. In column-major order, one array per column with a single array of pointers to the column arrays.



There are other ways to store matrices. In the ***block representation***, divide a matrix into blocks and then store each block contiguously. For example, divide a 4×4 matrix into 2×2 blocks, such as

$$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

and store the matrix in a single array in the order $\langle 1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16 \rangle$.

Stacks and Queues

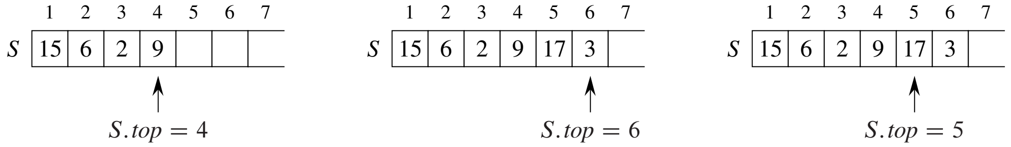
Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified.

Stack: the element deleted is the one that was most recently inserted. Stacks use a ***last-in, first-out***, or ***LIFO***, policy.

Queue: the element deleted is the one that has been in the set for the longest time. Queues use a ***first-in, first-out***, or ***FIFO***, policy.

Stacks

Implement a stack of at most n elements with an array $S[1:n]$. Attribute $S.top$ indexes the most recently inserted element. The stack contains elements $S[1:S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top. Attribute $S.size = n$ gives the size of the array.



Stack operations: PUSH, POP, STACK-EMPTY.

- STACK-EMPTY: When $S.top = 0$, the stack contains no elements and is empty.

```
STACK-EMPTY(S)
  if $S.top$ == 0
      return TRUE
  else return FALSE
```

- PUSH: The INSERT operation on a stack. Like pushing a plate on top of a stack of plates in a cafeteria. Pushing onto a full stack causes an overflow.

```
PUSH(S, x)
  if $S.top$ == $S.size$
      error "overflow"
  else $S.top = S.top + 1$
      $S[S.top] = x$
```

- POP: The DELETE operation on a stack. Like popping off the plate on the top the stack. Order in which plates are popped from the stack is reverse of order in which they were pushed. Popping an empty stack causes underflow.

```
POP(S)
  if STACK-EMPTY(S)
      error "underflow"
  else $S.top = S.top - 1$
      return $S[S.top + 1]$
```

All three stack operations take $O(1)$ time.

The figure above shows an array implementation of a stack S.

- Left: Stack S has 4 elements. The top element is 9.
- Middle: Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$.
- Right: Stack S after the call POP(S) has returned the element 3. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

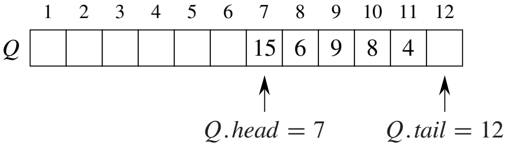
Queues

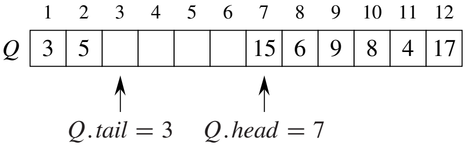
Inserting into a queue is ***enqueuing***, and deleting from a queue is ***dequeuing***.

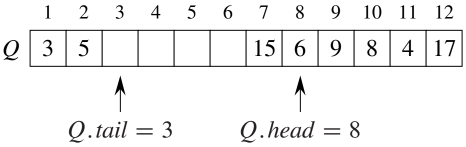
The FIFO property of a queue causes it to operate like a line of customers waiting for service. A queue has a ***head*** and a ***tail***.

- When an element is enqueued, it goes to the tail of the queue, just as a newly arriving customer takes a place at the end of the line.
- The element dequeued is the one at the head of the queue, like the customer at the head of the line who has waited the longest.

Implement a queue of at most $n - 1$ elements with an array $Q[1:n]$.







- $Q.head$ indexes the head.
- $Q.tail$ indexes the next location at which at which a new element will be inserted into the queue.
- Elements reside in $Q.head, Q.head + 1, \ldots, Q.tail - 1$, wrapping around so that $Q[1]$ follows $Q[n]$.
- Initially, $Q.head = Q.tail = 1$.
- $Q.head = Q.tail \Rightarrow$ queue is empty. Attempting to dequeue causes underflow.
- $Q.head = Q.tail + 1$ or both $Q.head = 1$ and $Q.tail = n \Rightarrow$ the queue is full. Attempting to enqueue causes overflow.
- Attribute $Q.size$ gives the size n of the array.

[The ENQUEUE *and* DEQUEUE *procedures here omit error checking for overflow and underflow. Exercise 10.1-5 in the book adds these checks.]*

```
ENQUEUE($Q, x$)
  $Q[Q.tail] = x$
  if $Q.tail == Q.size$
      $Q.tail = 1$
  else $Q.tail = Q.tail + 1$
```

```
DEQUEUE($Q, n$)
  $x = Q[Q.head]$
  **if** $Q.head == Q.size$
      $Q.head = 1$
  **else** $Q.head = Q.head + 1$
  **return** $x$
```

The two queue operations take $O(1)$ time.

The figure above shows a queue implemented using an array $Q[1:12]$.

- Top: Queue Q has 5 elements, in locations $Q[7:11]$.
- Middle: Queue Q after the calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5).
- Bottom: Queue Q after the call DEQUEUE(Q) has returned the key value 15 formerly at the head. The new head has key 6. Although 15 is still in the array, it is not in the queue.

Linked lists

A ***linked list*** has

- Objects arranged in a linear order.
- Order is determined by a pointer in each object.

In a ***doubly linked list***, each element x has the following attributes:

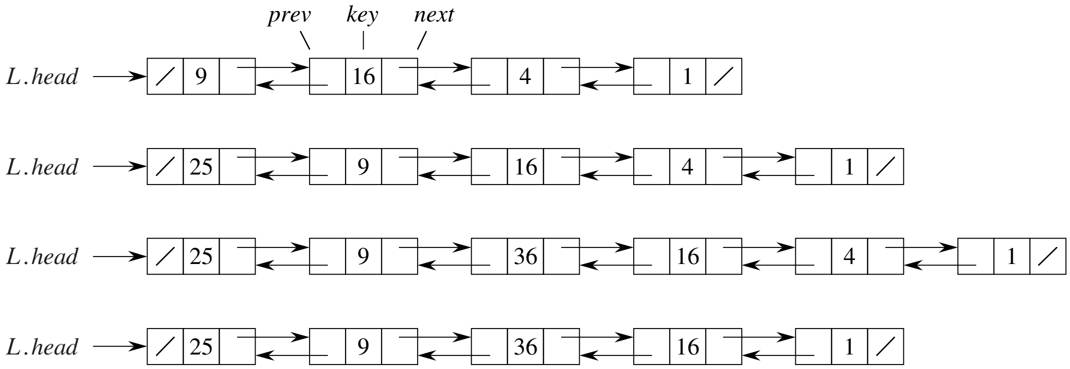
- $x.key$
- $x.next$: the successor of x, NIL if x has no successor so that it's the ***tail***
- $x.prev$: the predecessor of x, NIL if x has no predecessor so that it's the ***head***

$L.head$ points to the first element of the list, NIL if the list is empty.

Linked lists come in several types:

- ***Singly linked***: each element has a *next* attribute but not a *prev* attribute.
- ***Sorted***: the linear order of the list follows the linear order of keys stored in elements of the list.
- ***Unsorted***: the elements can appear in any order.
- ***Circular***: the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head.

[Start with just the topmost list in the following figure.] Here is a doubly linked list whose elements have keys 9, 16, 4, 1. Slashes indicate NIL.



Searching a linked list

LIST-SEARCH finds the first element with key k in list L by a linear search. It returns either a pointer x to the element, or NIL if no element has key k. The worst-case time on a list with n elements is $\Theta(n)$.

```
LIST-SEARCH$(L, k)$
  $x = L.head$
  while $x \neq$ NIL and $x.key \neq k$
      $x = x.next$
  return $x$
```

In the first figure above, searching for key 16 returns a pointer to the second element in list L. Searching for key 49 returns NIL.

Inserting into a linked list

There are two scenarios for inserting into a doubly linked list: inserting a new first element and inserting anywhere else. Given an element x with the *key* element set, the procedure LIST-PREPEND adds x to the front of the list L in $O(1)$ time.

```
LIST-PREPEND$(L, x)$
  $x.next = L.head$
  $x.prev =$ NIL
  if $L.head \neq$ NIL
      $L.head.prev = x$
  $L.head = x$
```

The second figure above shows the result of prepending 25.

To insert elsewhere, LIST-INSERT "splices" a new element x into the list, immediately following y. Since the list object L is not referenced, it's not supplied as a parameter. Like LIST-PREPEND, this procedure takes $O(1)$ time.

```
LIST-INSERT$(x, y)$
  $x.next = y.next$
  $x.prev = y$
  if $y.next \neq$ NIL
      $y.next.prev = x$
  $y.next = x$
```

The third figure above shows the result of inserting 36 after 9.

Deleting from a linked list

Given a pointer to x, LIST-DELETE removes x from L in $O(1)$ time.

```
LIST-DELETE$(L, x)$
  if $x.prev \neq$ NIL
      $x.prev.next = x.next$
  else $L.head = x.next$
  if $x.next \neq$ NIL
      $x.next.prev = x.prev$
```

The fourth figure above shows the result of deleting 4.

To delete an element just given a key, first call LIST-SEARCH, then call LIST-DELETE. This makes the worst-case running time $\Theta(n)$.

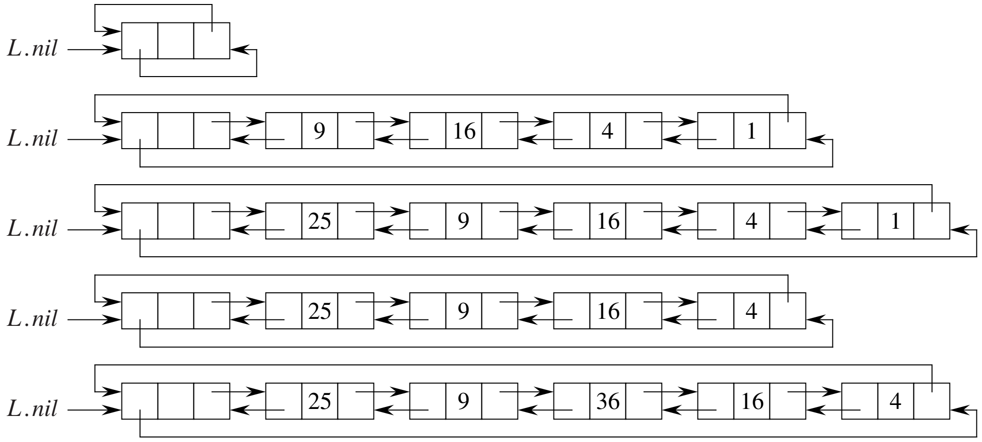
Linked list and array performance

Insertion and deletion are faster on doubly linked lists than on arrays. In an array, insertion and deletion take $\Theta(n)$ time in the worst case, where all elements need to be shifted. In a doubly linked list, they take $O(1)$ time.

Access by index, however, is faster in an array. Accessing the kth element in the linear order would take $\Theta(k)$ time in a linked list, but only $O(1)$ time in an array.

Sentinels

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. In a ***circular doubly linked list with a sentinel***, replace NIL with a reference to the sentinel $L.nil$. $L.nil.next$ points to the head of the list, and $L.nil.prev$ points to the tail. *prev* attribute of the head and *next* attribute of the tail both point to $L.nil$. No attribute $L.head$ needed.



The first figure above shows an empty list: $L.nil.next = L.nil.prev = L.nil$. The second figure shows a list whose elements have keys $9, 16, 4, 1$.

How to delete an element x no longer depends on where in the list x is located. No need to supply L as a parameter. Never delete the sentinel $L.nil$ unless you want to delete the entire list.

```
LIST-DELETE′(x)
  $x.prev.next = x.next$
  $x.next.prev = x.prev$
```

Now one procedure to insert fits all situations. To insert x at the head of the list, set y to $L.nil$. To insert x at the tail, set y to $L.nil.prev$.

```
LIST-INSERT′(x, y)
  $x.next = y.next$
  $x.prev = y$
  $y.next.prev = x$
  $y.next = x$
```

The third figure above inserts an element with key 25 after $L.nil$. The fourth figure deletes the element with key 1 by calling LIST-DELETE′$(L.nil.prev)$. The fifth figure inserts an element with key 36 after the element with key 9.

Searching has the same asymptotic running time as without a sentinel, but the constant factor can be better by removing one test per loop iteration. Instead of checking first whether the search has hit the end of the list and then, if it hasn't, whether the key is in the current element, eliminate the check for hitting the end of the list. The trick is to guarantee that the key will be found, by putting it in the sentinel. Start at the head. If the key is really in the list, it will be found before getting back to the sentinel. If the key is not really in the list, it is found only in the sentinel.

```
LIST-SEARCH'(L, k)
  $L.nil.key = k$          // store the key in the sentinel to guarantee it is in list
  $x = L.nil.next$         // start at the head of the list
  while $x.key \neq k$
      $x = x.next$
  if $x == L.nil$          // found $k$ in the sentinel
      return NIL           // $k$ was not really in the list
  else return $x$          // found $k$ in element $x$
```

Representing rooted trees

For linear relationships, linked lists work well. But how to handle non-linear relationships?

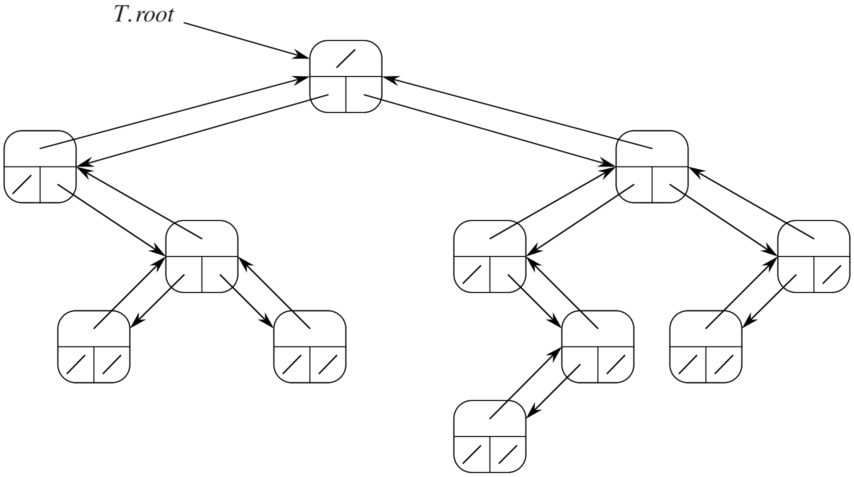
Represent a rooted tree by linked data structures. Each node of a tree is an object with a *key* attribute, like linked lists, and also has attributes that are pointers to other nodes.

Binary trees

Give each node in a binary tree the following attributes: p (parent), *left*, *right*.

If $x.p =$ NIL, x is the root. The root of tree T is $T.root$. If $T.root =$ NIL, then T is empty.

If x has no left child, then $x.left =$ NIL. Same for right child.



Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). *key* attributes are not shown.

Rooted trees with unbounded branching

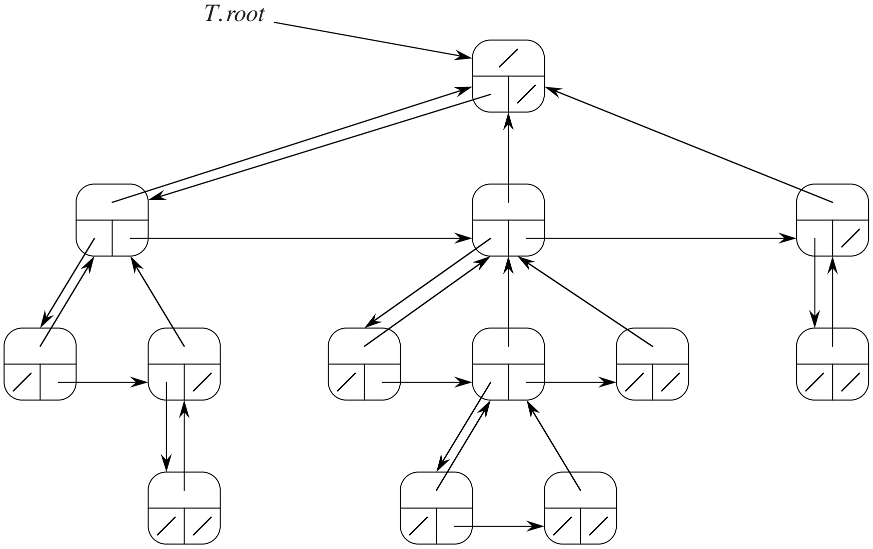
In a binary tree, each node has at most two children. Can extend this representation to let each node have at most k children: replace the *left* and *right* attributes by $child_1, child_2, \ldots, child_k$.

Problems with this representation:

- If the number of children of a node is unbounded, do not know how many attributes to allocate in advance.
- Even if the number of children is bounded, but most nodes have a small number of children, can waste a lot of memory.

Solution: ***Left-child, right-sibling representation***. Now, each node still has attribute p, but each node has two other pointers:

1. x.*left-child* points to the leftmost child of node x, and
2. x.*right-sibling* points to the sibling of x immediately to its right.



Each node x has the attributes $x.p$ (top), x.*left-child* (lower left), x.*right-sibling* (lower right). *key* attributes are not shown.

Other tree representations

There are other ways to represent rooted trees. Some examples:

- A heap, detailed in Chapter 6, which is a complete binary tree represented by a single array with an attribute giving the index of the last node in the heap.
- Chapter 19 uses trees with no pointers to children, only pointers to parents are present because trees are traversed only toward the root.

The best scheme depends on the application of the tree.

Solutions for Chapter 10: Elementary Data Structures

Solution to Exercise 10.1-1

We can construct the binary representation of the $(\lg m + \lg n)$-bit index as $\langle i_{\lg m-1}, j_{\lg n-1}, i_{\lg m-2}, \ldots, i_0, j_{\lg n-2}, \ldots, j_0 \rangle$.

The two most significant bits in block order correspond to the block row and column. These bits come from the most significant bits of the row and column numbers i and j. The relative orderings within each block remain the same as in the full matrix, so that the rest of the bits are the $\lg m - 1$ least significant bits of i followed by the $\lg n - 1$ least significant bits of j.

Solution to Exercise 10.1-3

One stack starts at$A[1]$ and expands to higher indices of$A$. The other stack starts at $A[n]$ and expands to lower indices of A.

Solution to Exercise 10.1-5

```
ENQUEUE($Q, x$)
  if $Q.head$ == $Q.tail + 1$ or ($Q.head$ == 1 and $Q.tail$ == $Q.size$)
      error "overflow"
  else $Q[Q.tail] = x$
      if $Q.tail$ == $Q.size$
          $Q.tail = 1$
      else $Q.tail = Q.tail + 1$
```

```
DEQUEUE($Q$)
  if $Q.head == Q.tail$
      error "underflow"
  else $x = Q[Q.head]$
      if $Q.head == Q.size$
          $Q.head = 1$
      else $Q.head = Q.head + 1$
      return $x$
```

Solution to Exercise 10.1-7

Call the two stacks S_1 and S_2.

To ENQUEUE, push a new element onto stack S_1. This operation takes $O(1)$ time.

To DEQUEUE, pop the top element from stack S_2. If stack S_2 is empty when a DEQUEUE is requested, first empty stack S_1 into stack S_2 by popping elements one at a time from stack S_1 and pushing them onto stack S_2. Copying the stack reverses its order, so that the oldest element is then on top and can be removed with DEQUEUE.

DEQUEUE takes $O(1)$ time in the best case and $O(n)$ time in the worst case. Each element is moved from stack S_1 to stack S_2 at most one time, so that the time averaged over all operations is $O(1)$.

Solution to Exercise 10.1-8

Call the two queues Q_1 and Q_2.

To PUSH, enqueue a new element onto queue Q_1. This operation takes $O(1)$ time.

To POP, dequeue all but one element from queue Q_1, enqueuing them into queue Q_2, leaving the last element. Return the last element, and dequeue it from Q_1. After returning the last element, relabel the queues so that the queue holding all of the elements is queue Q_1 and the empty queue is queue Q_2.

The POP operation takes $O(n)$ time, since all of the elements have to be dequeued.

Solution to Exercise 10.2-1

LIST-PREPEND and LIST-INSERT for a singly linked list are easily done in $O(1)$ time:

```
LIST-PREPEND($L, x$)
  $x.next = L.head$
  $L.head = x$
```

```
LIST-INSERT$(x, y)$
  $x.next = y.next$
  $y.next = x$
```

DELETE, however, takes $\Theta(n)$ time in the worst case because without the *prev* pointers, you first need to search through the list to find the predecessor of the element being deleted so that you can update its *next* value. The procedure would look like this:

```
LIST-DELETE$(L, x)$
  if $L.head == x$
      $L.head = x.next$
  else $predecessor = L.head$
      while $predecessor.next \neq x$
          $predecessor = predecessor.next$
      $predecessor.next = x.next$
```

Solution to Exercise 10.2-2

To implement a stack by a singly linked list, PUSH is LIST-PREPEND and POP is LIST-DELETE$(L, L.head)$.

Solution to Exercise 10.2-3

To implement a queue by a singly linked list, you need to add another attribute, *tail*, to the list, pointing to the last element in the list. ENQUEUE is PREPEND and DEQUEUE is LIST-DELETE$(L, L.tail)$. The procedures LIST-PREPEND, LIST-INSERT, and LIST-DELETE would all have to update $L.tail$ when the tail element changes.

Solution to Exercise 10.2-4

Represent each set by a circular, doubly linked list with a sentinel. The UNION operation just appends S_2 to S_1 and declares the result to be S_1:

```
UNION$(S_1, S_2)$
  $S_2.nil.next.prev = S_1.nil.prev$
  $S_2.nil.prev.next = S_1.nil$
  $S_1.nil.prev.next = S_2.nil.next$
  $S_1.nil.prev. = S_2.nil.prev$
  return $S_1$
```

Solution to Exercise 10.2-5

```
LIST-REVERSE(L)
  previous = NIL
  current = L.head
  while current ≠ NIL
      successor = current.next
      current.next = previous
      previous = current
      current = successor
  L.head = previous
```

Solution to Exercise 10.2-6

We describe how to implement a doubly linked list with only one pointer. The list has the following attributes: *head*—a pointer to the first element, and *tail*—a pointer to the last element. (The *tail* attribute comes in handy when reversing the list.) Observe that since the first element does not have a previous element, its *np* attribute is a pointer to the second element, because XORing some pointer p with NIL yields back the value of p. Similarly, the *np* attribute of the last element is a pointer to the next-to-last element.

The SEARCH operation involves scanning the list from beginning to end, until the desired key is found. In the worst case, this operation takes $\Theta(n)$ time. While scanning the list, the procedure keeps track of two pointers, *current*, a pointer to the current element, and *previous*, a pointer to the previous element. Accessing the next element is done by XORing the pointer to the previous element with the *np* attribute of the current element. The two pointers *current* and *previous* are then updated to point to the next element and the current element, respectively. If the search succeeds, the procedure returns both *current* and *previous* because in order to insert immediately after an element or to delete an element, we need both a pointer to the element and a pointer to either its predecessor or its successor. Since we typically need to search before inserting after an element or deleting an element, it makes sense for the search operation to return both pointers. Of course, if the desired key is not present, NIL is returned.

In what follows, we denote the XOR operation with the operator $\oplus$.

```
LIST-SEARCH$(L, k)$
  $current = L.head$
  $previous =$ NIL
  while $current \neq$ NIL
      if $current.key == k$
          return $(current, previous)$
      else $successor = current.np \oplus previous$
          $previous = current$
          $current = successor$
  return NIL
```

Just as a regular doubly linked list has LIST-PREPEND and LIST-INSERT procedures, so does this type of list. Each takes $O(1)$ time.

```
LIST-PREPEND$(L, x)$
  $x.np = L.head$
  if $L.head \neq$ NIL
      $L.head.np = L.head.np \oplus x$
  else $L.tail = x$
  $L.head = x$
```

LIST-INSERT inserts element x after element y in the list. The procedure takes an extra parameter, *previous*, which is y's predecessor in the list. This pointer would normally be returned by a call of LIST-SEARCH, above. The procedure also takes the list object L as a parameter, in case x becomes the new tail.

```
LIST-INSERT$(L, x, y, previous)$
  $successor = y.np \oplus previous$
  $x.np = y \oplus successor$
  $y.np = y.np \oplus successor \oplus x$
  $successor.np = successor.np \oplus y \oplus x$
  if $y == L.tail$
      $L.tail = x$
```

Like LIST-INSERT, the LIST-DELETE operation needs not only the element to delete, but also its predecessor and the list object L.

```
LIST-DELETE$(L, x, previous)$
  if $previous ==$ NIL
      $L.head = x.np$
      if $L.head ==$ NIL
          $L.tail =$ NIL
      else $L.head.np = L.head.np \oplus x$
  else $successor = x.np \oplus previous$
      $previous.np = previous.np \oplus x \oplus successor$
      if $successor ==$ NIL
          $L.tail = previous$
      else $successor.np = successor.np \oplus x \oplus previous$
```

Reversing such a list is relatively simple, since the *np* attribute is symmetric with respect to the *prev* and *next* attributes. All that is needed is to switch the roles of the two external pointers, *head* and *tail*, which takes $O(1)$ time.

```
LIST-REVERSE(L)
  exchange L.head with L.tail
```

Solution to Exercise 10.3-2

Perform a recursive inorder tree traversal. The initial call is at *T.root*.

```
PRINT-BINARY-TREE(x)
  if x ≠ NIL
      PRINT-BINARY-TREE(x.left-child)
      print x.key
      PRINT-BINARY-TREE(x.right-child)
```

Solution to Exercise 10.3-3

The following nonrecursive traversal is inorder. It uses a stack that can hold as many nodes as necessary, so that the parameter for the stack size is omitted from calls to PUSH.

```
PRINT-BINARY-TREE-NONRECURSIVE(T)
  S = empty stack
  x = T.root
  while x ≠ NIL
      PUSH(S, x)
      x = x.left
  while not IS-EMPTY(S)
      x = POP(S)
      print x.key
      x = x.right
      while x ≠ NIL
          PUSH(S, x)
          x = x.left
```

Solution to Exercise 10.3-4

The initial call is at *T.root*.

```
PRINT-UNBOUNDED-DEGREE-TREE($x$)
  if $x \neq$ NIL
      print $x.key$
      PRINT-UNBOUNDED-DEGREE-TREE($x.left\text{-}child$)
      PRINT-UNBOUNDED-DEGREE-TREE($x.right\text{-}sibling$)
```

Solution to Exercise 10.3-5

The idea is to keep a pointer x to the current node being visited and another pointer y to the previous node visited. When visiting x, if coming from x's parent, go left if possible; otherwise go right if possible; otherwise go to x's parent. When coming from x's left child, go right if possible; otherwise go to x's parent. When coming from x's right child, go to x's parent. Don't go to a NIL child, so that when x is NIL, it must be the parent of the root; in this case, all nodes have been visited. Here is pseudocode to print the keys in preorder:

```
PRINT-BINARY-TREE-CONSTANT-EXTRA-SPACE($T$)
  $x = T.root$
  $y =$ NIL
  while $x \neq$ NIL
      $z = x$                 // save $x$ to assign to $y$ for next iteration
      if $x.p == y$
          print $x.key$       // coming from parent
          if $x.left \neq$ NIL
              $x = x.left$
          elseif $x.right \neq$ NIL
              $x = x.right$
          else $x = x.p$
      elseif $x.left == y$ and $x.right \neq$ NIL
          $x = x.right$       // coming from left child
      else $x = x.p$          // coming from right child
      $y = z$                 // $x$, stored in $z$, becomes previous node next time
```

Solution to Exercise 10.3-6

Each node stores its left child and either its right sibling or its parent. If it has a right sibling, it stores the right sibling. Otherwise (i.e., the node is the rightmost sibling), it stores its parent. The boolean value indicates whether the pointer is to the right sibling or the parent.

Solution to Problem 10-3

a. From the problem description, it is clear that COMPACT-LIST-SEARCH is correct. In order to show that both algorithms return the same result, we will show that COMPACT-LIST-SEARCH$'$ is correct.

COMPACT-LIST-SEARCH$'$ starts at the head of the list. In lines 2–7 the procedure, it attempts to skip ahead to a randomly chosen position t times. If the skip helps, i.e., if *key*$[j]$ is larger than *key*$[i]$ and no larger than k, then the procedure skips to position j. If the skip does not help, the loop continues on to its next iteration, so that the skip ahead is never incorrect. The rest of the procedure, from line 8 until the end, is an ordinary algorithm for searching a sorted linked list.

Because both algorithms are correct searches, they must return the same result.

Suppose that COMPACT-LIST-SEARCH makes r iterations of the **while** loop of lines 2–8. We need to show that the total number of iterations of both the **for** and **while** loops within COMPACT-LIST-SEARCH$'$ is at least r.

First, consider the case where t, the parameter in COMPACT-LIST-SEARCH$'$, is greater than or equal to r.

If COMPACT-LIST-SEARCH happens to choose a j such that *key*$[j] = k$, then that must occur in the rth iteration. Because the sequence of random numbers is the same for both algorithms, the **for** loop of lines 2–7 of COMPACT-LIST-SEARCH$'$ must iterate r times before returning a value.

If the **while** loop in COMPACT-LIST-SEARCH terminates after r iterations without returning a value, the **for** loop of lines 2–7 of COMPACT-LIST-SEARCH$'$ still runs at least r times, since $r \le t$.

Now, consider the case where $t < r$.

If COMPACT-LIST-SEARCH chooses a j such that *key*$[j] = k$ in the rth iteration, the **for** loop of lines 2–7 of COMPACT-LIST-SEARCH$'$ runs t times without returning a value, since it does not produce the random value of j that COMPACT-LIST-SEARCH gets to. Let $\hat{i}$ be the value that COMPACT-LIST-SEARCH had for i at the tth iteration of the **while** loop. At the end of its **for** loop, i in COMPACT-LIST-SEARCH$'$ is at a position in the list at or toward the head from $\hat{i}$.

In this case, COMPACT-LIST-SEARCH$'$ proceeds to line 8 and performs a linear search on the list until it finds an i such that *key*$[i] \ge k$. Line 8 of the **while** loop in COMPACT-LIST-SEARCH advances through the list in a linear fashion. After the tth iteration, COMPACT-LIST-SEARCH has $r - t$ iterations remaining. Therefore, by the rth iteration of its **while** loop, COMPACT-LIST-SEARCH advances to a position of at least $\hat{i} + r - t$ in the list.

Therefore, COMPACT-LIST-SEARCH$'$ needs to check at least the next $r - t$ positions of the list, and so the **while** loop of lines 8–9 iterates at least $r - t$ times. Thus, the total number of iterations of the **for** and **while** loops is at least $t + (r - t) = r$.

If the **while** loop in COMPACT-LIST-SEARCH terminates after r iterations without returning a value, the loop must have terminated because $i = \text{NIL}$ or because it is not the case that $key[i] < k$.

After t iterations of the **for** loop in COMPACT-LIST-SEARCH′, a value where $key[i] = k$ has not been found, since both algorithms have the same sequence of random numbers. As before, the **while** loop in lines 8–9 of COMPACT-LIST-SEARCH′ iterates at least $r - t$ times until $i = \text{NIL}$ or $key[i] \geq k$, for a total of at least r iterations.

b. The **for** loop of lines 2–7 iterates t times, with an $O(1)$ cost per iteration. The **while** loop of lines 8–9 runs X_t times, also with an $O(1)$ cost per iteration, so that its expected running time is $O(\mathrm{E}[X_t])$. All other lines of the code run in $O(1)$ time. Therefore, the expected running time is $O(t + \mathrm{E}[X_t])$.

c. By equation (C.28),

$$\mathrm{E}[X_t] = \sum_{r=1}^{\infty} \Pr\{X_t \geq r\} .$$

The probability of getting a distance larger than n is 0, since all chosen positions are within the n positions in the list, and so

$$\mathrm{E}[X_t] = \sum_{r=1}^{\infty} \Pr\{X_t \geq r\} = \sum_{r=1}^{n} \Pr\{X_t \geq r\} .$$

Now, we need to find $\Pr\{X_t \geq r\}$.

Let Y_i be the event that the ith randomly chosen position is at least r positions from key k. The probability of a randomly chosen position being within 0 to $r - 1$ positions of key k is r/n. Therefore, the probability of Y_i is $1 - r/n$.

For X_t to be at least r away from the location of key k, all t randomly chosen positions must be at least r away from the location of key k. Therefore, $\Pr\{X_t \geq r\} = \Pr\{Y_1 \cap \cdots \cap Y_t\}$. Because the events Y_i are independent, $\Pr\{Y_1 \cap \cdots \cap Y_t\} = \prod_{i=1}^{t} \Pr\{Y_i\}$, which is $(1 - r/n)^t$.

Thus,

$$\mathrm{E}[X_t] = \sum_{r=1}^{n} \Pr\{X_t \geq r\} = \sum_{r=1}^{n} (1 - r/n)^t .$$

d. By inequality (A.18),

$$\sum_{r=0}^{n-1} r^t \leq \int_0^n r^t \, dr = \frac{n^{t+1}}{t+1} .$$

e. We have

$$\begin{aligned}
\mathrm{E}[X_t] &\leq \sum_{r=1}^{n} (1 - r/n)^t \quad \text{(from part (c))} \\
&= \sum_{r=1}^{n} \frac{(n-r)^t}{n^t}
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{n^t} \sum_{s=0}^{n-1} s^t && \text{(by equation (A.13))} \\
&\le \frac{1}{n^t} \left(\frac{n^{t+1}}{t+1} \right) && \text{(by part (d))} \\
&= \frac{n}{t+1} \; .
\end{aligned}
$$

f. From part (b), we know that the expected running time of COMPACT-LIST-SEARCH′ is $O(t + \mathrm{E}[X_t])$. Therefore, we have $t + \mathrm{E}[X_t] \le t + n/(t+1) = O(t + n/t)$.

g. The expected running time of COMPACT-LIST-SEARCH is at most the expected running time of COMPACT-LIST-SEARCH′ for any value of t. Choose the value $t = \sqrt{n}$. Then the expected running time of COMPACT-LIST-SEARCH′ is $O(\sqrt{n} + n/\sqrt{n}) = O(\sqrt{n})$.

h. Suppose that there are equal keys. If line 3 of COMPACT-LIST-SEARCH happens to choose an index j such that $key[i] = key[j]$ and j is later in the list than i, the procedure does not update i in line 5, which would be inefficient. This problem could be solved by changing the test $key[i] < key[j]$ in line 4 to $key[i] \le key[j]$. But then if line 3 chooses an index j such that $key[i] = key[j]$ and j is earlier in the list than i, the procedure moves backward in the list. Therefore, it is necessary to assume that all keys are distinct.

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, store the element whose key is k in position k of the array.
- Given a key k, to find the element whose key is k, just look in the kth position of the array. This is called ***direct addressing***.
- Direct addressing is applicable when you can afford to allocate an array with one position for every possible key.

Use a hash table when 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 several approaches.
- What to do when the hash function maps multiple keys to the same table entry. We'll look at chaining and open addressing.

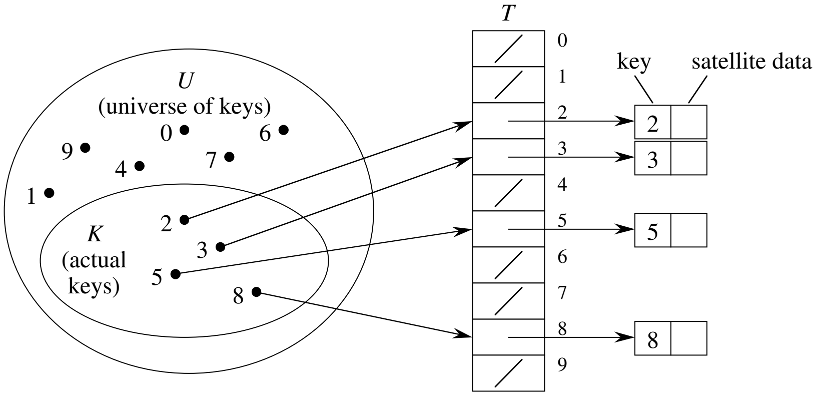
Direct-address tables

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[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
  T[x.key] = 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*** and T a ***hash table***.
- $h : U \rightarrow \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k ***hashes*** to slot $h(k)$.

Collision

When two or more keys hash to the same slot.

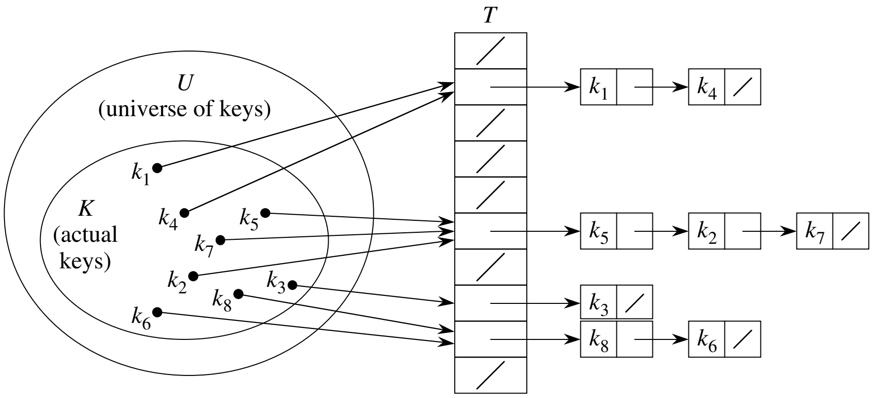
- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing. We'll examine both.

Independent uniform hashing

- Ideally, $h(k)$ would be randomly and independently chosen uniformly from $\{0, 1, \ldots, m-1\}$. Once $h(k)$ is chosen, each subsequent evalution of $h(k)$ must yield the same result.
- Such an idea hash function is an ***independent uniform hash function***, or ***random oracle***.
- It's an ideal theoretical abstraction. Cannot be reasonably implemented in practice. Use it to analyze hashing behavior, and find practical approximations to the ideal.

Collision resolution by chaining

Like a nonrecursive type of divide-and-conquer: use the hash function to divide the n elements randomly into m subsets, each with approximately $\lfloor n/m \rfloor$ elements. Manage each subset independently as a linked list.



[This figure shows singly linked lists. If needed 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$(T, x)$
  LIST-PREPEND$(T[h(x.key)], x)$
```

  - 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$(T, k)$
  return LIST-SEARCH$(T[h(k)], k)$
```

  Worst-case running time is proportional to the length of the list of elements in slot $h(k)$.

- ***Deletion:***

```
CHAINED-HASH-DELETE$(T, x)$
  LIST-DELETE$(T[h(x.key)], x)$
```

  - 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 need to 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.

Focus on average-case performance of hashing with chaining.

- Assume ***independent uniform hashing***: any given element is equally likely to hash into any of the m slots, independent of where any other elements hash to.
- Independent uniform hashing is ***universal***: probability that any two distinct keys collide is $1/m$.
- For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ by n_j, so that $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Expected value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.
- Assume that the hash function takes $O(1)$ time to compute, 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 independent uniform hashing applies. The theorems in the book spell out these assumptions.]

Unsuccessful search

Theorem
An unsuccessful search takes average-case time $\Theta(1 + \alpha)$.

Proof Independent 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 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 average-case 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 one more than the number of elements that appear before x in x's list. These are the elements inserted *after* x was inserted (because elements are inserted 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 $k_i = x_i.\mathit{key}$.

For each slot q and each pair of distinct keys k_i and k_j, define indicator random variable $X_{ijq} = \mathrm{I}\{\text{the search is for } x_i \text{ and } h(k_i) = h(k_j) = q\}$. $X_{ijq} = 1$ when k_i and k_j collide at slot q and the search is for k_i.

$\Pr\{\text{the search is for } x_i\} = 1/n$,
$\Pr\{h(k_i) = q\} = 1/m$,
$\Pr\{h(k_j) = q\} = 1/m$,
and these events are all independent
$\Rightarrow \Pr\{X_{ijq} = 1\} = 1/nm^2$
$\Rightarrow \mathrm{E}[X_{ijq}] = 1/nm^2$ (by Lemma 5.1).

For each element x_j, define indicator random variable

$$\begin{aligned} Y_j &= \mathrm{I}\{x_j \text{ appears in a list prior to the element being searched for}\} \\ &= \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq} \; . \end{aligned}$$

At most one of the X_{ijq} equals 1, which occurs when x_i is being searched for, x_j is in the same list as x_i, slot q points to this list, and $i < j$ so that x_j appears before x_i in the list.

One more random variable: $Z = \sum_{j=1}^{n} Y_j$ counts how many elements appear in the list prior to the element being searched for. Counting the element being searched for plus all the elements appearing before it in its list, we want $\mathrm{E}[Z + 1]$:

$$\begin{aligned} \mathrm{E}[Z + 1] &= \mathrm{E}\left[1 + \sum_{j=1}^{n} Y_j\right] \\ &= 1 + \mathrm{E}\left[\sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}\right] \end{aligned}$$

$$
\begin{aligned}
&= 1 + \mathrm{E}\left[\sum_{q=0}^{m-1}\sum_{j=1}^{n}\sum_{i=1}^{j-1} X_{ijq}\right] \\
&= 1 + \sum_{q=0}^{m-1}\sum_{j=1}^{n}\sum_{i=1}^{j-1} \mathrm{E}\left[X_{ijq}\right] \qquad \text{(linearity of expectation)} \\
&= 1 + \sum_{q=0}^{m-1}\sum_{j=1}^{n}\sum_{i=1}^{j-1} \frac{1}{nm^2} \\
&= 1 + m \cdot \binom{n}{2} \cdot \frac{1}{nm^2} \\
&= 1 + \frac{n(n-1)}{2} \cdot \frac{1}{nm} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \, .
\end{aligned}
$$

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 hash-function properties and several ways to design hash functions.

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of independent uniform hashing: each key is equally likely to hash to any of the m slots, independent of any other key.
- In practice, it's not possible to satisfy this assumption, since you don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- If you know the distribution of keys, you can take advantage of it.

Example: If keys are random real numbers independently and uniformly distributed in the half-open interval $[0, 1)$, then can use $h(k) = \lfloor km \rfloor$.

- We'll see "static hashing," which uses a single fixed hash function. And "random hashing," which chooses a hash function at random from a family of hash functions. With random hashing, don't need to know the probability distribution of the keys. Instead, the randomization is in the choice of hash function. We recommend random hashing.

Keys are integers, vectors, or strings

In practice, hash functions assume that the keys are either

- A short nonnegative integer that fits in a machine word (typically 32 or 64 bits), or
- A short vector of nonnegative integers, each of bounded size, e.g., a string of bytes.

For now, assume that keys are short nonnegative integers. We'll look at keys as vectors later.

Static hashing

A single, fixed hash function. Randomization comes only from the hoped-for distribution of the keys.

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.

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, $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$, where $kA \bmod 1 = kA - \lfloor kA \rfloor =$ fractional part of kA.

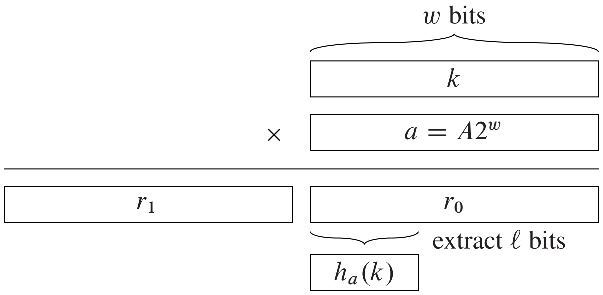
Disadvantage: Slower than division method.

Advantage: Value of m is not critical. Can choose it independently of A.

Multiply-shift method

A special case of the multiplication method.

- Let the word size of the machine be w bits.
- Set $m = 2^\ell$ for some integer $\ell \le w$.
- Assume that the key k fits into a single word. (k takes w bits.)
- Choose a fixed w-bit positive integer $a = A\,2^w$ in the range $0 < a < 2^w$.



- Multiply k by a.
- Multiplying two w-bit words ⇒ the result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order w-bit word of the product and r_0 is the low-order w-bit word of the product.
- Hash value is $h_a(k) = \ell$ most significant bits of r_0. Since $\ell \le w$, don't need the r_1 part of the product. Need only r_0.
- Define the operator $\ggg$ as logical right shift, so that $x \ggg b$ shifts x right by b bits, filling in the vacated positions on the left with zeros. Then $h_a(k) = (ka \bmod 2^w) \ggg (w - \ell)$. Here, $ka \bmod 2^w$ zeroes out r_1 (the high-order w bits of $ka = kA\,2^w$), and shifting right by $w - \ell$ bits moves the ℓ most significant bits of r_0 into the ℓ rightmost positions (same as dividing by $2^{w-\ell}$ and taking the floor of the result).
- Need only three machine instructions to compute $h_a(k)$: multiply, subtract, logical right shift.
- ***Example:*** $k = 123456$, $\ell = 14$, $m = 2^{14} = 16384$, $w = 32$. Choose $a = 2654435759$. Then $ka = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$ $\Rightarrow r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 give $h_a(k) = 67$.

Multiply-shift is fast, but doesn't guarantee good average-case performance. Can get good average-case performance by picking a as a randomly chosen odd integer.

Random hashing

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 they could choose keys that all hash to the same slot, giving worst-case behavior. Any static hash function is vulnerable to this type of attack.

One way to defeat the adversary is to choose a hash function randomly independent of the keys. We describe a special case, ***universal hashing***, which can yield provably good performance average when collisions are resolved by chaining, no matter the keys.

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 $k_1, k_2 \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k_1) = h(k_2)$ is $\leq |\mathcal{H}| / m$.

In other words, $\mathcal{H}$ 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 the $1/m$ chance of just choosing two slots randomly and independently.

Corollary to the previous theorem on average-case time for a successful search with chaining:

Corollary

Using chaining and universal hashing and starting with an initially empty table with m slots, the expected time is $\Theta(s)$ to handle any sequence of s INSERT, SEARCH, or DELETE operations with $n = O(m)$ INSERT operations.

Proof INSERT and DELETE take constant time (recall: DELETE has a pointer to the element to delete, so no search required as part of DELETE).

$n = O(m) \Rightarrow \alpha = O(1)$. Expected time for each SEARCH is $O(1)$, because the proof of the theorem depended only on the collision probabilities, which rely on the independent uniform hashing assumption. A universal hash function fulfills this assumption. Since each search has expected time $O(1)$, linearity of expectation gives that the expected time for any sequence of s operations is $O(s)$.

Each operation takes $\Omega(1)$ time $\Rightarrow$ entire sequence takes $\Omega(s)$ time $\Rightarrow \Theta(s)$. ■

Achievable properties of random hashing

Families of hash functions may exhibit any of several properties. Consider a family $\mathcal{H}$ of hash functions over domain U and with range $\{0, 1, \ldots, m - 1\}$, keys in U, slot numbers in $\{0, 1, \ldots, m - 1\}$, and a hash function h picked randomly from $\mathcal{H}$. The following properties may pertain to $\mathcal{H}$:

Uniform: The probability over picks of h that $h(k) = q$ is $1/m$.

Universal: For any distinct keys k_1, k_2, the probability that $h(k_1) = h(k_2)$ is at most $1/m$.

ϵ-universal: For any distinct keys k_1, k_2 the probability that $h(k_1) = h(k_2)$ is at most ϵ (so that universal means $1/m$-universal).

d-independent: For any distinct keys $k_1, k_2, \ldots, k_d$ and any slots $q_1, q_2, \ldots, q_d$, not necessarily distinct, the probability that $h(k_i) = q_i$ is $1/m^d$.

Designing a universal family of hash functions

[The book covers two methods. One is based on number theory and, given number-theoretic properties, is more easily proved universal. The other is a randomized variant of the multiply-shift method and is faster in practice.]

Based on number theory

- Choose a prime number p large enough so that every possible key is in the set $\{0, 1, \ldots, p-1\}$. Assume that $p > m$ (otherwise, just use direct addressing). m need not be prime.
- Denote $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$.
- Given any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$, define

  $$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \ .$$

  Example: $p = 17, m = 6, a = 3, b = 4 \Rightarrow$

  $$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5 \ . \end{aligned}$$

- Given p and m, the family of hash functions is

  $$\mathcal{H}_{pm} = \left\{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\right\} \ .$$

  Each maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$.
- This family of hash functions contains $p(p-1)$ functions, one for each combination of a and b.

Theorem
The family $\mathcal{H}_{pm}$ of hash functions defined above is universal.

Proof Let $k_1, k_2 \in \mathbb{Z}_p$, $k_1 \neq k_2$. For hash function h_{ab}, let $r_1 = (ak_1 + b) \bmod p$, $r_2 = (ak_2 + b) \bmod p$.

Must have $r_1 \neq r_2$. That's because $r_1 - r_2 = a(k_1 - k_2) \pmod{p}$, p is prime, both a and $(k_1 - k_2)$ are nonzero modulo $p \Rightarrow a(k_1 - k_2) \neq 0 \pmod{p}$. Therefore, distinct k_1, k_2 map to different values r_1, r_2 modulo p. (No collisions yet at the "mod p level.")

Each of the $p(p-1)$ choices for a, b ($a \neq 0$) yields a different pair $r_1 \neq r_2$. That is because can solve for a, b given r_1, r_2:

$$\begin{aligned} a &= \big((r_1 - r_2)((k_1 - k_2)^{-1} \bmod p)\big) \bmod p \ , \\ b &= (r_1 - ak_1) \bmod p \ . \end{aligned}$$

$((k_1 - k_2)^{-1} \bmod p)$ is the unique multiplicative inverse, modulo p, of $k_1 - k_2$.

Each of the p possible values of r_1 has only $p - 1$ possible values of r_2 that differ from r_1
$\Rightarrow$ only $p(p-1)$ possible pairs r_1, r_2 with $r_1 \neq r_2$
$\Rightarrow$ 1-1 correspondence between pairs a, b with $a \neq 0$ and pairs r_1, r_2 with $r_1 \neq r_2$
$\Rightarrow$ if a, b are picked uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$ and $k_1 \neq k_2$, then r_1, r_2 are equally likely to be any pair of distinct values modulo p.

The probability that $k_1 \neq k_2$ collide equals the probability that $r_1 = r_2 \pmod{m}$ when r_1, r_2 are randomly chosen as distinct values modulo p. For a given value of r_1, of the $p-1$ possible values of r_2, the number for which $r_2 \neq r_1$ and $r_2 = r_1 \pmod{m}$ is at most

$$\begin{aligned} \left\lceil \frac{p}{m} \right\rceil - 1 &\leq \frac{p+m-1}{m} - 1 \quad \text{(inequality (3.7))} \\ &= \frac{p-1}{m} . \end{aligned}$$

r_2 is one of these $p-1$ possible values $\Rightarrow$ the probability that r_2 collides with r_1 when computed modulo m is $\leq ((p-1)/m)/(p-1) = 1/m$. Therefore, for $k_1, k_2 \in \mathbb{Z}_p$ with $k_1 \neq k_2$, $\Pr\{h_{ab}(k_1) = h_{ab}(k_2)\} \leq 1/m$, and $\mathcal{H}_{pm}$ is universal. ■

Based on multiply-shift

A hash function family that is $2/m$-universal and very efficient in practice:

$\mathcal{H} = \{h_a : a \text{ is odd}, 1 \leq a < m, \text{and } h_a(k) = (ka \bmod 2^w) \ggg (w - \ell)\}$.

[Proof omitted from the book and these notes.]

Although this family has a higher probability of collision than the number-theory based family, the speed of computing the hash function based on multiply-shift is usually worth the extra collisions.

Long inputs such as vectors or strings

[The book does not get particularly specific on how to handle long key values.]

It is possible to extend the universal family of hash functions based on number theory to handle long key values. Another way is with cryptographic hash functions, which take as input an arbitrary byte string and return a fixed-length output. Can then take that output modulo m as the hash function value.

Open addressing

An alternative to chaining for handling collisions.

Idea

- Store all elements in the hash table itself.
- Each slot contains either an element or NIL.
- The hash table can fill up, but the load factor can never be > 1.
- How to handle collisions during insertion:
  - Determine the element's "first-choice" location in the hash table.
  - If the first-choice location is unoccupied, put the element there.
  - Otherwise, determine the element's "second-choice" location. If unoccupied, put the element there.

  - Otherwise, try the "third-choice" location. And so on, until an unoccupied location is found.
  - Different elements have different preference orders.
- How to search:
  - Same idea as for insertion.
  - But upon finding an unoccupied slot in the hash table, conclude that the element being searched for is not present.
- Open addressing avoids the pointers needed for chaining. You can use the extra space to make the hash table larger.

More specifically, 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.
- If slot $h(k)$ contains a key that is not k, compute the index of some other slot, based on k and on which probe (count from 0: 0th, 1st, 2nd, etc.).
- Keep probing until either find key k (successful search) or find a slot holding NIL (unsuccessful search).
- Need the sequence of slots probed to be a permutation of the slot numbers $\langle 0, 1, \ldots, m-1 \rangle$ (so that all slots are examined if necessary, and so that no slot is examined more than once).
- Thus, the hash function is $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$.
- 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 searching, and insert at the first NIL slot encountered.

Pseudocode for insertion

HASH-INSERT either returns the slot number where the new key k goes or flags an error because the table is full.

```
HASH-INSERT(T, k)
  i = 0
  repeat
      q = h(k, i)
      if T[q] == NIL
          T[q] = k
          return q
      else i = i + 1
  until i == m
  error "hash table overflow"
```

Pseudocode for searching

HASH-SEARCH returns either the slot number where the key k resides or NIL if key k is not in the table.

```
HASH-SEARCH(T, k)
  i = 0
  repeat
      q = h(k, i)
      if T[q] == k
          return q
      i = i + 1
  until T[q] == NIL or i == m
  return NIL
```

Deletion

Cannot just put NIL into the slot containing the key to be deleted.

- Suppose key k in slot q is inserted.
- And suppose that sometime after inserting key k, key k' was inserted into slot q', and during this insertion slot q (which contained key k) was probed.
- And suppose that key k was deleted by storing NIL into slot q.
- And then a search for key k' occurs.
- The search would probe slot q *before* probing slot q', which contains key k'.
- 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.

A simple special case of open addressing, called linear probing, avoids having to mark slots with DELETED. *[Deletion with linear probing will be covered later in this chapter.]*

How to compute probe sequences

The ideal situation is ***independent uniform permutation hashing*** (also known as ***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 independent uniform hashing for a hash function that produces a whole probe sequence rather than just a single number.)

It's hard to implement true independent uniform permutation hashing. Instead, 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. Double hashing can generate at most m^2 probe sequences, and linear probing can generate only m. They will make use of ***auxiliary hash functions***, which map $U \rightarrow \{0, 1, \ldots, m-1\}$.

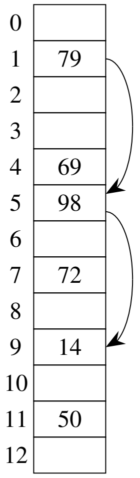
Double hashing

Uses auxiliary hash functions h_1, h_2 and probe number i:

$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$.

The first probe goes to slot $h_1(k)$ (because i starts at 0). Successive probes are offset from previous slots by $h_2(k)$ modulo $m \Rightarrow$ the probe sequence depends on the key in two ways.

Example: $m = 13$, $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$, inserting key $k = 14$. First probe is to slot 1 (14 mod 13 = 1), which is occupied. Second probe is to slot 5 (((14 mod 13)+(1+(14 mod 11))) mod 13 = (1+4) mod 13 = 5), which is occupied. Third probe is to slot 9 (offset from slot 5 by 4), which is free, so key 14 goes there.



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.
- Could let m be prime and have $1 < h_2(k) < m$.

  Example: $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$ (as in the example above, with $m = 13$, $m' = 11$).

Linear probing

Special case of double hashing.

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.

[Will revisit linear probing later in the chapter and in these notes.]

Analysis of open-address hashing

Assumptions

- Analysis is in terms of load factor α. Assume that the table never completely fills, so always have $0 \le n < m \Rightarrow 0 \le \alpha < 1$.
- Assume independent uniform permutation 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 $1/(1-\alpha)$.

Intuition behind the proof: The first probe always occurs. The first probe finds an occupied slot with probability (approximately) α, so that a second probe happens. With probability (approximately) α^2, the first two slots are occupied, so that a third probe occurs. Get the geometric series $1 + \alpha + \alpha^2 + \alpha^3 + \cdots = 1/(1-\alpha)$ (since $\alpha < 1$).

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.2-5,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\qquad \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} .\end{aligned}$$

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 of claim 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 independent uniform permutation 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 has not yet been 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,

$$\Pr\{X \ge i\} = \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}}_{i-1 \text{ factors}} .$$

$n < m \Rightarrow (n-j)/(m-j) \le n/m$ for $j \ge 0$, which implies

$$\begin{aligned}
\Pr\{X \ge i\} &\le \left(\frac{n}{m}\right)^{i-1} \\
&= \alpha^{i-1} .
\end{aligned}$$

By equation (C.28),

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \ge i\} \\
&= \sum_{i=1}^{n+1} \Pr\{X \ge i\} + \sum_{i>n+1} \Pr\{X \ge i\} \\
&\qquad\qquad \text{(the } (n+1)\text{st probe must be to an empty slot)} \\
&\le \sum_{i=1}^{\infty} \alpha^{i-1} + 0 \\
&= \sum_{i=0}^{\infty} \alpha^{i} \\
&= \frac{1}{1-\alpha} . \qquad \blacksquare \text{ (theorem)}
\end{aligned}$$

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 $\dfrac{1}{\alpha} \ln \dfrac{1}{1-\alpha}$.

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:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i}$$

$$
\begin{aligned}
&= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} \frac{1}{k} \\
&\le \frac{1}{\alpha} \int_{m-n}^{m} (1/x)\,dx \qquad \text{(by inequality (A.19))} \\
&= \frac{1}{\alpha} \ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \qquad \blacksquare
\end{aligned}
$$

Practical considerations

Modern CPUs have features that affect hashing:

Memory hierarchies: ***Caches*** are small, fast memory units closer to where instructions execute. They are organized in ***cache lines*** of a specific size (e.g., 64 bytes) of contiguous bytes from main memory. It's much faster to reuse a cache line than to fetch from main memory. Iterating through a cache line is relatively fast.

Advanced instruction sets: Advanced primitives for encryption and cryptography can also be used to compute hash functions.

Linear probing

Linear probing performs poorly in the standard RAM model, but it does well in the presence of hierarchical memory because successive probes are likely to be within the same cache line.

Deletion

Don't need to use the DELETED marker with linear probing—can actually delete an element from the hash table.

Uses inverse of the hash function for linear probing: $g(k, q)$ takes the key k and slot number q and returns the probe number for slot q when searching for key k: $h(k, i) = q \Rightarrow g(k, q) = i$.

$$
\begin{aligned}
h(k, i) &= (h_1(k) + i) \bmod m \,, \\
g(k, q) &= (q - h_1(k)) \bmod m \,.
\end{aligned}
$$

$h(k, g(k, q)) = q$.

```
LINEAR-PROBING-HASH-DELETE$(T, q)$
  while TRUE
      $T[q] = $ NIL                          // make slot $q$ empty
      $q' = q$                               // starting point for search
      repeat
          $q' = (q' + 1) \bmod m$            // next slot number with linear probing
          $k' = T[q']$                       // next key to try to move
          if $k'$ == NIL
              return                         // return when an empty slot is found
      until $g(k', q) < g(k', q')$           // was empty slot $q$ probed before $q'$?
      $T[q] = k'$                            // move $k'$ into slot $q$
      $q = q'$                               // free up slot $q'$
```

How it works: First, deletes the key in position q by setting $T[q]$ to NIL. Then searches for a slot q' containing a key that should be moved to the slot just vacated. The test $g(k', q) < g(k', q')$ asks whether the key k' in slot q' needs to be moved to the previously vacated slot q to preserve that k' can be found.

If $g(k', q) < g(k', q')$, then when k' was inserted, slot q was checked and found to be occupied, but now slot q is empty. Move k' there and continue with q being the slot that k' had been in.

Example: $m = 10$, $h_1(k) = k \bmod 10$.
Left: After inserting, in order, 74, 43, 93, 18, 82, 38, 92.
Right: After deleting 43 from slot 3. 93 moves up to slot 3, 92 moves up to slot 5 where 93 had been.

0		0	
1		1	
2	82	2	82
3	43	3	93
4	74	4	74
5	93	5	92
6	92	6	
7		7	
8	18	8	18
9	38	9	38

Analysis of linear probing

Linear probing exhibits ***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.

Primary clustering slows things down in the RAM model, but it helps in hierarchical memory because searching stays in the same cache line for as long as possible.

Theorem
If h_1 is 5-independent and $\alpha \leq 2/3$, then it takes expected constant time to search for, insert, or delete a key in a hash table using linear probing. ■

[Proof omitted in the book and these notes.]

[The book contains a starred subsection on hash functions for hierarchical memory models. It is omitted from these notes.]

Solutions for Chapter 11: Hash Tables

Solution to Exercise 11.1-1

The obvious way is to scan the entire table from m down to 1, looking for a non-NIL entry. Finding the maximum element takes $\Theta(m)$ time in the worst case.

You can reduce the time to find the maximum down to $\Theta(1)$ by also maintaining the maximum key as a separate table attribute. You might also have to update the maximum key upon an insertion or deletion operation. For insertion, if the key being inserted is greater than the maximum, then the new key becomes the maximum, but this extra work costs only $\Theta(1)$. For deletion, if the maximum key is being deleted, then you need to scan from that entry down toward 1 to find the next highest key value, taking $\Theta(m)$ time in the worst case. In other words, this idea moves the operation taking $\Theta(m)$ time in the worst case from finding the maximum to deletion.

Solution to Exercise 11.1-2

Bit i is 1 if i is in the set, 0 if i is not in the set. To insert i, just set bit i to 1. To delete i, set bit i to 0. To search for i, just return whether bit i is 1.

Solution to Exercise 11.1-3

Store elements with the same key k in a doubly linked list pointed to by $T[k]$. To insert an element with key k, insert at the head of the list that $T[k]$ points to. To search for an element with key k, just return the the head of the list that $T[k]$ points to (since any element with key k suffices). To delete, just delete the element from its list. Because each list is doubly linked, all three operations take $O(1)$ time.

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 $S.top$, so that only entries $S[1:S.top]$ 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 $1 \le T[k] \le S.top$, $S[T[k]] = k$, 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 $S.top = 0$, so that there are no valid entries in the stack.
- SEARCH: Given key k, check whether key k is in a validating cycle, i.e., whether $1 \le T[k] \le S.top$ and $S[T[k]] = k$. If so, return $S'[T[k]]$; otherwise, return NIL.
- INSERT: To insert object x with key k, assuming that this object is not already in the dictionary, increment $S.top$, set $S[S.top] = k$, set $S'[S.top] = x$, and set $T[k] = S.top$.
- 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 being vacated—and then fixing up *that* entry's validating cycle. That is, execute the following sequence of assignments:

$$
\begin{aligned}
&S[T[k]] = S[S.top] \\
&S'[T[k]] = S'[S.top] \\
&T[S[T[k]]] = T[k] \\
&T[k] = \text{NIL} \\
&S.top = S.top - 1
\end{aligned}
$$

Each of these operations—initialization, SEARCH, INSERT, and DELETE—takes $O(1)$ time.

Solution to Exercise 11.2-1

This solution is also posted publicly

For each pair of keys k, l, where $k \ne l$, define the indicator random variable $X_{kl} = \text{I}\{h(k) = h(l)\}$. Since we assume independent uniform hashing, $\Pr\{X_{kl} = 1\} = \Pr\{h(k) = h(l)\} = 1/m$, and so $\text{E}[X_{kl}] = 1/m$.

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 11.2-3

Keeping the lists in sorted order does not help much. We cannot use binary search on a linked list to speed up searching. The time for a successful search, therefore, does not change. An unsuccessful search for key k can terminate once an element whose value is greater than k is found. Insertion can now take as long as the length of the list, instead of $O(1)$. Deletion is unchanged.

Solution to Exercise 11.2-4
This solution is also posted publicly

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, x 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 11.2-5

If $|U| = (n-1)m$, then the only way to avoid some slot having n keys is for every slot to have exactly $n-1$ keys. Adding one more key means that some slot will have n keys.

Solution to Exercise 11.2-6

We can view the hash table as if it had m rows and L columns; each row stores one chain. This imaginary array has mL entries storing n keys, and $mL - n$ empty values. Suppose that we also have an array $\mathit{length}[0:m-1]$ giving the length of each chain. Randomly pick a row $i \in \{0, \ldots, m-1\}$ and a column $j \in \{1, \ldots, L\}$ until $j \leq \mathit{length}[i]$, so that i and j represent an element that is in the jth position of the chain for slot i. Then, go to the chain in slot i, traverse j places down the chain, and return the key in the jth position.

We can view the process of selecting row i and column j by a geometric distribution with the probability of success as $n/mL = \alpha/L$. By equation (C.36), the expected number of trials is L/α. Then $O(L)$ time is needed to traverse the chain to find the element, giving an expected time of $O(L \cdot (1 + 1/\alpha))$.

Solution to Exercise 11.3-1

Use the hash value of each element as a quick way to reject it. That is, if the hash value of an element does not equal the hash value of key k, then reject the element. If the hash values are equal, then compare the strings in the element and the key.

Solution to Exercise 11.3-2

Apply the division method's hash function $h(k) = k \bmod m$ on one character at a time, accumulating the values. That is, for a string $s = \langle s_1, \ldots, s_r \rangle$, compute $h(s) = \sum_{i=1}^{r}(s_i \bmod m)$.

Solution to Exercise 11.3-3

First, we observe that we can generate any permutation by a sequence of interchanges of pairs of characters. It's possible to 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

$$\begin{aligned}
&(h(x) - h(y)) \bmod (2^p - 1) \\
&\quad = ((x_a 2^{ap} + x_b 2^{bp}) - (y_a 2^{ap} + y_b 2^{bp})) \bmod (2^p - 1) \\
&\quad = ((x_a 2^{ap} + x_b 2^{bp}) - (x_b 2^{ap} + x_a 2^{bp})) \bmod (2^p - 1) \\
&\quad = ((x_a - x_b) 2^{ap} - (x_a - x_b) 2^{bp}) \bmod (2^p - 1)
\end{aligned}$$

$$\begin{aligned} &= ((x_a - x_b)(2^{ap} - 2^{bp})) \bmod (2^p - 1) \\ &= ((x_a - x_b)2^{bp}(2^{(a-b)p} - 1)) \bmod (2^p - 1) \; . \end{aligned}$$

By equation (A.6),

$$\sum_{i=0}^{a-b-1} 2^{pi} = \frac{2^{(a-b)p} - 1}{2^p - 1} \; ,$$

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,

$$\begin{aligned} &(h(x) - h(y)) \bmod (2^p - 1) \\ &= \left((x_a - x_b)2^{bp}\left(\sum_{i=0}^{a-b-1} 2^{pi}\right)(2^p - 1)\right) \bmod (2^p - 1) \\ &= 0 \; , \end{aligned}$$

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 $q = |Q|$ and $u = |U|$. We start by showing that the total number of collisions is minimized by a hash function that maps u/q elements of U to each of the q values in Q. For a given hash function, let u_j be the number of elements that map to $j \in Q$. We have $u = \sum_{j \in Q} u_j$. We also have that the number of collisions for a given value of $j \in Q$ is $\binom{u_j}{2} = u_j(u_j - 1)/2$.

Lemma

The total number of collisions is minimized when $u_j = u/q$ for each $j \in Q$.

Proof If $u_j \le u/q$, let us call j ***underloaded***, and if $u_j \ge u/q$, let us call j ***overloaded***. Consider an unbalanced situation in which $u_j \ne u/q$ for at least one value $j \in Q$. We can think of converting a balanced situation in which all u_j equal u/q 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 Q as representing buckets, we are repeatedly moving elements from buckets containing at most u/q elements to buckets containing at least u/q 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, $u_j \le u/q \le u_k$. 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 $u_j = u/q$ for each $j \in Q$. ■

By the above lemma, for any hash function, the total number of collisions must be at least $q(u/q)(u/q - 1)/2$. 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

$$\begin{aligned}
\frac{q(u/q)(u/q-1)/2}{u(u-1)/2} &= \frac{u/q - 1}{u - 1} \\
&> \frac{u/q - 1}{u} \\
&= \frac{1}{q} - \frac{1}{u} \,.
\end{aligned}$$

Thus, the bound ϵ on the probability of a collision for any pair of distinct elements can be no less than $1/q - 1/u = 1/|Q| - 1/|U|$.

Solution to Exercise 11.4-3

By Theorem 11.6, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$. This quantity equals 4 when $\alpha = 3/4$, and it equals 8 when $\alpha = 7/8$.

Theorem 11.8 bounds the expected number of probes in a successful search by $(1/\alpha)\ln(1/(1-\alpha))$. This quantity equals $1.8483\ldots$ when $\alpha = 3/4$, and it equals $2.376\ldots$ when $\alpha = 7/8$.

Solution to Problem 11-1

a. Since we assume independent uniform permutation 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 $\Pr\{X \geq i\} \leq \alpha^{i-1}$. Since $n \leq m/2$, we have $\alpha \leq 1/2$. Letting $i = p + 1$, we have $\Pr\{X > p\} = \Pr\{X \geq p + 1\} \leq (1/2)^{(p+1)-1} = 2^{-p}$.

b. Substituting $p = 2\lg n$ into the statement of part (a) yields that the probability that the ith insertion requires more than $p = 2\lg n$ probes is at most $2^{-2\lg n} = (2^{\lg n})^{-2} = n^{-2} = 1/n^2$.

We must deal with the possibility that $2\lg n$ is not an integer, however. Then the event that the ith insertion requires more than $2\lg n$ probes is the same as the event that the ith insertion requires more than $\lfloor 2\lg n \rfloor$ probes. Since $\lfloor 2\lg n \rfloor > 2\lg n - 1$, we have that the probability of this event is at most $2^{-\lfloor 2\lg n \rfloor} < 2^{-(2\lg n - 1)} = 2/n^2 = O(1/n^2)$.

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 $\Pr\{A_i\} = O(1/n^2)$ 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.21), we have

$$\begin{aligned}
\Pr\{A\} &\le \Pr\{A_1\} + \Pr\{A_2\} + \cdots + \Pr\{A_n\} \\
&\le n \cdot O(1/n^2) \\
&= O(1/n) .
\end{aligned}$$

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 $\sum_{k=\lceil 2\lg n \rceil + 1}^{n} \Pr\{X = k\} \le \Pr\{X > 2\lg n\} = O(1/n)$, by part (c). Therefore,

$$\begin{aligned}
\mathrm{E}[X] &\le \lceil 2\lg n \rceil \cdot 1 + n \cdot O(1/n) \\
&= \lceil 2\lg n \rceil + O(1) \\
&= O(\lg n) .
\end{aligned}$$

Solution to Problem 11-2

a. Store the n elements in an array. Preprocess by using an in-place sorting algorithm to sort the elements in the array. Either insertion sort or heapsort works for preprocessing. Then, SEARCH is just binary search, taking $O(\lg n)$ worst-case time.

b. Recall that the average unsuccessful search time for independent uniform permutation hashing is no more than $1/(1-\alpha)$, where $\alpha = n/m$ is the load factor of the hash table. We require

$$\frac{1}{1-n/m} \le c \lg n$$

for some positive constant c, which is equivalent to

$$m \ge \frac{cn \lg n}{c \lg n - 1} .$$

We now bound $m - n$ from below by

$$\begin{aligned} m - n &\ge \frac{cn \lg n}{c \lg n - 1} - n \\ &= \frac{n}{c \lg n - 1} \\ &= \Omega(n / \lg n) . \end{aligned}$$

Solution to Problem 11-3

This solution is also posted publicly

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,

$$\begin{aligned} P_k &= \Pr\{M = k\} \\ &= \Pr\{\max\{X_i : 1 \le i \le n\} = k\} \\ &= \Pr\{\text{there exists } i \text{ such that } X_i = k \text{ and that } X_i \le k \text{ for } i = 1, 2, \ldots, n\} \\ &\le \Pr\{\text{there exists } i \text{ such that } X_i = k\} \\ &= \Pr\{A_1 \cup A_2 \cup \cdots \cup A_n\} \\ &\le \Pr\{A_1\} + \Pr\{A_2\} + \cdots + \Pr\{A_n\} \qquad \text{(by inequality (C.21))} \\ &= nQ_k . \end{aligned}$$

c. We start by showing two facts. First, $1 - 1/n < 1$ and $n - k \ge 0$, which imply that $(1 - 1/n)^{n-k} \le 1$. 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 (3.25), we have

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \frac{n!}{k!(n-k)!}$$

$$
\begin{aligned}
&\le \frac{n!}{n^k k!(n-k)!} && ((1-1/n)^{n-k} < 1) \\
&< \frac{1}{k!} && (n!/(n-k)! < n^k) \\
&< \frac{e^k}{k^k} && (k! > (k/e)^k) \ .
\end{aligned}
$$

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
- $\max\{c : 3 < cx \text{ and } 3 \le n < n_0\}$, 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,

$$
Q_k < e^k/k^k
$$

$$
\begin{aligned}
&\le e^{k_0}/k^{k_0} \\
&= Q_{k_0} \\
&< 1/n^3
\end{aligned}
$$

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

$$
\begin{aligned}
\mathrm{E}[M] &\le k_0 \cdot 1 + n \cdot (1/n) \\
&= k_0 + 1 \\
&= O(\lg n / \lg\lg n) .
\end{aligned}
$$

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.

[Previous editions contained a section analyzing the height of a randomly built binary search tree. This section has been removed in the fourth edition.]

Binary search trees

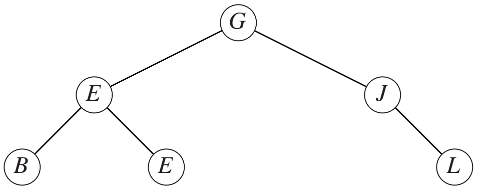
Binary search trees are an important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where $h =$ height of tree.
- As in Section 10.3, represent a binary tree by a linked data structure in which each node is an object.
- $T.root$ points to the root of tree T.
- Each node contains the attributes
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - p: points to parent. $T.root.p = \text{NIL}$.

- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in left subtree of x, then $y.key \leq x.key$.
  - If y is in right subtree of x, then $y.key \geq x.key$.

Draw sample tree.

[Using alphabetic order for comparison. It's OK to have duplicate keys. 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
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

Example

Do the inorder tree walk on the example above, getting the output $B\ E\ E\ G\ J\ L$.

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 it visits and prints each node once. *[Book has formal proof.]*

Querying a binary search tree

Searching

```
TREE-SEARCH$(x, k)$
  if $x$ == NIL or $k$ == $key[x]$
      return $x$
  if $k < x.key$
      return TREE-SEARCH$(x.left, k)$
  else return TREE-SEARCH$(x.right, k)$
```

Initial call is TREE-SEARCH$(T.root, k)$.

Example

Search for values E, L, F, and H 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.

Iterative version

The iterative version unrolls the recursion into a **while** loop. It's usually more efficient than the recursive version, since it avoids the overhead of recursive calls.

```
ITERATIVE-TREE-SEARCH$(x, k)$
  while $x \neq$ NIL and $k \neq x.key$
      if $k < x.key$
          $x = x.left$
      else $x = x.right$
  return $x$
```

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. In the following procedures, the parameter x is the root of a subtree. The first call has $x = T.root$.

```
TREE-MINIMUM$(x)$
  while $x.left \neq$ NIL
      $x = x.left$
  return $x$
```

```
TREE-MAXIMUM(x)
  while $x.right \neq$ NIL
      $x = x.right$
  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 $y.key$ is the smallest key $> x.key$. 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 x's successor is NIL.

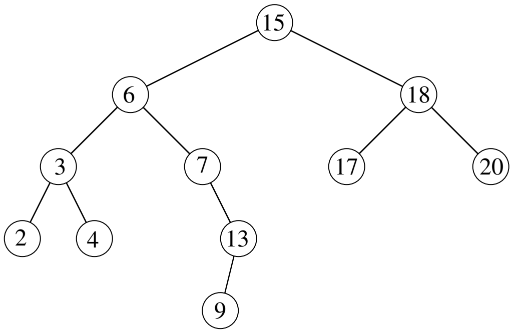
There are two cases:

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 $x.right \neq$ NIL
      return TREE-MINIMUM($x.right$)   // leftmost node in right subtree
  else // find the lowest ancestor of $x$ whose left child is an ancestor of $x$
      $y = x.p$
      while $y \neq$ NIL and $x == y.right$
          $x = y$
          $y = y.p$
      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

Both the TREE-SUCCESSOR and TREE-PREDECESSOR procedures 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 allow 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.

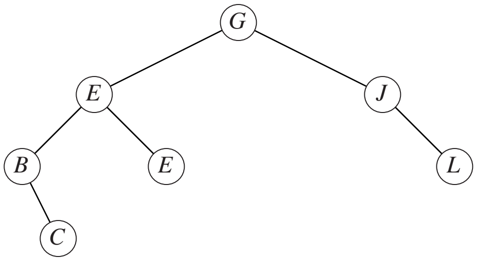
Insertion

```
TREE-INSERT(T, z)
  x = T.root              // node being compared with z
  y = NIL                 // y will be parent of z
  while x ≠ NIL           // descend until reaching a leaf
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y                 // found the location—insert z with parent y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

- To insert value v into the binary search tree, the procedure is given node z, with $z.key$ already filled in, $z.left = \text{NIL}$, and $z.right = \text{NIL}$.
- 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 $x.key$ with $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare $z.key$ with $y.key$, and insert z at either y's *left* or *right*, appropriately.

Example

Run TREE-INSERT(T, 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

[Deletion from a binary search tree changed in the third edition. In the first two editions, when the node z passed to TREE-DELETE *had two children, z's successor y was the node actually removed, with y's contents copied into z. The problem with that approach is that if there are external pointers into the binary search tree, then a pointer to y from outside the binary search tree becomes stale. In the third edition, the node z passed to* TREE-DELETE *is always the node actually removed, so that all external pointers to nodes other than z remain valid. The fourth edition keeps the same treatment from the third edition.]*

Conceptually, deleting node z from binary search tree T has three cases:

1. If z has no children, just remove it by changing z's parent to point to NIL instead of to z.
2. If z has just one child, then make that child take z's position in the tree by changing z's parent to point to z's child instead of to z, dragging the child's subtree along.
3. If z has two children, then find z's successor y and replace z by y in the tree. y must be in z's right subtree and have no left child. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.

   This case is a little tricky because the exact sequence of steps taken depends on whether y is z's right child.

The code organizes the cases a bit differently. Since it will move subtrees around within the binary search tree, it uses a subroutine, TRANSPLANT, to replace one subtree as the child of its parent by another subtree.

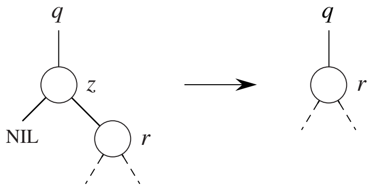
```
TRANSPLANT$(T, u, v)$
  if $u.p$ == NIL
      $T.root = v$
  elseif $u == u.p.left$
      $u.p.left = v$
  else $u.p.right = v$
  if $v \neq$ NIL
      $v.p = u.p$
```

TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v:

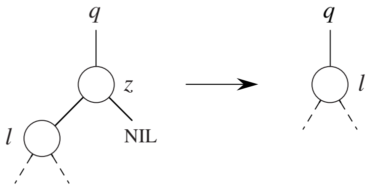
- Makes u's parent become v's parent (unless u is the root, in which case it makes v the root).
- u's parent gets v as either its left or right child, depending on whether u was a left or right child.
- Doesn't update $v.left$ or $v.right$, leaving that up to TRANSPLANT's caller.

TREE-DELETE(T, z) has four cases when deleting node z from binary search tree T:

- If z has no left child, replace z by its right child. The right child may or may not be NIL. (If z's right child is NIL, then this case handles the situation in which z has no children.)



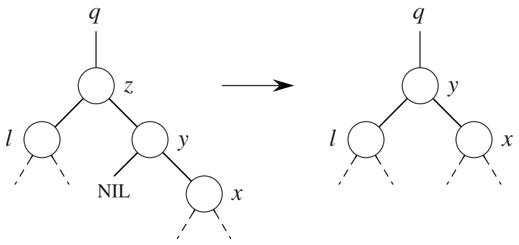
- If z has just one child, and that child is its left child, then replace z by its left child.



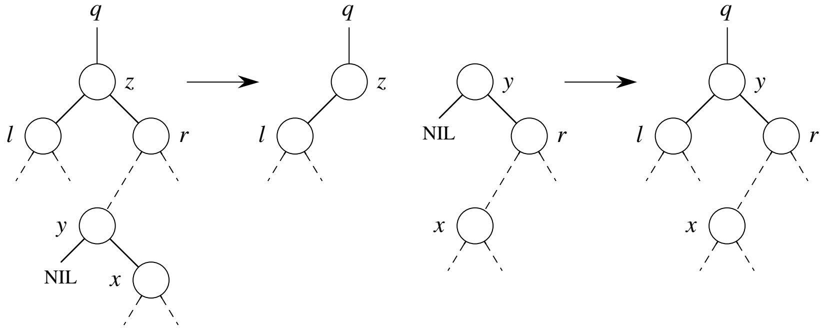
- Otherwise, z has two children. Find z's successor y. y must lie in z's right subtree and have no left child. (The solution to Exercise 12.2-5 on page 12-13 of this manual shows why.)

  Goal is to replace z by y, splicing y out of its current location.

  - If y is z's right child, replace z by y and leave y's right child alone.



- Otherwise, y lies within z's right subtree but is not the root of this subtree. Replace y by its own right child. Then replace z by y.

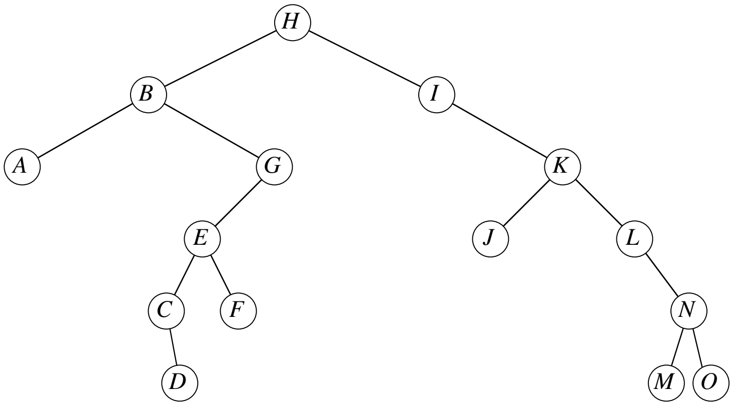


```
TREE-DELETE(T, z)
  if $z.left$ == NIL
      TRANSPLANT(T, z, z.right)          // replace $z$ by its right child
  elseif $z.right$ == NIL
      TRANSPLANT(T, z, z.left)           // replace $z$ by its left child
  else $y = $ TREE-MINIMUM($z.right$)    // $y$ is $z$'s successor
      if $y \neq z.right$                // is $y$ farther down the tree?
          TRANSPLANT(T, y, y.right)      // replace $y$ by its right child
          $y.right = z.right$            // $z$'s right child becomes
          $y.right.p = y$                //       $y$'s right child
      TRANSPLANT(T, z, y)                // replace $z$ by its successor $y$
      $y.left = z.left$                  // and give $z$'s left child to y,
      $y.left.p = y$                     //       which had no left child
```

Note that the last three lines execute when z has two children, regardless of whether y is z's right child.

Example

On this binary search tree T,



run the following. *[You can either start with the original tree each time or start with the result of the previous call. The tree is designed so that either way will elicit all four cases.]*

- TREE-DELETE(T, I) shows the case in which the node deleted has no left child.
- TREE-DELETE(T, G) shows the case in which the node deleted has a left child but no right child.
- TREE-DELETE(T, K) shows the case in which the node deleted has both children and its successor is its right child.
- TREE-DELETE(T, B) shows the case in which the node deleted has both children and its successor is not its right child.

Time

$O(h)$, on a tree of height h. Everything is $O(1)$ except for the call to TREE-MINIMUM.

[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 $O(n)$ that we can see in "plain" binary search trees. Red-black trees are covered in detail in Chapter 13.]

Solutions for Chapter 12: Binary Search Trees

Solution to Exercise 12.1-2

This solution is also posted publicly

In a heap, a node's key is greater than or equal to both of its children's keys. In a binary search tree, a node's key is greater than or equal to its left child's key, but less than or equal to its right child's key.

The heap property, unlike the binary-search-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 from Theorem 8.1 that a comparison sort must take $\Omega(n \lg n)$ time.

Solution to Exercise 12.1-3

[Except for the procedure name, the stack-based solution is the same as the solution to Exercise 10.3-3.]

The following nonrecursive inorder tree walk uses a stack that can hold as many nodes as necessary, so that the parameter for the stack size is omitted from calls to PUSH.

```
INORDER-TREE-WALK-NONRECURSIVE(T)
  S = empty stack
  x = T.root
  while x ≠ NIL
      PUSH(S, x)
      x = x.left
  while not IS-EMPTY(S)
      x = POP(S)
      print x.key
      x = x.right
      while x ≠ NIL
          PUSH(S, x)
          x = x.left
```

[The non-stack-based solution is similar to the solution to Exercise 10.3-5, but it prints at different times in the tree walk.]

```
INORDER-TREE-WALK-NONRECURSIVE(T)
  x = T.root
  y = NIL
  while x ≠ NIL
      z = x                    // save x to assign to y for next iteration
      if x.p == y
          // Coming from parent.
          if x.left ≠ NIL
              x = x.left
          else print x.key
              if x.right ≠ NIL
                  x = x.right
              else x = x.p
      elseif x.left == y
          print x.key          // coming from left child
          if x.right ≠ NIL:
              x = x.right
      else x = x.p             // coming from right child
      y = z                    // x, stored in z, becomes previous node next time
```

Solution to Exercise 12.1-4

```
PREORDER-TREE-WALK(x)
  if x ≠ NIL
      print x.key
      PREORDER-TREE-WALK(x.left)
      PREORDER-TREE-WALK(x.right)
```

```
POSTORDER-TREE-WALK($x$)
  if $x \neq$ NIL
      POSTORDER-TREE-WALK($x.left$)
      POSTORDER-TREE-WALK($x.right$)
      print $x.key$
```

Solution to Exercise 12.1-5

Because INORDER-TREE-WALK runs in $\Theta(n)$ time, if we could build a binary search tree by a comparison-based algorithm in $o(n \lg n)$ time in the worst case, then we could sort in $o(n \lg n)$ time in the worst case by building a binary search tree and then performing an inorder walk. But sorting takes $\Omega(n \lg n)$ time in the worst case. Therefore, any comparison-based algorithm to build a binary search tree must take $\Omega(n \lg n)$ time in the worst case.

Solution to Exercise 12.2-1

As search for key k goes down a binary search tree, keys less than k must monotonically increase and keys greater than k must monotonically decrease. Sequences (c) and (e) do not have this property. In sequence (c), the keys greater than 363 go from 911 to 912. In sequence (e) the keys less than 363 go from 347 to 299.

Each of the other three sequences obey the above rule:

- In (a), keys less than 363 go $2, 252, 330, 344$, and keys greater than 363 go $401, 398, 397$.
- In (b), keys less than 363 go $220, 244, 258, 362$, and keys greater than 363 go $924, 911, 898$.
- In (d), keys less than 363 go $2, 219, 266, 278$, and keys greater than 363 go $399, 387, 382, 381$.

Solution to Exercise 12.2-2

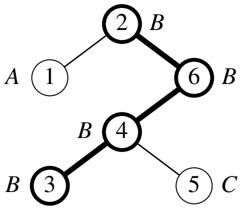
```
RECURSIVE-TREE-MINIMUM($x$)
  if $x.left$ == NIL
      return $x$
  else return RECURSIVE-TREE-MINIMUM($x.left$)

RECURSIVE-TREE-MAXIMUM($x$)
  if $x.right$ == NIL
      return $x$
  else return RECURSIVE-TREE-MAXIMUM($x.right$)
```

Solution to Exercise 12.2-3

```
TREE-PREDECESSOR(x)
  if $x.left \neq$ NIL
      return TREE-MAXIMUM($x.left$)
  else $y = x.p$
      while $y \neq$ NIL and $x == y.left$
          $x = y$
          $y = y.p$
      return $y$
```

Solution to Exercise 12.2-4

Search for 3 in this binary search tree:



The search path is highlighted. 5 ($\in C$) $<$ 6 ($\in B$).

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 12.2-6

We claim that y must be an ancestor of x. We prove this claim by contradiction. Suppose that y is not an ancestor of x. Since y is x's successor, it cannot be in x's left subtree, and since x's right subtree is empty, the only other possibility is that x and y have a lowest common ancestor z with x and y in different subtrees rooted at z. But then we would have $x < z < y$, so that y would not be x's successor.

Since y is an ancestor of x, it must be the case that x is in y's left subtree, which implies that $y.\mathit{left}$ must be an ancestor of x. Now, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x. Let z be this node. Since z is an ancestor of x, x is in y's left subtree, and z is lower than y, then z must also be in y's left subtree. Then it must be the case that $z < y$. Since z's left child is an ancestor of x, it must be the case that $x < z$. Thus, we have the contradiction that $x < z < y$, so that y is not x's successor.

Solution to Exercise 12.2-7
This solution is also posted publicly

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, the walk 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 looking for the successor of a node that has no right subtree.

Suppose that v is u's left child.

- Before printing u, the walk 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 12.2-8

Denote the starting node by x and its kth successor, where the series of calls to TREE-SUCCESSOR winds up, by y. Let node z be the lowest common ancestor of x and y. Since successive calls to TREE-SUCCESSOR perform an inorder walk, each node is visited at most three times. Either a node is one of the first k successors of x or it is not. If it is, then the time to visit it is $O(k)$. If not, then the node is either on the path from x to z or the path from z to y. The total lengths on these two paths is at most $2h$, so that the time spent visiting nodes not among the first k successors of x is $O(h)$. Thus, the total time for k successive calls to TREE-SUCCESSOR is $O(k + h)$.

Solution to Exercise 12.2-9

Suppose that x is y's left child. Starting at node y, call TREE-PREDECESSOR. The call returns x. Now suppose that x is y's right child and starting at node y, call TREE-SUCCESSOR. The call returns x.

Solution to Exercise 12.3-1

```
RECURSIVE-TREE-INSERT(T, z)
  if T.root == NIL
      T.root = z
      z.p = NIL
  else SUBTREE-INSERT(T.root, z)
```

```
SUBTREE-INSERT$(x, z)$
  if $z.key < x.key$
      if $x.left$ == NIL
          $x.left = z$
          $z.p = x$
      else SUBTREE-INSERT$(x.left, z)$
  else
      if $x.right$ == NIL
          $x.right = z$
          $z.p = x$
      else SUBTREE-INSERT$(x.right, z)$
```

Solution to Exercise 12.3-2

Inserting a key k into a binary search tree follows the same path down the tree as searching for k would follow. The difference is that after inserting, the node containing k is present in the tree, but during insertion that node is not yet present.

Solution to Exercise 12.3-3
This solution is also posted publicly

Here's the algorithm:

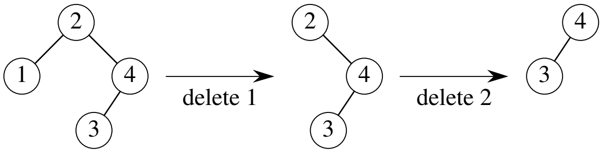
```
TREE-SORT$(A)$
  let $T$ be an empty binary search tree
  for $i = 1$ to $n$
      TREE-INSERT$(T, A[i])$
  INORDER-TREE-WALK$(T.root)$
```

Worst case: $\Theta(n^2)$, which occurs when a linear chain of nodes results from the repeated TREE-INSERT operations.

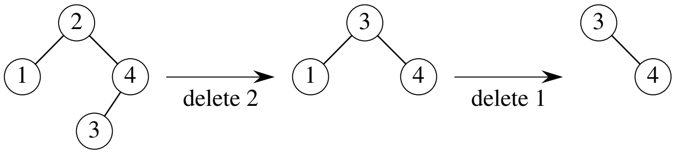
Best case: $\Theta(n \lg n)$, which occurs when a binary tree of height $\Theta(\lg n)$ results from the repeated TREE-INSERT operations.

Solution to Exercise 12.3-5

Deletion is not commutative. In the following binary search tree, delete 1, then delete 2:



Starting from the same binary search tree, delete 2, then delete 1:



The resulting binary search trees differ.

Solution to Exercise 12.3-6

When using the *succ* attribute instead of the p attribute, the pseudocode for TREE-SEARCH, TREE-MINIMUM, and TREE-MAXIMUM does not change, but the pseudocode for all other procedures in the chapter must be updated. Because many of the procedures need to access the root of the binary search tree, we include it as a parameter.

TREE-SUCCESSOR is easy. We add the binary search tree T as a parameter just for consistency with the change we'll need to make to TREE-PREDECESSOR.

```
TREE-SUCCESSOR(T, x)
  return x.succ
```

The procedure TREE-PARENT returns the parent of node x in binary search tree T, replacing the p attribute. TREE-PARENT mimics the search procedure, keeping track of the parent of the current node being examined. Note that this procedure requires all keys to be distinct so that the search for x will always go in the correct direction.

```
TREE-PARENT(T, x)
  y = T.root
  parent = NIL
  while y ≠ x
      parent = y
      if x.key < y.key
          y = y.left
      else y = y.right
  return parent
```

Normally, TREE-PREDECESSOR needs to find the parent of a node. We can't call TREE-PARENT within TREE-PREDECESSOR repeatedly if we want to maintain the $O(h)$ time bound, however, since each call of TREE-PARENT can traverse down the tree. Therefore, we need to implement TREE-PREDECESSOR without calling TREE-PARENT. If x has a nonempty left subtree, just return the rightmost node in that subtree. Otherwise, go to the root and start searching for x, tracking the current node y being checked and y's parent z. Once y and x are the same node, or y becomes NIL, return z. Like TREE-PARENT, this procedure requires all keys to be distinct so that the search for x will always go in the correct direction.

```
TREE-PREDECESSOR$(T, x)$
  if $x.left \neq$ NIL
      return TREE-MAXIMUM$(x.left)$
  else $y = T.root$
      $z =$ NIL
      while $y \neq$ NIL and $y \neq x$
          if $x.key < y.key$
              $y = y.left$
          else $z = y$
              $y = y.right$
      return $z$
```

Now, with TREE-PREDECESSOR, we can write TREE-INSERT.

```
TREE-INSERT$(T, z)$
  $y =$ NIL
  $x = T.root$
  while $x \neq$ NIL
      $y = x$
      if $z.key < x.key$
          $x = x.left$
      else $x = x.right$
  if $y ==$ NIL
      $T.root = z$
  elseif $z.key < y.key$
      $w =$ TREE-PREDECESSOR$(T, y)$
      if $w \neq$ NIL
          $w.succ = y$
      $z.succ = y$
      $y.left = z$
  else $z.succ = y.succ$
      $y.succ = z$
      $y.right = z$
```

Compared with TREE-INSERT in the text, this version omits assigning to $z.p$, but it must maintain the *succ* attributes correctly. The new node z becomes a child of node y. If z becomes y's left child, then y should be z's successor. The code also needs to find y's predecessor w and set w's successor to be z. If z becomes y's right child, things are a little easier. We just need to set z's successor as y's successor and then make y's successor be z.

The TRANSPLANT procedure replaces values of the p attribute by the node returned by calling TREE-PARENT.

```
TRANSPLANT$(T, u, v)$
  $z =$ TREE-PARENT$(T, u)$
  if $z ==$ NIL
      $T.root = v$
  elseif $u == z.left$
      $z.left = v$
  else $z.right = v$
```

Finally, TREE-DELETE omits references to the p attribute and also makes the predecessor of the node z being deleted have its successor become z's successor.

```
TREE-DELETE$(T, z)$
  $x$ = TREE-PREDECESSOR$(T, z)$
  if $x \neq$ NIL
      $x.succ = z.succ$
  if $z.left$ == NIL
      TRANSPLANT$(T, z, z.right)$
  elseif $z.right$ == NIL
      TRANSPLANT$(T, z, z.left)$
  else $y$ = TREE-MINIMUM$(z.right)$
      if $y \neq z.right$
          TRANSPLANT$(T, y, y.right)$
          $y.right = z.right$
      TRANSPLANT$(T, z, y)$
      $y.left = z.left$
```

Because each call of TREE-PREDECESSOR and TREE-PARENT takes $O(h)$ time, both TREE-INSERT and TREE-DELETE take $O(h)$ time.

Solution to Problem 12-2
This solution is also posted publicly

To sort the strings of S, 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 correspond to tan nodes 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 prints the current node and calls itself recursively on the left and right subtrees, so that 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.

Here is pseudocode for the preorder tree walk. It assumes that each node has attributes *left* and *right*, pointing to its children (NIL for children that are not present), and a boolean attribute *string* to indicate whether the node indicates an actual string (i.e., a tan node in Figure 12.5 of the text). The initial call is PREORDER-RADIX-TREE-WALK$(T.root, \varepsilon)$, where ε denotes an empty string. The symbol $\|$ denotes the concatenation of strings.

```
PREORDER-RADIX-TREE-WALK(x, string-so-far)
  if x.string == TRUE
      print string-so-far
  if x.left ≠ NIL
      PREORDER-RADIX-TREE-WALK(x.left, string-so-far ‖ 0)
  if x.right ≠ NIL
      PREORDER-RADIX-TREE-WALK(x.left, string-so-far ‖ 1)
```

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 (7.3) in part (c) of Problem 7-3, we replace $\mathrm{E}[T(\cdot)]$ by $P(\cdot)$ and we replace q by k, we get the same recurrence as in part (d) of Problem 12-3. We can use the same technique as was used in Problem 7-3 to solve it.

We start by solving part (d) of Problem 7-3: 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 7-3(d), we split the summation into two parts:

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k .$$

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,

$$\begin{aligned}
\sum_{k=2}^{n-1} k \lg k &< (\lg n - 1)\sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\le \frac{1}{2}n(n-1)\lg n - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2} \\
&\le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2
\end{aligned}$$

if $n \ge 2$.

Now we show that the recurrence

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n)$$

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

$$\begin{aligned}
P(n) &= \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) \\
&\le \frac{2}{n}\sum_{k=1}^{n-1}(ak \lg k + b) + \Theta(n) \\
&= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n)
\end{aligned}$$

$$
\begin{aligned}
&\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \\
&< an \lg n - \frac{a}{4}n + 2b + \Theta(n) \\
&= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \\
&\le an \lg n + b \;,
\end{aligned}
$$

since we can choose a large enough so that $an/4$ 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 with 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 with 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. 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.

- A single sentinel, $T.nil$, represents all the leaves of red-black tree T.
- $T.nil.color$ is black.
- The root's parent is also $T.nil$.

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 $T.nil$.

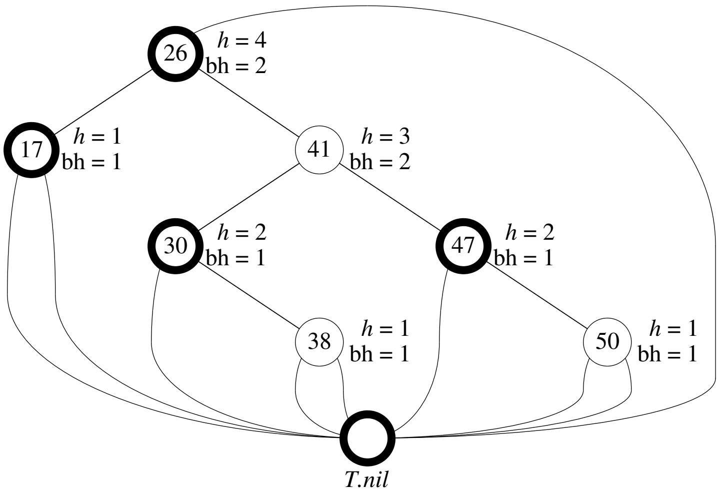
Red-black properties

[Leave these up on the board.]

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($T.nil$) 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 T.nil 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: $\text{bh}(x)$ is the number of black nodes (including $T.nil$) 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 of claim 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 $\geq 2^{\text{bh}(x)} - 1$ internal nodes.

Proof of claim 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?

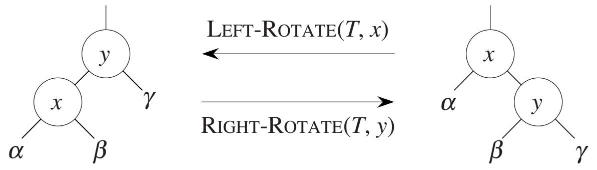
- Red? Might violate property 4.
- Black? Might violate property 5.

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)
  $y = x.right$
  $x.right = y.left$              // turn $y$'s left subtree into $x$'s right subtree
  if $y.left \neq T.nil$          // if $y$'s left subtree is not empty ...
      $y.left.p = x$              // ... then $x$ becomes the parent of the subtree's root
  $y.p = x.p$                     // $x$'s parent becomes $y$'s parent
  if $x.p == T.nil$               // if $x$ was the root ...
      $T.root = y$                // ... then $y$ becomes the root
  elseif $x == x.p.left$          // otherwise, if $x$ was a left child ...
      $x.p.left = y$              // ... then $y$ becomes a left child
  else $x.p.right = y$            // otherwise, $x$ was a right child, and now $y$ is
  $y.left = x$                    // make $x$ become $y$'s left child
  $x.p = y$
```

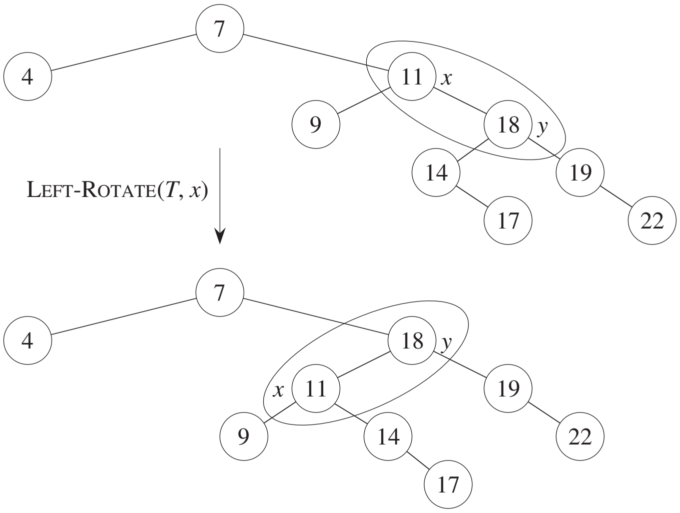
The pseudocode for LEFT-ROTATE assumes that

- $x.right \neq T.nil$, and
- root's parent is $T.nil$.

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.

[Rotation is a basic operation, also used in AVL trees and splay trees. Some books use the terminology of rotating on an edge rather than on a node.]

Insertion

Start by doing regular binary-search-tree insertion:

```
RB-INSERT(T, z)
  x = T.root                  // node being compared with z
  y = T.nil                   // y will be parent of z
  while x ≠ T.nil             // descend until reaching the sentinel
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y                     // found the location—insert z with parent y
  if y == T.nil
      T.root = z              // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil              // both of z's children are the sentinel
  z.right = T.nil
  z.color = RED               // the new node starts out red
  RB-INSERT-FIXUP(T, z)       // correct any violations of red-black properties
```

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because it could have violated a red-black property.

Which property might be violated?

1. OK. (Every node is still either red or black.)
2. If z is the root, then there's a violation. (The root must be black.) Otherwise, OK.
3. OK. (All leaves are still black.)
4. If $z.p$ is red, there's a violation: both z and $z.p$ are red. (Not allowed to have two red nodes in a row.)
5. OK. (Adding a red node doesn't change any black-heights.)

Remove the violation by calling RB-INSERT-FIXUP:

```
RB-INSERT-FIXUP$(T, z)$
  while $z.p.color$ == RED
      if $z.p == z.p.p.left$                // is $z$'s parent a left child?
          $y = z.p.p.right$                 // $y$ is $z$'s uncle
          if $y.color$ == RED               // are $z$'s parent and uncle both red?
              $z.p.color$ = BLACK           ⎫
              $y.color$ = BLACK             ⎬ case 1
              $z.p.p.color$ = RED           ⎪
              $z = z.p.p$                   ⎭
          else
              if $z == z.p.right$
                  $z = z.p$                 ⎫ case 2
                  LEFT-ROTATE$(T, z)$       ⎭
              $z.p.color$ = BLACK           ⎫
              $z.p.p.color$ = RED           ⎬ case 3
              RIGHT-ROTATE$(T, z.p.p)$      ⎭
      else (same as then part, but with "right" and "left" exchanged)
  $T.root.color$ = BLACK
```

[The pseudocode in the book spells out the ***else*** *part line by line. That's a bit much for an in-class situation.]*

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 $z.p$ 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 is guaranteed to terminate. If only case 1 occurs, each iteration moves z toward the root. If case 2 occurs, it falls through into case 3. If case 3 occurs, we'll see that $z.p$ becomes black, so that the loop terminates.

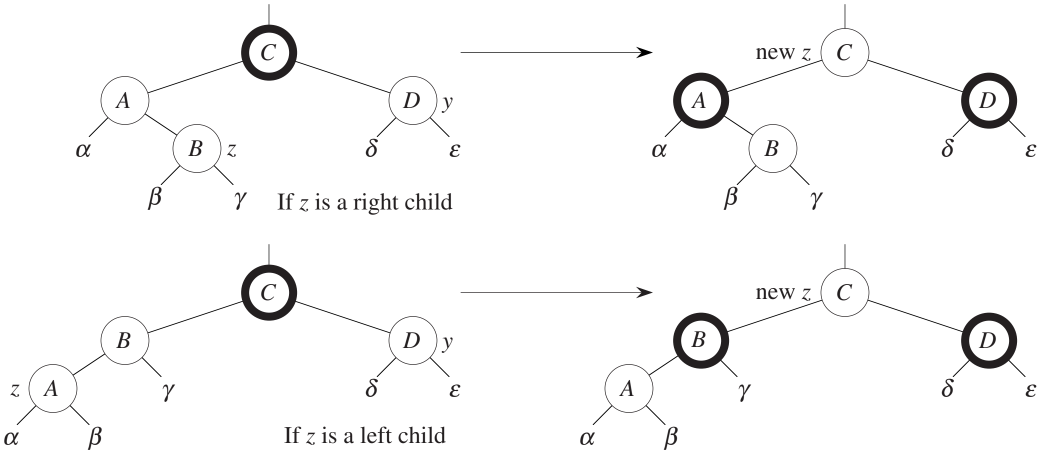
Having established that the loop terminates, it does so because $z.p$ is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.

Maintenance: The loop terminates when z is the root (since then $z.p$ is the sentinel $T.nil$, which is black). Upon starting the first iteration of the loop body, the only possible violation is of property 4.

There are six cases, three of which are symmetric to the other three. The cases are not mutually exclusive. We'll consider cases in which $z.p$ is a left child.

Let y be z's uncle ($z.p$'s sibling).

Case 1: y is red

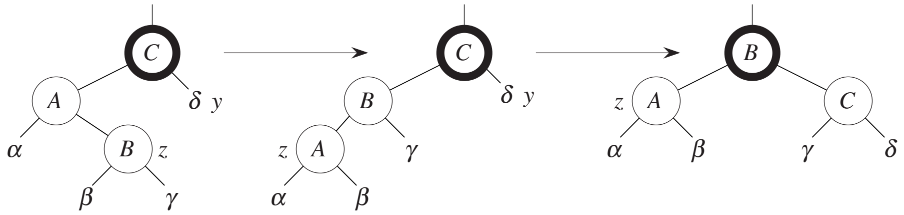


If z is a right child

If z is a left child

- $z.p.p$ (z's grandparent) must be black, since z and $z.p$ are both red and there are no other violations of property 4.
- Make $z.p$ and y black $\Rightarrow$ now z and $z.p$ are not both red. But property 5 might now be violated.
- Make $z.p.p$ red $\Rightarrow$ restores property 5.
- The next iteration has $z.p.p$ 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

- Move z up one level (make $z.p$ the new z), then left rotate around the new z $\Rightarrow$ now z is a left child, and both z and $z.p$ are red.
- Falls through into case 3.

Case 3: y is black, z is a left child

- Make $z.p$ black and $z.p.p$ red.
- Then right rotate on $z.p.p$.
- No longer have 2 reds in a row.
- $z.p$ 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 ⇒ $O(\lg n)$ time.
- Also note that there are at most two rotations overall. *[The constant number of rotations becomes important in Chapter 17 on augmenting data structures.]*

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

Deletion

[Because deletion from a binary search tree changed in the third edition, so did deletion from a red-black tree. As with deletion from a binary search tree, the node z deleted from a red-black tree is always the node z passed to the deletion procedure.]

Based on the TREE-DELETE procedure for binary search trees:

```
RB-DELETE(T, z)
  $y = z$
  $y$-original-color = $y.color$
  if $z.left == T.nil$
      $x = z.right$
      RB-TRANSPLANT$(T, z, z.right)$          // replace $z$ by its right child
  elseif $z.right == T.nil$
      $x = z.left$
      RB-TRANSPLANT$(T, z, z.left)$           // replace $z$ by its left child
  else $y$ = TREE-MINIMUM$(z.right)$          // $y$ is $z$'s successor
      $y$-original-color = $y.color$
      $x = y.right$
      if $y \neq z.right$                       // is $y$ farther down the tree?
          RB-TRANSPLANT$(T, y, y.right)$      // replace $y$ by its right child
          $y.right = z.right$                   // $z$'s right child becomes
          $y.right.p = y$                       //       $y$'s right child
      else $x.p = y$                            // in case $x$ is $T.nil$
      RB-TRANSPLANT$(T, z, y)$                // replace $z$ by its successor $y$
      $y.left = z.left$                         // and give $z$'s left child to $y$,
      $y.left.p = y$                            //       which had no left child
      $y.color = z.color$
  if $y$-original-color == BLACK        // if any red-black violations occurred,
      RB-DELETE-FIXUP$(T, x)$         //       correct them
```

RB-DELETE calls a special version of TRANSPLANT (used in deletion from binary search trees), customized for red-black trees:

```
RB-TRANSPLANT($T, u, v$)
  if $u.p == T.nil$
      $T.root = v$
  elseif $u == u.p.left$
      $u.p.left = v$
  else $u.p.right = v$
  $v.p = u.p$
```

Differences between RB-TRANSPLANT and TRANSPLANT:

- RB-TRANSPLANT references the sentinel $T.nil$ instead of NIL.
- Assignment to $v.p$ occurs even if v points to the sentinel. In fact, we exploit the ability to assign to $v.p$ when v points to the sentinel.

RB-DELETE has almost twice as many lines as TREE-DELETE, but you can find each line of TREE-DELETE within RB-DELETE (with NIL replaced by $T.nil$ and calls to TRANSPLANT replaced by calls to RB-TRANSPLANT).

Differences between RB-DELETE and TREE-DELETE:

- y is either the node z removed from the tree (when z has fewer than 2 children) or z's successor, which is moved within the tree (when z has 2 children).
- Need to save y's original color (in *y-original-color*) to test it at the end, because if it's black, then removing or moving y could cause red-black properties to be violated.
- x is the node that moves into y's original position. It's either y's only child, or $T.nil$ if y has no children.
- Sets $x.p$ to point to the original position of y's parent, even if $x = T.nil$. $x.p$ is set in one of the following ways:
  - If z has only a right child, then x is that right child. The first call of RB-TRANSPLANT sets $x.p$ to z's parent.
  - If z has only a left child, then x is that left child, and the second call of RB-TRANSPLANT sets $x.p$ to z's parent.
  - If z has 2 children and y is z's right child, then y moves up into z's position and x remains y's child. The line "**else** $x.p = y$" seems unnecessary, since x is already y's child. But it is needed in case x is $T.nil$.
  - Finally, if z has 2 children and y is not z's right child, then y will move into z's position in the tree, and x moves into y's position. The third call of RB-TRANSPLANT sets $x.p$ to y's original parent.
- If y's original color was red, no violations of the red-black properties occur:
  - No black-heights change.
  - No red nodes become adjacent, for the following reasons:
    - If z has at most one child, then y and z are the same node, and that node is removed. A child takes its place. Because the node is red, neither its parent nor child can be red, so moving a child to take z's place can't make two red nodes become adjacent.

    - If z has 2 children, then y takes z's place in the tree, and y also takes z's color. So that change can't cause two red nodes in a row. If y is not z's right child, y's original child x takes y's place in the tree. Since y is red, x must be black, and that change can't cause two red nodes in a row, either.
  - The root is black, so y can't be the root. The root remains black.
- If y's original color was black, the changes to the tree structure might cause red-black properties to be violated. The call RB-DELETE-FIXUP at the end resolves the violations.

If y was originally black, what violations of red-black properties could arise?

1. No violation.
2. If y is the root and x is red, then the root has become red.
3. No violation.
4. Violation if $x.p$ and x are both red.
5. Any simple 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 $x.color$ = BLACK) or ***red & black*** (if $x.color$ = RED).
   - The attribute $x.color$ 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 ≠ T.root and x.color == BLACK
      if x == x.p.left                    // is x a left child?
          w = x.p.right                   // w is x's sibling
          if w.color == RED
              w.color = BLACK           }
              x.p.color = RED           }
              LEFT-ROTATE(T, x.p)       } case 1
              w = x.p.right             }
          if w.left.color == BLACK and w.right.color == BLACK
              w.color = RED             }
              x = x.p                   } case 2
          else
              if w.right.color == BLACK
                  w.left.color = BLACK  }
                  w.color = RED         }
                  RIGHT-ROTATE(T, w)    } case 3
                  w = x.p.right         }
              w.color = x.p.color       }
              x.p.color = BLACK         }
              w.right.color = BLACK     } case 4
              LEFT-ROTATE(T, x.p)       }
              x = T.root                }
      else (same as then part, but with "right" and "left" exchanged)
  x.color = BLACK
```

Idea

Move the extra black up the tree until

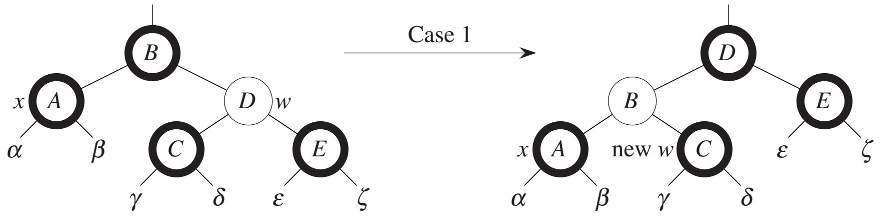
- x points to a red & black node ⇒ turn it into a black node,
- x points to the root ⇒ just remove the extra black, or
- can perform certain rotations and recolorings and then finish.

Within the **while** loop:

- x always points to a nonroot doubly black node.
- w is x's sibling.
- w cannot be $T.nil$, since that would violate property 5 at $x.p$.

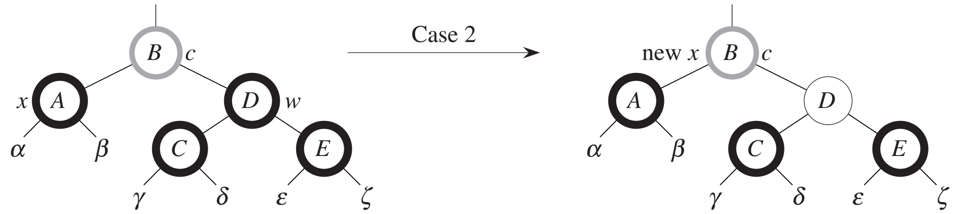
There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which x is a left child.

Case 1: w is red



- w must have black children.
- Make w black and $x.p$ red.
- Then left rotate on $x.p$.
- New sibling of x was a child of w before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

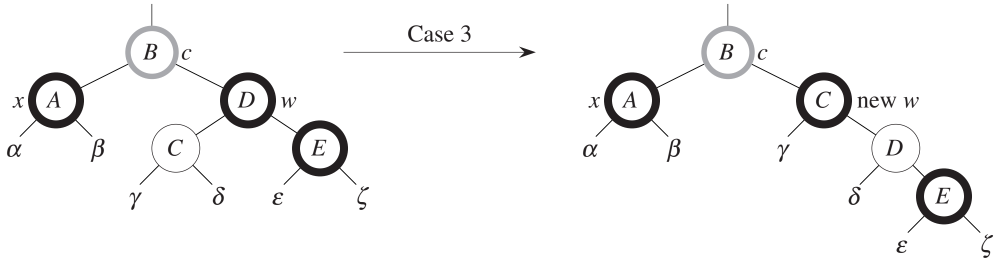
Case 2: w is black and both of w's children are black



[Node with gray outline is of unknown color, denoted by c.]

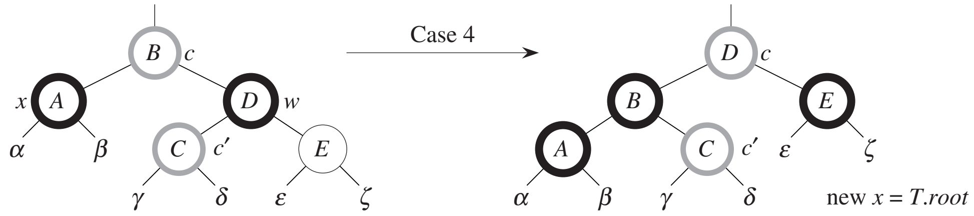
- Take 1 black off x (⇒ singly black) and off w (⇒ red).
- Move that black to $x.p$.
- Do the next iteration with $x.p$ as the new x.
- If entered this case from case 1, then $x.p$ was red ⇒ new x is red & black ⇒ color attribute of new x is RED ⇒ loop terminates. Then new x is made black in the last 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 ⇒ case 4.

Case 4: w 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 $x.p$'s color (c).
- Make $x.p$ black and w's right child black.
- Then left rotate on $x.p$.
- 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 17, 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. The book by Sedgewick and Wayne (citation [402] in the text) mentions "left-leaning red-black binary search trees," which have more concise code than the code given here, but left-leaning red-black binary search trees do not bound the number of rotations per operation to a constant.]

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 13.1-3

Yes, the resulting tree is a red-black tree, and black-heights remain the same.

Solution to Exercise 13.1-4

This solution is also posted publicly

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

This solution is also posted publicly

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 $\text{bh}(x)$ black nodes, even the shortest path from x to a descendant leaf has length at least $\text{bh}(x)$. By definition, the longest path from x to a descendant leaf has length $\text{height}(x)$. Since the longest path has $\text{bh}(x)$ black nodes and at least half the nodes on the longest path are black (by property 4), $\text{bh}(x) \geq \text{height}(x)/2$, so that

$$\text{length of longest path} = \text{height}(x) \leq 2 \cdot \text{bh}(x) \leq \text{ twice length of shortest path .}$$

Solution to Exercise 13.1-6

First, we answer for the smallest number of internal nodes in a red-black tree with black-height k. Such a red-black tree contains only black nodes and k levels of internal nodes. The total number of internal nodes is then $2^{k+1} - 1$.

A red-black tree with black-height k and the largest number of internal nodes would have black nodes at even depths and red nodes at odd depths. With black-height k, there are k depths with black nodes and also k depths with red nodes, making the total number of internal nodes $2^{2k+1} - 1$.

Solution to Exercise 13.1-7

Since a red-black tree can have no red nodes, the smallest possible ratio of red internal nodes to black internal nodes is 0.

A red-black tree with the largest possible ratio of red internal nodes to black internal nodes would have black nodes at even depths and red nodes at odd depths. For every black node, there are two red children, so that the ratio would be 2:1.

Solution to Exercise 13.1-8

Let x be a red node and y be its non-NIL child. By property 4, y must be black. Without loss of generality, let y be x's right child. Then, because x's left child is NIL, x has one black node—the NIL—on its left path down to a leaf. Because y is black and has at least one NIL below it, x has at least two black nodes on any right path down to a leaf. The differing numbers of black nodes on the left and right paths down to leaves violate property 5.

Solution to Exercise 13.2-1

```
RIGHT-ROTATE(T, x)
 y = x.left
 x.left = y.right          // turn y's right subtree into x's left subtree
 if y.right ≠ T.nil        // if y's right subtree is not empty ...
     y.right.p = x         // ... then x becomes the parent of the subtree's root
 y.p = x.p                 // x's parent becomes y's parent
 if x.p == T.nil           // if x was the root ...
     T.root = y            // ... then y becomes the root
 elseif x == x.p.right     // otherwise, if x was a right child ...
     x.p.right = y         // ... then y becomes a right child
 else x.p.left = y         // otherwise, x was a left child, and now y is
 y.right = x               // make x become y's right child
 x.p = y
```

Solution to Exercise 13.2-2

Start by noting that if a node has no children, it cannot be the argument to either LEFT-ROTATE or RIGHT-ROTATE. If a node has two children, it can be the argument to either rotation procedure. And if a node has one child, then if that child is a left child, the node can be the argument to only RIGHT-ROTATE, and if that child is a right child, the node can be the argument to only LEFT-ROTATE. To put it simply, the number of rotations equals the sum of the degrees of the nodes.

This sum equals the total number of edges in the tree. By Theorem B.2, a free tree with n vertices has $n - 1$ edges. The same property holds for rooted trees, since every vertex except for the root has exactly one edge from its parent. Therefore, any n-node binary search tree has exactly $n - 1$ possible rotations.

Solution to Exercise 13.2-3

After performing a left rotation, the depth of a increases by 1, the depth of b is unchanged, and the depth of c decreases by 1.

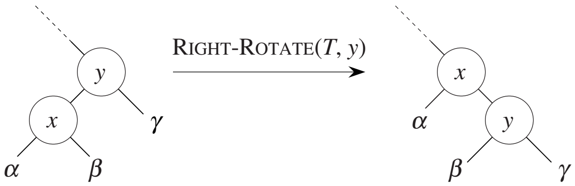
Solution to Exercise 13.2-4

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, any binary search tree can be converted 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 by performing this sequence in reverse —turning each right rotation into its inverse left rotation—T' would transform back into T.

Therefore, here is how to 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 $k, k' \le n-1$. 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 13.2-5

Let T_1 and T_2 each contain nodes x and y. In T_1, x is the root and y is its right child. In T_2, y is the root and x is its left child. Although T_1 can be converted into T_2 by a left rotation on x, no right rotation applies to T_1.

Here is how to see that $O(n^2)$ calls of RIGHT-ROTATE suffice to right-convert T_1 into T_2 if it is at all possible to do so. Let r be the root of T_2. With at most $n-1$ calls of RIGHT-ROTATE, transform T_1 to have r as the root. Root r now has two subtrees. Recursively call RIGHT-ROTATE to transform each of these subtrees into the subtrees of T_2's root.

To see that the total number of calls of RIGHT-ROTATE is $O(n^2)$, let $R(n)$ be the number of calls of RIGHT-ROTATE. After at most $n-1$ calls to get r into the root position, let the left subtree of r contain k nodes and the right subtree of r contain $n-k-1$ nodes, where $k < n$. We get the recurrence

$$R(n) \le R(k) + R(n-k-1) + (n-1) .$$

To show that $R(n) = O(n^2)$, we show by substitution that $R(n) \le n^2$. We have

$$\begin{aligned} R(n) &\le R(k) + R(n-k-1) + (n-1) \\ &\le k^2 + (n-k-1)^2 + (n-1) \\ &= k^2 + (n^2 - 2kn - 2n + k^2 + k + 1) + (n-1) \\ &= n^2 - 2kn - 2n + 2k^2 + k + n - 1 , \end{aligned}$$

which is less than or equal to n^2 if $-2kn - 2n + 2k^2 + k + n - 1 \le 0$, or equivalently, if $2k^2 + k \le 2kn + n + 1$. Since $k < n$, we have that $2k^2 < 2kn$, and so the required condition holds.

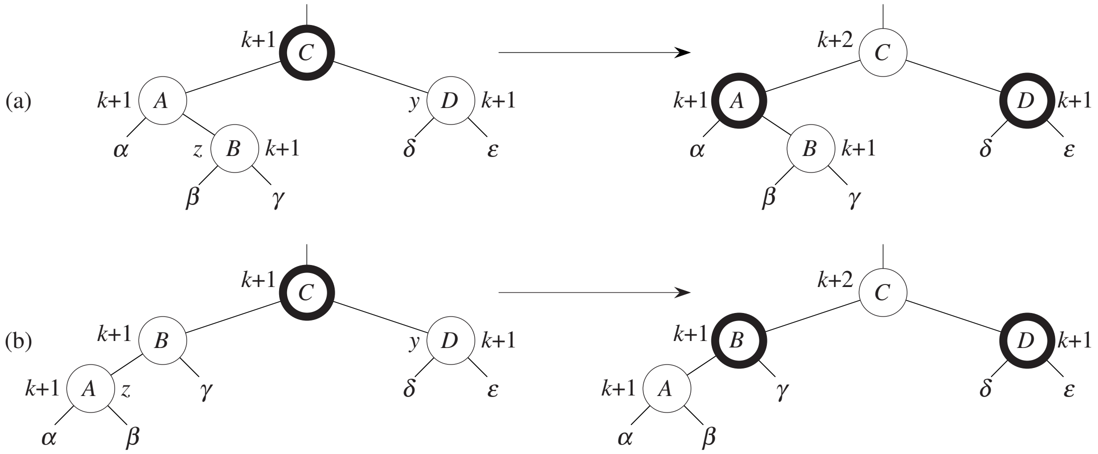
Solution to Exercise 13.3-1

If the inserted node z was set to black, then black-heights would be inconsistent, violating property 5. Correcting this problem would be harder than fixing two consecutive red nodes.

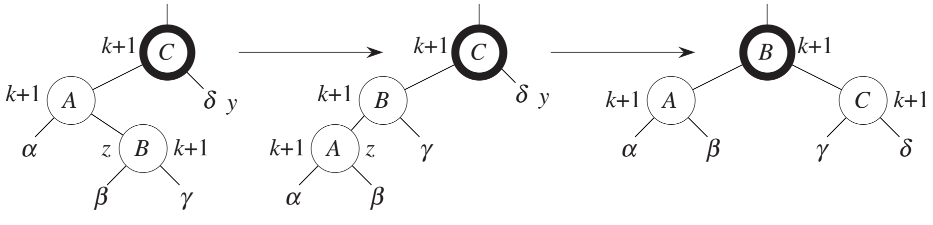
Solution to Exercise 13.3-3
This solution is also posted publicly

Note: In the figures below, nodes with a heavy outline are black, and nodes with a regular outline are red.

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 $k + 1$.



Case 2 Case 3

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 $z.p.p$ that is reddened. If $z.p.p$ is the sentinel, then $z.p$ is the root. By part (b) of the loop invariant and line 1 of RB-INSERT-FIXUP, if $z.p$ is the root, then the loop terminates. The only subtlety is in case 2, which sets $z = z.p$ before case 3 recolors $z.p.p$ red. Because case 2 rotates before case 3 recolors, the identity of $z.p.p$ is the same before and after case 2, so there's no problem.

Solution to Exercise 13.3-5

If $n > 1$, then the inserted node is not the root. In case 1, the node z that is inserted remains red. If case 3 occurs without coming from case 2, then the inserted node z remains red. If case 2 occurs, both the inserted node z and its parent are red. The original parent of z becomes the new z, and this node remains red after case 3.

Solution to Exercise 13.4-1

If $y = z$, then z has either zero or one non-NIL child. Because y is red, then by Exercise 13.1-8, if $y = z$, then z has no non-NIL children. In this case, z is replaced by $T.nil$ and the black-height of $z.p$ does not change, and therefore no other black-heights change.

If $y \neq z$, then z has two children, and y, having the minimum key in z's right subtree, has NIL for its left child. Again, by Exercise 13.1-8, because y is red, its right child x must also be NIL. When y moves into z's position, it takes on z's color, which does not cause any black-heights to change. When x moves into y's position, then as in the case when $y = z$, the black-height of $y.p$ does not change, and thus no other black-heights change.

Solution to Exercise 13.4-2

The **while** loop of RB-DELETE-FIXUP terminates in one of two cases. Either $x = T.root$, in which case the root's color is set to black on the last line of the procedure, or $x.color = \text{RED}$, in which case the procedure did not climb all the way to the root, so that the root's color is still black.

Solution to Exercise 13.4-3

If x is red, then the test in line 1 of RB-DELETE-FIXUP fails the first time, and x is colored black in line 44.

Solution to Exercise 13.4-5

As pointed out in the text, the node x passed to RB-DELETE-FIXUP could be the sentinel $T.nil$, which is why line 16 of RB-DELETE sets $x.p = y$ even though x is y's right child. Therefore, every line of RB-DELETE-FIXUP that accesses x or $x.p$ could access $T.nil$. No lines in RB-DELETE-FIXUP modify the attributes of x, and so no lines in RB-DELETE-FIXUP modify $T.nil$.

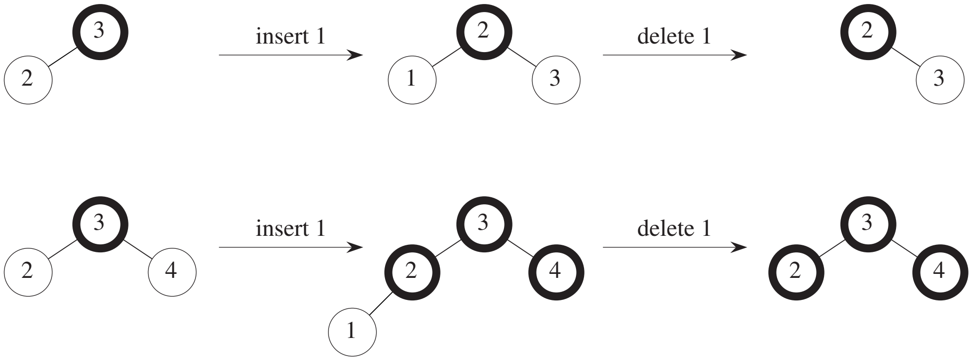
Solution to Exercise 13.4-7

Case 1 occurs only if x's sibling w is red. If $x.p$ were red, then there would be two reds in a row, namely $x.p$ (which is also $w.p$) and w, and these two red nodes would have been consecutive even before calling RB-DELETE.

Solution to Exercise 13.4-8

Note: In the figures below, nodes with a heavy outline are black, and nodes with a regular outline are red.

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 Exercise 13.4-9

Assume that the procedures TREE-MINIMUM and TREE-SEARCH have been modified to use *T.nil* in place of NIL.

We start by modifying the TREE-SUCCESSOR procedure from Section 12.2 to find the successor of node x within red-black tree T, but confined to the subtree rooted at node r:

```
SUBTREE-SUCCESSOR(T, r, x)
  if $x.right \neq T.nil$
      return TREE-MINIMUM($x.right$)
  $y = x.p$
  while $y \neq r.p$ and $x == y.right$
      $x = y$
      $y = y.p$
  if $y == r.p$
      return $T.nil$
  else return $y$
```

Assuming that the subtree rooted at node r contains nodes with keys a and b, then a search within the subtree finds the node with key a, and then repeated calls to SUBTREE-SUCCESSOR find nodes in the range $[a, b]$ until either running out of nodes in the subtree or finding a node whose key equals b. Alternatively, the search can be for b, followed by calls to SUBTREE-PREDECESSOR, which is analogous to SUBTREE-SUCCESSOR. Here is RB-ENUMERATE, assuming that the keys a and b are present in the subtree rooted at node r:

```
RB-ENUMERATE(T, r, a, b)
  $x$ = TREE-SEARCH($r, a$)
  while $x \neq T.nil$ and $x.key \leq b$
      print $x.key$
      $x$ = SUBTREE-SUCCESSOR($T, r, x$)
```

The test for $x \neq T.nil$ is necessary in case b is the greatest key in the subtree rooted at r.

Without the assumption that the values a and b appear as keys in the subtree rooted at node r, RB-ENUMERATE must handle both the case in which a appears in the subtree and in which a does not. If no node has key a, then insert a temporary node z with key a in the subtree rooted at r, repeatedly call SUBTREE-SUCCESSOR starting at z's successor, and then delete node z. The insertion procedure RB-SUBTREE-INSERT is like RB-INSERT except that it starts at node r and does not call RB-INSERT-FIXUP, since the temporary node z is going to be deleted anyway.

```
RB-SUBTREE-INSERT(T, r, z)
  y = T.nil
  x = r
  while x ≠ T.nil
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == T.nil
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
```

With these ideas in mind, here is the more robust version of RB-ENUMERATE:

```
RB-ENUMERATE(T, r, a, b)
  y = TREE-SEARCH(r, a)
  if y == T.nil
      allocate a new node z
      z.key = a
      RB-SUBTREE-INSERT(T, r, z)
      x = SUBTREE-SUCCESSOR(T, r, z)
  else x = y
  while x ≠ T.nil and x.key ≤ b
      print x.key
      x = SUBTREE-SUCCESSOR(T, r, x)
  if y == T.nil
      RB-DELETE(T, z)
```

To see why RB-ENUMERATE runs in $O(m + \lg n)$ time, searching for a node with key a takes $O(\lg n)$ time in a red-black tree, as do inserting and deleting the temporary node z. The **while** loop in the second version of RB-ENUMERATE calls SUBTREE-SUCCESSOR at most $m + 1$ times, since it stops upon finding the first node whose key is greater than b, or finding $T.nil$ if b is the greatest key in the subtree rooted at r. By Exercise 12.2-8, the $m + 1$ consecutive calls to SUBTREE-SUCCESSOR take $O(m + \lg n)$ time in a red-black tree.

(The exercise asks about TREE-SUCCESSOR, but the upper bound is the same for SUBTREE-SUCCESSOR.) The first version of RB-ENUMERATE calls SUBTREE-SUCCESSOR m times, since the first key printed is found by TREE-SEARCH rather than by SUBTREE-SUCCESSOR. Thus, the running time of either version of RB-ENUMERATE is $O(m + \lg n)$.

Solution to Problem 13-1

This solution is also posted publicly

a. When inserting a node, 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 node z, three possibilities may occur:

- If z has at most one child, then z will be spliced out, so that all ancestors of z must be changed. (As with insertion, the need for a new child pointer propagates up from the removed node.)
- If z has two children and its successor y is z's right child, then replace z by y, so that all ancestors of z must be changed (i.e., the same as if z has at most one child).
- If z has two children and its successor y is not z's right child, then replace z by y and replace y by y's right child x. Since y and z are ancestors of x, all ancestors of y must be changed.

Since there is no parent attribute, no other nodes need to be changed.

b. 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 without using parent attributes.

```
PERSISTENT-TREE-INSERT(T, z)
  create a new persistent binary search tree $T'$
  $T'.root$ = COPY-NODE($T.root$)
  $y$ = NIL
  $x = T'.root$
  while $x \neq$ NIL
      $y = x$
      if $z.key < x.key$
          $x$ = COPY-NODE($x.left$)
          $y.left = x$
      else $x$ = COPY-NODE($x.right$)
          $y.right = x$
  if $y$ == NIL
      $new\text{-}root = z$
  elseif $z.key < y.key$
      $y.left = z$
  else $y.right = z$
  return $T'$
```

The second uses a recursive subroutine, PERSISTENT-SUBTREE-INSERT(r, z) that inserts node z into the subtree rooted at node r in T, copying nodes as needed, and returning either node z or the copy in T' of node r.

```
PERSISTENT-TREE-INSERT(T, z)
  create a new persistent binary search tree T'
  T'.root = PERSISTENT-SUBTREE-INSERT(T.root, z)
  return T'
```

```
PERSISTENT-SUBTREE-INSERT(r, z)
  if r == NIL
      x = z
  else x = COPY-NODE(r)
      if z.key < r.key
          x.left = PERSISTENT-SUBTREE-INSERT(r.left, z)
      else x.right = PERSISTENT-SUBTREE-INSERT(r.right, z)
  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 attributes, 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 that 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. (We'll look at deletion a little later.) To do so, we will need to show two things:

- How to still find the parent pointers that are needed in $O(1)$ time without using a parent attribute. We cannot use a parent attribute because a persistent tree with parent attributes requires $\Omega(n)$ 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.

Here is how to find each parent pointer needed during insertion in $O(1)$ time without having a parent attribute. 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 attribute.

Rotation and recoloring change nodes as follows:

- RB-INSERT-FIXUP performs at most two rotations, and each rotation updates the child pointers in three nodes (the node being rotated around, that node's parent, and one of the children of the node being rotated around). Thus, at most six nodes are directly modified by rotation during RB-INSERT-FIXUP. In a persistent tree, all ancestors of a changed node are copied, so that 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 $O(\lg n)$ ancestors, hence $O(\lg n)$ 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, the procedure needs to walk down the tree to the deepest node being changed, to build up a stack of parents as discussed above for insertion. This walk relies on keys being distinct.

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 three rotations, which as discussed above for insertion requires $O(\lg n)$ time to change nodes due to persistence. It also makes $O(\lg n)$ 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 $O(\lg n)$.

Solution to Problem 13-2

a. An empty red-black tree has $T.bh = 0$. A red-black tree's black-height increases only upon an RB-INSERT operation that has to recolor a red root to black, in which case $T.bh$ increases by 1. A red-black tree's black-height decreases only upon an RB-DELETE operation in which the node x with the extra black goes all the way up to the root, when $T.bh$ decreases by 1. In these cases, the additional running time is just a constant.

In order to determine the black-height of each node visited while descending through a red-black tree, maintain a running black-height, say b, initialized to $T.bh + 1$ before visiting the root. Upon visiting a black node, immediately decrement b; the black-height of the black node is then b. Upon visiting a red node, leave b alone, and the black-height of the red node is b. We have to initialize b to $T.bh + 1$ instead of $T.bh$ so that the root has the correct black-height. The additional time required is just a constant per node.

b. Starting at $T_1.root$, descend through T_1, going right at each node that has a right child, and otherwise going left. Use the method from the solution to part (a) to determine the black-height of each node. Upon encountering a black node in T_1 whose black-height is $T_2.bh$, stop: that is the node y being searched for. Because the search starts at $T_1.root$ and descends down, it takes $O(\lg n)$ time.

c. Start by creating a new tree T' with root x, $x.left = y$, and $x.right = T_2.root$. Since $x_1.key \le x.key \le x_2.key$ for all nodes x_1 in T_1 and x_2 in T_2, tree T' is a binary search tree. Replace node y within T_1 by node x. This operation just changes pointers in x, y, $y.left$, $y.right$, and $T_2.root$, taking $O(1)$ time.

d. To ensure that properties 1, 3, and 5 hold, color x red. Since $T_2.root$ has the same black-height as y, and they are both children of x, coloring x red keeps every node as either red or black (property 1) and it keeps the black-heights consistent throughout the tree (property 5). No leaves change, so that property 3 is maintained as well.

If x happens to be the root of the new tree, recoloring it as black restores property 2. If $x.p$ happens to be red, then calling RB-INSERT-FIXUP(T_1, x) restores all the red-black properties in $O(\lg n)$ time.

e. If $T_1.bh \le T_2.bh$, then find a black node in T_2 with the largest key from among those nodes in T_2 whose black-height is $T_1.bh$. Do so by descending from $T_2.root$, going left at each node that has a left child, and otherwise going right, until finding a black node whose black-height is $T_1.bh$. Call that node y. Then create a new tree T' with root x, $x.left = T_1.root$, and $x.right = y$. Replace node y within T_2 by node x.

f. There is no additional asymptotic cost to maintaining the $bh.a$ttribute for a red-black tree. Finding the node y takes $O(\lg n)$ time, replacing node y takes $O(1)$ time, and restoring the red-black properties takes $O(\lg n)$ time. Thus, RB-JOIN takes $O(\lg n)$ time.

Solution to Problem 13-3

a. Let $N(h)$ be the minimum number of nodes in an AVL tree of height h. We will show by induction on h that $N(h) \ge F_h$.

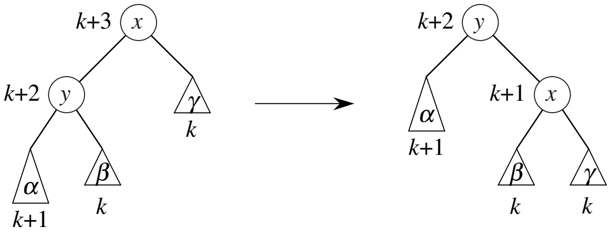
The bases cases are for $h = 0$ and $h = 1$. A tree with height 0 has only a root, so that $N(0) = 1 > 0 = F_0$. A tree with height 1 has at least two nodes, so that $N(1) = 2 > 1 = F_1$.

For the inductive step, the root of an AVL tree of height h has two children. One has height $h - 1$ and the other has height at least $h - 2$. By the inductive hypothesis, $N(h-1) \geq F_{h-1}$ and $N(h-2) \geq F_{h-2}$. Therefore, we have that $N(h) \geq N(h-1) + N(h-2) \geq F_{h-1} + F_{h-2} = F_h$. By how we defined $N(h)$, we have $N(h) \leq n$ for any n-node AVL tree of height h, so that $n \geq F_h$.

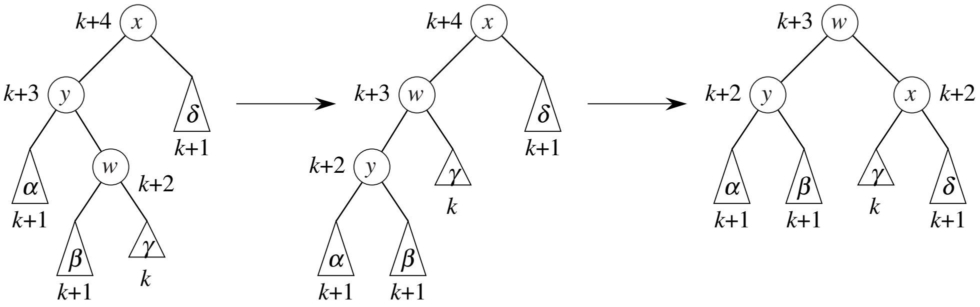
Since $F_h \geq \phi^h/\sqrt{5}$, we have $n \geq \phi^h/\sqrt{5}$ or, equivalently, $h \leq \log_\phi \sqrt{5}n$. By equation (3.19), $h = O(\lg n)$.

b. Following the hint, we'll use rotations to balance out subtree heights. The h attribute should be such that $x.h = 1 + \max\{x.\mathit{left}.h, x.\mathit{right}.h\}$. Under this rule, an empty subtree has an h attribute of -1 so that a leaf has $h = 0$.

To illustrate the possibilities, we'll look only at the situation where $x.\mathit{left}.h = x.\mathit{right}.h + 2$. The situations in which $x.\mathit{right}.h = x.\mathit{left}.h + 2$ are symmetric. In the illustrations, the value k denotes the heights of some subtrees, and other heights are in terms of k. There are two possibilities. Let $y = x.\mathit{left}$. In the first possibility, $y.\mathit{left}.h = y.\mathit{right}.h + 1$. A right rotation on x restores the AVL property.



In the other possibility, $y.\mathit{right}.h = y.\mathit{left}.h + 1$. A left rotation on y followed by a right rotation on x restores the AVL property.



Here is the BALANCE procedure. It assumes that if $x.\mathit{left}.h$ and $x.\mathit{right}.h$ differ by more than 1, then they differ by 2. It also assumes that the parent of the root is NIL. It returns a pointer to the root of the subtree that originally had root x.

```
BALANCE(x)
  if |x.left.h − x.right.h| > 1
      if x.left.h > x.right.h
          y = x.left
          if y.right.h > y.left.h
              RIGHT-ROTATE(x)
              x.h = x.h − 1
              y.h = y.h + 1
              return y
          else w = y.right
              LEFT-ROTATE(y)
              RIGHT-ROTATE(x)
              x.h = x.h − 2
              y.h = y.h − 1
              w.h = w.h + 1
              return w
      else y = x.right
          if y.left.h > y.right.h
              LEFT-ROTATE(x)
              x.h = x.h − 1
              y.h = y.h + 1
              return y
          else w = y.left
              RIGHT-ROTATE(y)
              LEFT-ROTATE(x)
              x.h = x.h − 2
              y.h = y.h − 1
              w.h = w.h + 1
              return w
  else return x        // the subtree rooted at x did not change
```

A call of BALANCE takes $O(1)$ time and performs $O(1)$ rotations.

c. Here is pseudocode for AVL-INSERT:

```
AVL-INSERT(T, z)
  TREE-INSERT(T, z)
  x = z.p
  if x.h == 0
      x.h = 1
      repeat
          x = x.p
          subtree-height = x.h
          x = BALANCE(x)
      until x == NIL or subtree-height == x.h
```

The procedure sets x to be z's parent after inserting z via TREE-INSERT. Before calling TREE-INSERT, x was either a leaf or had one child. If x had a child, then its height had to have been 1 beforehand, and adding z as x's other child does not change x's height. Since x's height doesn't change, the tree remains an AVL tree.

If x was a leaf, however, then its height goes from 0 to 1. If x does not have a sibling, then that missing sibling's height is -1, and now x's parent has children whose heights differ by 2. We need to run BALANCE on x's parent. If that execution of BALANCE doesn't change x's parent's height, we're done. Otherwise, we have to go up one level in the tree. We keep going up until either a node's height does not change or we have run BALANCE on the root.

d. Since the height of the tree is $O(\lg n)$ and each call of BALANCE takes $O(1)$ time and makes $O(1)$ rotations, a call of AVL-INSERT takes $O(\lg n)$ and performs $O(\lg n)$ rotations.

Lecture Notes for Chapter 14: 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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod cutting

How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integer number of inches.

Input: A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.

Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

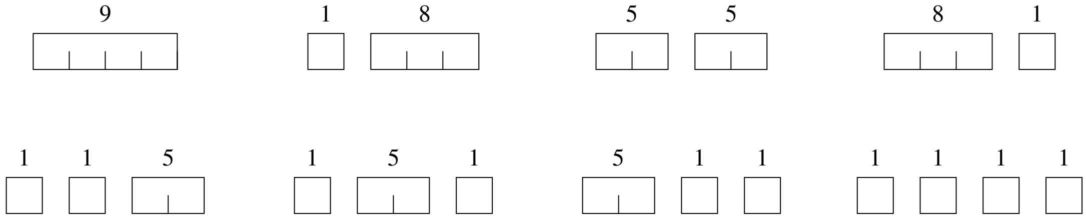
If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

Example: *[Using the first* 8 *values from the example in the textbook.]*

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.

Here are all 8 ways to cut a rod of length 4, with the costs from the example:



The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$.

Let r_i be the maximum revenue for a rod of length i. Can express a solution as a sum of individual rod lengths.

Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2+2$
5	13	$2+3$
6	17	6 (no cuts)
7	18	$1+6$ or $2+2+3$
8	22	$2+6$

Can determine optimal revenue r_n by taking the maximum of

- p_n: the revenue from not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
- $r_{n-1} + r_1$.

That is,

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} .$$

Optimal substructure: To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, two subproblems remain. The optimal solution to the original problem incorporates optimal solutions to the subproblems. May solve the subproblems independently.

Example: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. Need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

A simpler way to decompose the problem: Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.

- Need to divide only the remainder, not the first piece.
- Leaves only one subproblem to solve, rather than two subproblems.
- Say that the solution with no cuts has first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.
- Gives a simpler version of the equation for r_n:

  $$r_n = \max\{p_i + r_{n-i} : 1 \le i \le n\} \ .$$

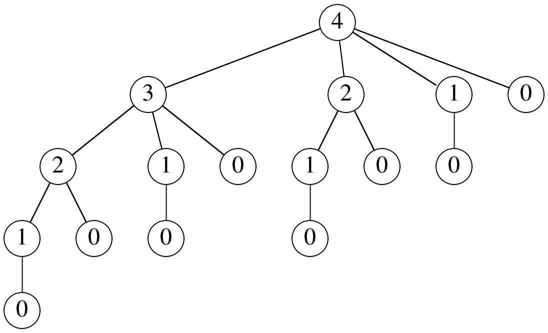
Recursive top-down solution

Direct implementation of the simpler equation for r_n.
The call CUT-ROD(p, n) returns the optimal revenue r_n:

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = -∞
  for i = 1 to n
      q = max {q, p[i] + CUT-ROD(p, n - i)}
  return q
```

This procedure works, but it is terribly *inefficient*. If you code it up and run it, it could take more than an hour for $n = 40$. Running time approximately doubles each time n increases by 1.

Why so inefficient?: CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



Lots of repeated subproblems. Solves the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

Exponential growth: Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0 , \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 . \end{cases}$$

Summation counts calls where second parameter is $j = n - i$.
Solution to recurrence is $T(n) = 2^n$.

Dynamic-programming solution

Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
"Store, don't recompute" ⇒ time-memory trade-off.
Can turn an exponential-time solution into a polynomial-time solution.

Two basic approaches: top-down with memoization, and bottom-up.

Top-down with memoization

Solve recursively, but store each result in a table.
To find the solution to a subproblem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.

Memoizing is remembering what has been computed previously. *["Memoizing," not "memo**r**izing."]*

Memoized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:

```
MEMOIZED-CUT-ROD(p, n)
  let r[0 : n] be a new array        // will remember solution values in r
  for i = 0 to n
      r[i] = −∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0                  // already have a solution for length n?
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n           // i is the position of the first cut
          q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
  r[n] = q                     // remember the solution value for length n
  return q
```

Bottom-up

Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems needed.

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0 : n] be a new array     // will remember solution values in r
  r[0] = 0
  for j = 1 to n                  // for increasing rod length j
      q = −∞
      for i = 1 to j              // i is the position of the first cut
          q = max {q, p[i] + r[j − i]}
      r[j] = q                    // remember the solution value for length j
  return r[n]
```

Running time

Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.

- Bottom-up: Doubly nested loops. Number of iterations of inner **for** loop forms an arithmetic series.
- Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop iterates n times ⇒ over all recursive calls, total number of iterations forms an arithmetic series. *[Actually using aggregate analysis, which Chapter 16 covers.]*

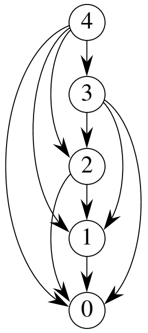
Subproblem graphs

How to understand the subproblems involved and how they depend on each other.

Directed graph:

- One vertex for each distinct subproblem.
- Has a directed edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y.

Example: For rod-cutting problem with $n = 4$:



Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.

Subproblem graph can help determine running time. Because each subproblem is solved just once, running time is sum of times needed to solve each subproblem.

- Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

When these conditions hold, running time is linear in number of vertices and edges.

Reconstructing a solution

So far, have focused on computing the *value* of an optimal solution, rather than the *choices* that produced an optimal solution.

Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
  let $r[0:n]$ and $s[1:n]$ be new arrays
  $r[0] = 0$
  for $j = 1$ to $n$                  // for increasing rod length $j$
      $q = -\infty$
      for $i = 1$ to $j$              // $i$ is the position of the first cut
          if $q < p[i] + r[j-i]$
              $q = p[i] + r[j-i]$
              $s[j] = i$              // best cut location so far for length $j$
      $r[j] = q$                      // remember the solution value for length $j$
  return $r$ and $s$
```

Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.

To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION$(p, n)$
  $(r, s)$ = EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
  while $n > 0$
      print $s[n]$          // cut location for length $n$
      $n = n - s[n]$        // length of the remainder of the rod
```

Example: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$		1	2	3	2	2	6	1	2

A call to PRINT-CUT-ROD-SOLUTION$(p, 8)$ calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

Matrix-chain multiplication

Problem: Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$ using standard matrix multiplication (not Strassen's method) while minimizing the number of scalar multiplications.

How to parenthesize the product to minimize the number of scalar multiplications?

Suppose multiplying matrices A and B: $C = A \cdot B$. *[The textbook has a procedure to compute $C = C + A \cdot B$, but it's easier in a lecture situation to just use $C = A \cdot B$.]* The matrices must be compatible: number of columns of A equals number of rows of B. If A is $p \times q$ and B is $q \times r$, then C is $p \times r$ and takes pqr scalar multiplications.

Example: $A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$. Compute $A_1 A_2 A_3$, which is 10×50.

- Try parenthesizing by $((A_1 A_2) A_3)$. First perform $10 \cdot 100 \cdot 5 = 5000$ multiplications, then perform $10 \cdot 5 \cdot 50 = 2500$, for a total of 7500.
- Try parenthesizing by $(A_1 (A_2 A_3))$. First perform $100 \cdot 5 \cdot 50 = 25{,}000$ multiplications, then perform $10 \cdot 100 \cdot 50 = 50{,}000$, for a total of 75,000.
- The first way is 10 times faster.

Input to the problem: Let A_i be $p_{i-1} \times p_i$. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

Note: Not actually multiplying matrices. Just deciding an order with the lowest cost.

Counting the number of parenthesizations

Let $P(n)$ denote the number of ways to parenthesize a product of n matrices. $P(1) = 1$.

When $n \geq 2$, can split anywhere between A_k and A_{k+1} for $k = 1, 2, \ldots, n-1$. Then have to split the subproducts. Get

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \displaystyle\sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 . \end{cases}$$

The solution is $P(n) = \Omega(4^n / n^{3/2})$. *[The textbook does not prove the solution to this recurrence.]* So brute force is a bad strategy.

Step 1: Structure of an optimal solution

Let $A_{i:j}$ be the matrix product $A_i A_{i+1} \cdots A_j$.

If $i < j$, then must split between A_k and A_{k+1} for some $i \leq k < j \Rightarrow$ compute $A_{i:k}$ and $A_{k+1:j}$ and then multiply them together. Cost is

$$\begin{aligned} & \text{cost of computing } A_{i:k} \\ + \; & \text{cost of computing } A_{k+1:j} \\ + \; & \text{cost of multiplying them together} . \end{aligned}$$

Optimal substructure: Suppose that optimal parenthesization of $A_{i:j}$ splits between A_k and A_{k+1}. Then the parenthesization of $A_{i:k}$ must be optimal. Otherwise, if there's a less costly way to parenthesize it, you'd use it and get a parenthesization of $A_{i:j}$ with a lower cost. Same for $A_{k+1:j}$.

Therefore, to build an optimal solution to $A_{i:j}$, split it into how to optimally parenthesize $A_{i:k}$ and $A_{k+1:j}$, find optimal solutions to these subproblems, and then combine the optimal solutions. Need to consider all possible splits.

Step 2: A recursive solution

Define the cost of an optimal solution recursively in terms of optimal subproblem solutions.

Let $m[i,j]$ be the minimum number of scalar multiplications to compute $A_{i:j}$. For the full problem, want $m[1,n]$.

If $i = j$, then just one matrix $\Rightarrow m[i,i] = 0$ for $i = 1, 2, \ldots, n$.

If $i < j$, then suppose the optimal split is between A_k and A_{k+1}, where $i \le k < j$. Then $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_i p_j$.

But that's assuming you know the value of k. Have to try all possible values and pick the best, so that

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j : i \le k < j\} & \text{if } i < j \ . \end{cases}$$

That formula gives the cost of an optimal solution, but not how to construct it. Define $s[i,j]$ to be a value of k to split $A_{i:j}$ in an optimal parenthesization. Then $s[i,j] = k$ such that $m[i,j] = m[i,k] + m[k_1,j] + p_{i-1}p_k p_j$.

Step 3: Compute the optimal costs

Could implement a recursive algorithm based on the above equation for $m[i,j]$. *Problem:* It would take exponential time.

There are not all that many subproblems: just one for each i, j such that $1 \le i \le j \le n$. There are $\binom{n}{2} + n = \Theta(n^2)$ of them. Thus, a recursive algorithm would solve the same subproblems over and over.

In other words, this problem has overlapping subproblems.

Here is a tabular, bottom-up method to solve the problem. It solves subproblems in order of increasing chain length. The variable $l = j - i + 1$ indicates the chain length.

```
MATRIX-CHAIN-ORDER$(p, n)$
  let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
  for $i = 1$ to $n$                        // chain length 1
      $m[i,i] = 0$
  for $l = 2$ to $n$                        // $l$ is the chain length
      for $i = 1$ to $n - l + 1$            // chain begins at $A_i$
          $j = i + l - 1$                   // chain ends at $A_j$
          $m[i,j] = \infty$
          $m[i,j] = \infty$
          for $k = i$ to $j - 1$
              $q = m[i,k] + m[k+1, j] + p_{i-1} p_k p_j$
              if $q < m[i,j]$
                  $m[i,j] = q$              // remember this cost
                  $s[i,j] = k$              // remember this index
  return $m$ and $s$
```

All n chains of length 1 are initialized so that $m[i,i] = 0$ for $i = 1, 2, \ldots, n$. Then $n - 1$ chains of length 2 are computed, then $n - 2$ chains of length 3, and so on, up to 1 chain of length n.

[We don't include an example here because the arithmetic is hard for students to process in real time.]

Time: $O(n^3)$, from triply nested loops. Also $\Omega(n^3) \Rightarrow \Theta(n^3)$.

Step 4: Construct an optimal solution

With the s table filled in, recursively print an optimal solution.

```
PRINT-OPTIMAL-PARENS$(s, i, j)$
  if $i == j$
      print "$A$"$_i$
  else print "("
      PRINT-OPTIMAL-PARENS$(s, i, s[i,j])$
      PRINT-OPTIMAL-PARENS$(s, s[i,j] + 1, j)$
      print ")"
```

Initial call is PRINT-OPTIMAL-PARENS$(s, 1, n)$

Longest common subsequence

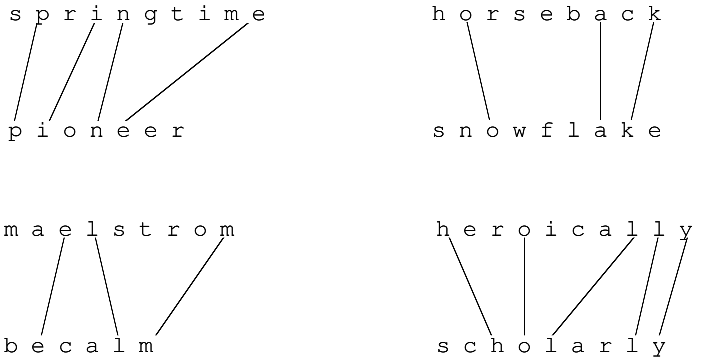
[The textbook has the section on elements of dynamic programming next, but these lecture notes reserve that section for the end of the chapter so that it may refer to two more examples of dynamic programming.]

Problem: Given two 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 common subsequence of two strings appears at http://www.cs.dartmouth.edu/~thc/code/lcs.c The comments in the code refer to the second edition of the textbook, but the code is correct.]*

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.

Step 1: Characterize an LCS

Notation:

$$X_i = \text{prefix } \langle x_1, \ldots, x_i \rangle$$
$$Y_i = \text{prefix } \langle y_1, \ldots, y_i \rangle$$

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 $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then 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.

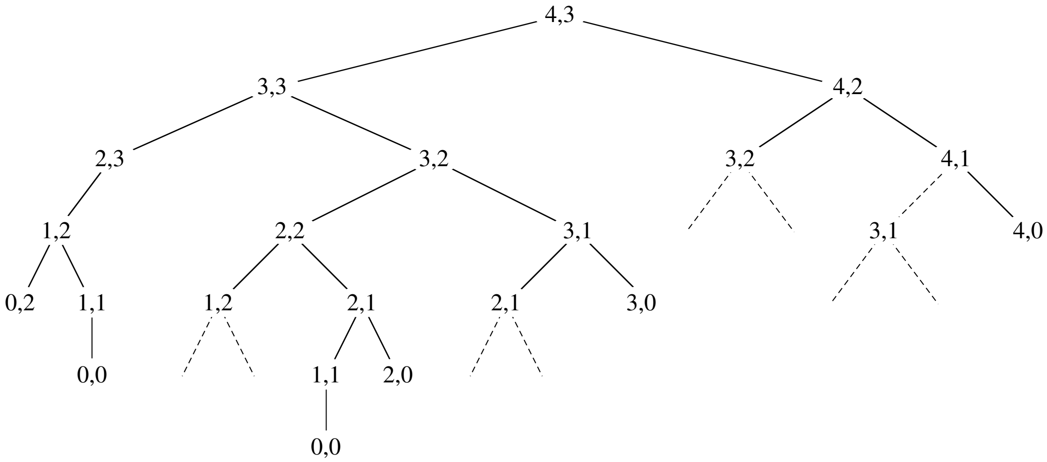
Step 2: Recursively define an optimal solution

Define $c[i, j]$ = length of LCS of X_i and Y_j. 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, could write a recursive algorithm based on this formulation.

Try with $X = \langle a, t, o, m \rangle$ and $Y = \langle a, n, t \rangle$. Numbers in nodes are values of i, j in each recursive call. Dashed lines indicate subproblems already computed.



- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

Step 3: Compute the length of an LCS

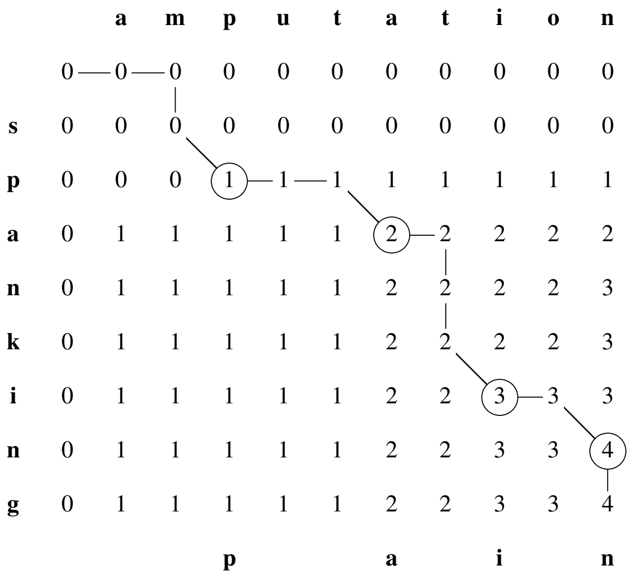
```
LCS-LENGTH($X, Y, m, n$)
  let $b[1:m, 1:n]$ and $c[0:m, 0:n]$ be new tables
  for $i = 1$ to $m$
      $c[i, 0] = 0$
  for $j = 0$ to $n$
      $c[0, j] = 0$
  for $i = 1$ to $m$          // compute table entries in row-major order
      for $j = 1$ to $n$
          if $x_i == y_j$
              $c[i, j] = c[i-1, j-1] + 1$
              $b[i, j] =$ "$\nwarrow$"
          else if $c[i-1, j] \geq c[i, j-1]$
                  $c[i, j] = c[i-1, j]$
                  $b[i, j] =$ "$\uparrow$"
              else $c[i, j] = c[i, j-1]$
                  $b[i, j] =$ "$\leftarrow$"
  return $c$ and $b$
```

```
PRINT-LCS($b, X, i, j$)
  if $i == 0$ or $j = 0$
      return                  // the LCS has length 0
  if $b[i, j] ==$ "$\nwarrow$"
      PRINT-LCS($b, X, i-1, j-1$)
      print $x_i$             // same as $y_j$
  elseif $b[i, j] ==$ "$\uparrow$"
      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 was used in solving LCS of X_i and Y_j.
- When $b[i, j] = \nwarrow$, LCS extended by one character. So longest common subsequence = entries with $\nwarrow$ in them.

Demonstration

What do `spanking` and `amputation` have in common? *[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**

Answer: `pain`.

Time

$\Theta(mn)$

Improving the code

Don't really need the b table. $c[i, j]$ depends only on $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given $c[i, j]$, can determine in constant time which of the three values was used to compute $c[i, j]$. *[Exercise 14.4-2.]*

Or, if only need the length of an LCS, and don't need to construct the LCS itself, can get away with storing only one row of the c table plus a constant amount of additional entries. *[Exercise 14.4-4.]*

Optimal binary search trees

- 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]

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

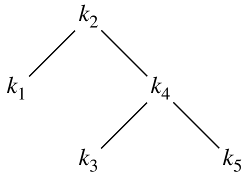
$$
\begin{aligned}
&= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \\
&= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i \qquad \text{(since probabilities sum to 1)} \qquad (*)
\end{aligned}
$$

[Keep equation (∗) on board.]

[Similar to optimal BST problem in the textbook, but simplified here: we assume that all searches are successful. Textbook has probabilities of searches between keys in tree.]

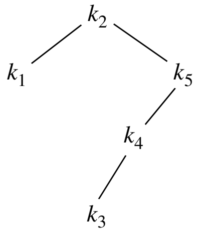
Example

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3



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, E [search cost] = 2.10, which turns out to be optimal.

Observations

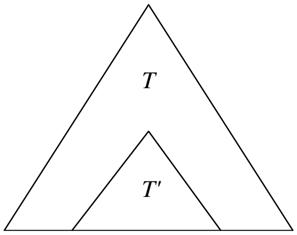
- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.

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.

Step 1: The structure of an optimal binary search tree

Consider any subtree of a BST. It contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.

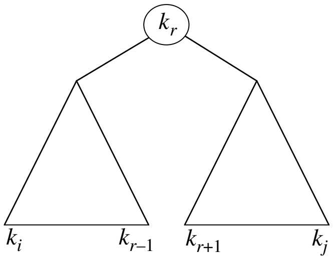


If T is an optimal BST and T contains subtree T' with keys $k_i, \ldots, k_j$, then T' must be an optimal BST for keys $k_i, \ldots, k_j$.

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
  - you examine all candidate roots k_r, for $i \le r \le j$, and
  - you determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$,

  then you're guaranteed to find an optimal BST for $k_i, \ldots, k_j$.

Step 2: Recursive solution

Subproblem domain:

- Find optimal BST for $k_i, \ldots, k_j$, where $i \geq 1$, $j \leq n$, $j \geq i - 1$.
- 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 $j \geq i$,

- Select a root k_r, for some $i \leq r \leq j$.
- 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 $k_{j+1}, \ldots, k_j$. These subtreees are empty.

When a subtree becomes a subtree of a node:

- Depth of every node in subtree goes up by 1.
- Expected search cost increases by

$$w(i, j) = \sum_{l=i}^{j} p_l \qquad \text{(refer to equation } (*)) \ .$$

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:

$$e[i, j] = \begin{cases} 0 & \text{if } j = i - 1 \ , \\ \min\{e[i, r-1] + e[r+1, j] + w(i, j) : i \leq r \leq j\} & \text{if } i \leq j \ . \end{cases}$$

Could write a recursive algorithm...

Step 3: Computing the expected search cost of an optimal binary search tree

As "usual," store the values in a table:

$$e[\ \underbrace{1:n+1}_{\substack{\text{can store} \\ e[n+1, n]}}\ ,\ \underbrace{0:n}_{\substack{\text{can store} \\ e[1, 0]}}\ ]$$

- Will use only entries $e[i, j]$, where $j \geq i - 1$.

- 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)
  let e[1 : n + 1, 0 : n], w[1 : n + 1, 0 : n], and root[1 : n, 1 : n] be new tables
  for i = 1 to n + 1              // base cases
      e[i, i − 1] = 0
      w[i, i − 1] = 0
  for l = 1 to n
      for i = 1 to n − l + 1
          j = i + l − 1
          e[i, j] = ∞
          w[i, j] = w[i, j − 1] + p_j
          for r = i to j              // try all possible roots r
              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
              if t < e[i, j]          // new minimum?
                  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:

					j		
	e	0	1	2	3	4	5
	1	0	.25	.65	.8	1.25	2.10
	2		0	.2	.3	.75	1.35
	3			0	.05	.3	.85
i	4		p_i		0	.2	.7
	5					0	.3
	6						0

				j			
	w	0	1	2	3	4	5
i	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

				j		
	root	1	2	3	4	5
i	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, $\Theta(n^3)$.

Step 4: Construct an optimal binary search tree

[Exercise 14.5-1 asks to write this pseudocode.]

```
CONSTRUCT-OPTIMAL-BST(root)
  $r = root[1, n]$
  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 $i \leq j$
      $t = root[i, j]$
      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 more 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 the dynamic-programming gods tell 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. *[The dynamic-programming gods are 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

Rod cutting

- Space of subproblems was rods of length $n - i$, for $1 \le i \le n$.
- No need to try a more general space of subproblems.

Matrix-chain multiplication

- Suppose we had tried to constrain the space of subproblems to parenthesizing $A_1 A_2 \cdots A_j$.
- An optimal parenthesization splits at some matrix A_k.
- Get subproblems for $A_1 \cdots A_k$ and $A_{k+1} \cdots A_j$.
- Unless we could guarantee that $k = j - 1$, so that the subproblem for $A_{k+1} \cdots A_j$ has only A_j, then this subproblem is *not* of the form $A_1 A_2 \cdots A_j$.
- Thus, needed to allow the subproblems to vary at both ends—allow both i and j to vary.

Longest commmon subsequence

- Space of subproblems for $\langle x_1, \ldots, x_i \rangle$ and $\langle y_1, \ldots, y_j \rangle$ was just $\langle x_1, \ldots, x_{i-1} \rangle$ and $\langle y_1, \ldots, y_{j-1} \rangle$.
- No need to try a more general space of subproblems.

Optimal binary search trees

- Similar to matrix-chain multiplication.
- 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.

- Rod cutting:
  - 1 subproblem (of size $n - i$)
  - n choices
- Matrix-chain multiplication:
  - 2 subproblems ($A_i \cdots A_k$ and $A_{k+1} \cdots A_j$)
  - $j - i$ choices for A_k in $A_i, A_{i+1}, \ldots, A_{j-1}$. Having found optimal solutions to subproblems, choose from among the $j - i$ candidates for A_k.
- 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 $k_i, \ldots, k_j$. Having found optimal solutions to subproblems, choose from among the $j - i + 1$ candidates for k_r.

Informally, running time depends on (# of subproblems overall) × (# of choices).

- Rod cutting: $\Theta(n)$ subproblems, $\le n$ choices for each $\Rightarrow O(n^2)$ running time.
- Matrix-chain multiplication: $\Theta(n^2)$ subproblems, $O(n)$ choices for each $\Rightarrow O(n^3)$ 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.

Can use the subproblem graph to get the same analysis: count the number of edges.

- Each vertex corresponds to a subproblem.
- Choices for a subproblem are vertices that the subproblem has edges going to.
- For rod cutting, subproblem graph has n vertices and $\leq n$ edges per vertex $\Rightarrow O(n^2)$ running time.
  In fact, can get an exact count of the edges: for $i = 0, 1, \ldots, n$, vertex for subproblem size i has out-degree $i \Rightarrow$ # of edges $= \sum_{i=0}^{n} i = n(n+1)/2$.
- Subproblem graph for matrix-chain multiplication has $\Theta(n^2)$ vertices, each with degree $\leq n - 1$
  $\Rightarrow O(n^3)$ 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.

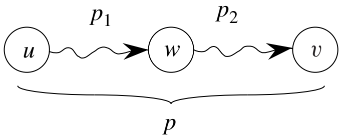
Here are two problems that look similar. In both, we're given an *unweighted, directed graph* $G = (V, E)$.

- 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 a 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 a *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 going $u \rightsquigarrow w$.
- Then p_1 is a shortest path $u \rightsquigarrow w$.

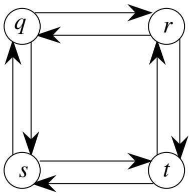
Proof Suppose there exists a shorter path p'_1 going $u \leadsto w$. Cut out p_1, replace it with p'_1, get path $u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$ with fewer edges than p. ■

Therefore, can find shortest path $u \leadsto v$ by considering all intermediate vertices w, then finding shortest paths $u \leadsto w$ and $w \leadsto v$.

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 $q \leadsto t$. Are its subpaths longest paths?

No!

- Subpath $q \leadsto r$ is $q \to r$.
- Longest simple path $q \leadsto r$ is $q \to s \to t \to r$.
- Subpath $r \leadsto t$ is $r \to t$.
- Longest simple path $r \leadsto t$ is $r \to q \to s \to t$.

Not only isn't there optimal substructure, but 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 you do, the path isn't simple.
- But you *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, for 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, get a cycle.]*

Independent subproblems in our examples:

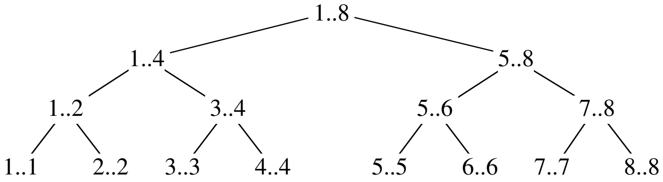
- Rod cutting and longest common subsequence
  - 1 subproblem ⇒ automatically independent.
- Matrix-chain multiplication
  - $A_i \cdots A_k$ and $A_{k+1} \cdots A_j \Rightarrow$ independent.
- Optimal binary search tree
  - $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j \Rightarrow$ independent.

Overlapping subproblems

These occur when a recursive algorithm revisits the same problem over and over.

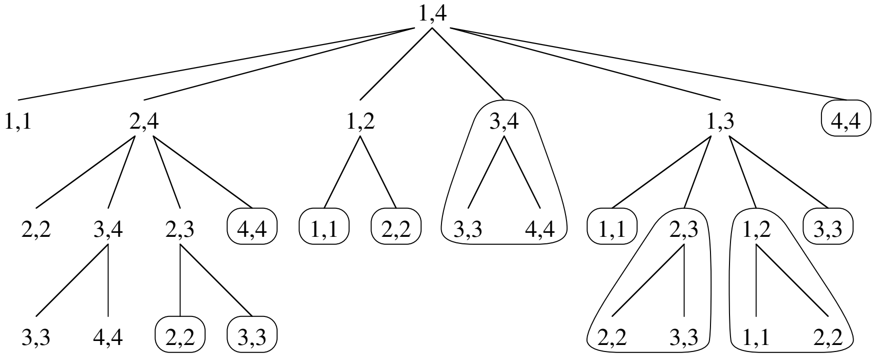
Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

Example: merge sort



Alternative approach to dynamic programming: ***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.
- For matrix-chain multiplication:



Each node has the parameters i and j. Computations performed in highlighted subtrees are replaced by a single table lookup if computing recursively with memoization.

- In bottom-up dynamic programming, we go one step further. Determine in what order to access the table, and fill it in that way.

Solutions for Chapter 14: Dynamic Programming

Solution to Exercise 14.1-1

We can verify that $T(n) = 2^n$ is a solution to the given recurrence by the substitution method. We note that for $n = 0$, the formula is true since $2^0 = 1$. For $n > 0$, substituting into the recurrence and using the formula for summing a geometric series yields

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

Solution to Exercise 14.1-2

Here is a counterexample for the "greedy" strategy:

length i	1	2	3	4
price p_i	1	20	33	36
p_i/i	1	10	11	9

Let the given rod length be 4. According to a greedy strategy, we first cut out a rod of length 3 for a price of 33, which leaves us with a rod of length 1 of price 1. The total price for the rod is 34. The optimal way is to cut it into two rods of length 2 each fetching us 40 dollars.

Solution to Exercise 14.1-3

```
MODIFIED-CUT-ROD(p, n, c)
  let r[0 : n] be a new array
  r[0] = 0
  for j = 1 to n
      q = p[j]
      for i = 1 to j - 1
          q = max(q, p[i] + r[j - i] - c)
      r[j] = q
  return r[n]
```

The major modification required is in the body of the inner **for** loop, which now reads $q = \max(q, p[i] + r[j-i] - c)$. This change reflects the fixed cost of making the cut, which is deducted from the revenue. We also have to handle the case in which we make no cuts (when i equals j); the total revenue in this case is simply $p[j]$. Thus, we modify the inner **for** loop to run from i to $j-1$ instead of to j. The assignment $q = p[j]$ takes care of the case of no cuts. If we did not make these modifications, then even in the case of no cuts, we would be deducting c from the total revenue.

Solution to Exercise 14.1-4

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = p[n]
  for i = 1 to ⌊n/2⌋
      q = max {q, p[i] + CUT-ROD(p, n - i)}
  return q
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  q = p[n]
  for i = 1 to ⌊n/2⌋
      q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
  r[n] = q
  return q
```

Note that in addition to changing the loop bounds, for both procedures, instead of initializing $q = -\infty$, initialize $q = p[n]$. In MEMOIZED-CUT-ROD-AUX, we can also remove the case for $n == 0$.

The memoized code still takes $\Theta(n^2)$ time. The running time for the recursive CUT-ROD procedure reduces to $\Theta(2^{n/2})$.

Solution to Exercise 14.1-5

```
MEMOIZED-CUT-ROD(p, n)
  let $r[0:n]$ and $s[0:n]$ be new arrays
  for $i = 0$ to $n$
      $r[i] = -\infty$
  $(val, s) =$ MEMOIZED-CUT-ROD-AUX$(p, n, r, s)$
  print "The optimal value is " $val$ " and the cuts are at "
  $j = n$
  while $j > 0$
      print $s[j]$
      $j = j - s[j]$

MEMOIZED-CUT-ROD-AUX(p, n, r, s)
  if $r[n] \geq 0$
      return $r[n]$
  if $n == 0$
      $q = 0$
  else $q = -\infty$
      for $i = 1$ to $n$
          $(val, s) =$ MEMOIZED-CUT-ROD-AUX$(p, n - i, r, s)$
          if $q < p[i] + val$
              $q = p[i] + val$
              $s[n] = i$
  $r[n] = q$
  return $(q, s)$
```

PRINT-CUT-ROD-SOLUTION constructs the actual lengths where a cut should happen. Array entry $s[i]$ contains the value j indicating that an optimal cut for a rod of length i is j inches. The next cut is given by $s[i - j]$, and so on.

Solution to Exercise 14.1-6

```
FIBONACCI(n)
  let $fib[0:n]$ be a new array
  $fib[0] = fib[1] = 1$
  for $i = 2$ to $n$
      $fib[i] = fib[i-1] + fib[i-2]$
  return $fib[n]$
```

FIBONACCI directly implements the recurrence relation of the Fibonacci sequence. Each number in the sequence is the sum of the two previous numbers in the sequence. The running time is clearly $O(n)$.

The subproblem graph consists of $n + 1$ vertices, $v_0, v_1, \ldots, v_n$. For $i = 2, 3, \ldots, n$, vertex v_i has two leaving edges: to vertex v_{i-1} and to vertex v_{i-2}. No edges leave vertices v_0 or v_1. Thus, the subproblem graph has $2n - 2$ edges.

Solution to Exercise 14.2-1

The *m* and *s* tables:

m					j		
		1	2	3	4	5	6
	1	0	150	330	405	1655	2010
	2		0	360	330	2430	1950
i	3			0	180	930	1770
	4				0	3000	1860
	5					0	1500
	6						0

s				j		
		2	3	4	5	6
	1	1	2	2	4	2
	2		2	2	2	2
i	3			3	4	4
	4				4	4
	5					5

The optimal parenthesization printed is $((A_1 A_2)((A_3 A_4)(A_5 A_6)))$.

Solution to Exercise 14.2-2

```
MATRIX-CHAIN-MULTIPLY($A, s, i, j$)
  if $j > i$
      $X$ = MATRIX-CHAIN-MULTIPLY($A, s, i, s[i, j]$)
      $Y$ = MATRIX-CHAIN-MULTIPLY($A, s, s[i, j] + 1, j$)
      return MATRIX-MULTIPLY($X, Y$)
  else return $A_i$
```

Solution to Exercise 14.2-3

We show that $P(n) \geq 2^{n-2}$ for all $n \geq 1$. The base cases are $n = 1, 2, 3, 4$, where we have $P(1) = 1 > 2^{-1}$, $P(2) = 1 = 2^0$, $P(3) = 2 = 2^1$, and $P(4) = 5 > 2^2$. For the inductive step, $n \geq 5$, and we assume that $P(k) \geq 2^{k-2}$ for all $1 \leq k < n$. Then

$$\begin{aligned} P(n) &= \sum_{k=1}^{n-1} P(k)P(n-k) \\ &\geq \sum_{k=1}^{n-1} 2^{k-2} \cdot 2^{n-k-2} \end{aligned}$$

$$\begin{aligned}
&= \sum_{k=1}^{n-1} 2^{n-4} \\
&\geq 4 \cdot 2^{n-4} \qquad \text{(because } n \geq 5\text{)} \\
&= 2^{n-2} .
\end{aligned}$$

Solution to Exercise 14.2-4

The vertices of the subproblem graph are the ordered pairs v_{ij}, where $i \leq j$. If $i = j$, then there are no edges out of v_{ij}. If $i < j$, then for every k such that $i \leq k < j$, the subproblem graph contains edges (v_{ij}, v_{ik}) and $(v_{ij}, v_{k+1,j})$. These edges indicate that to solve the subproblem of optimally parenthesizing the product $A_i \cdots A_j$, we need to solve subproblems of optimally parenthesizing the products $A_i \cdots A_k$ and $A_{k+1} \cdots A_j$. The number of vertices is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} 1 = \frac{n(n+1)}{2} ,$$

and the number of edges is

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j=i}^{n} (j-i) &= \sum_{i=1}^{n} \sum_{t=0}^{n-i} t \qquad \text{(substituting } t = j - i\text{)} \\
&= \sum_{i=1}^{n} \frac{(n-i)(n-i+1)}{2} .
\end{aligned}$$

Substituting $r = n - i$ and reversing the order of summation, we obtain

$$\begin{aligned}
&\sum_{i=1}^{n} \frac{(n-i)(n-i+1)}{2} \\
&= \frac{1}{2} \sum_{r=0}^{n-1} (r^2 + r) \\
&= \frac{1}{2} \left(\frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} \right) \qquad \text{(by equations (A.4) and (A.1))} \\
&= \frac{(n-1)n(n+1)}{6} .
\end{aligned}$$

Thus, the subproblem graph has $\Theta(n^2)$ vertices and $\Theta(n^3)$ edges.

Solution to Exercise 14.2-5
This solution is also posted publicly

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 $\sum_{l=2}^{n} 2(n-l+1)(l-1)$. Thus,

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) &= \sum_{l=2}^{n} 2(n-l+1)(l-1) \\
&= 2\sum_{l=1}^{n-1}(n-l)l \\
&= 2\sum_{l=1}^{n-1} nl - 2\sum_{l=1}^{n-1} l^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3} .
\end{aligned}$$

Solution to Exercise 14.2-6

Each multiplication corresponds to one pair of parentheses. Multiplying n matrices entails $n-1$ multiplications. Ergo, $n-1$ pairs of parentheses.

Solution to Exercise 14.3-1
This solution is also posted publicly

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 14.2 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 14.2 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 will 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 will 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 $n \ge 2$ we have

$$\begin{aligned} T(n) &\le 2 \sum_{i=1}^{n-1} T(i) + cn \\ &\le 2 \sum_{i=1}^{n-1} ci3^{i-1} + cn \\ &= c \cdot \left(2 \sum_{i=1}^{n-1} i3^{i-1} + n \right) \\ &= c \cdot \left(2 \cdot \left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2} \right) + n \right) \qquad \text{(see below)} \\ &= cn3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n \right) \\ &= cn3^{n-1} + \frac{c}{2}(2n + 1 - 3^n) \\ &\le cn3^{n-1} \quad \text{for all } c > 0, n \ge 1 \,. \end{aligned}$$

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 following fact:

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} \,.$$

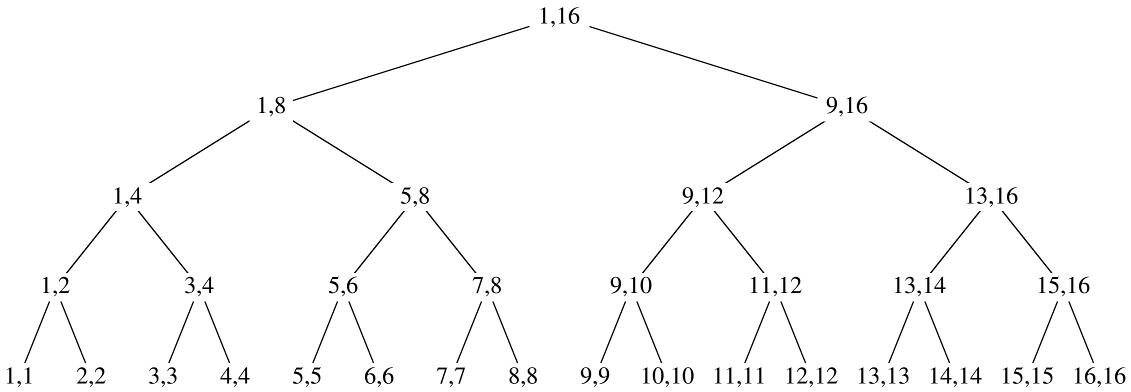
This equation can be derived from equation (A.6) by taking the derivative. Let

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n - 1}{x-1} - 1 \,.$$

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 14.3-2



Looking at the recursion tree for MERGE-SORT, we see that there are no overlapping subproblems: every problem solved is unique. Therefore, memoization would not decrease the amount of work done and would fail to speed up MERGE-SORT. In general, good divide-and-conquer algorithms consider each subproblem only once, so that memoization does not speed them up.

Solution to Exercise 14.3-3

Yes, this problem exhibits optimal substructure.

Since the subproblems are independent, a cut-and-paste argument works. Suppose that we have a parenthesization P of a sequence of matrices $A_1, \ldots, A_j$ that maximizes the number of scalar multiplications. Split the sequence between any two matrices A_k and A_{k+1}. If this problem did not exhibit optimal substructure, there could be a parenthesization $A_1, \ldots, A_k$ with even more scalar multiplications. We could then replace the parenthesization of $A_1, \ldots, A_k$ in P to achieve an even greater number of scalar multiplications, contradicting our assumption.

Solution to Exercise 14.3-4

Let $p_0 = 1$, $p_1 = 1$, $p_2 = 10$, and $p_3 = 2$.

By multiplying $(A_1 A_2) A_3$, the number of scalar multiplications is $10 + 20 = 30$. By multiplying $A_1(A_2 A_3)$, the number of scalar multiplications is $20 + 2 = 22$. Choosing k before solving subproblems leads to the first way, since the first matrix multiplication would cost only 10 scalar multiplications, rather than costing 20. The other parenthesization would require fewer scalar multiplications, however, so that this greedy approach is suboptimal.

Solution to Exercise 14.3-5

A problem exhibits the optimal substructure property when optimal solutions to a problem incorporate optimal solutions to related subproblems, *which may be solved independently* (i.e., they do not share resources). When the number of pieces of size i that may be produced is limited, the subproblems can no longer be solved *independently*. For example, consider a rod of length 4 with the following prices and limits:

length i	1	2	3	4
price p_i	15	20	33	36
limit l_i	2	1	1	1

This instance has only three solutions that do not violate the limits: length 4 with price 36; lengths 1 and 3 with price 48; and lengths 1, 1, and 2 with price 50. The optimal solution, therefore is to cut into lengths 1, 1, and 2. The subproblem for length 2 has two solutions that do not violate the limits: length 2 with price 20, and lengths 1 and 1 with price 30. The optimal solution for length 2, therefore, is to cut into lengths 1 and 1. But this optimal solution for the subproblem cannot be part of the optimal solution for the original problem, because it would result in using four rods of length 1 to solve the original problem, violating the limit of two length-1 rods.

Solution to Exercise 14.4-1

The LCS-LENGTH procedure finds the LCS $\langle 1, 0, 0, 1, 1, 0\rangle$. The sequences $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$ have four other LCSs: $\langle 0, 1, 0, 1, 0, 1\rangle$, $\langle 1, 0, 1, 0, 1, 0\rangle$, $\langle 1, 0, 1, 0, 1, 1\rangle$, and $\langle 1, 0, 1, 1, 0, 1\rangle$.

Solution to Exercise 14.4-2

```
RECONSTRUCT-LCS$(c, X, Y, i, j)$
  if $i == 0$ or $j == 0$
      return
  if $x_i == y_j$
      RECONSTRUCT-LCS$(c, X, Y, i - 1, j - 1)$
      print $x_i$
  elseif $c[i, j] == c[i - 1, j]$
      RECONSTRUCT-LCS$(c, X, Y, i - 1, j)$
  else RECONSTRUCT-LCS$(c, X, Y, i, j - 1)$
```

This procedure emulates the PRINT-LCS procedure. When x_i and y_i are the same, then the LCS must have been extended to add this symbol, so that the procedure prints the LCS of X_{i-1} and Y_{j-1} and then prints x_i. Otherwise, $c[i, j]$ must equal either $c[i-1, j]$ or $c[i, j-1]$ (or both). If $c[i, j]$ equals $c[i-1, j]$, then the LCS-LENGTH procedure would have put "↑" into $b[i, j]$, and so RECONSTRUCT-LCS recurses with $i-1$ and j. Otherwise, LCS-LENGTH would have put "←" into $b[i, j]$, and so RECONSTRUCT-LCS recurses with i and $j-1$.

Solution to Exercise 14.4-3

```
MEMOIZED-LCS-LENGTH$(X, Y, m, n)$
  let $c[1:m, 1:n]$ be a new table
  for $i = 0$ to $m$
      for $j = 0$ to $n$
          $c[i, j] =$ NIL
  return LOOKUP-LCS$(c, m, n)$

LOOKUP-LCS$(c, i, j)$
  if $c[i, j] \neq$ NIL
      return $c[i, j]$
  if $x_i == y_j$
      $c[i, j] =$ LOOKUP-LCS$(i-1, j-1) + 1$
  else $c[i, j] = \max\{$LOOKUP-LCS$(i-1, j)$, LOOKUP-LCS$(i, j-1)\}$
  return $c[i, j]$
```

Solution to Exercise 14.4-4
This solution is also posted publicly

When computing a particular row of the c table, no rows before the previous row are needed. Thus only two rows—$2n$ entries—need to be kept in memory at a time. (Note: Each row of c actually has $n+1$ entries, but we don't need to store the column of 0s—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 $k \leq j-1$ (i.e., earlier entries in the current row, which will be needed to compute the next row), and $c[i-1, k]$ for $k \geq j-1$ (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 holds 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 three 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 Exercise 14.4-5

The longest monotonically increasing subsequence of X is the longest common subsequence of X and a sorted version of X. Therefore, to find the longest monotonically increasing subsequence of X, do the following: sort X, producing a sequence X', and find the longest common subsequence of X and X'.

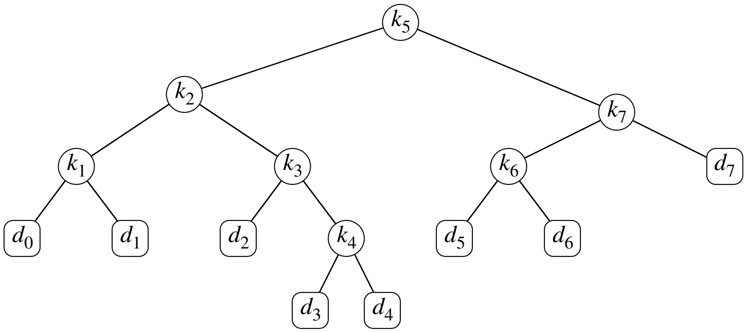
The sorting time is $O(n^2)$ (in fact, $O(n \lg n)$ using merge sort or heapsort), and the time to find the longest common subsequence is $O(n^2)$, since both sequences have length n.

Solution to Exercise 14.5-1

```
CONSTRUCT-OPTIMAL-BST($root, n$)
  $r = root[1, n]$
  print "$k$"$_r$ " is the root"
  CONSTRUCT-OPTIMAL-SUBTREE($root, 1, r - 1, r$, "left")
  CONSTRUCT-OPTIMAL-SUBTREE($root, r + 1, n, r$, "right")

CONSTRUCT-OPTIMAL-SUBTREE($root, i, j, r, dir$)
  if $j < i$
      print "$d$"$_j$ " is the " $dir$ " child of $k$"$_r$
  else $t = root[i, j]$
      print "$k$"$_t$ " is the " $dir$ " child of $k$"$_r$
      CONSTRUCT-OPTIMAL-SUBTREE($root, i, t - 1, t$, "left")
      CONSTRUCT-OPTIMAL-SUBTREE($root, t + 1, j, t$, "right")
```

Solution to Exercise 14.5-2

The optimal binary search tree has cost 3.12 and this structure:



Here are the *e* and *root* tables:

e		*j*							
		0	1	2	3	4	5	6	7
	1	0.06	0.28	0.62	1.02	1.34	1.83	2.44	3.12
	2		0.06	0.30	0.68	0.93	1.41	1.96	2.61
	3			0.06	0.32	0.57	1.04	1.48	2.13
i	4				0.06	0.24	0.57	1.01	1.55
	5					0.05	0.30	0.72	0.78
	6						0.05	0.32	0.78
	7							0.05	0.34
	8								0.05

root		*j*						
		1	2	3	4	5	6	7
	1	1	2	2	2	3	3	5
	2		2	3	3	3	5	5
	3			3	3	4	5	5
i	4				4	5	5	6
	5					5	6	6
	6						6	7
	7							7

Solution to Exercise 14.5-3

Computing $w(i, j)$ directly in line 9 would not change the running time at all. The innermost **for** loop of lines 10–14 already runs from i to j, so that changing line 9 into **for** loops running from i to j and from $i - 1$ to j would not change the asymptotic running time of OPTIMAL-BST.

Solution to Problem 14-1

We will make use of the optimal substructure property of longest paths in *acyclic* graphs. Let u be some vertex of the graph. If $u = t$, then the longest path from u

to t has zero weight. If $u \neq t$, let p be a longest path from u to t. Path p has at least two vertices. Let v be the second vertex on the path. Let p' be the subpath of p from v to t (p' might be a zero-length path). That is, the path p looks like $u \to v \overset{p'}{\leadsto} t$.

We claim that p' is a longest path from v to t.

To prove the claim, we use a cut-and-paste argument. If p' were not a longest path, then there exists a longer path p'' from v to t. We could cut out p' and paste in p'' to produce a path $u \to v \overset{p''}{\leadsto} t$ which is longer than p, thus contradicting the assumption that p is a longest path from u to t.

It is important to note that the graph is *acyclic*. Because the graph is acyclic, path p'' cannot include the vertex u, for otherwise there would be a cycle of the form $u \to v \leadsto u$ in the graph. Thus, we can indeed use p'' to construct a longer path. The acyclicity requirement ensures that by pasting in path p'', the overall path is still a *simple* path (there is no cycle in the path). This difference between the cyclic and the acyclic case allows us to use dynamic programming to solve the acyclic case.

Let *dist*$[u]$ denote the weight of a longest path from u to t. The optimal substructure property allows us to write a recurrence for *dist*$[u]$ as

$$
\mathit{dist}[u] = \begin{cases} 0 & \text{if } u = t \ , \\ \max\{w(u,v) + \mathit{dist}[v] : (u,v) \in E\} & \text{otherwise} \ . \end{cases}
$$

This recurrence allows us to construct the following procedure:

```
LONGEST-PATH-AUX($G, u, t, dist, next$)
  if $u == t$
      $dist[u] = 0$
      return ($dist, next$)
  elseif $next[u] \neq$ NIL
      return ($dist, next$)
  else for each vertex $v \in G.Adj[u]$
          ($dist, next$) = LONGEST-PATH-AUX($G, v, t, dist, next$)
          if $w(u,v) + dist[v] > dist[u]$
              $dist[u] = w(u,v) + dist[v]$
              $next[u] = v$
  return ($dist, next$)
```

(See Section 20.1 for an explanation of the notation $G.Adj[u]$.)

LONGEST-PATH-AUX is a memoized, recursive procedure, which returns the tuple (*dist*, *next*). The array *dist* is the memoized array that holds the solution to subproblems. That is, after the procedure returns, *dist*$[u]$ will hold the weight of a longest path from u to t. The array *next* serves two purposes:

- It holds information necessary for printing out an actual path. Specifically, if u is a vertex on the longest path that the procedure found, then *next*$[u]$ is the next vertex on the path.
- The value in *next*$[u]$ is used to check whether the current subproblem has been solved earlier. A non-NIL value indicates that this subproblem has been solved earlier.

The first **if** condition checks for the base case $u = t$. The second **if** condition checks whether the current subproblem has already been solved. The **for** loop iterates over each adjacent edge (u, v) and updates the longest distance in *dist*$[u]$.

What is the running time of LONGEST-PATH-AUX? Each subproblem represented by a vertex u is solved at most once due to the memoization. For each vertex, its adjacent edges are examined. Thus, each edge is examined at most once, and the overall running time is $O(E)$. (Section 20.1 discusses how to achieve $O(E)$ time by representing the graph with adjacency lists.)

The PRINT-PATH procedure prints out the vertices in the path using information stored in the *next* array:

```
PRINT-PATH($s, t, next$)
  $u = s$
  print $u$
  while $u \neq t$
      print $next[u]$
      $u = next[u]$
```

The LONGEST-PATH-MAIN procedure is the main driver. It creates and initializes the *dist* and the *next* arrays. It then calls LONGEST-PATH-AUX to find a path and PRINT-PATH to print out the actual path.

```
LONGEST-PATH-MAIN($G, s, t$)
  $n = |G.V|$
  let $dist[1:n]$ and $next[1:n]$ be new arrays
  for $i = 1$ to $n$
      $dist[i] = -\infty$
      $next[i] =$ NIL
  $(dist, next) =$ LONGEST-PATH-AUX($G, s, t, dist, next$)
  if $dist[s] == -\infty$
      print “No path exists”
  else print “The weight of the longest path is ” $dist[s]$
      PRINT-PATH($s, t, next$)
```

Initializating the *dist* and *next* arrays takes $O(V)$ time. Thus, the overall running time of LONGEST-PATH-MAIN is $O(V + E)$.

Alternative solution

We can also solve the problem using a bottom-up aproach. To do so, we need to ensure that we solve “smaller” subproblems before we solve “larger” ones. In our case, we can use a topological sort (see Section 20.4) to obtain a bottom-up procedure, imposing the required ordering on the vertices in $\Theta(V + E)$ time.

```
LONGEST-PATH-BOTTOM-UP$(G, s, t)$
  let $dist[1:n]$ and $next[1:n]$ be new arrays
  topologically sort the vertices of $G$
  for $i = 1$ to $|G.V|$
      $dist[i] = -\infty$
  $dist[s] = 0$
  for each $u$ in topological order, starting from $s$
      for each edge $(u, v) \in G.Adj[u]$
          if $dist[u] + w(u, v) > dist[v]$
              $dist[v] = dist[u] + w(u, v)$
              $next[u] = v$
  print "The longest distance is " $dist[t]$
  PRINT-PATH$(s, t, next)$
```

The running time of LONGEST-PATH-BOTTOM-UP is $\Theta(V + E)$.

Solution to Problem 14-2

We solve the longest palindrome subsequence (LPS) problem in a manner similar to how we compute the longest common subsequence in Section 14.4.

Step 1: Characterizing a longest palindrome subsequence

The LPS problem has an optimal-substructure property, where the subproblems correspond to pairs of indices, starting and ending, of the input sequence.

For a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, we denote the subsequence starting at x_i and ending at x_j by $X_{ij} = \langle x_i, x_{i+1}, \ldots, x_j \rangle$.

Theorem (Optimal substructure of an LPS)
Let $X = \langle x_1, x_2, \ldots, x_n \rangle$ be the input sequence, and let $Z = \langle z_1, z_2, \ldots, z_m \rangle$ be any LPS of X.

1. If $n = 1$, then $m = 1$ and $z_1 = x_1$.
2. If $n = 2$ and $x_1 = x_2$, then $m = 2$ and $z_1 = z_2 = x_1 = x_2$.
3. If $n = 2$ and $x_1 \neq x_2$, then $m = 1$ and z_1 is equal to either x_1 or x_n.
4. If $n > 2$ and $x_1 = x_n$, then $m > 2$, $z_1 = z_m = x_1 = x_n$, and $Z_{2,m-1}$ is an LPS of $X_{2,n-1}$.
5. If $n > 2$, $x_1 \neq x_n$, and $z_1 \neq x_1$, then $Z_{1,m}$ is an LPS of $X_{2,n}$.
6. If $n > 2$, $x_1 \neq x_n$, and $z_m \neq x_n$, then $Z_{1,m}$ is an LPS of $X_{1,n-1}$.

Proof Properties (1), (2), and (3) follow trivially from the definition of LPS.

(4) If $n > 2$ and $x_1 = x_n$, then we can choose x_1 and x_n as the ends of Z and at least one more element of X as part of Z. Thus, it follows that $m > 2$. If $z_1 \neq x_1$, then we could append $x_1 = x_n$ to the ends of Z to obtain a palindrome subsequence of X with length $m + 2$, contradicting the supposition that Z is a

longest palindrome subsequence of X. Thus, we must have $z_1 = x_1$ $(= x_n = z_m)$. Now, $Z_{2,m-1}$ is a length-$(m-2)$ palindrome subsequence of $X_{2,n-1}$. We wish to show that it is an LPS. Suppose for the purpose of contradiction that there exists a palindrome subsequence W of $X_{2,n-1}$ with length greater than $m-2$. Then, appending $x_1 = x_n$ to the ends of W produces a palindrome subsequence of X whose length is greater than m, which is a contradiction.

(5) If $z_1 \neq x_1$, then Z is a palindrome subsequence of $X_{2,n}$. If there were a palindrome subsequence W of $X_{2,n}$ with length greater than m, then W would also be a palindrome subsequence of X, contradicting the assumption that Z is an LPS of X.

(6) The proof is symmetric to (2). ■

The way that the theorem characterizes longest palindrome subsequences tells us that an LPS of a sequence contains within it an LPS of a subsequence of the sequence. Thus, the LPS problem has an optimal-substructure property.

Step 2: A recursive solution

The theorem implies that we should examine either one or two subproblems when finding an LPS of $X = \langle x_1, x_2, \ldots, x_n \rangle$, depending on whether $x_1 = x_n$.

Let us define $p[i, j]$ to be the length of an LPS of the subsequence X_{ij}. If $i = j$, the LPS has length 1. If $j = i + 1$, then the LPS has length either 1 or 2, depending on whether $x_i = x_j$. The optimal substructure of the LPS problem gives the following recursive formula:

$$p[i, j] = \begin{cases} 1 & \text{if } i = j \ , \\ 2 & \text{if } j = i + 1 \text{ and } x_i = x_j \ , \\ 1 & \text{if } j = i + 1 \text{ and } x_i \neq x_j \ , \\ p[i+1, j-1] + 2 & \text{if } j > i + 1 \text{ and } x_i = x_j \ , \\ \max\{p[i, j-1], p[i+1, j]\} & \text{if } j > i + 1 \text{ and } x_i \neq x_j \ . \end{cases}$$

Step 3: Computing the length of an LPS

The procedure LONGEST-PALINDROME takes a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ as input. The procedure fills table entries $p[i, i]$, where $1 \leq i \leq n$, and $p[i, i+1]$, where $1 \leq i \leq n-1$, as the base cases. It then starts filling entries $p[i, j]$, where $j > i + 1$. The procedure fills the p table row by row, starting with row $n-2$ and moving toward row 1. (Rows $n-1$ and n are already filled as part of the base cases.) Within each row, the procedure fills the entries from left to right. The procedure also maintains the table $b[1:n, 1:n]$ to help us construct an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $p[i, j]$. The procedure returns the b and p tables. The entry $p[1, n]$ contains the length of an LPS of X. The running time of LONGEST-PALINDROME is clearly $\Theta(n^2)$.

```
LONGEST-PALINDROME$(X, n)$
  let $p[1:n, 1:n]$ and $b[1:n, 1:n]$ be new tables
  for $i = 1$ to $n - 1$
      $p[i,i] = 1$
      $j = i + 1$
      if $x_i == x_j$
          $p[i,j] = 2$
          $b[i,j] =$ “$\swarrow$”
      else $p[i,j] = 1$
          $b[i,j] =$ “$\downarrow$”
  $p[n,n] = 1$
  for $i = n - 2$ downto 1
      for $j = i + 2$ to $n$
          if $x_i == x_j$
              $p[i,j] = p[i+1, j-1] + 2$
              $b[i,j] =$ “$\swarrow$”
          elseif $p[i+1, j] \geq p[i, j-1]$
              $p[i,j] = p[i+1, j]$
              $b[i,j] =$ “$\downarrow$”
          else $p[i,j] = p[i, j-1]$
              $b[i,j] =$ “$\leftarrow$”
  return $p$ and $b$
```

Step 4: Constructing an LPS

The b table returned by LONGEST-PALINDROME enables us to quickly construct an LPS of $X = \langle x_1, x_2, \ldots, x_m \rangle$. We simply begin at $b[1, n]$ and trace through the table by following the arrows. A “$\swarrow$” in entry $b[i, j]$ means that $x_i = y_j$ are the first and last elements of the LPS that LONGEST-PALINDROME found. The following recursive procedure returns a sequence S that contains an LPS of X. The initial call is GENERATE-LPS$(b, X, 1, n, \langle\rangle)$, where $\langle\rangle$ denotes an empty sequence. Within the procedure, the symbol $\|$ denotes concatenation of a symbol and a sequence.

```
GENERATE-LPS$(b, X, i, j, S)$
  if $i > j$
      return $S$
  elseif $i == j$
      return $S \| x_i$
  elseif $b[i,j] ==$ “$\swarrow$”
      return $x_i \|$ GENERATE-LPS$(b, X, i+1, j-1, S) \| x_i$
  elseif $b[i,j] ==$ “$\downarrow$”
      return GENERATE-LPS$(b, X, i+1, j, S)$
  else return GENERATE-LPS$(b, X, i, j-1, S)$
```

Solution to Problem 14-3

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 $1 \le i \le j \le n$:

$$\begin{aligned} b[1, 2] &= |p_1 p_2| , \\ b[i, j] &= b[i, j-1] + |p_{j-1} p_j| \quad \text{for } i < j-1 , \\ b[j-1, j] &= \min \{ b[k, j-1] + |p_k p_j| : 1 \le k < j-1 \} . \end{aligned}$$

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 that $i = j-1$. 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.) Therefore, when $k < j-1$, the length of $P_{i,j}$ equals $\min \{ b[k, j-1] + |p_k p_j| : 1 \le k < j-1 \}$.

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}, giving

$$b[n, n] = b[n-1, n] + |p_{n-1} p_n| .$$

To reconstruct the points on the shortest bitonic tour, we define $r[i, j]$ to be the index of the immediate predecessor of p_j on a shortest bitonic path $P_{i,j}$. Because the immediate predecessor of p_2 on $P_{1,2}$ is p_1, we know that $r[1,2]$ must be 1. The pseudocode below shows how to compute $b[i,j]$ and $r[i,j]$. It fills in only entries $b[i,j]$ where $1 \le i \le n-1$ and $i+1 \le j \le n$, or where $i = j = n$, and only entries $r[i,j]$ where $1 \le i \le n-2$ and $i+2 \le j \le n$.

```
EUCLIDEAN-TSP($p, n$)
  sort the points so that $\langle p_1, p_2, p_3, \ldots, p_n \rangle$ are in order
          of increasing $x$-coordinate
  let $b[1:n, 2:n]$ and $r[1:n-1, 3:n]$ be new tables
  $b[1,2] = |p_1 p_2|$
  for $j = 3$ to $n$
      for $i = 1$ to $j-2$
          $b[i,j] = b[i,j-1] + |p_{j-1} p_j|$
          $r[i,j] = j-1$
      $b[j-1,j] = \infty$
      for $k = 1$ to $j-2$
          $q = b[k,j-1] + |p_k p_j|$
          if $q < b[j-1,j]$
              $b[j-1,j] = q$
              $r[j-1,j] = k$
  $b[n,n] = b[n-1,n] + |p_{n-1} p_n|$
  return $b$ and $r$
```

To print out the tour found, start at p_n, then print a leftgoing subpath that includes p_{n-1}, from right to left, until arriving at p_1. Then print right-to-left the remaining subpath, which does not include p_{n-1}. For the example in Figure 14.11(b) on page 408, 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 it goes deeper into the recursion, and the left-to-right subpath is printed as it backs out. The initial call is PRINT-TOUR(r, p, n).

```
PRINT-TOUR($r, p, n$)
  print $p_n$
  print $p_{n-1}$
  $k = r[n-1,n]$
  PRINT-PATH($r, p, k, n-1$)
  print $p_k$
```

```
PRINT-PATH($r, p, i, j$)
  if $i < j$
      $k = r[i,j]$
      if $k \ne i$
          print $p_k$
      if $k > 1$
          PRINT-PATH($r, p, i, k$)
  else $k = r[j,i]$
      if $k > 1$
          PRINT-PATH($r, p, k, j$)
          print $p_k$
```

The relative values of the parameters i and j in each call of PRINT-PATH indicate which subpath it's working on. If $i < j$, it's on the right-to-left subpath, and if $i > j$, it's on the left-to-right subpath. The test for $k \neq i$ prevents it from printing p_1 an extra time, which could occur when it calls PRINT-PATH$(r, p, 1, 2)$.

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 14-4
This solution is also posted publicly

We start by defining some quantities 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 $\mathit{extras}[i, j] = M - j + i - \sum_{k=i}^{j} l_k$ 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:

$$\mathit{lc}[i, j] = \begin{cases} \infty & \text{if } \mathit{extras}[i, j] < 0 \text{ (i.e., words } i, \ldots, j \text{ don't fit)} , \\ 0 & \text{if } j = n \text{ and } \mathit{extras}[i, j] \geq 0 \text{ (last line costs 0)} , \\ (\mathit{extras}[i, j])^3 & \text{otherwise} . \end{cases}$$

  By making the line cost infinite when the words don't fit on it, we prevent such an arrangement from being part of a minimum 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 runs from 1 to 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 $1, \ldots, i-1$. (The usual type of cut-and-paste argument applies.)

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 $c[j] = c[i-1] + \mathit{lc}[i, j]$. As a base case, when we're computing $c[1]$, we need $c[0]$. If we set $c[0] = 0$, then $c[1] = \mathit{lc}[1, 1]$, 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

$$c[j] = \begin{cases} 0 & \text{if } j = 0 , \\ \min\{c[i-1] + lc[i,j] : 1 \le i \le j\} & \text{if } j > 0 . \end{cases}$$

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 cannot 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($l, n, M$)
  let $extras[1:n, 1:n]$, $lc[1:n, 1:n]$, $c[0:n]$, and $p[1:n]$ be new tables
  // Compute $extras[i, j]$ for $1 \le i \le j \le n$.
  for $i = 1$ to $n$
      $extras[i, i] = M - l_i$
      for $j = i + 1$ to $n$
          $extras[i, j] = extras[i, j-1] - l_j - 1$
  // Compute $lc[i, j]$ for $1 \le i \le j \le n$.
  for $i = 1$ to $n$
      for $j = i$ to $n$
          if $extras[i, j] < 0$
              $lc[i, j] = \infty$
          elseif $j == n$ and $extras[i, j] \ge 0$
              $lc[i, j] = 0$
          else $lc[i, j] = (extras[i, j])^3$
  // Compute $c[j]$ for $0 \le j \le n$ and $p[j]$ for $1 \le j \le n$.
  $c[0] = 0$
  for $j = 1$ to $n$
      $c[j] = \infty$
      for $i = 1$ to $j$
          if $c[i-1] + lc[i, j] < c[j]$
              $c[j] = c[i-1] + lc[i, j]$
              $p[j] = i$
  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 compute and store only *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 the output. The call PRINT-LINES(p, j) prints all words from word 1 through word j.

```
PRINT-LINES$(p, j)$
  if $j > 0$
      $i = p[j]$
      PRINT-LINES$(p, i - 1)$
      print the line containing words $i$ through $j$,
          with one space between each pair of words
```

The initial call is PRINT-LINES(p, n). Since the value of j decreases in each recursive call, PRINT-LINES takes a total of $O(n + k)$ time to print all n words, where k is the total length of all the words. (Note that because each word contains at least one character, even counting spaces and linefeeds as printed characters, the total number of characters printed is at most $2k$.)

Solution to Problem 14-5

a. This problem is a little like the longest-common-subsequence problem. In fact, we 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 are 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 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 $c[i, j] = c[i - 1, j - 1] + Q_C$.

- If it was a replace, then we must have had $x[i] \neq y[j]$. (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 $c[i, j] = c[i-1, j-1] + Q_R$.
- If it was a twiddle, then we must have had both $x[i] = y[j-1]$ and $x[i-1] = y[j]$, along with the implicit assumption that $i, j \geq 2$. Now our subproblem is $X_{i-2} \to Y_{j-2}$ and, assuming that the last operation was a twiddle, we have $c[i, j] = c[i-2, j-2] + Q_T$.
- 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 $c[i, j] = c[i-1, j] + Q_D$.
- 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 $c[i, j] = c[i, j-1] + Q_I$.
- 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 $0 \leq i < m$, as a subproblem. We pick the best one, and so assuming that the last operation was a kill, we have

  $$c[m, n] = \min\{c[i, n] : 0 \leq i < m\} + Q_K .$$

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 $c[0, j] = j \cdot Q_I$. Similarly, we convert X_i into Y_0 by a sequence of i deletes, so that $c[i, 0] = i \cdot Q_D$. 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:

$$c[i, j] = \min \begin{cases} c[i-1, j-1] + Q_C & \text{if } x[i] = y[j] , \\ c[i-1, j-1] + Q_R & \text{if } x[i] \neq y[j] , \\ c[i-2, j-2] + Q_T & \text{if } i, j \geq 2, x[i] = y[j-1], \\ & \text{and } x[i-1] = y[j] , \\ c[i-1, j] + Q_D & \text{always} , \\ c[i, j-1] + Q_I & \text{always} , \\ \min\{c[i, n] : 0 \leq i < m\} + Q_K & \text{if } i = m \text{ and } j = n . \end{cases}$$

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, the code fills in the table $op[i, j]$, holding which operation was used.

```
EDIT-DISTANCE$(x, y, m, n)$
  let $c[0:m, 0:n]$ and $op[0:m, 0:n]$ be new tables
  for $i = 0$ to $m$
      $c[i, 0] = i \cdot Q_D$
      $op[i, 0] =$ DELETE
  for $j = 1$ to $n$
      $c[0, j] = j \cdot Q_I$
      $op[0, j] =$ INSERT
  for $i = 1$ to $m$
      for $j = 1$ to $n$
          $c[i, j] = \infty$
          if $x[i] == y[j]$
              $c[i, j] = c[i-1, j-1] + Q_C$
              $op[i, j] =$ COPY
          else $c[i, j] = c[i-1, j-1] + Q_R$
              $op[i, j] =$ REPLACE (by $y[j]$)
          if $i \geq 2$ and $j \geq 2$ and $x[i] == y[j-1]$ and
              $x[i-1] == y[j]$ and
              $c[i-2, j-2] + Q_T < c[i, j]$
              $c[i, j] = c[i-2, j-2] + Q_T$
              $op[i, j] =$ TWIDDLE
          if $c[i-1, j] + Q_D < c[i, j]$
              $c[i, j] = c[i-1, j] + Q_D$
              $op[i, j] =$ DELETE
          if $c[i, j-1] + Q_I < c[i, j]$
              $c[i, j] = c[i, j-1] + Q_I$
              $op[i, j] =$ INSERT ($y[j]$)
  for $i = 0$ to $m - 1$
      if $c[i, n] + Q_K < c[m, n]$
          $c[m, n] = c[i, n] + Q_K$
          $op[m, n] =$ 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 occurs when $i = j = 0$. The first call is OP-SEQUENCE(op, m, n).

```
OP-SEQUENCE($op, i, j$)
  if $i == 0$ and $j == 0$
      return
  if $op[i, j]$ == COPY or $op[i, j]$ == REPLACE
      $i' = i - 1$
      $j' = j - 1$
  elseif $op[i, j]$ == TWIDDLE
      $i' = i - 2$
      $j' = j - 2$
  elseif $op[i, j]$ == DELETE
      $i' = i - 1$
      $j' = j$
  elseif $op[i, j]$ == INSERT          // don't care yet what character is inserted
      $i' = i$
      $j' = j - 1$
  else      // must be KILL, and must have $i = m$ and $j = n$
      let $op[i, j]$ == KILL $k$
      $i' = k$
      $j' = j$
  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

$$\begin{aligned} Q_C &= -1 , \\ Q_R &= +1 , \\ Q_D &= +2 , \\ Q_I &= +2 , \end{aligned}$$

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 apply the copy operation only when the characters are equal.

Solution to Problem 14-6

We want to maximize the conviviality of the party, while not inviting an employee and their immediate supervisor to the party. Starting with any employee x, we can choose whether or not to include x in the party. Think of x as a node in the tree representing the company hierarchy and the children of x as the direct reports of node x, that is, the employees for whom x is their immediate supervisor. If we choose to not include x, then we can include the direct reports of node x and maximize the sum of the conviviality of each subtree rooted at the direct reports, possibly including the direct reports. If we include x, then we cannot include the direct reports of x, but we can maximize the sum of the subtrees rooted at their direct reports. In order to solve this problem, we work in a bottom-up fashion to get to the president, P, of the hierarchy and maximize the conviviality of the entire party.

Therefore, we calculate two values for each employee x: the maximum conviviality of the subtree rooted at x if x is included and the maximum conviviality of the subtree rooted at x if x is not included. These values are calculated for every employee, and then the maximum overall conviviality can be determined by seeing whether the president of the corporation will be included.

First, let us define some notation. Each employee is represented by a node x, with the set of all employees x directly supervises denoted as $x.\mathit{reports}$. Denote the conviviality at a node x as $x.\mathit{conviviality}$. The maximum conviviality of the subtree rooted at node x with x included is the attribute $x.\mathit{included}$, and the maximum conviviality of the subtree rooted at x with x excluded is $x.\mathit{excluded}$.

Now, we can define the optimal substructure of the problem. At every node x, we calculate two values:

$$\begin{aligned} x.\mathit{included} &= x.\mathit{conviviality} + \sum_{r \in x.\mathit{reports}} r.\mathit{excluded} \,, \\ x.\mathit{excluded} &= \sum_{r \in x.\mathit{reports}} \max\{r.\mathit{excluded}, r.\mathit{included}\} \,. \end{aligned}$$

The base case occurs when a node x has no direct reports, so that $x.\mathit{reports} = \emptyset$. In this instance, $x.\mathit{included} = x.\mathit{conviviality}$ and $x.\mathit{excluded} = 0$.

The root of the tree is P, the president node. MAXIMIZE-CONVIVIALITY(r) returns the maximum total conviviality possible in the subtree rooted at x, recursively computing $x.\mathit{included}$ and $x.\mathit{excluded}$. The result of maximizing conviviality over the entire company hierarchy is the greater of the two values $P.\mathit{included}$ and $P.\mathit{excluded}$ returned by calling MAXIMIZE-CONVIVIALITY(P).

```
MAXIMIZE-CONVIVIALITY($x$)
  $x.included = x.conviviality$
  $x.excluded = 0$
  for each node $r$ in $x.reports$
      $(r.excluded, r.included) =$ MAXIMIZE-CONVIVIALITY($r$)
      $x.included = x.included + r.excluded$
      $x.excluded = x.excluded + \max\{r.excluded, r.included\}$
  return $(x.excluded, x.included)$
```

To calculate the time complexity, note that MAXIMIZE-CONVIVIALITY(P) calls MAXIMIZE-CONVIVIALITY once for each node in the tree, and aside from the recursive calls, it runs in constant time. This procedure visits each node and edge once, so that if the company has n employees, it runs in $\Theta(n)$ time. The space complexity is also $\Theta(n)$, since the left-child, right-sibling representation takes $\Theta(n)$ space and each node has three non-structural attributes: *conviviality*, *included*, and *excluded*.

In order to print the guest list, we need to recurse on the tree, using the values computed in MAXIMIZE-CONVIVIALITY to decide whether to invite each employee, which we indicate with the boolean parameter *invited*. Assume that each node x has an attribute $x.\mathit{name}$ giving the employee's name.

```
OUTPUT-GUEST-LIST(P)
  (P.excluded, P.included) = MAXIMIZE-CONVIVIALITY(P)
  if P.excluded > P.included
      PRINT-GUESTS(P, TRUE)
  else PRINT-GUESTS(P, FALSE)

PRINT-GUESTS(r, invited)
  if invited
      print x.name "is invited"
      for each node r in x.reports
          PRINT-GUESTS(a, FALSE)
  else for each node r in x.reports
          if r.included > r.excluded
              PRINT-GUESTS(r, TRUE)
          else PRINT-GUESTS(r, FALSE)
```

The call OUTPUT-GUEST-LIST(P) has PRINT-GUESTS visit every node. Since the company structure is a tree, the time complexity of this procedure is $\Theta(n)$.

Solution to Problem 14-8

a. Let us set up a recurrence for the number of valid seams as a function of m. Suppose we are in the process of carving out a seam row by row, starting from the first row. Let the last pixel carved out be $A[i, j]$. How many choices do we have for the pixel in row $i + 1$ such that the pixel continues the seam? If the last pixel $A[i, j]$ is in the first or last column ($j = 1$ or $j = n$), then there are two choices for the next pixel. When $j = 1$, the two choices for the next pixel are $A[i + 1, j]$ and $A[i + 1, j + 1]$. When $j = n$, the two choices for the next pixel are $A[i + 1, j - 1]$ and $A[i + 1, j]$. Otherwise—when the last pixel is not in the first or last column—there are three choices for the next pixel: $A[i+1, j-1]$, $A[i+1, j]$, and $A[i+1, j+1]$. Thus, for a general pixel $A[i, j]$, there are at least two possible choices for a pixel p in the next row such that p continues a seam ending in $A[i, j]$. Let $T(i)$ denote the number of possible seams from row 1 to row i. Then, we have $T(1) = n$ (since the seam can start at any column in row 1) and $T(i) \geq 2T(i - 1)$ for $i > 1$.

We guess that $T(i) \geq n2^{i-1}$, which we verify by direct substitution. We have $T(1) = n \geq n \cdot 2^0$, and for $i > 1$, we have

$$\begin{aligned} T(i) &\geq 2T(i-1) \\ &\geq 2 \cdot n2^{i-2} \\ &= n2^{i-1} . \end{aligned}$$

Thus, the total number $T(m)$ of seams is at least $n2^{m-1}$. We conclude that the number of seams grows at least exponentially in m.

b. As proved in the previous part, it is infeasible to systematically check every seam, since the number of possible seams grows exponentially.

The structure of the problem allows us to build the solution row by row. Consider a pixel $A[i, j]$. We ask the question: "If i were the first row of the picture, what is the minimum disruptive measure of seams that start with the pixel $A[i, j]$?"

Let S^* be a seam of minimum disruptive measure among all seams that start with pixel $A[i, j]$. Let $A[i+1, p]$, where $p \in \{j-1, j, j+1\}$, be the pixel of S^* in the next row. Let S' be the sub-seam of S^* that starts with $A[i+1, p]$. We claim that S' has the minimum disruptive measure among seams that start with $A[i+1, p]$. Why? Suppose there exists another seam S'' that starts with $A[i+1, p]$ and has disruptive measure less than that of S'. By using S'' as the sub-seam instead of S', we can obtain another seam that starts with $A[i, j]$ and has a disruptive measure which is less than that of S^*. Thus, we obtain a contradiction to our assumption that S^* is a seam of minimum disruptive measure.

Let $\mathit{disr}[i, j]$ be the value of the minimum disruptive measure among all seams that start with pixel $A[i, j]$. For row m, the seam with the minimum disruptive measure consists of just one point. We can now state a recurrence for $\mathit{disr}[i, j]$ as follows. In the base case, $\mathit{disr}[m, j] = d[m, j]$ for $j = 1, 2, \ldots, n$. In the recursive case, for $j = 1, 2, \ldots, n$,

$$\mathit{disr}[i, j] = d[i, j] + \min\{\mathit{disr}[i+1, j+k] : k \in K\} ,$$

where the set K of index offsets is

$$K = \begin{cases} \{0, 1\} & \text{if } j = 1 , \\ \{-1, 0, 1\} & \text{if } 1 < j < n , \\ \{-1, 0\} & \text{if } j = n . \end{cases}$$

Since every seam has to start with a pixel of the first row, we simply find the minimum $\mathit{disr}[1, j]$ for pixels in the first row to obtain the minimum disruptive measure.

```
COMPRESS-IMAGE$(d, m, n)$
  let $disr[1:m, 1:n]$ and $next[1:m, 1:n]$ be new tables
  **for** $j = 1$ **to** $n$
      $disr[m, j] = d[m, j]$
  **for** $i = m - 1$ **downto** 1
      **for** $j = 1$ **to** $n$
          **if** $j == 1$
              $low = 0$
          **else** $low = -1$
          **if** $j == n$
              $high = 0$
          **else** $high = 1$
          $min\text{-}neighbor\text{-}disruption = \infty$
          **for** $k = low$ **to** $high$
              **if** $disr[i + 1, j + k] < min\text{-}neighbor\text{-}disruption$
                  $min\text{-}neighbor\text{-}disruption = disr[i + 1, j + k]$
                  $next[i, j] = j + k$
          $disr[i, j] = min\text{-}neighbor\text{-}disruption + d[i, j]$
  $min\text{-}overall\text{-}disruption = \infty$
  $column = 1$
  **for** $j = 1$ **to** $n$
      **if** $disr[1, j] < min\text{-}overall\text{-}disruption$
          $min\text{-}overall\text{-}disruption = disr[1, j]$
          $column = j$
  print "The minimum value of the disruptive measure is "
          $min\text{-}overall\text{-}disruption$
  **for** $i = 1$ **to** $m$
      print "cut point at " $(i, column)$
      $column = next[i, column]$
```

The procedure COMPRESS-IMAGE is simply an implementation of this recurrence in a bottom-up fashion.

It first initializes the base cases, which are the cases when row $i = m$. The minimum disruptive measure for the base cases is simply $d[m, j]$ fpr column $j = 1, 2, \ldots, n$.

The next **for** loop runs down from $m - 1$ to 1. Thus, $disr[i + 1, j]$ is already available before computing $disr[i, j]$ for the pixels of row i.

The assignments to *low* and *high* allow the index offset k to range over the correct set K from above. The code sets *low* to 0 when $j = 1$ and to -1 when $j > 1$, and it sets *high* to 0 when $j = n$ and to 1 when $j < n$. The innermost **for** loop finds the minimum value of $disr[i + 1, j + k]$ for all $k \in K$. Then the code sets $disr[i, j]$ to this minimum value plus the disruption $d[i, j]$.

The *next* table is for reconstructing the actual seam. For a given pixel, it records which pixel was used as the next pixel. Specifically, for a pixel $A[i, j]$, if $next[i, j] = p$, where $p \in \{j - 1, j, j + 1\}$, then the next pixel of the seam is $A[i + 1, p]$.

The next **for** loop finds the minimum overall disruptive measure, which is over pixels in the first row. The procedure prints the minimum overall disruptive measure as the answer.

The rest of the code reconstructs the actual seam, using the information stored in the *next* array.

Noting that the innermost **for** loop runs over at most three values of k, we see that the running time of COMPRESS-IMAGE is $O(mn)$. The space requirement is also $O(mn)$.

Solution to Problem 14-9

Our first step will be to identify the subproblems that satisfy the optimal-substructure property. Before we frame the subproblem, we make two simplifying modifications to the input:

- We sort L so that the indices in L are in ascending order.
- We prepend the index 0 to the beginning of L and append n to the end of L.

Let $L[i:j]$ denote a subarray of L that starts from index i and ends at index j. Denote by $S[1:n]$ the n-character string to be broken. Define the subproblem denoted by (i, j) as "What is the cheapest sequence of breaks to break the substring $S[L[i]+1:L[j]]$?" Note that the first and last elements of the subarray $L[i:j]$ define the ends of the substring, and we have to worry about only the indices of the subarray $L[i+1:j-1]$.

For example, let $L = \langle 20, 17, 14, 11, 25 \rangle$ and $n = 30$. First, sort L. Then, prepend 0 and append n as explained to get $L = \langle 0, 11, 14, 17, 20, 25, 30 \rangle$. Now, what is the subproblem $(2, 6)$? We obtain a substring by breaking S after character $L[2] = 11$ and character $L[6] = 25$. We ask "What is the cheapest sequence of breaks to break the substring $S[12:25]$?" We have to worry about only indices in the subarray $L[3:5] = \langle 14, 17, 20 \rangle$, since the other indices are not present in the substring.

At this point, the problem looks similar to matrix-chain multiplication (see Section 14.2). We can make the first break at any element of $L[i+1:j-1]$.

Suppose that an optimal sequence of breaks σ for subproblem (i, j) makes the first break at $L[k]$, where $i < k < j$. This break gives rise to two subproblems:

- The "prefix" subproblem (i, k), covering the subarray $L[i+1:k-1]$,
- The "suffix" subproblem (k, j), covering the subarray $L[k+1:j-1]$.

The overall cost can be expressed as the sum of the length of the substring, the prefix cost, and the suffix cost.

We show optimal substructure by claiming that the sequence of breaks in σ for the prefix subproblem (i, k) must be an optimal one. Why? If there were a less costly way to break the substring $S[L[i]+1:L[k]]$ represented by the subproblem (i, k), then substituting that sequence of breaks in σ would produce another sequence of breaks whose cost is lower than that of σ, which would be a contradiction. A similar observation holds for the sequence of breaks for the suffix subproblem (k, j): it must be an optimal sequence of breaks.

Let $cost[i, j]$ denote the cost of the cheapest solution to subproblem (i, j), where $1 \le i \le j \le m$. Since subproblems of the form (i, i) and $(i, i+1)$ have no possible locations at which to break within them, $cost[i, j] = 0$ for $j \le i + 1$. We write the recurrence relation for $cost$ as

$$cost[i, j] = \begin{cases} 0 & \text{if } j \le i + 1 , \\ \min\{cost[i, k] + cost[k, j] + (L[j] - L[i]) : i < k < j\} & \\ & \text{if } j > i + 1 . \end{cases}$$

Thus, our approach to solving the subproblem (i, j) tries splitting the respective substring at all values of k strictly between i and j and then choosing a break that results in the minimum cost. We need to be careful to solve smaller subproblems before solving larger subproblems. In particular, we solve subproblems in increasing order of the length $j - i$.

```
BREAK-STRING(L, m, n)
  prepend 0 to the start of L and append n to the end of L
  sort L into increasing order
  let cost[1 : m, 1 : m] and break[1 : m, 1 : m] be new tables
  for i = 1 to m - 1
      cost[i, i] = 0
      cost[i, i + 1] = 0
  cost[m, m] = 0
  for length = 3 to m
      for i = 1 to m - length + 1
          j = i + length - 1
          min-cost = ∞
          for k = i + 1 to j - 1
              if cost[i, k] + cost[k, j] < min-cost
                  min-cost = cost[i, k] + cost[k, j]
                  break[i, j] = k
          cost[i, j] = min-cost + L[j] - L[i]
  print "The minimum cost of breaking the string is " cost[1, m]
  PRINT-BREAKS(L, break, 1, m)
```

After sorting L, the code initializes the base cases, in which $i = j$ or $j = i + 1$.

The nested **for** loops represent the main computation. The outermost **for** loop runs for $length = 3$ to m, so that it considers subarrays of L with length at least 3, since the first and the last element define the substring, and we need at least one more element to specify a break. The increasing values of *length* also ensure that subproblems with smaller length are solved solving subproblems with greater length.

The **for** loop on i runs from 1 to $m - length + 1$. The upper bound of $m - length + 1$ is the largest value that the index i can take such that $i + length - 1 \le m$.

The innermost **for** loop tries each possible location k as the place to make the first break for subproblem (i, j). The first such place is $L[i + 1]$, and not $L[i]$, since

$L[i]$ represents the start of the substring (and thus not a valid place for a break). Similarly, the last valid place is $L[j-1]$, because $L[j]$ represents the end of the substring. The **if** condition tests whether k is the best place for a break found so far, and it updates the best value in *min-cost* if so. We use $\mathit{break}[i, j]$ to record that the best place for the first break is k. Specifically, if $\mathit{break}[i, j] = k$, then an optimal sequence of breaks for (i, j) makes the first break at $L[k]$. Having found the minimum-cost break, the code sets $\mathit{cost}[i, j]$ to the cost of this break plus the length $L[j]-L[i]$ of the substring, since regardless where the first break is, it costs a price equal to the length of the substring to make a break.

The lowest cost for the original problem ends up in $\mathit{cost}[1, m]$. By our initialization, $L[1] = 0$ and $L[m] = n$. Thus, $\mathit{cost}[1, m]$ will hold the optimum price of cutting the substring from $L[1] + 1 = 1$ to $L[m] = n$, which is the entire string.

The running time is $\Theta(m^3)$, and it is dictated by the three nested **for** loops. They fill in the entries above the main diagonal of the two tables, except for entries in which $j = i + 1$. That is, they fill in rows $i = 1, 2, \ldots, m-2$ and columns $j = i+2, i+3, \ldots, m$. To fill in entry $[i, j]$, the code checks values of k running from $i+1$ to $j-1$, or $j-i-1$ entries. Thus, the total number of iterations of the innermost **for** loop is

$$\begin{aligned}
\sum_{i=1}^{m-2} \sum_{j=i+2}^{m} (j-i-1) &= \sum_{i=1}^{m-2} \sum_{d=1}^{m-i-1} d && \text{(reindexing: } d = j-i-1\text{)} \\
&= \sum_{i=1}^{m-2} \Theta((m-i)^2) && \text{(equation (A.2))} \\
&= \sum_{h=2}^{m-1} \Theta(h^2) && \text{(reindexing: } h = m-i\text{)} \\
&= \Theta(m^3) && \text{(equation (A.4))} \ .
\end{aligned}$$

Since each iteration of the innermost **for** loop takes constant time, the total running time is $\Theta(m^3)$. The running time does not depend on the length n of the string.

```
PRINT-BREAKS($L, break, i, j$)
  if $j > i + 1$
      $k = break[i, j]$
      print "break at " $L[k]$
      PRINT-BREAKS($L, break, i, k$)
      PRINT-BREAKS($L, break, k, j$)
```

PRINT-BREAKS uses the information stored in *break* to print out the actual sequence of breaks.

Solution to Problem 14-11

We state the subproblem (k, s) as "What is the cheapest way to satisfy all the demands of months $k, \ldots, n$ when starting with a surplus of s before the kth month?"

A ***plan*** for the subproblem (k, s) would specify the number of machines to manufacture for each month $k, \ldots, n$ such that demands are satisfied.

Consider some optimal plan P for subproblem (k, s), let P' be the part of P for months $k+1, \ldots, n$, and let s' be the surplus after the kth month (i.e., at the start of month $k+1$). We claim that P' is an optimal plan for the subproblem $(k+1, s')$. Why? Suppose P' were not an optimal plan and let P'' be an optimal plan for $(k+1, s')$. If we modify plan P by cutting out P' and pasting in P'' (i.e., by using plan P'' for months $k+1, \ldots, n$), we obtain another plan for (k, s) with lower cost than plan P. Thus, we obtain a contradiction to the assumption that plan P was optimal.

The lower and upper bounds for how many machines to manufacture in month k are as follows:

- Lower bound: at least the number of machines so that, along with surplus s, there are enough machines to satisfy the demand for month k. Denoting this lower bound by $L(k, s)$, we have

  $$L(k, s) = \max\{d_k - s, 0\} .$$

- Upper bound: at most the number of machines such that there are enough machines to satisfy the demands of all the following months. Denoting this upper bound by $U(k, s)$, we have

  $$U(k, s) = \left(\sum_{i=k}^{n} d_i\right) - s .$$

For the last month, the company needs to manufacture only the minimum required number of machines, given by $L(n, s) = \max\{d_n - s, 0\}$. We also know that entering month 1, the surplus is 0.

Now, let's see how to compute the costs incurred in month k when it starts with a surplus of s machines. Suppose the company manufactures q machines in month k. The cost of manufacturing q machines is $c \cdot \max\{q - m, 0\}$. We also have to consider the holding cost. The company starts with a surplus of s machines, manufactures q more, and sells d_k machines, so that month $k+1$ starts with a surplus of $s + q - d_k$ machines, thereby giving a holding cost of $h(s + q - d_k)$.

We are now ready to write a recurrence for *cost*$[k, s]$, the cost of an optimal plan for the subproblem (k, s). We already know that the company should manufacture $L(n, s) = \max\{d_n - s, 0\}$ machines in month n. For any other month k, we try all possible values of q, ranging from $L(k, s)$ to $U(k, s)$, and incorporating the cost of the remaining subproblem, giving the recurrence

$$\mathit{cost}[k, s] = \begin{cases} c \cdot \max\{L(n, s) - m, 0\} + h(s + L(n, s) - d_n) & \text{if } k = n ,\\ \min\{\mathit{cost}[k+1, s + q - d_k] + c \cdot \max\{q - m, 0\} & \\ \qquad + h(s + q - d_k) : L(k, s) \le q \le U(k, s)\} & \text{if } 1 \le k < n . \end{cases}$$

The recurrence suggests how to build an optimal plan in a bottom-up fashion. We now present a procedure for constructing an optimal plan.

```
INVENTORY-PLANNING$(d, n, m, c, h)$
  compute $D = \sum_{i=1}^{n} d_i$
  let $cost[1:n, 0:D]$ and $make[1:n, 0:D]$ be new tables
  // Compute $cost[n, 0:D]$ and $make[n, 0:D]$.
  for $s = 0$ to $D$
      $q = \max\{d_n - s, 0\}$
      $cost[n, s] = c \cdot \max\{q - m, 0)\} + h(s + q - d_n)$
      $make[n, s] = q$
  // Compute $cost[1:n-1, 0:D]$ and $make[1:n-1, 0:D]$.
  $U = d_n$
  for $k = n - 1$ downto 1
      $U = U + d_k$
      for $s = 0$ to $D$
          $cost[k, s] = \infty$
          for $q = \max\{d_k - s, 0\}$ to $U - s$
              $val = cost[k+1, s + q - d_k] + c \cdot \max\{q - m, 0\}$
                      $+ h(s + q - d_k)$
              if $val < cost[k, s]$
                  $cost[k, s] = val$
                  $make[k, s] = q$
  print $cost[1, 0]$
  PRINT-PLAN$(make, d, n)$
```

```
PRINT-PLAN$(make, d, n)$
  $s = 0$
  for $k = 1$ to $n$
      print "For month " $k$ " manufacture " $make[k, s]$ " machines"
      $s = s + make[k, s] - d_k$
```

The INVENTORY-PLANNING procedure builds the solution month by month, starting from month n and moving backward toward month 1. First, it solves the subproblems for the last month and for all surpluses. Then, for each month and for each surplus entering that month, it calculates the lowest-cost way to satisfy demand for that month based on the solved subproblems of the next month.

- q is a possible number of machines that to manufacture in month k.
- $cost[k, s]$ holds the lowest-cost way to satisfy demands of months $k, \ldots, n$, starting with a surplus of s machines at the beginning of month k.
- $make[k, s]$ holds the number of machines to manufacture in month k and the surplus s of an optimal plan. This table helps in reconstructing an optimal plan in the procedure PRINT-PLAN.

After summing the monthly demands to determine the total demand D and allocating the *cost* and *make* tables, the procedure initializes the base cases, which are the cases for month n starting with surplus s, for $s = 0, \ldots, D$. If $d_n > s$, it suffices to manufacture $d_n - s$ machines, since no surplus is needed after month n. If $d_n \le s$, no machines need to be manufactured in month n.

The procedure then calculates the total cost for month n as the sum of the cost $c \cdot \max\{q - m, 0\}$ to hire extra labor and the inventory cost $h(s + q - d_n)$ for leftover surplus, which can be nonzero if the month started with a large surplus. As we saw, for month n, the production q is just $\max\{d_n - s, 0\}$.

The outer **for** loop of the next block of code runs down from month $n-1$ to 1, thus ensuring that upon considering month k, it has already solved the subproblems for month $k+1$.

The next inner **for** loop iterates through all possible values of q. For every choice of q for a given month k, the total cost of subproblem (k, s) is given by the cost of extra labor (if any) plus the cost of inventory (if there is a surplus) plus the cost of the subproblem $(k+1, s+q-d_k)$. This value is checked and updated.

Finally, the required answer is the answer to the subproblem $(1, 0)$, which appears in $cost[1, 0]$. That is, it is the cheapest way to satisfy all the demands of months $1, \ldots, n$ when starting with a surplus of 0.

The procedure PRINT-PLAN uses the *make* table to print an optimal number of machines to manufacture in each month. It keeps a running notion of the surplus s so that it can consult the appropriate entry $make[k, s]$ for each month k.

The running time of INVENTORY-PLANNING is clearly $O(nD^2)$. (The innermost **for** loop runs at most $D+1$ times, since $U \leq D$.) The space requirement is $O(nD)$.

Solution to Problem 14-12

Since the order of choosing players for the positions does not matter, we may assume that we make our decisions starting from position 1, moving toward position N. For each position, we decide to either sign one player or sign no players. Suppose we decide to sign player p, who plays position 1. Then, we are left with an amount of $X - p.cost$ dollars to sign players at positions $2, \ldots, N$. This observation guides us in how to frame the subproblems.

We define the cost and WAR of a *set* of players as the sum of costs and the sum of WARs of all players in that set. Let (i, x) denote the following subproblem: "Suppose we consider only positions $i, i+1, \ldots, N$ and we can spend at most x dollars. What set of players—with at most one player for each position under consideration—yields the maximum WAR?" A ***valid*** set of players for (i, x) is one in which each player in the set plays one of the positions $i, i+1, \ldots, N$, each position has at most one player, and the cost of the players in the set is at most x dollars. An ***optimal*** set of players for (i, x) is a valid set with the maximum WAR. We now show that the problem exhibits optimal substructure.

Theorem (Optimal substructure of the WAR maximization problem)
Let $L = \{p_1, p_2, \ldots, p_k\}$ be a set of players, possibly empty, with maximum WAR for the subproblem (i, x).

1. If $i = N$, then L has at most one player. If all players in position N have cost more than x, then L has no players. Otherwise, $L = \{p_1\}$, where p_1 has the maximum WAR among players for position N with cost at most x.

2. If $i < N$ and L includes player p for position i, then $L' = L - \{p\}$ is an optimal set for the subproblem $(i + 1, x - p.cost)$.
3. If $i < N$ and L does not include a player for position i, then L is an optimal set for the subproblem $(i + 1, x)$.

Proof Property (1) follows trivially from the problem statement.

(2) Suppose that L' is not an optimal set for the subproblem $(i + 1, x - p.cost)$. Then, there exists another valid set L'' for $(i + 1, x - p.cost)$ that has WAR more than L'. Let $L''' = L'' \cup \{p\}$. The cost of L''' is at most x, since L'' has a cost at most $x - p.cost$. Moreover, L''' has at most one player for each position $i, i + 1, \ldots, N$. Thus, L''' is a valid set for (i, x). But L''' has WAR more than L, thus contradicting the assumption that L had the maximum WAR for (i, x).

(3) Clearly, any valid set for $(i + 1, x)$ is also a valid set for (i, x). If L were not an optimal set for $(i + 1, x)$, then there exists another valid set L' for $(i + 1, x)$ with WAR more than L. The set L' would also be a valid set for (i, x), which contradicts the assumption that L had the maximum WAR for (i, x). ■

The theorem suggests that when $i < N$, we examine two subproblems and choose the better of the two. Let $w[i, x]$ denote the maximum WAR for (i, x). Let $S(i, x)$ be the set of players who play position i and cost at most x. In the following recurrence for $w[i, x]$, we assume that the max function returns $-\infty$ when invoked over an empty set:

$$w[i, x] = \begin{cases} \max\{p.war : p \in S(N, x)\} & \text{if } i = N , \\ \max\big\{w[i + 1, x], & \\ \quad \max\{p.war + w[i + 1, x - p.cost] : p \in S(i, x)\}\big\} & \text{if } i < N . \end{cases}$$

This recurrence lends itself to implementation in a straightforward way. Let p_{ij} denote the jth player who plays position i.

```
FREE-AGENT-WAR$(p, N, P, X)$
  let $w[1:N][0:X]$ and $who[1:N][0:X]$ be new tables
  for $x = 0$ to $X$
      $w[N, x] = -\infty$
      $who[N, x] = 0$
      for $k = 1$ to $P$
          if $p_{Nk}.cost \leq x$ and $p_{Nk}.war > w[N, x]$
              $w[N, x] = p_{Nk}.war$
              $who[N, x] = k$
  for $i = N - 1$ downto 1
      for $x = 0$ to $X$
          $w[i, x] = w[i + 1, x]$
          $who[i, x] = $ NIL
          for $k = 1$ to $P$
              if $p_{ik}.cost \leq x$ and $w[i + 1, x - p_{ik}.cost] + p_{ik}.war > w[i, x]$
                  $w[i, x] = w[i + 1, x - p_{ik}.cost] + p_{ik}.war$
                  $who[i, x] = k$
  print "The maximum value of WAR is " $w[1, X]$
  $spent = 0$
  for $i = 1$ to $N$
      $k = who[i, X - spent]$
      if $k \neq$ NIL
          print "sign player " $p_{ik}$
          $spent = spent + p_{ik}.cost$
  print "The total money spent is " $spent$
```

The input to FREE-AGENT-WAR is the list of players p and N, P, and X, as given in the problem. The table $w[i, x]$ holds the maximum WAR for the subproblem (i, x). The table $who[i, x]$ holds information necessary to reconstruct the actual solution. Specifically, $who[i, x]$ holds the index of the player to sign for position i, or NIL if no player should be signed for position i. The first set of nested **for** loops initializes the base cases, in which $i = N$. For every amount x, the inner loop simply picks the player with the highest WAR who plays position N and whose cost is at most x.

The next group of three nested **for** loops represents the main computation. The outermost **for** loop runs down from position $N-1$ to 1. This order ensures that smaller subproblems are solved before larger ones. Initializing $w[i, x]$ as $w[i + 1, x]$ takes care of the case in which we decide not to sign any player who plays position i. The innermost **for** loop tries to sign each player (if enough money remains) in turn, and it keeps track of the maximum WAR possible.

The maximum WAR for the entire problem ends up in $w[1, X]$. The final **for** loop uses the information in *who* table to print out which players to sign. The running time of FREE-AGENT-WAR is clearly $\Theta(NPX)$, and it uses $\Theta(NX)$ space.

Lecture Notes for Chapter 15: Greedy Algorithms

[The fourth edition removed the starred sections on matroids and task scheduling (an application of matroids). These sections were replaced by a new, unstarred section covering offline caching, which had been the subject of Problem 16-5 in the third edition.]

Chapter 15 overview

Similar to dynamic programming.

Used for optimization problems.

Idea

When you 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 then study two other applications of the greedy method: Huffman coding and offline caching. *[Later chapters use the greedy method as well: minimum spanning tree, Dijkstra's algorithm for single-source shortest paths, and a greedy set-covering heuristic.]*

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.

Could have many other objectives:

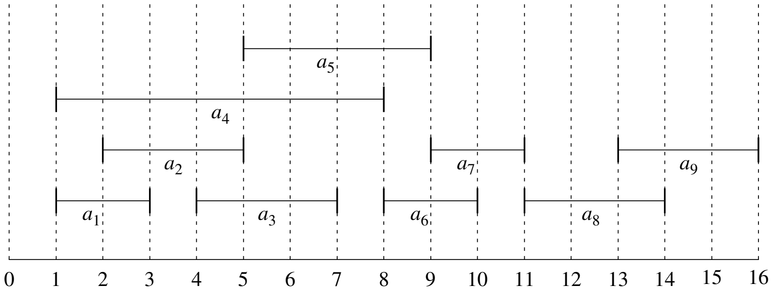
- Schedule room for longest time.
- Maximize income rental fees.

Assume that activities are sorted by finish time: $f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n$.

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

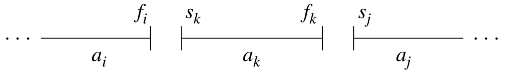


Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_1, a_3, a_6, a_9\}$, $\{a_1, a_3, a_7, a_8\}$, $\{a_1, a_3, a_7, a_9\}$, $\{a_1, a_5, a_7, a_8\}$, $\{a_1, a_5, a_7, a_9\}$, $\{a_2, a_5, a_7, a_8\}$, $\{a_2, a_5, a_7, a_9\}$.

Optimal substructure of activity selection

$$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\} \qquad \textit{[Leave on board]}$$
$$= \text{activities that start after } a_i \text{ finishes and finish before } a_j \text{ starts .}$$



Activities in S_{ij} are compatible with

- all activities that finish by f_i, and
- all activities that start no earlier than s_j.

Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij}.

Let $a_k \in A_{ij}$ be some activity in A_{ij}. Then we have two subproblems:

- Find mutually compatible activities in S_{ik} (activities that start after a_i finishes and that finish before a_k starts).
- Find mutually compatible activities in S_{kj} (activities that start after a_k finishes and that finish before a_j starts).

Let

$$A_{ik} = A_{ij} \cap S_{ik} = \text{activities in } A_{ij} \text{ that finish before } a_k \text{ starts ,}$$
$$A_{kj} = A_{ij} \cap S_{kj} = \text{activities in } A_{ij} \text{ that start afer } a_k \text{ finishes .}$$

Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

$\Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

Claim

Optimal solution A_{ij} must include optimal solutions for the two subproblems for S_{ik} and S_{kj}.

Proof of claim Use the usual cut-and-paste argument. Will show the claim for S_{kj}; proof for S_{ik} is symmetric.

Suppose we could find a set A'_{kj} of mutually compatible activities in S_{kj}, where $|A'_{kj}| > |A_{kj}|$. Then use A'_{kj} instead of A_{kj} when solving the subproblem for S_{ij}. Size of resulting set of mutually compatible activities would be $|A_{ik}|+|A'_{kj}|+1 > |A_{ik}| + |A_{kj}| + 1 = |A|$. Contradicts assumption that A_{ij} is optimal. ■ (claim)

One recursive solution

Since optimal solution A_{ij} must include optimal solutions to the subproblems for S_{ik} and S_{kj}, could solve by dynamic programming.

Let $c[i, j]$ = size of optimal solution for S_{ij}. Then

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

But we don't know which activity a_k to choose, so we have to try them all:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset . \end{cases}$$

Could then develop a recursive algorithm and memoize it. Or could develop a bottom-up algorithm and fill in table entries.

Instead, we will look at a greedy approach.

Making the greedy choice

Choose an activity to add to optimal solution *before* solving subproblems. For activity-selection problem, we can get away with considering only the greedy choice: the activity that leaves the resource available for as many other activities as possible.

Question: Which activity leaves the resource available for the most other activities? Answer: The first activity to finish. (If more than one activity has earliest finish time, can choose any such activity.)

Since activities are sorted by finish time, just choose activity a_1.

That leaves only one subproblem to solve: finding a maximum size set of mutually compatible activities that start after a_1 finishes. (Don't have to worry about activities that finish before a_1 starts, because $s_1 < f_1$ and no activity a_i has finish time $f_i < f_1 \Rightarrow$ no activity a_i has $f_i \leq s_1$.)

Since have only subproblem to solve, simplify notation:

$$S_k = \{a_i \in S : s_i \geq f_k\} = \text{activities that start after } a_k \text{ finishes} .$$

Making greedy choice of $a_1 \Rightarrow S_1$ remains as only subproblem to solve. *[Slight abuse of notation: referring to S_k not only as a set of activities but as a subproblem consisting of these activities.]*

By optimal substructure, if a_1 is in an optimal solution, then an optimal solution to the original problem consists of a_1 plus all activities in an optimal solution to S_1.

But need to prove that a_1 is always part of some optimal solution.

Theorem

If S_k is nonempty and a_m has the earliest finish time in S_k, then a_m is included in some optimal solution.

Proof Let A_k be an optimal solution to S_k, and let a_j have the earliest finish time of any activity in A_k. If $a_j = a_m$, done. Otherwise, let $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but with a_m substituted for a_j.

Claim

Activities in A'_k are disjoint.

Proof of claim Activities in A_k are disjoint, a_j is first activity in A_k to finish, and $f_m \le f_j$. ■ (claim)

Since $|A'_k| = |A_k|$, conclude that A'_k is an optimal solution to S_k, and it includes a_m. ■ (theorem)

So, don't need full power of dynamic programming. Don't need to work bottom-up.

Instead, can just repeatedly choose the activity that finishes first, keep only the activities that are compatible with that one, and repeat until no activities remain.

Can work top-down: make a choice, then solve a subproblem. Don't have to solve subproblems before making a choice.

Recursive greedy algorithm

Start and finish times are represented by arrays s and f, where f is assumed to be already sorted in monotonically increasing order.

To start, add fictitious activity a_0 with $f_0 = 0$, so that $S_0 = S$, the entire set of activities.

Procedure RECURSIVE-ACTIVITY-SELECTOR takes as parameters the arrays s and f, index k of current subproblem, and number n of activities in the original problem.

```
RECURSIVE-ACTIVITY-SELECTOR$(s, f, k, n)$
  $m = k + 1$
  while $m \le n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
      $m = m + 1$
  if $m \le n$
      return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
  else return $\emptyset$
```

Initial call

RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Idea

The **while** loop checks $a_{k+1}, a_{k+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_k (need $s_m \geq f_k$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve S_m, 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 $\{a_1, a_3, a_6, a_8\}$.

Time

$\Theta(n)$—each activity examined exactly once, assuming that activities are already sorted by finish times.

Iterative greedy algorithm

Can convert the recursive algorithm to an iterative one. It's already almost tail recursive.

```
GREEDY-ACTIVITY-SELECTOR$(s, f, n)$
  $A = \{a_1\}$
  $k = 1$
  for $m = 2$ to $n$
      if $s[m] \geq f[k]$          // is $a_m$ in $S_k$?
          $A = A \cup \{a_m\}$     // yes, so choose it
          $k = m$                  // and continue from there
  return $A$
```

Go through example given earlier. Should again get $\{a_1, a_3, a_6, a_8\}$.

Time

$\Theta(n)$, if activities are already sorted by finish times.

For both the recursive and iterative algorithms, add $O(n \lg n)$ time if activities need to be sorted.

Elements of the 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. Show that if you make the greedy choice, only one subproblem remains.
4. Prove that it's always safe to make the greedy choice.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.

At first, it looked like dynamic programming. In the activity-selection problem, we started out by defining subproblems S_{ij}, where both i and j varied. But then found that making the greedy choice allowed us to restrict the subproblems to be of the form S_k.

Could instead have gone straight for the greedy approach: in our first crack at defining subproblems, use the S_k form. Could then have proven that the greedy choice a_m (the first activity to finish), combined with optimal solution to the remaining compatible activities S_m, gives an optimal solution to S_k.

Typically, we streamline these 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. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

No general way to tell whether a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.

Greedy-choice property

Can assemble a globally optimal solution by making locally optimal (greedy) choices.

Dynamic programming

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Solve *bottom-up* (unless memoizing).

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 an optimal solution.
- If it includes the greedy choice, done.
- Otherwise, modify the optimal solution 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 ⇒ 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. Take items in decreasing order of value/weight. Will take all of the items with the greatest value/weight, and possibly a fraction of the next item.

```
FRACTIONAL-KNAPSACK($v, w, W$)
  $load = 0$
  $i = 1$
  while $load < W$ and $i \leq n$
      if $w_i \leq W - load$
          take all of item $i$
      else take $(W - load)/w_i$ of item $i$
      add what was taken to $load$
      $i = i + 1$
```

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.

Huffman codes

Goal: Compress a data file made up of characters. You know how often each character appears in the file—its ***frequency***. Each character is represented by some bit sequence: a ***codeword***. Use as few bits as possible to represent the file.

Fixed-length code: All codewords have the same number of bits. For $n \geq 2$ characters, need $\lceil \lg n \rceil$ bits.

Variable-length code: Represent different characters with differing numbers of bits. In particular, give frequently occurring characters shorter codewords and infrequently occurring characters longer codewords.

Example: For a data file of 100,000 characters:

	`a`	`b`	`c`	`d`	`e`	`f`
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

For a fixed-length code, need 3 bits per character. For 100,000 characters, need 300,000 bits. For this variable-length code, need

$$\begin{array}{rcl}
45{,}000 \cdot 1 & = & 45{,}000 \\
+\ 13{,}000 \cdot 3 & = & 39{,}000 \\
+\ 12{,}000 \cdot 3 & = & 36{,}000 \\
+\ 16{,}000 \cdot 3 & = & 48{,}000 \\
+\ \ 9{,}000 \cdot 4 & = & 36{,}000 \\
+\ \ 5{,}000 \cdot 4 & = & 20{,}000 \\
\hline
 & = & 224{,}000 \text{ bits}
\end{array}$$

Prefix-free codes

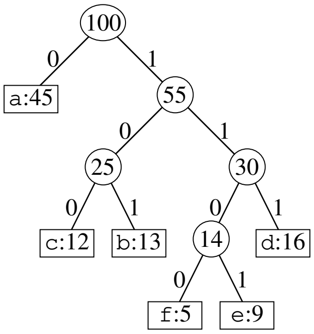
No codeword is also a prefix of any other codeword. *[Called "prefix codes" in earlier editions of the book. Changed to "prefix-free codes" in the fourth edition because each codeword is free of prefixes of other codes.]* A prefix-free code can always achieve the optimal compression.

Encoding: Just concatenate codewords for each character in the file. ***Example:*** To encode `face`: $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where $\cdot$ is concatenation.

Decoding: Since no codeword is a prefix of any other codeword, just process bits until you get a match. Then discard the bits and go from the rest of the compressed file. ***Example:*** If encoding is 100011001101, get a match on $100 =$ `c`. That leaves 011001101. Get a match on $0 =$ `a`. That leaves 11001101. Get a match on $1100 =$ `f`. That leaves 1101. Get a match on $1101 =$ `e`. So the encoded file represents `cafe`.

Binary tree representation

Use a binary tree whose leaves are the characters. The codeword for a character is given by the simple path from the root down to that character's leaf, where going left is 0 and going right is 1.



Here, each leaf has its character and frequency (in thousands). Each internal node holds the sum of the frequencies of the leaves in its subtree.

An optimal code is always given by a full binary tree: each internal node has 2 children $\Rightarrow$ if C is the alphabet for the characters, then the tree has $|C|$ leaves and $|C| - 1$ internal nodes.

How to compute the number of bits to encode a file for alphabet C given tree T: For each character $c \in C$, denote its frequency by $c.freq$. Denote the depth of c in T by $d_T(c)$, which equals the length of c's codeword. Then the number of bits to encode the file, the ***cost*** of T, is

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \; .$$

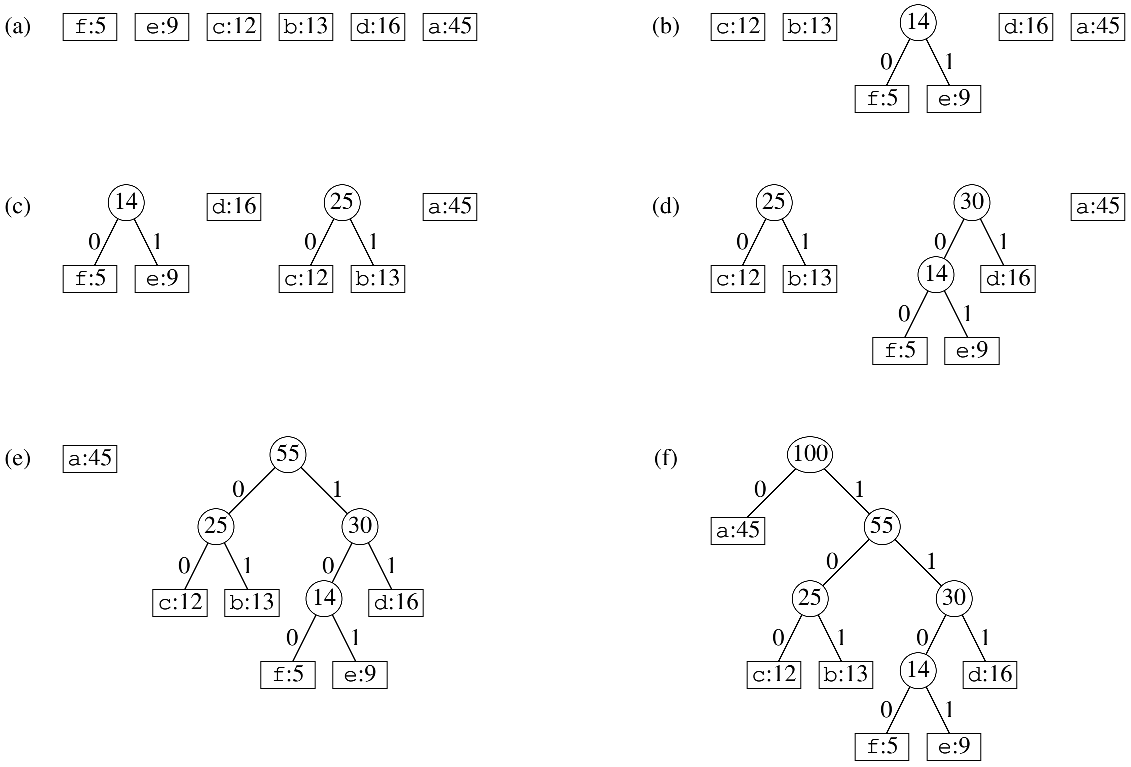
Constructing a Huffman code

[Named after David Huffman.] The algorithm builds tree T bottom-up. It repeatedly selects two nodes with the lowest frequency and makes them children of

a new node whose frequency is the sum of the two nodes' frequencies. It uses a min-priority queue Q keyed on the *freq* attribute, which all nodes have.

```
HUFFMAN($C$)
  $n = |C|$
  $Q = C$
  for $i = 1$ to $n - 1$
      allocate a new node $z$
      $x = $ EXTRACT-MIN($Q$)
      $y = $ EXTRACT-MIN($Q$)
      $z.left = x$
      $z.right = y$
      $z.freq = x.freq + y.freq$
      INSERT($Q, z$)
  return EXTRACT-MIN($Q$)      // the root of the tree is the only node left
```

Example: Using the frequencies from before:



Running time: Let $n = |C|$. The running time depends on how the min-priority queue Q is implemented. If with a binary min-heap, can initialize Q in $O(n)$ time. The **for** loop runs $n - 1$ times, and each INSERT and EXTRACT-MIN call takes $O(\lg n)$ time $\Rightarrow$ $O(n \lg n)$ time in all.

Correctness

Show the greedy-choice and optimal-substructure properties.

Lemma (Greedy-choice property)

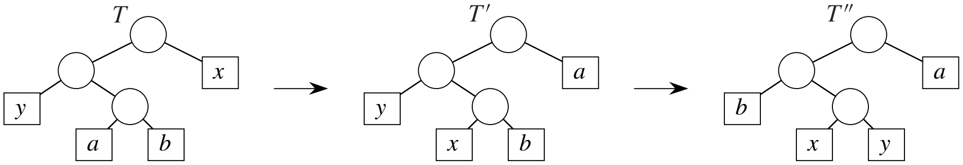
For alphabet C, let x and y be the two characters with the lowest frequencies. Then there exists an optimal prefix-free code for C where the codewords for x and y have the same length and differ only in the last bit.

Proof Given a tree T for some optimal prefix-free code, modify it so that x and y are sibling leaves of maximum depth. Then the codewords for x and y will have the same length and differ in the last bit.

Let a, b be two characters that are sibling leaves of maximum depth in T. Assume wlog that $a.\mathit{freq} \le b.\mathit{freq}$ and $x.\mathit{freq} \le y.\mathit{freq}$. Must have $x.\mathit{freq} \le a.\mathit{freq}$ and $y.\mathit{freq} \le b.\mathit{freq}$.

Could have $x.\mathit{freq} = a.\mathit{freq}$ or $y.\mathit{freq} = b.\mathit{freq}$. If $x.\mathit{freq} = b.\mathit{freq}$, then $a.\mathit{freq} = b.\mathit{freq} = x.\mathit{freq} = y.\mathit{freq}$ (Exercise 15.3-1), and the lemma is trivially true. So assume that $x.\mathit{freq} \ne b.\mathit{freq} \Rightarrow x \ne b$.

In T: exchange a and x, producing T'.
In T': exchange b and y, producing T''.
In T'', x and y are sibling leaves of maximum depth.



Claim

$B(T') \le B(T)$. (Exchanging a and x does not increase the cost.)

Proof of claim

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) - \sum_{c \in C} c.\mathit{freq} \cdot d_{T'}(c) \\
&\quad = x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_{T'}(x) - a.\mathit{freq} \cdot d_{T'}(a) \\
&\quad = x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_T(a) - a.\mathit{freq} \cdot d_T(x) \\
&\quad = (a.\mathit{freq} - x.\mathit{freq})(d_T(a) - d_T(x)) \\
&\quad \ge 0 .
\end{aligned}$$

The last line follows because $x.\mathit{freq} \le a.\mathit{freq}$ and a is a maximum-depth leaf $\Rightarrow d_T(a) \ge d_T(x)$. ■ (claim)

Similarly, $B(T'') \le B(T')$ because exchanging y and b doesn't increase the cost. Therefore, $B(T'') \le B(T') \le B(T)$. T is optimal $\Rightarrow B(T) \le B(T'') \Rightarrow B(T'') = B(T) \Rightarrow T''$ is optimal, and x and y are sibling leaves of maximum depth. ■

The lemma shows that to build up an optimal tree, can begin with the greedy choice of merging the two characters with lowest frequency. Greedy because the cost of a merger is the sum of the frequencies of its children and the cost of a tree equals the sum of the costs of its mergers (Exercise 15.3-4).

Lemma (Optimal-substructure property)

For alphabet C, let x, y be the two characters with minimum frequency. Let $C' = (C - \{x, y\}) \cup z$ for a new character z with $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Let T' be a tree representing an optimal prefix-free code for C', and T be T' with the leaf for z replaced by an internal node with children x and y. Then T represents an optimal prefix-free code for C.

Proof $c \in C - \{x, y\} \Rightarrow d_T(c) = d_{T'}(c) \Rightarrow c.\mathit{freq} \cdot d_T(c) = c.\mathit{freq} \cdot d_{T'}(c)$. $d_T(x) = d_T(y) = d_{T'}(z) + 1 \Rightarrow$

$$\begin{aligned} x.\mathit{freq} \cdot d_T(x) + y.\mathit{freq} \cdot d_T(y) &= (x.\mathit{freq} + y.\mathit{freq})(d_{T'}(z) + 1) \\ &= z.\mathit{freq} \cdot d_{T'}(z) + (x.\mathit{freq} + y.\mathit{freq}) , \end{aligned}$$

so that $B(T) = B(T') + x.\mathit{freq} + y.\mathit{freq}$, which is equivalent to $B(T') = B(T) - x.\mathit{freq} - y.\mathit{freq}$.

Now suppose T doesn't represent an optimal prefix-free code for C. Then $B(T'') < B(T)$ for some optimal tree T''. By the previous lemma, without loss of generality, T'' has x and y as siblings. Replace the common parent of x and y by a leaf z with $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$ and call the resulting tree T'''. Then,

$$\begin{aligned} B(T''') &= B(T'') - x.\mathit{freq} - y.\mathit{freq} \\ &< B(T) - x.\mathit{freq} - y.\mathit{freq} \\ &= B(T') , \end{aligned}$$

so that T' was not optimal, a contradiction. ■

Theorem

HUFFMAN produces an optimal prefix-free code.

Proof The greedy-choice and optimal-substructure properties both apply. ■

Offline caching

In a computer, a ***cache*** is memory that is smaller but faster than main memory. It holds a small subset of what's in main memory. Caches store data in ***blocks***, also known as ***cache lines***, usually 32, 64, or 128 bytes. *[We use the term* blocks *in this discussion, rather than* cache lines.*]*

A program makes a sequence of memory requests to blocks. Each block usually has several requests to some data that it holds.

The cache size is limited to k blocks, starting out empty before the first request. Each request causes either 0 or 1 block to enter the cache, and either 0 or 1 block to be evicted. A request for block b may have one of three outcomes:

1. b is already in the cache due to some previous request $\Rightarrow$ ***cache hit***. The cache remains unchanged.
2. b is not already in the cache, but the cache is not yet full (contains $< k$ blocks). b goes into the cache, so that the cache now contains one more block than before the request.

3. b is not already in the cache, but the cache is full (contains k blocks). Some block already in the cache is evicted, and b goes into the cache.

The latter two outcomes are ***cache misses***.

Goal: Given a sequence of block requests, minimize the number of cache misses.

When a cache miss occurs but the cache is not full, that's a ***compulsory miss***—no way to avoid it.

When a cache miss occurs and the cache is full, want to choose which block to evict to allow for the fewest cache misses over the entire sequence of requests.

Typically an online problem: don't know the request sequence in advance. Have to process each request as it arrives, with no future knowledge.

Instead, we consider offline caching: know the entire request sequence in advance. Why study a scenario so unrealistic?

- Sometimes you do know the entire request sequence in advance.
  Example: Viewing main memory as the cache and the full data as residing on a disk, there are algorithms that plan out the entire set of disk reads and writes in advance.
- Can model real-world problems.
  Example: Have a fixed schedule of n events and locations. Managing k agents, need to ensure that there's one agent at each location when an event occurs. Want to minimize how many times some agent has to move. Agents = blocks, events = requests, moving an agent = cache miss.
- Can use offline caching performance as a baseline for evaluating online caching algorithms. (See Section 27.3.)

Furthest in future

Strategy for offline caching: evict the block whose next access in the request sequence comes furthest in the future. Intuitively, makes sense—don't keep the block if you're not going to need it soon.

Optimal substructure

Define subproblem (C, i): processing requests for blocks $b_i, \ldots, b_n$ with cache configuration C at the time b_i is requested. C is a subset of all the blocks, $|C| \le k$.

A solution to (C, i) is a sequence of decisions that specifies which block, if any, to evict for each request. An optimal solution minimizes the number of cache misses.

Let S be an optimal solution to (C, i). Let C' be the contents of the cache after processing the request for b_i and S' be the solution to subproblem $(C', i + 1)$ (the next subproblem). Request for b_i gives a cache hit $\Rightarrow C' = C$. A cache miss $\Rightarrow$ $C' \ne C$.

Either way, S' must be an optimal solution to $(C', i + 1)$. *[Here comes the usual cut-and-paste argument.]* If not, then some other solution S'' for $(C', i + 1)$ is optimal, with fewer cache misses than S'. Combining S'' with the decision of S yields another solution for (C, i) with fewer misses than $S \Rightarrow S$ was not optimal.

Recursive solution

Define $R_{C,i}$ as the set of all cache configurations that can immediately follow configuration C after processing a request for b_i.

- Cache hit ⇒ no change in the cache ⇒ $R_{C,i} = \{C\}$.
- Compulsory miss ⇒ insert b_i into the cache ⇒ $R_{C,i} = \{C \cup \{b_i\}\}$.
- Non-compulsory miss (cache is full) ⇒ evict any of the k blocks in the cache and insert $b_i \Rightarrow R_{C,i} = \{(C - \{x\}) \cup \{b_i\} : x \in C\}$.

Let $miss(C, i)$ = the minimum number of cache misses in a solution for subproblem (C, i). Recurrence for $miss(C, i)$:

$$miss(C,i) = \begin{cases} 0 & \text{if } i = n \text{ and } b_n \in C \ , \\ 1 & \text{if } i = n \text{ and } b_n \notin C \ , \\ miss(C, i+1) & \text{if } i < n \text{ and } b_i \in C \ , \\ 1 + \min\{miss(C', i+1) : C' \in R_{C,i}\} & \text{if } i < n \text{ and } b_i \notin C \ . \end{cases}$$

Therefore, offline caching has optimal substructure.

Greedy-choice property

Solving offline caching either bottom-up or recursively with memoization would generate a huge solution space, since it would have to consider all possible k-subsets of the distinct blocks in the request sequence. Instead, can make the greedy choice: furthest-in-future.

Theorem

Suppose that a cache miss occurs when the cache is full with configuration C. When block b_i is requested, let $z = b_m$ be the block in C whose next request is furthest in the future. Then, evicting z when processing the request for b_i is included in some optimal solution for subproblem (C, i).

[A rigorous proof of this theorem is rather complicated. It is omitted in these notes.]

This theorem says that furthest-in-future has the greedy-choice property. Since offline caching also exhibits optimal substructure, furthest-in-future gives the minimum number of cache misses.

Solutions for Chapter 15: Greedy Algorithms

Solution to Exercise 15.1-1

The tricky part is determining which activities are in the set S_{ij}. If activity k is in S_{ij}, then we must have $i < k < j$, which means that $j - i \geq 2$, but we must also have that $f_i \leq s_k$ and $f_k \leq s_j$. If we start k at $j - 1$ and decrement k, we can stop once k reaches i, but we can also stop once we find that $f_k \leq f_i$, since then activities $i + 1$ through k cannot be compatible with activity i.

We create two fictitious activities, a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$. We are interested in a maximum-size set $A_{0,n+1}$ of mutually compatible activities in $S_{0,n+1}$. We'll use tables $c[0:n+1, 0:n+1]$, as in recurrence (15.2) (so that $c[i,j] = |A_{ij}|$), and $\mathit{activity}[0:n+1, 0:n+1]$, where $\mathit{activity}[i,j]$ is the activity k that we choose to put into A_{ij}.

We fill the tables in according to increasing difference $j - i$, which we denote by l in the pseudocode. Since $S_{ij} = \emptyset$ if $j - i < 2$, we initialize $c[i,i] = 0$ for all i and $c[i, i+1] = 0$ for $0 \leq i \leq n$. As in RECURSIVE-ACTIVITY-SELECTOR and GREEDY-ACTIVITY-SELECTOR, the start and finish times are given as arrays s and f, where we assume that the arrays already include the two fictitious activities and that the activities are sorted by monotonically increasing finish time.

```
DYNAMIC-ACTIVITY-SELECTOR$(s, f, n)$
  let $c[0:n+1, 0:n+1]$ and $activity[0:n+1, 0:n+1]$ be new tables
  for $i = 0$ to $n$
      $c[i,i] = 0$
      $c[i,i+1] = 0$
  $c[n+1, n+1] = 0$
  for $l = 2$ to $n+1$
      for $i = 0$ to $n-l+1$
          $j = i+l$
          $c[i,j] = 0$
          $k = j-1$
          while $f[i] < f[k]$
              if $f[i] \le s[k]$ and $f[k] \le s[j]$ and $c[i,k] + c[k,j] + 1 > c[i,j]$
                  $c[i,j] = c[i,k] + c[k,j] + 1$
                  $activity[i,j] = k$
              $k = k-1$
  print "A maximum-size set of mutually compatible activities
          has size " $c[0, n+1]$
  print "The set contains "
  PRINT-ACTIVITIES$(c, activity, 0, n+1)$
```

```
PRINT-ACTIVITIES$(c, activity, i, j)$
  if $c[i,j] > 0$
      $k = activity[i,j]$
      print $k$
      PRINT-ACTIVITIES$(c, activity, i, k)$
      PRINT-ACTIVITIES$(c, activity, k, j)$
```

The PRINT-ACTIVITIES procedure recursively prints the set of activities placed into the optimal solution A_{ij}. It first prints the activity k that achieved the maximum value of $c[i,j]$, and then it recurses to print the activities in A_{ik} and A_{kj}. The recursion bottoms out when $c[i,j] = 0$, so that $A_{ij} = \emptyset$.

Whereas GREEDY-ACTIVITY-SELECTOR runs in $\Theta(n)$ time, the DYNAMIC-ACTIVITY-SELECTOR procedure runs in $O(n^3)$ time.

Solution to Exercise 15.1-2

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 15.1-3

- 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 15.1. It will be the first activity selected, and no other activities are compatible with it.

Solution to Exercise 15.1-4
This solution is also posted publicly

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. Moreover, it can produce a result that uses more lecture halls than necessary. Consider activities with the intervals $\{[1,4),[2,5),[6,7),[4,8)\}$. GREEDY-ACTIVITY-SELECTOR would choose the activities with intervals $[1,4)$ and $[6,7)$ for the first lecture hall, and then each of the activities with intervals $[2,5)$ and $[4,8)$ would have to go into its own hall, for a total of three halls used. An optimal solution would put the activities with intervals $[1,4)$ and $[4,8)$ into one hall and the activities with intervals $[2,5)$ and $[6,7)$ into another hall, for only two halls used.

There is a correct 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 $s_i \leq t$ but won't finish until $f_i > t$) and halls that are free at time t. (As in the activity-selection problem in Section 15.1, we are assuming that activity time intervals are half open—i.e., that if $s_i \geq f_j$, 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 $m \leq n$ 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 17.3-6.]

Solution to Exercise 15.1-5

We can no longer use the greedy algorithm to solve this problem. However, as we show, the problem still has an optimal substructure which allows us to formulate a dynamic programming solution. The analysis here follows closely the analysis of Section 15.1 in the book. We define the value of a set of compatible events as the sum of values of events in that set. Let S_{ij} be defined as in Section 15.1. An *optimal solution* to S_{ij} is a subset of mutually compatible events of S_{ij} that has maximum value. Let A_{ij} be an optimal solution to S_{ij}. Suppose A_{ij} includes an event a_k. Let A_{ik} and A_{kj} be defined as in Section 15.1. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the value of maximum-value set A_{ij} is equal to the value of A_{ik} plus the value of A_{kj} plus v_k.

The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where the value of A'_{kj} is greater than the value of A_{kj}, then we could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. We would have constructed a set of mutually compatible activities with greater value than that of A_{ij}, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik}.

Let us denote the value of an optimal solution for the set S_{ij} by *value*$[i, j]$. Then, we would have the recurrence

$$value[i, j] = value[i, k] + value[k, j] + v_k \; .$$

Of course, since we do not know that an optimal solution for the set S_{ij} includes activity a_k, we would have to examine all activities in S_{ij} to find which one to choose, so that

$$value[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \; , \\ \max\{value[i, k] + value[k, j] + v_k : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset \; . \end{cases}$$

While implementing the recurrence, the tricky part is determining which activities are in the set S_{ij}. If activity k is in S_{ij}, then we must have $i < k < j$, which means that $j - i \geq 2$, but we must also have that $f_i \leq s_k$ and $f_k \leq s_j$. If we start k at $j - 1$ and decrement k, we can stop once k reaches i, but we can also stop once we find that $f_k \leq f_i$, since then activities $i + 1$ through k cannot be compatible with activity i.

We create two fictitious activities, a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$. We are interested in a maximum-size set $A_{0,n+1}$ of mutually compatible activities in $S_{0,n+1}$. We'll use tables *value*$[0 : n + 1, 0 : n + 1]$, as in the recurrence, and *activity*$[0 : n + 1, 0 : n + 1]$, where *activity*$[i, j]$ is the activity k that we choose to put into A_{ij}.

We fill the tables in according to increasing difference $j - i$, which we denote by l in the pseudocode. Since $S_{ij} = \emptyset$ if $j - i < 2$, we initialize *value*$[i, i] = 0$ for all i and *value*$[i, i + 1] = 0$ for $0 \leq i \leq n$. As in RECURSIVE-ACTIVITY-SELECTOR and GREEDY-ACTIVITY-SELECTOR, the start and finish times are given as arrays s and f, where we assume that the arrays already include the two fictitious activities and that the activities are sorted by monotonically increasing finish time. The array v specifies the value of each activity.

```
MAX-VALUE-ACTIVITY-SELECTOR$(s, f, v, n)$
  let $value[0:n+1, 0:n+1]$ and $activity[0:n+1, 0:n+1]$ be new tables
  for $i = 0$ to $n$
      $value[i,i] = 0$
      $value[i,i+1] = 0$
  $value[n+1, n+1] = 0$
  for $l = 2$ to $n+1$
      for $i = 0$ to $n-l+1$
          $j = i+l$
          $value[i,j] = 0$
          $k = j-1$
          while $f[i] < f[k]$
              if $f[i] \leq s[k]$ and $f[k] \leq s[j]$ and
                      $value[i,k] + value[k,j] + v_k > value[i,j]$
                  $value[i,j] = value[i,k] + value[k,j] + v_k$
                  $activity[i,j] = k$
              $k = k-1$
  print "A maximum-value set of mutually compatible activities has value "
          $value[0, n+1]$
  print "The set contains "
  PRINT-ACTIVITIES$(value, activity, 0, n+1)$
```

```
PRINT-ACTIVITIES$(value, activity, i, j)$
  if $value[i,j] > 0$
      $k = activity[i,j]$
      print $k$
      PRINT-ACTIVITIES$(value, activity, i, k)$
      PRINT-ACTIVITIES$(value, activity, k, j)$
```

The PRINT-ACTIVITIES procedure recursively prints the set of activities placed into the optimal solution A_{ij}. It first prints the activity k that achieved the maximum value of $value[i,j]$, and then it recurses to print the activities in A_{ik} and A_{kj}. The recursion bottoms out when $value[i,j] = 0$, so that $A_{ij} = \emptyset$.

Whereas GREEDY-ACTIVITY-SELECTOR runs in $\Theta(n)$ time, the MAX-VALUE-ACTIVITY-SELECTOR procedure runs in $O(n^3)$ time.

Solution to Exercise 15.2-1

To show that the fractional knapsack problem has the greedy-choice property, suppose that some choice is not the greedy choice. Suppose that the greedy choice at that point is item m with value per pound v_m/w_m, but that the choice made is p pounds of item q, where $v_q/w_q < v_m/w_m$. Let $r = \min\{p, w_m\}$ be the amount of item m that can replace item q as the choice. The value taken decreases by rv_q/w_q and increases by rv_m/w_m. Since $v_q/w_q < v_m/w_m$, we have that $rv_q/w_q < rv_m/w_m$ and the overall value increases.

Solution to Exercise 15.2-2
This solution is also posted publicly

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, then v_i value is added, and the thief can choose from items $1, \ldots, i - 1$ up to the weight limit $w - w_i$, gaining $c[i - 1, w - w_i]$ additional value. On the other hand, if the thief decides not to take item i, then choices remain from items $1, \ldots, i - 1$ up to the weight limit w, giving $c[i - 1, w]$ value. The better of these two choices should be made.

The algorithm takes as inputs the maximum weight W, the number n of items, 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($v, w, n, W$)
  let $c[0:n, 0:W]$ be a new array
  for $w = 0$ to $W$
      $c[0, w] = 0$
  for $i = 1$ to $n$
      $c[i, 0] = 0$
      for $w = 1$ to $W$
          if $w_i \leq w$ and $v_i + c[i-1, w-w_i] > c[i-1, w]$
              $c[i, w] = v_i + c[i-1, w-w_i]$
          else $c[i, w] = c[i-1, w]$
```

We can use the c table to deduce the set of items to take 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 15.2-4

The optimal strategy is the obvious greedy one. Starting with both bottles full, Professor Gekko should go to the westernmost place that he can refill his bottles within m miles of Grand Forks. Fill up there. Then go to the westernmost refilling location he can get to within m miles of where he filled up, fill up there, and so on.

Looked at another way, at each refilling location, Professor Gekko should check whether he can make it to the next refilling location without stopping at this one. If he can, skip this one. If he cannot, then fill up. Professor Gekko doesn't need to know how much water he has or how far the next refilling location is to implement this approach, since at each fillup, he can determine which is the next location at which he'll need to stop.

This problem has optimal substructure. Suppose there are n possible refilling locations. Consider an optimal solution with s refilling locations and whose first stop is at the kth location. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining $n - k$ 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 refilling locations beyond the start that are within m miles of the start. The greedy solution chooses the kth location as its first stop. No station beyond the kth works as a first stop, since Professor Gekko would run out of water first. If a solution chooses a location $j < k$ as its first stop, then Professor Gekko could choose the kth location instead, having at least as much water when he leaves the kth location as if he'd chosen the jth location. Therefore, he would get at least as far without filling up again if he had chosen the kth location.

If there are n refilling locations on the map, Professor Gekko needs to inspect each one just once. The running time is $O(n)$.

Solution to Exercise 15.2-5

A simple greedy algorithm solves the problem. The following procedure takes as input a set $P = \{x_1, x_2, \ldots, x_n\}$ of points on the real line and returns a set S, which is the smallest set of unit-length closed intervals that contains all the points in P. Because P is a set, we assume that the x_i values are unique.

```
FIND-UNIT-INTERVALS(P)
  sort the set $P$ so that $x_1 < x_2 < \cdots < x_n$
  $S = \emptyset$
  $i = 1$
  while $i \le n$  // consider the points in order
      create a unit interval $z = [z', z'']$, where $z' = x_i$ and $z'' = x_i + 1$
      $S = S \cup \{z\}$
      $i = i + 1$
      while $i \le n$ and $x_i \le z''$   // see which other points $z$ contains
          $i = i + 1$
  return $S$
```

To prove that this algorithm returns an optimal set of unit intervals, we need to prove optimal substructure and the greedy-choice property.

Optimal substructure follows from the usual cut-and-paste argument. Let's characterize a subproblem as taking $x_i, x_{i+1}, \ldots, x_j$, where the points are consecutive after sorting, so that the original problem is $x_1, \ldots, x_n$. Suppose that an optimal choice is an interval that covers the k points $x_r, x_{r+1}, \ldots, x_{r+k}$. That leaves the subproblems $x_i, \ldots, x_{r-1}$ and $x_{r+k+1}, \ldots, x_j$. Optimal substructure says that an optimal solution to the $x_i, \ldots, x_j$ problem includes optimal solutions to these two subproblems. If an optimal solution to the $x_i, \ldots, x_j$ subproblem included a solution to the $x_i, \ldots, x_{r-1}$ subproblem with m unit-length intervals, and an *optimal* solution to the $x_i, \ldots, x_{r-1}$ subproblem had $m' < m$ intervals, then we could substitute the solution with m' intervals in the optimal solution for the $x_i, \ldots, x_j$ subproblem and get a solution with fewer intervals. A similar argument holds for the $x_{r+k+1}, \ldots, x_j$ subhproblem.

To show the greedy-choice property, we are considering only subproblems for the points $x_i, \ldots, x_n$. We want to show that the interval $z = [x_i, x_i + 1]$ is included in some optimal solution. An optimal set S' of intervals must contain x_i, and so there must be an interval $y = [y', y'']$ in S' such that $y' \le x_i \le y''$. If $y' = x_i$, then $y = z$, and we're done. Otherwise, let $S = S' - \{y\} \cup \{z\}$. Set S contains the same number of intervals as S', and any point of $x_i, \ldots, x_n$ contained in y is also contained in z, so that S contains all the points in $x_i, \ldots, x_n$. Therefore, S is optimal.

The running time is $O(n \lg n)$, dominated by the time to sort the points. The two nested **while** loops consider each point once.

Solution to Exercise 15.2-6

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, if $W_G + W_E \ge W$, then take all items in set G, and take as much of the items in set E as will fit in the remaining capacity $W - W_G$.
- Otherwise, $W_G + W_E < W$. In this case, take all the items in sets G and E, and then 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 15.2-7
This solution is also posted publicly

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 15.3-1

We are given that $x.\mathit{freq} \le y.\mathit{freq}$ are the two lowest frequencies in order, and that $a.\mathit{freq} \le b.\mathit{freq}$. Now,

$$\begin{aligned} & b.\mathit{freq} = x.\mathit{freq} \\ \Rightarrow\ & a.\mathit{freq} \le x.\mathit{freq} \\ \Rightarrow\ & a.\mathit{freq} = x.\mathit{freq} \quad \text{(since } x.\mathit{freq} \text{ is the lowest frequency)}\ , \end{aligned}$$

and since $y.\mathit{freq} \le b.\mathit{freq}$,

$$\begin{aligned} & b.\mathit{freq} = x.\mathit{freq} \\ \Rightarrow\ & y.\mathit{freq} \le x.\mathit{freq} \\ \Rightarrow\ & y.\mathit{freq} = x.\mathit{freq} \quad \text{(since } x.\mathit{freq} \text{ is the lowest frequency)}\ . \end{aligned}$$

Thus, if we assume that $x.\mathit{freq} = b.\mathit{freq}$, then we have that each of $a.\mathit{freq}$, $b.\mathit{freq}$, and $y.\mathit{freq}$ equals $x.\mathit{freq}$, and so $a.\mathit{freq} = b.\mathit{freq} = x.\mathit{freq} = y.\mathit{freq}$.

Solution to Exercise 15.3-2

Let T be a nonfull binary tree. Then T contains an internal node u with only one child, v. Replace edge (u, v) by edges from u to the child or children of v, obtaining a tree T'. Tree T' represents a better coding than T because all the characters in the subtree rooted at v have have one bit removed from their codewords. Thus, a nonfull binary tree does not correspond to a prefix-free code.

Solution to Exercise 15.3-3

An optimal Huffman code for the first 8 Fibonacci numbers:

frequency	character	codeword
1	`a`	0000000
1	`b`	0000001
2	`c`	000001
3	`d`	00001
5	`e`	0001
8	`f`	001
13	`g`	01
21	`h`	1

In general, after merging i nodes, the tree is a rightgoing spine of internal nodes, with each internal node having a leaf as its left child and the deepest internal node having `a` and `b` as its children. The frequency in the root after merging i nodes is $F_{i+2} - 1$, which can be shown by induction. Since $F_{i+2} - 1 = F_i + F_{i+1} - 1$ and the character with frequency F_i has already been merged after merging i nodes, the next merge takes the root with frequency $F_{i+2} - 1$ and the character with frequency F_{i+1}.

To generalize the code for the first n Fibonacci numbers, the code for the character with frequency F_i is 0^{n-1} if $i = 1$ and $0^{n-i}1$ if $i \geq 2$, where 0^j represents a string of j 0s.

Solution to Exercise 15.3-4

The proof is by induction on the number of merge operations that created the tree. Note that the proof does not rely on the merge operations being performed in the order given by the HUFFMAN procedure, just that the full binary tree can be constructed by merge operations. For a tree T, let $I(T)$ denote the set of internal nodes of T and $C(T)$ denote the set of leaves of T. Then, the goal is to prove that $B(T) = \sum_{x \in I(T)} x.\mathit{freq}$, where $B(T)$ is defined as $\sum_{c \in C(T)} c.\mathit{freq} \cdot d_T(c)$.

The basis is a tree T with one internal node z. The children of this node are leaves, say l and r, with frequencies $l.\mathit{freq}$ and $r.\mathit{freq}$ and depth 1 in T. We have

$$\begin{aligned}
\sum_{x \in I(T)} x.\mathit{freq} &= z.\mathit{freq} \\
&= l.\mathit{freq} + r.\mathit{freq} \\
&= (l.\mathit{freq} \cdot d_T(l)) + (r.\mathit{freq} \cdot d_T(r)) \\
&= \sum_{c \in C(T)} c.\mathit{freq} \cdot d_T(c)
\end{aligned}$$

$$= B(T) .$$

Now consider a tree T with left and right subtrees L and R, respectively. The inductive hypothesis holds for both L and R: $B(L) = \sum_{x \in I(L)} x.\mathit{freq}$ and $B(R) = \sum_{x \in I(R)} x.\mathit{freq}$. Let T have root z, so that $z.\mathit{freq} = \sum_{c \in C(T)} c.\mathit{freq}$. Then, we have

$$\begin{aligned}
B(T) &= \sum_{c \in C(L)} c.\mathit{freq} \cdot d_T(c) + \sum_{c \in C(R)} c.\mathit{freq} \cdot d_T(c) \\
&= \sum_{c \in C(L)} c.\mathit{freq} \cdot (d_L(c) + 1) + \sum_{c \in C(R)} c.\mathit{freq} \cdot (d_R(c) + 1) \\
&= \sum_{c \in C(L)} c.\mathit{freq} \cdot d_L(c) + \sum_{c \in C(R)} c.\mathit{freq} \cdot d_R(c) + \sum_{c \in C(L)} c.\mathit{freq} + \sum_{c \in C(R)} c.\mathit{freq} \\
&= B(L) + B(R) + \sum_{c \in C(T)} c.\mathit{freq} \\
&= \sum_{x \in I(L)} x.\mathit{freq} + \sum_{x \in I(R)} x.\mathit{freq} + z.\mathit{freq} \qquad \text{(by the inductive hypothesis)} \\
&= \sum_{x \in I(T)} x.\mathit{freq} .
\end{aligned}$$

Solution to Exercise 15.3-5

A full binary tree with n leaves has $n - 1$ internal nodes, for a total of $2n - 1$ nodes. You can specify the structure of the tree by a preorder walk, with a 1 at a node indicating that it's an internal node and a 0 meaning that the node is a leaf. Because the code is on n characters, $\lceil \lg n \rceil$ bits are needed to represent each character, so that if each leaf represents a character, $n \lceil \lg n \rceil$ bits represent all the characters. Store the characters in the order in which the preorder walk visits the leaves. The total number of bits is then $2n - 1 + n \lceil \lg n \rceil$.

Solution to Exercise 15.3-6

As long as $n \neq 1, 2, 4$, there is a full ternary tree with n leaves. To adapt Huffman's algorithm for ternary codewords, instead of merging the two nodes with lowest frequency, merge the three nodes with lowest frequency. The proof of optimality is analogous to the proofs of Lemmas 15.2 and 15.3.

Solution to Exercise 15.3-7

Let f be the minimum frequency among the 256 characters, so that the maximum character frequency is less than $2f$. The first merge creates an internal node with frequency at least $2f$, so that this internal node won't be selected for merging until all the characters have been merged. The same holds for each of the other 127 merges of characters, so that the first 128 merges create internal nodes with frequencies at least $2f$ and less than $4f$. The next merge merges two such internal nodes, creating an internal node with frequency at least $4f$ and less than $8f$. This internal node won't be selected for merging until all the other internal nodes with frequencies less than $4f$ have been merged. This process continues, always merging internal nodes of the same height, until a single complete binary tree has emerged. The leaves of this tree will all have depth 8, so that all the Huffman codes comprise exactly 8 bits.

Solution to Exercise 15.3-8

Suppose that the input file consists of n bits. There are 2^n possible n-bit input files. The total number of bits in all possible n-bit input files is $n2^n$, so that decompression must be able to recover $n2^n$ bits in all. Since the compression scheme is lossless, there must be at least 2^n different output files.

Suppose that all compressed files contain fewer than n bits. For $k = 0, 1, \ldots, n-1$, there are 2^k different files with k bits, so that the total number of bits in all compressed files is

$$\begin{aligned}\sum_{k=0}^{n-1} k2^k &= n2^n - 2^{n+1} + 2 \qquad \text{(see below)} \\ &< n2^n \,,\end{aligned}$$

so that there are not enough bits in the compressed files to recover all of the original input files.

To see why $\sum_{k=0}^{n-1} k2^k = n2^n - 2^{n+1} + 2$, we start with equation (A.6): $\sum_{k=0}^{n} x^k = (x^{n+1}-1)/(x-1)$. Substituting $n-1$ for n gives $\sum_{k=0}^{n-1} x^k = (x^n-1)/(x-1)$. Taking derivatives of both sides with respect to x and multiplying both sides by x gives

$$\begin{aligned}\sum_{k=0}^{n-1} kx^k &= x \cdot \frac{(x-1)(nx^{n-1}) - (x^n-1)(x-1)}{(x-1)^2} \\ &= x \cdot \frac{nx^n - nx^{n-1} - x^{n+1} + x^n + x - 1}{(x-1)^2} \\ &= x \cdot \frac{n(x^n - x^{n-1}) - (x-1)x^n + x + 1}{(x-1)^2} \\ &= x \cdot \frac{nx^{n-1} - (x-1)x^n + x - 1}{(x-1)^2} \,.\end{aligned}$$

Setting $x = 2$ gives

$$\begin{aligned}\sum_{k=0}^{n-1} k2^k &= 2 \cdot \frac{n2^{n-1} - (2-1)2^n + 2 - 1}{(2-1)^2} \\ &= 2(n2^{n-1} - 2^n + 1) \\ &= n2^n - 2^{n+1} + 2 \,.\end{aligned}$$

Solution to Exercise 15.4-2

Let $k = 2$, and consider a request sequence that cycles among three blocks: $b_1, b_2, b_3, b_1, b_2, b_3, b_1, b_2, b_3, b_1, b_2, b_3, \ldots$. Here are the states of caches that use LRU and furthest-in-future after each request, with each cache miss marked with an X. For the LRU cache, the least recently used block is listed above the most recently used block. For the furthest-in-future cache, the lower-numbered block is listed above the higher-numbered block.

request	b_1	b_2	b_3	b_1	b_2	b_3	b_1	b_2	b_3	b_1	b_2	b_3	$\ldots$
	b_1	b_1	b_2	b_3	b_1	b_2	b_3	b_1	b_2	b_3	b_1	b_2	$\ldots$
LRU		b_2	b_3	b_1	b_2	b_3	b_1	b_2	b_3	b_1	b_2	b_3	$\ldots$
	X	X	X	X	X	X	X	X	X	X	X	X	$\ldots$
furthest-	b_1	b_1	b_1	b_1	b_2	b_2	b_1	b_1	b_1	b_1	b_2	b_2	$\ldots$
in-		b_2	b_3	b_3	b_3	b_3	b_2	b_2	b_3	b_3	b_3	b_3	$\ldots$
future	X	X	X		X		X		X		X		$\ldots$

After the compulsory misses to fill the cache, the LRU cache continues to have a cache miss upon every request, but the furthest-in-future cache has cache misses only on alternate requests.

Solution to Exercise 15.4-3

The problem arises when $|D_j| = k-1$, $b_j \notin D_j$, solution S evicts a block $w \in D_j$ that is not block z, and $b_j = x$. Solution S' has a cache hit, so that $C_{S',j+1} = D_j \cup \{b_j\}$ and $w \in C_{S',j+1}$. As the proof points out, $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$, $C_{S,j+1} = D_{j+1} \cup \{z\}$, and $C_{S',j+1} = D_{j+1} \cup \{w\}$. Since $x \notin D_j$, block w cannot be block x, and so $C_{S',j+1} \neq D_{j+1} \cup \{x\}$.

Solution to Exercise 15.4-4

For a given request sequence, let S be the sequence of blocks brought into the cache upon each request, where some requests may entail multiple blocks brought into the cache. We'll construct a sequence S' that brings in at most one block per request and at most as many blocks as S brings in altogether.

Suppose that S and S' have the same cache contents until a request for block b_i causes a cache miss, and that S brings in not only b_i but also some other block $b_k \neq b_i$. Sequence S' does not bring in b_k, but instead waits until the next request for b_k before bringing it in, assuming that there is a future request for b_k. If there is a future request for b_k, then S' incurs a cache miss upon that request. Since S has already brought b_k into the cache, both S and S' incur the cost of bringing in b_k. If there is no future request for b_k, then S incurs a cost that S' does not.

Solution to Problem 15-1

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 $c \le n$. 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 $1 \le n < 5$, then $c = 1$. A solution may consist only of pennies, and so it must contain the greedy choice.
- If $5 \le n < 10$, 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 $25 \leq n$, 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 $k \leq n$, 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 $j = \max\{0 \leq i \leq k : c^i \leq n\}$, 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 $a_i \geq c$ for some $0 \leq i < k$, 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 $j = \max\{0 \leq i \leq k : c^i \leq n\}$, 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 $n \geq c^j$, we have that $\sum_{i=0}^{j-1} a_i c^i \geq c^j$. Now suppose that the non-greedy solution is optimal. By the above lemma, $a_i \leq c - 1$ for $i = 0, 1, \ldots, j-1$. Thus,

$$\sum_{i=0}^{j-1} a_i c^i \leq \sum_{i=0}^{j-1} (c-1) c^i$$

$$
\begin{aligned}
&= (c-1)\sum_{i=0}^{j-1} c^i \\
&= (c-1)\,\frac{c^j - 1}{c-1} \\
&= c^j - 1 \\
&< c^j \ ,
\end{aligned}
$$

which contradicts our earlier assertion that $\sum_{i=0}^{j-1} a_i c^i \geq c^j$. 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 $j \geq 1$.

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 $j \leq 0$.

To develop a recursive formulation, we have to check all denominations, giving

$$
c[j] = \begin{cases} 0 & \text{if } j \leq 0 \ , \\ 1 + \min\limits_{1 \leq i \leq k} \{c[j - d_i]\} & \text{if } j > 1 \ . \end{cases}
$$

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 examining $c[j]$ for $j < 0$ by ensuring that $j \geq d_i$ 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)
  let c[0 : n] and denom[1 : n] be new arrays
  c[0] = 0
  for j = 1 to n
      c[j] = ∞
      for i = 1 to k
          if j ≥ d_i and 1 + c[j − d_i] < c[j]
              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
      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.

Solution to Problem 15-2

a. To minimize the average completion time, run the tasks in monotonically increasing order of their processing times.

Suppose the tasks run in the order $a_1, a_2, \ldots, a_n$. Then task a_1 has completion time $c_1 = p_1$, task a_2 has completion time $c_2 = p_1 + p_2$, task a_3 has completion time $c_3 = p_1 + p_2 + p_3$, and so on, so that task a_k has completion time $\sum_{i=1}^{k} p_i$. The average completion time is minimized by minimizing $\sum_{i=1}^{n} c_i$. Noting that

$$\sum_{i=1}^{n} c_i = np_1 + (n-1)p_2 + (n-2)p_3 + \cdots + p_n ,$$

we see that this sum is minimized when $p_1, p_2, \ldots, p_n$ are in monotonically increasing order.

b. To minimize the average completion time, run the task that has been released and not yet completed with the shortest time remaining. When a task is released, if its processing time is less than the time remaining for the running task, preempt the running task by the new task, keeping track of the time remaining in the preempted task. A priority queue of ready tasks can determine which task to run next.

One way to think about this situation is that if a task is running while a new task is released, then break the running task into the portion already run and the portion yet to be run. This situation reduces to the situation in part (a).

A little more formally, suppose that at time t, task a_1 has time remaining r_1 and task a_2 has time remaining r_2, where $r_1 < r_2$. The greedy choice is to run a_1 before a_2. If a_1 runs before a_2, then the average completion time for the two tasks is $((t+r_1)+(t+r_1+r_2))/2 = (2t+2r_1+r_2)/2$. If a_2 runs before a_1, then the average completion time for the two tasks is $((t+r_2)+(t+r_2+r_1))/2 = (2t+2r_2+r_1)/2$. Since $r_1 < r_2$, the first way gives a lower average completion time.

Lecture Notes for Chapter 16: Amortized Analysis

Chapter 16 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)

```
MULTIPOP(S, k)
  while not STACK-EMPTY(S) and $k > 0$
      POP(S)
      $k = k - 1$
```

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 $A[i] == 1$
      $A[i] = 0$
      $i = i + 1$
  if $i < k$
      $A[i] = 1$
```

Example: $k = 3$

[Underlined bits flip. Show costs later.]

counter	A	
value	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
⋮	⋮	15

Cost of INCREMENT $= \Theta(\#\text{ of bits flipped})$.

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. The cost represents the cost to get to the value on that line, not the cost of getting to the next line.]

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	$\lfloor n/4 \rfloor$
	⋮	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	⋮	
$i \geq k$	never	0

$$\begin{aligned}
\text{Therefore, total \# of flips} &= \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \\
&< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\
&= n \cdot \frac{1}{1 - 1/2} \\
&= 2n \ .
\end{aligned}$$

Therefore, cost of n INCREMENTs is $O(n)$.

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*.

$$\begin{aligned}\text{Let } c_i &= \text{actual cost of } i\text{th operation} ,\\ \widehat{c}_i &= \text{amortized cost of } i\text{th operation} .\end{aligned}$$

Then require $\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

$$\text{Total credit stored} = \sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\;\geq\;}_{\text{had better be}} 0 .$$

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.

$$\begin{aligned}
\text{Let } D_i &= \text{data structure after } i\text{th operation} , \\
D_0 &= \text{initial data structure} , \\
c_i &= \text{actual cost of } i\text{th operation} , \\
\widehat{c}_i &= \text{amortized cost of } i\text{th operation} .
\end{aligned}$$

Potential function $\Phi : D_i \to \mathbb{R}$

$\Phi(D_i)$ 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,

$$\begin{aligned}\Delta\Phi(D_i) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi(D_i) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 .\end{aligned}$$

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 an item is inserted into a table, it's an ***elementary insertion***.

```
TABLE-INSERT($T, x$)
  if $T.size$ == 0
      allocate $T.table$ with 1 slot
      $T.size = 1$
  if $T.num$ == $T.size$                              // expand?
      allocate $new\text{-}table$ with $2 \cdot T.size$ slots
      insert all items in $T.table$ into $new\text{-}table$   // $T.num$ elem insertions
      free $T.table$
      $T.table = new\text{-}table$
      $T.size = 2 \cdot T.size$
  insert $x$ into $T.table$                           // 1 elem insertion
  $T.num = T.num + 1$
```

Initially, $T.num = T.size = 0$.

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, not every operation triggers an expansion:

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is exact power of } 2 \text{ ,} \\ 1 & \text{otherwise .} \end{cases}$$

$$\begin{aligned} \text{Total cost} &= \sum_{i=1}^{n} c_i \\ &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} \\ &< n + 2n \\ &= 3n \end{aligned}$$

Therefore, **aggregate analysis** says amortized cost per operation = 3.

Accounting method

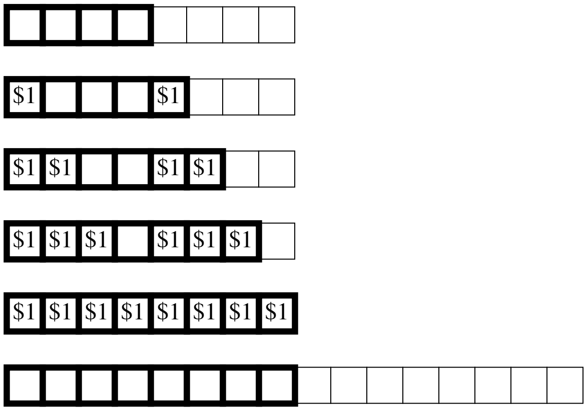
Charge \$3 per insertion of x.

- \$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 the table has just expanded, $\mathit{size} = \mathit{num} = m$ just before the expansion, and $\mathit{size} = 2m$ and $\mathit{num} = m$ just after the expansion. The next expansion occurs when $\mathit{size} = \mathit{num} = 2m$.

- 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!

Example: Table with *size* = 8 and *num* = 4, so that it has no stored credit.



[A slot has an item stored in it only if it is drawn with a heavy outline.]

Each insertion costs \$1, puts \$1 on the item just inserted, and puts \$1 on some other item. By the time table fills ($num = size = 8$), each item has \$1 stored on it ⇒ enough to pay to reinsert all the items after doubling the table size.

Potential method

Think of potential being 0 just after an expansion—when $num = size/2$. (Just as the accounting method had no stored credit just after an expansion.)

As elementary insertions occur, the table needs to build enough potential to pay to reinsert all the items at the next expansion. The next expansion occurs when $num = size$, which is after $size/2$ insertions. The potential needs to be *size* at that time.
⇒ Over $size/2$ insertions, potential needs to go from 0 to *size*.
⇒ Potential increase per insertion is

$$\frac{size}{size/2} = 2 .$$

Use the potential function

$$\Phi(T) = 2(T.num - T.size/2) .$$

[Since the potential is defined on a table T, now using attribute notation: $T.num$ and $T.size$ instead of just num and size.]

The potential equals 0 just after expansion, when $T.num = T.size/2$.
The potential equals $T.size$ when the table fills, when $T.num = T.size$.

Initial potential is 0, and the potential is always nonnegative ⇒ sum of the amortized costs gives an upper bound on the sum of the actual costs.

[Previous editions of the text used the potential function $\Phi(T) = 2 \cdot T.num - T.size$, which works out the same as above. It's more convenient to think of how far $T.num$ is from $T.size/2$ when building up potential, which is why we use the version above in the fourth edition.]

$$\begin{aligned}\Phi_i &= \text{potential after the } i\text{th operation },\\ \Delta\Phi_i &= \text{change in potential due to the } i\text{th operation },\\ \widehat{c}_i &= c_i + \Delta\Phi_i \;.\end{aligned}$$

When the i th insertion does not trigger expansion

$c_i = 1$ and $\Delta\Phi_i = 2 \Rightarrow \widehat{c}_i = c_i + \Delta\Phi_i = 1 + 2 = 3$.

When the i th insertion triggers an expansion

$$\begin{aligned}\mathit{num}_i &= \text{number of items in the table after the } i\text{th operation },\\ \mathit{size}_i &= \text{size of the table after the } i\text{th operation }.\end{aligned}$$

Before insertion:
$\mathit{size}_{i-1} = \mathit{num}_{i-1} = i - 1 \Rightarrow$

$$\begin{aligned}\Phi_{i-1} &= 2(\mathit{size}_{i-1} - \mathit{size}_{i-1}/2)\\ &= \mathit{size}_{i-1}\\ &= i - 1 \;.\end{aligned}$$

After expansion: $\Phi = 0$.
After inserting the new item: $\Phi = 2$.
$\Rightarrow \Delta\Phi_i = 2 - (i - 1) = 3 - i$.
Actual cost $c_i = i$ ($i - 1$ reinsertions + 1 insertion of new item)
$\Rightarrow$ amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Delta\Phi_i\\ &= i + (3 - i)\\ &= 3 \;,\end{aligned}$$

so that the amortized cost of an insertion is $O(1)$.

[Figure showing size_i (dashed line), num_i (thin solid line), and Φ_i (thicker solid line. You might instead want to project Figure 16.4, available with all the figures in the text on the MIT Press website for the book.]

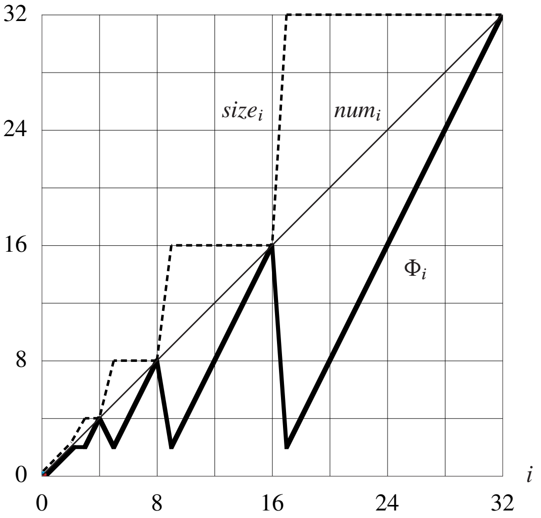


Table expansion and contraction

Now supporting both insertion and deletion from the table.

Want to limit wasted space in the table ⇒ contract the table when the load factor becomes too small. Allocate a new, smaller table and copy over the items.

Requirements:

- Bound the load factor from above by 1 and from below by some positive constant (we'll use $1/4$).
- Bound the amortized cost of insertion and deletion by a constant.

Actual cost of an operation is the number of elementary insertions or deletions.

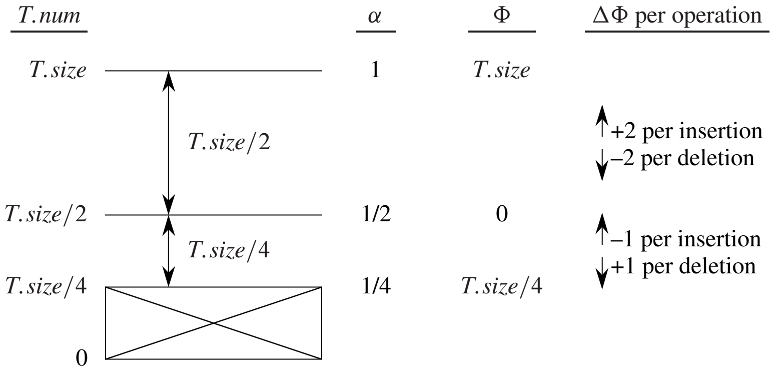
Bad approach: Since doubling the table size when it becomes full, try halving the size when the load factor drops below $1/2$. This strategy can cause the amortized costs to be high:

- Perform n operations on a table of size $n/2$, n is a power of 2.
- The first $n/2$ are insertions. Total cost is $\Theta(n)$. Now have $\mathit{num} = \mathit{size} = n/2$.
- The second $n/2$ operations are the sequence

  insert, delete, delete, insert, insert, delete, delete, insert, insert, ...
- The first insertion in the sequence triggers an expansion. The second deletion triggers a contraction. Each pair of insertions then triggers an expansion, and each pair of deletions triggers a contraction.
- Cost of each expansion or contraction is $\Theta(n)$, and there are $\Theta(n)$ of them ⇒ total cost is $\Theta(n^2)$ ⇒ amortized cost of each operation is $\Theta(n)$, not constant.
- The flaw in the strategy is that after expansion, not enough deletions occur to pay for a contraction. And after contraction, not enough insertions occur to pay for expansion.

Solution: Allow the load factor to drop below $1/2$, down to $1/4$. Halve the table size when deletion causes the load factor to drop below $1/4$ (instead of below $1/2$) ⇒ load factor bounded from below by $1/4$.

Idea: An expansion or contraction should deplete all the built-up potential so that immediately after expansion or contraction, the load factor is $1/2$ and the potential is 0.



As the load factor moves away from $1/2$, the potential builds up to pay for copying all the items ⇒ the potential needs to increase to *num* by the time the load factor reaches either 1 or $1/4$.

How to design the potential function

When the load factor is $\geq 1/2$: Use the same potential function as for insertion only: $\Phi(T) = 2(T.num - T.size/2)$. Each insertion increases the potential by 2, and each deletion decreases the potential by 2.

When the load factor is $< 1/2$ (but always $\geq 1/4$): Need $size/4$ deletions to get the load factor from $1/2$ down to $1/4$. The potential needs to be $size/4$ to pay for the $size/4$ reinsertions ⇒ potential increase per deletion (but only when load factor $< 1/2$) is

$$\frac{size/4}{size/4} = 1 .$$

And when the load factor is $< 1/2$, each insertion should decrease the potential by 1. For $1/4 \leq \alpha < 1/2$, use the potential function

$$\Phi(T) = T.size/2 - T.num .$$

Each deletion increases the potential by 1, and each insertion decreases the potential by 1.

Overall potential function:

$$\Phi(T) = \begin{cases} 2(T.num - T.size/2) & \text{if } \alpha(T) \geq 1/2 , \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 . \end{cases}$$

Initially: $num_0 = 0$, $size_0 = 0$, $\Phi_0 = 0$.

When no expansion or contraction occurs and the load factor does not cross 1/2

Actual cost $c_i = 1$.
Amortized cost $\widehat{c}_i = c_i + \Delta\Phi_i$.

- Insertion when $\alpha_{i-1} \geq 1/2$: $\Delta\Phi_i = 2 \Rightarrow \widehat{c}_i = 1 + 2 = 3$.
- Deletion when $\alpha_i \geq 1/2$: $\Delta\Phi = -2 \Rightarrow \widehat{c}_i = 1 - 2 = -1$.
- Insertion when $\alpha_i < 1/2$: $\Delta\Phi = -1 \Rightarrow \widehat{c}_i = 1 - 1 = 0$.
- Deletion when $\alpha_{i-1} < 1/2$: $\Delta\Phi = 1 \Rightarrow \widehat{c}_i = 1 + 1 = 2$.

When expansion or contraction occurs

An insertion that causes expansion is the same as when the only operation was insertion: $\widehat{c}_i = 3$.

For a deletion that causes contraction: $num_{i-1} = size_{i-1}/4$ beforehand ⇒

$$\begin{aligned} \Phi_{i-1} &= size_{i-1}/2 - num_{i-1} \\ &= size_{i-1}/2 - size_{i-1}/4 \\ &= size_{i-1}/4 . \end{aligned}$$

Then the item is deleted, then $num_i = size/2 - 1 \Rightarrow \alpha_i < 1/2 \Rightarrow$

$$\begin{aligned} \Phi_i &= size_i/2 - num_i \\ &= 1 \end{aligned}$$

$\Rightarrow \Delta\Phi_i = 1 - size_{i-1}/4$.
Actual cost $c_i = size_{i-1}/4$, for deleting 1 item and copying $size_{i-1}/4 - 1$ items $\Rightarrow$

$$\begin{aligned}\widehat{c}_i &= c_i + \Delta\Phi_i \\ &= size_{i-1}/4 + (1 - size_{i-1}/4) \\ &= 1\ .\end{aligned}$$

When the load factor crosses 1/2

Deletion: $num_{i-1} = size_{i-1}/2$ and $\alpha_{i-1} = 1/2$ before.
$num_i = size_i/2 - 1$ and $\alpha_i < 1/2$ after.
$\alpha_{i-1} = 1/2 \Rightarrow \Phi_{i-1} = 0$.
$\alpha_i < 1/2 \Rightarrow \Phi_i = size_i/2 - num_i = 1$.
Therefore, $\Delta\Phi_i = 1 - 0 = 1$.
No contraction occurs $\Rightarrow c_i = 1$
$\Rightarrow \widehat{c}_i = 1 + 1 = 2$.

Insertion: When the load factor goes from below $1/2$ to $1/2$, $\Delta\Phi_i$ is the opposite for the case of deletion $\Rightarrow \Delta\Phi = -1$.
Actual cost $c_i = 1$
$\Rightarrow \widehat{c}_i = 1 - 1 = 0$.

Summary

In each case, the amortized cost of insertion or deletion is bounded by a constant $\Rightarrow$ actual time for any sequence of n operations is $O(n)$.

Solutions for Chapter 16: Amortized Analysis

Solution to Exercise 16.1-1

With a MULTIPUSH operation, the amortized cost of stack operations would no longer be $O(1)$. The cost of a single MULTIPUSH that pushes k items onto the stack is $\Theta(k)$.

Solution to Exercise 16.1-2

Let n be a power of 2, and consider the following sequence of n INCREMENT and DECREMENT operations on a counter with $k = \lg n$ bits. Start with $n/2$ INCREMENT operations, so that the counter's value is $n/2$, with 1 in the leftmost bit and 0s in the rightmost $k-1$ bits. Then perform an alternating sequence of $n/2$ DECREMENT and INCREMENT operations.

The total cost of the first $n/2$ INCREMENT operations is $\Theta(n)$. But each subsequent operation has to flip all k bits each time, for a total cost of $\Theta(nk)$.

Solution to Exercise 16.1-3
This solution is also posted publicly

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 $\sum 2^j$.)

$$\text{Average cost of operation} = \frac{\text{Total cost}}{\text{\# operations}} < 3 \ .$$

By aggregate analysis, the amortized cost per operation $= O(1)$.

Solution to Exercise 16.2-1

Charge \$2 for each PUSH and POP operation and \$0 for each COPY operation, which copies the entire stack. Upon calling PUSH, \$1 pays for the operation, and the other \$1 is stored on the item pushed. Upon calling POP, again \$1 pays for the operation, and the other \$1 is stored 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 k PUSH and POP operations occur between two consecutive COPY operations, \$$k$ of credit are 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 16.2-2

This solution is also posted publicly

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	Amortized 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, $\displaystyle\sum_{i=1}^{n} \widehat{c}_i = 3n$.

We know from Exercise 16.1-3 that $\displaystyle\sum_{i=1}^{n} c_i < 3n$.

Then we have $\displaystyle\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i \Rightarrow \text{credit} = \text{amortized cost} - \text{actual cost} \geq 0$.

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 16.2-3

This solution is also posted publicly

We introduce a new field $A.max$ to hold the index of the high-order 1 in A. Initially, $A.max$ is set to -1, since the low-order bit of A is at index 0 and there are initially no 1s in A. The value of $A.max$ is updated as appropriate when the counter is incremented or reset, and this value limits 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 INCREMENT operations.

```
INCREMENT(A, k)
  i = 0
  while i < k and A[i] == 1
      A[i] = 0
      i = i + 1
  if i < k
      A[i] = 1
      // Additions to book's INCREMENT start here.
      A.max = max {A.max, i}
  else A.max = -1

RESET(A)
  for i = 0 to A.max
      A[i] = 0
  A.max = -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 $A.max$.

Setting and resetting of bits by INCREMENT will work exactly as for the original counter in the book: \$1 pays to set one bit to 1, \$1 is placed on the bit that is set to 1 as credit, and the credit on each 1 bit pays to reset the bit during incrementing.

In addition, \$1 pays for updating *max*, and if *max* increases, 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 $A.max$, 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 $A.max$, 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 that the sequence of n INCREMENT and RESET operations takes $O(n)$ time.

Solution to Exercise 16.3-1

Set $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$. Since $\Phi(D_i) \geq \Phi(D_0)$, we have $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \geq 0$. The amortized cost of the ith operation using Φ' is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi'(D_i) - \Phi'(D_{i-1}) \\ &= c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_{i-1}) - \Phi(D_0)) \\ &= c_i + \Phi(D_i) - \Phi(D_{i-1}) ,\end{aligned}$$

which is the amortized cost using Φ.

Solution to Exercise 16.3-2

Define the potential of D_i by

$$\Phi(D_i) = \begin{cases} 0 & \text{if } i = 0 , \\ 2i - 2^{\lfloor \lg i \rfloor + 1} & \text{if } i \geq 1 . \end{cases}$$

Since $2^{\lfloor \lg i \rfloor} \leq i$ for $i \geq 1$, the value of $\Phi(D_i)$ is nonnegative for all i.

If i is not a power of 2, then the amortized cost of the ith operation is

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2i - 2^{\lfloor \lg i \rfloor + 1}) - (2(i-1) - 2^{\lfloor \lg (i-1) \rfloor + 1}) \\ &= 1 + (2i - 2^{\lfloor \lg i \rfloor + 1}) - (2(i-1) - 2^{\lfloor \lg i \rfloor + 1}) \\ &= 3 . \end{aligned}$$

If $i = 2^k$ for some nonnegative integer k, then the amortized cost of the ith operation is

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= i + (2i - 2^{\lfloor \lg i \rfloor + 1}) - (2(i-1) - 2^{\lfloor \lg (i-1) \rfloor + 1}) \\ &= 2^k + (2 \cdot 2^k - 2^{\lfloor \lg 2^k \rfloor + 1}) - (2(2^k - 1) - 2^{\lfloor \lg (2^k - 1) \rfloor + 1}) \\ &= 2^k + (2^{k+1} - 2^{k+1}) - (2^{k+1} - 2 - 2^{(k-1)+1}) \\ &= 2^k - (2^k - 2) \\ &= 2 . \end{aligned}$$

Solution to Exercise 16.3-3

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 $\Phi(D_i) \geq 0$.

Before proving that we achieve the desired amortized times, we show that if $n \geq 2$, then $n \ln \frac{n}{n-1} \leq 2$. We have

$$\begin{aligned} n \ln \frac{n}{n-1} &= n \ln \left(1 + \frac{1}{n-1}\right) \\ &= \ln \left(1 + \frac{1}{n-1}\right)^n \\ &\leq \ln \left(e^{\frac{1}{n-1}}\right)^n \qquad \text{(since } 1 + x \leq e^x \text{ for all real } x\text{)} \\ &= \ln e^{\frac{n}{n-1}} \\ &= \frac{n}{n-1} \\ &\leq 2 , \end{aligned}$$

assuming that $n \geq 2$. (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

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln 1 + k \cdot 1 \ln 1 - 0 \\
&= 0 .
\end{aligned}$$

If the ith operation inserts into a nonempty heap, then $n_i = n_{i-1} + 1 \geq 2$, and the amortized cost is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln n_i + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_i + k n_i \ln n_i - k(n_i - 1) \ln(n_i - 1) \\
&= k \ln n_i + k n_i \ln n_i - k n_i \ln(n_i - 1) + k \ln(n_i - 1) \\
&< 2k \ln n_i + k n_i \ln \frac{n_i}{n_i - 1} \\
&\leq 2k \ln n_i + 2k \qquad \text{(since } n_i \geq 2\text{)} \\
&= O(\lg n_i) .
\end{aligned}$$

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

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln 1 + 0 - k \cdot 1 \ln 1 \\
&= 0 .
\end{aligned}$$

If the ith operation extracts from a heap with more than one item, then $n_i = n_{i-1} - 1$ and $n_{i-1} \geq 2$, and the amortized cost is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln n_{i-1} + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_{i-1} + k(n_{i-1} - 1) \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_{i-1} + k n_{i-1} \ln(n_{i-1} - 1) - k \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\
&= k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln \frac{n_{i-1} - 1}{n_{i-1}} \\
&< k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln 1 \\
&= k \ln \frac{n_{i-1}}{n_{i-1} - 1} \\
&\leq k \ln 2 \qquad \text{(since } n_{i-1} \geq 2\text{)} \\
&= O(1) .
\end{aligned}$$

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 $\Phi(D_i) \geq 0$, 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 Exercise 16.3-4

Starting with

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) ,$$

subtracting $\Phi(D_i) - \Phi(D_{i-1})$ from both sides gives

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (\hat{c}_i + \Phi(D_{i-1}) - \Phi(D_i)) \\
&= \sum_{i=0}^{n} \hat{c}_i + \Phi(0) - \Phi(D_n) && \text{(telescoping sum)} \\
&= \sum_{i=0}^{n} \hat{c}_i + s_0 - s_n && (\Phi(D_i) \text{ equals number of objects in the stack}) \\
&\leq 2n + s_0 - s_n && (\hat{c}_i \leq 2) .
\end{aligned}$$

Solution to Exercise 16.3-5

The implementation is the same as in the answer given for Exercise 10.1-7:

> Call our two stacks S_1 and S_2.
>
> To ENQUEUE, push a new element onto stack S_1. This operation takes $O(1)$ time.
>
> To DEQUEUE, pop the top element from stack S_2. If stack S_2 is empty when a DEQUEUE is requested, first empty stack S_1 into stack S_2 by popping elements one at a time from stack S_1 and pushing them onto stack S_2. Copying the stack reverses its order, so that the oldest element is then on top and can be removed with DEQUEUE.
>
> DEQUEUE takes $O(1)$ time in the best case and $O(n)$ time in the worst case. Each element is moved from stack S_1 to stack S_2 at most one time, so that the time averaged over all operations is $O(1)$.

Let the constant k the denote the maximum of the actual costs of pushing onto or popping from a stack, p_i denote the number of elements in stack S_1 after the ith operation, and q_i denote the number of elements in stack S_2 after the i operation. Use the potential function $\Phi(D_i) = 2k(2p_i + q_i)$. We get the following amortized costs:

- ENQUEUE: $c_i \le k$, $p_i = p_{i-1} + 1$, and $q_i = q_{i-1}$. Then, $\Phi(D_i) - \Phi(D_{i-1}) = 4k$, so that
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le 5k \\ &= O(1) . \end{aligned}$$
- DEQUEUE when stack S_2 is not empty: $c_i \le k$, $p_i = p_{i-1}$, and $q_i = q_{i-1} + 1$. Then, $\Phi(D_i) - \Phi(D_{i-1}) = -2k$, so that
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k - 2k \\ &= -k \\ &= O(1) . \end{aligned}$$
- DEQUEUE when stack S_2 is empty: $c_i \le 2kp_{i-1} + 1$ (each item in stack S_1 is popped from S_1 and pushed onto stack S_2, and then one item is popped from S_2), $p_i = 0$, and $q_i = p_{i-1} + 1$. Then,
$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= 2kp_{i-1} - 2k(2p_{i-1}) \\ &= -2kp_{i-1} , \end{aligned}$$
so that
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le (2kp_{i-1} + 1) - 2kp_{i-1} \\ &= 1 \\ &= O(1) . \end{aligned}$$

Solution to Exercise 16.3-6

The data structure is as simple as it gets: an unsorted array. The INSERT operation takes $O(1)$ actual worst-case time. Outputting the elements of multiset S consists of just printing the elements in the array, taking $O(|S|)$ time. To perform DELETE-LARGER-HALF, first use the linear-time median-finding algorithm to find the median. Then use the PARTITION procedure to partition the array around the median, and just delete the larger side.

We can use the accounting method or the potential method for the amortized analysis. In the accounting method, charge \$3 per insertion. One dollar pays for the insertion, and the other \$2 sits on the inserted element. During a DELETE-LARGER-HALF operation, spend \$1 per element for finding the median and partitioning. Each element that is deleted gives its remaining dollar to one of the elements that is not deleted, so that each element still has \$2 of credit after the DELETE-LARGER-HALF operation.

With the potential method, let n_i be the number of elements after the ith INSERT or DELETE-LARGER-HALF operation, and let the constant k be the time spent per element during DELETE-LARGER-HALF. Assume that the time to insert one element is at most k. Define the potential function after the ith INSERT or DELETE-LARGER-HALF operation to be $\Phi_i = 2kn_i$. For INSERT, we have $c_i \le k$ and $n_i = n_{i-1} + 1$, so that

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= k + 2kn_i - 2k(n_i - 1) \\ &= 3k \ .\end{aligned}$$

For the DELETE-LARGER-HALF operation, we have $c_i = kn_{i-1}$ and $n_i = n_{i-1}/2$, so that

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= kn_{i-1} + 2k(n_{i-1}/2) - 2kn_{i-1} \\ &= 0 \ .\end{aligned}$$

Since the amortized costs of the operations are each $O(1)$, any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time.

Solution to Exercise 16.4-1

Before the first insertion, the table is empty, so that $\Phi_0 = 0$. After the first insertion, $num_1 = size_1 = 1$, so that $\Phi_1 = 2(1 - 1/2) = 1$ and $\hat{c}_1 = c_1 + \Phi_1 - \Phi_0 = 1 + 1 - 0 = 2$.

Solution to Exercise 16.4-2

Theorem 11.6, Corollary 11.7, and Theorem 11.8 all require the load factor α to be less than 1. Otherwise, because the number of probes has a fraction $1/(1-\alpha)$, the number of probes for inserting and searching could be unbounded.

One way to make the amortized cost per insertion (and per deletion) constant would be to double the table size when α exceeds 3/4, still halving the table size when α reduces below 1/4. Use the potential function

$$\Phi(T) = \begin{cases} 3(T.num - T.size/2) & \text{if } \alpha(T) \ge 1/2 \ , \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 \ . \end{cases}$$

When the load factor increases from 1/2 to 3/4, the potential increases from 0 to $3T.size/4$, which is enough to pay for copying all $(3/4)T.size$ elements to a new table.

Because some insertions require copying all the elements in the hash table, the expected value of the actual cost per insertion is not constant for every insertion.

Solution to Exercise 16.4-3

Take a cue from the potential function. When deleting from a table with $\alpha \geq 1/2$, if the item being deleted has 1 dollar on it, then remove 1 dollar from the slot of the item being deleted and remove 1 dollar from some other item currently in the table that has 1 dollar on it. Otherwise, remove 1 from two items that each have 1 dollar on them. When deleting from a table with $\alpha < 1/2$, leave 1 dollar in the vacated slot.

When inserting into a table with $\alpha \geq 1/2$, do as before, and put 1 dollar on the newly inserted item and 1 dollar on some item that does not yet have 1 dollar on it. When inserting into a table with $\alpha < 1/2$, remove a dollar from some empty slot.

We need to show that our balance will never go below 0. To do this, consider the cases where $\alpha = 1/2$, $\alpha > 1/2$, and $\alpha < 1/2$.

When $\alpha = 1/2$, our balance is 0.

When $\alpha > 1/2$, the number of dollars in the table is twice the number of items that exceed the load factor $1/2$. By putting in dollars when inserting, we will have enough money by the time $\alpha = 1$ to pay for moving each item when the table expands. By taking off dollars when deleting, we are able to maintain a positive balance.

When $\alpha < 1/2$, the number of dollars in the table is equal to the number of slots that would need to be filled to get to $\alpha = 1/2$. Therefore, when $\alpha < 1/4$, there will be enough dollars in the table to pay for moving each of the items in the table.

Solution to Exercise 16.4-4

Let's rewrite the potential function as

$$\Phi(T) = \begin{cases} 2(T.num - T.size/2) & \text{if } \alpha \geq 1/2 \ , \\ 2(T.size/2 - T.num) & \text{if } \alpha < 1/2 \ . \end{cases}$$

Note that when $\alpha \geq 1/2$, that is, when $T.num \geq T.size/2$, this potential function is the same as in the book. Therefore, the amortized cost of TABLE-DELETE when $\alpha_i \geq 1/2$ is still -1. If $\alpha_{i-1} = 1/2$, then the operation deleted one item, so that $\alpha_i < 1/2$. Then, $\Phi_{i-1} = 0$ and $\Phi_i = 2$; since $c_i = 1$, we get that $\hat{c}_i = 1 + 2 - 0 = 3$.

Now, suppose that $\alpha_{i-1} = 1/3$, so that TABLE-DELETE causes the table to contract. Then we have $num_{i-1} = size_{i-1}/3$, $size_i = (2/3)size_{i-1}$, and $num_i = num_{i-1} - 1$. The actual cost is num_{i-1}, since one item is deleted, and the remaining $num_{i-1} - 1$ items are copied to the new table upon contraction. The potential just before the call of TABLE-DELETE is

$$\begin{aligned} \Phi_{i-1} &= 2(size_{i-1}/2 - num_{i-1}) \\ &= 2(size_{i-1}/2 - size_{i-1}/3) \\ &= 2(size_{i-1}/6) \end{aligned}$$

$$
\begin{aligned}
&= size_{i-1}/3 \\
&= num_{i-1} \; .
\end{aligned}
$$

After contraction, the table goes from being one item fewer than $1/3$ full to one item fewer than $1/2$ full. That is, $num_i = size_i/2 - 1$, so that the potential afterward is $\Phi_i = 2$. Therefore, the amortized cost of TABLE-DELETE when the table contracts is $\widehat{c}_i = c_i + \Phi_i - \Phi_{i-1} = num_{i-1} + 2 - num_{i-1} = 2$.

Therefore, the amortized cost of TABLE-DELETE is at most 3 in all cases.

Solution to Problem 16-1

a. The bit that flips is the rightmost 1 in the binary representation of i.

b. Let $B(k)$ denote the number of bits examined when finding the rightmost 1 in the numbers 1 through $2^k - 1$. For $k = 1$, we have $B(1) = 1$. When you count from 1 to $2^k - 1$, you count from 1 to $2^{k-1} - 1$, then you have 2^{k-1}, and then $2^{k-1} + 1$ to $2^k - 1$. The pattern of rightmost 1s in the first and last $2^{k-1} - 1$ numbers is the same. For the number 2^{k-1}, all k bits must be examined to find the rightmost 1. Thus, we get the recurrence $B(k) = 2B(k-1)+k$. We'll show by substitution that this recurrence has the solution $B(k) = 2^{k+1} - (k+2)$. For the base case, we have $B(1) = 1 = 2^2 - 3$. For the inductive step, the inductive hypothesis is that $B(k-1) = 2^k - (k+1)$. We have

$$
\begin{aligned}
B(k) &= 2B(k-1) + k \\
&= 2(2^k - (k+1)) + k \\
&= 2^{k+1} - 2k - 2 + k \\
&= 2^{k+1} - (k+2) \; .
\end{aligned}
$$

Therefore, over all 2^k numbers, the total number of bits examined is $\Theta(2^k)$. Since $2^k - 1$ bits are flipped altogether and each bit flip takes constant time, the total time is $\Theta(2^k)$.

Solution to Problem 16-2

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. 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 replace A_0 with the new sorted array. Otherwise, merge sort the two arrays into another sorted array of size 2. If A_1 is empty, then replace A_1 with the new array; otherwise merge sort the arrays as before and continue. Since array A_i has size 2^i, merge sorting two arrays of size 2^i each produces one of size 2^{i+1}, which is the size of A_{i+1}. Thus, this method will result in another list of arrays with the same structure as before.

Let us analyze its worst-case running time. Assume that merge sort takes time $2m$ 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

$$\begin{aligned} T(n) &= 2\left(2^0 + 2^1 + \cdots + 2^{k-2}\right) \\ &= 2(2^{k-1} - 1) \\ &= 2^k - 2 \\ &= \Theta(n) . \end{aligned}$$

Therefore, the worst-case time to insert an element into this data structure is $\Theta(n)$.

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. Implement DELETE(x) 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 on. Mark array A_j as empty. The new arrays are created already sorted.

The cost of DELETE is $\Theta(n)$ in the worst case, occurring when $i = k - 1$ 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 16-4

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, with the leaves (nodes whose children are NIL) being red. 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 17: Augmenting Data Structures

Chapter 17 overview

- 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 red-black trees, plus:

- OS-SELECT(x, i): return a pointer to the 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.

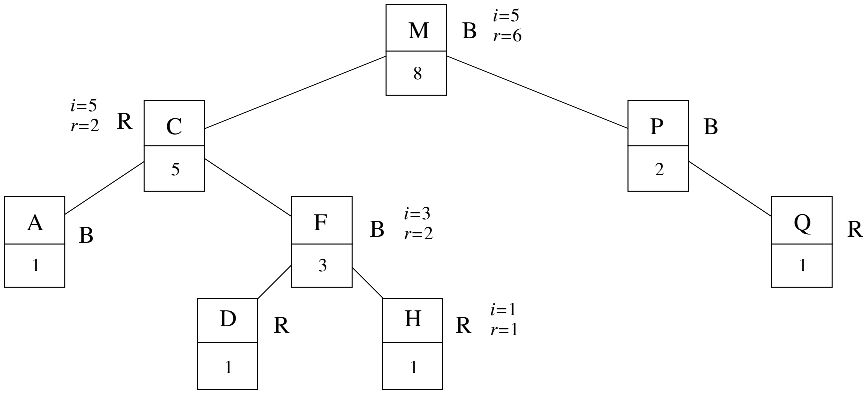
Augment by storing in each node x:

$x.size$ = number of nodes in subtree rooted at x .

- Includes x itself.
- Does not include leaves (sentinels).

Define for sentinel $T.nil.size = 0$.

Then $x.size = x.left.size + x.right.size + 1$.



*[**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.

Find the element with a given rank

```
OS-SELECT(x, i)
  r = x.left.size + 1    // rank of x within the subtree rooted at x
  if i == r
      return x
  elseif i < r
      return OS-SELECT(x.left, i)
  else return OS-SELECT(x.right, i − r)
```

Initial call: OS-SELECT($T.root$, i)

Try OS-SELECT($T.root$, 5).
[Values shown in figure above. Returns the node whose key is H.
OS-SELECT($T.root$, 3) *returns the node whose key is D.*
OS-SELECT($T.root$, 7) *returns the node whose key is P.]*

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 red-black tree has $O(\lg n)$ levels, have $O(\lg n)$ calls ⇒ $O(\lg n)$ time.

Find the rank of an element

```
OS-RANK(T, x)
  r = x.left.size + 1            // rank of x within the subtree rooted at x
  y = x                          // root of subtree being examined
  while y ≠ T.root
      if y == y.p.right                   // if root of a right subtree ...
          r = r + y.p.left.size + 1       // ... add in parent and its left subtree
      y = y.p                             // move y toward the root
  return r
```

Demo: Node D.

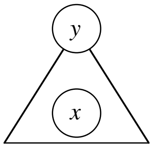
Why does this work?

> **Loop invariant:** At start of each iteration of **while** loop, r = rank of $x.key$ in subtree rooted at y.

Initialization: Initially, r = rank of $x.key$ in subtree rooted at x, and $y = x$.

Termination: Each iteration moves y toward the root and loop terminates when $y = T.root$ ⇒ the loop terminates and subtree rooted at y is entire tree. Therefore, r = rank of $x.key$ in entire tree.

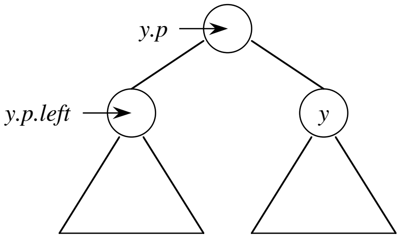
Maintenance: At end of each iteration, set $y = y.p$. So, show that if r = rank of $x.key$ in subtree rooted at y at start of loop body, then r = rank of $x.key$ in subtree rooted at $y.p$ at end of loop body.



[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 ⇒ 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 ⇒ add size of y's sibling's subtree, plus 1 for $y.p$, into r.



Analysis

y goes up one level in each iteration ⇒ $O(\lg n)$ time.

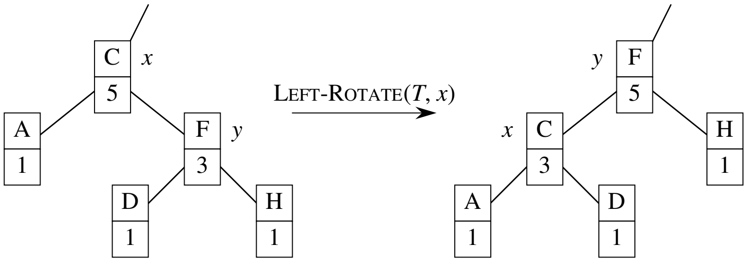
Maintaining subtree sizes

- Need to maintain *size* attributes 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* attribute 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.



$$\begin{aligned} y.\mathit{size} &= x.\mathit{size} \\ x.\mathit{size} &= x.\mathit{left}.\mathit{size} + x.\mathit{right}.\mathit{size} + 1 \end{aligned}$$

- Similar for right rotation.
- Therefore, can update in $O(1)$ time per rotation $\Rightarrow$ $O(1)$ time spent updating *size* attributes during fixup.
- Therefore, $O(\lg n)$ to insert.

Delete

Also 2 phases.

1. 4 cases, as shown in Figure 12.4. (Node being deleted is z. Node y is z's successor, and node x is y's right child.)
   - In cases (a) and (b), need to decrement the *size* of q and each ancestor of q, up to the root.
   - In case (c), need to recompute $y.size$ and then decrement the *size* of q and each ancestor of q, up to the root.
   - In case (d), first decrement $r.size$. Then recompute $y.size$. Then decrement the *size* of q and each ancestor of q, up to the root.

   Traversing path to the root and changing *size* attributes takes $O(\lg n)$ time.
2. Fixup.
   - During fixup, like insertion, only color changes and rotations.
   - ≤ 3 rotations $\Rightarrow$ $O(1)$ time spent updating *size* attributes 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 you 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. red-black tree.
2. $x.size$.
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 red-black tree with attribute f, where $x.f$ can be computed in $O(1)$ time based only on information in x, $x.left$, and $x.right$ (including $x.left.f$ and $x.right.f$). Then can maintain values of f in all nodes during insert and delete without affecting $O(\lg n)$ performance.

Proof Since $x.f$ depends only on x and its children, altering information in x propagates changes only upward (to $x.p$, $x.p.p$, $x.p.p.p$, $\ldots$, *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

When removing a node, need to update f for all its ancestors. Might first need to update f for two of its descendants (r and y in Figure 12.4). Fixup has ≤ 3 rotations. $O(\lg n)$ per rotation to propagate changes up to the root $\Rightarrow O(\lg n)$ to update f during fixup. ■ (theorem)

For some attributes, can get away with $O(1)$ per rotation. Example: *size* attribute.

Advantage of red-black trees

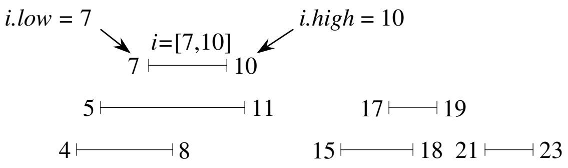
If an update after rotation requires traversing up to the root, then each rotation costs $O(\lg n)$ instead of $O(1)$. Because insertion and deletion in red-black trees bound the number of rotations to a constant, get $O(\lg n)$ time per insertion or deletion in the above theorem.

There are other balanced-tree schemes that can have $\Theta(\lg n)$ rotations per operation (example: AVL trees, Problem 13-3). If each rotation costs $\Theta(\lg n)$, operations could take $\Theta(\lg^2 n)$ time.

Interval trees

Maintain a set of intervals. For instance, time intervals.

Represent an interval i by an object with attributes $i.low$ and $i.high$.



[leave on board]

Each node x in an interval tree has an attribute $x.int$, so that there are attributes $x.int.low$ and $x.int.high$.

Operations

- Interval-Insert(T, x): $x.int$ already filled in.
- Interval-Delete(T, x)
- Interval-Search(T, i): return pointer to a node x in T such that $x.int$ overlaps interval i. Any overlapping node in T is OK. Return pointer to sentinel $T.nil$ if no overlapping node in T.

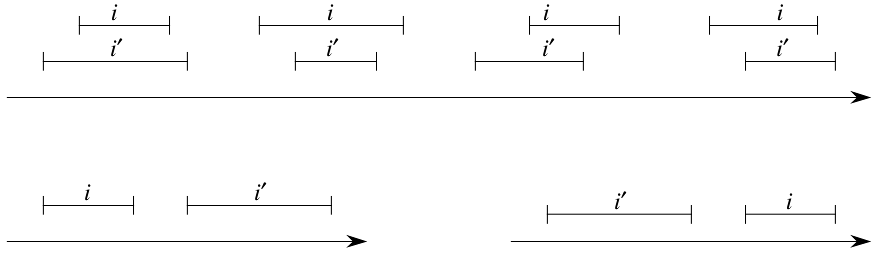
i and j overlap if and only if $i.low \le j.high$ and $j.low \le i.high$.

Interval trichotomy: For intervals i and i', exactly one of the following holds:

a. i and i' overlap.

b. i is to the left of i' ($i.high < i'.low$).

c. i is to the right of i' ($i'.high < i.low$).



[Go through examples of proper inclusion, overlap without proper inclusion, no overlap.]

Another way: i and j *don't* overlap if and only if
$i.low > j.high$ or $j.low > i.high$.
[leave this on board]

Recall the 4-part methodology.

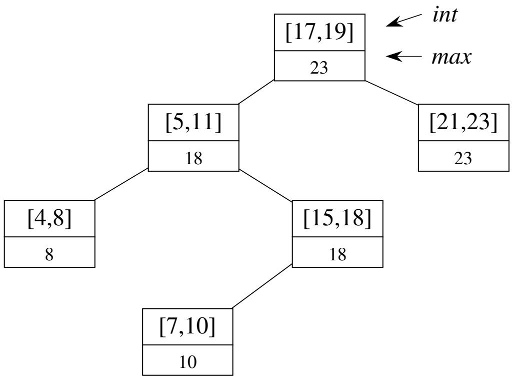
For interval trees

Underlying data structure: Use red-black trees.

- Each node x contains interval $x.int$.
- Key is low endpoint ($x.int.low$).
- Inorder walk would list intervals sorted by low endpoint.

Additional information: Each node x contains

$x.max =$ max endpoint value in subtree rooted at x .



[Node colors are not shown, since they are not necessary to understand how an interval tree works. Leave on board.]

$$x.max = max \begin{cases} x.int.high , \\ x.left.max , \\ x.right.max \end{cases}$$

Could $x.left.max > x.right.max$? Sure. Position in tree is determined only by low endpoints, not high endpoints.

Maintaining the information:

- This is easy—$x.max$ depends only on:
  - information in x: $x.int.high$
  - information in $x.left$: $x.left.max$
  - information in $x.right$: $x.right.max$
- Apply the theorem.
- In fact, can update *max* on way down during insertion, and in $O(1)$ time per rotation.

Developing new operations:

```
INTERVAL-SEARCH(T, i)
  x = T.root
  while x ≠ T.nil and i does not overlap x.int
      if x.left ≠ T.nil and x.left.max ≥ i.low
          x = x.left   // overlap in left subtree or no overlap in right subtree
      else x = x.right // no overlap in left subtree
  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

[Stated differently from textbook.]

If search goes right, then either:

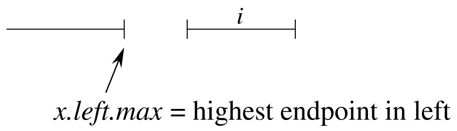
- There is an overlap in right subtree, or
- There is no overlap in either subtree.

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 that there is no overlap in left. Went right because
  - $x.\mathit{left} = T.\mathit{nil} \Rightarrow$ no overlap in left.
    OR
  - $x.\mathit{left}.\mathit{max} < i.\mathit{low} \Rightarrow$ 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 that there is no overlap in right.
  - Went left because:
    $$\begin{aligned} i.\mathit{low} &\le x.\mathit{left}.\mathit{max} \\ &= j.\mathit{high} \text{ for some } j \text{ in left subtree} . \end{aligned}$$
  - Since there is no overlap in left, i and j don't overlap.
  - Refer back to: no overlap if
    $$i.\mathit{low} > j.\mathit{high} \text{ or } j.\mathit{low} > i.\mathit{high} .$$
  - Since $i.\mathit{low} \le j.\mathit{high}$, must have $j.\mathit{low} > i.\mathit{high}$.
  - Now consider *any* interval k in *right* subtree.
  - Because keys are low endpoint,
    $$\underbrace{j.\mathit{low}}_{\text{in left}} \le \underbrace{k.\mathit{low}}_{\text{in right}} .$$
  - Therefore, $i.\mathit{high} < j.\mathit{low} \le k.\mathit{low}$.
  - Therefore, $i.\mathit{high} < k.\mathit{low}$.
  - Therefore, i and k do not overlap. ■ (theorem)

Solutions for Chapter 17: Augmenting Data Structures

Solution to Exercise 17.1-1

In the first call, x points to the node with key 26 and $i = 10$. In the second call, x points to the node with key 17 and $i = 10$. In the third call, x points to the node with key 21 and $i = 2$. In the fourth call, x points to the node with key 19 and $i = 2$. In the fifth call, x points to the node with key 20 and $i = 1$. This node, with key 20, is returned.

Solution to Exercise 17.1-2

Entering the **while** loop, $r = 1$ and y points to the node with key 35. The first iteration of the loop leaves r unchanged and makes y point to the node with key 38. The second iteration sets $r = 3$ and makes y point to the node with key 30. The third iteration makes y point to the node with key 41. The fourth iteration sets $r = 16$ and makes y point to the root, with key 26. The loop terminates and the rank $r = 16$ is returned.

Solution to Exercise 17.1-3

```
OS-SELECT($x, i$)
  $r = x.left.size + 1$
  while $i \neq r$
      if $i < r$
          $x = x.left$
      else $x = x.right$
          $i = i - r$
  return $x$
```

Solution to Exercise 17.1-4

This solution assumes that the TREE-SEARCH procedure from Section 12.2 has been modified to use the sentinel $T.nil$ in a red-black tree.

```
OS-KEY-RANK$(T, k)$
  $x$ = TREE-SEARCH$(T.root, k)$
  if $x == T.nil$
      error "key not found"
  else return OS-RANK$(T, x)$
```

Solution to Exercise 17.1-5

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$(T, x, i)$
  return OS-SELECT$(T.root,$ OS-RANK$(T, x) + i)$
```

Since OS-RANK and OS-SELECT each take $O(\lg n)$ time, so does the procedure OS-SUCCESSOR.

Solution to Exercise 17.1-6

Inserting node z entails a search down the tree for the proper place for z. For each node x on this path, add 1 to $x.rank$ if z is inserted within x's left subtree, and leave $x.rank$ unchanged if z is inserted within x's right subtree.

Deleting node z is a little more involved. Decrement the rank of every proper ancestor a of z for which a path from the root to z includes a's left child. Then, referring to the four cases shown in Figure 12.4, do the following in each case:

a. No other changes.

b. No other changes.

c. Set $y.rank = z.rank$.

d. Decrement the rank of each node on the left-going path from r to y, including r but not y, and set $y.rank = z.rank$.

Similarly when deleting, subtract 1 from $x.rank$ whenever the spliced-out node 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). Leave $x.rank$ unchanged, and letting $r = y.rank$ before the rotation, set $y.rank = r + x.rank$. Right rotations are handled in an analogous manner.

Solution to Exercise 17.1-7
This solution is also posted publicly

Let $A[1:n]$ 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:

$$\text{\# of inversions} = \sum_{j=1}^{n} |\mathit{Inv}(j)| \ ,$$

where $\mathit{Inv}(j) = \{i : i < j \text{ and } A[i] > A[j]\}$.

Note that $|\mathit{Inv}(j)|$ is related to $A[j]$'s rank in the subarray $A[1:j]$ because the elements in $\mathit{Inv}(j)$ 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 $j = r(j) + |\mathit{Inv}(j)|$, so that we can compute

$$|\mathit{Inv}(j)| = j - r(j)$$

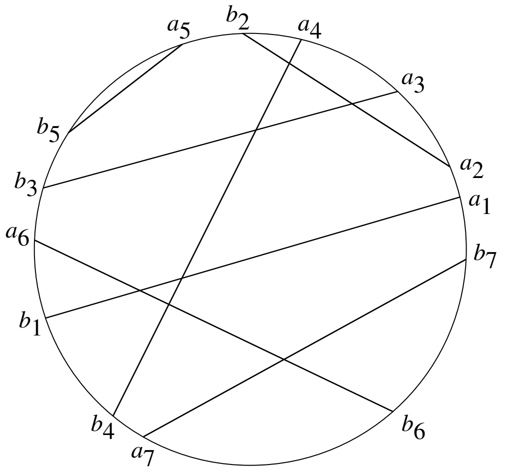
by inserting $A[1], \ldots, A[n]$ into an order-statistic tree and using OS-RANK to find the rank of each $A[j]$ in the tree immediately after it is inserted into the tree. (This OS-RANK value is $r(j)$.)

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 17.1-8

Start by using an $O(n \lg n)$-time algorithm to sort the endpoints of the chords according to the angles at which they intersect the circle, relative to the positive x-axis that starts at the circle's center. Then go around the circle, starting from the rightmost point on the circle until arriving back at the starting point. (Either direction works; assume counterclockwise.) The trip around the circle encounters both endpoints of each chord. Number the chords according to the first time the trip encounters an endpoint of the chord. Call the first endpoint encountered of the ith chord a_i, and call the second endpoint encountered of this chord b_i. Two chords i and j, where $i < j$, intersect if and only if their endpoints are encountered in the order $a_i \, a_j \, b_i \, b_j$.

Here's an example:



For example, chords 4 and 6 intersect because, going counterclockwise starting from the rightmost point on the circle, their endpoints are encountered in the order $a_4\, a_6\, b_4\, b_6$.

To count the intersections, keep an order-statistic tree T whose keys are the numbers of the chords. Also keep track of how many nodes are in T at all times; call this number c. Initialize the number x of intersections to 0. Upon encountering the first endpoint a_i of chord i, insert i into T and increment c. Upon encountering the second endpoint b_i of chord i, determine how many chords with numbers greater than i are currently in T, and add this number into x. If $j > i$ is in T upon encountering b_i, then the endpoints of chords i and j were encountered in the order $a_i\, a_j\, b_i$, with b_j to be encountered at some time in the future. Therefore, chords i and j intersect. The number of chords with numbers greater than i currently in T equals $c - \text{OS-RANK}(T, i)$. Add this number into x, delete i from T, and decrement c. Once all endpoints of all chords have been encountered, the total number of intersections is x.

In our example, here is what happens upon encountering each endpoint b_i, with x starting at 0:

b_i	chords in T	c	OS-RANK(T, i)	new value of x
b_2	1 2 3 4	4	2	2
b_5	1 3 4 5	4	4	2
b_3	1 3 4	3	2	3
b_1	1 4 6	3	1	5
b_4	4 6	2	1	6
b_6	6 7	2	1	7
b_7	7	1	1	7

There are n INSERT operations, n DELETE operations, and n OS-RANK operations. Along with the sorting time, the total time is $O(n \lg n)$.

Solution to Exercise 17.2-1

Keep the nodes in a doubly linked list that is sorted by the keys, and maintain pointers to the nodes with the minmum and maximum values. Each of the queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR then takes $O(1)$ time. Since rotations don't affect the key values in nodes, no changes to rotation code need to occur. Therefore, the procedures RB-INSERT-FIXUP and RB-DELETE-FIXUP don't need to change.

Deleting a node from a doubly linked list takes $O(1)$ time, so that when deleting a node from an order-statistic tree, it takes only $O(1)$ time to update the linked list. When inserting, the new node z becomes the child of a leaf (a non-NIL leaf, that is) before RB-INSERT-FIXUP runs. If z is the left child of $z.p$, then z is the predecessor of $z.p$, and if z is the right child of $z.p$, then z is the successor of $z.p$. Either way, z can be inserted into the doubly linked list in $O(1)$ time.

Solution to Exercise 17.2-2
This solution is also posted publicly

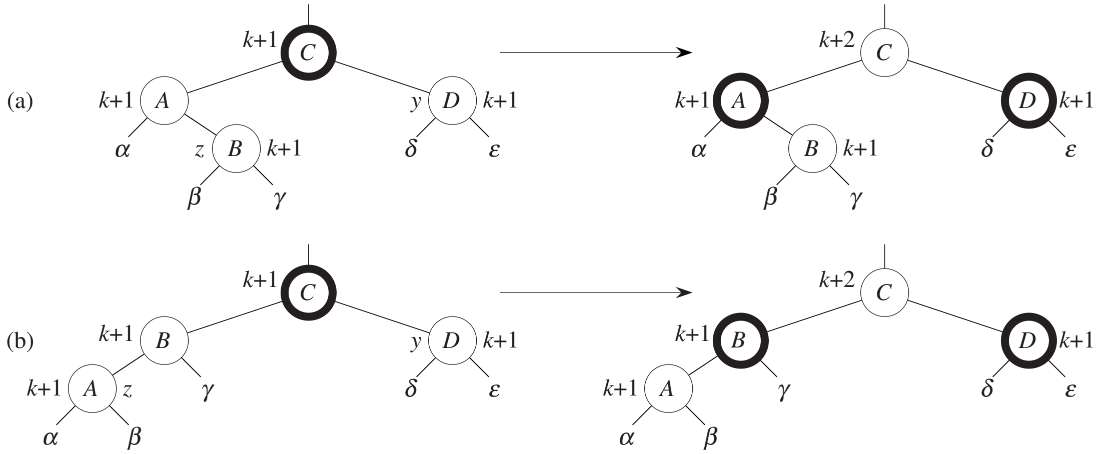
Yes, it is possible to maintain black-heights as attributes in the nodes of a red-black tree without affecting the asymptotic performance of the red-black tree operations. We appeal to Theorem 17.1, because the black-height of a node can be computed from the information at the node and its two children. 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.

The RB-INSERT-FIXUP and RB-DELETE-FIXUP procedures change node colors, and each color change can potentially cause $O(\lg n)$ black-height changes. We'll 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 attribute $x.bh$.

[In the figures below, nodes with a regular outline are red, nodes with a heavy black outline are black, and nodes with a heavy gray outline have an undetermined color.]

For RB-INSERT-FIXUP, there are three cases to examine.

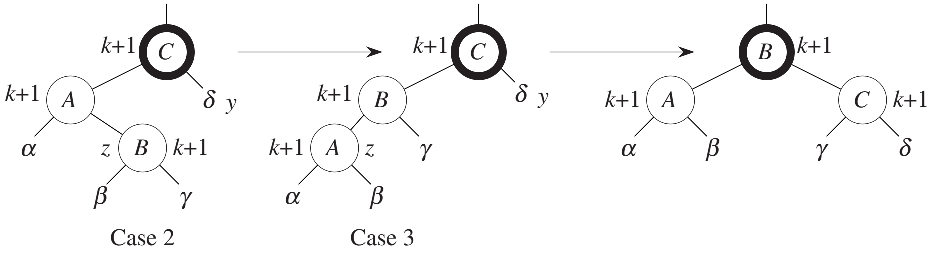
Case 1: z's uncle is red.



- 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 $z.p.p.bh = z.p.p.bh + 1$ after lines 7 and 21 in RB-INSERT-FIXUP.
- Since the number of black nodes between $z.p.p$ and z remains the same, nodes above $z.p.p$ 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.

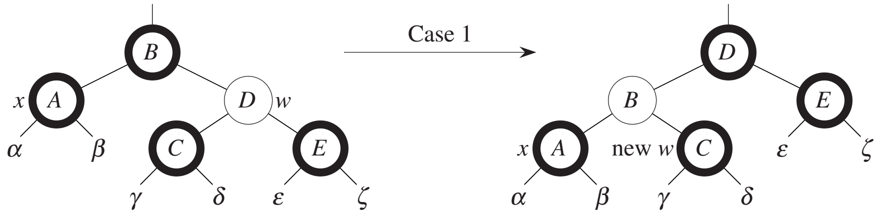


- With subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ of black-height k, even with color changes and rotations, the black-heights of nodes A, B, and C remain the same $(k + 1)$.

Thus, RB-INSERT-FIXUP maintains its original $O(\lg n)$ time.

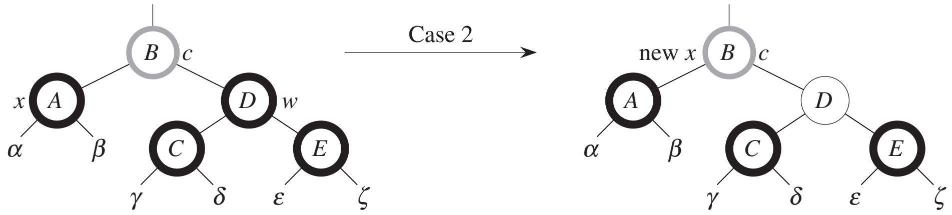
For RB-DELETE-FIXUP, there are four cases to examine.

Case 1: x's sibling w is red.



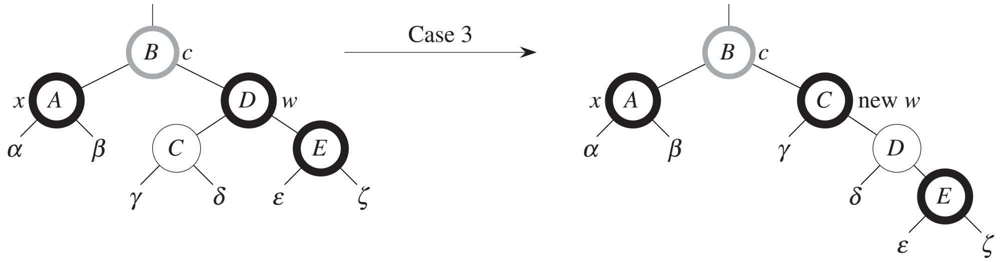
- Even though case 1 changes colors of nodes and does a rotation, black-heights are not changed.
- Case 1 changes the structure of the tree, but waits for cases 2, 3, and 4 to deal with the "extra black" on x.

Case 2: x's sibling w is black, and both of w's children are black.



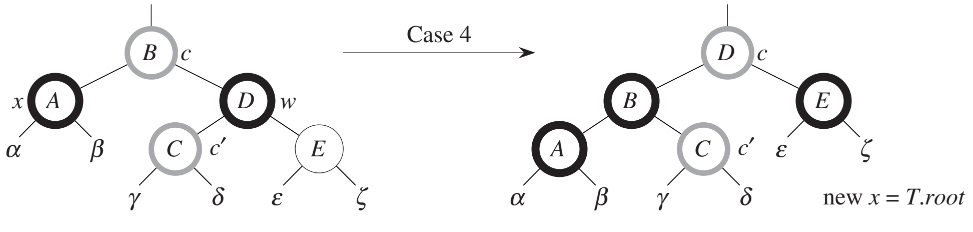
- w is colored red, and x's "extra" black is moved up to $x.p$.
- Add $x.p.bh = x.bh$ after lines 10 and 31 in RB-DELETE-FIXUP.
- This is a constant-time update. Then, keep looping to deal with the extra black on $x.p$.

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 lines 21 and 42:

$$x.p.bh = x.bh + 1$$
$$x.p.p.bh = x.p.bh + 1$$

- The extra black is taken care of, and the 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 attributes in red-black trees without affecting the asymptotic performance of red-black tree operations.

For the second part of the question, no, we cannot maintain node depths without affecting the asymptotic performance of red-black tree operations. 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 17.2-3

We'll show how to update the f attribute after a left rotation on node x; the solution for a right rotation is analogous. Suppose that before the left rotation, x has α as its left child and y as its right child, and y has β as its left child and γ as its right child. After the rotation, the f attributes are as follows: $x.f = \alpha.f \otimes x.a \otimes \beta.f$, and $y.f = x.f \otimes y.a \otimes \gamma.f$. (The order of the assignments is important: $x.f$ must be computed before $y.f$.) These updates take $O(1)$ time.

If we define $x.a$ to be 1, and the $\otimes$ operation to be the regular integer addition, then $x.f$ is the size of the subtree rooted at x, as in the text. This shows how to update the *size* attribute in $O(1)$ time.

Solution to Exercise 17.3-1

To the end of LEFT-ROTATE, add the following two lines:

$$\begin{aligned} x.max &= \max\{x.int.high, x.left.max, x.right.max\} \\ y.max &= \max\{y.int.high, x.max, y.right.max\} \end{aligned}$$

Solution to Exercise 17.3-2

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, the left subtree needs to be checked as well. If there is no overlap in the left subtree but node x overlaps i, then return x. 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 the left subtree might be checked, 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 $T.nil$ if there is no such node.

```
MIN-INTERVAL-SEARCH(T, i)
  return MIN-INTERVAL-SEARCH-FROM(T, T.root, i)
```

```
MIN-INTERVAL-SEARCH-FROM(T, x, i)
  if x.left ≠ T.nil and x.left.max ≥ i.low
      y = MIN-INTERVAL-SEARCH-FROM(T, x.left, i)
      if y ≠ T.nil
          return y
      elseif i overlaps x.int
          return x
      else return T.nil
  elseif i overlaps x.int
      return x
  else return MIN-INTERVAL-SEARCH-FROM(T, x.right, 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 17.3-5

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 attributes in each node x:

   - $x.min\text{-}gap$ 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 $T.nil$), let $x.min\text{-}gap = \infty$.
   - $x.min\text{-}val$ contains the minimum value (key) in the subtree rooted at x.
   - $x.max\text{-}val$ contains the maximum value (key) in the subtree rooted at x.
3. Maintaining the information:
   The three attributes added to the tree can each be computed from information in the node and its children. Hence by Theorem 17.1, they can be maintained during insertion and deletion without affecting the $O(\lg n)$ running time:

$$x.\mathit{min\text{-}val} = \begin{cases} x.\mathit{left}.\mathit{min\text{-}val} & \text{if there is a left subtree ,} \\ x.\mathit{key} & \text{otherwise ,} \end{cases}$$

$$x.\mathit{max\text{-}val} = \begin{cases} x.\mathit{right}.\mathit{max\text{-}val} & \text{if there is a right subtree ,} \\ x.\mathit{key} & \text{otherwise ,} \end{cases}$$

$$x.\mathit{min\text{-}gap} = \min \begin{cases} x.\mathit{left}.\mathit{min\text{-}gap} & (\infty \text{ if no left subtree}) , \\ x.\mathit{right}.\mathit{min\text{-}gap} & (\infty \text{ if no right subtree}) , \\ x.\mathit{key} - x.\mathit{left}.\mathit{max\text{-}val} & (\infty \text{ if no left subtree}) , \\ x.\mathit{right}.\mathit{min\text{-}val} - x.\mathit{key} & (\infty \text{ if no right subtree}) . \end{cases}$$

In fact, the reason for defining the *min-val* and *max-val* attributes 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 $x.\mathit{min\text{-}gap}$ to figure out where $x.\mathit{min\text{-}gap}$ came from. If it came from a subtree's *min-gap* attribute, 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 17.3-6
This solution is also posted publicly

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 (use merge sort or heap sort).
- $O(n \lg n)$ for interval-tree operations (insert, delete, and check for overlap).

Solution to Problem 17-1

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 search tree of the endpoints. That is, to insert an interval, 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 the intervals ($n/2$ intervals, since each interval has two endpoints). 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 $s(1, i)$.

For each node x in the tree, let $l(x)$ and $r(x)$ be the indices in the sorted order of the leftmost and rightmost endpoints, respectively, in the subtree rooted at x. Then the subtree rooted at x contains the endpoints $e_{l(x)}, e_{l(x)+1}, \ldots, e_{r(x)}$.

Each node x stores three new attributes. Store $x.v = s(l(x), r(x))$, the sum of the values of all nodes in the subtree rooted at x. Also store $x.m$, the maximum value obtained by the expression $s(l(x), i)$ for any i in $\{l(x), l(x)+1, \ldots, r(x)\}$. Finally, store $x.o$ as the value of i for which $x.m$ achieves its maximum. For the sentinel, define $T.\mathit{nil}.v = T.\mathit{nil}.m = 0$.

We can compute these attributes in a bottom-up fashion to satisfy the requirements of Theorem 17.1:

$$x.v = x.\mathit{left}.v + p(x) + x.\mathit{right}.v \; ,$$

$$x.m = \max \begin{cases} x.\mathit{left}.m & \text{(max is in } x\text{'s left subtree)} \; , \\ x.\mathit{left}.v + p(x) & \text{(max is at } x) \; , \\ x.\mathit{left}.v + p(x) + x.\mathit{right}.m & \text{(max is in } x\text{'s right subtree)} \; . \end{cases}$$

Computing $x.v$ is straightforward. Computing $x.m$ bears further explanation. Recall that it is the maximum value of the sum of the p values for the nodes in the subtree rooted at x, starting at the node for $e_{l(x)}$, which is the leftmost endpoint in x's subtree, and ending at any node for e_i in x's subtree. The endpoint e_i that maximizes this sum—let's call it e_{i^*}—corresponds to either a node in x's left subtree, x itself, or a node in x's right subtree. If e_{i^*} corresponds to a node in x's left subtree, then $x.\mathit{left}.m$ represents a sum starting at the node for $e_{l(x)}$ and ending at a node in x's left subtree, and hence $x.m = x.\mathit{left}.m$. If e_{i^*} corresponds to x itself, then $x.m$ represents the sum of all p values in x's left subtree, plus $p(x)$, so that $x.m = x.\mathit{left}.v + p(x)$. Finally, if e_{i^*} corresponds to a node in x's right subtree, then $x.m$ represents the sum of all p values in x's left subtree, plus $p(x)$, plus the sum of some subset of p values in x's right subtree. Moreover, the values taken from x's right subtree must start from the leftmost endpoint stored in the right subtree. To maximize this sum, we need to maximize the sum from the right subtree, and that value is precisely $x.\mathit{right}.m$. Hence, in this case, $x.m = x.\mathit{left}.v + p(x) + x.\mathit{right}.m$.

Once we understand how to compute $x.m$, it is straightforward to compute $x.o$ 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 $T.\mathit{root}.o$.

Because of how we have defined the new attributes, Theorem 17.1 says that each operation runs in $O(\lg n)$ time. In fact, FIND-POM takes only $O(1)$ time.

Solution to Problem 17-2

a. Use a circular linked list in which each element has two attributes, *key* and *next*. At the beginning, 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* attributes). Make the list circular by letting the *next* attribute of the last element point to the first element.

Then, start scanning the list from the beginning. Output and then delete every mth element, until the list becomes empty. (The scan needs to keep track of the current element and its predecessor so that the *next* attribute of the predecessor can be updated to skip over the deleted element. Alternatively, the circular linked list could be doubly linked.) 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 17.1. 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 $k \leq n$ people remaining. Then we 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
      create a node x with x.key = j
      OS-INSERT(T, x)
  k = n
  j = m
  while k > 2
      x = OS-SELECT(T.root, j)
      print x.key
      OS-DELETE(T, x)
      k = k - 1
      j = ((j + m - 2) mod k) + 1
  print OS-SELECT(T.root, 1).key
```

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
      create a node x with x.key = j
      OS-INSERT(T, x)
  j = 1
  for k = n downto 1
      j = ((j + m - 2) mod k) + 1
      x = OS-SELECT(T.root, j)
      print x.key
      OS-DELETE(T, x)
```

Either way, it takes $O(n \lg n)$ time to build up the order-statistic tree T, followed by $O(n)$ 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 19: Data Structures for Disjoint Sets

Chapter 19 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 you ask for the representative twice without modifying the set, you 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 19.4.]

Operations

- $\text{M}\textsc{ake}\text{-S}\textsc{et}(x)$: make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$.
- $\text{U}\textsc{nion}(x, y)$: if $x \in S_x$, $y \in S_y$, then $\mathcal{S} = \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$.
  - 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).
- $\text{F}\textsc{ind}\text{-S}\textsc{et}(x)$: return representative of set containing x.

Analysis in terms of:

- n = # of elements = # of $\text{M}\textsc{ake}\text{-S}\textsc{et}$ 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(G)
  for each vertex $v \in G.V$
      MAKE-SET($v$)
  for each edge $(u, v) \in G.E$
      if FIND-SET($u$) $\neq$ FIND-SET($v$)
          UNION($u, v$)
```

```
SAME-COMPONENT($u, v$)
  if FIND-SET($u$) == FIND-SET($v$)
      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, represented by an object with attributes
  - *head*: the first element in the list, assumed to be the set's representative, and
  - *tail*: the last element in the list.

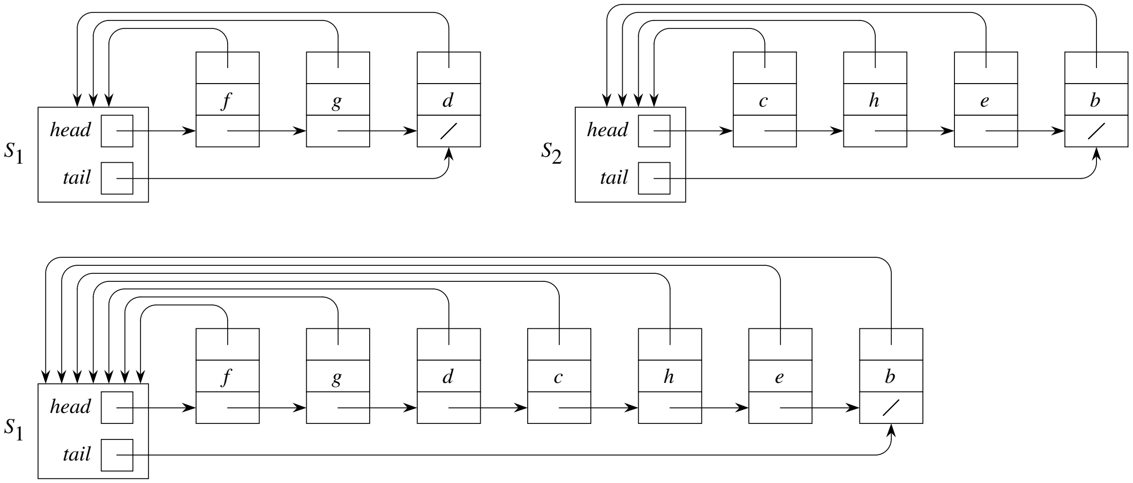
  Objects may appear within the list in any order.
- Each object in the list has attributes for
  - the set member,
  - pointer to the set object, and
  - next.

MAKE-SET: create a singleton list.

FIND-SET: follow the pointer back to the list object, and then follow the *head* pointer to the representative.

UNION: a couple of ways to do it.

1. UNION(x, y): append y's list onto end of x's list. Use x's tail pointer to find the end.



[The top part of the figure shows two sets before the operation UNION(S_1, S_2)*. The bottom part shows the result of the* UNION *operation.]*

- Need to update the pointer back to the set object for every node on y's list.
- If appending a large list onto a small list, it can take a while.

Operation	# objects updated
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
UNION(x_5, x_4)	4
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$
	$\Theta(n^2)$ total

Amortized time per operation $= \Theta(n)$.

2. ***Weighted-union heuristic:*** Always append the smaller list to the larger list. (Break ties arbitrarily.)

   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	$\geq 2^k$
$\vdots$	$\vdots$
$\lg n$	$\geq n$

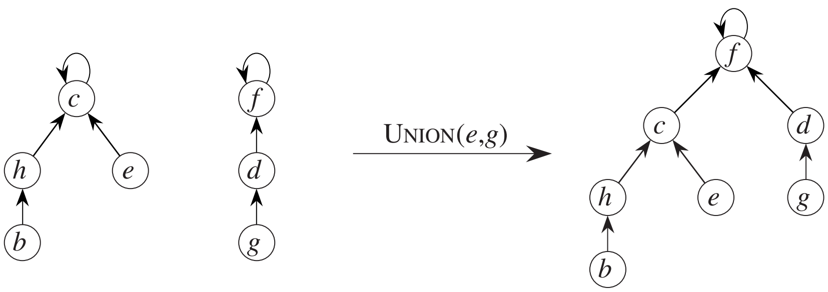
Therefore, each representative is updated $\leq \lg n$ times. ■ (theorem)

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.

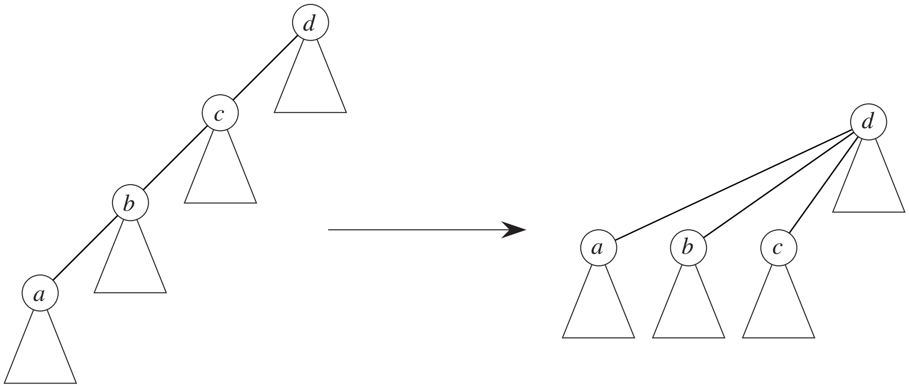
Not so good—could get a linear chain of nodes.

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.



Each node has two attributes, p (parent) and *rank*.

```
MAKE-SET(x)
  x.p = x
  x.rank = 0
```

```
UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)
  if x.rank > y.rank
      y.p = x
  else x.p = y
      // If equal ranks, choose y as parent and increment its rank.
      if x.rank == y.rank
          y.rank = y.rank + 1
```

```
FIND-SET(x)
  if x ≠ x.p                    // not the root?
      x.p = FIND-SET(x.p)       // the root becomes the parent
  return x.p                    // return the root
```

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, $O(m\,\alpha(n))$.

$\alpha(n)$ is a *very* slowly growing function:

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 19.4, if you dare. It's $\gg 10^{80} \approx$ # of atoms in observable universe.

This bound is tight—there exists a sequence of operations that takes $\Omega(m\ \alpha(n))$ time.

Solutions for Chapter 19: Data Structures for Disjoint Sets

Solution to Exercise 19.1-2

Denote the number of edges by m (so that $m = |E|$). For $k = 0, 1, 2, \ldots, m$, define E_k to be the set of edges considered in iterations 1 through k of the **for** loop of lines 3–5 of CONNECTED-COMPONENTS, where $E_0 = \emptyset$, and define $G_k = (V, E_k)$. We use the following loop invariant:

> **Loop invariant:**
>
> After k iterations of the **for** loop of lines 3–5, vertices x and y are in the same connected component of G_k if and only if they are in the same set.

Initialization: Initially, G_0 has no edges so that each vertex is its own connected component, and each vertex is its own singleton set.

Maintenance: Before iteration k of the **for** loop, vertices x and y are in the same connected component of G_{k-1} if and only if they are in the same set. Because creating G_k by adding an edge to G_{k-1} cannot disconnect vertices, if x and y are in the same connected component of G_{k-1}, they are in the same connected component of G_k and remain in the same set.

Now suppose that x and y are not in the same connected component of G_{k-1}. By the loop invariant, they are not in the same set after $k - 1$ iterations. In iteration k, if u is in the connected component of x and v is in the connected component of y, or vice-versa, then then there is a path in G_k going $x \leadsto u \to v \leadsto y$, so that x and y are in the same connected component of G_k, and line 5 puts x and y into the same set. If u is not in the connected component of x or v is not in the connected component of y, or vice-versa, then adding edge (u, v) into G_{k-1} either does not change the connected components (if u and v are already within the same connected component of G_{k-1}), or at least one of the components that (u, v) connects is neither x's component nor y's component. Therefore, x and y remain in separate connected components of G_k, and the UNION operation in line 5 does not unite the sets containing x and y.

Termination: The **for** loop terminates, since it iterates m times. At termination, $E_m = E$, so that x and y are in the same connected component of $G_m = G$ if and only if they are in the same set.

Solution to Exercise 19.1-3

FIND-SET is called twice for each edge, or $2|E|$ times. This count assumes that UNION does not also call FIND-SET (as it does in the disjoint-set forest implementation). Each call of UNION reduces the number of connected components, and sets, by 1. The procedure starts with $|V|$ connected components and ends with k connected components, so that UNION is called $|V| - k$ times.

Solution to Exercise 19.2-1

Assume that a set object has attributes *head* and *tail*, pointing to the first and last elements in the list, and *size*, giving the size of the set, and that an element object has attributes *set*, pointing to its set object, and *next*, pointing to the next element in the list.

```
MAKE-SET($x$)
  create a set object $S$ and an element object that $x$ points to
  $x.set = S$
  $x.next =$ NIL
  $S.head = x$
  $S.tail = x$
  $S.size = 1$
```

```
FIND-SET($x$)
  return $x.set.head$
```

```
UNION($x, y$)
  $S_x = x.set$
  $S_y = y.set$
  if $S_y.size > S_x.size$
      exchange $S_x$ with $S_y$
  $z = S_y.head$
  while $z \neq$ NIL
      $z.set = S_x$
      $z = z.next$
  $S_x.tail.next = S_y.head$
  $S_x.tail = S_y.tail$
  $S_x.size = S_x.size + S_y.size$
```

Solution to Exercise 19.2-2

The resulting data structure is one linked list with members, in order, 1 through 16, so that *head* points to the node with member 1 and *tail* points to the node with member 16. The representative is 1, and so both calls to FIND-SET return 1.

Solution to Exercise 19.2-3

This solution is also posted publicly

We want to show how to 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 $u < n$ be the number of UNION operations. (Recall the discussion in Section 19.1 about there being at most $n - 1$ UNION operations.) Then there are n MAKE-SET operations, u UNION operations, and $m - n - u$ FIND-SET operations.

The theorem didn't separately name the number u of UNION operations; rather, it bounded the number by n. If you go through the proof of the theorem with u UNION operations, you get the time bound $O(m - u + u \lg u) = O(m + u \lg u)$ for the sequence of operations. That is, the actual time taken by the sequence of operations is at most $c(m + u \lg u)$, for some constant c.

Thus, we want to assign operation charges such that

$$\begin{array}{ll} & (\text{MAKE-SET charge}) \cdot n \\ + & (\text{FIND-SET charge}) \cdot (m - n - u) \\ + & (\text{UNION charge}) \cdot u \\ \hline \geq & c(m + u \lg u) , \end{array}$$

so that the amortized costs give an upper bound on the actual costs.

The following assignments work, where $c' \geq c$ is some constant:

- MAKE-SET: c'
- FIND-SET: c'
- UNION: $c'(\lg n + 1)$

Substituting into the above sum gives

$$\begin{aligned} c'n + c'(m - n - u) + c'(\lg n + 1)u &= c'm + c'u \lg n \\ &= c'(m + u \lg n) \\ &> c(m + u \lg u) . \end{aligned}$$

Solution to Exercise 19.2-4

The first argument in each of the UNION operations in Figure 19.3 is a singleton set, so that each of the $n - 1$ UNION operations takes constant time. Since each of the n MAKE-SET operations also takes constant time, the total running time of the entire sequence is $\Theta(n)$.

Solution to Exercise 19.2-5

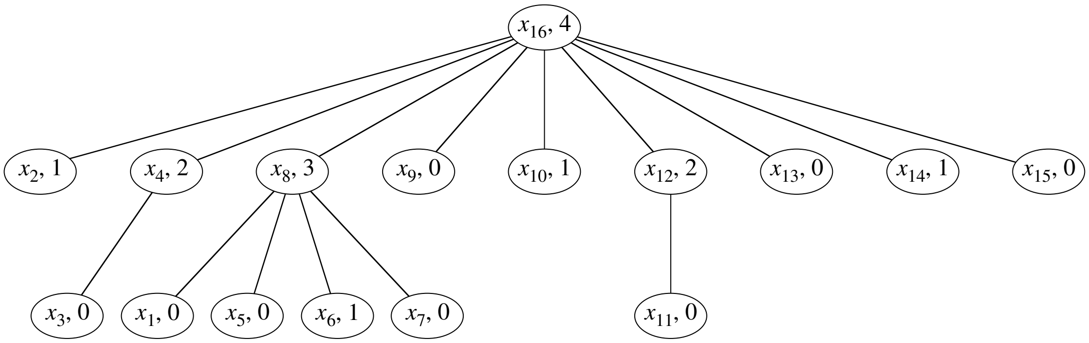
As the hint suggests, make the representative of each set be the tail of its linked list. Except for the tail element, each element's representative pointer points to the tail. The tail's representative pointer points to the head. An element is the tail if its next pointer is NIL. Now we can get to the tail in $O(1)$ time: if $x.\mathit{next}$ == NIL, then $\mathit{tail} = x$, else $\mathit{tail} = x.\mathit{rep}$. We can get to the head in $O(1)$ time as well: if $x.\mathit{next}$ == NIL, then $\mathit{head} = x.\mathit{rep}$, else $\mathit{head} = x.\mathit{rep}.\mathit{rep}$. The set object needs only to store a pointer to the tail, though a pointer to any list element would suffice.

Solution to Exercise 19.2-6
This solution is also posted publicly

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 pointers to the set object anyway, so we can just make the last element of B point to the second element of A.

Solution to Exercise 19.3-1

Each node shows the name of the set element and its rank.



Solution to Exercise 19.3-2

```
FIND-SET($x$)
  $y = x$
  **while** $y \neq y.p$
      $y = y.p$
  **while** $x \neq x.p$
      $z = x.p$
      $x.p = y$        // $y$ is the root
      $x = z$
  **return** $y$
```

Solution to Exercise 19.3-3

You need to find a sequence of m operations on n elements that takes $\Omega(m \lg n)$ time. Let n be a power of 2. Start with n MAKE-SET operations 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 $\lg n$.

UNION(x_1, x_2)	$n/2$ of these
UNION(x_3, x_4)	
UNION(x_5, x_6)	
$\vdots$	
UNION(x_{n-1}, x_n)	
UNION(x_2, x_4)	$n/4$ of these
UNION(x_6, x_8)	
UNION(x_{10}, x_{12})	
$\vdots$	
UNION(x_{n-2}, x_n)	
UNION(x_4, x_8)	$n/8$ of these
UNION(x_{12}, x_{16})	
UNION(x_{20}, x_{24})	
$\vdots$	
UNION(x_{n-4}, x_n)	
$\vdots$	
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 $m \geq 3n$, the sequence contains more than $m/3$ FIND-SET operations, so that the total cost is $\Omega(m \lg n)$.

Solution to Exercise 19.3-4

Maintain a circular, singly linked list of the nodes of each set. To print, just follow the list until you get back to node x, printing each member of the list. The only other operations that change are MAKE-SET, which sets $x.next = x$, and LINK, which exchanges the pointers $x.next$ and $y.next$.

Solution to Exercise 19.3-5

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 19.4-1

As the text suggests, the proof is an induction on the number of MAKE-SET, UNION, and FIND-SET operations.

The basis is when no operations have been performed, and the lemma trivially holds. The inductive hypothesis is that the lemma holds after the first $k - 1$ operations. We will show that it continues to hold after the kth operation. There are three cases, one for each procedure.

- If the kth operation is MAKE-SET, then the only change is creating a singleton set for x, where $x.p = x$ and an initial value of $x.rank = 0$, so that $x.rank = x.p.rank$.
- If the kth operation is FIND-SET, no ranks change, but each node on the find path becomes a child of the root. Before path compression, the inductive hypothesis says that ranks strictly increase along the find path, heading up toward the root, so that if x is a node along the find path to root y, we have $x.rank < y.rank$ before path compression. After path compression, $x.p = y$ for all nodes x along the find path, so that the inductive hypothesis continues to hold.
- If the kth operation is UNION, it first entails two calls to FIND-SET, and we have already established that the inductive hypothesis holds after these calls. Now suppose that the call to LINK makes node x a child of node y, so that x and y are both roots before the call to LINK and y remains a root afterward. It must have been the case that $x.rank \le y.rank$ before the LINK. If $x.rank < y.rank$ before the LINK, no ranks change. If $x.rank = y.rank$ before the LINK, then $y.rank$ increases by 1, and that is the only way that a rank can increase. In either case, because $x \ne x.p$ after the LINK and only roots can have their rank increase, $x.rank$ will never change after the LINK. Thus, the inductive hypothesis continues to hold.

Solution to Exercise 19.4-2

Define $x.size$ to be the number of nodes in the tree rooted at node x, including x itself.

Claim

For all tree roots x, we have $x.size \ge 2^{x.rank}$.

Proof of claim The proof is by induction on the number of LINK operations, since FIND-SET operations do not change ranks or tree sizes. The basis is before the first LINK, when $x.size = 1$ and $x.rank = 0$ for each node x.

Now assume that the claim holds before performing the operation LINK(x, y). Let *rank* and *rank*′ denote ranks before and after the LINK, respectively. Define *size* and *size*′ similarly.

If $x.rank \ne y.rank$, assume without loss of generality that $x.rank < y.rank$. Because y is the root of the tree formed by the LINK, we have

$$\begin{aligned} y.size' &= x.size + y.size \\ &\ge 2^{x.rank} + 2^{y.rank} \\ &\ge 2^{y.rank} \\ &= 2^{y.rank'} . \end{aligned}$$

No ranks or sizes change for any nodes other than y.

If $x.rank = y.rank$, then again y becomes the root of the new tree, and

$$y.size' = x.size + y.size$$

$$
\begin{aligned}
&\geq 2^{x.\mathit{rank}} + 2^{y.\mathit{rank}} \\
&= 2^{y.\mathit{rank}+1} \\
&= 2^{y.\mathit{rank}'} .
\end{aligned}
$$

■ (claim)

Claim

For any integer $r \geq 0$, there are at most $n/2^r$ nodes of rank r.

Proof of claim Fix a particular value of r. Suppose that upon assigning a rank r to a node x in either MAKE-SET or LINK, each node in the tree rooted at x gets the label x attached to it. By the previous claim, at least 2^r nodes are labeled each time. Suppose that the root of the tree containing node x changes. Lemma 19.4 assures us that the rank of the new root (or of any proper ancestor of x) is at least $r + 1$. Since labels are assigned only when a root is assigned rank r (the fixed value), no node is this new tree will ever again be labeled. Thus, each node is labeled at most once, when its root is first assigned rank r. Since there are n nodes altogether, there are at most n labeled nodes, with at least 2^r labels assigned for each node of rank r. If there were more than $n/2^r$ nodes of rank r, then more than $2^r \cdot (n/2^r) = n$ nodes would be labeled by a node of rank r, a contradiction. Therefore, at most $n/2^r$ nodes are ever assigned rank r. ■ (claim)

Claim

Every node has rank at most $\lfloor \lg n \rfloor$.

Proof of claim Let $r > \lg n$. Then by the previous claim, there are at most $n/2^r < 1$ nodes of rank r. Since ranks are natural numbers, the claim follows. ■ (claim)

Solution to Exercise 19.4-3

Since every rank is at most $\lfloor \lg n \rfloor$, the number of bits needed to store a rank is $\lfloor \lg \lfloor \lg n \rfloor \rfloor + 1$, which is the same as $\lfloor \lg \lg n \rfloor + 1$.

Solution to Exercise 19.4-4

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 19.7 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 19.4-5

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 $A_1(1) = 3$, $\text{level}(x) = 1$, and since $A_0(3) = 4$, $\text{level}(y) = 0$. Yet y follows x on the find path.

Solution to Exercise 19.4-6

Make the following changes:

- Change the definition (19.7) of the potential function to

$$\phi_q(x) = \begin{cases} c\,\alpha(n) \cdot x.\mathit{rank} & \text{if } x \text{ is a root or } x.\mathit{rank} = 0\ , \\ c((\alpha(n) - \text{level}(x)) \cdot x.\mathit{rank} - \text{iter}(x)) & \\ & \text{if } x \text{ is not a root and } x.\mathit{rank} \geq 1\ . \end{cases}$$

- Change the inequality in Lemma 19.8 to $0 \leq \phi_q(x) \leq c\,\alpha(n) \cdot x.\mathit{rank}$. In the proof of the lemma, if x is a root or $x.\mathit{rank} = 0$, then $\phi_q(x) = c\,\alpha(n) \cdot x.\mathit{rank}$. In the other case, multiply the right-hand side of each displayed equation by c.
- Change the inequality in Corollary 19.9 to $\phi_q(x) < c\,\alpha(n) \cdot x.\mathit{rank}$.
- In the proof of Lemma 19.10, change the inequality $\phi_q(x) \leq \phi_{q-1}(x) - 1$, which appears twice, to $\phi_q(x) \leq \phi_{q-1}(x) - c$.
- In the proof of Lemma 19.12, wherever $\alpha(n)$ appears, change it to $c\alpha(n)$, except for the expression $O(\alpha(n))$.
- In the proof of Lemma 19.13:
  - In the first, third, and fourth paragraphs, where there is mention of a potential decrease of at least 1, change that to a potential decrease of at least c.
  - In the second paragraph, if x is a root, change its potential to $c\,\alpha(n) \cdot x.\mathit{rank}$.
  - Change the last sentence of the next-to-last paragraph to read

    > By Lemma 19.10, $\phi_q(x) \leq \phi_{q-1}(x) - c$, so that x's potential decreases by at least c.

  - Change the last paragraph to read

    > The amortized cost of the FIND-SET operation is the actual cost plus the change in potential. The actual cost is $O(s)$, and we have shown that the total potential decreases by at least $\max\{0, c(s - (\alpha(n) + 2))\}$.

The amortized cost, therefore, is at most $O(s) - c(s - (\alpha(n) + 2)) = O(s) - cs + O(\alpha(n)) = O(\alpha(n))$, since we can scale up the constant c to dominate the constant hidden in $O(s)$.

Solution to Exercise 19.4-7

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 $x.rank \geq 1$. With this definition of $\alpha'(n)$, we have $A_{\alpha'(n)}(x.rank) \geq A_{\alpha'(n)}(1) \geq \lg(n + 1) > \lg n \geq x.p.rank$. The rest of the proof goes through with $\alpha'(n)$ replacing $\alpha(n)$.

Solution to Problem 19-1

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 OFFLINE-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 OFFLINE-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 $\mathrm{I}_j \cup \mathrm{I}_{j+1}$. 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, 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 OFFLINE-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 $j = \text{FIND-SET}(i)$.
- Line 5 ("let l be the smallest value greater than j for which set K_l exists") turns into $K_l = K_j.\mathit{next}$.
- Line 6 ("$K_l = K_j \cup K_l$, destroying K_j") turns into $l = \text{LINK}(j, l)$ and remove K_j from the linked list.

To analyze the running time, 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 $O(m\,\alpha(n)) = O(n\,\alpha(n))$.

Solution to Problem 19-2

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), $\ldots$, 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)
  $v.p = v$
  $v.rank = 0$
  $v.d = 0$
```

It is correct to set $v.d$ 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 $v.d$.

c. FIND-DEPTH will call a procedure FIND-ROOT:

```
FIND-ROOT(v)
  if $v.p \neq v.p.p$
      $y = v.p$
      $v.p =$ FIND-ROOT$(y)$
      $v.d = v.d + y.d$
  return $v.p$
```

```
FIND-DEPTH(v)
  FIND-ROOT(v)                // no need to save the return value
  if $v == v.p$
      return $v.d$
  else return $v.d + v.p.d$
```

FIND-ROOT performs path compression and updates pseudodistances along the find path from v. It is similar to FIND-SET on page 530, 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 $v.p = v.p.p$) in the disjoint-set forest, the procedure does not have to recurse; instead, it just returns $v.p$. Second, when it recurses, it saves the pointer $v.p$ into a new variable y. Third, when it recurses, it updates $v.d$ 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, $y.d$ is added into $v.d$. Note that $y.d$ has been updated prior to updating $v.d$, 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 $v.p$ (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 $v.d$, and in the latter case, the depth is $v.d + v.p.d$.

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 $r'.rank > v'.rank$
      $v'.p = r'$
      $r'.d = r'.d + z + 1$
      $v'.d = v'.d - r'.d$
  else $r'.p = v'$
      $r'.d = r'.d + z + 1 - v'.d$
      if $r'.rank == v'.rank$
          $v'.rank = v'.rank + 1$
```

This procedure works as follows. First, it calls 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. The procedure then calls FIND-DEPTH(v), saving the depth of v in the variable z. (Since v's find path has just been compressed, this call of FIND-DEPTH takes $O(1)$ time.) Next, 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, 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 $r'.d$ 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, the computed depth of all nodes in the disjoint-set tree rooted at v' has just increased by $r'.d$. These computed depths should not have changed, however. Thus, the procedure subtracts off $r'.d$ from $v'.d$, so that the sum $v'.d + r'.d$ after making v' a child of r' equals $v'.d$ 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, leaving $v'.d$ alone. The procedure needs to update $r'.d$, however, so that after making r' a child of v', the depth of each node in r's disjoint-set tree is increased by $z + 1$. The procedure adds $z + 1$ to $r'.d$, but it also subtracts out $v'.d$, since it has just made r' a child of v'. Finally, if the ranks of r' and v' are equal, the rank of v' is incremented, 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 20: Elementary Graph Algorithms

Graph representation

Given graph $G = (V, E)$. In pseudocode, represent vertex set by $G.V$ and edge set by $G.E$.

- G may be either directed or undirected.
- Two common ways to represent graphs 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: $O(V + E)$ really means $O(|V| + |E|)$.

[The introduction to Part VI talks more about this.]

Adjacency lists

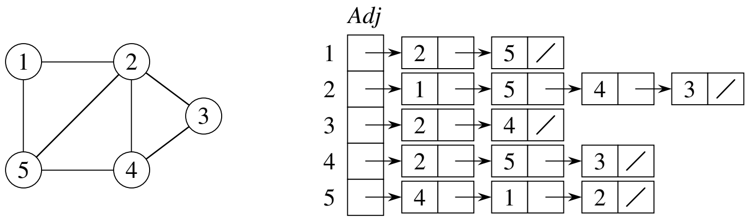
Array *Adj* of $|V|$ lists, one per vertex.

Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

In pseudocode, denote the array as attribute $G.Adj$, so will see notation such as $G.Adj[u]$.

Example

For an undirected graph:



If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

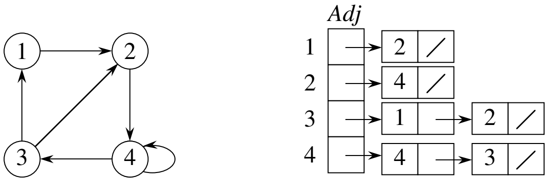
Space: $\Theta(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

Example

For a directed graph:



Same asymptotic space and time.

Adjacency matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array} \qquad \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine whether $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

We'll use both representations in these lecture notes.

Representing graph attributes

Graph algorithms usually need to maintain attributes for vertices and/or edges. Use the usual dot-notation: denote attribute d of vertex v by $v.d$.

Use the dot-notation for edges, too: denote attribute f of edge (u, v) by $(u, v).f$.

Implementing graph attributes

No one best way to implement. Depends on the programming language, the algorithm, and how the rest of the program interacts with the graph.

If representing the graph with adjacency lists, can represent vertex attributes in additional arrays that parallel the *Adj* array, e.g., $d[1:|V|]$, so that if vertices adjacent to u are in $Adj[u]$, store $u.d$ in array entry $d[u]$.

But can represent attributes in other ways. Example: represent vertex attributes as instance variables within a subclass of a `Vertex` class.

Breadth-first search

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.

Output:

- $v.d$ = distance (smallest # of edges) from s to v, for all $v \in V$.
- $v.\pi$ is v's ***predecessor*** on a shortest path (smallest # of edges) from s.
  (u, v) is last edge on shortest path $s \rightsquigarrow v$.
  Predecessor subgraph contains edges (u, v) such that $v.\pi = u$.
  The predecessor subgraph forms a tree, called the ***breadth-first tree***.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple.

[Omitting colors of vertices. Used in book to reason about the algorithm.]

Intuition

Breadth-first search expands the frontier between discovered and undiscovered vertices uniformly across the breath of the frontier.

Discovers vertices in waves, starting from s.

- First visits all vertices 1 edge from s.
- From there, visits all vertices 2 edges from s.
- Etc.

Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has visited v but has not come out of v yet.
- Q contains vertices at a distance k, and possibly some vertices at a distance $k + 1$. Therefore, at any time Q contains portions of two consecutive waves.

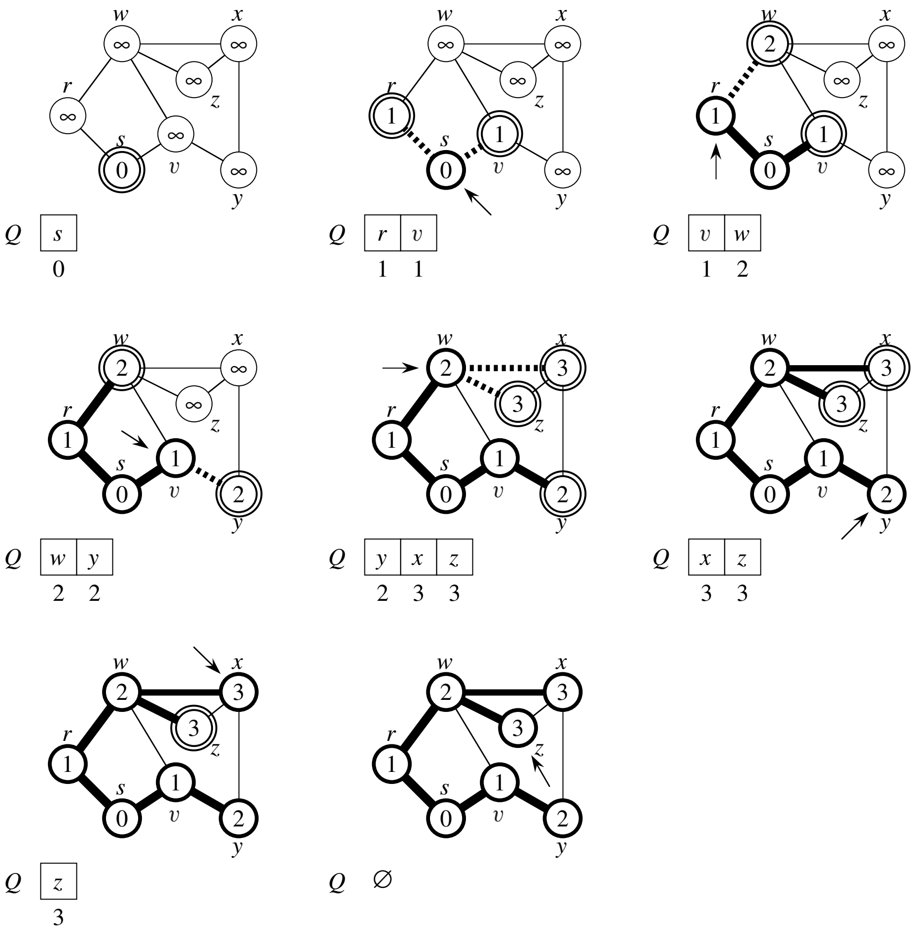
```
BFS(V, E, s)
  for each vertex $u \in V - \{s\}$
      $u.d = \infty$
      $u.\pi = $ NIL
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE(Q, s)
  while $Q \neq \emptyset$
      $u = $ DEQUEUE(Q)
      for each vertex v in G.Adj[u]   // search the neighbors of u
          if $v.d == \infty$           // is v being discovered now?
              $v.d = u.d + 1$
              $v.\pi = u$
              ENQUEUE(Q, v)           // v is now on the frontier
      // u is now behind the frontier.
```

[In the book, the test for whether v is being newly discovered uses the colors. Checking whether $v.d$ is finite or infinite works just as well, since once v is discovered it gets a finite d value. Can also check for whether $v.\pi$ equals NIL*.]*

Example

BFS on an undirected graph: *[There is a more detailed, colorized example in book. Go through this example, showing how vertices are discovered and Q is updated]* .

- Arrows point to the vertex being visited.
- Edges drawn with heavy lines are in the predecessor subgraph.
- Dashed lines go to newly discovered vertices. They are drawn with heavy lines because they are also now in the predecessor subgraph.
- Double-outline vertices have been discovered and are in Q, waiting to be visited.
- Heavy-outline vertices have been discovered, dequeued from Q, and visited.



Can show that Q consists of vertices with d values.

$k \quad k \quad k \quad \ldots \quad k \quad k+1 \quad k+1 \quad \ldots \quad k+1$

- 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 edge (u, v) is examined only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

To print the vertices on a shortest path from s to v:

```
PRINT-PATH($G, s, v$)
  if $v == s$
      print $s$
  elseif $v.\pi$ == NIL
      print "no path from" $s$ "to" $v$ "exists"
  else PRINT-PATH($G, s, v.\pi$)
      print $v$
```

Depth-first search

Input: $G = (V, E)$, directed or undirected. No source vertex given.

Output:

- 2 ***timestamps*** on each vertex:
  - $v.d =$ ***discovery time***
  - $v.f =$ ***finish time***

  These will be useful for other algorithms later on.
- $v.\pi$ is v's predecessor in the ***depth-first forest*** of ≥ 1 ***depth-first trees***. If $u = v.\pi$, then (u, v) is a ***tree edge***.

Methodically explores *every* edge.

- Start over from different vertices as necessary.

As soon as a vertex is discovered, explore from it.

- Unlike BFS, which puts a vertex on a queue so that it's explored from 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, $v.d < v.f$.

In other words, $1 \leq v.d < v.f \leq 2\,|V|$.

Pseudocode

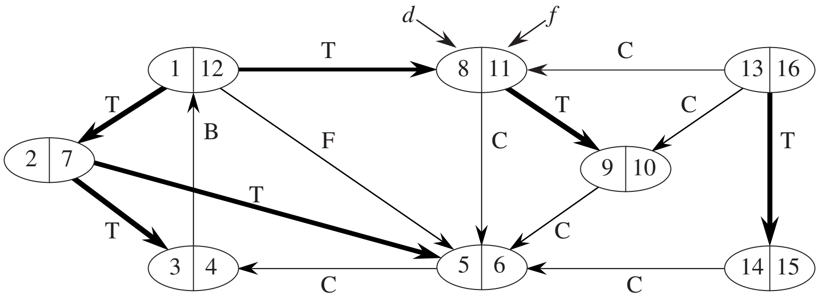
Uses a global timestamp *time*.

```
DFS($G$)
  for each vertex $u \in G.V$
      $u.color$ = WHITE
      $u.\pi$ = NIL
  $time = 0$
  for each vertex $u \in G.V$
      if $u.color$ == WHITE
          DFS-VISIT($G, u$)
```

```
DFS-VISIT$(G, u)$
  $time = time + 1$                  // white vertex $u$ has just been discovered
  $u.d = time$
  $u.color =$ GRAY
  for each vertex $v$ in $G.Adj[u]$  // explore each edge $(u, v)$
      if $v.color ==$ WHITE
          $v.\pi = u$
          DFS-VISIT$(G, v)$
  $time = time + 1$
  $u.f = time$
  $u.color =$ BLACK                  // blacken $u$; it is finished
```

Example

[Go through this example of DFS on a directed graph, adding in the d and f values as they're computed. Show colors as they change. Don't put in the edge types yet, except that the tree edges are drawn with heavy lines.]



Time $= \Theta(V + E)$.

- Similar to BFS analysis.
- Θ, not just O, since guaranteed to examine every vertex and edge.

Each depth-first 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. $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ (i.e., the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint) and neither of u and v is a descendant of the other.
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u. (v is discovered after and finished before u.)
3. $v.d < u.d < u.f < v.f$ and u is a descendant of v. (u is discovered after and finished before v.)

So $u.d < v.d < u.f < v.f$ (v is both discovered and finished after u) *cannot* happen.

Like parentheses:

- OK: ( ) [ ] ([ ]) [( )]
- Not OK: ([ )] [( ])

Corollary

v is a proper descendant of u if and only if $u.d < v.d < v.f < u.f$.

Theorem (White-path theorem)

[Proof omitted.]

v is a descendant of u if and only if at time $u.d$, there is a path $u \leadsto v$ 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.]

A DFS of an *undirected* graph yields only tree and back edges. No forward or cross edges.

Topological sort

Directed acyclic graph (dag)

A directed graph with no cycles.

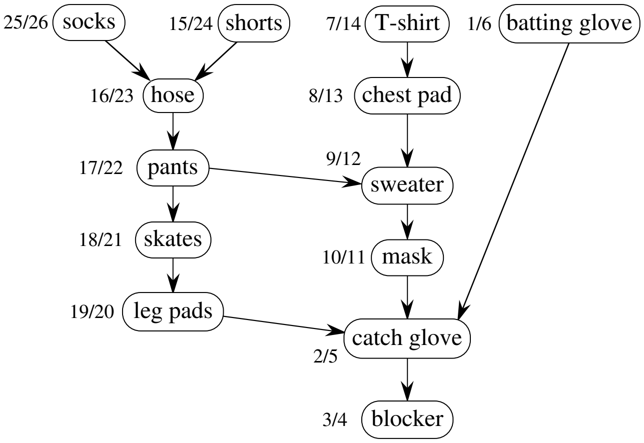
Good for modeling processes and structures that have a ***partial order:***

- $a > b$ 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 for ice hockey: *[Leave on board, but show without discovery and finish times. Will put them in later.]*

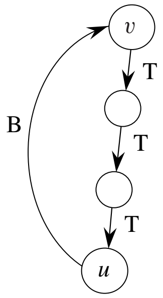


Lemma

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 $v \rightsquigarrow u \to v$ 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 $v.d$, 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($G$)
  call DFS($G$) to compute finish times $v.f$ for all $v \in G.V$
  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 $v.f < u.f$.
When edge (u, v) is explored, 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, $u.d < v.d < \underline{v.f < u.f}$.
- Is v black?
  - Then v is already finished.
    Since exploring (u, v), u is not yet finished.
    Therefore, $v.f < u.f$. ■

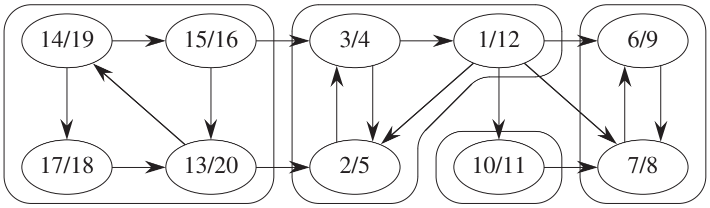
Strongly connected components

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 $u \leadsto v$ and $v \leadsto u$.

Example

[Don't show discovery/finish times yet.]



Algorithm uses G^{T} = ***transpose*** of G.

- $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$.
- G^{T} is G with all edges reversed.

Can create G^{T} in $\Theta(V + E)$ time if using adjacency lists.

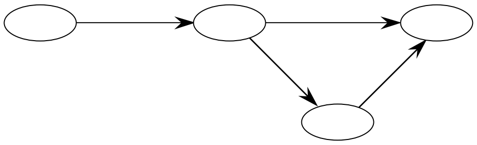
Observation

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

- $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$.
- 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

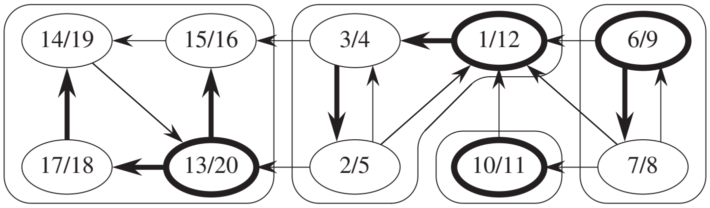
G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \leadsto u'$ in G. Then there cannot also be a path $v' \leadsto v$ in G.

Proof Suppose there is a path $v' \leadsto v$ in G. Then there are paths $u \leadsto u' \leadsto v'$ and $v' \leadsto v \leadsto u$ 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 finish times u.f for each vertex u
  create G^T
  call DFS(G^T), but in the main loop, consider vertices in order of decreasing u.f
        (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 in G. *[Now show discovery and finish times in G.]*
2. G^{T}.
3. DFS in G^{T}. *[Discovery and finish times are from first DFS in G. Roots in second DFS in G^{T} are drawn with heavy outlines, tree edges in second DFS are drawn with heavy lines.]*



Time: $\Theta(V + E)$.

How can this possibly work?

Idea

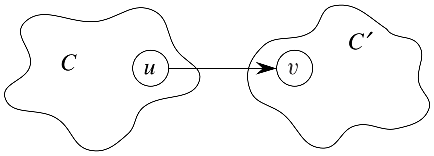
By considering vertices in second DFS in decreasing order of finish times from first DFS, visiting vertices of the component graph in topological sort order.

To prove that it works, first deal with 2 notational issues:

- Will be discussing $u.d$ and $u.f$. These always refer to *first* DFS.
- Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(U) = \min\{u.d : u \in U\}$ (earliest discovery time in U)
  - $f(U) = \max\{u.f : u \in U\}$ (latest finish time in U)

Lemma

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose that 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 $x.d$, 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, $x.f = f(C) > f(C')$.
- If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $y.d$, 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, $y.f = f(C')$.

  At time $y.d$, 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 $y.f$, all vertices in C are still white.

  Therefore, for all $w \in C$, $w.f > y.f$, 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.

The second DFS, on G^{T}, starts with an SCC C such that $f(C)$ is maximum. The second DFS starts from some $x \in C$, and it visits all vertices in C. The 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, the second DFS visits *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'.

The process continues.

Each root chosen for the second DFS 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.

Visiting vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in reverse of topologically sorted order.

[The book has a formal proof.]

Solutions for Chapter 20: Elementary Graph Algorithms

Solution to Exercise 20.1-1

It takes $\Theta(V + E)$ time to compute either the in-degree of every vertex or the out-degree of every vertex. For the out-degrees, just count the length of each adjacency list. For the in-degrees, do the following:

```
for each vertex $u \in G.V$
    $u.\mathit{in\text{-}degree} = 0$
for each vertex $u \in G.V$
    for each vertex $v$ in $G.\mathit{Adj}[u]$
        $v.\mathit{in\text{-}degree} = v.\mathit{in\text{-}degree} + 1$
```

When this code completes, the *in-degree* attribute of each vertex has that vertex's in-degree. The code runs in $\Theta(V + E)$ time.

Solution to Exercise 20.1-2

Adjacency list contents:
1: 2, 3
2: 1, 4, 5
3: 1, 6, 7
4: 2
5: 2
6: 3
7: 3

Adjacency matrix:

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
$$

Solution to Exercise 20.1-3

```
TRANSPOSE-ADJACENCY-LIST(G)
  allocate $G^{\mathrm{T}}.Adj$ with $|G.V|$ entries, each an empty linked list
  for each vertex $u \in G.V$
      for each vertex $v$ in $G.Adj[u]$
          add edge $(v, u)$ to the linked list in $G^{\mathrm{T}}.Adj[v]$
  return $G^{\mathrm{T}}$
```

This procedure runs in $\Theta(V + E)$ time because each edge is examined once and its reverse is added to a linked list once.

```
TRANSPOSE-ADJACENCY-MATRIX(G)
  $n = |G.V|$
  allocate an $n \times n$ adjacency matrix $G^{\mathrm{T}} = (g_{ij}^{\mathrm{T}})$
  for $i = 1$ to $n$
      for $j = 1$ to $n$
          $g_{ji}^{\mathrm{T}} = g_{ij}$
  return $G^{\mathrm{T}}$
```

This procedure runs in $\Theta(V^2)$ time because it examines and writes each of n^2 entries once.

Solution to Exercise 20.1-4

To convert a multigraph $G = (V, E)$ represented by adjacency list to an undirected graph $G' = (V, E')$ with no self-loops, start by creating a list L of all edges $(u, v) \in E$. Then sort L using radix sort. Go through the sorted list L, and the first time encountering edge (u, v) such that $u \neq v$, add edge (u, v) to $G'.Adj[u]$.

Radix sort requires 2 passes of counting sort, each taking $O(V + E)$ time: list L contains $|E|$ edges, and the vertices in each edge are integers in the range 1 to $|V|$. The remainder of the algorithm takes $O(V + E)$ time as well.

Solution to Exercise 20.1-5

With the adjacency-matrix representation, computing the square of a graph is akin to squaring a matrix, but keeping all entries as 0 or 1. We also have to add self-loops for all vertices because each vertex contains a path with 0 edges from itself to itself.

```
SQUARE-ADJACENCY-MATRIX(G)
  allocate $n \times n$ matrix $G^2 = (g^2_{ij})$, with all entries initially 0
  for $i = 1$ to $n$
      $g^2_{ii} = 1$                    // path of length 0: $i \leadsto i$
      for $k = 1$ to $n$
          if $g_{ik} == 1$
              $g^2_{ik} = 1$            // path of length 1: $i \leadsto k$
          for $j = 1$ to $n$
              if $g_{ik} == 1$ and $g_{kj} == 1$
                  $g^2_{ij} = 1$        // path of length 2: $i \leadsto k \leadsto j$
  return $G^2$
```

This procedure takes $O(V^3)$ time.

With the adjacency-list representation, a procedure can go through each edge (i, j) and then find all the edges (j, k) in j's adjacency list, adding edge (i, k) to G^2. There is one hitch, however: making sure not to add an edge multiple times. If there are edges (i, j), (j, k), (i, h), and (h, k), then there are two paths of length 2 from i to k ($i \leadsto j \leadsto k$ and $i \leadsto h \leadsto k$), but edge (i, k) should appear in i's adjacency list in G^2 just once. We'll build an adjacency matrix for G^2, just like the output of SQUARE-ADJACENCY-MATRIX, but by going through the adjacency lists of G. As we'll see, that takes $O(V^2 + VE)$ time, which beats $O(V^3)$ for SQUARE-ADJACENCY-MATRIX if $|E| = o(V^2)$. Once the adjacency matrix of G^2 is built, converting it to adjacency lists takes $O(V^2)$ time, for a total of $O(V^2 + VE)$ time.

```
SQUARE-ADJACENCY-LISTS(G)
  // Since $G^2$ is the output adjacency list representation, use a different name
  // for the adjacency matrix.
  allocate $n \times n$ matrix $M = (m_{ij})$, with all entries initially 0
  for each vertex $i$ in $G.Adj$
      $m_{ii} = 1$
      for each vertex $j$ in $G.Adj[i]$
          $m_{ij} = 1$
          for each vertex $k$ in $G.Adj[j]$
              $m_{ik} = 1$
  // $M$ is the complete adjacency matrix for $G^2$. Convert to adjacency lists.
  allocate $G^2.Adj$ with $|G.V|$ entries, each an empty linked list
  for each vertex $i$ in $G.Adj$
      for each vertex $j$ in $G.Adj$
          if $m_{ij} == 1$
              add edge $(i, j)$ to the linked list in $G^2.Adj[i]$
  return $G^2$
```

Allocating M and converting M to adjacency lists takes $O(V^2)$ time. The first set of triply-nested **for** loops takes $O(VE)$ time because for each of the $|E|$ edges (i, j) in the middle loop, at most $|V|$ edges appear in j's adjacency list. Thus, the entire procedure takes time $O(V^2 + VE)$.

Solution to Exercise 20.1-6

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 $|V| \times |V|$
  for $j = 1$ to $|V|$            // check for a 1 in row $k$
      if $a_{kj} == 1$
          return FALSE
  for $i = 1$ to $|V|$            // check for an off-diagonal 0 in column $k$
      if $a_{ik} == 0$ and $i \neq k$
          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 $i \neq j$, so that no other vertex i could be a universal sink.

The following procedure takes a $|V| \times |V|$ 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 $|V| \times |V|$
  $i = 1$
  $j = 1$
  while $i \leq |V|$ and $j \leq |V|$
      if $a_{ij} == 1$
          $i = i + 1$
      else $j = j + 1$
  if $i \leq |V|$ and IS-SINK$(A, i)$ == TRUE
      return $i$ "is a universal sink"
  else return "there is no 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 all vertices have been eliminated 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 the procedure 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 the procedure found a 0 in every column, it found a 0 in every column k such that $i < k \le |V|$. Moreover, it never examined any matrix entries in rows greater than i, and so it never examined the diagonal entry in any column k such that $i < k \le |V|$. Therefore, all the 0s that were 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 20.1-7
This solution is also posted publicly

$$BB^{\mathrm{T}}(i,j) = \sum_{e \in E} b_{ie} b_{ej}^{\mathrm{T}} = \sum_{e \in E} b_{ie} b_{je} \ .$$

- 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 $i \ne j$, then $b_{ie}b_{je} = -1$ when $e = (i,j)$ or $e = (j,i)$, and 0 otherwise.

Thus,

$$BB^{\mathrm{T}}(i,j) = \begin{cases} \text{in-degree of } i + \text{ out-degree of } i & \text{if } i = j \ , \\ -(\text{\# of edges connecting } i \text{ and } j) & \text{if } i \ne j \ . \end{cases}$$

Solution to Exercise 20.1-8

By Theorems 11.1 and 11.2, the expected search time is $\Theta(1 + \alpha)$, where α is the load factor of the hash table. If α is a constant, then so is the expected search time.

Using a hash table creates a couple of disadvantages. First, the worst-case search time for a neighbor of vertex u is $\Theta(\text{degree}(u))$. Second, creating a hash table for each vertex requires significant extra space compared with plain adjacency lists.

One way to reduce the worst-case search time is to sort each linked list in a hash-table slot. Then, by using binary search, the worst-case search time for a neighbor of vertex u is $\Theta(\lg \text{degree}(u))$. Alternatively, the neighbors of a vertex can be stored in a balanced binary search tree, again giving a worst-case search time for a neighbor of vertex u as $\Theta(\lg \text{degree}(u))$. The expected search time would no longer be constant, however.

Solution to Exercise 20.2-1

	1	2	3	4	5	6
d	∞	3	0	2	1	1
π	NIL	4	NIL	5	3	3

Solution to Exercise 20.2-2

	r	s	t	u	v	w	x	y	z
d	2	1	1	0	2	3	2	1	3
π	s	u	u	NIL	s	r	y	u	x

Solution to Exercise 20.2-3

The BFS procedure cares only whether a vertex is white or not. A vertex v must become non-white at the same time that $v.d$ is assigned a finite value so that the procedure does not attempt to assign to $v.d$ again. That is why lines 5 and 14 change the color of a vertex. Once a vertex's becomes non-white, its color does not need to change again.

A vertex is non-white if and only if it has a finite d value. All the lines that assign colors to vertices can be eliminated from the BFS procedure, and line 13 could be changed to read "**if** $v.d == \infty$".

Solution to Exercise 20.2-4

BFS runs in $O(V^2)$ time with an adjacency-matrix representation of a graph because there are $O(V^2)$ potential edges to check.

Solution to Exercise 20.2-5

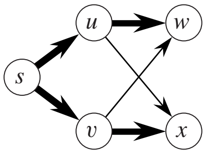
This solution is also posted publicly

The correctness proof for the BFS algorithm shows that $u.d = \delta(s, u)$, and the algorithm doesn't assume that the adjacency lists are in any particular order.

In Figure 20.3, 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 20.2-6

The edges in E_π are drawn with heavy lines 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 20.2-7

Create a graph G where each vertex represents a wrestler and each edge represents a rivalry. The graph contains 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 faces and all wrestlers whose distance is odd to be heels (or vice-versa). Then check each edge to verify that it goes between a face and a heel. This solution takes $O(n + r)$ time for the BFS, $O(n)$ time to designate each wrestler as a face or heel, and $O(r)$ time to check edges, which is $O(n + r)$ time overall.

Solution to Exercise 20.2-8

To find the diameter of tree T, select any vertex x in T, and run BFS from x. Let y be the last vertex discovered in the BFS from x. Now run BFS from y, and let

z be the last vertex discovered in this second BFS. We claim that the diameter of T is the value of $z.d$ from the second BFS. In other words, since there is a unique simple path between each pair of vertices in T, we claim that the diameter equals $\delta(y, z)$.

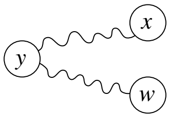
To prove that the diameter of T is the distance between y and z, let u and v be any two vertices in T such that $\delta(u, v)$ equals the diameter of T.

Claim

$\delta(y, v) = \delta(u, v)$.

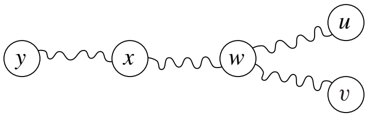
Proof of claim Let w be the first vertex on the path $u \leadsto v$ discovered during the first BFS. There are three possibilities for the path relationships among x, y, and w.

- x is not on the path $w \leadsto y$ and w is not on the path $x \leadsto y$. Then the path $x \leadsto w$ must go through y:

  Since w is farther from x than y is, we get the contradiction that y is not the last vertex discovered during the BFS from x. Therefore, this case cannot occur.

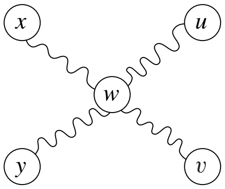
- x is on the path $w \leadsto y$. Then because w is the first vertex discovered in the BFS from x, no other vertex in the path $u \leadsto v$ can be on the path $x \leadsto w$, for it would have been discovered before w in the BFS from x:

  In this case, we have $\delta(w, y) \geq \delta(x, y)$. Since y is the last vertex discovered in the BFS from x, we have $\delta(x, y) \geq \delta(x, u)$. Since the path $x \leadsto u$ includes the subpath $w \leadsto u$, we have $\delta(x, u) \geq \delta(w, u)$. Putting these inequalities together gives $\delta(w, y) \geq \delta(x, y) \geq \delta(x, u) \geq \delta(w, u)$, so that $\delta(w, y) \geq \delta(w, u)$. Since the path $v \leadsto w$ is not a subpath of $u \leadsto y$, we can add $\delta(v, w)$ to both sides, giving $\delta(y, v) = \delta(v, y) = \delta(v, w) + \delta(w, y) \geq \delta(v, w) + \delta(w, u) = \delta(v, u) = \delta(u, v)$, so that $\delta(y, v) \geq \delta(u, v)$.

  Since the diameter equals $\delta(u, v)$, we have $\delta(y, v) \leq \delta(u, v)$, and so $\delta(y, v) = \delta(u, v)$.

- w is on the path $x \leadsto y$. As in the previous case, no other vertex in the path $u \leadsto v$ can be on the path $x \leadsto w$, for it would have been discovered before w in the BFS from x:

Since $x \rightsquigarrow w$ is a subpath of both $x \rightsquigarrow y$ and $x \rightsquigarrow u$ and y is the last vertex discovered in the BFS from x, we must have $\delta(x,w) + \delta(w,y) = \delta(x,y) \geq \delta(x,u) = \delta(x,w) + \delta(w,u)$. Subtracting $\delta(x,w)$ from both sides gives $\delta(w,y) \geq \delta(w,u)$. Because the diameter equals $\delta(v,u)$, we have $\delta(v,u) \geq \delta(v,y)$. Then, we have $\delta(v,w) + \delta(w,u) = \delta(v,u) \geq \delta(v,y) = \delta(v,w) + \delta(w,y)$, and subtracting $\delta(v,w)$ from both sides gives $\delta(w,u) \geq \delta(w,y)$. Having both $\delta(w,y) \geq \delta(w,u)$ and $\delta(w,u) \geq \delta(w,y)$ means that $\delta(w,y) = \delta(w,u)$. Now, adding back $\delta(v,w)$ to both sides gives $\delta(y,v) = \delta(v,y) = \delta(v,w) + \delta(w,y) = \delta(v,w) + \delta(w,u) = \delta(v,u) = \delta(u,v)$. ■ (claim)

Because the diameter equals $\delta(u,v)$, we have $\delta(y,z) \leq \delta(u,v)$. And because z is the last vertex discovered in the BFS from y, we have $\delta(y,z) \geq \delta(y,v)$, giving $\delta(y,v) \leq \delta(y,z) \leq \delta(u,v)$. From our claim that $\delta(y,v) = \delta(u,v)$, this inequality collapses to an equality, giving $\delta(u,v) = \delta(y,z)$, so that the diameter equals $\delta(y,z)$.

Finding the diameter, therefore, takes two executions of BFS, each of which takes $\Theta(V + E)$ time. Since T is a tree, $|E| = |V| - 1$, giving a total time of just $\Theta(V)$.

Solution to Exercise 20.3-1

The entries in the charts show the edge types that may occur at some point during DFS.

T = tree edge
B = back edge
F = forward edge
C = cross edge

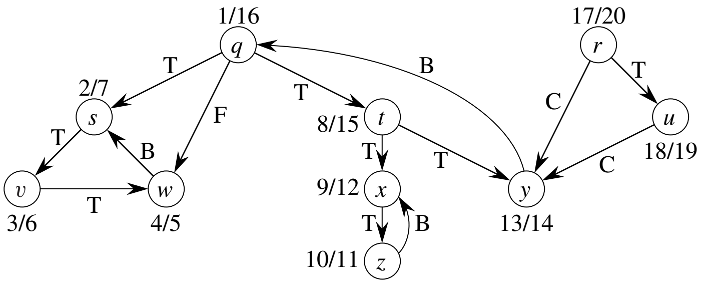
Directed graph:

			to	
		white	gray	black
from	white	T B F C	B C	C
	gray	T F	T B F	T F C
	black		B	T B F C

Undirected graph:

			to	
		white	gray	black
from	white	T B	T B	
	gray	T B	T B	T B
	black		T B	T B

Solution to Exercise 20.3-2



Solution to Exercise 20.3-3

Parenthesis structure: $(u\ (v\ (y\ (x\ x)\ y)\ v)\ u)\ (w\ (z\ z)\ w)$

Solution to Exercise 20.3-4

The DFS and DFS-VISIT procedures care only whether a vertex is white or not. Coloring vertex u gray when it is first visited, in line 3 of DFS-VISIT, ensures that u will not be visited again. Once a vertex's color becomes non-white, it need not change again.

Solution to Exercise 20.3-5

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 20.8, therefore, (u, v) is a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$.

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 $v.d = u.d < u.f = v.f$. If (u, v) is not a self-loop, then u is a descendant of v in the depth-first forest, and by Corollary 20.8, $v.d < u.d < u.f < v.f$.

Now, suppose that $v.d \le u.d < u.f \le v.f$. If u and v are the same vertex, then $v.d = u.d < u.f = v.f$, and (u, v) is a self-loop and hence a back edge. If u and v are distinct, then $v.d < u.d < u.f < v.f$. By the parenthesis theorem, interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, 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, the parenthesis theorem says that the intervals $[u.d, u.f]$ and $[v.d, v.f]$

are entirely disjoint. Thus, we must have either $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$. We claim that we cannot have $u.d < v.d$ if (u, v) is a cross edge. Why? If $u.d < v.d$, then v is white at time $u.d$. By the white-path theorem, v is a descendant of u, which contradicts (u, v) being a cross edge. Thus, we must have $v.d < v.f < u.d < u.f$.

Now suppose that $v.d < v.f < u.d < u.f$. By the parenthesis theorem, neither u nor v is a descendant of the other, which means that (u, v) must be a cross edge.

Solution to Exercise 20.3-6

The nonrecursive version of DFS needs to change the DFS-VISIT procedure, but not the DFS procedure.

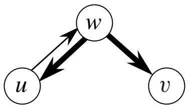
It is tempting to put only vertices on the stack, but that is insufficient to truly execute a depth-first search. You need to keep track of the position in the adjacency list as well. Therefore, the stack contains ordered pairs (u, v), where u is a vertex and v is either an element in u's adjacency list or NIL. Because we are working with u's adjacency list explicitly, we assume that it is singly linked, with an attribute *head*, and that each element has attributes *vertex*, naming an adjacent vertex, and *next*, for the next element in u's adjacency list (which is NIL if there is no next element in u's adjacency list).

```
DFS-VISIT$(G, u)$
  $n = |G.V|$
  create an empty stack $S$ with $n$ slots
  $time = time + 1$
  $u.d = time$
  $u.color$ = GRAY
  PUSH$(S, (u, G.Adj[u].head), n)$
  while TRUE          // break out with the return statement
      $(u, element)$ = POP$(S)$
      while $element$ == NIL
          $u.color$ = BLACK
          $time = time + 1$
          $u.f = time$
          if STACK-EMPTY$(S)$
              return
          else $(u, element)$ = POP$(S)$
      PUSH$(S, (u, element.next), n)$
      $v = element.vertex$
      if $v.color$ == WHITE
          $v.\pi = u$
          $time = time + 1$
          $v.d = time$
          $v.color$ = GRAY
          PUSH$(S, (v, G.Adj[v].head), n)$
```

Solution to Exercise 20.3-7

Consider the example graph and depth-first search below.

	d	f
w	1	6
u	2	3
v	4	5

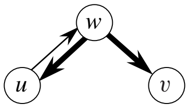


Clearly, there is a path from u to v in G. The heavy edges are in the depth-first forest produced by the depth-first search. The search produces $u.d < v.d$, but v is not a descendant of u in the forest.

Solution to Exercise 20.3-8

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 heavy edges of G are in the depth-first forest produced by the depth-first search. However, $v.d > u.f$ and the conjecture is false.

Solution to Exercise 20.3-9

For a directed graph, the DFS procedure does not need to change, but DFS-VISIT does:

```
DFS-VISIT$(G, u)$
  $time = time + 1$                  // white vertex $u$ has just been discovered
  $u.d = time$
  $u.color =$ GRAY
  for each vertex $v$ in $G.Adj[u]$  // explore each edge $(u, v)$
      if $v.color ==$ WHITE
          $v.\pi = u$
          print $(u, v)$ "is a tree edge"
          DFS-VISIT$(G, v)$
      elseif $v.color ==$ GRAY
          print $(u, v)$ "is a back edge"
      elseif $u.d < v.d$             // $v.color$ is black
          print $(u, v)$ "is a forward edge"
      else print $(u, v)$ "is a cross edge"
  $time = time + 1$
  $u.f = time$
  $u.color =$ BLACK                  // blacken $u$; it is finished
```

The procedure discerns between the two cases when $v.color$ is BLACK by using Exercise 20.3-5 to compare discovery times of vertices u and v.

For an undirected graph, if vertex v is white when exploring edge (u, v), the result is the same as for a directed graph: (u, v) is a tree edge. If v is gray, then by Theorem 20.10, (u, v) is either a back edge or is a tree edge originally explored while visiting v's neighbors. Unfortunately, the discovery times $u.d$ and $v.d$ do not contain enough information to discern the difference between tree and back edges. Instead, we need to mark each edge to indicate whether it has already been visited. If (u, v) has already been visited, then do not print it out a second time. The DFS procedure needs to initialize all edges as unvisited.

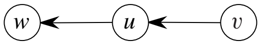
```
DFS$(G)$
  for each vertex $u \in G.V$
      $u.color =$ WHITE
      $u.\pi =$ NIL
      for each vertex $v$ in $G.Adj[u]$
          $(u, v).visited =$ FALSE
  $time = 0$
  for each vertex $u \in G.V$
      if $u.color ==$ WHITE
          DFS-VISIT$(G, u)$
```

```
DFS-VISIT$(G, u)$
  $time = time + 1$                  // white vertex $u$ has just been discovered
  $u.d = time$
  $u.color =$ GRAY
  for each vertex $v$ in $G.Adj[u]$  // explore each edge $(u, v)$
      if $v.color ==$ WHITE
          $v.\pi = u$
          print $(u, v)$ "is a tree edge"
          $(u, v).visited =$ TRUE
          DFS-VISIT$(G, v)$
      elseif $(u, v).visited ==$ FALSE
          print $(u, v)$ "is a back edge"
  $time = time + 1$
  $u.f = time$
  $u.color =$ BLACK                  // blacken $u$; it is finished
```

Solution to Exercise 20.3-10

Consider the example graph and depth-first search below.

	d	f
w	1	2
u	3	4
v	5	6



Vertex u has both incoming and outgoing edges in G, but a depth-first search of G produces a depth-first forest where each vertex is in a tree by itself.

Solution to Exercise 20.3-11

To compute a path in connected, undirected graph $G = (V, E)$ that traverses each edge exactly once in each direction, first perform a depth-first search of G. Since G is connected and undirected, it doesn't matter which vertex is passed to the call of DFS-VISIT. A single depth-first tree, let's say T, is produced. (For the first part of this question, a breadth-first search and breadth-first tree would do just as well, but not for the second part of this question.) Then perform a full walk of T, where a full walk is as defined in the proof of Theorem 35.2 on page 1112: it lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. When visiting vertex v from vertex u in the full walk, append edge (u, v) to the path.

To find your way out of a maze given a large supply of pennies, you emulate the DFS-VISIT procedure to perform a full walk of the depth-first tree. (You cannot necessarily navigate through a maze using breadth-first search, which is why the first part of the problem couched the solution in terms of DFS.) Each junction in the maze is a vertex in the graph, and each hallway is an edge. When you reach a junction, you need to know whether you have already visited each of the hallways it connects and which hallway got you to that junction so that you can backtrack. One way to record this information is when you arrive at a junction for which none of the hallways have pennies, lay down two pennies in the hallway that you just came through, so that you can see them from the junction. When you start down a hallway, lay down one penny so that you can see it from the junction. That way, when you arrive at a junction, you know whether it connects to any unexplored hallways, because they don't have pennies. If you have explored all the connecting hallways, then backtrack through the one hallway where you had laid down two pennies.

Solution to Exercise 20.3-12

This solution is also posted publicly

The following pseudocode modifies the DFS and DFS-VISIT procedures to assign values to the *cc* attributes of vertices.

```
DFS(G)
  for each vertex $u \in G.V$
      $u.color = \text{WHITE}$
      $u.\pi = \text{NIL}$
  $time = 0$
  $counter = 0$
  for each vertex $u \in G.V$
      if $u.color == \text{WHITE}$
          $counter = counter + 1$
          DFS-VISIT$(G, u, counter)$

DFS-VISIT$(G, u, counter)$
  $u.cc = counter$              // label the vertex
  $time = time + 1$
  $u.d = time$
  $u.color = \text{GRAY}$
  for each vertex $v$ in $G.Adj[u]$
      if $v.color == \text{WHITE}$
          $v.\pi = u$
          DFS-VISIT$(G, v, counter)$
  $time = time + 1$
  $u.f = time$
  $u.color = \text{BLACK}$
```

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 $u.cc = v.cc$ 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 are 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 20.3-13

To determine whether a directed graph is singly connected, run DFS-VISIT once from each vertex and classify the edges in each execution. If any edge in any execution of DFS-VISIT is classified as a forward or cross edge, then the graph is not singly connected. If every edge in every execution of DFS-VISIT is classified as either a tree or back edge, then the graph is singly connected. Obviously, you can stop calling DFS-VISIT as soon as any edge is classified as a forward or cross edge. The running time for this algorithm is $O(V^2 + VE)$.

We need to prove that $G = (V, E)$ is singly connected if and only if no call of DFS-VISIT yields a forward or cross edge. To do so, we'll prove the contrapositive:

Lemma

Directed graph $G = (V, E)$ is not singly connected if and only if for some vertex $u \in V$, a call of DFS-VISIT(G, u) yields a forward or cross edge.

Proof Suppose that G is not singly connected. Then there are vertices u and v such that G contains more than one simple path $u \leadsto v$. When DFS-VISIT(G, u) executes, it creates a simple path p_1 of tree edges $u \overset{p_1}{\leadsto} v$. Let p_2 be another simple path $u \overset{p_2}{\leadsto} v$. Path p_2 must contain some edge e not in p_1. Edge e cannot be a tree edge, for otherwise the depth-first tree would not be a tree. Nor can edge e be a back edge, for otherwise path p_2 would not be simple. Therefore, e must be either a forward edge or a cross edge.

Now suppose that some call DFS-VISIT(G, u) yields an edge (v, w) that is a forward or cross edge. If (v, w) is a forward edge, then there are two simple paths $v \leadsto w$: the single edge (v, w) and the path $v \leadsto w$ in the depth-first tree. If (v, w)

is a cross edge, there are two simple paths $u \rightsquigarrow w$: the path $u \rightsquigarrow w$ in the depth-first tree, and the path $u \overset{p}{\rightsquigarrow} v \to w$, where p is the path $u \rightsquigarrow v$ in the depth-first tree. ■

It is possible to reduce the running time to $O(V^2)$. See "Determining Single Connectivity in Directed Graphs" by Adam L. Buchsbaum and Martin C. Carlisle, Princeton University Computer Science Research Report CS-TR-390-92, September 1992.

Solution to Exercise 20.4-1

The depth-first search produces the following discovery and finish times:

	m	n	o	p	q	r	s	t	u	v	w	x	y	z
d	1	21	22	27	2	6	23	3	7	10	11	15	9	12
f	20	26	25	28	5	19	24	4	8	17	14	16	18	13

The topologically sorted order is $p, n, o, s, m, r, y, v, x, w, z, u, q, t$.

Solution to Exercise 20.4-2

To count the number of simple paths from s to t in dag G, add an attribute *count* to each vertex and then execute the following procedure.

```
COUNT-PATHS($G, s, t$)
  for each vertex $v \in G.V$
      $v.count = 0$
  $t.count = 1$
  topologically sort $G$, and let the topologically sorted order be
      $\langle v_1, v_2, \ldots, v_n \rangle$, where $n = |G.V|$
  let $s = v_i$ and $t = v_j$, where $i \le j$
  for $k = j$ downto $i$   // process vertices in reverse topo order from $t$ to $s$
      for each vertex $u$ in $G.Adj[v_k]$
          $v_k.count = v_k.count + u.count$
  return $v_i.count$         // $v_i.count$ is same as $s.count$
```

We show that $s.count$ is correct by the following loop invariant:

> **Loop invariant:** After an iteration of the second **for** loop (the loop with the header "**for** $k = j$ **downto** i"), each value $v_k.count, v_{k+1}.count, \ldots, v_n.count$ contains the number of simple paths from v_k to t.

Initialization: Because vertices $v_{j+1}, v_{j+2}, \ldots, v_n$ all follow $v_j = t$ in the topologically sorted order, their *count* values never change from their initial values of 0, which is correct because there are no paths to t from vertices following t in the topologically sorted order. The first iteration of the **for** loop sets $t.count$ to 1, which is correct since there is just one simple path from t to itself: a path with no edges.

Maintenance: An iteration for vertex v_k sets $v_k.count$ to the sum of the *count* values for all vertices adjacent to v_k. By the loop invariant, these *count* values are correct for these vertices. For each vertex u that is adjacent to v_k, there are simple paths from v_k to t going $v_k \to u \rightsquigarrow t$, so that for a fixed vertex u, the number of such paths is given by $u.count$. Summing the *count* values of the v_k's neighbors into v_k gives the total number of simple paths from v_k to t.

Termination: The loop terminates because it visits vertices from t to s in reverse topologically sorted order. By the loop invariant, after the iteration for $v_k = v_i = s$, the value $v_i.count = s.count$ contains the number of simple paths from s to t.

Solution to Exercise 20.4-3

This solution is also posted publicly

An undirected graph is acyclic (i.e., a forest) if and only if a DFS yields no back edges.

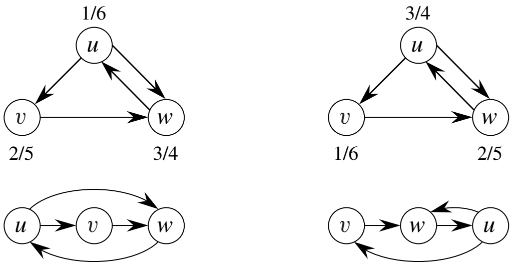
- If there's a back edge, there's a cycle.
- If there's no back edge, then by Theorem 20.10, there are only tree edges. Hence, the graph is acyclic.

Thus, to determine whether an undirected graph contains a cycle, run DFS and classify the edges: if any edge is a back edge, there's a cycle.

- Time: $O(V)$.
  Not $O(V + E)$: Once $|V|$ distinct edges have been seen, at least one of them must be a back edge because (by Theorem B.2 on page 1169) in an acyclic (undirected) forest, $|E| \le |V| - 1$.

Solution to Exercise 20.4-4

The conjecture is false. In the graph below, performing depth-first search starting from vertex u gives the result on the left, with the topologically sorted graph below. There is only one "bad" edge: (w, u). If depth-first search starts from vertex v instead, the result is on the right, with two "bad" edges: (u, v) and (u, w). Therefore, the TOPOLOGICAL-SORT procedure does not necessarily minimize the number of "bad" edges.



Solution to Exercise 20.4-5

```
TOPOLOGICAL-SORT($G$)
  // Initialize in-degree, $\Theta(V)$ time.
  for each vertex $u \in G.V$
      $u.in\text{-}degree = 0$
  // Compute in-degree, $\Theta(V + E)$ time.
  for each vertex $u \in G.V$
      for each vertex $v$ in $G.Adj[u]$
          $v.in\text{-}degree = v.in\text{-}degree + 1$
  // Initialize queue, $\Theta(V)$ time.
  $Q = \emptyset$
  for each vertex $u \in G.V$
      if $u.in\text{-}degree == 0$
          ENQUEUE($Q, u$)
  // while loop takes $O(V + E)$ time.
  while $Q \neq \emptyset$
      $u$ = DEQUEUE($Q$)
      output $u$
      // for loop executes $O(E)$ times total.
      for each vertex $v$ in $G.Adj[u]$
          $v.in\text{-}degree = v.in\text{-}degree - 1$
          if $v.in\text{-}degree == 0$
              ENQUEUE($Q, v$)
  // Check for cycles, $O(V)$ time.
  for each vertex $u \in G.V$
      if $u.in\text{-}degree \neq 0$
          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, 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. Computing all in-degrees takes $\Theta(V + E)$ time ($|V|$ adjacency lists accessed, $|E|$ edges total found in those lists, $\Theta(1)$ work for each edge).

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 FIFO queue works.) Initializing the queue takes one pass over the vertices doing $\Theta(1)$ work per vertex, for total time $\Theta(V)$.

When processing each vertex from the queue, the procedure effectively removes its outgoing edges from the graph by decrementing the in-degree of each vertex one of those edges enters, and it enqueues any vertex whose in-degree goes down to 0. There is 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 $\Theta(E)$, $O(E)$ 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. To see why, assume for the purpose of contradiction that some vertex v_0 is not output. Vertex v_0 cannot start out with in-degree 0 (or it would be output), so that there are edges entering 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, and so we must have $v_i = v_j$ for some i and j in this sequence, which gives the contradiction that there is a cycle.

Conversely, if there are cycles, not all vertices will be output, because some in-degrees never become 0. To see why, assume for the purpose of contradiction 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 20.5-1

Adding an edge cannot increase the number of strongly connected components, but it can reduce the number of strongly connected components by as little as 0 or as much as $|V| - 1$. A change of 0 occurs if the entire graph is already strongly connected before adding the edge. A decrease of $|V| - 1$ occurs if the graph is linear ($v_1 \to v_2 \to v_3 \to \cdots \to v_{|V|-1} \to v_{|V|}$) and the added edge $(v_{|V|}, v_1)$ completes the cycle.

Solution to Exercise 20.5-2

The finish times, as computed in line 1 of STRONGLY-CONNECTED-COMPONENTS on the graph of Figure 20.6, in order of decreasing finish times:

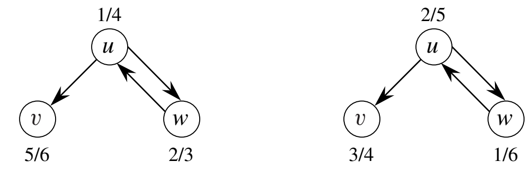
vertex	r	u	q	t	y	x	z	s	v	w
finish time	20	19	16	15	14	12	11	7	6	5

The tree edges in the forest produced in line 3: (q, y), (y, t), (x, z), (s, w), (w, v).

The connected components: $\{r\}, \{u\}, \{q, t, y\}, \{x, z\}, \{s, v, w\}$.

Solution to Exercise 20.5-3

No, Professor Bacon's algorithm can produce an incorrect result. In the graph below, the left side shows one possible result of the first depth-first search. Since vertex w has the lowest finish time, the second depth-first search, on the right, starts from w. But the second depth-first search puts all three vertices into the same depth-first tree. The result is that all three vertices are erroneously placed in the same connected component, instead of the correct answer of $\{u, w\}, \{u\}$.



Solution to Exercise 20.5-4

Since G and G^{T} have the same SCCs, the vertices of G^{SCC} and $(G^{\mathrm{T}})^{\mathrm{SCC}}$ are the same. The only difference between G^{SCC} and $(G^{\mathrm{T}})^{\mathrm{SCC}}$ is that the edge directions are reversed. Taking the transpose of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ reverses the reversed edge directions, so that $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$ is the same as G^{SCC}.

Solution to Exercise 20.5-5

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 $u.scc$, giving the number of the strongly connected component containing vertex u, for each vertex u. Without loss of generality, assume that $u.scc$ is an integer in the set $\{1, 2, \ldots, |V|\}$.

Construct the multiset (a set that can contain the same object more than once) $T = \{u.scc : u \in V\}$, and sort it by using counting sort. Since the values being sorted are integers in the range 1 to $|V|$, the time to sort is $O(V)$. Go through the sorted multiset T and upon finding 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

$$S = \{(x, y) : \text{there is an edge } (u, v) \in E, x = u.scc, \text{ and } y = v.scc\} .$$

Construct this set in $\Theta(E)$ time by going through all edges in E and looking up $u.scc$ and $v.scc$ for each edge $(u, v) \in E$.

Having constructed S, remove all elements of the form (x, x). Alternatively, when constructing S, do not put an element in S when there is an edge (u, v) for which $u.scc = v.scc$. 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, perform two passes of counting sort. The time to do so is $O(V + E)$, since the values being sorted are integers in the range 1 to $|V|$.

Finally, go through the sorted set S, and upon finding 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 (x, y) is added 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 20.5-6

The idea is to replace the edges within each SCC by one simple, directed cycle and then remove redundant edges between SCCs. 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 SCCs of G. Time: $\Theta(V + E)$, using the SCC algorithm in Section 20.5.
2. Form the component graph G^{SCC}. Time: $O(V + E)$, by Exercise 20.5-5.
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 SCCs: $O(V)$.
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 20.5-7

To determine whether $G = (V, E)$ is semiconnected, do the following:

1. Call STRONGLY-CONNECTED-COMPONENTS.

2. Form the component graph. (By Exercise 20.5-5, this step takes $O(V + E)$ time.)
3. Topologically sort the component graph. (Recall that it's a dag.) Assuming that G contains k SCCs, 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 G is semiconnected; otherwise it is not.

Because all vertices in each SCC are mutually reachable from each other, the key idea is 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. A topological sort has to respect every edge in the graph. If there's a linear chain, a topological sort *must* output the vertices in order.

Now we'll show that G is semiconnected if and only if its component graph contains a linear chain.

First, suppose that the component graph contains a linear chain. Then for every pair of vertices u and v in the component graph, there is a path between them. If u precedes v in the linear chain, then there's a path $u \leadsto v$, so that there is a path from every vertex in u's component to every vertex in v's component. Otherwise, v precedes u, so that there's a path $v \leadsto u$ and thus a path from every vertex in v's component to every vertex in u's component.

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, there are no paths from vertices in v_i's component to vertices in v_{i+1}'s component or from vertices in v_{i+1}'s component to vertices in v_i's component, and the original graph G 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 $O(V + E)$. 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}) by going through each adjacency list once.

Thus, the total running time is $\Theta(V + E)$.

Solution to Exercise 20.5-8

The algorithm relies on finding the strongly connected components of G and, using Exercise 20.5-5, computing its component graph G^{SCC}. For each component C, we need to find a path ending in C that achieves the greatest Δ value. To do so, we find a vertex C_{max} within C with the maximum label and a vertex C_{min} in G that can reach C_{max} and has the minimum label. Vertices s and t that achieve $\Delta l(s,t)$ are those with the greatest difference $l(C_{\text{max}}) - l(C_{\text{min}})$.

The asymmetry between C_{max} being limited to the vertices in C and C_{min} possibly coming from outside C arises because a path ending in C can start in any component that has a path to some vertex in C.

The following procedure finds the vertices s and t that achieve the value of $\Delta l(s,t)$.

```
FIND-DELTA($G$)
  compute the strongly connected components of $G$
  compute the component graph $G^{\text{SCC}}$
  topologically sort $G^{\text{SCC}}$
  $\Delta = -\infty$
  for each component $C$ in $G^{\text{SCC}}$
      $C_{\text{min}}$ = any vertex in $C$'s component with the minimum label
      $C_{\text{max}}$ = any vertex in $C$'s component with the maximum label
      if $l(C_{\text{max}}) - l(C_{\text{min}}) > \Delta$
          $s = C_{\text{min}}$
          $t = C_{\text{max}}$
          $\Delta = l(C_{\text{max}}) - l(C_{\text{min}})$
  for each component $C$ in $G^{\text{SCC}}$, taken in topologically sorted order
      for each edge $(C, C') \in E^{\text{SCC}}$
          if $l(C_{\text{min}}) < l(C'_{\text{min}})$
              $C'_{\text{min}} = C_{\text{min}}$
              if $l(C'_{\text{max}}) - l(C'_{\text{min}}) > \Delta$
                  $s = C'_{\text{min}}$
                  $t = C'_{\text{max}}$
                  $\Delta = l(C'_{\text{max}}) - l(C'_{\text{min}})$
  return $s$ and $t$
```

To see that this algorithm runs in $O(V+E)$ time, it takes $O(V+E)$ time to find the strongly connected components, compute G^{SCC}, and topologically sort G^{SCC}. The first **for** loop takes $O(V)$ time, and the nested **for** loops take a total of $O(V + E)$ time. The total time comes to $O(V + E)$.

Solution to Problem 20-1
This solution is also posted publicly

a. 1. Suppose (u, v) is a back edge or a forward edge in a BFS of an undirected graph. Without loss of generality, let u be a proper ancestor of v in the breadth-first tree. Since all edges of u are explored before exploring any edges of any of u's descendants, edge (u, v) must be explored when exploring from u. But then (u, v) must be a tree edge.

2. In BFS, an edge (u, v) is a tree edge when the procedure sets $v.\pi = u$. But that occurs only when the procedure also sets $v.d = u.d + 1$. Since neither $u.d$ nor $v.d$ ever changes thereafter, we have $v.d = u.d + 1$ when BFS completes.
3. Consider a cross edge (u, v) where, without loss of generality, u is visited before v. When the edges incident on u are explored, 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 $v.d \le u.d + 1$ by Lemma 20.3. By Corollary 20.4, we have $v.d \ge u.d$. Thus, either $v.d = u.d$ or $v.d = u.d + 1$.

b. 1. Suppose (u, v) is a forward edge. Then it would have been explored while exploring from u, and it would have been a tree edge.
2. Same as for undirected graphs.
3. For any edge (u, v), regardless of whether it's a cross edge, we cannot have $v.d > u.d + 1$, since the BFS visits v at the latest when it explores edge (u, v). Thus, $v.d \le u.d + 1$.
4. Clearly, $v.d \ge 0$ for all vertices v. For a back edge (u, v), v is an ancestor of u in the breadth-first tree, which means that $v.d \le u.d$. (Note that since self-loops are considered to be back edges, we could have $u = v$.)

Solution to Problem 20-3

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 $v \in V$, then $G = (V, E)$ 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 $v \in V$, then we can pick any vertex u for which in-degree(u) = out-degree$(u) \ge 1$ 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$(v) \ge 1$, 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.

Here is the nonconstructive proof. Assume that in-degree(v) = out-degree(v) for all vertices $v \in V$. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a longest path in G that contains each edge at most once. Note that p might not be a simple path: it may visit one or more vertices multiple times, but it traverses each edge no more than once. Path p must include every edge that leaves v_k, since otherwise p could be extended by an edge leaving v_k and it would not be a longest path that contains each edge at most once. Likewise, p must include every edge that enters v_0. Because the in-degree equals the out-degree of every vertex, the only way that p can contain every edge leaving v_k and entering v_0 is if v_0 and v_k are the same vertex, i.e., p is a (not necessarily simple) cycle. We claim that p is, in fact, an Euler tour, which we prove by contradiction. Suppose that p is not an Euler tour. Then, because G is strongly connected, G contains some edge not in p that either enters or leaves some vertex in p. If the edge enters vertex v_i in p, let's call the edge (u, v_i). But then we can construct a path p' that uses this edge that is longer than p but still contains each edge at most once: $p' = \langle u, v_i, v_{i+1}, \ldots, v_k, v_1, v_2, \ldots, v_i \rangle$. Likewise if the edge not in p leaves vertex v_i in p and the edge is (v_i, u), then $p' = \langle v_i, v_{i+1}, \ldots, v_k, v_1, v_2, \ldots, v_i, u \rangle$ is a path longer than p that contains each edge at most once. Either way, we contradict the assumption that p is a longest path containing each edge at most once, so that p must be an Euler tour.

Here is the constructive proof. 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 suffices. The output of this algorithm is a doubly linked list T of vertices which, read in list order, 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.

The algorithm also maintains 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 $v \in G.V$, NIL)
  while $L$ is not empty
      remove ($v$, location-in-T) from $L$
      $C$ = VISIT($G, L, v$)
      if location-in-T == NIL
          $T = C$
      else splice $C$ into $T$ just before location-in-T
  return $T$

VISIT($G, L, v$)
  $C$ = empty sequence of vertices
  $u = v$
  while out-degree($u$) > 0
      let $w$ be the first vertex in $G.Adj[u]$
      remove $w$ from $G.Adj[u]$, decrementing out-degree($u$)
      add $u$ onto the end of $C$
      if out-degree($u$) > 0
          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 that EULER-TOUR removes a vertex v from the list L, it calls VISIT(G, L, 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 (u, w) will not be visited 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 each edge is removed 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.

Solution to Problem 20-4

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 $v.d$ that the depth-first search started from v, it would have already discovered w (because w is somewhere in a depth-first tree with root r such that $L(r) \leq L(w) < L(v)$), so that $w.d < v.d$. By the parenthesis theorem, either the intervals $[v.d, v.f]$, and $[w.d, w.f]$ are disjoint and neither v nor w is a descendant of the other, or we have the ordering $w.d < v.d < v.f < w.f$ 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 $w.d < v.d$, we must have $w.d < w.f < v.d < v.f$. In this case, since u is reachable from w in G^{T} (because $w \in R(u)$), the depth-first search of G^{T} would have discovered u by the time $w.f$, so that $u.d < w.f$. Since u was discovered during a search that started at v, we have $v.d \leq u.d$. Thus, $v.d \leq u.d < w.f < v.d$, which is a contradiction. We conclude that no such vertex w can exist.

Solution to Problem 20-5

Use an adjacency list, but augmented to store two additional attributes per vertex: $v.number$ is the order in which vertex v was inserted, starting from 1, and $v.newest = \max\{u.number : (u, v) \in E\}$ is the *number* value of v's newest neighbor (NIL if v has no neighbors).

To implement NEWEST-NEIGHBOR(G, v), just return $v.newest$, taking $O(1)$ actual time. To implement INSERT$(G, v, neighbors)$, keep a global counter *next-number* of the number of the next vertex to be inserted, initialized to 1. When inserting a new vertex v, add v to the adjacency lists as usual, set $v.number = next\text{-}number$, increment *next-number*, set $u.newest = v.number$ for each vertex u in the *neighbors* array, and set $v.newest$ to the maximum *number* value of any of the vertices in the *neighbors* array (or to NIL if the *neighbors* array is empty). If *neighbors* contains k vertices, then the actual cost of INSERT is $k + 1$: a cost of 1 to insert v, plus a cost of k to update the *newest* value and query the *number* value of each vertex in *neighbors*.

To analyze the amortized cost of INSERT, use the potential function $\Phi(G) = 3|V| - |E|$. This value is initially 0, and because $|E| < 3|V|$, it can never go negative and is a valid potential function. Let G_i be the graph after the ith operation. If the ith operation is NEWEST-NEIGHBOR, then $c_i = O(1)$ and $G_i = G_{i-1}$, so that $\Phi(G_i) = \Phi(G_{i-1})$. The amortized cost is then

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(G_i) - \Phi(G_{i-1}) \\ &= c_i \\ &= O(1) .\end{aligned}$$

If the ith operation is INSERT and the *neighbor* array consists of k vertices, then $|E_i| = |E_{i-1}| + k$ and the amortized cost is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(G_i) - \Phi(G_{i-1}) \\ &= (k + 1) + (3|V_i| - |E_i|) - (3|V_{i-1}| - |E_{i-1}|) \\ &= (k + 1) + (3(|V_{i-1}| + 1) - (|E_{i-1}| + k)) - |E_{i-1}|) \\ &= (k + 1) + 3 - k \\ &= 4 .\end{aligned}$$

Lecture Notes for Chapter 21: Minimum Spanning Trees

Chapter 21 overview

Problem

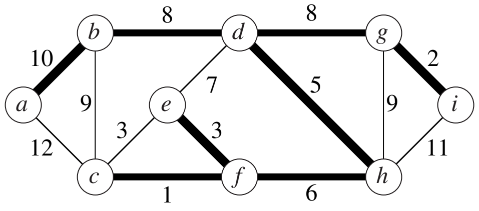
- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that
  1. everyone stays connected: can reach every house from all other houses, and
  2. total 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 *[Differs from Figure 21.1 in the textbook. Edges in the MST are drawn with heavy lines.]* :



In this example, there is more than one MST. Replace edge (e, f) in the MST 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

- Build a set A of edges.
- Initially, A has no edges.
- As edges are added 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 add only safe edges.

Generic MST algorithm

```
GENERIC-MST(G, w)
  $A = \emptyset$
  while $A$ does not form a spanning tree
      find an edge $(u, v)$ that is safe for $A$
      $A = A \cup \{(u, v)\}$
  return $A$
```

Use the loop invariant to show that this generic algorithm works.

Initialization: The empty set trivially satisfies the loop invariant.

Maintenance: Since only safe edges are added, A remains a subset of some MST.

Termination: The loop must terminate by the time it considers all edges. All edges added to A are in an MST, so upon termination. A is a spanning tree that is also an MST.

Finding a safe edge

How to 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 proper subset 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

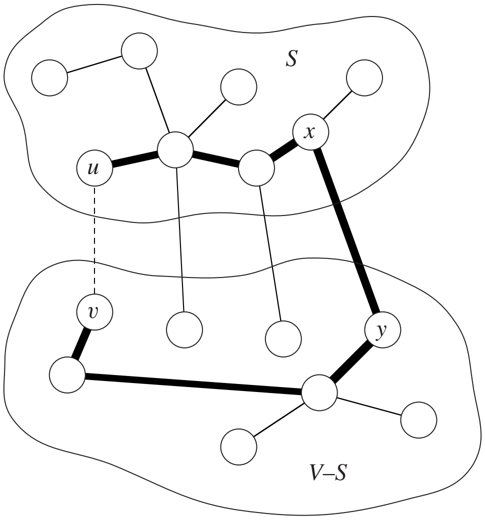
Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

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). 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. Edges with heavy lines 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.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\le w(T) , \end{aligned}$$

since $w(u, v) \le w(x, y)$. Since T' is a spanning tree, $w(T') \le w(T)$, 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 idea naturally leads to the algorithm known as Kruskal's algorithm to solve the minimum-spanning-tree problem.

Kruskal's algorithm

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \to \mathbb{R}$.

- 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.

```
MST-KRUSKAL$(G, w)$
  $A = \emptyset$
  for each vertex $v \in G.V$
      MAKE-SET$(v)$
  create a single list of the edges in $G.E$
  sort the list of edges into nondecreasing order by weight $w$
  for each edge $(u, v)$ taken from the sorted list in order
      if FIND-SET$(u) \neq$ FIND-SET$(v)$
          $A = A \cup \{(u, v)\}$
          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, there is only one component, so that all other edges will be rejected. *[Could add a test to the main loop of* KRUSKAL *to stop once* $|V| - 1$ *edges have been added to* A*.]*

Get the heavy edges shown in the figure.

Suppose (c, e) had been examined *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 19, that uses union by rank and path compression:

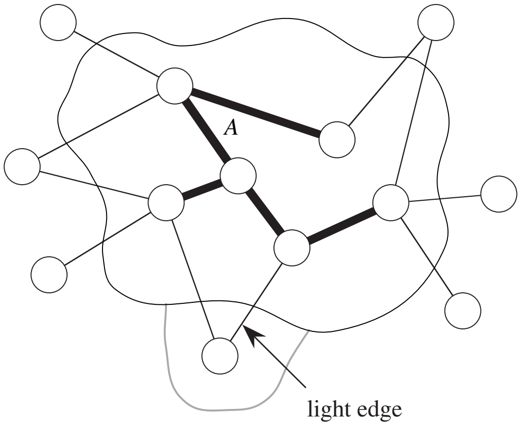
  $$O((V + E)\,\alpha(V)) + O(E \lg E) .$$

- 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 $O(E \lg E)$.

- $|E| \le |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.
- 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 connecting A to an isolated vertex. Such an edge must be safe for A. Add this edge to A.



[Edges of A are drawn with heavy lines.]

How to find the light edge quickly?

Use a priority queue Q:

- Each object is a vertex *not* in A.
- $v.key$ is the minimum weight of any edge connecting v to a vertex in A. $v.key = \infty$ if no such edge.
- $v.\pi$ is v's parent in A.
- Maintain A implicitly as $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
- At completion, Q is empty and the minimum spanning tree is $A = \{(v, v.\pi) : v \in V - \{r\}\}$.

```
MST-PRIM($G, w, r$)
  for each vertex $u \in G.V$
      $u.key = \infty$
      $u.\pi = \text{NIL}$
  $r.key = 0$
  $Q = \emptyset$
  for each vertex $u \in G.V$
      INSERT($Q, u$)
  while $Q \neq \emptyset$
      $u$ = EXTRACT-MIN($Q$)          // add $u$ to the tree
      for each vertex $v$ in $G.Adj[u]$   // update keys of $u$'s non-tree neighbors
          if $v \in Q$ and $w(u, v) < v.key$
              $v.\pi = u$
              $v.key = w(u, v)$
              DECREASE-KEY($Q, v, w(u, v)$)
```

Loop invariant: Prior to each iteration of the **while** loop,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq \text{NIL}$, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

Do example from the graph on page 21-1. *[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 DECREASE-KEY could take $O(1)$ *amortized* time.

  Then $\leq |E|$ DECREASE-KEY calls take $O(E)$ time altogether $\Rightarrow$ total time becomes $O(V \lg V + E)$.

  In fact, there is a way to perform DECREASE-KEY in $O(1)$ amortized time: Fibonacci heaps, mentioned in the introduction to Part V.

Solutions for Chapter 21: Minimum Spanning Trees

Solution to Exercise 21.1-1

This solution is also posted publicly

Theorem 21.1 shows this.

Let A be the empty set and S be any set containing u but not v.

Solution to Exercise 21.1-2

Using the graph of Figure 21.1, and let $S = \{a, b, c, h, i\}$, $V - S = \{d, e, f, g\}$, and $A = \{(a, b), (b, c), (c, i)\}$, so that the cut $(S, V - S)$ respects A. Edges (c, d), (c, f), (h, g), and (i, g) all cross the cut, with edge (h, g) the light edge for this cut. Yet, edges (c, d) and (c, f) are in the minimum spanning tree, so that they are safe for A, but are not light edges for this cut.

Solution to Exercise 21.1-3

Let T be a minimum spanning tree containing edge (u, v). Let $T' = T - \{(u, v)\}$ be T with edge (u, v) removed, and define the cut $(S, V - S)$ such that

$$\begin{aligned} S &= \{x \in V : \text{there is a path } u \leadsto x \text{ in } T'\} \ , \\ V - S &= \{x \in V : \text{there is a path } v \leadsto x \text{ in } T'\} \ . \end{aligned}$$

Let (y, z) be a light edge crossing this cut, so that $w(y, z) \leq w(u, v)$, and define the spanning tree $T'' = T' \cup \{(y, z)\}$. Because $w(y, z) \leq w(u, v)$, we have that $w(T'') \leq w(T)$. Since T is a minimum spanning tree, we also have $w(T) \leq w(T'')$, which implies that $w(T'') = w(T)$ and hence $w(y, z) = w(u, v)$. Therefore, (u, v) is a light edge for the cut $(S, V - S)$.

Solution to Exercise 21.1-4
This solution is also posted publicly

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 a cycle, so it is not a minimum spanning tree.

Solution to Exercise 21.1-5

Let T be a minimum spanning tree for G. If T does not contain e, then we are done.

So now suppose that T contains e. We will construct another minimum spanning tree that does not contain e. Let $e = (u, v)$ and let $T' = T - \{(u, v)\}$ be T with edge (u, v) removed. Define the cut $(S, V - S)$ such that

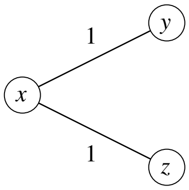
$$\begin{aligned} S &= \{x \in V : \text{there is a path } u \rightsquigarrow x \text{ in } T'\} \ , \\ V - S &= \{x \in V : \text{there is a path } v \rightsquigarrow x \text{ in } T'\} \ . \end{aligned}$$

Because e is on a cycle, some other edge e' in the cycle crosses the cut $(S, V - S)$, and by the definition of e, we have $w(e') \le w(e)$. Construct the tree $T'' = T' \cup \{e'\}$. Tree T'' is a spanning tree for G with weight $w(T'') = w(T') + w(e') = w(T) - w(e) + w(e') \le w(T)$. Since we assume that T is a minimum spanning tree for G, T'' must be one as well, and it does not include e.

Solution to Exercise 21.1-6
This solution is also posted publicly

Suppose that for every cut of G, there is a unique light edge crossing the cut. Let us consider two distinct minimum spanning trees, T and T', of G. Because T and T' are distinct, T contains some edge (u, v) that is not in T'. If (u, v) is removed 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 21.1-3) and, by our assumption, it's the only light edge crossing this cut. Because (u, v) is the only light edge crossing $(S, V - S)$ and (u, v) is not in T', each edge in T' that crosses $(S, V - S)$ must have weight strictly greater than $w(u, v)$. As in the proof of Theorem 21.1, we can identify the unique edge (x, y) in T' that crosses $(S, V - S)$ and lies on the cycle that results if we add (u, v) to T'. By our assumption, we know that $w(u, v) < w(x, y)$. Then, we can then remove (x, y) from T' and replace it by (u, v), giving a spanning tree with weight strictly less than $w(T')$. Thus, T' was not a minimum spanning tree, contradicting the assumption that the graph had two unique minimum spanning trees.

Here's a counterexample for the converse:

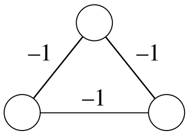


Here, the graph is its own minimum spanning tree, and so the minimum spanning tree is unique. Consider the cut $(\{x\}, \{y, z\})$. Both of the edges (x, y) and (x, z) are light edges crossing the cut, and they are both light edges.

Solution to Exercise 21.1-7

Let $A \subseteq E$ be a minimum-weight set of edges that connects all the vertices, but is not a tree. Then, A contains a cycle. Choose some edge e on the cycle and consider the set $A' = A - \{e\}$. Since $w(e) > 0$, we have $w(A') < w(A)$. But A' still connects all the vertices and has weight less than A, contradicting the assumption that A is a minimum-weight set of edges that connects all the vertices.

If weights may be nonpositive, consider the following graph, where all three edges have weight -1:



Then E is the minimum-weight set of edges that connects all the vertices, but it is not a tree.

Solution to Exercise 21.1-9

Because T' is a subgraph of T and T is a tree, T' must be a forest. Moreover, because T' is connected and is the subgraph induced by V', it must be a spanning tree of G'.

To see that T' is minimum spanning tree of G', suppose that G' has a spanning tree S such that $w(S) < w(T')$. Let $\widehat{T} = T - T'$ be the edges in T that are not in T', so that $T = \widehat{T} \cup T'$, and let $T'' = \widehat{T} \cup S$. We claim that T'' is a spanning tree of G with $w(T'') < w(T)$, which contradicts the assumption that T is a minimum spanning tree of G.

We first show that $w(T'') < w(T)$. Just as the edges in T' are disjoint from the edges in $\widehat{T}$, so are the edges in S (or any spanning tree of G'). Thus, we have

$$\begin{aligned} w(T'') &= w(\widehat{T} \cup S) \\ &= w(\widehat{T}) + w(S) \\ &< w(\widehat{T}) + w(T') \\ &= w(\widehat{T} \cup T') \\ &= w(T) . \end{aligned}$$

To see that T'' is a spanning tree of G, we show that T'' is acyclic and that $|T''| = |T|$. The latter property follows easily, since both T' and S are spanning trees of G', so that $|T'| = |S|$, and

$$\begin{aligned}
|T''| &= \left|\widehat{T} \cup S\right| \\
&= \left|\widehat{T}\right| + |S| \\
&= \left|\widehat{T}\right| + |T'| \\
&= \left|\widehat{T} \cup T'\right| \\
&= |T| \ .
\end{aligned}$$

To see that T'' is acyclic, suppose that it has a cycle. That cycle must include edges from both $\widehat{T}$ and S, since each of these sets are, on their own, acyclic. Since the cycle includes at least one edge from S, it must include two vertices $u, v \in V'$. Because S is a tree connecting u and v, there is a unique simple path $u \leadsto v$ in S. Similarly, there is a unique simple path $u \leadsto v$ in T'. Adding the edges in $\widehat{T}$ to the edges in T' creates a cycle in the resulting set T of edges, contradicting the assumption that T is a tree. Thus, T'' is acyclic and thus a spanning tree of G with weight less than $w(T)$, contradicting the assumption that T is a minimum spanning tree.

Solution to Exercise 21.1-10

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 21.2-1

When sorting and edges have equal weight, put the edges in T before the edges not in T.

Solution to Exercise 21.2-2

Store the attribute $v.key$ as in MST-PRIM or in an array indexed by the vertex number. (Since the graph is represented by an adjacency matrix, a number from 1 to $|V|$ identifies each vertex.) The call to EXTRACT-MIN in line 9 of MST-PRIM just goes through all the vertices to find the minimum *key* value, taking $O(V)$ time. The **for** loop of lines 10–14 iterates $|V|$ times per vertex, for a total of $|V|^2$ iterations. Each call of DECREASE-KEY in line 14 takes constant time. The entire procedure, therefore, runs in $O(V^2)$ time.

Solution to Exercise 21.2-3

Using a binary-heap will give a total running time of $O(E \lg V + V \lg V)$ (due to $V - 1$ EXTRACT-MIN and $O(E)$ DECREASE-KEY operations) as compared with the $O(E + V \lg V)$ time for Fibonacci heaps. If $E = \Theta(V)$, the two are asymptotically the same, but if $E = \Theta(V^2)$, the Fibonacci-heap implementation is asymptotically faster. The Fibonacci-heap implementation is faster if $|E| = \omega(V)$, because the binary-heap implementation runs in $O(\omega(V) \lg V)$ time and the Fibonacci-heap version runs in $O(\omega(V) + V \lg V)$ time.

Solution to Exercise 21.2-4

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 all of the edge weights in the graph are integers in the range from 1 to $|V|$, then counting sort can sort the edges in $O(V + E)$ time. Since the graph is connected, $|V| = O(E)$, and so the sorting time is just $O(E)$. This change 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 sorting step uses the weight values.

If the edge weights are integers in the range from 1 to W for some constant W, then counting sort could sort the edges more quickly. Sorting would then take $O(E + W) = O(E)$ time, since W is a constant. As in the first part, we get a total running time of $O(E\,\alpha(V))$.

Kruskal's algorithm can avoid sorting altogether if edge weights are integers in the range 1 to W, where W is a constant. Instead of sorting, create W buckets, and put an edge in bucket k if the edge's weight is k. The **for** loop in lines 6–9 goes through the edges by making a single pass through the buckets, from bucket 1 to bucket W, examining the edges (if any) in each bucket. The loop would have to examine each bucket and each edge, so that the running time would still be $O(E + W + E\,\alpha(V)) = O(E\,\alpha(V))$.

Solution to Exercise 21.2-5

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.

When the edge weights are integers in the range 1 to W, where W is a constant, the running time can be improved by implementing the queue as an array $Q[1 : W + 1]$

(using the $W+1$ slot for keys with value ∞), where slot k holds a doubly linked list of vertices with key k. 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). The total running time reduces to $O(E)$.

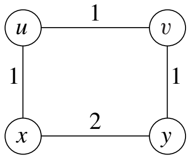
If the range of integer edge weights is instead 1 to $|V|$, then EXTRACT-MIN takes $O(V)$ time with this data structure. The total time spent in EXTRACT-MIN calls becomes $O(V^2)$, slowing the algorithm to $O(E + V^2) = O(V^2)$. In this case, it is better to use the Fibonacci-heap min-priority queue, which gave the $O(E+V \lg V)$ time.

Other data structures yield better running times:

- van Emde Boas trees (see the introduction to Part V) give 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 22) gives an upper bound of $O\left(E + V\sqrt{\lg V}\right)$ for Prim's algorithm.

Solution to Exercise 21.2-6

The algorithm does not work. Consider this graph:



Let $V_1 = \{u, v\}$ and $V_2 = \{x, y\}$. The minimum spanning trees for V_1 and V_2 have weights 1 and 2, respectively. The minimum-weight edge that crosses the cut has weight 1, for a total weight of 4 for the spanning tree $\{(u, v), (x, y), (u, x)\}$. But the spanning tree $\{(u, v), (u, x), (v, y)\}$ has weight 3.

Solution to Exercise 21.2-7

If the edge weights are uniformly distributed over $[0, 1)$, then by using BUCKET-SORT (Section 8.4) with $|E|$ buckets, the *expected* sorting time in Kruskal's algorithm goes down to $O(E)$, giving a total expected running time of $O(E + V + E\,\alpha(V)) = O(E\,\alpha(V))$.

If we use b buckets to maintain the min-priority queue in Prim's algorithm, then when the min-priority queue contains $|V|$ vertices, the expected bucket occupancy is $|V|/b$, so that the expected time to find the vertex with the minimum key is $O(b + |V|/b)$ ($O(b)$ to scan for a nonempty bucket and $O(|V|/b)$ expected time to find the minimum key in the first nonempty bucket). This quantity is minimized when $b = \sqrt{V}$, giving $O(\sqrt{V})$. As Prim's algorithm progresses, however, the key of each vertex monotonically decreases, so that vertices move to lower-indexed buckets. Then again, the vertices extracted from the min-priority queue tend to come from lower-indexed buckets as well. If we just use $O(\sqrt{V})$ as the expected time for EXTRACT-MIN, then the expected time for MST-PRIM becomes $O(V\sqrt{V} + E)$, which beats Kruskal's algorithm if $|E| = O(V\sqrt{V})$.

Solution to Exercise 21.2-8

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 21.2-1, 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)

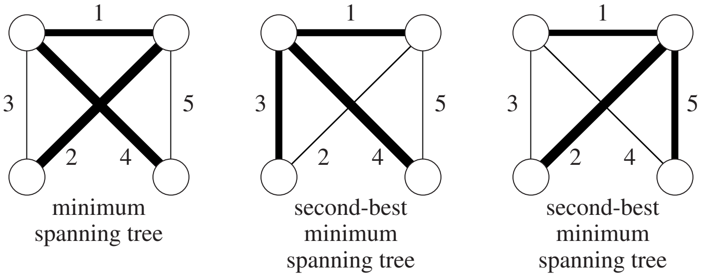
We use this lemma as follows. Let $G' = (V', E')$ be the graph $G = (V, E)$ with the one new vertex and its incident edges added. Suppose that we have a minimum spanning tree T for G. We compute a minimum spanning tree for G' by creating the graph $G'' = (V', E'')$, where E'' consists of the edges of 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 min-priority queue. Since $|V'| = |V| + 1$ and $|E''| \leq 2|V| - 1$ (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 min-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 21-1

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 21.1-6, 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:


1
3
5
2
4
minimum spanning tree
1
3
5
2
4
second-best minimum spanning tree
1
3
5
2
4
second-best minimum spanning tree

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 20 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 20.3. 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$(G, T, w)$
  let $max$ be a new table with an entry $max[u, v]$ for each $u, v \in G.V$
  for each vertex $u \in G.V$
      for each vertex $v \in G.V$
          $max[u, v] = \text{NIL}$
      $Q = \emptyset$
      ENQUEUE$(Q, u)$
      while $Q \neq \emptyset$
          $x = $ DEQUEUE$(Q)$
          for each vertex $v$ in $G.Adj[x]$
              if $max[u, v] == \text{NIL}$ and $v \neq u$
                  if $x == u$ or $w(x, v) > w(max[u, x])$
                      $max[u, v] = (x, v)$
                  else $max[u, v] = max[u, x]$
                  ENQUEUE$(Q, v)$
  return $max$
```

Here's the depth-first search approach:

```
DFS-FILL-MAX$(G, T, w)$
  let $max$ be a new table with an entry $max[u, v]$ for each $u, v \in G.V$
  for each vertex $u \in G.V$
      for each vertex $v \in G.V$
          $max[u, v] = \text{NIL}$
      DFS-FILL-MAX-VISIT$(G, u, u, max)$
  return $max$
```

```
DFS-FILL-MAX-VISIT$(G, u, x, max)$
  for each vertex $v$ in $G.Adj[x]$
      if $max[u, v]$ == NIL and $v \neq u$
          if $x == u$ or $w(x, v) > w(max[u, x])$
              $max[u, v] = (x, v)$
          else $max[u, v] = max[u, x]$
          DFS-FILL-MAX-VISIT$(G, u, v, max)$
```

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 $w(u, v) - w(max[u, v])$ 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 min-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 $w(u, v) - w(max[u, v])$. 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 $O(V^2)$.

Solution to Problem 21-3

a. The tree T that MAYBE-MST-A returns is a minimum spanning tree.

To see that T is a spanning tree, note that it consists only of edges that, if not included, result in a disconnected graph. By Theorem B.2, T is a tree. Since no part of it disconnects any vertex in G, it must be a spanning tree.

To see why T is a *minimum* spanning tree, consider each edge $e \in T$. At the time the **for** loop considers e, suppose that e had *not* been left in T. Then T would be disconnected, so that we could define a cut $(S, V - S)$, where S consists of the vertices in one side of the disconnected T, and $V - S$ consists of the vertices in the other side. All other edges in E with weights greater than or equal to $w(e)$ and that cross the cut $(S, V - S)$ have already been considered and removed from T, for otherwise $T - \{e\}$ would be connected. Therefore, e is a light edge crossing the cut $(S, V - S)$. Letting A be the edges within S and within $V - S$ that are used in a minimum spanning tree, and e being a light edge crossing the cut, Theorem 21.1 says that e is safe to add to the minimum spanning tree.

Implementation: Sort the edges by weight in $O(E \lg E) = O(E \lg V)$ time. Represent T using adjacency lists, so that initially, T is a copy of G. To check whether $T - \{e\}$ is connected, run breadth-first search and see whether any vertices have an infinite shortest-path distance, taking $O(V + E) = O(E)$ time ($O(E)$ because G being connected implies that $|E| \geq |V| - 1$). Sorting happens just once, breadth-first search is run $O(E)$ times, and edges are removed from T at total of $O(E)$ times. Therefore, the total running time is $O(E \lg V + E^2 + E) = O(E^2)$.

b. MAYBE-MST-B does not necessarily produce a minimum spanning tree. For a counterexample, let $G = (V, E)$ be a complete graph on three vertices, where $V = \{x, y, z\}$ and $E = \{(x, y), (y, z), (x, z)\}$. Let $w(x, y) = w(y, z) = 1$ and $w(x, z) = 2$. If MAYBE-MST-B takes edge (x, z) first, it adds it to T, and T will contain either $\{(x, y), (x, z)\}$ or $\{(y, z), (x, z)\}$. Either way, $w(T) = 3$, which is more than the weight 2 of the minimum spanning tree, which has edges $\{(x, y), (y, z)\}$.

Implementation: MAYBE-MST-B can be implemented in a similar manner to Kruskal's algorithm. Maintain disjoint sets of vertices. To check for a cycle, check whether the endpoints of edge e are in the same set by calling FIND-SET on each endpoint and see whether the results are equal. When adding e to T, also perform a UNION on the sets containing e's endpoints. There are $|V|$ MAKE-SET operations, $2|E|$ FIND-SET operations, and $|V| - 1$ UNION operations. Using the disjoint-set forest representation with the path-compression and union-by-rank heuristics, the running time is $O(E\,\alpha(V))$.

c. MAYBE-MST-C returns a minimum spanning tree. We will use a loop invariant to prove it. Note that as the algorithm progresses, the set T of edges is a forest and not necessarily a single tree. Suppose that the edges are considered in the order $e_1, e_2, \ldots, e_{|E|}$. Let $E_k = \{e_1, e_2, \ldots, e_k\}$ be the first k edges considered, where $E_0 = \emptyset$, and let V_k be the set of vertices that any edge in E_k is incident on, so that $V_0 = \emptyset$. Let $G_k = (V_k, E_k)$.

> **Loop invariant:**
>
> After considering the first k edges, T is a minimum spanning forest for the graph G_k.

Initialization: Initially, G_0 has no edges and T is empty.

Maintenance: Assume that after considering the first k edges, T is a minimum spanning forest for G_k. The next edge considered is $e_{k+1} = (u, v)$ for some vertices $u, v \in V$. There are three cases to consider.

- Case 1: $u, v \notin V_k$. Then e_k forms its own tree in the forest T, spanning u and v, which have not been seen before. We have $V_{k+1} = V_k \cup \{u, v\}$

and $E_{k+1} = E_k \cup e_{k+1}$, and T is a minimum spanning forest for $G_k = (V_k, E_k)$.

- Case 2: Exactly one of u and v is in V_k. Assume without loss of generality that $u \in V_k$ and $v \notin V_k$. Then, since edge e_{k+1} newly connects v to some tree in T, this edge cannot cause a cycle to occur. We have $V_{k+1} = V_k \cup \{v\}$ and $E_{k+1} = E_k \cup e_{k+1}$, and T is a minimum spanning forest for $G_k = (V_k, E_k)$.
- Case 3: $u, v \in V_k$. Since T was a spanning forest before adding edge e_{k+1}, either e_{k+1} connects two distinct trees in T, or it creates a cycle. In the former case, e_{k+1} must be the first edge considered that connects these two trees, for otherwise some other edge connecting them would have been considered and added to T. By Corollary 21.2, e_{k+1} can be added to T. In the latter case, adding e_{k+1} into T creates a cycle within some tree in T that was a minimum spanning tree for its component of G_k. Removing any edge from that cycle will restore that tree being a spanning tree for that component, and by removing the maximum-weight edge, that tree is a minimum spanning tree for that component. Thus, T is a minimum spanning forest for $G_k = (V_k, E_k)$.

Termination: The procedure terminates once all $|E|$ edges have been examined. At that point, all vertices have been seen as endpoints of the edges, and so $G_{|E|} = G$. Therefore, T is a minimum spanning tree for $G_{|E|} = G$.

Implementation: Represent T using adjacency lists. In each iteration, determine whether adding an edge causes a cycle by running breadth-first search from one of the endpoints of the edge. If we explore an edge (x, y) from vertex x and find that $y.d \neq \infty$ and $y.\pi \neq x$, then there is a cycle containing vertex y. The breadth-first searches will all be in T which, because it is a tree, has at most $|V| - 1$ edges. Thus, each breadth-first search runs in $O(V)$ time. Adding an edge to T takes $O(1)$ time. Since there are $|E|$ edges considered, the running time is $O(VE)$.

Lecture Notes for Chapter 22: 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 $w : E \to \mathbb{R}$

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:

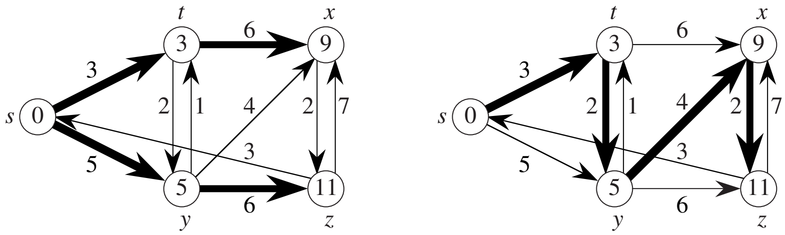
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there exists a path } u \leadsto v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Example

shortest paths from s

[δ values appear inside vertices. Heavy edges show shortest paths.]



This example shows that a 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, and
- 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

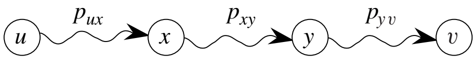
OK, as long as no negative-weight cycles are reachable from the source.

- 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.



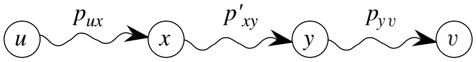
Suppose this path p is a shortest path from u to v.

Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \overset{p'_{xy}}{\leadsto} y$.

Then $w(p'_{xy}) < w(p_{xy})$.

Construct p':



Then

$$\begin{aligned} w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\ &< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\ &= w(p) . \end{aligned}$$

Contradicts the assumption that p is a shortest path. ■ (lemma)

Cycles

Shortest paths can't contain cycles:

- Already ruled out negative-weight cycles.
- Positive-weight ⇒ we can get a shorter path by omitting the cycle.
- 0-weight: no reason to use them ⇒ assume that our solutions won't use them.

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

- $v.d = \delta(s, v)$.
  - Initially, $v.d = \infty$.
  - Reduces as algorithms progress. But always maintain $v.d \geq \delta(s, v)$.
  - Call $v.d$ a ***shortest-path estimate***.
- $v.\pi$ = predecessor of v on a shortest path from s.
  - If no predecessor, $v.\pi$ = NIL.
  - π induces a tree—***shortest-path tree***.
  - We won't prove properties of π in lecture—see text.

Initialization

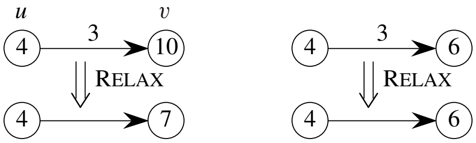
All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE.

```
INITIALIZE-SINGLE-SOURCE($G, s$)
  for each vertex $v \in G.V$
      $v.d = \infty$
      $v.\pi$ = NIL
  $s.d = 0$
```

Relaxing an edge (u, v)

Can the shortest-path estimate for v be improved by going through u and taking (u, v)?

```
RELAX($u, v, w$)
  if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$
      $v.\pi = u$
```



For all the single-source shortest-paths algorithms we'll look at,

- start by calling INITIALIZE-SINGLE-SOURCE,
- then relax edges.

The algorithms differ in the order and how many times they relax each edge.

Shortest-paths properties

[The textbook states these properties in the chapter introduction and proves them in a later section. You might elect to just state these properties at first and prove them later.]

Based on calling INITIALIZE-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 $s \leadsto v$ is $\leq$ weight of any path $s \leadsto v$. Path $s \leadsto u \to v$ is a path $s \leadsto v$, and if we use a shortest path $s \leadsto u$, its weight is $\delta(s, u) + w(u, v)$. ■

Upper-bound property: Always have $v.d \geq \delta(s, v)$ for all v. Once $v.d$ gets down to $\delta(s, v)$, it never changes.

Proof Initially true.

Suppose there exists a vertex such that $v.d < \delta(s, v)$.

Without loss of generality, v is first vertex for which this happens.

Let u be the vertex that causes $v.d$ to change.

Then $v.d = u.d + w(u, v)$.

So,

$$\begin{aligned} v.d &< \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) && \text{(triangle inequality)} \\ &\leq u.d + w(u, v) && \text{(v is first violation)} \\ \Rightarrow v.d &< u.d + w(u, v)\ . \end{aligned}$$

Contradicts $v.d = u.d + w(u, v)$.

Once $v.d$ 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 $v.d = \infty$ always.

Proof $v.d \geq \delta(s, v) = \infty \Rightarrow v.d = \infty$. ■

Convergence property: If $s \leadsto u \to v$ is a shortest path, $u.d = \delta(s, u)$, and edge (u, v) is relaxed, then $v.d = \delta(s, v)$ afterward.

Proof After relaxation:

$$\begin{aligned} v.d &\leq u.d + w(u, v) && (\text{RELAX code}) \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && (\text{lemma—optimal substructure}) \end{aligned}$$

Since $v.d \geq \delta(s, v)$, must have $v.d = \delta(s, v)$. ■

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 the edges of p are relaxed, *in the order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s, v_k)$.

Proof Induction to show that $v_i.d = \delta(s, v_i)$ after (v_{i-1}, v_i) is relaxed.

Basis: $i = 0$. Initially, $v_0.d = 0 = \delta(s, v_0) = \delta(s, s)$.

Inductive step: Assume $v_{i-1}.d = \delta(s, v_{i-1})$. Relax (v_{i-1}, v_i). By convergence property, $v_i.d = \delta(s, v_i)$ afterward and $v_i.d$ never changes. ■

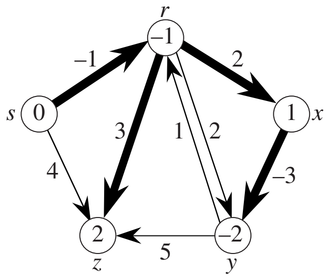
The Bellman-Ford algorithm

- Allows negative-weight edges.
- Computes $v.d$ and $v.\pi$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

```
BELLMAN-FORD($G, w, s$)
  INITIALIZE-SINGLE-SOURCE($G, s$)
  for $i = 1$ to $|G.V| - 1$
      for each edge $(u, v) \in G.E$
          RELAX($u, v, w$)
  for each edge $(u, v) \in G.E$
      if $v.d > u.d + w(u, v)$
          return FALSE
  return TRUE
```

Time: $O(V^2 + VE)$. The first **for** loop makes $|V| - 1$ passes over the edges, and each pass takes $\Theta(V + E)$ time. We use O rather than Θ because sometimes $< |V| - 1$ passes are enough (Exercise 22.1-3).

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 that $k \leq |V| - 1$.

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, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

How about the TRUE/FALSE return value?

- Suppose there is no negative-weight cycle reachable from s.

  At termination, for all $(u, v) \in E$,

  $$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \quad \text{(triangle inequality)} \\ &= u.d + w(u, v) . \end{aligned}$$

  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 $\displaystyle\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.

  Suppose (for contradiction) that BELLMAN-FORD returns TRUE.

  Then $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.

  Sum around c:

  $$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

Each vertex appears once in each summation $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d \Rightarrow$

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i) \ .$$

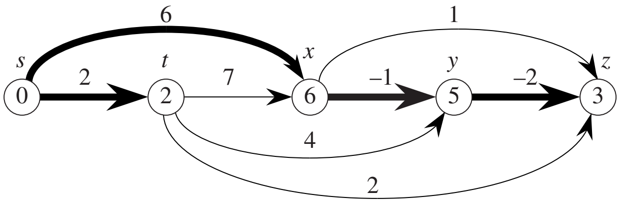
Contradicts c being a negative-weight cycle. ■

Single-source shortest paths in a directed acyclic graph

Since a dag, we're guaranteed no negative-weight cycles.

```
DAG-SHORTEST-PATHS$(G, w, s)$
  topologically sort the vertices of $G$
  INITIALIZE-SINGLE-SOURCE$(G, s)$
  for each vertex $u \in G.V$, taken in topologically sorted order
      for each vertex $v$ in $G.Adj[u]$
          RELAX$(u, v, w)$
```

Example



Time

$\Theta(V + E)$.

Correctness

Because vertices are processed in topologically sorted order, edges of *any* path must be relaxed in order of appearance in the path.
⇒ Edges on any shortest path are relaxed in order.
⇒ 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 ($v.d$).
- Can think of waves, like BFS.

- A wave emanates from the source.
- The first time that a wave arrives at a vertex, a new wave emanates from that vertex.
- The time it takes for the wave to arrive at a neighboring vertex equals the weight of the edge. (In BFS, each wave takes unit time to arrive at each neighbor.)

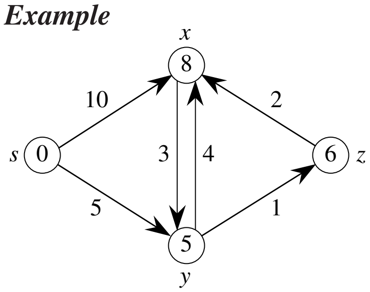
Have two sets of vertices:

- S = vertices whose final shortest-path weights are determined,
- Q = priority queue = $V - S$.

```
DIJKSTRA$(G, w, s)$
  INITIALIZE-SINGLE-SOURCE$(G, s)$
  $S = \emptyset$
  $Q = \emptyset$
  **for** each vertex $u \in G.V$
      INSERT$(Q, u)$
  **while** $Q \neq \emptyset$
      $u$ = EXTRACT-MIN$(Q)$
      $S = S \cup \{u\}$
      **for** each vertex $v$ in $G.Adj[u]$
          RELAX$(u, v, w)$
          **if** the call of RELAX decreased $v.d$
              DECREASE-KEY$(Q, v, v.d)$
```

- Looks a lot like Prim's algorithm, but computing $v.d$, 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

We will show that at the start of each iteration of the **while** loop, $v.d = \delta(s, v)$ for all $v \in S$. The algorithm terminates when $S = V$, so that $v.d = \delta(s, v)$ for all $v \in V$.

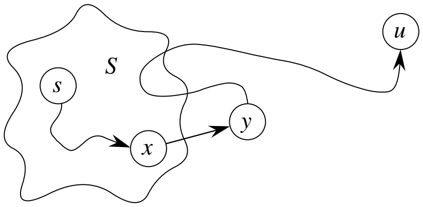
The proof is by induction on the number of iterations of the **while** loop, i.e., on $|S|$. The bases are for $|S| = 0$, so that $S = \emptyset$ and the claim is trivially true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = \delta(s, s) = 0$.

Inductive hypothesis: $v.d = \delta(s, v)$ for all $v \in S$.

Inductive step: The algorithm extracts vertex u from $V - S$. Because the algorithm adds u into S, we need to show that $u.d = \delta(s, u)$ at that time. If there is no path from s to u, then we are done, by the no-path property.

If there is a path from s to u:

- Let y be the first vertex on a shortest path from s to u that is *not* in S.
- Let $x \in S$ be the predecessor of y on that shortest path.
- Could have $y = u$ or $x = s$.



- y appears no later than u on the shortest path and all edge weights are nonnegative $\Rightarrow \delta(s, y) \le \delta(s, u)$.
- How we chose $u \Rightarrow u.d \le y.d$ at the time u is extracted from $V - S$.
- Upper-bound property $\Rightarrow \delta(s, u) \le u.d$.
- $x \in S \Rightarrow x.d = \delta(s, x)$. Edge (x, y) was relaxed when x was added into S. Convergence property $\Rightarrow$ set $y.d = \delta(s, y)$ at that time.
- Thus, we have $\delta(s, y) \le \delta(s, u) \le u.d \le y.d$ and $y.d = \delta(s, y) \Rightarrow \delta(s, y) = \delta(s, u) = u.d = y.d$.
- Hence, $u.d = \delta(s, u)$. Upper-bound property $\Rightarrow u.d$ doesn't change afterward. ■

Analysis

$|V|$ INSERT and EXTRACT-MIN operations.
$\le |E|$ DECREASE-KEY operations.

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 $\Theta(V)$ INSERT and EXTRACT-MIN operations, taking $O(\lg V)$ amortized time each.
  - Therefore, time is $O(V \lg V + E)$.

Difference constraints

Special case of linear programming.

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)

Example application

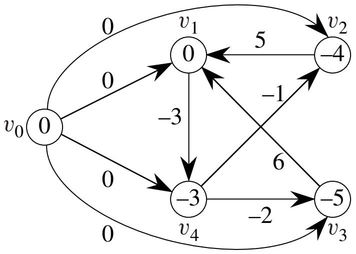
x_i are times when events are to occur.

- Suppose that event x_j must occur after event x_i occurs but no more than b_k time units after event x_i occurs. Get constraints $x_j - x_i \ge 0$, which is equivalent to $x_i - x_j \le 0$, and $x_j - x_i \le b_k$.
- What if x_j must occur at least b_k time units after x_i? Get $x_j - x_i \ge b_k$, which is equivalent to $x_i - x_j \le -b_k$.

Constraint graph

$G = (V, E)$, weighted, directed.

- $V = \{v_0, v_1, v_2, \ldots, v_n\}$: one vertex per variable $+ v_0$
- $E = \{(v_i, v_j) : x_j - x_i \le b_k \text{ is a constraint}\} \cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$
- $w(v_0, v_j) = 0$ for all $j = 1, 2, \ldots, n$
- $w(v_i, v_j) = b_k$ 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 $\Rightarrow x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)$ is a feasible solution.

   Need to show that $x_j - x_i \leq b_k$ 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,

   $$\begin{aligned} \delta(v_0, v_j) &\leq \delta(v_0, v_i) + w(v_i, v_j) \\ x_j &\leq x_i + b_k \\ x_j - x_i &\leq b_k\ . \end{aligned}$$

   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

   $$\begin{aligned} x_2 - x_1 &\leq w(v_1, v_2)\ , \\ x_3 - x_2 &\leq w(v_2, v_3)\ , \\ &\vdots \\ x_{k-1} - x_{k-2} &\leq w(v_{k-2}, v_{k-1})\ , \\ x_k - x_{k-1} &\leq w(v_{k-1}, v_k)\ . \end{aligned}$$

   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. BELLMAN-FORD returns FALSE ⇒ no feasible solution.

   BELLMAN-FORD returns TRUE ⇒ set $x_i = \delta(v_0, v_i)$ for all $i = 1, 2, \ldots, n$.

Solutions for Chapter 22: Single-Source Shortest Paths

Solution to Exercise 22.1-2

If there is a path from s to v, then some edge (u, v) must be relaxed, and this relaxation will set $v.d$ to a finite value. Conversely, if there is no path from s to v, then by the no-path property, $v.d = \infty$.

Solution to Exercise 22.1-3

This solution is also posted publicly

If the greatest number of edges on any shortest path from the source is m, then the path-relaxation property tells us that after m iterations of BELLMAN-FORD, every vertex v has achieved its shortest-path weight in $v.d$. By the upper-bound property, after m iterations, no d values will ever change. Therefore, no d values will change in the $(m + 1)$st iteration. Because we do not know m in advance, we cannot make the algorithm iterate exactly m times and then terminate. But if the algorithm just stops when nothing changes any more, it will stop after $m + 1$ iterations.

```
BELLMAN-FORD-EARLY-TERMINATION$(G, w, s)$
  INITIALIZE-SINGLE-SOURCE$(G, s)$
  repeat
      $changes$ = FALSE
      for each edge $(u, v) \in G.E$
          if RELAX$'(u, v, w)$
              $changes$ = TRUE
  until $changes$ == FALSE

RELAX$'(u, v, w)$
  if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$
      $v.\pi = u$
      return TRUE
  else return FALSE
```

Because the exercise specifies that G has no negative-weight cycles, the test for a negative-weight cycle (based on there being a d value that would change if another relaxation step was done) has been removed. If there were a negative-weight cycle, this version of the algorithm would never get out of the **repeat** loop because some d value would change in each iteration.

Solution to Exercise 22.1-4

If the **for** loop of lines 5–7 returns FALSE, it is because edge (u, v) is on a negative-weight cycle that is reachable from the source. Instead of returning FALSE, this loop builds a list of vertices known to be on such a negative-weight cycle. Once this list is built, we can run a DFS-like procedure to find all vertices reachable from vertices in this list. Instead of maintaining discovery and finishing times, we just use vertex colors (white and black, no need for gray) to keep track of whether a vertex has been visited.

```
BELLMAN-FORD-NEGATIVE-INFINITY$(G, w, s)$
  INITIALIZE-SINGLE-SOURCE$(G, s)$
  for $i = 1$ to $|G.V| - 1$
      for each edge $(u, v) \in G.E$
          RELAX$(u, v, w)$
  create an empty linked list $L$ of vertices
  for each edge $(u, v) \in G.E$
      if $v.d > u.d + w(u, v)$
          LIST-PREPEND$(L, u)$
  for each vertex $u \in G.V$
      $u.color$ = WHITE
  for each vertex $u$ in list $L$
      if $u.color$ == WHITE
          DFS-VISIT$'(G, u)$

DFS-VISIT$'(G, u)$
  $u.d = -\infty$
  for each vertex $v$ in $G.Adj[u]$
      if $v.color$ == WHITE
          DFS-VISIT$'(G, v)$
  $u.color$ = BLACK
```

Solution to Exercise 22.1-5

The **for** loops of lines 3–4 and 5–7 iterate through all the edges, so that they each take $O(E)$ time to go through all $|E|$ edges. The **for** loop of lines 2–4 then runs in $O(VE)$ time.

If the input graph is represented with adjacency lists, convert the adjacency lists to a single list of edges, taking $O(V + E)$ time, and then do as described above. The resulting procedure runs in $O(V + E + VE) = O(VE)$ time.

Solution to Exercise 22.1-6

Since there is always a path of weight 0 from every vertex to itself, we have $\delta^*(v) \le 0$ for all $v \in V$. Strictly speaking, the only change we need to make to the Bellman-Ford algorithm is to change the initial values in line 2 of INITIALIZE-SINGLE-SOURCE from ∞ to 0. At the completion of BELLMAN-FORD, we have $v.d = \delta^*(v)$ for all $v \in V$. After the ith iteration of the **for** loop of lines 2–4 of BELLMAN-FORD, all paths of at most i edges into each vertex have been relaxed. Since all simple paths have at most $|V| - 1$ edges, $|V| - 1$ iterations suffice to relax all the edges on every path into every vertex. Since we're not building a shortest-paths tree in this instance, the π attributes are meaningless.

Solution to Exercise 22.1-7

See the solution to part (b) of Problem 22-3.

Solution to Exercise 22.2-2

The last vertex in a topological sort of a dag has no edges leaving it. This vertex is the last one taken in the **for** loop of DAG-SHORTEST-PATHS, and so no d or π attributes change during the last iteration of the **for** loop.

Solution to Exercise 22.2-3

Instead of modifying the DAG-SHORTEST-PATHS procedure, we'll modify the structure of the graph so that we can run DAG-SHORTEST-PATHS on it. In fact, 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 $|V'| \le 2|V|$ and $|E'| \le |V| + |E|$. We can then run on G' the same algorithm to find a longest path through a dag as is given in Section 22.2 of the text.

The first way transforms each vertex $v \in V$ into two vertices v' and v'' in V'. All edges in E that enter v enter v' in E', and all edges in E that leave v 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.

The second way leaves vertices in V alone, but adds 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, $|V'| = |V| + 1$, $|E'| \leq |V| + |E|$ (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 22.2-4

To count all the paths in a dag, first topologically sort the dag. Let's add a vertex attribute $u.\mathit{paths}$ to count the paths that start at vertex u. If we have the values of $v.\mathit{paths}$ for all vertices v that are adjacent to vertex u, then we can compute

$$u.\mathit{paths} = 1 + \sum_{(u,v) \in E} v.\mathit{paths} .$$

The 1 in the formula comes from the 0-length path from u to itself, and the summation comes from all the paths that are reachable from vertex u. To compute the values of the *paths* attribute, process the vertices in reverse topologically sorted order. Once all the *paths* attributes have been computed, sum them up to get the total number of paths in the dag.

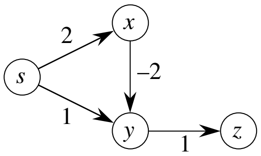
Here is pseudocode:

```
COUNT-PATHS($G$)
  topologically sort $G$
  for each vertex $u \in G.V$, taken in the reverse of topologically sorted order
      $u.paths = 1$
      for each vertex $v$ in $G.Adj[u]$
          $u.paths = u.paths + v.paths$
  $total\text{-}paths = 0$
  for each vertex $u \in G.V$
      $total\text{-}paths = total\text{-}paths + u.paths$
  return $total\text{-}paths$
```

This procedure takes $\Theta(V + E)$ time: $\Theta(V + E)$ time to topologically sort G, $\Theta(V + E)$ time to compute the *paths* attributes for all vertices, and $\Theta(V)$ time to sum the *paths* values.

Solution to Exercise 22.3-2

Dijkstra's algorithm produces an incorrect answer for $z.d$ in this graph:



Because edge (y, z) is relaxed when $y.d = 1$, before $y.d$ gets its ultimate value of 0, $z.d$ gets the value 2 instead of the correct value, 1.

The proof of Theorem 22.6 fails because with negative edge weights, we could have $\delta(s, y) > \delta(s, u)$ so that we do not get the key inequality $\delta(s, y) \le \delta(s, u) \le u.d \le y.d$.

Solution to Exercise 22.3-3

This solution is also posted publicly

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 $u.d = \delta(s, u) = \infty$. If u is reachable from s, then there is a shortest path $p = s \leadsto x \to u$. When the vertex x was extracted, $x.d = \delta(s, x)$ and then the edge (x, u) was relaxed; thus, $u.d = \delta(s, u)$.

Solution to Exercise 22.3-4

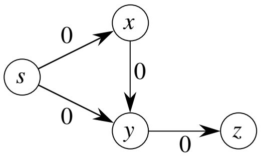
```
DIJKSTRA($G, w, s$)
  INITIALIZE-SINGLE-SOURCE($G, s$)
  $S = \emptyset$
  $Q = \emptyset$
  INSERT($Q, s$)
  while $Q \ne \emptyset$
      $u$ = EXTRACT-MIN($Q$)
      $S = S \cup \{u\}$
      for each vertex $v$ in $G.Adj[u]$
          if $v.d == \infty$
              INSERT($Q, v$)
          RELAX($u, v, w$)
          if the call of RELAX decreased $v.d$
              DECREASE-KEY($Q, v, v.d$)
```

Solution to Exercise 22.3-5

1. Verify that $s.d = 0$ and $s.\pi = \text{NIL}$.
2. Verify that $v.d = v.\pi.d + w(v.\pi, v)$ for all $v \ne s$.
3. Verify that $v.d = \infty$ if and only if $v.\pi = \text{NIL}$ for all $v \ne s$.
4. If any of the above verification tests fail, declare the output to be incorrect. Otherwise, run one pass of Bellman-Ford, i.e., relax each edge $(u, v) \in E$ one time. If any values of $v.d$ change, then declare the output to be incorrect; otherwise, declare the output to be correct.

Solution to Exercise 22.3-6

Consider this graph:



Dijkstra's algorithm could relax edges in the order (s, y), (s, x), (y, z), (x, y). The graph has two shortest paths from s to z: $\langle s, x, y, z \rangle$ and $\langle s, y, z \rangle$, both with weight 0. The edges on the shortest path $\langle s, x, y, z \rangle$ are relaxed out of order, because (x, y) is relaxed after (y, z).

Solution to Exercise 22.3-7
This solution is also posted publicly

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 $s \overset{p}{\leadsto} t$ such that $\prod_{(u,v) \in p} r(u, v)$ is maximized. This is equivalent to maximizing $\lg \left(\prod_{(u,v) \in p} r(u, v) \right) = \sum_{(u,v) \in p} \lg r(u, v)$, 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.

Alternative solution

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 make the following changes:

- In INITIALIZE-SINGLE-SOURCE, line 2 becomes
  $v.d = -\infty$
- RELAX becomes

```
RELAX$(u, v, r)$
  if $v.d < u.d \cdot r(u, v)$
      $v.d = u.d \cdot r(u, v)$
      $v.\pi = u$
```

- In DIJKSTRA, Q becomes a max-priority queue, line 7 becomes
  $u =$ EXTRACT-MAX(Q)
  and lines 11–12 become

```
  if the call of RELAX increased $v.d$
      INCREASE-KEY$(Q, v, v.d)$
```

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 22.3-9

Observe that if a shortest-path estimate is not ∞, then it's at most $(|V| - 1)W$. Why? In order to have $v.d < \infty$, the algorithm must have relaxed an edge (u, v) with $u.d < \infty$. By induction, we can show that if edge (u, v) is relaxed, then $v.d$ 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 $v.d \le (|V| - 1)W$. Note also that $v.d$ 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 the initial $|V|$ INSERT operations occur, no others ever happen. The only other way that a key value can change is by a DECREASE-KEY operation. Since edge weights are nonnegative, upon relaxing an edge (u, v), we have that $u.d \le v.d$. Since u is the minimum vertex that was just extracted, we know that any other vertex extracted later has a key value that is at least $u.d$.

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. 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. Implement each bucket by a circular, doubly linked list with a sentinel, so that a vertex can be inserted into or deleted from each bucket in $O(1)$ time. 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: Maintain an index *min* of the value of the smallest key extracted. Initially, *min* is 0. To find the smallest key, look in $A[\mathit{min}]$ 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

increment *min* until either finding 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. *min* is incremented at most k times, and some element is removed and returned 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, 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 m of operations 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 22.3-10

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 $v.d$ 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 $u.d \le y.d$ for all vertices $y \in V - S$. After relaxing edge (u, v), we have $v.d \le u.d + W$. Since any other vertex $y \in V - S$ with $y.d < \infty$ also had its estimate changed by a relaxation of some edge x with $x.d \le u.d$, we must have $y.d \le x.d + W \le u.d + W$. Thus, at the time of relaxing edges from a vertex u, we must have, for all vertices $v \in V - S$, that $u.d \le v.d \le u.d + W$ or $v.d = \infty$. Since shortest-path estimates are integer values (except for ∞), at any given moment we have at most $W + 2$ different ones: $u.d, u.d + 1, u.d + 2, \ldots, u.d + W$ 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, this array can be updated 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 22.3-11

Two of the properties that the proof of Theorem 22.6 relies on are that after the first iteration of the **while** loop, set S is nonempty and that $\delta(s, y) \leq \delta(s, u)$, where u is a vertex in $V - S$ with the smallest d value and y is the first vertex on a shortest path from s to u that is in $V - S$. Once the source s has been placed into set S and the edges leaving s—which may have negative weights—have been relaxed, all edges between vertices in $V - S$ have nonnegative weights, so that the key inequality $\delta(s, y) \leq \delta(s, u)$ still holds and the proof goes through.

Solution to Exercise 22.3-12

Implement the priority queue as an array $B[0 : (|V| - 1)C]$, where $B[i]$ is a bucket containing all vertices v such that $(i-1)C \leq v.d \leq iC$, plus a bucket for vertices v such that $v.d = \infty$. Each bucket can just maintain its vertices in a linked list.

The key idea is to modify the EXTRACT-MIN operation so that it returns any vertex u in the lowest-indexed nonempty bucket. Vertex u might not have the lowest d value of all vertices in $V - S$, but the algorithm still produces the correct shortest paths. To see why, let's think about the proof of Theorem 22.6 when there is a path from s to u. We denoted by y the first vertex in $V - S$ that is on a shortest path from s to u. Although there could be vertices in $V - S$ with d values smaller than $u.d$, we claim that none of them are on a shortest path from s to u. If this claim holds, then $y = u$, and the edge (x, u) was relaxed when y's predecessor x was added to S, so that $u.d = \delta(s, u)$.

To see why the claim is true, since all edge weights are at least C, if such a vertex $y \neq u$ exists, then since there is at least one edge of weight at least C between y and u on the shortest path from s, we must have $\delta(s, y) \leq \delta(s, u) - C$. But then if vertex u is in bucket $B[i]$, vertex y would have to be in bucket $B[j]$ for some $j < i$, and vertex u would not have been chosen by the (pseudo) EXTRACT-MIN call.

As in Exercise 22.3-9, the running time is $O(WV + E)$, where $W = 2C$. Since C is a constant, the running time comes to $O(V + E)$.

Solution to Exercise 22.4-3

No shortest-path weight from v_0 in a constraint graph can be positive. Because there is a 0-weight edge from v_0 to each of the other n vertices, the shortest-path weight from v_0 to any vertex must be at most 0.

Solution to Exercise 22.4-4

Let $\delta(u)$ be the shortest-path weight from s to u. Then we want to find $\delta(t)$.

δ must satisfy

$$\begin{aligned} \delta(s) &= 0 \\ \delta(v) - \delta(u) &\le w(u, v) \text{ for all } (u, v) \in E \qquad \text{(Lemma 22.10)} , \end{aligned}$$

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 additional maximization requirement does what we need 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 22.9:

  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 that

  $$\delta(t) = \sum_{(u,v) \in p} w(u, v) \ge \sum_{(u,v) \in p} (x_v - x_u) = x_t - x_s .$$

  But $x_s = 0$, so that $x_t \le \delta(t)$.

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 is a negative-weight cycle, the inequalities have no feasible solution (as demonstrated in the proof of Theorem 22.9); and if t is not reachable from s, then x_t is unbounded.

Solution to Exercise 22.4-5

The graph used to solve a system of difference constraints has $n + 1$ vertices and $n + m$ edges, giving a running time of $O((n + 1)(n + m)) = O(n^2 + nm)$ for the Bellman-Ford algorithm. After the first pass, relaxing any of the edges leaving v_0 cannot change any d values, so that these n edges need be relaxed only once, as the first n edges relaxed in the first pass. After the first pass, the remaining $n - 1$ passes relax only m edges, giving a total running time of $O(n + m + nm) = O(nm)$.

Solution to Exercise 22.4-6

For each equality constraint $x_j - x_i = b_k$, add two difference constraints:

$$\begin{aligned} x_j - x_i &\le b_k \ , \\ x_i - x_j &\le -b_k \ . \end{aligned}$$

The latter difference constraint is equivalent to $x_j - x_i \ge b_k$.

Solution to Exercise 22.4-7
This solution is also posted publicly

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 vertex but initializing all shortest path estimates to 0 instead of ∞.

Solution to Exercise 22.4-10

To allow for single-variable constraints, 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.

After finding shortest-path weights from v_0, set $x_i = \delta(v_0, v_i)$ for $i = 0, 1, \ldots, n$; 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 $x_i \le b_k$, 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 $-x_i \le b_k$, 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, $-x_i \le b_k$.

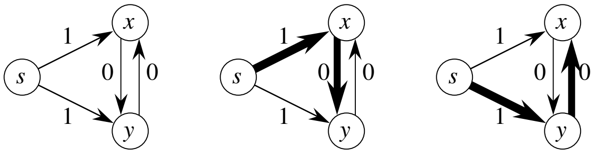
If $\delta(v_0, v_0) < 0$, so that $x_0 < 0$, then there is a negative-weight cycle containing v_0. The portion of the proof of Theorem 22.9 that deals with negative-weight cycles carries through but with v_0 on the negative-weight cycle, and we see that there is no feasible solution.

Solution to Exercise 22.4-11

In each difference constraint, replace b_k by $\lfloor b_k \rfloor$. Then solve the resulting system. After all, if x_j and x_i are integers such that $x_j - x_i \le b_k$, then $x_j - x_i$ is an integer value, and it is at most $\lfloor b_k \rfloor$.

Solution to Exercise 22.5-2

The graph below on the left has one shortest-paths tree, shown in the middle, and another, shown on the right. Edges (s, x) and (x, y) are in the tree in the middle but not on the right. Edges (s, y) and (y, x) are in the tree on the right but not the middle.



Solution to Exercise 22.5-3

If $\delta(s, u) = \infty$, then $\delta(s, v) \le \delta(s, u)$, so that $\delta(s, v) \le \delta(s, u) + w(u, v)$. If $\delta(s, v) = \infty$, then $\delta(s, u)$ must also equal ∞, because otherwise there would be a path $s \leadsto u \to v$ and $\delta(s, v)$ would be finite.

If $\delta(s, u) = -\infty$, then $\delta(s, u) + w(u, v) = -\infty$, so that there is a path $s \leadsto u \to v$ with weight $-\infty$. If $\delta(s, v) = -\infty$, then $\delta(s, v) \le \delta(s, u)$, so that $\delta(s, v) \le \delta(s, u) + w(u, v)$.

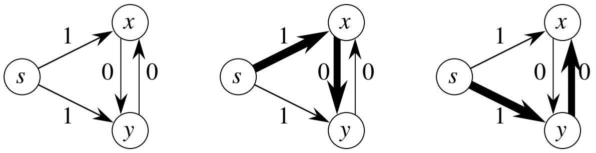
Solution to Exercise 22.5-4
This solution is also posted publicly

Whenever RELAX sets π for some vertex, it also reduces the vertex's d value. Thus if $s.\pi$ gets set to a non-NIL value, $s.d$ is reduced from its initial value of 0 to a negative number. But $s.d$ 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 22.5-5

[The same graph as in the solution to Exercise 22.5-2 works here.]

The graph below on the left has one shortest-paths tree, shown in the middle, and another, shown on the right. In the tree in the middle, $y.\pi = x$, and in the tree on the right, $x.\pi = y$. Thus, G_π has the cycle $\langle (x, y), (y, x) \rangle$.



Solution to Exercise 22.5-6

Suppose there is a path from s to every vertex in V_π. This is trivially true at the beginning with $V_\pi = \{s\}$. Now, if edge (u, v) is relaxed and v is added to V_π, then u must have been in V_π because u was reachable. Thus, there is a path from s to u to v in G_π. Thus, the inductive hypothesis is proven and the property is maintained as an invariant over any sequence of relaxations.

Solution to Exercise 22.5-7

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 $v.d = \delta(s, v)$ for all $v \in V$. Since G_π contains at most $|V| - 1$ edges, only $|V| - 1$ edges need to be relaxed to obtain $v.d = \delta(s, v)$ for all $v \in V$.

Solution to Exercise 22.5-8

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 $v.d = \infty$ 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, 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 an edge (u_{i-1}, u_i) or (v_{j-1}, v_j) is relaxed, for $i = 1, 2, \ldots, l$ and $j = 1, 2, \ldots, k-1$ (note that the last edge of cycle c has not yet been relaxed), $u_i.d$ or $v_j.d$ reduces 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 $v_j.d$.

Basis: The next edge relaxed after the opening sequence is (v_{k-1}, v_k). Before relaxation, $v_k.d = w(p)$, and after relaxation, $v_k.d = w(p) + w(c) < w(p)$, 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 $v_j.d$ was updated was by a relaxation of this same edge. By the inductive hypothesis, $v_{j-1}.d$ has just been reduced. Thus, $v_{j-1}.d + w(v_{j-1}, v_j) < v_j.d$, and so the relaxation will reduce the value of $v_j.d$.

Solution to Problem 22-1

a. Assume for the purpose 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.

The sequence $\langle v_1, v_2, \ldots, v_{|V|} \rangle$ 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 $v.d$ will be ∞. Thus, we need to consider only the case where $v.d$ 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 tjat 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 vertex 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 it performs only $\lceil |V|/2 \rceil$ passes instead of $|V| - 1$ passes, there are still $O(V)$ passes. Each pass still takes $\Theta(E)$ time, so that the running time remains $O(VE)$.

Solution to Problem 22-2

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, so that 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 (i.e., no box nests inside itself). 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 edges (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 22.2 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 22-3
This solution is also posted publicly

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 $|E| = n(n-1)$.)

We are looking for a cycle $\langle i_1, i_2, i_3, \ldots, i_k, i_1 \rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 .$$

Taking logarithms of both sides of this inequality gives

$$\lg R[i_1, i_2] + \lg R[i_2, i_3] + \cdots + \lg R[i_{k-1}, i_k] + \lg R[i_k, i_1] > 0 .$$

If we negate both sides, we get

$$(-\lg R[i_1, i_2]) + (-\lg R[i_2, i_3]) + \cdots \\ + (-\lg R[i_{k-1}, i_k]) + (-\lg R[i_k, i_1]) < 0 ,$$

and so we want to determine whether G contains a negative-weight cycle 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. Note: The solution to this part also serves as a solution to Exercise 22.1-7.

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. Go through the edges once again. Upon finding an edge (u, v) for which $u.d + w(u, v) < v.d$, we know that either vertex v is on a negative-weight cycle or is reachable from one. We can find a vertex on the negative-weight cycle by tracing back the π values from v, keeping track of which vertices we've visited until we reach a vertex x that we've visited before. Then we can trace back π values from x until we get back to x, and all vertices in between, along with x, will constitute a negative-weight cycle. We can use the recursive method given by the PRINT-PATH procedure of Section 20.2, but stop it when it returns to vertex x.

The running time is $O(n^3)$ to run BELLMAN-FORD, plus $O(m)$ to check all the edges and $O(n)$ to print the vertices of the cycle, for a total of $O(n^3)$ time.

Solution to Problem 22-4

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 $v.d = i$. Initialize $v.d$ for $v \neq s$ to $|E|+1$ instead of to ∞, so that all vertices have a place in Q. (Any initial $v.d > \delta(s, v)$ works in the algorithm, since $v.d$ 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 $v.d$ decreases, just add v to the front of the list in $Q[v.d]$.
- EXTRACT-MIN removes the head of the list in the first nonempty slot of Q. To perform EXTRACT-MIN without scanning all of Q, keep track of the smallest index j for which $Q[j]$ is not empty. The key point is that when $v.d$ decreases due to relaxation of edge (u, v), $v.d$ remains at least $u.d$, so that 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-MIN operations together.

b. For all $(u, v) \in E$, we have $w_1(u, v) \in \{0, 1\}$, so that $\delta_1(s, v) \leq |V| - 1 \leq |E|$. 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 that 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), and the third line follows from Lemma 22.10: $\delta_{i-1}(s,v)\leq\delta_{i-1}(s,u)+w_{i-1}(u,v)$.

e. Observe that if we compute $\widehat{w}_i(p)$ for any path $p:u\leadsto v$, 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 relationship will be shown in detail in equation (23.11) within the proof of Lemma 23.1.) 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

$$\begin{aligned}\widehat{\delta}_i(s,v) &= \delta_i(s,v)+2\delta_{i-1}(s,s)-2\delta_{i-1}(s,v)\\ &= \delta_i(s,v)-2\delta_{i-1}(s,v)\ .\end{aligned}$$

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 $\delta(s,v)=\delta_k(s,v)$ in time $O(Ek)=O(E\lg W)$.

Solution to Problem 22-6

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.) The path-relaxation property (Lemma 22.15) 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 23: All-Pairs Shortest Paths

Chapter 23 overview

Given a directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, $|V| = n$. Assume that the vertices are numbered $1, 2, \ldots, n$.

Goal: create an $n \times n$ matrix $D = (d_{ij})$ of shortest-path distances, so that $d_{ij} = \delta(i, j)$ for all vertices i and j.

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.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{weight of edge } (i, j) & \text{if } i \neq j, (i, j) \in E \ , \\ \infty & \text{if } i \neq j, (i, j) \notin E \ . \end{cases}$$

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 $l_{ij}^{(r)}$ = weight of shortest path $i \leadsto j$ that contains $\leq r$ edges.

- $r = 0$

  $\Rightarrow$ there is a shortest path $i \leadsto j$ with $\leq r$ edges if and only if $i = j$

  $$\Rightarrow l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \ , \\ \infty & \text{if } i \neq j \ . \end{cases}$$

- $r \geq 1$

  $$\begin{aligned} \Rightarrow l_{ij}^{(r)} &= \min\left\{l_{ij}^{(r-1)}, \ \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n\right\}\right\} \\ &\qquad\qquad (k \text{ ranges over all possible predecessors of } j) \\ &= \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n\right\} \qquad (\text{since } w_{jj} = 0 \text{ for all } j) \ . \end{aligned}$$

- Observe that when $r = 1$, must have $l_{ij}^{(1)} = w_{ij}$.

  Conceptually, when the path is restricted to at most 1 edge, the weight of the shortest path $i \leadsto j$ must be w_{ij}.

  And the math works out, too:

  $$\begin{aligned} l_{ij}^{(1)} &= \min\left\{l_{ik}^{(0)} + w_{kj} : 1 \leq k \leq n\right\} \\ &= l_{ii}^{(0)} + w_{ij} \qquad (l_{ii}^{(0)} \text{ is the only non-}\infty \text{ among } l_{ik}^{(0)}) \\ &= w_{ij} \ . \end{aligned}$$

All simple shortest paths contain $\leq n - 1$ edges
$\Rightarrow \delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots$

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 $L^{(r-1)}$ to $L^{(r)}$:

```
EXTEND-SHORTEST-PATHS($L^{(r-1)}, W, L^{(r)}, n$)
  // Assume that the elements of $L^{(r)}$ are initialized to $\infty$.
  for $i = 1$ to $n$
      for $j = 1$ to $n$
          for $k = 1$ to $n$
              $l_{ij}^{(r)} = \min\{l_{ij}^{(r)}, l_{ik}^{(r-1)} + w_{kj}\}$
```

Compute each $L^{(r)}$:

```
SLOW-APSP($W, L^{(0)}, n$)
  let $L = (l_{ij})$ and $M = (m_{ij})$ be new $n \times n$ matrices
  $L = L^{(0)}$
  for $r = 1$ to $n - 1$
      $M = \infty$     // initialize $M$
      EXTEND-SHORTEST-PATHS($L, W, M, n$)
      $L = M$
  return $L$
```

Time

- EXTEND-SHORTEST-PATHS: $\Theta(n^3)$.
- SLOW-ALL-APSP: $\Theta(n^4)$.

Observation

EXTEND-SHORTEST-PATHS is like matrix multiplication. Make the following substitutions in the equation $l_{ij}^{(r)} = \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \le k \le n\right\}$:

$l^{(r-1)} \rightarrow a$

$w \rightarrow b$

$l^{(r)} \rightarrow c$

$\min \rightarrow +$

$+ \rightarrow \cdot$

You get $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$, which is the equation for computing c_{ij} in matrix multiplication.

Making these changes to EXTEND-SHORTEST-PATHS and replacing ∞ (identity for min) with 0 (identity for $+$) gives a procedure for matrix multiplication:

```
for $i = 1$ to $n$
    for $j = 1$ to $n$
        for $k = 1$ to $n$
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
```

So, we can view EXTEND-SHORTEST-PATHS 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-SHORTEST-PATHS $n-1$ times).

Could compute $A, A^2, A^4, A^8, \ldots$

If we knew $A^r = A^{n-1}$ for all $r \ge n-1$, could just finish with A^r, where r is the smallest power of 2 that is $\ge n-1$ ($r = 2^{\lceil \lg(n-1) \rceil}$).

```
FASTER-ALL-PAIRS-SHORTEST-PATHS($W, n$)
  let $L$ and $M$ be new $n \times n$ matrices
  $L = W$
  $r = 1$
  while $r < n - 1$
      $M = \infty$      // initialize $M$
      EXTEND-SHORTEST-PATHS($L, L, M, n$)     // compute $M = L^2$
      $r = 2r$
      $L = M$      // ready for the next iteration
  return $L$
```

OK to overshoot, since products don't change after $L^{(n-1)}$.

Time

$\Theta(n^3 \lg n)$.

Floyd-Warshall algorithm

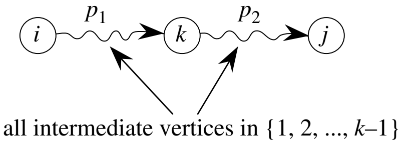
A different dynamic-programming approach.

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($W, n$)
  $D^{(0)} = W$
  for $k = 1$ to $n$
      let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
      for $i = 1$ to $n$
          for $j = 1$ to $n$
              $d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$
  return $D^{(n)}$
```

Can drop superscripts. (See Exercise 23.2-4 in text.)

Time

$\Theta(n^3)$.

Computing predecessors

Can compute predecessor matrix Π while computing the D matrices. Let $\Pi^{(k)} = \left(\pi_{ij}^{(k)}\right)$ for $k = 0, 1, \ldots, n$.

Define $\pi_{ij}^{(k)}$ recursively. For $k = 0$, a shortest path from i to j has no intermediate vertices:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \,, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \,. \end{cases}$$

For $k \geq 1$:

- If shortest path from i to j has k as an intermediate vertex, then it's $i \leadsto k \leadsto j$ where $k \neq j$. Choose j's predecessor to be the predecessor of j on a shortest path from k to j with all intermediate vertices $< k$: $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$.
- Otherwise, shortest path from i to j does not have k as an intermediate vertex. Keep the same predecessor as shortest path from i to j with all intermediate vertices $< k$: $\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$.

Transitive closure

Given $G = (V, E)$, directed.

Compute $G^* = (V, E^*)$.

- $E^* = \{(i, j) : \text{there is a path } i \leadsto j \text{ in } G\}$.

Could assign weight of 1 to each edge, then run FLOYD-WARSHALL.

- If $d_{ij} < n$, then there is a path $i \leadsto j$.
- 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)
- $t_{ij}^{(k)} = \begin{cases} 1 & \text{if there is path } i \leadsto j \text{ with all intermediate vertices in } \{1, 2, \ldots, k\} \,, \\ 0 & \text{otherwise} \,. \end{cases}$
- $t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E \,, \\ 1 & \text{if } i = j \text{ or } (i, j) \in E \,. \end{cases}$
- $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$.

```
TRANSITIVE-CLOSURE$(G, n)$
  let $T^{(0)} = \left(t_{ij}^{(0)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ to $n$
      for $j = 1$ to $n$
          if $i == j$ or $(i, j) \in G.E$
              $t_{ij}^{(0)} = 1$
          else $t_{ij}^{(0)} = 0$
  for $k = 1$ to $n$
      let $T^{(k)} = \left(t_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
      for $i = 1$ to $n$
          for $j = 1$ to $n$
              $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
  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 $G = (V, E), w : E \to \mathbb{R}$. Let h be any function such that $h : V \to \mathbb{R}$. 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, 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

$$\begin{aligned}
\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\
&= w(c) \qquad \text{(since } v_0 = v_k\text{)} .
\end{aligned}$$

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 $h : V \to \mathbb{R}$ 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 of claim 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

```
JOHNSON$(G, w)$
  compute $G'$, where $G'.V = G.V \cup \{s\}$,
        $G'.E = G.E \cup \{(s,v) : v \in G.V\}$, and
        $w(s,v) = 0$ for all $v \in G.V$
  if BELLMAN-FORD$(G', w, s)$ == FALSE
        print "the input graph contains a negative-weight cycle"
  else for each vertex $v \in G'.V$
              set $h(v)$ to the value of $\delta(s,v)$
                    computed by the Bellman-Ford algorithm
        for each edge $(u,v) \in G'.E$
              $\widehat{w}(u,v) = w(u,v) + h(u) - h(v)$
        let $D = (d_{uv})$ be a new $n \times n$ matrix
        for each vertex $u \in G.V$
              run DIJKSTRA$(G, \widehat{w}, u)$ to compute $\widehat{\delta}(u,v)$ for all $v \in G.V$
              for each vertex $v \in G.V$
                    $d_{uv} = \widehat{\delta}(u,v) + h(v) - h(u)$
        return $D$
```

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 23: All-Pairs Shortest Paths

Solution to Exercise 23.1-2

If $w_{ii} < 0$, then there would be a negative-weight cycle from a self-loop. If $w_{ii} > 0$, then no shortest path would use the self-loop $i \to i$, since using it strictly increases the weight of a path.

Solution to Exercise 23.1-3

This solution is also posted publicly

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 23.1-4

To show that matrix multiplication, as used in EXTEND-SHORTEST-PATHS, is associative, we first establish two properties of the operations:

- $\min\{x, y\} + z = \min\{x + z, y + z\}$ (addition distributes over min).
- $\min\{\min\{x, y\}, z\} = \min\{x, \min\{y, z\}\}$ (associativity of min).

We need to show that using the operations $+$ and min in place of $\cdot$ and $+$, we have that $(AB)C = A(BC)$, where A, B, and C are conforming matrices. In this application, and to keep things simple, we assume that each matrix is $n \times n$.

Let $Y = (y_{ij}) = (AB)C$ and $X = (x_{ik}) = AB$, so that $Y = XC = (AB)C$. We have

$$\begin{aligned}
x_{ik} &= \min_{1 \le p \le n} \{a_{ip} + b_{pk}\} \\
y_{ij} &= \min_{1 \le k \le n} \{x_{ik} + c_{kj}\} \\
&= \min_{1 \le k \le n} \left\{ \min_{1 \le p \le n} \{a_{ip} + b_{pk}\} + c_{kj} \right\} \\
&= \min_{1 \le k \le n} \left\{ \min_{1 \le p \le n} \{a_{ip} + c_{kj}, b_{pk} + c_{kj}\} \right\} \quad (+ \text{ distributes over min}) .
\end{aligned}$$

Now, let $T = (t_{ij}) = A(BC)$ and $S = (s_{pj}) = BC$, so that $T = AS = A(BC)$. We have

$$\begin{aligned}
s_{pj} &= \min_{1 \le k \le n} \{b_{pk} + c_{kj}\} \\
t_{ij} &= \min_{1 \le p \le n} \{a_{ip} + s_{pj}\} \\
&= \min_{1 \le p \le n} \left\{ a_{ip} + \min_{1 \le k \le n} \{b_{pk} + c_{kj}\} \right\} \\
&= \min_{1 \le p \le n} \left\{ \min_{1 \le k \le n} \{a_{ip} + c_{kj}, b_{pk} + c_{kj}\} \right\} \quad (+ \text{ distributes over min}) \\
&= \min_{1 \le k \le n} \left\{ \min_{1 \le p \le n} \{a_{ip} + c_{kj}, b_{pk} + c_{kj}\} \right\} \quad (\text{associativity of min}) \\
&= y_{ij} .
\end{aligned}$$

Since $y_{ij} = t_{ij}$, we conclude that $(AB)C = A(BC)$, and so this version of matrix multiplication is associative.

Solution to Exercise 23.1-5

This solution is also posted publicly

The all-pairs shortest-paths algorithm in Section 23.1 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 23.1-7

For all vertices i, j calculate π_{ij}, the predecessor of vertex j on the shortest path from i to j, by finding the vertex k such that $l_{ij} = l_{ik} + w(k, j)$. For each pair of vertices, we need to try n candidates for vertex k, for a total time of $\Theta(n^3)$.

Solution to Exercise 23.1-8

The idea is to update the Π matrix whenever the L matrix is updated.

```
SLOW-APSP($W, L^{(0)}, n$)
  let $L = (l_{ij})$, $M = (m_{ij})$, and $\Pi = (\pi_{ij})$ be new $n \times n$ matrices
  $L = L^{(0)}$
  for $r = 1$ to $n - 1$
      $M = \infty$    // initialize $M$
      EXTEND-SHORTEST-PATHS($L, W, M, \Pi, n$)
      $L = M$
  return $L$
```

```
EXTEND-SHORTEST-PATHS($L^{(r-1)}, W, L^{(r)}, \Pi, n$)
  // Assume that the elements of $L^{(r)}$ are initialized to $\infty$.
  for $i = 1$ to $n$
      for $j = 1$ to $n$
          $\pi_{ij} = $ NIL
          for $k = 1$ to $n$
              if $l_{ik}^{(r-1)} + w_{kj} < l_{ij}^{(r)}$
                  $l_{ij}^{(r)} = l_{ik}^{(r-1)} + w_{kj}$
                  $\pi_{ij} = k$
```

Solution to Exercise 23.1-9

Continue for one more iteration than the procedure does presently and see whether any shortest-path weights change. If they do, it is because there is a path with more than $|V| - 1$ edges that reaches a vertex with lower weight than the shortest path with at most $|V| - 1$ edges. Thus, this improvement comes from a path with at least $|V|$ edges, meaning that the graph has a negative-weight cycle.

Solution to Exercise 23.1-10

Run SLOW-APSP on the graph. Look at the diagonal elements of $L^{(r)}$. Return the first value of r for which one (or more) of the diagonal elements ($l_{ii}^{(r)}$) is negative. If r 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 r^*, where $r^* = \infty$ if the graph has no negative-weight cycles.

Correctness

Let's assume that for some value $r^* \leq n$ and some value of i, we find that $l_{ii}^{(r^*)} < 0$. Then the graph has a cycle with r^* edges that goes from vertex i to itself, and this cycle has negative weight (stored in $l_{ii}^{(r^*)}$). This is the minimum-length negative-weight cycle because SLOW-APSP computes all paths of 1 edge, then all paths of 2 edges, and so on, and all cycles shorter than r^* edges were checked before and did not have negative weight. Now assume that for all $r \leq n$, there is no negative $l_{ii}^{(r)}$ element. Then, there is no negative-weight cycle in the graph, because all cycles have length at most n.

Time

$O(n^4)$. More precisely, $\Theta(n^3 \cdot \min\{n, r^*\})$.

Faster solution

Run FASTER-APSP on the graph until the first time that the matrix $L^{(r)}$ has one or more negative values on the diagonal, or until it has computed $L^{(r)}$ for some $r > n$. If there are any negative entries on the diagonal, we know that the minimum-length negative-weight cycle has more than $r/2$ edges and at most r edges. We just need to binary search for the value of r^* in the range $r/2 < r^* \leq r$. The key observation is that on the way to computing $L^{(r)}$, the procedure computed $L^{(1)}$, $L^{(2)}$, $L^{(4)}$, $L^{(8)}$, …, $L^{(r/2)}$, and these matrices suffice to compute every matrix we'll need. Here's pseudocode:

```
FIND-MIN-LENGTH-NEG-WEIGHT-CYCLE(W, n)
  let $L^{(1)} = (l_{ij})$ be a new $n \times n$ matrix
  $L^{(1)} = W$
  $r = 1$
  while $r \le n$ and no diagonal entries of $L^{(r)}$ are negative
      let $L^{(2r)} = (l_{ij}^{(2r)})$ be a new $n \times n$ matrix
      EXTEND-SHORTEST-PATHS($L^{(r)}, L^{(r)}, L^{(2r)}, n$)
      $r = 2r$
  if $r > n$ and no diagonal entries of $L^{(r)}$ are negative
      return "no negative-weight cycles"
  elseif $r \le 2$
      return $r$
  else
      $low = r/2$
      $high = r$
      $d = r/4$
      while $d \ge 1$
          $s = low + d$
          let $L^{(s)} = (l_{ij}^{(s)})$ be a new $n \times n$ matrix
          $L^{(s)} = \infty$
          EXTEND-SHORTEST-PATHS($L^{(low)}, L^{(d)}, L^{(s)}, n$)
          if $L^{(s)}$ has any negative entries on the diagonal
              $high = s$
          else $low = s$
          $d = d/2$
      return $high$
```

Correctness

If, after the first **while** loop, $r > n$ and no diagonal entries of $L^{(r)}$ are negative, then there is no negative-weight cycle. Otherwise, if $r \le 2$, then either $r = 1$ or $r = 2$, and $L^{(r)}$ is the first matrix with a negative entry on the diagonal. Thus, the correct value to return is r.

If $r > 2$, then we maintain an interval bracketed by the values *low* and *high*, such that the correct value r^* is in the range $low < r^* \le high$. 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. $low < r^* \le high$.

Initialization: Initially, r is an integer power of 2 and $r > 2$. Since $d = r/4$, we have that d is an integer power of 2 and $d \ge 1$, so that $d = 2^p$ for some integer $p \ge 0$. We also have $(high - low)/2 = (r - (r/2))/2 = r/4 = d$. Finally, $L^{(r)}$ has a negative entry on the diagonal and $L^{(r/2)}$ does not. Since $low = r/2$ and $high = r$, we have that $low < r^* \le high$.

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 $p \geq -1$ implies $d' = 2^{p'}$ for some integer $p' \geq -1$, that $d = (high - low)/2$ implies $d' = (high' - low')/2$, and that $low < r^* \leq high$ implies $low' < r^* \leq high'$.

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 $r^* \leq s$. Because $high' = s$ and $low' = low$, we have that $low' < r^* \leq high'$.
- 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 $r^* > s$. Because $low' = s$ and $high' = high$, we have that $low' < r^* \leq high'$.

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 $low < r^* \leq high$. Since $high - low = 2d = 1$ and $r^* > low$, the only possible value for r^* 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 upon each call EXTEND-SHORTEST-PATHS$(L^{(low)}, L^{(d)}, L^{(s)}, n)$, the procedure has already computed $L^{(low)}$ and $L^{(d)}$. Initially, since $low = r/2$, it 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 the procedure has just computed $L^{(s)}$. As for $L^{(d)}$, observe that d takes on the values $r/4, r/8, r/16, \ldots, 1$, and again, the procedure computed all of these L matrices in the first **while** loop. Thus, the claim is proven. Each of the two **while** loops iterates $\Theta(\lg r^*)$ times. Since the procedure has 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 $\Theta(n^3 \lg r^*)$.

In general, therefore, the running time is $\Theta(n^3 \lg \min\{n, r^*\})$.

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 $\Theta(\lg \min\{n, r^*\})$ matrices, and so the space requirement increases to $\Theta(n^3 \lg \min\{n, r^*\})$.

Solution to Exercise 23.2-2

Instead of starting with the matrix W, the algorithm starts with a matrix $T^{(1)} = \left(t_{ij}^{(1)}\right)$, indicating whether there is a path with at most 1 edge from vertex i to vertex j:

$$t_{ij}^{(1)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \ , \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \ , \end{cases}$$

$T^{(1)}$ has the same values as $T^{(0)}$ in Section 23.2. Here is a procedure to create $T^{(1)}$:

```
INITIALIZE-TRANSITIVE-CLOSURE$(G, n)$
  let $T^{(1)} = \left(t_{ij}^{(1)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ to $n$
      for $j = 1$ to $n$
          if $i == j$ or $(i,j) \in G.E$
              $t_{ij}^{(1)} = 1$
          else $t_{ij}^{(1)} = 0$
  return $T^{(1)}$
```

Here is the analog of the EXTEND-SHORTEST-PATHS procedure, but for transitive closure. Instead of taking W as its second parameter, it takes $T(1)$:

```
EXTEND-TRANSITIVE-CLOSURE$(T^{(r-1)}, T^{(1)}, T^{(r)}, n)$
  // Assume that the elements of $T^{(r)}$ are initialized to 0.
  for $i = 1$ to $n$
      for $j = 1$ to $n$
          for $k = 1$ to $n$
              $t_{ij}^{(r)} = t_{ij}^{(r)} \vee (t_{ik}^{(r-1)} \wedge t_{kj}^{(1)})$
```

Now the analog of SLOW-APSP:

```
SLOW-TRANSITIVE-CLOSURE$(G, n)$
  $T^{(1)} =$ INITIALIZE-TRANSITIVE-CLOSURE$(G, n)$
  let $T = (t_{ij})$ and $M = (m_{ij})$ be new $n \times n$ matrices
  $T = T^{(1)}$
  for $r = 2$ to $n - 1$
      $M = 0$
      EXTEND-TRANSITIVE-CLOSURE$(T, T^{(1)}, M, n)$
      $T = M$
  return $T$
```

And the analog of FASTER-APSP:

```
FASTER-TRANSITIVE-CLOSURE(G, n)
  $T^{(1)}$ = INITIALIZE-TRANSITIVE-CLOSURE(G, n)
  let $T = (t_{ij})$ and $M = (m_{ij})$ be new $n \times n$ matrices
  $T = T^{(1)}$
  $r = 1$
  while $r < n - 1$
      $M = 0$
      EXTEND-TRANSITIVE-CLOSURE(T, T, M, n)
      $r = 2r$
      $T = M$
  return T
```

Solution to Exercise 23.2-4

This solution is also posted publicly

With the superscripts, the computation is $d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$. If, having dropped the superscripts, the procedure were to compute and store d_{ik} or d_{kj} before using these values to compute d_{ij}, it might be computing one of the following:

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}\right\} ,$$
$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)}\right\} ,$$
$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)}\right\} .$$

In any of these scenarios, the code computes 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 23.2-5

The alternative definition of Π is incorrect. Try it on the following matrix W:

$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & -3 & 1 \\ \infty & 3 & 0 & 4 \\ \infty & \infty & \infty & 0 \end{pmatrix} .$$

The resulting Π matrix should be

$$\begin{pmatrix} \text{NIL} & 1 & 2 & 2 \\ \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 3 \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \end{pmatrix} ,$$

but instead comes to

$$\begin{pmatrix} \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \end{pmatrix} .$$

Solution to Exercise 23.2-6

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 that 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 that $d_{ii}^{(k)}$ 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, you 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 23.2-8

Just run breadth-first search or depth-first search once from each vertex to fill in each row of the transitive closure matrix. That takes time $O(V(V + E)) = O(VE)$ since $|V| = O(E)$.

Solution to Exercise 23.2-9

Start by computing the component graph $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$, which takes $\Theta(V + E)$ time. While computing the component graph, keep track of which vertices are in each strongly connected component by making a linked list of vertices for each component. Because G^{SCC} is a directed acyclic graph, we can compute its transitive closure $G^{\text{SCC}*}$ in $f(|V^{\text{SCC}}|, |E^{\text{SCC}}|)$ time, which is $O(f(|V|, |E|))$ since $|V^{\text{SCC}}| \le |V|$ and $|E^{\text{SCC}}| \le |E|$.

The edges in E^* are the ordered pairs of vertices (u, v) such that either

- u and v are in the same strongly connected component, or
- u is in the strongly connected component represented in G^{SCC} by u', v is in the strongly connected component represented in G^{SCC} by v', and the edge (u', v') is in $G^{\text{SCC}*}$.

To compute the edges in E^*, therefore, first add all pairs of vertices that are in the same strongly connected component of G. Then, for each edge (u', v') in $G^{\text{SCC}*}$, find all pairs of vertices for which the first vertex is in the strongly connected component for u' and the second vertex is in the strongly connected component for v', and add each pair into E^*. There are $|E^*|$ such edges, and so this step takes $\Theta(E^*)$ time.

The total time, therefore, is $O((V + E) + f(|V|, |E|) + E^*)$. Since the transitive closure of a directed graph must have at least as many edges as the graph, we have $|E^*| \ge |E|$, and so the running time is $O(f(|V|, |E|) + V + E^*)$.

Solution to Exercise 23.3-2

Adding the new vertex s ensures that the shortest-path weights computed by the Bellman-Ford algorithm are all finite. That way, each value $h(v)$ is finite, and so all the reweighted edge weights $\widehat{w}(u, v)$ are well defined.

Solution to Exercise 23.3-3

If $w(u, v) \ge 0$ for all edges $(u, v) \in E$, then $h(u) = 0$ for all $u \in V$ because the shortest path from s to u is simply $s \to u$, with weight 0. Thus, $\widehat{w}(u, v) = w(u, v)$ for all edges $(u, v) \in E$.

Solution to Exercise 23.3-4
This solution is also posted publicly

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 23.3-5

If G contains a 0-weight cycle c, then $\sum_{(u,v)\in c} w(u,v) = 0$. We also have

$$\begin{aligned}\sum_{(u,v)\in c} \widehat{w}(u,v) &= \sum_{(u,v)\in c} (w(u,v) + h(u) - h(v)) \\ &= \sum_{(u,v)\in c} w(u,v) ,\end{aligned}$$

because each vertex appears once in the summation as $h(u)$ and once as $h(v)$. Thus, $\sum_{(u,v)\in c} \widehat{w}(u,v) = 0$. Since $\widehat{w}(u,v) \geq 0$ for all edges $(u,v) \in E$, we must have $\widehat{w}(u,v) = 0$ for all edges $(u,v) \in c$.

Solution to Exercise 23.3-6

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. Running Bellman-Ford from s gives $h(s) = \delta(s, s) = 0$ and $h(u) = \delta(s, u) = \infty$. Reweighting produces $\widehat{w}(u, s) = 0 + \infty - 0 = \infty$. The procedure computes $\widehat{\delta}(u, s) = \infty$, so that $d_{us} = \infty + 0 - \infty \neq 0$. Since $\delta(u, s) = 0$, the result is incorrect.

If the graph G is strongly connected, then $h(v) = \delta(s, v) < \infty$ 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 23.1 are finite, and so the lemma holds. Therefore, the conditions required 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, $\widehat{\delta}(u, v) < \infty$ 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 23-1

a. Let $T = (t_{ij})$ be the $|V| \times |V|$ matrix representing the transitive closure, such that t_{ij} is 1 if there is a path from i to j, and 0 otherwise.

Initialize T (when there are no edges in G) as follows:

$$t_{ij} = \begin{cases} 1 & \text{if } i = j , \\ 0 & \text{otherwise} . \end{cases}$$

Update T as follows when an edge (u, v) is added to G:

```
TRANSITIVE-CLOSURE-UPDATE$(T, u, v)$
  let $T$ be $|V| \times |V|$
  for $i = 1$ to $|V|$
      for $j = 1$ to $|V|$
          if $t_{iu} == 1$ and $t_{vj} == 1$
              $t_{ij} = 1$
```

- With this procedure, 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 $t_{uv} = 1$, because both t_{uu} and t_{vv} are initialized to 1.
- This procedure takes $\Theta(V^2)$ time because of the two nested loops.

b. Consider inserting the edge $(v_{|V|}, v_1)$ into the straight-line graph $v_1 \to v_2 \to \cdots \to v_{|V|}$.

Before this edge is inserted, only $|V|(|V|+1)/2$ 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 that all $|V|^2$ entries in T are 1. Hence $|V|^2 - (|V|(|V|+1)/2) = \Theta(V^2)$ entries must be changed in T, so that 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, and 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 $t_{iv} = 1$. That is, if there is already a path $i \leadsto v$, then adding the edge (u, v) cannot make any new vertices reachable from i. The loop to set t_{ij} to 1 for j such that there exists a path $v \leadsto j$ is just setting entries that are already 1. Eliminate this redundant processing as follows:

```
TRANSITIVE-CLOSURE-UPDATE$(T, u, v)$
  let $T$ be $|V| \times |V|$
  for $i = 1$ to $|V|$
      if $t_{iu} == 1$ and $t_{iv} == 0$
          for $j = 1$ to $|V|$
              if $t_{vj} == 1$
                  $t_{ij} = 1$
```

We show that this procedure takes $O(V^3)$ time to update the transitive closure for any sequence of m insertions:

- There cannot be more than $|V|^2$ edges in G, so that $m \le |V|^2$.
- Summed over m insertions, the time for the outer **for** loop header and the test for $t_{iu} == 1$ and $t_{iv} == 0$ is $O(mV) = O(V^3)$.

- The last three lines, which take $\Theta(V)$ time, are executed only $O(V^2)$ times for m insertions. To see why, notice that the last three lines are executed only when t_{iv} equals 0, and in that case, the last line sets $t_{iv} = 1$. 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 m insertions is $O(V^3)$.

Lecture Notes for Chapter 24: Maximum Flow

Chapter 24 overview

Network flow

[The third and fourth editions treat flow networks differently from the first two editions. The concept of net flow is gone, except that we do discuss net flow across a cut. Skew symmetry is also gone, as is implicit summation notation. The third and fourth editions count flows on edges directly. We find that although the mathematics is not quite as slick as in the first two editions, the approach in the newer editions matches intuition more closely, and therefore students tend to pick it up more quickly.]

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. The textbook has an example where a flow is how many trucks per day can ship hockey pucks between cities.

Want to compute the maximum rate that material can be shipped from a designated ***source*** to a designated ***sink***.

Flow networks

$G = (V, E)$ directed.

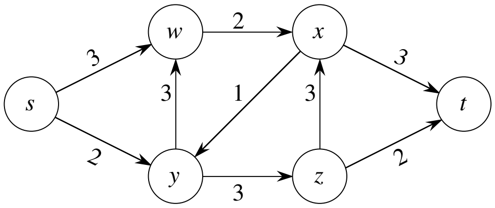
Each edge (u, v) has a ***capacity*** $c(u, v) \geq 0$.

If $(u, v) \notin E$, then $c(u, v) = 0$.

If $(u, v) \in E$, then reverse edge $(v, u) \notin E$. (Can work around this restriction.)

Source vertex s, ***sink*** vertex t, assume $s \leadsto v \leadsto t$ for all $v \in V$, so that each vertex lies on a path from source to sink.

Example: *[Edges are labeled with capacities.]*



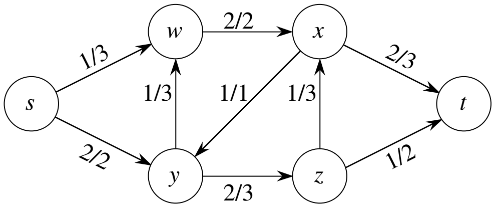
Flow

A function $f : V \times V \to \mathbb{R}$ satisfying

- ***Capacity constraint:*** For all $u, v \in V, 0 \le f(u, v) \le c(u, v)$,
- ***Flow conservation:*** For all $u \in V - \{s, t\}$, $\underbrace{\sum_{v \in V} f(v, u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u, v)}_{\text{flow out of } u}$.

  Equivalently, $\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0$.

[Add flows to previous example. Edges here are labeled as flow/capacity. Leave on board.]



- Note that all flows are ≤ capacities.
- Verify flow conservation by adding up flows at a couple of vertices.
- Note that all flows = 0 is legitimate.

Value of flow f

$$\begin{aligned} &= |f| \\ &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \\ &= \text{flow out of source} - \text{flow into source} . \end{aligned}$$

In the example above, value of flow $f = |f| = 3$.

Antiparallel edges

Definition of flow network does not allow both (u, v) and (v, u) to be edges. These edges would be ***antiparallel***.

What if really need antiparallel edges?

- Choose one of them, say (u, v).
- Create a new vertex v'.
- Replace (u, v) by two new edges (u, v') and (v', v), with $c(u, v') = c(v', v) = c(u, v)$.
- Get an equivalent flow network with no antiparallel edges.

Multiple sources and sinks

If you need multiple sources $s_1, \ldots, s_m$, create a single ***supersource*** s with edges (s, s_i) for $i = 1, \ldots, m$ and infinite capacity on each edge.

Same idea for multiple sinks: create a single ***supersink*** with an infinite-capacity edge from each sink to the supersink.

Now there are just one source and one sink.

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 require $s \in S$ and $t \in T$.

For flow f, the ***net flow*** across cut (S, T) is

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) .$$

Capacity of cut (S, T) is

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) .$$

A ***minimum cut*** of G is a cut whose capacity is minimum over all cuts of G.

Asymmetry between net flow across a cut and capacity of a cut: For capacity, count only capacities of edges going from S to T. Ignore edges going in the reverse direction. For net flow, count flow on all edges across the cut: flow on edges going from S to T minus flow on edges going from T to S.

In previous example, consider the cut $S = \{s, w, y\}$, $T = \{x, z, t\}$.

$$\begin{aligned} f(S,T) &= \underbrace{f(w,x) + f(y,z)}_{\text{from } S \text{ to } T} - \underbrace{f(x,y)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3 . \\ c(S,T) &= \underbrace{c(w,x) + c(y,z)}_{\text{from } S \text{ to } T} \\ &= 2 + 3 \\ &= 5 . \end{aligned}$$

Now consider the cut $S = \{s, w, x, y\}$, $T = \{z, t\}$.

$$\begin{aligned} f(S,T) &= \underbrace{f(x,t) + f(y,z)}_{\text{from } S \text{ to } T} - \underbrace{f(z,x)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3 . \\ c(S,T) &= \underbrace{c(x,t) + c(y,z)}_{\text{from } S \text{ to } T} \\ &= 3 + 3 \\ &= 6 . \end{aligned}$$

Same flow as previous cut, higher capacity.

Lemma
For any cut (S, T), $f(S, T) = |f|$.
(Net flow across the cut equals value of the flow.)
[Leave on board.]

[This proof is much more involved than the proof in the first two editions. You might want to omit it, or just give the intuition that no matter where you cut the pipes in a network, you'll see the same flow volume coming out of the openings.]

Proof Rewrite flow conservation: for any $u \in V - \{s, t\}$,

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0 .$$

Take definition of $|f|$ and add in left-hand side of above equation, summed over all vertices in $S - \{s\}$. Above equation applies to each vertex in $S - \{s\}$ (since $t \notin S$ and obviously $s \notin S - \{s\}$), so just adding in lots of 0s:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) .$$

Expand right-hand summation and regroup terms:

$$\begin{aligned}
|f| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v, u) \\
&= \sum_{v \in V} \left(f(s, v) + \sum_{u \in S - \{s\}} f(u, v) \right) - \sum_{v \in V} \left(f(v, s) + \sum_{u \in S - \{s\}} f(v, u) \right) \\
&= \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u) .
\end{aligned}$$

Partition V into $S \cup T$ and split each summation over V into summations over S and T:

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u, v) + \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&\qquad + \left(\sum_{v \in S} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) \right) .
\end{aligned}$$

Summations within parentheses are the same, since $f(x, y)$ appears once in each summation, for any $x, y \in V$. These summations cancel:

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \\
&= f(S, T) .
\end{aligned}$$

■ (lemma)

Corollary
The value of any flow ≤ capacity of any cut.
[Leave on board.]

Proof Let (S, T) be any cut, f be any flow.

$$\begin{aligned}
|f| &= f(S,T) && \text{(lemma)} \\
&= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) && \text{(definition of } f(S,T)\text{)} \\
&\le \sum_{u\in S}\sum_{v\in T} f(u,v) && (f(v,u) \ge 0) \\
&\le \sum_{u\in S}\sum_{v\in T} c(u,v) && \text{(capacity constraint)} \\
&= c(S,T)\ . && \text{(definition of } c(S,T)\text{)}
\end{aligned}$$

■ (corollary)

Therefore, maximum flow $\le$ 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 be pushed directly from u to v?
That's the ***residual capacity***,

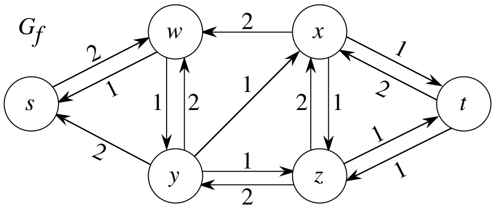
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\ , \\ f(v,u) & \text{if } (v,u) \in E\ , \\ 0 & \text{otherwise (i.e., } (u,v), (v,u) \notin E)\ . \end{cases}$$

The ***residual network*** is $G_f = (V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}\ .$$

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 $(v,u) \in E$.

Therefore, $|E_f| \le 2|E|$.

Residual network is similar to a flow network, except that it may contain antiparallel edges ((u,v) and (v,u)). Can define a flow in a residual network that satisfies the definition of a flow, but with respect to capacities c_f in G_f.

Given flows f in G and f' in G_f, define $(f \uparrow f')$, the ***augmentation*** of f by f', as a function $V \times V \to \mathbb{R}$:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \,, \\ 0 & \text{otherwise} \end{cases}$$

for all $u, v \in V$.

Intuition: Increase the flow on (u,v) by $f'(u,v)$ but decrease it by $f'(v,u)$ because pushing flow on the reverse edge in the residual network decreases the flow in the original network. Also known as ***cancellation***.

Lemma

Given a flow network G, a flow f in G, and the residual network G_f, let f' be a flow in G_f. Then $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

[See textbook for proof. It has a lot of summations in it. Probably not worth writing on the board.]

Augmenting path

A simple path $s \rightsquigarrow t$ in G_f.

- Admits more flow along each edge.
- Like a sequence of pipes through which can squirt more flow from s to t.

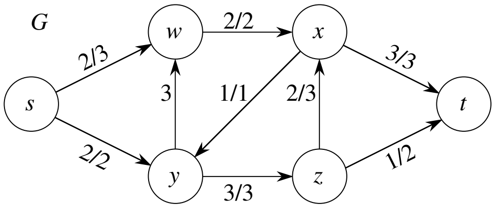
How much more flow can be pushed from s to t along augmenting path p? That is the ***residual capacity*** of p:

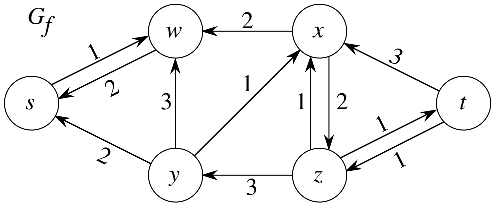
$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$.

For our example, consider the augmenting path $p = \langle s, w, y, z, x, t \rangle$.

Minimum residual capacity is 1.

After pushing 1 additional unit along p: *[Continue from G left on board from before. Edge (y,w) has $f(y,w) = 0$, which we omit, showing only $c(y,w) = 3$.]*





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 $f_p : V \times V \to \mathbb{R}$:

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

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. Then $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Theorem (Max-flow min-cut theorem)
The following are equivalent:

1. f is a maximum flow.
2. G_f has no augmenting path.
3. $|f| = c(S,T)$ for some cut (S,T).

Proof
(1) ⇒ (2): Show the contrapositive: if G_f has an augmenting path, then f is not a maximum flow. If G_f has augmenting path p, then by the above corollary, $f \uparrow f_p$ is a flow in G with value $|f| + |f_p| > |f|$, so that f was not a maximum flow.

(2) ⇒ (3): Suppose G_f has no augmenting path. Define

$$\begin{aligned} S &= \{v \in V : \text{there exists a path } s \leadsto v \text{ in } G_f\}\ , \\ T &= V - S\ . \end{aligned}$$

Must have $t \in T$; otherwise there is an augmenting path.
Therefore, (S,T) is a cut.
Consider $u \in S$ and $v \in T$:

- If $(u,v) \in E$, must have $f(u,v) = c(u,v)$; otherwise, $(u,v) \in E_f \Rightarrow v \in S$.
- If $(v,u) \in E$, must have $f(v,u) = 0$; otherwise, $c_f(u,v) = f(v,u) > 0 \Rightarrow (u,v) \in E_f \Rightarrow v \in S$.
- If $(u,v), (v,u) \notin E$, must have $f(u,v) = f(v,u) = 0$.

Then,

$$\begin{aligned} f(S,T) &= \sum_{u \in S}\sum_{v \in T} f(u,v) - \sum_{v \in T}\sum_{u \in S} f(v,u) \\ &= \sum_{u \in S}\sum_{v \in T} c(u,v) - \sum_{v \in T}\sum_{u \in S} 0 \\ &= c(S,T)\ . \end{aligned}$$

By lemma, $|f| = f(S,T) = c(S,T)$.

(3) ⇒ (1): An earlier corollary says that the value of any flow is $\leq$ the capacity of any cut, so that $|f| \leq c(S,T)$.
Therefore, $|f| = c(S,T) \Rightarrow f$ is a max flow. ■ (theorem)

Ford-Fulkerson algorithm

Keep augmenting flow along an augmenting path until there is no augmenting path.

Represent the flow attribute using the usual dot-notation, but on an edge: $(u, v).f$.

```
FORD-FULKERSON(G, s, t)
  for each edge $(u, v) \in G.E$
      $(u, v).f = 0$
  while there is an augmenting path $p$ in $G_f$
      $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
      for each edge $(u, v)$ in $p$        // augment $f$ by $c_f(p)$
          if $(u, v) \in G.E$
              $(u, v).f = (u, v).f + c_f(p)$
          else $(v, u).f = (v, u).f - c_f(p)$
  return $f$
```

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 textbook 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 $s \leadsto t$ 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 some flow augmentation 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 that $\delta_{f'}(s, v) < \delta_f(s, v)$.

Let a shortest path s to v in $G_{f'}$ be $s \leadsto u \rightarrow v$, so that $(u, v) \in E_{f'}$ 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 of claim If $(u, v) \in E_f$, then

$$\begin{aligned}\delta_f(s, v) &\le \delta_f(s, u) + 1 \quad \text{(triangle inequality)}\\ &\le \delta_{f'}(s, u) + 1\\ &= \delta_{f'}(s, v)\ ,\end{aligned}$$

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, u) as its last edge.

Therefore,

$$\begin{aligned}\delta_f(s, v) &= \delta_f(s, u) - 1\\ &\le \delta_{f'}(s, u) - 1\\ &= \delta_{f'}(s, v) - 2\ ,\end{aligned}$$

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 augmenting along p.

≥ 1 edge on any augmenting path is critical.

Will show that each of the $|E|$ edges can become critical $\le |V|/2$ times.

Consider $u, v \in V$ such that either $(u, v) \in E$ or $(v, u) \in E$. Since augmenting paths are shortest paths, when (u, v) becomes critical the 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, $\delta_f(s, v) \le \delta_{f'}(s, v) \Rightarrow$

$$\begin{aligned}\delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1\\ &\ge \delta_f(s, v) + 1\\ &= \delta_f(s, u) + 2\ .\end{aligned}$$

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 $\leq |V| - 2$
$\Rightarrow$ after (u, v) becomes critical the first time, it can become critical
$\leq (|V| - 2)/2 = |V|/2 - 1$ times more
$\Rightarrow$ (u, v) can become critical $\leq |V|/2$ times.

Since $O(E)$ pairs of vertices can have an edge between them in residual network, total # of critical edges during execution of Edmonds-Karp 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 the first three editions of the textbook give $O(V^3)$. The two sections on push-relabel algorithm were dropped from the fourth edition but are available from the MIT Press website for the book.]*

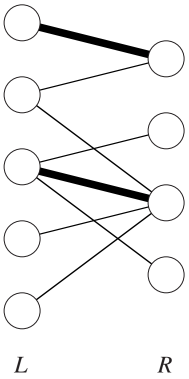
Maximum bipartite matching

Example of a problem that can be solved by turning it into a flow problem.

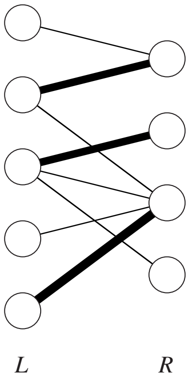
$G = (V, E)$ (undirected) is ***bipartite*** if there is a partition of the vertices $V = L \cup R$ such that all edges in E go between L and R.

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'.)



matching



maximum matching

[Edges in matchings are drawn with heavy lines.]

Problem

Given a bipartite graph (with the partition), find a maximum matching.

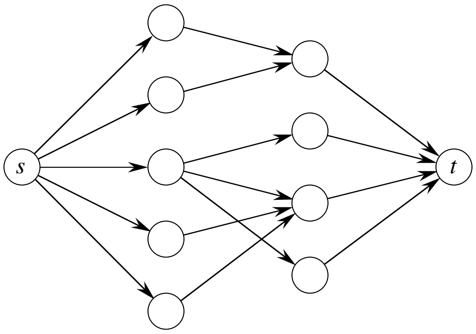
Application

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 $G' = (V', E')$.

- $V' = V \cup \{s, t\}$.
- $E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\}$.
- $c(u, v) = 1$ 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'. Textbook shows that it will have integer values for all (u, v).

Use edges (u, v) such that $u \in L$ and $v \in R$ that carry flow of 1 in matching.

Textbook proves that this method produces a maximum matching.

[The next chapter, Chapter 25, has a better algorithm (Hopcroft-Karp) to find a maximum matching, as well as other algorithms based on bipartite matchings.]

Solutions for Chapter 24: Maximum Flow

Solution to Exercise 24.1-1

We will prove that for every flow in $G = (V, E)$, we can construct a flow in $G' = (V', E')$ that has the same value as that of the flow in G. The required result follows since a maximum flow in G is also a flow. Let f be a flow in G. By construction, $V' = V \cup \{x\}$ and $E' = (E - \{(u, v)\}) \cup \{(u, x), (x, v)\}$. Construct f' in G' as follows:

$$f'(y, z) = \begin{cases} f(y, z) & \text{if } (y, z) \neq (u, x) \text{ and } (y, z) \neq (x, v) , \\ f(u, v) & \text{if } (y, z) = (u, x) \text{ or } (y, z) = (x, v) . \end{cases}$$

Informally, f' is the same as f, except that the flow $f(u, v)$ now passes through an intermediate vertex x. The vertex x has incoming flow (if any) only from u, and has outgoing flow (if any) only to vertex v.

We first prove that f' satisfies the required properties of a flow. It is obvious that the capacity constraint is satisfied for every edge in $E' - \{(u, x), (x, v)\}$ and that every vertex in $V' - \{u, v, x\}$ obeys flow conservation.

To show that edges (u, x) and (x, v) obey the capacity constraint, we have

$$\begin{aligned} f(u, x) &= f(u, v) \leq c(u, v) = c(u, x) , \\ f(x, v) &= f(u, v) \leq c(u, v) = c(x, v) . \end{aligned}$$

We now prove flow conservation for u. Assuming that $u \notin \{s, t\}$, we have

$$\begin{aligned} \sum_{y \in V'} f'(u, y) &= \sum_{y \in V' - \{x\}} f'(u, y) + f'(u, x) \\ &= \sum_{y \in V - \{v\}} f(u, y) + f(u, v) \\ &= \sum_{y \in V} f(u, y) \\ &= \sum_{y \in V} f(y, u) \qquad \text{(because } f \text{ obeys flow conservation)} \\ &= \sum_{y \in V'} f'(y, u) . \end{aligned}$$

For vertex v, a symmetric argument proves flow conservation.

For vertex x, we have

$$\begin{aligned}\sum_{y \in V'} f'(y,x) &= f'(u,x) \\ &= f'(x,v) \\ &= \sum_{y \in V'} f'(x,y) .\end{aligned}$$

Thus, f' is a valid flow in G'.

We now prove that the values of the flow in both cases are equal. If the source s is neither u nor v, the proof is trivial, since our construction assigns the same flows to incoming and outgoing edges of s. If $s = u$, then

$$\begin{aligned}|f'| &= \sum_{y \in V'} f'(u,y) - \sum_{y \in V'} f'(y,u) \\ &= \sum_{y \in V' - \{x\}} f'(u,y) - \sum_{y \in V'} f'(y,u) + f'(u,x) \\ &= \sum_{y \in V - \{v\}} f(u,y) - \sum_{y \in V} f(y,u) + f(u,v) \\ &= \sum_{y \in V} f(u,y) - \sum_{y \in V} f(y,u) \\ &= |f| .\end{aligned}$$

The case when $s = v$ is symmetric. We conclude that f' is a valid flow in G' with $|f'| = |f|$.

Solution to Exercise 24.1-2

Let f be a flow in original multiple-source, multiple-sink network G and f' be a flow in the corresponding single-source, single sink network G'. The value $|f|$ of the flow in G is just the sum of the flow values from the individual sources:

$$|f| = \sum_{s_i} \left(\sum_{v \in V} f(s_i, v) - \sum v \in V f(v, s_i) \right) .$$

In G', the flow f' from the supersource to each original source s_i is

$$f'(s, s_i) = \sum_{v \in V} f(s_i, v) - \sum v \in V f(v, s_i) ,$$

which is less than $c(s, s_i) = \infty$, so that the capacity constraint holds. Since the flow into each source s_i equals the flow out, flow conservation holds as well. A similar argument applies to the sinks and supersink.

Solution to Exercise 24.1-3

We show that, given any flow f' in the flow network $G = (V, E)$, we can construct a flow f as stated in the exercise. The result will follow when f' is a maximum flow. The idea is that even if there is a path from s to the strongly connected component of u, no flow can enter the component, since the flow has no path to reach t. Thus, all the flow inside the strongly connected component must be cyclic, which can be made zero without affecting the net value of the flow.

Two cases are possible: where u is not connected to t, and where u is not connected to s. We only analyze the former case. The analysis for the latter case is similar.

Let Y be the set of all vertices that have no path to t. Our roadmap will be to first prove that no flow can leave Y. We use this result and flow conservation to prove that no flow can enter Y. We shall then constuct the flow f, which has the required properties, and prove that $|f| = |f'|$.

The first step is to prove that there can be no flow from a vertex $y \in Y$ to a vertex $v \in V - Y$. That is, $f'(y, v) = 0$. This is so, because there are no edges (y, v) in E. If there were an edge $(y, v) \in E$, then there would be a path from y to t, which contradicts how we defined the set Y.

We will now prove that $f'(v, y) = 0$, too. We will do so by applying flow conservation to each vertex in Y and taking the sum over Y. By flow conservation, we have

$$\sum_{y \in Y} \sum_{v \in V} f'(y, v) = \sum_{y \in Y} \sum_{v \in V} f'(v, y) .$$

Partitioning V into Y and $V - Y$ gives

$$\begin{aligned} \sum_{y \in Y} \sum_{v \in V - Y} f'(y, v) + \sum_{y \in Y} \sum_{v \in Y} f'(y, v) \\ = \sum_{y \in Y} \sum_{v \in V - Y} f'(v, y) + \sum_{y \in Y} \sum_{v \in Y} f'(v, y) . \qquad (*) \end{aligned}$$

But we also have

$$\sum_{y \in Y} \sum_{v \in Y} f'(y, v) = \sum_{y \in Y} \sum_{v \in Y} f'(v, y) ,$$

since the left-hand side is the same as the right-hand side, except for a change of variable names v and y. We also have

$$\sum_{y \in Y} \sum_{v \in V - Y} f'(y, v) = 0 ,$$

since $f'(y, v) = 0$ for each $y \in Y$ and $v \in V - Y$. Thus, equation $(*)$ simplifies to

$$\sum_{y \in Y} \sum_{v \in V - Y} f'(v, y) = 0 .$$

Because the flow function is nonnegative, $f'(v, y) = 0$ for each $v \in V$ and $y \in Y$. We conclude that there can be no flow between any vertex in Y and any vertex in $V - Y$.

The same technique can show that if there is a path from u to t but not from s to u, and we define Z as the set of vertices that do not have have a path from s to u, then there can be no flow between any vertex in Z and any vertex in $V - Z$. Let $X = Y \cup Z$. We thus have $f'(v, x) = f'(x, v) = 0$ if $x \in X$ and $v \notin X$.

We are now ready to construct flow f:

$$f(u,v) = \begin{cases} f'(u,v) & \text{if } u,v \notin X \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

We note that f satisfies the requirements of the exercise. We now prove that f also satisfies the requirements of a flow function.

The capacity constraint is satisfied, since whenever $f(u,v) = f'(u,v)$, we have $f(u,v) = f'(u,v) \le c(u,v)$ and whenever $f(u,v) = 0$, we have $f(u,v) = 0 \le c(u,v)$.

For flow conservation, let x be some vertex other than s or t. If $x \in X$, then from the construction of f, we have

$$\sum_{v \in V} f(x,v) = \sum_{v \in V} f(v,x) = 0 \ .$$

Otherwise, if $x \notin X$, note that $f(x,v) = f'(x,v)$ and $f(v,x) = f'(v,x)$ for all vertices $v \in V$. Thus,

$$\begin{aligned} \sum_{v \in V} f(x,v) &= \sum_{v \in V} f'(x,v) \\ &= \sum_{v \in V} f'(v,x) \quad \text{(because } f' \text{ obeys flow conservation)} \\ &= \sum_{v \in V} f(v,x) \ . \end{aligned}$$

Finally, we prove that the value of the flow remains the same. Since $s \notin X$, we have $f(s,v) = f'(s,v)$ and $f(v,x) = f'(v,x)$ for all vertices $v \in V$, and so

$$\begin{aligned} |f| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) \\ &= \sum_{v \in V} f'(s,v) - \sum_{v \in V} f'(v,s) \\ &= |f'| \ . \end{aligned}$$

Solution to Exercise 24.1-4

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 flow conservation, observe that since f_1 and f_2 obey flow conservation, we have $\sum_{v \in V} f_1(v,u) = \sum_{v \in V} f_1(u,v)$ and $\sum_{v \in V} f_1(v,u) = \sum_{v \in V} f_1(u,v)$ for any $u \in V - \{s,t\}$. We need to show that

$$\sum_{v \in V} (\alpha f_1(v,u) + (1-\alpha) f_2(v,u)) = \sum_{v \in V} (\alpha f_1(u,v) + (1-\alpha) f_2(u,v))$$

for any $u \in V - \{s,t\}$. We multiply both sides of the equality for f_1 by α, multiply both sides of the equality for f_2 by $1-\alpha$, and add the left-hand and right-hand sides of the resulting equalities to get

$$\alpha \sum_{v \in V} f_1(v,u) + (1-\alpha) \sum_{v \in V} f_2(v,u) = \alpha \sum_{v \in V} f_1(u,v) + (1-\alpha) \sum_{v \in V} f_2(u,v) \ .$$

Observing that

$$\begin{aligned} \alpha \sum_{v \in V} f_1(v,u) + (1-\alpha) \sum_{v \in V} f_2(v,u) &= \sum_{v \in V} \alpha f_1(v,u) + \sum_{v \in V} (1-\alpha) f_2(v,u) \\ &= \sum_{v \in V} (\alpha f_1(v,u) + (1-\alpha) f_2(v,u)) \end{aligned}$$

and, likewise, that

$$\alpha \sum_{v \in V} f_1(u,v) + (1-\alpha) \sum_{v \in V} f_2(u,v) = \sum_{v \in V} (\alpha f_1(u,v) + (1-\alpha) f_2(u,v))$$

completes the proof that flow conservation holds, and thus that flows form a convex set.

Solution to Exercise 24.1-5

For each pair of distinct vertices u, v, create variables x_{uv} and x_{vu}, which represent the flows $f(u,v)$ and $f(v,u)$, respectively. The objective function is to maximize $\sum_{v \in V - \{s\}} (x_{sv} - x_{vs})$. The capacity constraints are $x_{uv} \le c(u,v)$ for all $u, v \in V$ such that $u \ne v$. The flow conservation constraints are

$$\sum_{v \in V} (x_{uv} - x_{vu}) \le 0 \quad \text{for all } u \in V - \{s,t\} \ ,$$

$$\sum_{v \in V} (x_{vu} - x_{uv}) \le 0 \quad \text{for all } u \in V - \{s,t\} \ .$$

We need both sets of constraints for flow conservation to make the difference between flow in and flow out equal 0.

Solution to Exercise 24.1-6

Create a vertex for each corner, and if there is a street between corners u and v, create directed edges (u,v), (v,v_u), and (v_u,u), where v_u is a unique vertex created for only this street between corners u and v. (We need vertex v_u to avoid antiparallel edges. Note that if there is a street between corners u and v and between corners x and v, then the vertices v_u and v_x are distinct.) Set the capacity of each edge to 1. Let the source be the 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 24.1-7

We construct G' by splitting each vertex v of G into two vertices v_1, v_2, joined by an edge of capacity $l(v)$. All incoming edges of v are now incoming edges to v_1. All outgoing edges from v are now outgoing edges from v_2.

More formally, construct $G' = (V', E')$ with capacity function c' as follows. For every $v \in V$, create two vertices v_1, v_2 in V'. Add an edge (v_1, v_2) in E' with $c'(v_1, v_2) = l(v)$. For every edge $(u, v) \in E$, create an edge (u_2, v_1) in E' with capacity $c'(u_2, v_1) = c(u, v)$. Make s_1 and t_2 as the new source and target vertices in G'. Clearly, $|V'| = 2|V|$ and $|E'| = |E| + |V|$.

Let f be a flow in G that respects vertex capacities. Create a flow function f' in G' as follows. For each edge $(u, v) \in G$, let $f'(u_2, v_1) = f(u, v)$. For each vertex $u \in V - \{t\}$, let $f'(u_1, u_2) = \sum_{v \in V} f(u, v)$. Let $f'(t_1, t_2) = \sum_{v \in V} f(v, t)$.

We readily see that there is a one-to-one correspondence between flows that respect vertex capacities in G and flows in G'. For the capacity constraint, every edge in G' of the form (u_2, v_1) has a corresponding edge in G with a corresponding capacity and flow and thus satisfies the capacity constraint. For edges in E' of the form (u_1, u_2), the capacities reflect the vertex capacities in G. Therefore, for $u \in V - \{s, t\}$, we have $f'(u_1, u_2) = \sum_{v \in V} f(u, v) \le l(u) = c'(u_1, u_2)$. We also have $f'(t_1, t_2) = \sum_{v \in V} f(v, t) \le l(t) = c'(t_1, t_2)$. Note that this constraint also enforces the vertex capacities in G.

Now, we prove flow conservation. By construction, every vertex of the form u_1 in G' has exactly one outgoing edge (u_1, u_2), and every incoming edge to u_1 corresponds to an incoming edge of $u \in G$. Thus, for all vertices $u \in V - \{s, t\}$, we have

$$\begin{aligned}
\text{incoming flow to } u_1 &= \sum_{v \in V'} f'(v, u_1) \\
&= \sum_{v \in V} f(v, u) \\
&= \sum_{v \in V} f(u, v) \qquad \text{(because } f \text{ obeys flow conservation)} \\
&= f'(u_1, u_2) \\
&= \text{outgoing flow from } u_1 \ .
\end{aligned}$$

For t_1, we have

$$\begin{aligned}
\text{incoming flow} &= \sum_{v \in V'} f'(v, t_1) \\
&= \sum_{v \in V} f(v, t) \\
&= f'(t_1, t_2) \\
&= \text{outgoing flow} \ .
\end{aligned}$$

Vertices of the form u_2 have exactly one incoming edge (u_1, u_2), and every outgoing edge of u_2 corresponds to an outgoing edge of $u \in G$. Thus, for $u_2 \ne t_2$,

$$
\begin{aligned}
\text{incoming flow} &= f'(u_1, u_2) \\
&= \sum_{v \in V} f(u, v) \\
&= \sum_{v \in V'} f'(u_2, v) \\
&= \text{outgoing flow} .
\end{aligned}
$$

Finally, we prove that $|f'| = |f|$:

$$
\begin{aligned}
|f'| &= \sum_{v \in V'} f'(s_1, v) \\
&= f'(s_1, s_2) \qquad \text{(because there are no other outgoing edges from } s_1\text{)} \\
&= \sum_{v \in V} f(s, v) \\
&= |f| .
\end{aligned}
$$

Solution to Exercise 24.2-1

Lemma

1. If $v \notin V_l(u)$, then $f(u, v) = 0$.
2. If $v \notin V_e(u)$, then $f(v, u) = 0$.
3. If $v \notin V_l(u) \cup V_e(u)$, then $f'(u, v) = 0$.
4. If $v \notin V_l(u) \cup V_e(u)$, then $f'(v, u) = 0$.

Proof

1. Let $v \notin V_l(u)$ be some vertex. From the definition of $V_l(u)$, there is no edge in G from u to v. Thus, $f(u, v) = 0$.
2. Let $v \notin V_e(u)$ be some vertex. From the definition of $V_e(u)$, there is no edge in G from v to u. Thus, $f(v, u) = 0$.
3. Let $v \notin V_l(u) \cup V_e(u)$ be some vertex. From the definition of $V_l(u)$ and $V_e(u)$, neither (u, v) nor (v, u) exists. Therefore, the third condition of the definition of residual capacity (equation (24.2)) applies, and $c_f(u, v) = 0$. Thus, $f'(u, v) = 0$.
4. Let $v \notin V_l(u) \cup V_e(u)$ be some vertex. By equation (24.2), we have that $c_f(v, u) = 0$ and thus $f'(v, u) = 0$. ■ (lemma)

We conclude that the summations in equation (24.6) equal the summations in equation (24.5).

Solution to Exercise 24.2-2

The flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$ is 19 (with 23 units going from $\{s, v_2, v_4\}$ to $\{v_1, v_3, t\}$ and 4 units going back), and the capacity of the cut is 31.

Solution to Exercise 24.2-4

$(\{s, v_1, v_2, v_4\}, \{v_3, t\})$ is the minimum cut for the flow. The augmenting path in part (c) cancels flow previously shipped from v_2 to v_1 in part (b).

Solution to Exercise 24.2-6

In the construction shown in Figure 24.3, set $c(s, s_i) = p_i$ for each source s_i, and set $c(t_j, t) = q_j$ for each sink t_j.

Solution to Exercise 24.2-7

We need to show that f_p obeys the capacity constraint and flow conservation. For the capacity constraint, $f_p(u, v) = 0$ if edge (u, v) is not in the augmenting path p, so that it obeys the capacity constraint. If (u, v) is in path p, then $f_p(u, v) = c_f(p) \le c_f(u, v)$, so that again it obeys the capacity constraint.

For flow conservation, if vertex $u \in V - \{s, t\}$ is not on path p, then for all $v \in V$, we have $f_p(u, v) = f_p(v, u) = 0$. If $u \in V - \{s, t\}$ is on path p, let x and y be u's predecessor and successor, respectively, in p, so that $f_p(x, u) = f_p(u, y) = c_f(p)$. Then, we have

$$\begin{aligned}
&\sum_{v \in V} f_p(v, u) - \sum_{v \in V} f_p(u, v) \\
&\quad = \left(\sum_{v \in V - \{x\}} f_p(v, u) + f_p(x, u) \right) - \left(\sum_{v \in V - \{y\}} f_p(u, v) + f_p(u, y) \right) \\
&\quad = (0 + f_p(x, u)) - (0 + f_p(u, y)) \\
&\quad = c_f(p) - c_f(p) \\
&\quad = 0 ,
\end{aligned}$$

so that f_p obeys flow conservation.

Solution to Exercise 24.2-8

Let G_f be the residual network just before an iteration of the **while** loop of FORD-FULKERSON, and let E_s be the set of residual edges of G_f into s. We'll show that the augmenting path p chosen by FORD-FULKERSON does not include an edge in E_s. Thus, even if we redefine G_f to disallow edges in E_s, the path p still remains an augmenting path in the redefined network. Since p remains unchanged, an iteration of the **while** loop of FORD-FULKERSON updates the flow in the same way as before the redefinition. Furthermore, by disallowing some edges, we do not introduce any new augmenting paths. Thus, FORD-FULKERSON still correctly computes a maximum flow.

Now, we prove that FORD-FULKERSON never chooses an augmenting path p that includes an edge $(v, s) \in E_s$. Why? The path p always starts from s, and if p included an edge (v, s), the vertex s would be repeated twice in the path. Thus, p would no longer be a *simple* path. Since FORD-FULKERSON chooses only simple paths, p cannot include (v, s).

Solution to Exercise 24.2-9

The augmented flow $f \uparrow f'$ satisfies the flow conservation property but not the capacity constraint property.

First, we prove that $f \uparrow f'$ satisfies the flow conservation property. We note that if edge $(u, v) \in E$, then $(v, u) \notin E$ and $f'(v, u) = 0$. Thus, we can rewrite the definition of flow augmentation (equation (24.4)), when applied to two flows, as

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) & \text{if } (u, v) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

The definition implies that the new flow on each edge is simply the sum of the two flows on that edge. We now prove that in $f \uparrow f'$, the net incoming flow for each vertex equals the net outgoing flow. Let $u \notin \{s, t\}$ be any vertex of G. We have

$$\begin{aligned}
&\sum_{v \in V} (f \uparrow f')(v, u) \\
&\quad = \sum_{v \in V} (f(v, u) + f'(v, u)) \\
&\quad = \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(v, u) \\
&\quad = \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \quad \text{(because } f,\ f' \text{ obey flow conservation)} \\
&\quad = \sum_{v \in V} (f(u, v) + f'(u, v)) \\
&\quad = \sum_{v \in V} (f \uparrow f')(u, v) .
\end{aligned}$$

We conclude that $f \uparrow f'$ satisfies flow conservation.

We now show that $f \uparrow f'$ need not satisfy the capacity constraint by giving a simple counterexample. Let the flow network G have just a source and a target vertex, with a single edge (s, t) having $c(s, t) = 1$. Define the flows f and f' to have $f(s, t) = f'(s, t) = 1$. Then, we have $(f \uparrow f')(s, t) = 2 > c(s, t)$. We conclude that $f \uparrow f'$ need not satisfy the capacity constraint.

Solution to Exercise 24.2-10

[This solution is identical to the solution to Problem 24-6, part (b).]

Imagine the flow augmentation process in reverse. Start with a maximum flow, and create a flow network $G' = (V, E')$, where E' comprises only the edges carrying a positive flow. Find a path p from s to t in G', and reduce the flow on every edge in p by the minimum capacity of any edge in p. At least one edge in p will then have a flow of 0. Repeat this process, starting with G', until all edges have a flow of 0. Since each step removes at least one edge, at most $|E|$ steps suffice to reduce all flows to 0. These augmenting paths, at most $|E|$ of them, produce the original maximum flow.

Solution to Exercise 24.2-11

This solution is also posted publicly

For any two vertices u and v in G, we can define a flow network G_{uv} consisting of the directed version of G with $s = u$, $t = v$, and all edge capacities set to 1. Because a flow network may not have antiparallel edges, for each edge in G, one of the directed edges in G_{uv} must be broken into two edges, with a new vertex added. Therefore, G_{uv} has $|V| + |E|$ vertices and $3|E|$ edges, so that it has $O(V + E)$ vertices and $O(E)$ edges, as required. Set all capacities in G_{uv} to be 1 so that the number of edges of G crossing a cut equals the capacity of the cut in G_{uv}. Let f_{uv} denote a maximum flow in G_{uv}.

We claim that the edge connectivity k equals $\min\{|f_{uv}| : v \in V - \{u\}\}$ for any vertex $u \in V$. We'll show below that this claim holds. Assuming that it holds, we can find k as follows:

```
EDGE-CONNECTIVITY(G)
  $k = \infty$
  select any vertex $u \in G.V$
  for each vertex $v \in G.V - \{u\}$
      set up the flow network $G_{uv}$ as described above
      find the maximum flow $f_{uv}$ on $G_{uv}$
      $k = \min\{k, |f_{uv}|\}$
  return $k$
```

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 $k = \min\{|f_{uv}| : v \in V - \{u\}\}$, 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 $k \geq \min\{|f_{uv}| : v \in V - \{u\}\}$:

  Let $m = \min\{|f_{uv}| : v \in V - \{u\}\}$. 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 vertex 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., $k \geq \min\{|f_{uv}| : v \in V - \{u\}\}$.

- Proof that $k \le \min\{|f_{uv}| : v \in V - \{u\}\}$:

  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 $k \le \min\{|f_{uv}| : v \in V - \{u\}\}$.
- Thus, the claim that $k = \min\{|f_{uv}| : v \in V - \{u\}\}$, for any $u \in V$ is true.

Solution to Exercise 24.2-12

The idea of the proof is that if $f(v,s) = 1$, then there must exist a cycle containing the edge (v,s) and for which each edge carries one unit of flow. If we reduce the flow on each edge in the cycle by one unit, we can reduce $f(v,s)$ to 0 without affecting the value of the flow.

Given the flow network G and the flow f, we say that vertex y is *flow-connected* to vertex z if there exists a path p from y to z such that each edge of p has a positive flow on it. We also define y to be flow-connected to itself. In particular, s is flow-connected to s.

We start by proving the following lemma:

Lemma

Let $G = (V, E)$ be a flow network and f be a flow in G. If s is not flow-connected to v, then $f(v,s) = 0$.

Proof The idea is that since s is not flow-connected to v, there cannot be any flow from s to v. By using flow conservation, we will prove that there cannot be any flow from v to s either, and thus that $f(v,s) = 0$.

Let Y be the set of all vertices y such that s is flow-connected to y. By applying flow conservation to vertices in $V - Y$ and taking the sum, we obtain

$$\sum_{z \in V-Y} \sum_{x \in V} f(x,z) = \sum_{z \in V-Y} \sum_{x \in V} f(z,x) .$$

Partitioning V into Y and $V - Y$ gives

$$\sum_{z \in V-Y} \sum_{x \in V-Y} f(x,z) + \sum_{z \in V-Y} \sum_{x \in Y} f(x,z) = \sum_{z \in V-Y} \sum_{x \in V-Y} f(z,x) + \sum_{z \in V-Y} \sum_{x \in Y} f(z,x) . \qquad (\dagger)$$

But we have

$$\sum_{z \in V-Y} \sum_{x \in V-Y} f(x,z) = \sum_{z \in V-Y} \sum_{x \in V-Y} f(z,x) ,$$

since the left-hand side is the same as the right-hand side, except for a change of variable names x and z. We also have

$$\sum_{z \in V-Y} \sum_{x \in Y} f(x,z) = 0 ,$$

since the flow from any vertex in Y to any vertex in $V - Y$ must be 0. Thus, equation (†) simplifies to

$$\sum_{z \in V-Y} \sum_{x \in Y} f(z, x) = 0 \; .$$

The above equation implies that $f(z, x) = 0$ for each $z \in V - Y$ and $x \in Y$. In particular, since $v \in V - Y$ and $s \in Y$, we have that $f(v, s) = 0$. ■

Now, we show how to construct the required flow f'. By the contrapositive of the lemma, $f(v, s) > 0$ implies that s is flow-connected to v through some path p. Let path p' be the path $s \overset{p}{\leadsto} v \to s$. Path p' is a cycle that has positive flow on each edge. Because we assume that all edge capacities are integers, the flow on each edge of p' is at least 1. If we subtract 1 from each edge of the cycle to obtain a flow f', then f' still satisfies the properties of a flow network and has the same value as $|f|$. Because edge (v, s) is in the cycle, we have that $f'(v, s) = f(v, s) - 1 = 0$.

The algorithm, therefore, is to find a path p from s to v in which each edge carries positive flow, using either breadth-first search or depth-first search. Assuming that each vertex has at least one entering edge or one leaving edge, the undirected version of the residual graph G_f is connected, and so $|E_f| \geq |V| - 1$. The search for path p then takes $O(E)$ time. Since the cycle $s \overset{p}{\leadsto} v \to s$ has at most $|E|$ edges, we can reduce the flow on all edges in the cycle in $O(E)$ time.

Solution to Exercise 24.2-13

Let (S, T) and (X, Y) be two cuts in G (and G'). Let c' be the capacity function of G'. One way to define c' is to add a small amount δ to the capacity of each edge in G. That is, if u and v are two vertices, we set

$$c'(u, v) = c(u, v) + \delta \; .$$

Thus, if $c(S, T) = c(X, Y)$ and (S, T) has fewer edges than (X, Y), then we would have $c'(S, T) < c'(X, Y)$. We have to be careful and choose a small δ, lest we change the relative ordering of two unequal capacities. That is, if $c(S, T) < c(X, Y)$, then no matter many more edges (S, T) has than (X, Y), we still need to have $c'(S, T) < c'(X, Y)$. With this definition of c', a minimum cut in G' will be a minimum cut in G that has the minimum number of edges.

How should we choose the value of δ? Let m be the minimum difference between capacities of two unequal-capacity cuts in G. Choose $\delta = m/(2|E|)$. For any cut (S, T), since the cut can have at most $|E|$ edges, we can bound $c'(S, T)$ by

$$c(S, T) \leq c'(S, T) \leq c(S, T) + |E| \cdot \delta \; .$$

Let $c(S, T) < c(X, Y)$. We need to prove that $c'(S, T) < c'(X, Y)$. We have

$$\begin{aligned} c'(S, T) &\leq c(S, T) + |E| \cdot \delta \\ &= c(S, T) + m/2 \\ &< c(X, Y) \qquad (\text{since } c(X, Y) - c(S, T) \geq m) \\ &\leq c'(X, Y) \; . \end{aligned}$$

Because all capacities are integral, we can choose $m = 1$, obtaining $\delta = 1/2|E|$. To avoid dealing with fractional values, we can scale all capacities by $2|E|$ to obtain

$$c'(u, v) = 2|E| \cdot c(u, v) + 1 .$$

Solution to Exercise 24.3-2

The proof is by inducation on the number of iterations of the Ford-Fulkerson method.

Basis: The initial flow f equals 0 in all edges, so that $|f| = 0$.

Inductive step: By the inductive hypothesis, at the start of each iteration, $f(u, v)$ is an integer for all $u, v \in V$, so that $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$ is also an integer. Moreover, all residual capacities are integer as well, since they are either $f(u, v)$ for some $u, v \in V$ or $c(u, v) - f(u, v)$ for some $u, v \in V$ and $c(u, v)$ is an integer. Each iteration of the Ford-Fulkerson method updates the flow on each edge by $c_f(p)$, where p is the augmenting path found. Since $c_f(p)$ equals the residual capacity of some edge in p, it is an integer. Therefore, the change in $f(u, v)$ for each $u, v \in V$ is plus or minus an integer value, so that it remains an integer value.

Solution to Exercise 24.3-3
This solution is also posted publicly

By definition, an augmenting path is a simple path $s \leadsto t$ in the residual network 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 Problem 24-1

a. *[This part of the problem is the same as Exercise 24.1-7. The solution to the exercise is repeated here with minor changes at the beginning.]*

Let G be a flow network with vertex and edge capacities, and let $l(v)$ denote the capacity of vertex v. We construct a flow network G' with only edge capacities by splitting each vertex v of G into two vertices v_1, v_2, joined by an edge of capacity $l(v)$. All incoming edges of v are now incoming edges to v_1. All outgoing edges from v are now outgoing edges from v_2.

More formally, construct $G' = (V', E')$ with capacity function c' as follows. For every $v \in V$, create two vertices v_1, v_2 in V'. Add an edge (v_1, v_2) in E' with $c'(v_1, v_2) = l(v)$. For every edge $(u, v) \in E$, create an edge (u_2, v_1) in E' with capacity $c'(u_2, v_1) = c(u, v)$. Make s_1 and t_2 as the new source and target vertices in G'. Clearly, $|V'| = 2\,|V|$ and $|E'| = |E| + |V|$.

Let f be a flow in G that respects vertex capacities. Create a flow function f' in G' as follows. For each edge $(u, v) \in G$, let $f'(u_2, v_1) = f(u, v)$. For each vertex $u \in V - \{t\}$, let $f'(u_1, u_2) = \sum_{v \in V} f(u, v)$. Let $f'(t_1, t_2) = \sum_{v \in V} f(v, t)$.

We readily see that there is a one-to-one correspondence between flows that respect vertex capacities in G and flows in G'. For the capacity constraint, every edge in G' of the form (u_2, v_1) has a corresponding edge in G with a corresponding capacity and flow and thus satisfies the capacity constraint. For edges in E' of the form (u_1, u_2), the capacities reflect the vertex capacities in G. Therefore, for $u \in V - \{s, t\}$, we have $f'(u_1, u_2) = \sum_{v \in V} f(u, v) \le l(u) = c'(u_1, u_2)$. We also have $f'(t_1, t_2) = \sum_{v \in V} f(v, t) \le l(t) = c'(t_1, t_2)$. Note that this constraint also enforces the vertex capacities in G.

Now, we prove flow conservation. By construction, every vertex of the form u_1 in G' has exactly one outgoing edge (u_1, u_2), and every incoming edge to u_1 corresponds to an incoming edge of $u \in G$. Thus, for all vertices $u \in V - \{s, t\}$, we have

$$\begin{aligned}
\text{incoming flow to } u_1 &= \sum_{v \in V'} f'(v, u_1) \\
&= \sum_{v \in V} f(v, u) \\
&= \sum_{v \in V} f(u, v) \qquad \text{(because } f \text{ obeys flow conservation)} \\
&= f'(u_1, u_2) \\
&= \text{outgoing flow from } u_1 \,.
\end{aligned}$$

For t_1, we have

$$\begin{aligned}
\text{incoming flow} &= \sum_{v \in V'} f'(v, t_1) \\
&= \sum_{v \in V} f(v, t) \\
&= f'(t_1, t_2) \\
&= \text{outgoing flow} \,.
\end{aligned}$$

Vertices of the form u_2 have exactly one incoming edge (u_1, u_2), and every outgoing edge of u_2 corresponds to an outgoing edge of $u \in G$. Thus, for $u_2 \ne t_2$,

$$\text{incoming flow} = f'(u_1, u_2)$$

$$
\begin{aligned}
&= \sum_{v \in V} f(u, v) \\
&= \sum_{v \in V'} f'(u_2, v) \\
&= \text{outgoing flow} .
\end{aligned}
$$

Finally, we prove that $|f'| = |f|$:

$$
\begin{aligned}
|f'| &= \sum_{v \in V'} f'(s_1, v) \\
&= f'(s_1, s_2) \qquad \text{(because there are no other outgoing edges from } s_1\text{)} \\
&= \sum_{v \in V} f(s, v) \\
&= |f| .
\end{aligned}
$$

b. To solve the escape problem, convert the grid to a multiple-source, multiple-sink maximum-flow problem as follows:

- Number the rows and columns from 1 to n and denote the vertex in row i and column j by v_{ij}. The undirected grid has vertical edges $(v_{ij}, v_{i+1,j})$ for $i = 1, \ldots, n-1$ and $j = 1, \ldots, n$ and horizontal edges $(v_{ij}, v_{i,j+1})$ for $i = 1, \ldots, n$ and $j = 1, \ldots, n-1$.
- For each vertical edge $(v_{ij}, v_{i+1,j})$, create a new vertex v'_{ij} and three directed edges $(v_{ij}, v_{i+1,j})$, $(v_{i+1,j}, v'_{ij})$ and (v'_{ij}, v_{ij}), each with capacity 1. The extra vertices and their incident edges are necessary to avoid antiparallel edges. For each horizontal edge $(v_{ij}, v_{i,j+1})$, create a new vertex v''_{ij} and three directed edges $(v_{ij}, v_{i,j+1})$, $(v_{i,j+1}, v''_{ij})$ and (v''_{ij}, v_{ij}), each also with capacity 1. The number of added vertices is $2n(n-1) = 2n^2 - 2n$, so that the total number of vertices is $3n^2 - 2n$. To total up the edges, observe that for each added vertex, there are three edges, so that there are $6n^2 - 6n$ edges. Finally, set the capacities of all vertices to 1.
- Designate each starting point in the escape problem as a source, and designate each of the $4n - 4$ boundary points in the escape problem as a sink. Then convert this multiple-source, multiple-sink network to a single-source, single-sink network as shown in Section 24.1. The resulting flow network comprises $3n^2 - 2n + 2$ vertices (the 2 extra vertices are the supersource and supersink) and $6n^2 - 2n - 4 + m$ edges (1 edge from each of the $4n - 4$ boundary vertices to the supersink and 1 edge from the supersource to each of the m starting points).
- Run Ford-Fulkerson on the resulting network, which has $\Theta(n^2)$ vertices and $\Theta(n^2)$ edges (since $m = O(n^2)$). Because the value of the maximum flow is at most m and all edge capacities are integers (1, in fact), the running time is $O(n^2 m)$, which is $O(n^4)$. Since the input size is $\Theta(n^2)$, the running time is at most quadratic in the input size. The value of the maximum flow found equals the number of vertex-disjoint paths from the starting points to the boundary. Thus, there are m vertex-disjoint such paths if and only if the value of the maximum flow is at least m.

Solution to Problem 24-2

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. Vertex x_i tells us about edges leaving i, and vertex 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, for the following reasons:

- In a bipartite matching, no vertex is used more than once. In a bipartite matching in G'', since no x_i is used more than once, at most one edge in the matching leaves any vertex i in G. Similarly, since no y_j is used more than once, at most one edge in the matching enters any vertex j in G.
- In a path cover, since no vertex appears in more than one path, 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 its 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, for the following reasons:

- 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. When building a path, we start or enter a vertex and then leave 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.

$$\begin{aligned}
|V| &= \sum_{p \in P} (\text{\# vertices in } p) && \text{(every vertex is on exactly 1 path)} \\
&= \sum_{p \in P} (1 + \text{\# edges in } p) \\
&= \sum_{p \in P} 1 + \sum_{p \in P} (\text{\# edges in } p) \\
&= |P| + |M| && \text{(by 1-to-1 correspondence)} \\
&= |P| + |f| && \text{(by Lemma 24.9)} .
\end{aligned}$$

Thus, for the fixed set V in our graph G, $|P|$ (the number of paths) is minimized when the flow f is maximized.

The overall algorithm is as follows:

- Use FORD-FULKERSON to find a maximum flow in G' and 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: $4 \rightarrow 1 \rightarrow 2 \rightarrow 3$, 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 that a maximum bipartite matching can have only 3 edges (and the maximum flow in G' has value 3). In fact, there are 2 possible maximum matchings. It is always possible to match (x_1, y_2) and (x_2, y_3), and then either (x_3, y_1) or (x_4, y_1) can be chosen, but not both.

Solution to Problem 24-3

a. Assume for the sake of contradiction that $C_k \notin T$ for some $C_k \in R_i$. Since $C_k \notin T$, we must have $C_k \in S$. On the other hand, we have $J_i \in T$. Thus, the edge (C_k, J_i) crosses the cut (S, T). But $c(C_k, J_i) = \infty$ by construction, which contradicts the assumption that (S, T) is a *finite*-capacity cut.

b. Let us define a ***project-plan*** as a set of jobs to accept and experts to hire. Let P be a project-plan. We assume that P has two attributes. The attribute $P.J$ denotes the set of accepted jobs, and $P.E$ denotes the set of hired experts.

A ***valid*** project-plan is one in which all experts that are required by the accepted jobs are hired. Specifically, let P be a valid project plan. If $J_i \in P.J$, then $C_k \in P.E$ for each $C_k \in R_i$. Note that Professor Fieri might decide to hire more experts than those that are actually required.

We define the ***revenue*** of a project-plan as the total profit from the accepted jobs minus the total cost of the hired experts. The problem asks us to find a valid project plan with maximum revenue.

We start by proving the following lemma, which establishes the relationship between the capacity of a cut in flow network G and the revenue of a valid project-plan.

Lemma (Min-cut max-revenue)

There exists a finite-capacity cut (S, T) of G with capacity $c(S, T)$ if and only if there exists a valid project-plan with net revenue $\left(\sum_{J_i \in J} p_i\right) - c(S, T)$.

Proof Let (S, T) be a finite-capacity cut of G with capacity $c(S, T)$. We prove one direction of the lemma by constructing the required project-plan.

Construct the project-plan P by including J_i in $P.J$ if and only if $J_i \in T$ and including C_k in $P.E$ if and only if $C_k \in T$. From part (a), P is a valid project-plan, since, for every $J_i \in P.J$, we have $C_k \in P.E$ for each $C_k \in R_i$.

Since the capacity of the cut is finite, there cannot be any edges of the form (C_k, J_i) crossing the cut, where $C_k \in S$ and $J_i \in T$. All edges going from a vertex in S to a vertex in T must be either of the form (s, C_k) or of the form (J_i, t). Let E_C be the set of edges of the form (s, C_k) that cross the cut, and let E_J be the set of edges of the form (J_i, t) that cross the cut, so that

$$c(S, T) = \sum_{(s, C_k) \in E_C} c(s, C_k) + \sum_{(J_i, t) \in E_J} c(J_i, t) .$$

Consider edges of the form (s, C_k). We have

$$\begin{aligned} (s, C_k) \in E_C \quad &\text{if and only if} \quad C_k \in T \\ &\text{if and only if} \quad C_k \in P.E . \end{aligned}$$

By construction, $c(s, C_k) = e_k$. Taking summations over E_C and over $P.E$, we obtain

$$\sum_{(s, C_k) \in E_C} c(s, C_k) = \sum_{C_k \in P.E} e_k .$$

Similarly, consider edges of the form (J_i, t). We have

$$\begin{aligned} (J_i, t) \in E_J & \text{ if and only if } J_i \in S \\ & \text{ if and only if } J_i \notin T \\ & \text{ if and only if } J_i \notin P.J \ . \end{aligned}$$

By construction, $c(J_i, t) = p_i$. Taking summations over E_J and over $P.J$, we obtain

$$\sum_{(J_i,t) \in E_J} c(J_i, t) = \sum_{J_i \notin P.J} p_i \ .$$

Let v be the net revenue of P. Then, we have

$$\begin{aligned} v &= \sum_{J_i \in P.J} p_i - \sum_{C_k \in P.E} e_k \\ &= \left(\sum_{J_i \in J} p_i - \sum_{J_i \notin P.J} p_i \right) - \sum_{C_k \in P.E} e_k \\ &= \sum_{J_i \in J} p_i - \left(\sum_{J_i \notin P.J} p_i + \sum_{C_k \in P.E} e_k \right) \\ &= \sum_{J_i \in J} p_i - \left(\sum_{(J_i,t) \in E_J} c(J_i, t) + \sum_{(s,C_k) \in E_C} c(s, C_k) \right) \\ &= \left(\sum_{J_i \in J} p_i \right) - c(S, T) \ . \end{aligned}$$

Now, we prove the other direction of the lemma by constructing the required cut from a valid project-plan.

Construct the cut (S, T) as follows. For every $J_i \in P.J$, let $J_i \in T$. For every $C_k \in P.E$, let $C_k \in T$.

First, we prove that the cut (S, T) is a finite-capacity cut. Since edges of the form (C_k, J_i) are the only infinite-capacity edges, it suffices to prove that there are no edges (C_k, J_i) such that $C_k \in S$ and $J_i \in T$.

For the purpose of contradiction, assume there is an edge (C_k, J_i) such that $C_k \in S$ and $J_i \in T$. By our constuction, we must have $J_i \in P.J$ and $C_k \notin P.E$. But since the edge (C_k, J_i) exists, we have $C_k \in R_i$. Since P is a valid project-plan, we derive the contradiction that C_k must have been in $P.E$.

From here on, the analysis is the same as the previous direction. In particular, the last equation from the previous analysis holds: the net revenue v equals $\left(\sum_{J_i \in J} p_i\right) - c(S, T)$. ■

We conclude that the problem of finding a maximum-revenue project-plan reduces to the problem of finding a minimum cut in G. Let (S, T) be a minimum cut. From the lemma, the maximum net revenue is given by

$$\left(\sum_{J_i \in J} p_i \right) - c(S, T) \ .$$

c. Construct the flow network G as shown in the problem statement. Obtain a minimum cut (S, T) by running any of the maximum-flow algorithms (say, Edmonds-Karp). Construct the project plan P as follows: add J_i to $P.J$ if and only if $J_i \in T$. Add C_k to $P.E$ if and only if $C_k \in T$.

First, we note that the number of vertices in G is $|V| = m + n + 2$, and the number of edges in G is $|E| = r + m + n$. Constructing G and recovering the project-plan from the minimum cut are clearly linear-time operations. The running time of our algorithm is thus asymptotically the same as the running time of the algorithm used to find the minimum cut. If we use Edmonds-Karp to find the minimum cut, the running time is $O(VE^2)$. Assuming that $|R_i| \geq 1$ for $i = 1, \ldots, m$ gives $r \geq m$, and the running time is $O((r + n)^3)$.

Solution to Problem 24-4

This solution is also posted publicly

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 network. So look for an augmenting path and update the flow if a path is found.

Time

$O(V + E) = O(E)$ by finding 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 $f(u, v) < c(u, v)$, we don't need to do anything.

If $f(u, v) = c(u, v)$, we need to update the maximum flow. Because $c(u, v)$ is an integer that decreases, it must be at least 1, so that $f(u, v) = c(u, v) \geq 1$.

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 flow conservation at u (unless $u = s$) and at v (unless $v = t$). 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 1 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 $u \rightsquigarrow s$ in G_f and augment the flow along that path by 1. (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 $t \rightsquigarrow v$ in G_f and augmenting the flow along that path by 1.

Time

$O(V + E) = O(E)$ by finding the paths with either DFS or BFS.

Solution to Problem 24-5

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 that we are looking for an augmenting path whose edges *all* have capacity at least K. Perform 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 with capacity at least 1. Since all the capacities are integers, and the capacity of an augmenting path is positive, when there are no augmenting paths with capacity at least 1, there must be no augmenting paths whatsoever in the residual network. 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 $K = 2^{\lfloor \lg C \rfloor}$, and so $2K = 2 \cdot 2^{\lfloor \lg C \rfloor} = 2^{\lfloor \lg C \rfloor + 1} > 2^{\lg C} = C$. Thus, 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, and 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. Thus, we want to show that when line 7 is reached, the capacity of a minimum cut of G_f is at most $K\,|E|$.

  Let G_f be the residual network when line 7 is reached. Upon reaching line 7, G_f contains no augmenting path with capacity at least K. Therefore, a maximum flow f' in G_f has value $|f'| < K\,|E|$. Then, by the max-flow min-cut theorem, a minimum cut in G_f has capacity less than $K\,|E|$.

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 has value at most $2K\,|E|$. The following lemma shows that the value of a maximum flow in G equals the value of the current flow f in G plus the value of a maximum flow in G_f.

Lemma

Let f be a flow in flow network G, and f' be a maximum flow in the residual network G_f. Then $f \uparrow f'$ is a maximum flow in G.

Proof By the max-flow min-cut theorem, $|f'| = c_f(S,T)$ for some cut (S,T) of G_f, which is also a cut of G. By Lemma 24.4, $|f| = f(S,T)$. By Lemma 24.1, $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$. We will show that $|f| + |f'| = c(S,T)$ which, by the max-flow min-cut theorem, will prove that $f \uparrow f'$ is a maximum flow in G.

We have

$$\begin{aligned}
|f| + |f'| &= f(S,T) + c_f(S,T) \\
&= \left(\sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u)\right) + \sum_{u\in S}\sum_{v\in T} c_f(u,v) \\
&= \left(\sum_{u\in S, v\in T} f(u,v) - \sum_{u\in S, v\in T} f(v,u)\right) \\
&\quad + \left(\sum_{\substack{u\in S, v\in T,\\ (u,v)\in E}} c(u,v) - \sum_{\substack{u\in S, v\in T,\\ (u,v)\in E}} f(u,v) + \sum_{\substack{u\in S, v\in T,\\ (v,u)\in E}} f(v,u)\right) .
\end{aligned}$$

Noting that $(u,v) \notin E$ implies $f(u,v) = 0$, we have that

$$\sum_{u\in S, v\in T} f(u,v) = \sum_{\substack{u\in S, v\in T,\\ (u,v)\in E}} f(u,v) .$$

Similarly,

$$\sum_{u\in S, v\in T} f(v,u) = \sum_{\substack{u\in S, v\in T,\\ (v,u)\in E}} f(v,u) .$$

Thus, the summations of $f(u,v)$ cancel each other out, as do the summations of $f(v,u)$. Therefore,

$$\begin{aligned}
|f| + |f'| &= \sum_{\substack{u\in S, v\in T,\\ (u,v)\in E}} c(u,v) \\
&= \sum_{u\in S}\sum_{v\in T} c(u,v) \\
&= c(S,T) .
\end{aligned}$$

■ (lemma)

By this lemma, we see that the value of a maximum flow in G is at most $2K\,|E|$ more than the value of the current flow f in G. Every time the inner **while** loop finds an augmenting path of capacity at least K, the flow in G increases by at least 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 **while** 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 $O(E^2 \lg C)$.

Solution to Problem 24-6

a. Start with simple changes to the INITIALIZE-SINGLE-SOURCE and RELAX procedures:

```
INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex $v \in G.V$
      $v.d = 0$
      $v.\pi = \text{NIL}$
  $s.d = \infty$
```

```
RELAX(u, v, w)
  if $v.d < \min\{u.d, c(u,v)\}$
      $v.d = \min\{u.d, c(u,v)\}$
      $v.\pi = u$
```

And change lines 7, 11, and 12 in the DIJKSTRA procedure:

```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  $S = \emptyset$
  $Q = \emptyset$
  for each vertex $u \in G.V$
      INSERT(Q, u)
  while $Q \neq \emptyset$
      $u$ = EXTRACT-MAX(Q)
      $S = S \cup \{u\}$
      for each vertex $v$ in $G.Adj[u]$
          RELAX(u, v, w)
          if the call of RELAX increased $v.d$
              INCREASE-KEY(Q, v, v.d)
```

b. Imagine the flow augmentation process in reverse. Start with a maximum flow, and create a flow network $G' = (V, E')$, where E' comprises only the edges carrying a positive flow. Find a path p from s to t in G', and reduce the flow on every edge in p by the minimum capacity of any edge in p. At least one edge in p will then have a flow of 0. Repeat this process, starting with G', until all edges have a flow of 0. Since each step removes at least one edge, at most $|E|$ steps suffice to reduce all flows to 0. These augmenting paths, at most $|E|$ of them, produce the original maximum flow.

c. Given a maximum flow f^* and a flow f, define the "remaining flow" f' by $f'(u, v) = f^*(u, v) - f(u, v)$ for all $(u, v) \in G_f$. This remaining flow f' is a flow in the residual graph G_f with flow value $|f'| = |f^*| - |f|$. By part (b), therefore, we can split f' into at most $|E|$ individual augmenting paths in G_f. The average residual capacity of these individual augmenting paths is at least $(|f^*| - |f|)/|E|$. There must be at least one augmenting path whose residual capacity is at least the average, and so some augmenting path p has residual capacity $c_f \geq (|f^*| - |f|)/|E|$.

d. By part (c), going from f_i to f_{i+1} reduces the value of the remaining flow by at least $(|f^*| - |f_i|)/|E|$, so that

$$\begin{aligned} |f^*| - |f_{i+1}| &\leq (|f^*| - |f_i|) - (|f^*| - |f_i|)/|E| \\ &= (|f^*| - |f_i|)(1 - 1/|E|) . \end{aligned}$$

Let $r_i = |f^*| - |f_i|$, so that $r_{i+1} \leq r_i(1 - 1/|E|)$ and $r_0 = |f^*|$. Iterating this formula starting from f_0 gives $r_i \leq r_0(1 - 1/|E|)^i$, so that $|f^*| - |f_i| \leq |f^*|(1 - 1/|E|)^i$.

e. By inequality (3.14), $1 + x < e^x$ for all $x \neq 0$. Thus, we have

$$\begin{aligned} |f^*| - |f_i| &\leq |f^*|(1 - 1/|E|)^i \\ &< |f^*|(e^{-1/|E|})^i \\ &= |f^*| e^{-i/|E|} . \end{aligned}$$

f. Letting $i = |E| \ln |f^*|$ gives

$$\begin{aligned} |f^*| - |f_{|E| \ln |f^*|}| &< |f^*| e^{-|E| \ln |f^*| / |E|} \\ &= |f^*| e^{-\ln |f^*|} \\ &= |f^*| \cdot 1/|f^*| \\ &= 1 . \end{aligned}$$

Because all capacities are integer, that means after augmenting the flow at most $|E| \ln |f^*|$ times, we have $|f^*| - |f_{|E| \ln |f^*|}| = 0$, so that the flow has achieved its maximum value.

Lecture Notes for Chapter 25: Matchings in Bipartite Graphs

In an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that every vertex in V has at most one incident edge in M.

This chapter considers matchings in bipartite graphs: $V = L \cup R$, and every edge is incident on one vertex in L and one vertex in R. A matching matches vertices in L with vertices in R.

In some applications, $|L| = |R|$. In other applications, L and R need not be the same size.

Example applications of matching

- Assigning job candidates to interview slots.
  - One vertex in L for each candidate.
  - One vertex in R for each interview slot.
  - Edge (l, r), where $l \in L$ and $r \in R$, if candidate l is available for an interview in slot r.
  - A matching assigns candidates to interview slots.
  - Want a ***maximum matching***: a matching of maximum size. That maximizes the number of candidates interviewed.
- U.S. National Resident Matching Program.
  - Assigns medical students to residencies in hospitals.
  - Each student ranks hospitals by preference.
  - Each hospital ranks students.
  - Goal is to assign students to hospitals so that never have a student and hospital that are not matched, yet each ranked the other higher than their match.
  - Example of the "stable-marriage problem."
- Assigning workers to tasks.
  - For each worker, have scores for how good the worker is at each task.
  - Assume equal numbers of workers and tasks.
  - Want to assign workers to tasks to maximize the overall score.

Matchings in non-bipartite graphs are also important, but not dealt with in this chapter. ***Example:*** Matching roommates.

Maximum matching in bipartite graphs

Section 24.3 uses maximum flow to find a maximum bipartite matching. Here, we see a more efficient method, the Hopcroft-Karp algorithm, which takes $O(\sqrt{V}E)$ time.

Definitions

Graph $G = (V, E)$, matching $M \subseteq E$.

- ***Matched vertex***: has an incident edge in M. Otherwise, the vertex is ***unmatched***.
- ***Maximal matching***: a matching that you cannot add any other edges to: if M is a matching, it is a maximal matching if for all $e \in E - M$, $M \cup \{e\}$ is not a matching. A maximum matching is always maximal, but a maximal matching is not always maximum.
- ***M-alternating path***: a simple path whose edges alternate between being in M and $E - M$.
- ***M-augmenting path***: an M alternating path whose first and last edges belong to $E - M$. (Also called an augmenting path with respect to M.) Contains 1 more edge in $E - M$ than in $M \Rightarrow$ has an odd number of edges.

M-augmenting paths

The key to Hopcroft-Karp. If you have a matching M and an M-augmenting path P, then make a new matching with 1 more edge than M by

- Removing from M the edges that are in P.
- Adding to M the edges in P that are in $E - M$.

Since P contains 1 more edge from $E - M$ than from M, the new matching has 1 edge more than the old matching.

Symmetric difference: For sets X, Y, $X \oplus Y = (X - Y) \cup (Y - X) = (X \cup Y) - (X \cap Y) =$ the elements in either X or Y, but not in both.

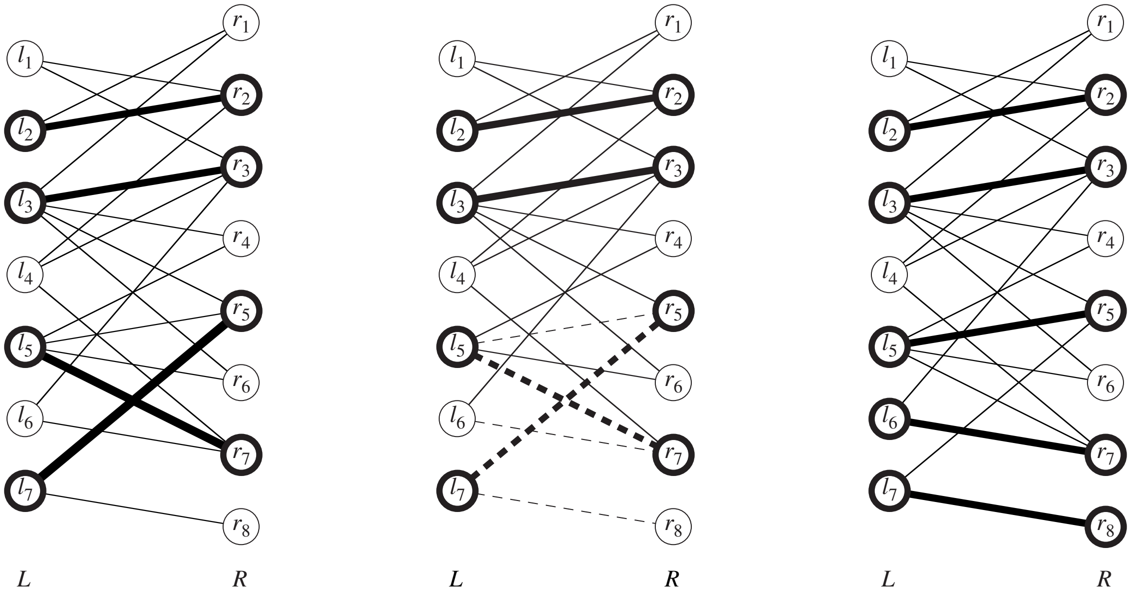
$\oplus$ is commutative and associative. Identity is the empty set.

Formally, use symmetric difference to create a new matching from M and an M-augmenting path:

Lemma

Let M be a matching and P be an M-augmenting path. Then $M' = M \oplus P$ is also a matching with $|M'| = |M| + 1$.

[Proof omitted, but the figure below shows the idea. The left part shows a matching M consisting of 4 edges, drawn with heavy lines and matched vertices drawn with heavy outlines. The middle part shows an M-augmenting path P, drawn with dashed lines. The right part shows the matching $M' = M \oplus P$ consisting of 5 edges. 2 edges leave the matching, and 3 edges enter, for a net gain of 1 edge.]



Corollary
Let M be a matching and $P_1, P_2, \ldots, P_k$ be vertex-disjoint augmenting paths. Then $M' = M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ is a matching with $|M'| = |M| + k$.

Proof Induction on i using the lemma shows that $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_{i-1})$ is a matching with $|M| + i - 1$ edges and P_i is an augmenting path with respect to $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_{i-1})$. ■

Examine the symmetric difference between 2 matchings:

Lemma
Let M and M^* be matchings in $G = (V, E)$. Consider $G' = (V, E')$, where $E' = M \oplus M^*$. Then, G' is a disjoint union of simple paths, simple cycles, and/or isolated vertices. Edges in each simple path or simple cycle alternate between M and M^*. If $|M^*| > |M|$, then G' contains $\geq |M^*| - |M|$ vertex-disjoint M-augmenting paths.

Proof ≤ 1 edge from M and ≤ 1 edge from M^* can be incident on each vertex.
$\Rightarrow \leq 2$ edges from E' are incident on each vertex.
$\Rightarrow$ Each connected component of G' is either a singleton vertex, an even-length simple cycle with edges alternately in M and M^*, or a path with edges alternately in M and M^*.
$E' = M \oplus M^* = (M \cup M^*) - (M \cap M^*)$ and $|M^*| > |M|$
$\Rightarrow E'$ contains $|M^*| - |M|$ more edges from M^* than from M.
Each cycle in G' has even number of edges, and they alternate between M and M^*.
$\Rightarrow$ The paths in G' account for the extra $|M^*| - |M|$ edges from M^*.
Each such path starts and ends with edges in M, with 1 more edge from M than M^*, or starts and ends with edges in M^*, with 1 more edge from M^* than M.
E' contains $|M^*| - |M|$ more edges from M^* than from M
$\Rightarrow \geq |M^*| - |M|$ paths with 1 more edge from M^* than M, and each is an

M-augmenting path.
$\Rightarrow \geq |M^*| - |M|$ M-augmenting paths.
Each vertex has ≤ 2 incident edges from E' $\Rightarrow$ these paths are vertex-disjoint. ■

Can design an algorithm to find a maximum matching by repeatedly finding augmenting paths to increase the size of the matching. Stop when there are no more augmenting paths:

Corollary
M is a maximum matching if and only if there are no M-augmenting paths.

Proof $\Leftarrow$: If there is an M-augmenting path P, then $M \oplus P$ is a matching with 1 more edge than $M \Rightarrow M$ is not a maximum matching.

$\Rightarrow$: Let M^* be a maximum matching and M be a matching that is not maximum. By the lemma, there are at least $|M^*| - |M| > 0$ vertex-disjoint M-augmenting paths. ■

Have enough for a maximum-matching algorithm already:

- Start with matching M empty.
- Repeatedly run either BFS or DFS from an unmatched vertex, taking alternating paths, until finding another unmatched vertex.
- Use the M-augmenting path to increment the size of M.
- Runs in $O(VE)$ time.

Hopcroft-Karp algorithm

Runs in $O(\sqrt{V}E)$ time.

Starts with an empty matching M. Repeatedly finds a maximal set of vertex-disjoint M-augmenting paths and updates M according to the corollary of the first lemma above. Stops when there are no M-augmenting paths, so that the matching is maximum according to the corollary of the second lemma above.

```
HOPCROFT-KARP(G)
  M = ∅
  repeat
      let 𝒫 = {P_1, P_2, ..., P_k} be a maximal set of vertex-disjoint
          shortest M-augmenting paths
      M = M ⊕ (P_1 ∪ P_2 ∪ ··· ∪ P_k)
  until 𝒫 == ∅
  return M
```

Need to show:

- How to find a maximal set of vertex-disjoint shortest M-augmenting paths in $O(E)$ time.
- That the **repeat** loop iterates $O(\sqrt{V})$ times.

How to find a maximal set of M-augmenting shortest paths

Three phases:

1. Using matching M, form a directed version G_M of the undirected bipartite graph G.
2. Create a dag H from G_M using a variant of BFS.
3. Find a maximal set of vertex-disjoint shortest M-augmenting paths by running a variant of DFS on H^{T}, the transpose of H. (H acyclic $\Rightarrow$ H^{T} acyclic.)

Creating G_M

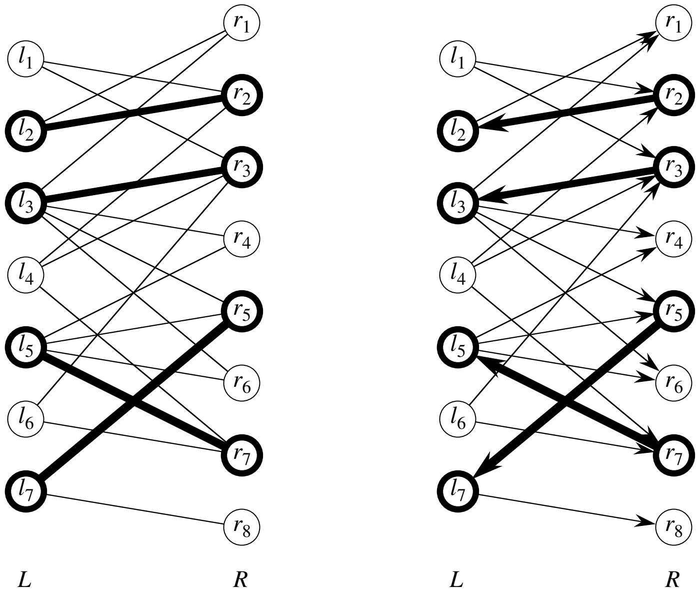
Think of an M-augmenting path P as starting at some unmatched vertex in L, traversing an odd number of edges, and ending at some unmatched vertex in R.

- Edges in P going $L \to R$ belong to $E - M$.
- Edges in P going $R \to L$ belong to M.

Create G_M by directing the edges as above: $G_M = (V, E_M)$,

$$\begin{aligned} E_M = {} & \{(l,r) : l \in L, r \in R, \text{ and } (l,r) \in E - M\} \quad \text{(edges from } L \text{ to } R) \\ & \cup \{(r,l) : r \in R, l \in L, \text{ and } (l,r) \in M\} \quad \text{(edges from } R \text{ to } L)\ . \end{aligned}$$

Example: The graph G and first matching M from the previous example on the left and G_M on the right.



Creating H

Dag $H = (V_H, E_H)$ has layers of vertices, according to minimum BFS distance from any unmatched vertex in L.

- Vertices in L are in even-numbered layers.
- Vertices in R are in odd-numbered layers.

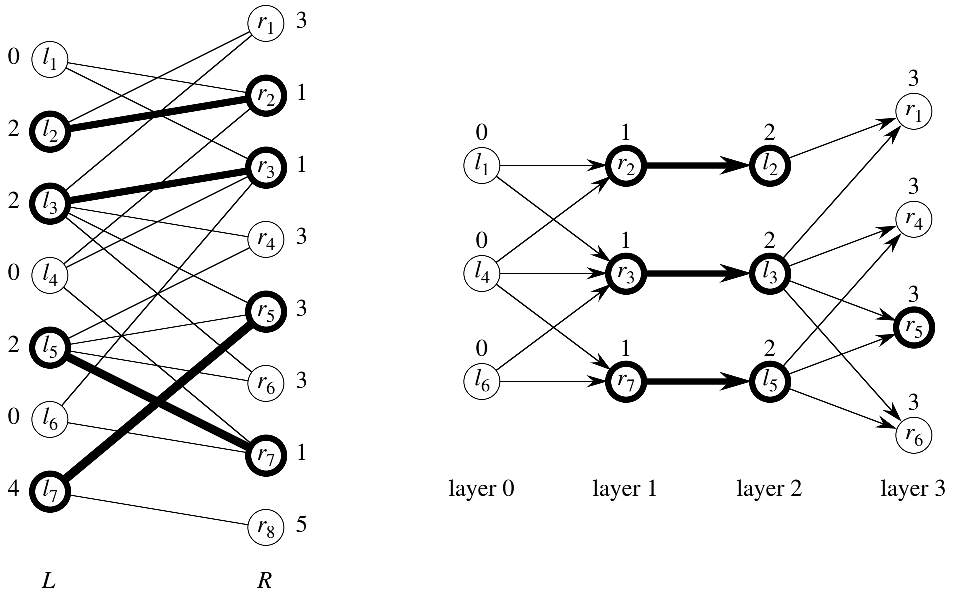
- If q = smallest distance in G_M from any unmatched vertex in L to an unmatched vertex in R, then the last layer is numbered q.
- Vertices with distance $> q$ do not appear in H.

$$E_H = \{(l,r) \in E_M : r.d \le q \text{ and } r.d = l.d + 1\} \cup \{(r,l) \in E_M : l.d \le q\} \ ,$$

where d means breadth-first distance in G_M from any unmatched vertex in L.

Omit edges that don't go between consecutive layers or go beyond layer q.

Example: G_M on left, H on right, $q = 3$. Numbers next to vertices are breadth-first distances. Because l_7 and r_8 have distances > 3, they are not in H. r_5 is in H, even though it is matched, because its distance is ≤ 3.

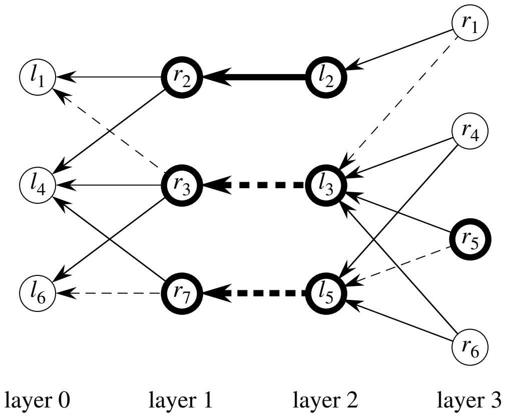


Run BFS on G_M to determine vertex distances, but starting from all unmatched vertices in L. In BFS code, replace the root vertex (s) by the set of unmatched vertices in L.

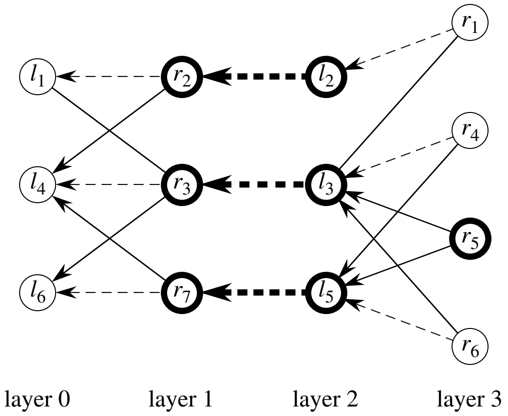
Every path in H from a vertex in layer 0 to an unmatched vertex in layer q corresponds to a shortest M-augmenting path in G. (Use the undirected versions of the directed edges in H.) Every shortest M-augmenting path in G is in H.

Finding a maximal set of vertex-disjoint shortest M-augmenting paths

- Create H^{T}.
- For each unmatched vertex r in layer q, perform depth-first search starting from r until either reaching a vertex in layer 0 or determining that no vertex in layer 0 is reachable. *[Paths found by DFS are indicated by dashed edges in the following figure.]*



- No need to keep discovery or finishing times. But need to keep track of which vertices have been discovered in any search because a vertex is searched from only when it is first discovered in any search.
- Keep track of predecessors to trace paths from layer 0 back to layer q.
- If a search from an unmatched vertex r in layer q does not reach a vertex in layer 0, then no M-augmenting path from r goes into the maximal set.
- The maximal set of vertex-disjoint shortest M-augmenting paths found might not be maximum. Here is a maximum set in the above example:



The Hopcroft-Karp algorithm needs only a maximal set, not necessarily a maximum set.

Analysis of the 3 phases

Need to show that all 3 phases take $O(E)$ time.

Assume that in G, each vertex has ≥ 1 incident edge $\Rightarrow |V| = O(E)$
$\Rightarrow |V| + |E| = O(E)$.

1. Creating G_M: Direct each edge of G. $O(E)$. $|V_M| = |V|$, $|E_M| = |E|$.
2. Creating H: Perform BFS on G_M. Can stop once the first distance in the BFS queue exceeds q. Dag H has $|V_H| \leq |V_M|$, $|E_H| \leq |E_M|$. $O(V_M + E_M) = O(E_M) = O(E)$.

3. Finding a maximal set of vertex-disjoint shortest M-augmenting paths: Third phase performs DFS from unmatched vertices in layer q. Vertex is not searched from after it is first discovered. $O(V_H + E_H) = O(E)$.
4. Updating the matching from the M-augmenting paths takes $O(E)$ time.

Therefore, each iteration of the **repeat** loop takes $O(E)$ time.

Bounding the iterations of the repeat loop

Start by showing that after each iteration of the **repeat** loop, the augmenting path length increases.

Lemma

Let

- M be a matching,
- q be the length of a shortest M-augmenting path,
- $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be a maximal set of vertex-disjoint shortest paths of length q,
- $M' = M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$, and
- P be a shortest M'-augmenting path.

Then, P has more than q edges.

Proof Consider separately cases where P is vertex-disjoint from the augmenting paths in $\mathcal{P}$ and where it is not.

Assume that P is vertex-disjoint from all augmenting paths in $\mathcal{P}$.
P contains edges in M but not in any of $P_1, \ldots, P_k$
$\Rightarrow$ P is also an M-augmenting path.
P disjoint from $P_1, \ldots, P_k$, P is an M-augmenting path, $\mathcal{P}$ is a maximal set of shortest M-augmenting paths
$\Rightarrow$ P is longer than any of the paths in $\mathcal{P}$, which have q edges
$\Rightarrow$ P has more than q edges.

Now assume that P visits at least 1 vertex from the M-augmenting paths in $\mathcal{P}$. By corollary of first lemma, M' is a matching with $|M'| = |M| + k$. By first lemma, $M' \oplus P$ is a matching with $|M' \oplus P| = |M'| + 1 = |M| + k + 1$.
Let $A = M \oplus M' \oplus P$.

Claim

$A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$.

Proof of claim

$$\begin{aligned}
A &= M \oplus M' \oplus P \\
&= M \oplus (M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)) \oplus P \\
&= (M \oplus M) \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && \text{(associativity of } \oplus) \\
&= \emptyset \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && (X \oplus X = \emptyset \text{ for all } X) \\
&= (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && (\emptyset \oplus X = X \text{ for all } X)\,.
\end{aligned}$$

■ (claim)

By second lemma above (let $M^* = M' \oplus P$), A contains $\geq |M' \oplus P| - |M| = k + 1$ vertex-disjoint M-augmenting paths.
Length of each M-augmenting path in A is $\geq q \Rightarrow |A| \geq (k+1)q = kq + q$.

Claim
P shares ≥ 1 edge with some M-augmenting path in $\mathcal{P}$.

Proof of claim
Subclaim: Under M', every vertex in each M-augmenting path in $\mathcal{P}$ is matched.
Proof of subclaim: Under M, only the first and last vertex in each path $P_i \in \mathcal{P}$ is unmatched, and they become matched under $M \oplus P_i$. Because the paths in $\mathcal{P}$ are vertex-disjoint, no other path $P_j \leq P_i$ in $\mathcal{P}$ changes the matched status of vertices in P_i. ■ (subclaim)

Suppose P shares a vertex v with some path $P_i \in \mathcal{P}$.
Endpoints of P unmatched under M' and all vertices of P_i are matched under M'.
$\Rightarrow v$ is not an endpoint of P.
$\Rightarrow v$ has an incident edge in P that belongs to M'.
Any vertex has ≤ 1 incident edge in a matching
$\Rightarrow$ this edge must also belong to P_i. ■ (claim)

$A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ and P shares ≥ 1 edge with some $P_i \in \mathcal{P}$
$\Rightarrow |A| < |P_1 \cup P_2 \cup \cdots \cup P_k| + |P|$.
Then,

$$\begin{aligned} kq + q &\leq |A| \\ &< |P_1 \cup P_2 \cup \cdots \cup P_k| + |P| \\ &= kq + |P| \ , \end{aligned}$$

so that $q < |P|$, meaning that P contains $> q$ edges. ■

Bound the size of a maximum matching based on the length of a shortest augmenting path.

Lemma
Let M be a matching, and let a shortest M-augmenting path contain q edges. Then the size of a maximum matching is $\leq |M| + |V|/(q+1)$.

Proof Let M^* be a maximum matching. By earlier lemma, there are $\geq |M^*| - |M|$ vertex-disjoint M-augmenting paths. Each contains $\geq q$ edges $\Rightarrow$ each contains $\geq q + 1$ vertices.
Paths are vertex-disjoint
$\Rightarrow (|M^*| - |M|)(q+1) \leq |V|$
$\Rightarrow |M^*| \leq |M| + |V|/(q+1)$. ■

Final lemma bounds the number of iterations of the **repeat** loop.

Lemma
The **repeat** loop iterates $O(\sqrt{V})$ times.

Proof By earlier lemma, the length q of the shortest M-augmenting paths found in each iteration increases from iteration to iteration. ⇒ After $\lceil \sqrt{|V|}\, \rceil$ iterations, must have $q \geq \lceil \sqrt{|V|}\, \rceil$.
Consider the situation after the first time that the matching is updated with augmenting paths whose length is $\geq \lceil \sqrt{|V|}\, \rceil$.
Size of a matching increases by ≥ 1 edge per iteration, previous lemma ⇒ number of additional iterations before reaching the maximum matching is at most

$$\begin{aligned} \frac{|V|}{\lceil \sqrt{|V|}\, \rceil + 1} &< \frac{|V|}{\sqrt{|V|}} \\ &= \sqrt{|V|}\,. \end{aligned}$$

So that the total number of iterations is $\leq 2\sqrt{|V|}$. ■

The final bound:

Theorem
HOPCROFT-KARP runs in $O(\sqrt{V}E)$ time.

Proof The **repeat** loop makes $O(\sqrt{V})$ iterations, each taking $O(E)$ time. ■

Stable-marriage problem

Input is a ***complete bipartite graph***: contains an edge from every vertex in L to every vertex in R. Assume that $|L| = |R| = n$.

Each vertex in L ranks all vertices in R, and vice-versa.

Goal: Match each vertex in L with a vertex in R in a way that avoids a ***blocking pair***: a vertex in L and a vertex in R that are not matched with each other but each prefers the other to their matched vertex. Want a ***stable matching***: no blocking pair. Otherwise, the matching is ***unstable***.

Because $|L| = |R|$, can match every vertex.

Example:
Women: Ann, Bea, Carol, Deb
Men: Ed, Fred, Greg, Henry

Preferences:

Ann: Greg, Henry, Ed, Fred
Bea: Fred, Henry, Ed, Greg
Carol: Greg, Fred, Henry, Ed
Deb: Greg, Henry, Fred, Ed

Ed: Ann, Deb, Carol, Bea
Fred: Ann, Carol, Deb, Bea
Greg: Carol, Deb, Ann, Bea
Henry: Carol, Ann, Bea, Deb

A stable matching:

Carol and Greg
Ann and Henry
Deb and Fred
Bea and Ed

[This example is just a renaming of the example in the textbook. Leave the preference lists on board—will need them for example later.]

No blocking pair in this matching.
If the last 2 pairs were

Bea and Fred
Deb and Ed

then Deb and Fred would be a blocking pair, because they were not paired together, Deb prefers Fred to Ed, and Fred prefers Deb to Bea.

Stable matchings might not be unique.

Example:
Women: Inez, Jill, Kate
Men: Leo, Mike, Neil

Preferences:

Inez: Leo, Mike, Neil
Jill: Mike, Neil, Leo
Kate: Neil, Leo, Mike

Leo: Jill, Kate, Inez
Mike: Kate, Inez, Jill
Neil: Inez, Jill, Kate

3 stable matchings:

Matching 1	Matching 2	Matching 3
Inez and Leo	Jill and Leo	Kate and Leo
Jill and Mike	Kate and Mike	Inez and Mike
Kate and Neil	Inez and Neil	Jill and Neil

[Also a renaming of the example in the textbook.]

- Matching 1: All women get first choice, all men get last choice.
- Matching 2: All men get first choice, all women get last choice.
- Matching 3: Everyone gets second choice.

Gale-Shapley algorithm

Can always devise a stable matching, regardless of the rankings. Gale-Shapley algorithm always produces a stable matching.

Two variants: “woman-oriented” and “man-oriented.” They mirror each other.

Woman-oriented version:

- Each woman or man is either “free” or “engaged.”
- Everyone starts out free.

- Engagement occurs when a woman proposes to a man.
- When a man is first proposed to, goes from free to engaged. Always stays engaged, but possibly to someone else later on.
- If an engaged man receives a proposal from a woman he prefers to the woman he's currently engaged to, he breaks the engagement, the woman he was engaged to becomes free, and the man and woman he prefers become engaged.
- Each woman proposes to the men in her preference list, in order, until the last time she becomes engaged.
- When a woman is engaged, temporarily stops proposing, but resumes if she becomes free, continuing down her list.
- Procedure stops once everyone is engaged.
- Allows for choice: next proposal may come from any free woman.

Man-oriented version swaps the roles.

```
GALE-SHAPLEY(men, women, rankings)
  assign each woman and man as free
  while some woman $w$ is free
      let $m$ be the first man on $w$'s ranked list to whom she has not proposed
      if $m$ is free
          $w$ and $m$ become engaged to each other (and not free)
      elseif $m$ ranks $w$ higher than the woman $w'$ he is currently engaged to
          $m$ breaks the engagement to $w'$, who becomes free
          $w$ and $m$ become engaged to each other (and not free)
      else $m$ rejects $w$, with $w$ remaining free
  return the stable matching comprising the engaged pairs
```

Example: With 4 women and 4 men from before:

1. Ann proposes to Greg. Greg is free. Ann and Greg become engaged.
2. Bea proposes to Fred. Fred is free. Bea and Fred become engaged.
3. Carol proposes to Greg. Greg is engaged to Ann, but prefers Carol. Greg breaks the engagement to Ann, who becomes free. Carol and Greg become engaged.
4. Deb proposes to Greg. Greg is engaged to Carol, whom he prefers to Deb. Greg rejects Deb, who remains free.
5. Deb proposes to Henry. Henry is free. Deb and Henry become engaged.
6. Ann proposes to Henry. Henry is engaged to Deb, but prefers Ann. Henry breaks the engagement with Deb, who becomes free. Ann and Henry become engaged.
7. Deb proposes to Fred. Fred is engaged to Bea, but prefers Deb. Fred breaks the engagement to Bea, who becomes free. Deb and Fred become engaged.
8. Bea proposes to Henry. Henry is engaged to Ann, whom he prefers to Bea. Henry rejects Bea, who remains free.
9. Bea proposes to Ed. Ed is free. Bea and Ed become engaged.

At this point, everyone is engaged and nobody is free, so that the **while** loop terminates. Get the stable matching from before.

Proof of Gale-Shapley algorithm

Theorem
GALE-SHAPLEY always terminates with a stable matching.

Note: Since GALE-SHAPLEY always terminates with a stable matching, a stable matching is always possible.

Proof First show that the **while** loop always terminates. Show by contradiction. Suppose that the loop fails to terminate.

Loop fails to terminate.
⇒ Some woman remains free.
⇒ That woman proposed to all the men and was rejected by each one.
For a man to reject, he must already be engaged.
⇒ All men are engaged.
Once a man is engaged, he stays engaged.
Equal number of men and women.
⇒ All women are engaged.
⇒ **while** loop terminates.

Show that the **while** loop makes a bounded number of iterations:
Each of n women proposes to $\leq n$ men.
⇒ $\leq n^2$ iterations.

Show no blocking pairs:
Suppose woman w is matched with man m, but she prefers man m'.
Will show that m' does not prefer w to his partner, so that w and m' is not a blocking pair.
w ranks m' higher than m.
⇒ w proposes to m' before m.
⇒ m' either rejected at the time or accepted and broke the engagement later.
If m' rejected, m' was engaged to another woman w' whom he preferred to w.
If m' accepted, m' was engaged to w but ended up with a partner he prefers to w.
Either way, m' prefers his partner to w.
⇒ w and m' is not a blocking pair. ■

Can implement GALE-SHAPLEY to run in $O(n^2)$ time (Exercise 25.2-1).

Since the **while** loop can choose any free woman as the next proposer, can different choices produce different stable matchings? No.

Theorem
For a given input, GALE-SHAPLEY always returns the same stable matching, regardless of which free woman it chooses. In this stable matching, each woman has the best partner possible.

Proof By contradiction. Suppose GALE-SHAPLEY returns stable matching M, but that there is another stable matching M'. ⇒ Some woman w is matched with man m in M, but is matched with some other man m' she prefers in M'.
⇒ w proposed to m' before proposing to m.
m' either rejected w or accepted and later broke the engagement.

Either way, at some moment, m' decided on some other woman w' over w.
Without loss of generality, let this moment be the first time this happens (any man rejects a partner who belongs in some stable matching).

Claim: w' cannot have a partner in a stable matching that she prefers to m'.
Proof of claim: Suppose w' has a partner m'' in a stable matching whom she prefers to m'. Then w' would have proposed to m'' and been rejected at some point before proposing to m'.
Since the moment that m' rejected w was the first rejection in some stable matching and w' proposed to m'' before proposing to m', m'' could not have rejected w' before m' rejected w. This is a contradiction that proves the claim.

By the claim, w' prefers m' to her partner in M'.
w' prefers m' to her partner in M' and m' prefers w' to his partner w in M'.
$\Rightarrow$ w' and m' is a blocking pair in M'.
$\Rightarrow$ M' is not a stable matching.
Contradicts the assumption that GALE-SHAPLEY returns some other stable matching.
There was no condition on the order in which free women proposed.
$\Rightarrow$ All possible orders result in the same matching. ■

Corollary

There can be matchings not found by GALE-SHAPLEY.

Proof Earlier example with 3 women and 3 men had 3 stable matchings. GALE-SHAPLEY would return only 1 of them. ■

The woman-oriented version produces the best partner for each woman, but the worst partner for each man:

Corollary

In the stable matching found by woman-oriented GALE-SHAPLEY, each man has the worst possible partner in any stable matching.

Proof Let GALE-SHAPLEY find stable matching M. Suppose there is another stable matching M' and a man m who prefers his partner w in M to his partner w' in M'. Let the partner of w in M' be m'.
By the theorem, m is the best partner w can have in any stable matching.
$\Rightarrow$ w prefers m to m'.
In M', w prefers m to m' and m prefers w to w'.
$\Rightarrow$ w and m is a blocking pair in M'.
$\Rightarrow$ M' is not a stable matching. ■

Assignment problem

Input: A weighted, complete bipartite graph $G = (V, E)$, with $V = L \cup R$ and $|L| = |R| = n$. For $l \in L$, $r \in R$, weight of edge (l, r) is $w(l, r)$.

Goal: Find a perfect matching M^* that maximizes the total weight.

For matching M, $w(m) = \sum_{(l,r) \in M} w(l,r)$.
Want M^* such that $w(M^*) = \max\{w(M) : M \text{ is a perfect matching}\}$.

Solve with the ***Hungarian algorithm***. Will show an $O(n^4)$-time solution, but Problem 25-2 shows how to reduce to $O(n^3)$.

Equality subgraph

The Hungarian algorithm works with a subgraph of G, the ***equality subgraph***. It changes over time.

Important property of the equality subgraph: Any perfect matching in the equality subgraph is an optimal solution to the assignment problem.

Relies on an attribute h, the ***label***, of each vertex.
h is a ***feasible vertex labeling*** of G if

$$l.h + r.h \geq w(l,r) \text{ for all } l \in L \text{ and } r \in R .$$

Can always find a feasible vertex labeling. *Example:* ***default vertex labeling***:

$$\begin{aligned} l.h &= \max\{w(l,r) : r \in R\} && \text{for all } l \in L , \\ r.h &= 0 && \text{for all } r \in R . \end{aligned}$$

The equality subgraph for a vertex labeling h is $G_h = (V, E_h)$, where

$$E_h = \{(l,r) \in E : l.h + r.h = w(l,r)\} .$$

($=$ instead of $\geq$)

Relationship between a perfect matching in an equality subgraph and an optimal solution to the assignment problem:

Theorem
If h is a feasible labeling of G and G_h contains a perfect matching M^*, then M^* is an optimal solution to the assignment problem for G.

Proof G_h and G have the same vertices $\Rightarrow$ M^* is also a perfect matching for G.
Each vertex has exactly 1 incident edge from any perfect matching $\Rightarrow$

$$\begin{aligned} w(M^*) &= \sum_{(l,r) \in M^*} w(l,r) \\ &= \sum_{(l,r) \in M^*} (l.h + r.h) \quad \text{(all edges in } M^* \text{ belong to } G_h) \\ &= \sum_{l \in L} l.h + \sum_{r \in R} r.h \quad (M^* \text{ is a perfect matching}) . \end{aligned}$$

Let M be any perfect matching in G.

$$\begin{aligned} w(M) &= \sum_{(l,r) \in M} w(l,r) \\ &\leq \sum_{(l,r) \in M} (l.h + r.h) \quad (h \text{ is a feasible vertex labeling}) \\ &= \sum_{l \in L} l.h + \sum_{r \in R} r.h \quad (M \text{ is a perfect matching}) . \end{aligned}$$

Then,

$$w(M) \leq \sum_{l \in L} l.h + \sum_{r \in R} r.h = w(M^*)$$

$\Rightarrow M^*$ is a maximum-weight perfect matching in G. ■

Repeat important point: Just need to find *some* perfect matching in *some* equality subgraph.

The Hungarian algorithm

[Named after work by Hungarian mathematicians D. Kőnig and J. Egervéry.]

Starts with any matching M and any feasible vertex labeling h. Repeatedly finds an M-augmenting path P in G_h and makes a new matching $M \oplus P$, updating M.

As long as some equality subgraph contains an M-augmenting path, size of the matching increases, until getting a perfect matching in G_h.

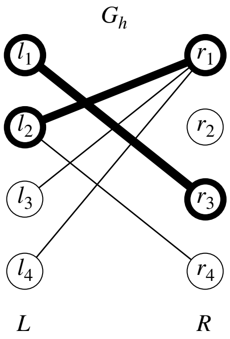
Questions to answer:

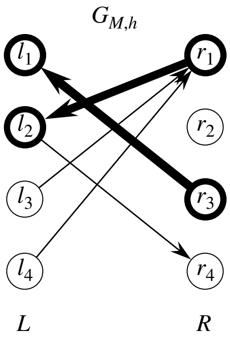
1. Which initial matching in G_h?
   Answer: Any matching. Even an empty matching. A greedy maximal matching works well. *[Will see greedy maximal matching later.]*
2. Which initial feasible vertex labeling h?
   Answer: The default vertex labeling given before.
3. If G_h has an M-augmenting path, how to find it?
   Answer: Use a variant of BFS. *[Will elaborate on this answer later.]*
4. What if G_h has no M-augmenting path?
   Answer: Update h to bring ≥ 1 new edge into G_h. *[Will elaborate on this answer later as well.]*

Answer these questions via a running example. *[This is a smaller example than the one in the textbook, but it shows the cases that can arise.]*

In the example below, $n = 4$. The matrix on the left shows edge weights, with the initial feasible vertex labeling given by the default vertex labeling above and to the left of the matrix. Underlined entries have $l_i.h + r_j.h = w(l_i, r_j)$, corresponding to edges (l_i, r_j) in E_h. G_h is in the middle, with an initial greedy maximal matching shown with heavy edges and matched vertices with heavy outlines. *[$G_{M,h}$ on the right will be discussed later.]*

$$\begin{array}{cc|cccc|}
 & & r_1 & r_2 & r_3 & r_4 \\
 & h & 0 & 0 & 0 & 0 \\ \hline
l_1 & 12 & 8 & 5 & \underline{12} & 9 \\
l_2 & 9 & \underline{9} & 6 & 7 & \underline{9} \\
l_3 & 9 & \underline{9} & 8 & 8 & 5 \\
l_4 & 6 & \underline{6} & 2 & 3 & 5 \\ \hline
\end{array}$$





Greedy maximal bipartite matching

Use this simple procedure:

```
GREEDY-BIPARTITE-MATCHING($G$)
  $M = \emptyset$
  for each vertex $l \in L$
      if $l$ has an unmatched neighbor in $R$
          choose any such unmatched neighbor $r \in R$
          $M = M \cup \{(l, r)\}$
  return $M$
```

Can show that it returns a matching at least $1/2$ the size of a maximum matching (Exercise 25.3-2).

How to find an M-augmenting path in G_h

Create the ***directed equality subgraph*** $G_{M,h}$.

As in Hopcroft-Karp, think of an M-augmenting path as starting from an umatched vertex in L, ending at an unmatched vertex in R, taking unmatched edges $L \to R$, and matched edges $R \to L$. Direct edges accordingly in $G_{M,h} = (V, E_{M,h})$, where

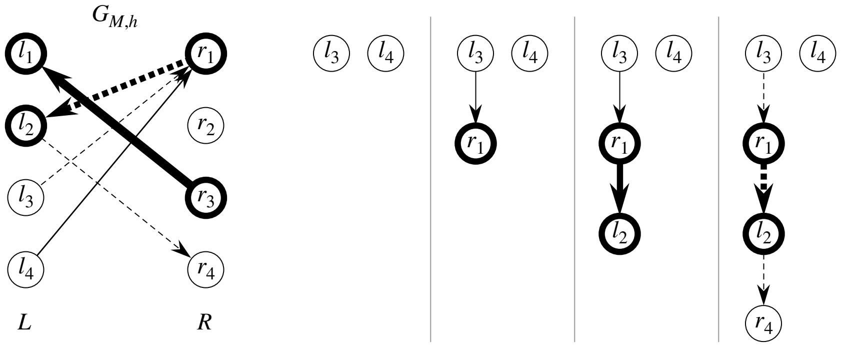
$$\begin{aligned} E_{M,h} = \; & \{(l, r) : l \in L, r \in R, \text{ and } (l, r) \in E_h - M\} \quad \text{(edges from } L \text{ to } R\text{)} \\ & \cup \{(r, l) : r \in R, l \in L, \text{ and } (l, r) \in M\} \qquad \text{(edges from } R \text{ to } L\text{)} . \end{aligned}$$

[Now show $G_{M,h}$ in the above figure.]

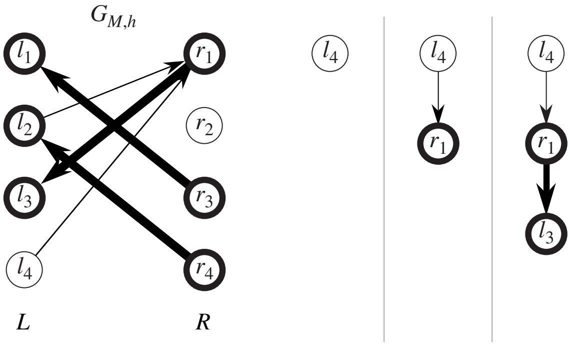
Hungarian algorithm searches for a directed path in $G_{M,h}$ from any unmatched vertex in L to any unmatched vertex in R. Can use any graph-searching method.

Choosing breadth-first search, but instead of just 1 root, use all vertices in L as roots. Do so by initializing the queue to contain all unmatched vertices in L. Stop as soon as the search finds an unmatched vertex in R. Don't need to keep track of distances, but need to keep track of predecessors.

In the figure below, the right side shows the evolution of the BFS in G_h, with the queue initially containing the unmatched vertices l_3 and l_4. BFS finds the M-augmenting path with edges $\langle (l_3, r_1), (r_1, l_2), (l_2, r_4) \rangle$, shown with dashed lines.



In the figure below, the matching M in $G_{M,h}$ is updated by taking the symmetric difference with the M-augmenting path found.



The right side shows the evolution of another BFS, now with the queue initially containing l_4, the only unmatched vertex in L.

This BFS ends before it finds an unmatched vertex in R, so that it fails to find an M-augmenting path in $G_{M,h}$.

When the search for an M-augmenting path fails

When the search for an M-augmenting path fails, the most recently discovered vertices must be in L. If the most recently discovered vertex is an unmatched vertex in R, then the search succeeded. When the search visits a matched vertex in R, it can take an edge in M to a vertex in L.

Have been constructing a breadth-first forest $F = (V_F, E_F)$, where each unmatched vertex in L is a root in F.

It's OK to change the equality subgraph on the fly, but don't undo any of the work already done. Update the feasible vertex labeling h to create a new equality subgraph, subject to

1. Every edge in F remains in the equality subgraph.
2. Every edge in M remains in the equality subgraph.
3. ≥ 1 edge (l, r) goes into E_h, where $l \in L \cap V_F$ and $r \in R - V_F$ ((l, r) extends F to at least one previously unreachable vertex in R).
4. Any edge that leaves the equality subgraph belongs to neither M nor F.

Enqueue newly discovered vertices in R. They might not be neighbors in the new $G_{M,h}$ of the most recently discovered vertices in L. *[The example graph in these notes does not show this possibility, but Figure 25.7 does, with edge (l_5, r_3) added to F. l_5 is a root and is discovered before l_4 and l_3, which are also in F.]*

How to update the feasible vertex labeling:

Let $F_L = L \cap V_F$, $F_R = R \cap V_F$ be the vertices in F from L and R.
Compute the smallest difference δ that an edge incident on a vertex in F_L missed being in the current G_h:

$$\delta = \min\{l.h + r.h - w(l, r) : l \in F_L \text{ and } r \in R - F_R\} .$$

Create a new feasible vertex labeling h' by

- subtracting δ from $l.h$ for all $l \in F_L$ and
- adding δ to $r.h$ for all $r \in F_R$:

$$v.h' = \begin{cases} v.h - \delta & \text{if } v \in F_L \text{ ,} \\ v.h + \delta & \text{if } v \in F_R \text{ ,} \\ v.h & \text{otherwise } (v \in V - V_F) \text{ .} \end{cases}$$

The following lemma shows that these changes adhere to the criteria above.

Lemma
h' has the following properties:

1. If $(u, v) \in E_F$ for $G_{M,h}$, then $(u, v) \in E_{M,h'}$ (every edge in F remains in the equality subgraph).
2. If $(l, r) \in M$ for G_h, then $(r, l) \in E_{M,h'}$ (every edge in M remains in the equality subgraph).
3. There are vertices $l \in F_L$, $r \in R - F_R$ such that $(l, r) \notin E_{M,h}$ but $(l, r) \in E_{M,h'}$ ((l, r) extends F to at least one previously unreachable vertex in R).

Proof First, show that h' is a feasible vertex labeling.
h is a feasible vertex labeling $\Rightarrow l.h + r.h \geq w(l, r)$ for all $l \in L$ and $r \in R$.
Need $l.h' + r.h' \geq w(l, r)$ for all $l \in L$ and $r \in R$.
$\Rightarrow$ Suffices to have $l.h' + r.h' \geq l.h + r.h$.
The only way that could have $l.h' + r.h' < l.h + r.h$ is if $l \in F_L$ and $r \in R - F_R$.
Amount of decrease equals δ.
$\Rightarrow l.h' + r.h' = l.h - \delta + r.h$.
But the equation for δ means that $l.h - \delta + r.h \geq w(l, r)$ for any $l \in F_L$ and $r \in R - F_R$.
$\Rightarrow l.h' + r.h' \geq w(l, r)$.
For all other edges, $l.h' + r.h' \geq l.h + r.h \geq w(l, r)$.
Therefore, h' is a feasible vertex labeling.

Now show that each of the three properties holds.

1. $l \in F_L$ and $r \in F_R \Rightarrow$
   $l.h' + r.h' = l.h + r.h$ because δ is added to label of l and subtracted from label of r
   $\Rightarrow$ If $(u, v) \in E_F$ for $G_{M,h}$, then $(u, v) \in E_{M,h'}$.
2. Whenever a matched vertex $r \in R$ is removed from the queue, the vertex $l \in L$ it's matched with is discovered.
   $\Rightarrow$ The only unmatched vertices in F_L are the roots.
   $\Rightarrow$ When h' is computed, if $(r, l) \in M$, then either both r and l are in V_F or neither of them are in V_F.
   Already saw that $l \in F_L$ and $r \in F_R \Rightarrow l.h' + r.h' = l.h + r.h$.

   If $l \in L - F_L$ and $r \in R - F_R$, then $l.h' = l.h$ and $r.h' = r.h$.
   $\Rightarrow l.h' + r.h' = l.h + r.h$.
   $\Rightarrow$ If $(l, r) \in M$ for G_h, then $(r, l) \in E_{M,h'}$.
3. Let $(l, r) \notin E_h$ such that $l \in F_L$, $r \in R - F_R$, and $\delta = l.h + r.h - w(l, r)$. (By defintion of δ, ≥ 1 such an edge exists.)

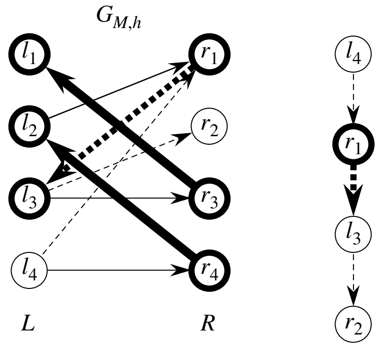
$$\begin{aligned} l.h' + r.h' &= l.h - \delta + r.h \\ &= l.h - (l.h + r.h - w(l, r)) + r.h \\ &= w(l, r) \text{ .} \end{aligned}$$

$\Rightarrow (l, r) \in E_{h'}$.
$(l, r) \notin E_h \Rightarrow (l, r) \notin M \Rightarrow (l, r)$ is directed $L \to R$ in $E_{M,h'}$. ■

The figure below continues the example. Here, $\delta = 1$, achieved at edges (l_3, r_2), (l_3, r_3), and (l_4, r_4), which enter E_h. Changes are circled in the matrix and h values.

(l_2, r_1) goes out of G_h. That is OK, because it was in neither the breadth-first forest F or the matching M.

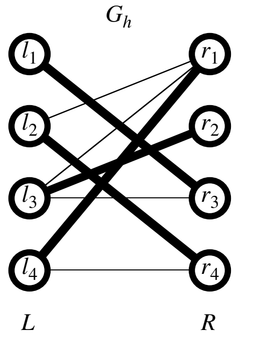
$$\begin{array}{cc|cccc|}
 & & r_1 & r_2 & r_3 & r_4 \\
 & h & \enclose{circle}{1} & 0 & 0 & 0 \\
\hline
l_1 & 12 & 8 & 5 & \underline{12} & 9 \\
l_2 & 9 & \enclose{circle}{9} & 6 & 7 & \underline{9} \\
l_3 & \enclose{circle}{8} & \underline{9} & \underline{\enclose{circle}{8}} & \underline{\enclose{circle}{8}} & 5 \\
l_4 & \enclose{circle}{5} & \underline{6} & 2 & 3 & \underline{\enclose{circle}{5}} \\
\hline
\end{array}$$



The search continues from l_3, finding the unmatched vertex r_2. That gives the M-augmenting path $\langle (l_4, r_1), (r_1, l_3), (l_3, r_2) \rangle$.

The next figure shows the final matching after updating M with this M-augmenting path. Matrix entries with a box around them are in the final matching.

$$\begin{array}{cc|cccc|}
 & & r_1 & r_2 & r_3 & r_4 \\
 & h & 1 & 0 & 0 & 0 \\
\hline
l_1 & 12 & 8 & 5 & \boxed{\underline{12}} & 9 \\
l_2 & 9 & 9 & 6 & 7 & \boxed{\underline{9}} \\
l_3 & 8 & \underline{9} & \boxed{\underline{8}} & \underline{8} & 5 \\
l_4 & 5 & \boxed{\underline{6}} & 2 & 3 & \underline{5} \\
\hline
\end{array}$$



The optimal assignment uses the matching (l_1, r_3), (l_2, r_4), (l_3, r_2), (l_4, r_1), with total weight $12 + 9 + 8 + 6 = 35$.

Pseudocode for the Hungarian algorithm

Property 3 of the last lemma ensures that when FIND-AUGMENTING-PATHS calls DEQUEUE, the queue is nonempty.

Uses attribute π for predecessors in the breadth-first forest.
Instead of coloring vertices, uses sets F_L and F_R for discovered vertices.

```
HUNGARIAN($G$)
  for each vertex $l \in L$
      $l.h = \max\{w(l, r) : r \in R\}$
  for each vertex $r \in R$
      $r.h = 0$
  let $M$ be any matching in $G_h$ (such as the matching returned by
      GREEDY-BIPARTITE-MATCHING)
  from $G$, $M$, and $h$, form the equality subgraph $G_h$
      and the directed equality subgraph $G_{M,h}$
  while $M$ is not a perfect matching in $G_h$
      $P$ = FIND-AUGMENTING-PATH($G_{M,h}$)
      $M = M \oplus P$
      update the equality subgraph $G_h$
          and the directed equality subgraph $G_{M,h}$
  return $M$
```

```
FIND-AUGMENTING-PATH($G_{M,h}$)
  $Q = \emptyset$
  $F_L = \emptyset$
  $F_R = \emptyset$
  for each unmatched vertex $l \in L$
      $l.\pi = \text{NIL}$
      ENQUEUE($Q, l$)
      $F_L = F_L \cup \{l\}$          // forest $F$ starts with unmatched vertices in $L$
  repeat
      if $Q$ is empty              // ran out of vertices to search from?
          $\delta = \min\{l.h + r.h - w(l,r) : l \in F_L \text{ and } r \in R - F_R\}$
          for each vertex $l \in F_L$
              $l.h = l.h - \delta$
          for each vertex $r \in F_R$
              $r.h = r.h + \delta$
          from $G$, $M$, and $h$, form a new directed equality graph $G_{M,h}$
          for each new edge $(l,r)$ in $G_{M,h}$     // continue search with new edges
              if $r \notin F_R$
                  $r.\pi = l$                  // discover $r$, add it to $F$
                  if $r$ is unmatched
                      an $M$-augmenting path has been found
                          (exit the repeat loop)
                  else ENQUEUE($Q, r$)        // can search from $r$ later
                      $F_R = F_R \cup \{r\}$
      $u$ = DEQUEUE($Q$)                    // search from $u$
      for each neighbor $v$ of $u$ in $G_{M,h}$
          if $v \in L$
              $v.\pi = u$
              $F_L = F_L \cup \{v\}$              // discover $v$, add it to $F$
              ENQUEUE($Q, v$)                // can search from $v$ later
          elseif $v \notin F_R$                   // $v \in R$
              $v.\pi = u$
              if $v$ is unmatched
                  an $M$-augmenting path has been found
                      (exit the repeat loop)
              else ENQUEUE($Q, v$)
                  $F_R = F_R \cup \{v\}$
  until an $M$-augmenting path has been found
  using the predecessor attributes $\pi$, construct an $M$-augmenting path $P$
      by tracing back from the unmatched vertex in $R$
  return $P$
```

Analysis of the Hungarian algorithm

In HUNGARIAN, initialization (the part before the **while** loop) takes $O(n^2)$ time. The **while** loop iterates at most n times, since each iteration increases the size of the matching by 1. Each update of M takes $O(n)$ time. Updating G_h and $G_{M,h}$ takes $O(n^2)$ time.

Claim that each call of FIND-AUGMENTING-PATH takes $O(n^3)$ time. Call the lines that execute when Q is empty a ***growth step***. Ignoring the growth steps, FIND-AUGMENTING-PATH is a BFS, which can be implemented in $O(n^2)$ time (have $O(n^2)$ edges and need to represent F_L and F_R). Within a single call of FIND-AUGMENTING-PATH, at most n growth steps occur, since each one discovers ≥ 1 vertex in R. Each edge becomes new in $G_{M,h}$ at most once, so that the "**for** each new edge (l, r) in $G_{M,h}$" loop iterates $\leq n^2$ times per call of FIND-AUGMENTING-PATH. Computing δ and $G_{M,h}$ takes $O(n^2)$ time. Total is $O(n^3)$ time per call of FIND-AUGMENTING-PATH.

Can reduce the time of FIND-AUGMENTING-PATH to $O(n^2)$ (Exercise 25.3-5 and Problem 25-2).

Solutions for Chapter 25: Matchings in Bipartite Graphs

Solution to Exercise 25.1-2

M-augmenting paths and augmenting paths in flow networks are similar in that both have specific criteria for their first and last vertices (for an M-augmenting path, the first vertex is in L and the last vertex is in R; for a flow network, the first vertex is the source and the last vertex is the sink) and both provide a means to increase the value of the solution (increasing the number of edges in the matching by 1; increasing the flow value by the residual capacity of the augmenting path).

M-augmenting paths and augmenting paths in flow networks differ in that M-augmenting paths do not necessarily have one specific first vertex and one specific last vertex, there is no notion of edge capacity in M-augmenting paths, and M-augmenting paths must alternate between unmatched and matched edges.

Solution to Exercise 25.1-3

A search in H from layer 0 to layer q could terminate at a matched vertex in layer q, which would fail to yield an M-augmenting path. By searching in H^{T} from unmatched vertices layer q to layer 0, every search that ends in layer 0 yields an M-augmenting path.

Solution to Exercise 25.1-4

Because an M-augmenting path must have an odd length, the length q of the shortest M-augmenting paths found in line 3 increase by at least 2 in each iteration. We get $q \geq \left\lceil \sqrt{|V|} \right\rceil$ after just $\left\lceil \sqrt{|V|}/2 \right\rceil$ iterations.

Solution to Exercise 25.1-5

$\Rightarrow$: Let G contain a perfect matching M, and let A be any subset of L. Since each vertex in A has an edge in M incident on it, as well as possibly other incident edges not in M, we have $|A| \leq |N(A)|$.

$\Leftarrow$: We show by contradiction that if $|A| \leq |N(A)|$ for every subset $A \subseteq L$, then G contains a perfect matching. Suppose that $|A| \leq |N(A)|$ for every subset $A \subseteq L$, but G contains no perfect matching. Let M^* be a maximum matching in G, so that $|M^*| < |L|$. Let $v \in L$ be unmatched under M^*, and let S be the set of all vertices connected to v by M^*-alternating paths. Since M^* is a maximum matching, Corollary 25.4 implies that G contains no M^*-augmenting path, which means that v is the only vertex in S that is unmatched under M^*. Let $S_L = S \cap L$ and $S_R = S \cap R$.

Since S is defined by M^*-alternating paths, each vertex in $S_L - \{v\}$ is matched in M^* with a vertex in S_R, so that $|S_R| = |S_L| - 1$. Because every vertex in S_L is adjacent to a vertex in S_R, we have $S_R \subseteq N(S_L)$. In fact, we have $S_R = N(S_L)$, since every vertex in $N(S_L)$ is connected to v by an M^*-alternating path. But now we have $|N(S_L)| = |S_R| = |S_L| - 1 < |S|$, which contradicts the assumption that $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

Solution to Exercise 25.1-6

Let G be a d-regular bipartite graph, where $V = L \cup R$. Choose any subset $A \subseteq L$, let E_A be the edges incident on vertices in A, and let $E_{N(A)}$ be the edges incident on vertices in $N(A)$. We have $|E_A| = d\,|A|$ and $|E_{N(A)}| = d\,|N(A)|$. Since every edge in E_A is incident on a vertex in $N(A)$, we have $E_A \subseteq E_{N(A)}$, so that $|E_A| \leq |E_{N(A)}|$. Thus, we have $d\,|A| = |E_A| \leq |E_{N(A)}| = d\,|N(A)|$, so that $|A| \leq |N(A)|$. By Hall's theorem, therefore, G contains a perfect matching.

To show that G contains d disjoint perfect matchings, take one of the perfect matchings and remove it. The result is a $(d-1)$-regular bipartite graph. It, too, contains a perfect matching. Keep going. There are d such perfect matchings altogether, and they are disjoint.

Solution to Exercise 25.2-2

Yes, it is possible to have an unstable matching with two men and two women. Suppose that the women are Mary and Nancy and that the men are Otto and Paul, with the following preferences:

Mary:	Otto, Paul
Nancy:	Paul, Otto
Otto:	Mary, Nancy
Paul:	Nancy, Mary

Then the following matching is unstable:

Mary and Paul
Nancy and Otto

Both pairs are blocking: Mary prefers Otto to Paul, Paul prefers Nancy to Mary, Nancy prefers Paul to Otto, and Otto prefers Mary to Nancy.

Solution to Exercise 25.2-3

To equalize the number of students and hospitals, add dummy students or hospitals as needed. Students propose to hospitals. The algorithm changes in that each hospital h is "engaged" to its favorite r_h students that have proposed.

Solution to Exercise 25.2-4

Let m be the last man to become engaged when the GALE-SHAPLEY procedure executes, and let the proposer at that time be woman w. No woman is rejected by m, since the **while** loop terminates once the last man receives his first proposal. By Theorem 25.11, m is the best partner that w can have in any stable matching, so that no woman matched to m can be strictly better off in any other stable matching.

Solution to Exercise 25.2-5

Wendy: Xenia, Yolanda, Zelda
Xenia: Yolanda, Wendy, Zelda
Yolanda: Wendy, Xenia, Zelda
Zelda: Wendy, Xenia, Yolanda

- Suppose that (Wendy, Xenia), (Yolanda, Zelda) is the matching. Then (Xenia, Yolanda) is unstable.
- Suppose that (Wendy, Yolanda), (Xenia, Zelda) is the matching. Then (Wendy, Xenia) is unstable.
- Suppose that (Wendy, Zelda), (Xenia, Yolanda) is the matching. Then (Wendy, Yolanda) is unstable.

Solution to Exercise 25.3-1

Instead of checking when a vertex is discovered, check when a vertex is removed from the queue. The downside of doing so is that the search could continue even after discovering an unmatched vertex in R.

Solution to Exercise 25.3-2

We'll show something much stronger: that in a general undirected graph (i.e., not necessarily bipartite), any maximal matching (not necessarily the greedy matching) is at least half the size of a maximum matching.

Let M be a maximal matching and M^* be a maximum matching. Every edge $e \in M^*$ must have at least one of its endpoints matched in M, for otherwise we could add e into M, which means that M was not maximal. Since every edge in M matches two vertices, the number of vertices matched in M is at least as large as the number of edges in M^*, or $2|M| \geq |M^*|$. Dividing both sides by 2 gives $|M| \geq |M^*|/2$.

Solution to Exercise 25.3-3

For an edge (l, r) to leave the directed equality subgraph, we must have $l.h' + r.h' > l.h + r.h$. This can happen only if $l \in V - T$ and $r \in F_R$.

Solution to Exercise 25.3-4

The only way that a vertex $l \in L$ is discovered is either by being a root of the breadth-first search, so that it is unmatched, or by having an edge (r, l) in the matching entering it. If the latter, its only entering edge in $G_{M,h}$ is (r, l) and l will be discovered only when searching the neighbors of r (and l will be r's only neighbor in the search).

Solution to Exercise 25.3-5

To test whether edge (l, r) is in $E_{M,h}$, just determine whether $l.h + r.h == w(l, r)$.

Solution to Exercise 25.3-6

Let W be the maximum edge weight. Compute new edge weights w', where $w'(l, r) = W - w(l, r)$ and solve the maximization problem with weights w'. Let M^* be a solution to the maximization problem, so that $\sum_{(l,r)\in M^*} w'(l, r) = \sum_{(l,r)\in M^*}(W - w(l, r)) = nW - \sum_{(l,r)\in M^*} w(l, r)$, since $|M^*| = n$. Thus, a matching M^* that maximizes $\sum_{(l,r)\in M^*} w'(l, r)$ also minimizes $\sum_{(l,r)\in M^*} w(l, r)$.

Solution to Exercise 25.3-7

Without loss of generality, let $k = |L| - |R| > 0$ and let W be the minimum edge weight. Create k dummy vertices in L, and give every edge incident on a dummy vertex a weight of $W - 1$. That way, it's always better to match a vertex in R with a non-dummy vertex in L than with a dummy vertex.

Solution to Problem 25-1

Solution to part (c).

The algorithm uses divide-and-conquer. If $d = 1$, then E is a perfect matching. Otherwise, $d > 1$ and d is even. Trace out an Euler tour on each connected component of G. As you trace, keep track of whether you take each edge from L to R or from R to L. Let E_{LR} be the edges taken from L to R and E_{RL} be the edges taken from R to L. Recurse on $G_{LR} = (V, E_{LR})$ and on $G_{RL} = (V, E_{RL})$; these subproblems will each have degree $d/2$. Because d is an exact power of 2, each subproblem will either have $d = 1$ or d even.

The time to trace out the Euler tours on $|E|$ edges is $\Theta(E)$, giving rise to two subproblems, each with $|E|/2$ edges. If you draw out the recursion tree, the time at each level comes to $\Theta(E)$, and the degree halves at each level, so that the base case of $d = 2$ comes after $\lg d$ levels. Thus, the total time is $O(E \lg d)$.

Solution to Problem 25-2

a. Compute $\delta = \min\{r.\sigma : r \in R - F_R\}$.

b. Since δ is subtracted from $l.h$ for all $l \in F_L$, but $r.h$ remains unchanged for all $r \in R - F_R$, set $r.\sigma = r.\sigma - \delta$ for all $r \in R - F_R$.

c. When a vertex $l \in L$ is discovered and enters F_L, it can cause $r.\sigma$ to decrease. Set $r.\sigma = \min\{r.\sigma, l.h + r.h - w(l, r)\}$ for all $r \in R - F_R$, taking $O(n)$ time per vertex in L. Since each vertex in L is added into F_L at most once per call of FIND-AUGMENTING-PATH, the total time spent updating $r.\sigma$ values is $O(n^2)$ per call.

d. Because $G_{M,h}$ does not need to be explicitly computed, each growth step now takes $O(n)$ time, and $O(n^2)$ is spent updating updating $r.\sigma$ values per call as vertices are added into L, each call of FIND-AUGMENTING-PATH takes $O(n^2)$ time. Since there are at most n calls of FIND-AUGMENTING-PATH, the HUNGARIAN procedure can be implemented to run in $O(n^3)$ time.

Solution to Problem 25-3

a. Just add 0-weight edges where there is no edge.

b. *[This solution is related to the solution to Exercise 25.3-7.]*

Let W be the minimum weight of any edge. If $(l, r) \notin E$, add edge (l, r) with weight $W - 1$.

c. We are given an undirected graph $G = (V, E)$. We assume that a cycle in a cycle cover can consist of just one vertex so that a cycle cover in G includes every vertex.

Form a bipartite graph $G' = (V', E')$ with $V' = L \cup R$, $L = \{u_L : u \in V\}$, and $R = \{u_R : u \in V\}$, so that $|L| = |R| = |V|$. Make G' be a complete bipartite graph, so that $E' = \{(u_L, v_R) : u_L \in L \text{ and } v_R \in R\}$. Define the weights of edges in E' by

$$w(u_L, v_R) = \begin{cases} w(u, v) & \text{if } (u, v) \in E \ , \\ 0 & \text{if } (u, v) \notin E \text{ and } u = v \ , \\ -\infty & \text{if } (u, v) \notin E \text{ and } u \neq v \ . \end{cases}$$

We claim that given a cycle cover C in G, we can find a perfect matching M^* in G' that includes no edges with weight $-\infty$. To see why, consider each cycle in C. It is either a single vertex or a cycle containing multiple vertices. If the cycle is a single vertex (u, u) then put the edge (u_L, u_R) into M^*. Otherwise, trace out the cycle. For every edge (u, v) in the cycle, put the edge (u_L, v_R) into M^*. Since every vertex in V is in some cycle in C, every vertex u has one edge in the cycle cover entering it and one edge in the cycle cover leaving it, which means that u_R is matched because of the entering edge and u_L is matched because of the leaving edge. Therefore, M^* is a perfect matching in G'. It corresponds to only edges in G or to single-vertex cycles, and hence M^* contains no edges with weight $-\infty$.

Now we claim that given a maximum-weight perfect matching M^* in G', we can find a maximum-weight cycle cover C in G. We have already shown that since G contains a cycle cover, G' has a perfect matching with no edges of weight $-\infty$. Thus, M^* has no such edges.

Consider a vertex $u \in V$. It has corresponding vertices u_L and u_R in V'. Since the matching M^* is perfect, both u_L and u_R have an incident edge. If they have the same incident edge, that is, $(u_L, u_R) \in M^*$, then the single-vertex cycle containing u is in C. Otherwise, u_L is matched with some vertex $v_R \neq u_R$ and u_R is matched with some vertex $v_L \neq u_L$.

Now consider the subgraph $\widehat{G} = (\widehat{V}, \widehat{E})$ of G, where

$$\begin{aligned} \widehat{V} &= \{u \in V : (u_L, u_R) \notin M^*\} \ , \\ \widehat{E} &= \left\{(u, v) : u, v \in \widehat{V} \text{ and } (u_L, v_R) \in M^*\right\} \ . \end{aligned}$$

Every vertex u in $\widehat{G}$ has exactly one entering edge ((v, u) where $(v_L, u_R) \in M^*$) and exactly one leaving edge ((u, v) where $(u_L, v_R) \in M^*$). Since every vertex in $\widehat{G}$ has exactly one entering edge and exactly one leaving edge, $\widehat{G}$ is a union of disjoint cycles. That is, $\widehat{G}$ plus the single-vertex cycles $\{u : (u_L, u_R) \in M^*\}$ forms a cycle cover of G.

Finally, we claim that the cycle cover C given by the maximum-weight perfect matching M^* in G' is a maximum-weight cycle cover. Suppose that G contains some cycle cover C' such that $w(C') > w(C)$. As we've seen, we can create a perfect matching M'^* in G' from C'. The weight of a cycle cover in G is just the sum of the weights of the edges in the corresponding perfect matching in G'. Thus, if $w(C') > w(C)$, then M'^* would have a higher weight than M^*, contradicting the assumption that M^* is a maximum-weight perfect matching.

Therefore, running the Hungarian algorithm on G' and translating the resulting maximum-weight matching to a cycle in G finds a maximum-weight cycle cover.

Solutions for Chapter 26: Parallel Algorithms

Solution to Exercise 26.1-2

There will be no change in the asymptotic work, span, or parallelism of P-FIB even if we were to spawn the recursive call to P-FIB$(n-2)$. The serialization of P-FIB under consideration would yield the same recurrence as that for FIB; we can, therefore, calculate the work as $T_1(n) = \Theta(\phi^n)$. Similarly, because the spawned calls to P-FIB$(n-1)$ and P-FIB$(n-2)$ can run in parallel, we can calculate the span in exactly the same way as in the text, $T_\infty(n) = \Theta(n)$, resulting in $\Theta(\phi^n/n)$ parallelism.

Solution to Exercise 26.1-6

By the work law for $P = 4$, we have $80 = T_4 \geq T_1/4$, or $T_1 \leq 320$. By the span law for $P = 64$, we have $T_\infty \leq T_{64} = 10$. Now we will use inequality (26.5) from Exercise 26.1-4 to derive a contradiction. For $P = 10$, we have

$$\begin{aligned} 42 &= T_{10} \\ &\leq \frac{320 - T_\infty}{10} + T_\infty \\ &= 32 + \frac{9}{10} T_\infty \end{aligned}$$

or, equivalently,

$$\begin{aligned} T_\infty &\geq \frac{10}{9} \cdot 10 \\ &> 10 , \end{aligned}$$

which contradicts $T_\infty \leq 10$.

Therefore, the running times reported by the professor are suspicious.

Solution to Exercise 26.1-7

```
FAST-MAT-VEC(A, x, n)
  let y be a new vector of length n
  parallel for i = 1 to n
      $y_i = 0$
  parallel for i = 1 to n
      $y_i$ = MAT-SUB-LOOP(A, x, i, 1, n)
  return y

MAT-SUB-LOOP(A, x, i, j, j')
  if j == j'
      return $a_{ij} x_j$
  else mid = $\lfloor (j + j')/2 \rfloor$
      lower-half = spawn MAT-SUB-LOOP(A, x, i, j, mid)
      upper-half = MAT-SUB-LOOP(A, x, i, mid + 1, j')
      sync
      return lower-half + upper-half
```

We calculate the work $T_1(n)$ of FAST-MAT-VEC by computing the running time of its serialization, i.e., by replacing the **parallel for** loop by an ordinary **for** loop. Therefore, we have $T_1(n) = n\,T_1'(n)$, where $T_1'(n)$ denotes the work of MAT-SUB-LOOP to compute a given output entry y_i. The work of MAT-SUB-LOOP is given by the recurrence

$$T_1'(n) = 2T_1'(n/2) + \Theta(1) .$$

By applying case 1 of the master theorem, we have $T_1'(n) = \Theta(n)$. Therefore, $T_1(n) = \Theta(n^2)$.

To calculate the span, we use

$$T_\infty(n) = \Theta(\lg n) + \max\{iter_\infty(i) : 1 \le i \le n\} .$$

Note that each iteration of the second **parallel for** loop calls procedure MAT-SUB-LOOP with the same parameters, except for the index i. Because MAT-SUB-LOOP recursively halves the space between its last two parameters (1 and n), does constant-time work in the base case, and spawns one of the recursive calls in parallel with the other, it has span $\Theta(\lg n)$. The procedure FAST-MAT-VEC, therefore, has a span of $\Theta(\lg n)$ and $\Theta(n^2/\lg n)$ parallelism.

Solution to Exercise 26.1-8

We analyze the work of P-TRANSPOSE, as usual, by computing the running time of its serialization, where we replace both the **parallel for** loops with simple **for** loops. We can compute the work of P-TRANSPOSE using the summation

$$\begin{aligned} T_1(n) &= \Theta\left(\sum_{j=2}^{n}(j-1)\right) \\ &= \Theta\left(\sum_{j=1}^{n-1} j\right) \\ &= \Theta(n^2) . \end{aligned}$$

The span of P-TRANSPOSE is determined by the span of the doubly nested **parallel for** loops. Although the number of iterations of the inner loop depends on the value of the variable j of the outer loop, each iteration of the inner loop does constant work. Let $\mathit{iter}_\infty(j)$ denote the span of the jth iteration of the outer loop and $\mathit{iter}'_\infty(i)$ denote the span of the ith iteration of the inner loop. We characterize the span $T_\infty(n)$ of P-TRANSPOSE as

$$T_\infty(n) = \Theta(\lg n) + \max\{\mathit{iter}_\infty(j) : 2 \le j \le n\} .$$

The maximum occurs when $j = n$, and in this case,

$$\mathit{iter}_\infty(n) = \Theta(\lg n) + \max\{\mathit{iter}'_\infty(i) : 1 \le i \le n-1\} .$$

As we noted, each iteration of the inner loop does constant work, and therefore $\mathit{iter}'_\infty(i) = \Theta(1)$ for all i. Thus, we have

$$\begin{aligned} T_\infty(n) &= \Theta(\lg n) + \Theta(\lg n) + \Theta(1) \\ &= \Theta(\lg n) . \end{aligned}$$

Since the work P-TRANSPOSE is $\Theta(n^2)$ and its span is $\Theta(\lg n)$, the parallelism of P-TRANSPOSE is $\Theta(n^2/\lg n)$.

Solution to Exercise 26.1-9

If we were to replace the inner **parallel for** loop of P-TRANSPOSE with an ordinary **for** loop, the work would still remain $\Theta(n^2)$. The span, however, would increase to $\Theta(n)$ because the last iteration of the **parallel for** loop, which dominates the span of the computation, would lead to $(n-1)$ iterations of the inner, serial **for** loop. The parallelism, therefore, would reduce to $\Theta(n^2)/\Theta(n) = \Theta(n)$.

Solution to Exercise 26.1-10

Based on the values of work and span given for the two versions of the chess program, we solve for P using

$$\frac{2048}{P} + 1 = \frac{1024}{P} + 8 .$$

The solution gives P between 146 and 147.

Solution to Exercise 26.2-3

```
P-FAST-MATRIX-MULTIPLY$(A, B, C, n)$
  parallel for $i = 1$ to $n$
      parallel for $j = 1$ to $n$
          $c_{ij} = c_{ij} +$ MATRIX-MULT-SUBLOOP$(A, B, i, j, 1, n)$

MATRIX-MULT-SUBLOOP$(A, B, i, j, k, k')$
  if $k == k'$
      return $a_{ik}b_{kj}$
  else $mid = \lfloor (k + k')/2 \rfloor$
      $lower\text{-}half$ = spawn MATRIX-MULT-SUBLOOP$(A, B, i, j, k, mid)$
      $upper\text{-}half$ = MATRIX-MULT-SUBLOOP$(A, B, i, j, mid + 1, k')$
      sync
      return $lower\text{-}half + upper\text{-}half$
```

We calculate the work $T_1(n)$ of P-FAST-MATRIX-MULTIPLY by computing the running time of its serialization, i.e., by replacing the **parallel for** loops by ordinary **for** loops. Therefore, we have $T_1(n) = n^2\, T_1'(n)$, where $T_1'(n)$ denotes the work of MATRIX-MULT-SUBLOOP to compute a given output entry c_{ij}. The work of MATRIX-MULT-SUBLOOP is given by the recurrence

$$T_1'(n) = 2T_1'(n/2) + \Theta(1) .$$

By applying case 1 of the master theorem, we have $T_1'(n) = \Theta(n)$. Therefore, $T_1(n) = \Theta(n^3)$.

To calculate the span, we use

$$T_\infty(n) = \Theta(\lg n) + \max\{iter_\infty(i) : 1 \le i \le n\} .$$

Note that each iteration of the outer **parallel for** loop does the same amount of work: it calls the inner **parallel for** loop. Similarly, each iteration of the inner **parallel for** loop calls procedure MATRIX-MULT-SUBLOOP with the same parameters, except for the indices i and j. Because MATRIX-MULT-SUBLOOP recursively halves the space between its last two parameters (1 and n), does constant-time work in the base case, and spawns one of the recursive calls in parallel with the other, it has span $\Theta(\lg n)$. Since each iteration of the inner **parallel for** loop, which has n iterations, has span $\Theta(\lg n)$, the inner **parallel for** loop has span $\Theta(\lg n)$. By similar logic, the outer **parallel for** loop, and hence procedure P-FAST-MATRIX-MULTIPLY, has span $\Theta(\lg n)$ and $\Theta(n^3/\lg n)$ parallelism.

Solution to Exercise 26.2-4

We can efficiently multiply a $p \times q$ matrix by a $q \times r$ matrix in parallel by using the solution to Exercise 26.2-3 as a base. We will replace the upper limits of the nested **parallel for** loops with p and r respectively and we will pass q as the last argument to the call of MATRIX-MULT-SUBLOOP. We present the pseudocode for a multithreaded algorithm for multiplying a $p \times q$ matrix by a $q \times r$ matrix in procedure P-GEN-MATRIX-MULTIPLY below. Because the pseudocode for procedure MATRIX-MULT-SUBLOOP (which P-GEN-MATRIX-MULTIPLY calls) remains the same as was presented in the solution to Exercise 26.2-3, we do not repeat it here.

```
P-GEN-MATRIX-MULTIPLY$(A, B, C, p, q, r)$
  parallel for $i = 1$ to $p$
      parallel for $j = 1$ to $r$
          $c_{ij} = c_{ij} +$ MATRIX-MULT-SUBLOOP$(A, B, i, j, 1, q)$
```

To calculate the work for P-GEN-MATRIX-MULTIPLY, we replace the **parallel for** loops with ordinary **for** loops. As before, we can calculate the work of MATRIX-MULT-SUBLOOP to be $\Theta(q)$ (because the input size to the procedure is q here). Therefore, the work of P-GEN-MATRIX-MULTIPLY is $T_1 = \Theta(pqr)$.

We can analyze the span of P-GEN-MATRIX-MULTIPLY as we did in the solution to Exercise 26.2-3, but we must take into account the different number of loop iterations. Each of the p iterations of the outer **parallel for** loop executes the inner **parallel for** loop, and each of the r iterations of the inner **parallel for** loop calls MATRIX-MULT-SUBLOOP, whose span is given by $\Theta(\lg q)$. We know that, in general, the span of a **parallel for** loop with n iterations, where the ith iteration has span $iter_\infty(i)$ is given by

$$T_\infty = \Theta(\lg n) + \max\{iter_\infty(i) : 1 \le i \le n\} .$$

Based on the above observations, we can calculate the span of P-GEN-MATRIX-MULTIPLY as

$$\begin{aligned} T_\infty &= \Theta(\lg p) + \Theta(\lg r) + \Theta(\lg q) \\ &= \Theta(\lg(pqr)) . \end{aligned}$$

The parallelism of the procedure is, therefore, given by $\Theta(pqr/\lg(pqr))$. To check whether this analysis is consistent with Exercise 26.2-3, we note that if $p = q = r = n$, then the parallelism of P-GEN-MATRIX-MULTIPLY would be $\Theta(n^3/\lg n^3) = \Theta(n^3/\lg n)$.

Solution to Exercise 26.2-5

```
P-FLOYD-WARSHALL$(W, n)$
  parallel for $i = 1$ to $n$
      parallel for $j = 1$ to $n$
          $d_{ij} = w_{ij}$
  for $k = 1$ to $n$
      parallel for $i = 1$ to $n$
          parallel for $j = 1$ to $n$
              $d_{ij} = \min\{d_{ij}, d_{ik} + d_{kj}\}$
  return $D$
```

By Exercise 23.2-4, we can compute all the d_{ij} values in parallel.

The work of P-FLOYD-WARSHALL is the same as the running time of its serialization, which we computed as $\Theta(n^3)$ in Section 23.2. The span of the doubly nested **parallel for** loops, which do constant work inside, is $\Theta(\lg n)$. Note, however, that the second set of doubly nested **parallel for** loops is executed within each of the n iterations of the outermost serial **for** loop. Therefore, P-FLOYD-WARSHALL has span $\Theta(n \lg n)$ and $\Theta(n^2/\lg n)$ parallelism.

Solution to Problem 26-1

a. Similar to MAT-VEC-MAIN-LOOP, the required procedure, which we name NESTED-SUM-ARRAYS, will take parameters i and j to specify the range of the array that is being computed in parallel. In order to perform the pairwise addition of two n-element arrays A and B and store the result into array C, we call NESTED-SUM-ARRAYS$(A, B, C, 1, n)$.

```
NESTED-SUM-ARRAYS$(A, B, C, i, j)$
  if $i == j$
      $C[i] = A[i] + B[i]$
  else $k = \lfloor (i + j)/2 \rfloor$
      spawn NESTED-SUM-ARRAYS$(A, B, C, i, k)$
      NESTED-SUM-ARRAYS$(A, B, C, k + 1, j)$
      sync
```

The work of NESTED-SUM-ARRAYS is given by the recurrence

$$\begin{aligned} T_1(n) &= 2T_1(n/2) + \Theta(1) \\ &= \Theta(n) , \end{aligned}$$

by case 1 of the master theorem. The span of the procedure is given by the recurrence

$$\begin{aligned} T_\infty(n) &= T_\infty(n/2) + \Theta(1) \\ &= \Theta(\lg n) , \end{aligned}$$

by case 2 of the master theorem. Therefore, the above algorithm has $\Theta(n/\lg n)$ parallelism.

b. Because ADD-SUBARRAY is serial, we can calculate both its work and span to be $\Theta(j - i + 1)$, which based on the arguments from the call in SUM-ARRAYS$'$ is $\Theta(\textit{grain-size})$, for all but the last call (which is $O(\textit{grain-size})$).

If $\textit{grain-size} = 1$, the procedure SUM-ARRAYS$'$ calculates r to be n, and each of the n iterations of the serial **for** loop spawns ADD-SUBARRAY with the same value, $k + 1$, for the last two arguments. For example, when $k = 0$, the last two arguments to ADD-SUBARRAY are 1, when $k = 1$, the last two arguments are 2, and so on. That is, in each call to ADD-SUBARRAY, its **for** loop iterates once and calculates a single value in the array C. When $\textit{grain-size} = 1$, the **for** loop in SUM-ARRAYS$'$ iterates n times and each iteration takes $\Theta(1)$ time, resulting in $\Theta(n)$ work.

Although the **for** loop in Sum-Arrays′ looks serial, note that each iteration spawns the call to Add-Subarray and the procedure waits for all its spawned children at the end of the **for** loop. That is, all loop iterations of Sum-Arrays′ execute in parallel. Therefore, one might be tempted to say that the span of Sum-Arrays′ is equal to the span of a single call to Add-Subarray plus the constant work done by the first three lines in Sum-Arrays′, giving $\Theta(1)$ span and $\Theta(n)$ parallelism. This calculation of span and parallelism would be wrong, however, because there are r spawns of Add-Subarray in Sum-Arrays′, where r is not a constant. Hence, we must add a $\Theta(r)$ term to the span of Sum-Arrays′ in order to account for the overhead of spawning r calls to Add-Subarray.

Based on the above discussion, the span of Sum-Arrays′ is $\Theta(r) + \Theta(\textit{grain-size}) + \Theta(1)$. When $\textit{grain-size} = 1$, we get $r = n$; therefore, Sum-Arrays′ has $\Theta(n)$ span and $\Theta(1)$ parallelism.

c. For a general *grain-size*, each iteration of the **for** loop in Sum-Arrays′ except for the last results in *grain-size* iterations of the **for** loop in Add-Subarray. In the last iteration of Sum-Arrays′, the **for** loop in Add-Subarray iterates $n \bmod \textit{grain-size}$ times. Therefore, we can claim that the span of Add-Subarray is $\Theta(\max\{\textit{grain-size},\ n \bmod \textit{grain-size}\}) = \Theta(\textit{grain-size})$.

Sum-Arrays′ achieves maximum parallelism when its span, given by $\Theta(r) + \Theta(\textit{grain-size}) + \Theta(1)$, is minimum. Since $r = \lceil n/\textit{grain-size} \rceil$, the minimum occurs when $r \approx \textit{grain-size}$, i.e., when $\textit{grain-size} \approx \sqrt{n}$.

Solution to Problem 26-2

a.

```
P-Matrix-Multiply-Recursive′(A, B, C, n)
    if n == 1
        $c_{11} = c_{11} + a_{11}b_{11}$
        return
    partition A, B, and C into $n/2 \times n/2$ submatrices
        $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$; and
        $C_{11}, C_{12}, C_{21}, C_{22}$; respectively
    spawn P-Matrix-Multiply-Recursive′($A_{11}, B_{11}, C_{11}, n/2$)
    spawn P-Matrix-Multiply-Recursive′($A_{11}, B_{12}, C_{12}, n/2$)
    spawn P-Matrix-Multiply-Recursive′($A_{21}, B_{11}, C_{21}, n/2$)
    P-Matrix-Multiply-Recursive′($A_{21}, B_{12}, C_{22}, n/2$)
    sync
    spawn P-Matrix-Multiply-Recursive′($A_{12}, B_{21}, C_{11}, n/2$)
    spawn P-Matrix-Multiply-Recursive′($A_{12}, B_{22}, C_{12}, n/2$)
    spawn P-Matrix-Multiply-Recursive′($A_{22}, B_{21}, C_{21}, n/2$)
    P-Matrix-Multiply-Recursive′($A_{22}, B_{22}, C_{22}, n/2$)
    sync
```

b. The recurrence for the work $M_1'(n)$ of P-Matrix-Multiply-Recursive′ is $8M_1'(n/2)+\Theta(1)$, which gives us $M_1'(n) = \Theta(n^3)$. Therefore, $T_1(n) = \Theta(n^3)$.

In P-MATRIX-MULTIPLY-RECURSIVE′, there are two groups of spawned recursive calls; therefore, the span $M'_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE′ is given by the recurrence $M'_\infty(n) = 2M'_\infty(n/2) + \Theta(1)$, which gives us $M'_\infty(n) = \Theta(n)$. Because the span $\Theta(n)$ of P-MATRIX-MULTIPLY-RECURSIVE′ dominates, we have $T_\infty(n) = \Theta(n)$.

c. The parallelism of P-MATRIX-MULTIPLY-RECURSIVE′ is $\Theta(n^3/n) = \Theta(n^2)$. Ignoring the constants in the Θ-notation, the parallelism for multiplying 1000×1000 matrices is $1000^2 = 10^6$, which is only a factor of 10 less than that of P-MATRIX-MULTIPLY-RECURSIVE. Although the parallelism of the new procedure is much less than that of P-MATRIX-MULTIPLY-RECURSIVE, the algorithm still scales well for a large number of processors.

Solution to Problem 26-4

a. Here is a multithreaded $\otimes$-reduction algorithm:

```
P-REDUCE(x, i, j)
  if i == j
      return x[i]
  else mid = ⌊(i + j)/2⌋
      lower-half = spawn P-REDUCE(x, i, mid)
      upper-half = P-REDUCE(x, mid + 1, j)
      sync
      return lower-half ⊗ upper-half
```

If we denote the length $j - i + 1$ of the subarray $x[i:j]$ by n, then the work for the above algorithm is given by the recurrence $T_1(n) = 2T_1(n/2) + \Theta(1) = \Theta(n)$. Because one of the recursive calls to P-REDUCE is spawned and the procedure does constant work following the recursive calls and in the base case, the span is given by the recurrence $T_\infty(n) = T_\infty(n/2) + \Theta(1) = \Theta(\lg n)$.

b. The work and span of P-SCAN-1-AUX dominate the work and span of P-SCAN-1. We can calculate the work of P-SCAN-1-AUX by replacing the **parallel for** loop with an ordinary **for** loop and noting that in each iteration, the running time of P-REDUCE will be equal to $\Theta(l)$. Since P-SCAN-1 calls P-SCAN-1-AUX with 1 and n as the last two arguments, the running time of P-SCAN-1, and hence its work, is $\Theta(1 + 2 + \cdots + n) = \Theta(n^2)$.

As we noted earlier, the **parallel for** loop in P-SCAN-1-AUX undergoes n iterations; therefore, the span of P-SCAN-1-AUX is given by $\Theta(\lg n)$ for the recursive splitting of the loop iterations plus the span of the iteration that has maximum span. Among the loop iterations, the call to P-REDUCE in the last iteration (when $l = n$) has the maximum span, equal to $\Theta(\lg n)$. Thus, P-SCAN-1 has $\Theta(\lg n)$ span and $\Theta(n^2/\lg n)$ parallelism.

c. In P-SCAN-2-AUX, before the **parallel for** loop in lines 7–8 executes, the following invariant is satisfied: $y[l] = x[i] \otimes x[i+1] \otimes \cdots \otimes x[l]$ for $l = i, i+1, \ldots, k$ and $y[l] = x[k+1] \otimes x[k+2] \otimes \cdots \otimes x[l]$ for

$l = k+1, k+2, \ldots, j$. The **parallel for** loop need not update $y[i], \ldots, y[k]$, since they have the correct values after the call to P-SCAN-2-AUX(x, y, i, k). For $l = k+1, k+2, \ldots, j$, the **parallel for** loop sets

$$\begin{aligned} y[l] &= y[k] \otimes y[l] \\ &= x[i] \otimes \cdots \otimes x[k] \otimes x[k+1] \otimes \cdots \otimes x[l] \\ &= x[i] \otimes \cdots \otimes x[l] \,, \end{aligned}$$

as desired. We can run this loop in parallel because the lth iteration depends only on the values of $y[k]$, which is the same in all iterations, and $y[l]$. Therefore, when the call to P-SCAN-2-AUX from P-SCAN-2 returns, array y represents the $\otimes$-prefix computation of array x.

Because the work and span of P-SCAN-2-AUX dominate the work and span of P-SCAN-2, we will concentrate on calculating these values for P-SCAN-2-AUX working on an array of size n. The work $PS2A_1(n)$ of P-SCAN-2-AUX is given by the recurrence $PS2A_1(n) = 2PS2A_1(n/2) + \Theta(n)$, which equals $\Theta(n \lg n)$ by case 2 of the master theorem. The span $PS2A_\infty(n)$ of P-SCAN-2-AUX is given by the recurrence $PS2A_\infty(n) = PS2A_\infty(n/2) + \Theta(\lg n)$, which equals $\Theta(\lg^2 n)$ per case 2 of the master theorem. That is, the work, span, and parallelism of P-SCAN-2 are $\Theta(n \lg n)$, $\Theta(\lg^2 n)$, and $\Theta(n/\lg n)$, respectively.

d. The missing expression in line 8 of P-SCAN-UP is $t[k] \otimes \textit{right}$. The missing expressions in lines 5 and 6 of P-SCAN-DOWN are v and $v \otimes t[k]$, respectively.

As suggested in the hint, we will prove that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$, so that the value $v \otimes x[i]$ stored into $y[i]$ in the base case of P-SCAN-DOWN is correct.

In order to compute the arguments that are passed to P-SCAN-DOWN, we must first understand what $t[k]$ holds as a result of the call to P-SCAN-UP. A call to P-SCAN-UP(x, t, i, j) returns $x[i] \otimes \cdots \otimes x[j]$; because $t[k]$ stores the return value of P-SCAN-UP(x, t, i, k), we can say that $t[k] = x[i] \otimes \cdots \otimes x[k]$.

The value $v = x[1]$ when P-SCAN-DOWN$(x[1], x, t, y, 2, n)$ is called from P-SCAN-3 clearly satisifies $v = x[1] \otimes \cdots \otimes x[i-1]$. Let us suppose that $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$ in a call of P-SCAN-DOWN(v, x, t, y, i, j). Therefore, v meets the required condition in the first recursive call, with i and k as the last two arguments, in P-SCAN-DOWN. If we can prove that the value $v \otimes t[k]$ passed to the second recursive call in P-SCAN-DOWN equals $x[1] \otimes x[2] \otimes \cdots \otimes x[k]$, we would have proved the required condition on v for all calls to P-SCAN-DOWN. Earlier, we proved that $t[k] = x[i] \otimes \cdots \otimes x[k]$; therefore,

$$\begin{aligned} v \otimes t[k] &= x[1] \otimes x[2] \otimes \cdots \otimes x[i-1] \otimes x[i] \otimes \cdots x[k] \\ &= x[1] \otimes x[2] \otimes \cdots \otimes x[k] \,. \end{aligned}$$

Thus, the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.

e. Let $PSU_1(n)$ and $PSU_\infty(n)$ denote the work and span of P-SCAN-UP and let $PSD_1(n)$ and $PSD_\infty(n)$ denote the work and span of P-SCAN-DOWN. Then the expressions $T_1(n) = PSU_1(n) + PSD_1(n) + \Theta(1)$ and $T_\infty(n) = PSU_\infty(n) + PSD_\infty(n) + \Theta(1)$ characterize the work and span of P-SCAN-3.

The work $PSU_1(n)$ of P-SCAN-UP is given by the recurrence

$$PSU_1(n) = 2PSU_1(n/2) + \Theta(1) ,$$

and its span is defined by the recurrence

$$PSU_\infty(n) = PSU_\infty(n/2) + \Theta(1) .$$

Using the master theorem to solve these recurrences, we get $PSU_1(n) = \Theta(n)$ and $PSU_\infty(n) = \Theta(\lg n)$.

Similarly, the recurrences

$$\begin{aligned} PSD_1(n) &= 2PSD_1(n/2) + \Theta(1) , \\ PSD_\infty(n) &= PSD_\infty(n/2) + \Theta(1) \end{aligned}$$

define the work and span of P-SCAN-DOWN, and they evaluate to $PSD_1(n) = \Theta(n)$ and $PSD_\infty(n) = \Theta(\lg n)$.

Applying the results for the work and span of P-SCAN-UP and P-SCAN-DOWN obtained above in the expressions for the work and span of P-SCAN-3, we get $T_1(n) = \Theta(n)$ and $T_\infty(n) = \Theta(\lg n)$. Hence, P-SCAN-3 has $\Theta(n/\lg n)$ parallelism. P-SCAN-3 performs less work than P-SCAN-1, but with the same span, and it has the same parallelism as P-SCAN-2 with less work and a lower span.

f. *[Solution for this part is omitted.]*

g. The procedure P-SCAN-4 first computes an ***exclusive scan***, which is where the value in the ith position is based on the values in positions 1 through $i - 1$ only. It then incorporates the value in the ith position by using the result of the exclusive scan in position $i + 1$. The value in the last position is computed early on and saved, so that it can be placed into the last position at the end. The parameter *id* to P-SCAN-4 is the identity for the $\otimes$ operator.

```
P-SCAN-4(A, n, id)
  P-SCAN-UP-IN-PLACE(A, 1, n)
  sum = A[n]                      // A[n] contains the reduction
  P-SCAN-DOWN-IN-PLACE(A, 1, n, id)
  parallel for i = 1 to n - 1
      A[i] = A[i + 1]
  A[n] = sum
```

```
P-SCAN-UP-IN-PLACE(A, p, r)
  if $p \neq r$
      $q = \lfloor (p + r)/2 \rfloor$
      spawn P-SCAN-UP-IN-PLACE(A, p, q)
      P-SCAN-UP-IN-PLACE(A, q + 1, r)
      sync
      $A[r] = A[r] \otimes A[q]$
```

```
P-SCAN-DOWN-IN-PLACE($A, p, r, prior\text{-}sum$)
  if $p == r$
      $A[p] = prior\text{-}sum$
  else $q = \lfloor (p + r)/2 \rfloor$
      $left\text{-}sum = A[q]$
      spawn P-SCAN-DOWN-IN-PLACE($A, p, q, prior\text{-}sum$)
      P-SCAN-DOWN-IN-PLACE($A, q + 1, r, prior\text{-}sum \otimes left\text{-}sum$)
      sync
```

These procedures have $\Theta(n)$ work and $\Theta(\lg n)$ span.

h. Taking a cue from the hint, create an array with $+1$ for every left parenthesis and -1 for every right parenthesis. Perform a $+$-scan on the array. Then check that the last value in the result of the scan is 0; if not, then the string is not well formed. If the last value is 0, then in parallel compare each number in the result of the scan with 0, creating a new array with 1 in every nonnegative position and 0 in every negative position. Compute an AND-reduction on this array. The string is well formed if and only if the result of the AND-reduction is 1.

Solution to Problem 26-5

a. In this part of the problem, we will assume that n is an exact power of 2, so that in a recursive step, when we divide the $n \times n$ matrix A into four $n/2 \times n/2$ matrices, we will be guaranteed that $n/2$ is an integer, for all $n \geq 2$. We make this assumption simply to avoid introducing $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ terms in the pseudocode and the analysis that follow. In the pseudocode below, we assume that we have a procedure BASE-CASE available to us, which calculates the base case of the stencil.

```
SIMPLE-STENCIL($A, i, j, n$)
  if $n == 1$
      $A[i, j]$ = BASE-CASE($A, i, j$)
  else // Calculate submatrix $A_{11}$.
      SIMPLE-STENCIL($A, i, j, n/2$)
      // Calculate submatrices $A_{12}$ and $A_{21}$ in parallel.
      spawn SIMPLE-STENCIL($A, i, j + n/2, n/2$)
      SIMPLE-STENCIL($A, i + n/2, j, n/2$)
      sync
      // Calculate submatrix $A_{22}$.
      SIMPLE-STENCIL($A, i + n/2, j + n/2, n/2$)
```

To perform a simple stencil calculation on an $n \times n$ matrix A, we call SIMPLE-STENCIL$(A, 1, 1, n)$. The recurrence for the work is $T_1(n) = 4T_1(n/2) + \Theta(1) = \Theta(n^2)$. Of the four recursive calls in the algorithm above, only two run in parallel. Therefore, the recurrence for the span is $T_\infty(n) = 3T_\infty(n/2) + \Theta(1) = \Theta(n^{\lg 3})$, and the parallelism is $\Theta(n^{2-\lg 3}) \approx \Theta(n^{0.415})$.

b. Similar to SIMPLE-STENCIL of the previous part, we present P-STENCIL-3, which divides A into nine submatrices, each of size $n/3 \times n/3$, and solves them recursively. To perform a stencil calculation on an $n \times n$ matrix A, we call P-STENCIL-3$(A, 1, 1, n)$.

```
P-STENCIL-3(A, i, j, n)
  if n == 1
      A[i, j] = BASE-CASE(A, i, j)
  else // Group 1: compute submatrix A_{11}.
      P-STENCIL-3(A, i, j, n/3)
      // Group 2: compute submatrices A_{12} and A_{21}.
      spawn P-STENCIL-3(A, i, j + n/3, n/3)
      P-STENCIL-3(A, i + n/3, j, n/3)
      sync
      // Group 3: compute submatrices A_{13}, A_{22}, and A_{31}.
      spawn P-STENCIL-3(A, i, j + 2n/3, n/3)
      spawn P-STENCIL-3(A, i + n/3, j + n/3, n/3)
      P-STENCIL-3(A, i + 2n/3, j, n/3)
      sync
      // Group 4: compute submatrices A_{23} and A_{32}.
      spawn P-STENCIL-3(A, i + n/3, j + 2n/3, n/3)
      P-STENCIL-3(A, i + 2n/3, j + n/3, n/3)
      sync
      // Group 5: compute submatrix A_{33}.
      P-STENCIL-3(A, i + 2n/3, j + 2n/3, n/3)
```

From the pseudocode, we can informally say that we can solve the nine subproblems in five groups, as shown in the following matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix} .$$

Each entry in the above matrix specifies the group of the corresponding $n/3 \times n/3$ submatrix of A; we can compute in parallel the entries of all submatrices that fall in the same group. In general, for $i = 2, 3, 4, 5$, we can calculate group i after completing the computation of group $i - 1$.

The recurrence for the work is $T_1(n) = 9T_1(n/3) + \Theta(1) = \Theta(n^2)$. The recurrence for the span is $T_\infty(n) = 5T_\infty(n/3) + \Theta(1) = \Theta(n^{\log_3 5})$. Therefore, the parallelism is $\Theta(n^{2-\log_3 5}) \approx \Theta(n^{0.535})$.

c. Similar to the previous part, we can solve the b^2 subproblems in $2b - 1$ groups:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & b-2 & b-1 & b \\ 2 & 3 & 4 & \cdots & b-1 & b & b+1 \\ 3 & 4 & 5 & \cdots & b & b+1 & b+2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ b-2 & b-1 & b & \cdots & 2b-5 & 2b-4 & 2b-3 \\ b-1 & b & b+1 & \cdots & 2b-4 & 2b-3 & 2b-2 \\ b & b+1 & b+2 & \cdots & 2b-3 & 2b-2 & 2b-1 \end{pmatrix} .$$

The recurrence for the work is $T_1(n) = b^2 T_1(n/b) + \Theta(1) = \Theta(n^2)$. The recurrence for the span is $T_\infty(n) = (2b-1)T_\infty(n/b) + \Theta(1) = \Theta(n^{\log_b(2b-1)})$. The parallelism is $\Theta(n^{2-\log_b(2b-1)})$.

As the hint suggests, in order to show that the parallelism must be $o(n)$ for any choice of $b \geq 2$, we need to show that $2 - \log_b(2b-1)$, which is the exponent of n in the parallelism, is strictly less than 1 for any choice of $b \geq 2$. Since $b \geq 2$, we know that $2b - 1 > b$, which implies that $\log_b(2b-1) > \log_b b = 1$. Hence, $2 - \log_b(2b-1) < 2 - 1 = 1$.

d. The idea behind achieving $\Theta(n/\lg n)$ parallelism is similar to that presented in the previous part, except without recursive division. We will compute $A[1,1]$ serially, which will enable us to compute entries $A[1,2]$ and $A[2,1]$ in parallel, after which we can compute entries $A[1,3]$, $A[2,2]$ and $A[3,1]$ in parallel, and so on. Here is the pseudocode:

```
P-STENCIL(A, n)
  // Calculate all entries on the antidiagonal and above it.
  for i = 1 to n
      parallel for j = 1 to i
          A[i - j + 1, j] = BASE-CASE(A, i - j + 1, j)
  // Calculate all entries below the antidiagonal.
  for i = 2 to n
      parallel for j = i to n
          A[n + i - j, j] = BASE-CASE(A, n + i - j, j)
```

For each value of index i of the first serial **for** loop, the inner loop iterates i times, doing constant work in each iteration. Because index i ranges from 1 to n in the first **for** loop, we require $\Theta(1 + 2 + \cdots + n) = \Theta(n^2)$ work to calculate all entries on the antidiagonal and above it. For each value of index i of the second serial **for** loop, the inner loop iterates $n - i + 1$ times, doing constant work in each iteration. Because index i ranges from 2 to n in the second **for** loop, we require $\Theta((n-1) + (n-2) + \cdots + 1) = \Theta(n^2)$ work to calculate all entries on the antidiagonal and above it. Therefore, the work of P-STENCIL is $T_1(n) = \Theta(n^2)$.

Note that both **for** loops in P-STENCIL, which execute **parallel for** loops within, are serial. Therefore, in order to calculate the span of P-STENCIL, we must add the spans of all the **parallel for** loops. Given that any **parallel for** loop in P-STENCIL does constant work in each iteration, the span of a **parallel for** loop with n' iterations is $\Theta(\lg n')$. Hence,

$$\begin{aligned} T_\infty(n) &= \Theta((\lg 1 + \lg 2 + \cdots + \lg n) + (\lg(n-1) + \cdots + 1)) \\ &= \Theta(\lg(n!) + \lg(n-1)!) \\ &= \Theta(n \lg n) , \end{aligned}$$

giving us $\Theta(n/\lg n)$ parallelism.

Lecture Notes for Chapter 30: Polynomials and the FFT

Chapter 30 overview

Chapter outline

Two ways to represent polynomials: coefficient representation and point-value representation. Takes $\Theta(n^2)$ time to multiply polynomials of degree n in coefficient form, using the straightforward method. But takes only $\Theta(n)$ time when in point-value form. Will see how to convert back and forth in $\Theta(n \lg n)$ time by using the fast Fourier transform (FFT), which will yield a $\Theta(n \lg n)$-time algorithm for multiplication in coefficient form.

Fourier transforms:

- Commonly used in signal processing.
- Signal is given in the ***time domain*** as a function mapping time to amplitude.
- Fourier analysis expresses the signal as a weighted sum of phase-shifted sinusoids of varying frequencies.
- Weights and phases associated with the frequencies characterize the signal in the ***frequency domain***.
- FFT is a fast way to compute a Fourier transform. We will see how to design a circuit to compute the FFT.

Note: In this chapter, i always means $\sqrt{-1}$. We don't use i to denote an index.

Polynomials

A ***polynomial*** in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum: $A(x) = \sum_{j=0}^{n-1} a_j x^j$.

- ***Coefficients*** of the polynomial: values $a_0, a_1, \ldots, a_{n-1}$.
- Coefficients and x are drawn from a field F, typically $\mathbb{C}$ (complex numbers).
- Polynomial $A(x)$ has ***degree*** k if its highest nonzero coefficient is a_k. Notation: $\text{degree}(A) = k$.
- Any integer strictly greater than the degree of a polynomial is a ***degree-bound*** of the polynomial.

A variety of operations extend to polynomials.

Polynomial addition: If $A(x)$ and $B(x)$ are polynomials of degree-bound n, their ***sum*** is a polynomial $C(x)$ also of degree-bound n, such that $C(x) = A(x) + B(x)$ for all x in the underlying field.

If $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$, then $C(x) = \sum_{j=0}^{n-1} c_j x^j$, where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$.

Example: Let $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$. Then, $A(x) + B(x) = 4x^3 + 7x^2 - 6x + 4$.

Polynomial multiplication: If $A(x)$ and $B(x)$ are polynomials of degree-bound n, their ***product*** $C(x)$ is a polynomial of degree-bound $2n - 1$ such that $C(x) = A(x)B(x)$ for all x in the underlying field.

Therefore, $C(x) = \sum_{j=0}^{2n-2} c_j x^j$, where $c_j = \sum_{k=0}^{j} a_k b_{j-k}$.

Example: Multiply $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$ as follows:

$$\begin{array}{rrrrrrrl}
 & & & 6x^3 & +\ 7x^2 & -\ 10x & +\ 9 & \\
 & & & -\ 2x^3 & & +\ 4x & -\ 5 & \\
\cline{4-7}
 & & & -\ 30x^3 & -\ 35x^2 & +\ 50x & -\ 45 & \text{(multiply } A(x) \text{ by } -5\text{)} \\
 & & 24x^4 & +\ 28x^3 & -\ 40x^2 & +\ 36x & & \text{(multiply } A(x) \text{ by } 4x\text{)} \\
-\ 12x^6 & -\ 14x^5 & +\ 20x^4 & -\ 18x^3 & & & & \text{(multiply } A(x) \text{ by } -2x^3\text{)} \\
\cline{1-7}
-\ 12x^6 & -\ 14x^5 & +\ 44x^4 & -\ 20x^3 & -\ 75x^2 & +\ 86x & -\ 45 &
\end{array}$$

Representing polynomials

Each polynomial in point-value form has a unique counterpart in coefficient form. We'll see these two ways to represent polynomials and how to combine the two representations in order to multiply two degree-bound n polynomials in $\Theta(n \lg n)$ time.

Coefficient representation

The ***coefficient representation*** of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n is a vector of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$.

[In this chapter, matrix equations generally treat vectors as column vectors.]

Operations on polynomials in coefficient representation:

Evaluating the polynomial $A(x)$ at a given point x_0 consists of computing the value of $A(x_0)$. To evaluate a polynomial in $\Theta(n)$ time, use ***Horner's rule***:

$$A(x_0) = a_0 + x_0 \Big(a_1 + x_0 \big(a_2 + \cdots + x_0 \left(a_{n-2} + x_0(a_{n-1}) \right) \cdots \big) \Big) .$$

Adding two polynomials represented by the coefficient vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$ takes $\Theta(n)$ time: just produce the coefficient vector $c = (c_0, c_1, \ldots, c_{n-1})$, where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$.

Multiplying two degree-bound n polynomials $A(x)$ and $B(x)$ represented in coefficient form to get $C(x)$ using the method described in the chapter overview takes $\Theta(n^2)$ time, since it multiplies each coefficient in the vector a by each coefficient in the vector b.

The resulting coefficient vector c is also called the ***convolution*** of the input vectors a and b, denoted $c = a \otimes b$.

Point-value representation

A ***point-value representation*** of a polynomial $A(x)$ of degree-bound n is a set of n ***point-value pairs*** $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ such that all of the x_k are distinct and $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$.

- A polynomial has many different point-value representations, since any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can serve as a basis for the representation.
- To compute a point-value representation, select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$. With Horner's method, evaluating a polynomial at n points takes time $\Theta(n^2)$. By choosing certain values of x_k, this computation can run in time $\Theta(n \lg n)$.
- Determining the coefficient form of a polynomial from a point-value representation is called ***interpolation***.
- Interpolation is well defined when the desired interpolating polynomial has a degree-bound equal to the given number of point-value pairs, as shown in the following theorem.

Theorem (Uniqueness of an interpolating polynomial)
For any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$. ■

Proof The equations $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$ are equivalent to the matrix equation

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

The matrix on the left is the ***Vandermonde matrix***, denoted $V(x_0, x_1, \ldots, x_{n-1})$, and its determinant is well defined. *[The formula for the determinant appears in the textbook. We don't need the formula for the determinant, just that there is one.]* Because the Vandermonde matrix has a determinant, it is invertible. Thus, given the vector $y = (y_0, y_1, \ldots, y_{n-1})$, the coefficients $a = (a_0, a_1, \ldots, a_{n-1})$ are given by $a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y$. ■

The coefficients of A can be computed in $\Theta(n^2)$ time using Lagrange's formula *[given in section 30.1]* . Therefore, n-point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation, and they can be computed in time $\Theta(n^2)$.

Operations on point-value representations

Addition: If $C(x) = A(x) + B(x)$, then $C(x_k) = A(x_k) + B(x_k)$ for any point x_k. Given point-value representations for A, $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$, and for B, $\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$, where A and B are evaluated at the *same* n points, then a point-value representation for C is $\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$.
⇒ Can add two polynomials of degree-bound n in point-value form in time $\Theta(n)$.

Multiplication: If $C(x) = A(x)B(x)$, then have $C(x_k) = A(x_k)B(x_k)$ for any point x_k.

- To obtain a point-value representation for C, pointwise multiply a point-value representation for A by a point-value representation for B.
- Recall: in polynomial multiplication, $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$.
- Therefore, if A and B have degree-bound n, then C has degree-bound $2n$.
- A standard point-value representation for A and B consists of n point-value pairs for each polynomial. Multiplying these together gives n point-value pairs, but $2n$ pairs are necessary to interpolate a unique polynomial C of degree-bound $2n$.
- Therefore, begin with "extended" point-value representations for A and for B consisting of $2n$ point-value pairs each.

Given an extended point-value representation for A, $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$, and a corresponding extended point-value representation for B, $\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\}$, then a point-value representation for C is $\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}$.
⇒ Multiplying polynomials in point-value form takes just $\Theta(n)$ time.

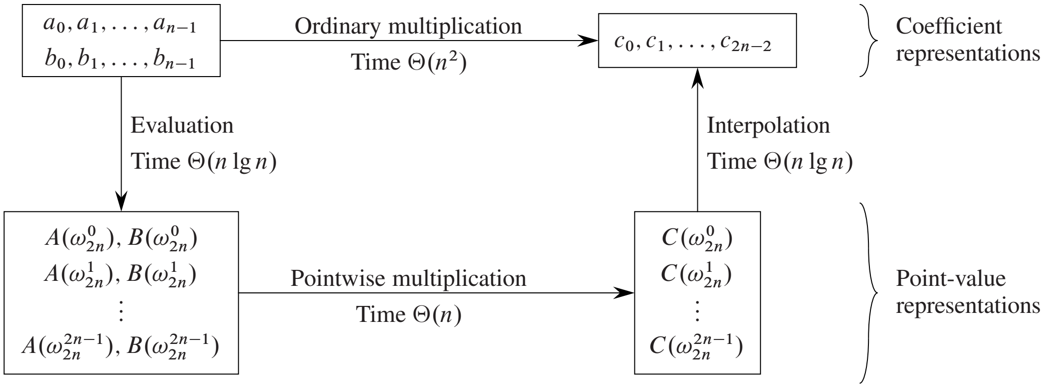
Evaluation at a new point: The simplest approach is to first convert the polynomial from point-value form to coefficient form and then evaluate it at the new point.

Fast multiplication of polynomials in coefficient form

Idea:

- Want to use linear-time multiplication method for polynomials in point-value form to expedite polynomial multiplication in coefficient form.
- Need to be able to evaluate and interpolate quickly.
- Any points can be evaluation points, but certain evaluation points allow to convert between representations in $\Theta(n \lg n)$ time.
- We'll see how to use "complex roots of unity" as the evaluation points for fast conversion.

[The figure below illustrates this strategy. The representations on the top are in coefficient form. Those on the bottom are in point-value form. Arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$th roots of unity, since degree-bounds must be doubled to $2n$.]



$a_0, a_1, \ldots, a_{n-1}$
$b_0, b_1, \ldots, b_{n-1}$
Ordinary multiplication
Time $\Theta(n^2)$
$c_0, c_1, \ldots, c_{2n-2}$
Coefficient representations
Evaluation
Time $\Theta(n \lg n)$
Interpolation
Time $\Theta(n \lg n)$
$A(\omega_{2n}^0), B(\omega_{2n}^0)$
$A(\omega_{2n}^1), B(\omega_{2n}^1)$
$\vdots$
$A(\omega_{2n}^{2n-1}), B(\omega_{2n}^{2n-1})$
Pointwise multiplication
Time $\Theta(n)$
$C(\omega_{2n}^0)$
$C(\omega_{2n}^1)$
$\vdots$
$C(\omega_{2n}^{2n-1})$
Point-value representations

To multiply two polynomials $A(x)$ and $B(x)$ of degree-bound n in $\Theta(n \lg n)$-time, where the input and output representations are in coefficient form, use the FFT to evaluate and interpolate:

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each. Time: $\Theta(n)$.
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ by applying the FFT of order $2n$ on each polynomial. These representations contain the values of the two polynomials at the $(2n)$th roots of unity. Time: $\Theta(n \lg n)$.
3. *Pointwise multiply:* Compute a point-value representation for the polynomial $C(x) = A(x)B(x)$ by multiplying these values together pointwise. This representation contains the value of $C(x)$ at each $(2n)$th root of unity. Time: $\Theta(n)$.
4. *Interpolate:* Create the coefficient representation of the polynomial $C(x)$ by applying the FFT on $2n$ point-value pairs to compute the inverse DFT. Time: $\Theta(n \lg n)$.

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

FFT taking $\Theta(n \lg n)$ time proves the following theorem:

Theorem

It is possible to multiply two polynomials of degree-bound n in time $\Theta(n \lg n)$ with both the input and output representations in coefficient form. ■

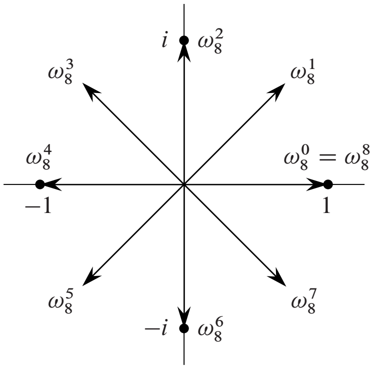
The DFT and FFT

Complex roots of unity

A ***complex nth root of unity*** is a complex number ω such that $\omega^n = 1$.

- There are exactly n complex nth roots of unity: $e^{2\pi ik/n}$ for $k = 0, 1, \ldots, n-1$.
- To interpret the above formula, use the definition of the exponential of a complex number: $e^{iu} = \cos(u) + i\,\sin(u)$.
- As the figure below illustrates, the n complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane.
- The ***principal nth root of unity*** is $\omega_n = e^{2\pi i/n}$. *[Many FFT treatments define $\omega_n = e^{-2\pi i/n}$, especially for signal processing. The underlying mathematics is essentially the same either way.]*
- All other complex nth roots of unity are powers of ω_n.

The values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity:



The n complex nth roots of unity, $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$, form a group under multiplication. This group has the same structure as the additive group $(\mathbb{Z}_n, +)$ modulo n, since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Similarly, $\omega_n^{-1} = \omega_n^{n-1}$.

The following lemmas establish some essential properties of the complex nth roots of unity. *[You might want to omit some of the proofs.]*

Lemma (Cancellation lemma)
For any integers $n \geq 0$, $k \geq 0$, and $d > 0$, $\omega_{dn}^{dk} = \omega_n^k$.

Proof

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^{k} \\ &= \omega_n^k\,. \end{aligned} \qquad \blacksquare$$

Corollary
For any even integer $n > 0$, $\omega_n^{n/2} = \omega_2 = -1$.

[Proof is Exercise 30.2-1 and uses the cancellation lemma: $\omega_n^{n/2} = \omega_{2n}^{2n/2} = \omega_{2n}^{n} = \omega_2 = e^{2\pi i/2} = e^{\pi i} = -1$.]

Lemma (Halving lemma)
If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

Proof *[This is the alternative proof given in the textbook.]*
$\omega_n^{n/2} = -1 \Rightarrow \omega_n^{k+n/2} = \omega_n^k \omega_n^{n/2} = -\omega_n^k \Rightarrow (\omega_n^{k+n/2})^2 = (-\omega_n^k)^2 = (\omega_n^k)^2$. ■

Lemma (Summation lemma)
For any integer $n \geq 1$ and nonzero integer k not divisible by n, $\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = 0$.

Proof The summation formula for a geometric series applies to complex numbers as well as to reals $\Rightarrow$

$$\begin{aligned}
\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\
&= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\
&= \frac{(1)^k - 1}{\omega_n^k - 1} \\
&= 0 .
\end{aligned}$$

$\omega_n^k = 1$ only when k is divisible by n, which the lemma statement prohibits
$\Rightarrow$ denominator is not 0. ■

The DFT

Goal: Evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ (at the n complex nth roots of unity).
Given in coefficient form: $a = (a_0, a_1, \ldots, a_{n-1})$.

Results: y_k, for $k = 0, 1, \ldots, n-1$:

$$\begin{aligned}
y_k &= A(\omega_n^k) \\
&= \sum_{j=0}^{n-1} a_j \omega_n^{kj} .
\end{aligned}$$

Vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the ***discrete Fourier transform (DFT)*** of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$.
Denote $y = \mathrm{DFT}_n(a)$.

The FFT

Fast Fourier transform (FFT) uses properties of the complex roots of unity to compute $\mathrm{DFT}_n(a)$ in time $\Theta(n \lg n)$.

[Assume throughout that n is an exact power of 2. Strategies for dealing with non-power-of-2 sizes exist, but they are not discussed in the textbook.]

Use a divide-and conquer strategy: separate the even-indexed and odd-indexed coefficients of $A(x)$ to define the two new polynomials $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ of degree-bound $n/2$:

$$\begin{aligned} A^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} , \\ A^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} . \end{aligned}$$

- A^{even} contains all the coefficients of A where the binary representation of the index ends in 0.
- A^{odd} contains all the coefficients where the binary representation of the index ends in 1.

$$A(x) = A^{\text{even}}(x^2) + x A^{\text{odd}}(x^2) ,$$

so that evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ becomes

1. Evaluate the degree-bound $n/2$ polynomials $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ at the points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$.
2. Combine the results according to $A(x) = A^{\text{even}}(x^2) + x A^{\text{odd}}(x^2)$.

FFT procedure idea:

- By the halving lemma, the list of values $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$ consists not of n distinct values but only of the $n/2$ complex $(n/2)$th roots of unity, with each root occurring exactly twice.
- FFT recursively evaluates the polynomials A^{even} and A^{odd} of degree-bound $n/2$ at the $n/2$ complex $(n/2)$th roots of unity.
- These subproblems have exactly the same form as the original problem, but are half the size, dividing an n-element DFT_n computation into two $n/2$-element $\mathrm{DFT}_{n/2}$ computations.
- This decomposition is the basis for the FFT procedure, which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is a power of 2.

FFT procedure

```
FFT(a, n)
  if n == 1
      return a                              // DFT of 1 element is the element itself
  $\omega_n = e^{2\pi i/n}$
  $\omega = 1$
  $a^{\text{even}} = (a_0, a_2, \ldots, a_{n-2})$
  $a^{\text{odd}} = (a_1, a_3, \ldots, a_{n-1})$
  $y^{\text{even}} = \text{FFT}(a^{\text{even}}, n/2)$
  $y^{\text{odd}} = \text{FFT}(a^{\text{odd}}, n/2)$
  for $k = 0$ to $n/2 - 1$                  // at this point, $\omega = \omega_n^k$
      $y_k = y_k^{\text{even}} + \omega\, y_k^{\text{odd}}$
      $y_{k+(n/2)} = y_k^{\text{even}} - \omega\, y_k^{\text{odd}}$
      $\omega = \omega\, \omega_n$
  return y
```

Procedure description:

- First two lines are the base case of the recursion, as the DFT of one element is the element itself.
- The lines $a^{\text{even}} = (a_0, a_2, \ldots, a_{n-2})$ and $a^{\text{odd}} = (a_1, a_3, \ldots, a_{n-1})$ define coefficient vectors for the polynomials A^{even} and A^{odd}.
- The lines $\omega_n = e^{2\pi i/n}$, $\omega = 1$, and $\omega = \omega\, \omega_n$ update ω properly.
- The lines
  $y^{\text{even}} = \text{RECURSIVE-FFT}(a^{\text{even}})$
  $y^{\text{odd}} = \text{RECURSIVE-FFT}(a^{\text{odd}})$
  perform the recursive $\text{DFT}_{n/2}$ computations.
  $\Rightarrow y_k^{\text{even}} = A^{\text{even}}(\omega_{n/2}^k)$ and $y_k^{\text{odd}} = A^{\text{odd}}(\omega_{n/2}^k)$.
- The lines $y_k = y_k^{\text{even}} + \omega\, y_k^{\text{odd}}$ and $y_{k+(n/2)} = y_k^{\text{even}} - \omega\, y_k^{\text{odd}}$ combine the results of the recursive $\text{DFT}_{n/2}$ calculations.

By the cancellation lemma, $\omega_{n/2}^k = \omega_n^{2k}$, so that $y_k^{\text{even}} = A^{\text{even}}(\omega_{n/2}^k) = A^{\text{even}}(\omega_n^{2k})$, and $y_k^{\text{odd}} = A^{\text{odd}}(\omega_{n/2}^k) = A^{\text{odd}}(\omega_n^{2k})$.

For the first $n/2$ results $y_0, y_1, \ldots, y_{n/2-1}$, the line $y_k = y_k^{\text{even}} + \omega\, y_k^{\text{odd}}$ gives

$$\begin{aligned} y_k &= y_k^{\text{even}} + \omega_n^k y_k^{\text{odd}} \\ &= A^{\text{even}}(\omega_n^{2k}) + \omega_n^k A^{\text{odd}}(\omega_n^{2k}) \\ &= A(\omega_n^k)\ . \end{aligned}$$

For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, letting $k = 0, 1, \ldots, n/2 - 1$, the line $y_{k+(n/2)} = y_k^{\text{even}} - \omega\, y_k^{\text{odd}}$ yields

$$\begin{aligned} y_{k+(n/2)} &= y_k^{\text{even}} - \omega_n^k y_k^{\text{odd}} \\ &= y_k^{\text{even}} + \omega_n^{k+(n/2)} y_k^{\text{odd}} && (\text{since } \omega_n^{k+(n/2)} = -\omega_n^k) \\ &= A^{\text{even}}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{\text{odd}}(\omega_n^{2k}) \\ &= A^{\text{even}}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{\text{odd}}(\omega_n^{2k+n}) && (\text{since } \omega_n^{2k+n} = \omega_n^{2k}) \\ &= A(\omega_n^{k+(n/2)})\ . \end{aligned}$$

Thus, the vector y returned by FFT is the DFT of the input vector a.

Factors ω_n^k are ***twiddle factors***, since each factor ω_n^k appears in both its positive and negative forms.

Running time: The **for** loop takes $\Theta(n)$ time. There are two recursive calls, each on an input of size $n/2$.
⇒ Recurrence for the running time is $T(n) = 2T(n/2) + \Theta(n)$.
By case 2 of the master theorem, the running time is $\Theta(n \lg n)$.

Interpolation at the complex roots of unity

For polynomial multiplication, need to convert from coefficient form to point-value form by evaluating the polynomial at the complex roots of unity, pointwise multiply, then convert from point-value form back to coefficient form by interpolating. How to interpolate?

To interpolate, write the DFT as a matrix equation $y = V_n a$, where V_n is a Vandermonde matrix containing the appropriate powers of ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

The (k, j) entry of V_n is ω_n^{kj}, for $j, k = 0, 1, \ldots, n-1$.
The exponents of the entries of V_n form a multiplication table for factors 0 to $n-1$.

For the inverse operation, $a = \text{DFT}_n^{-1}(y)$, multiply y by the matrix V_n^{-1}, the inverse of V_n.

Theorem
For $j, k = 0, 1, \ldots, n-1$, the (j, k) entry of V_n^{-1} is ω_n^{-jk}/n.

Proof Just need to show that $V_n^{-1} V_n = I$ (the $n \times n$ identity matrix).

The (k, k') entry of $V_n^{-1} V_n$ is

$$\begin{aligned} [V_n^{-1} V_n]_{kk'} &= \sum_{j=0}^{n-1} (\omega_n^{-jk}/n)(\omega_n^{jk'}) \\ &= \sum_{j=0}^{n-1} \omega_n^{j(k'-k)}/n . \end{aligned}$$

By the summation lemma, this summation is 0 if $k' \neq k$. Note that if $k' \neq k$, then $-(n-1) \leq k' - k < 0$ or $0 < k' - k \leq n - 1$, so that the summation lemma applies. If $k' = k$, then

$$\sum_{j=0}^{n-1} \omega_n^{j(k'-k)}/n = \sum_{j=0}^{n-1} \omega_n^0/n$$

$$
\begin{aligned}
&= \sum_{j=0}^{n-1} 1/n \\
&= n \cdot (1/n) \\
&= 1 . \qquad \blacksquare
\end{aligned}
$$

With the inverse matrix V_n^{-1} defined, $\text{DFT}_n^{-1}(y)$ is given by

$$
\begin{aligned}
a_j &= \sum_{k=0}^{n-1} y_k \frac{\omega^{-jk}}{n} \\
&= \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}
\end{aligned}
$$

for $j = 0, 1, \ldots, n-1$.

Comparing the equations $y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$ and $a_j = (1/n) \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$, if the FFT algorithm is modified to switch the roles of a and y, ω_n is replaced by ω_n^{-1}, and each element of the result is divided by n, the result is the inverse DFT. Therefore, DFT_n^{-1} is computable in $\Theta(n \lg n)$ time.

By using the FFT and the inverse FFT, it takes just $\Theta(n \lg n)$ time to transform a polynomial of degree-bound n back and forth between its coefficient representation and a point-value representation:

Theorem (Convolution theorem)
For any two vectors a and b of length n, where n is a power of 2,

$$
a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)) ,
$$

where the vectors a and b are padded with 0s to length $2n$ and $\cdot$ denotes the componentwise product of two $2n$-element vectors. ■

FFT circuits

- Because FFT's applications in signal processing require fast speed, often implemented as a circuit in hardware.
- FFT's divide-and-conquer structure allows for a circuit with a parallel structure.
- The circuit ***depth*** (maximum number of computational elements between any output and any input that can reach it) is $\Theta(\lg n)$.

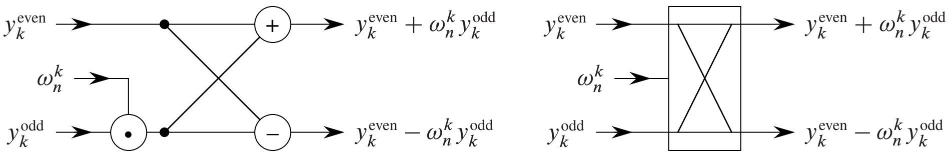
Butterfly operations

In the **for** loop of the FFT procedure, the value $\omega_n^k \, y_k^{\text{odd}}$ is computed twice per iteration, in the lines $y_k = y_k^{\text{even}} + \omega \, y_k^{\text{odd}}$ and $y_{k+(n/2)} = y_k^{\text{even}} - \omega \, y_k^{\text{odd}}$. An optimizing compiler would produce code that evaluates this ***common subexpression*** just once, storing its value into a temporary variable, so that the two lines would be treated like the three lines

$$\begin{aligned} t &= \omega\, y_k^{\text{odd}} \\ y_k &= y_k^{\text{even}} + t \\ y_{k+(n/2)} &= y_k^{\text{even}} - t \end{aligned}$$

This is called the ***butterfly operation***.

The figure below on the left shows the circuit for the butterfly operation, with two input values entering from the left, the twiddle factor ω_n^k multiplied by y_k^{odd}, and the sum and difference output on the right. The right figure shows a simplified drawing of a butterfly operation.


y_k^{even}
ω_n^k
y_k^{odd}
+
−
$y_k^{\text{even}} + \omega_n^k y_k^{\text{odd}}$
$y_k^{\text{even}} - \omega_n^k y_k^{\text{odd}}$
y_k^{even}
ω_n^k
y_k^{odd}
$y_k^{\text{even}} + \omega_n^k y_k^{\text{odd}}$
$y_k^{\text{even}} - \omega_n^k y_k^{\text{odd}}$

(a)

(b)

Recursive circuit structure

Recall that the FFT procedure follows the divide-and-conquer strategy:

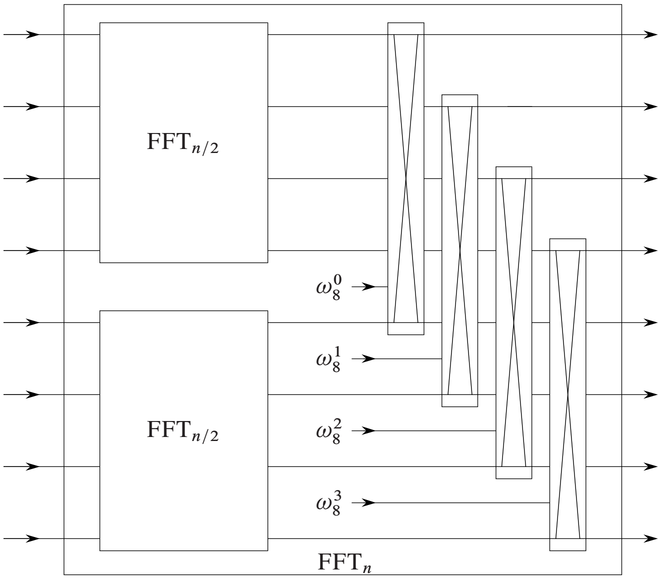
Divide the n-element input vector into its $n/2$ even-indexed and $n/2$ odd-indexed elements.

Conquer by recursively computing the DFTs of the two subproblems, each of size $n/2$.

Combine by performing $n/2$ butterfly operations. These butterfly operations work with twiddle factors $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n/2-1}$.

The circuit below shows the schema for the conquer and combine steps for an FFT circuit FFT_n with n inputs and n outputs.

- The inputs enter from the left, and the outputs exit from the right.
- The input values first go through two $\text{FFT}_{n/2}$ circuits, which are also constructed recursively.
- The values produced by the two $\text{FFT}_{n/2}$ circuits go into $n/2$ butterfly circuits, which combine the results.
- The base case occurs when $n = 1$, where the one output value is equal to the one input value, so an FFT_1 circuit would do nothing.

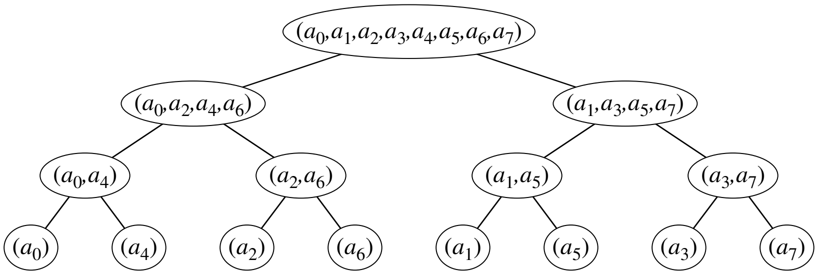


Permuting the inputs

How does the circuit incorporate the divide step?

Idea: Understand how input vectors to the various recursive calls of the FFT procedure relate to the original input vector in order to emulate the divide step at the start for all levels of recursion.

The tree of input vectors to the recursive calls of the FFT procedure for $n = 8$:



Intuition from looking at the tree:

- Arrange the elements of the initial vector a into the order in which they appear in the leaves, then trace the execution of the FFT procedure bottom up instead of top down.
- Take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT.
- The vector then holds $n/2$ 2-element DFTs.
- Next, take these $n/2$ DFTs in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two 2-element DFTs with one 4-element DFT.
- The vector then holds $n/4$ 4-element DFTs.

- Continue in this manner until the vector holds two $(n/2)$-element DFTs, which $n/2$ butterfly operations combine into the final n-element DFT.

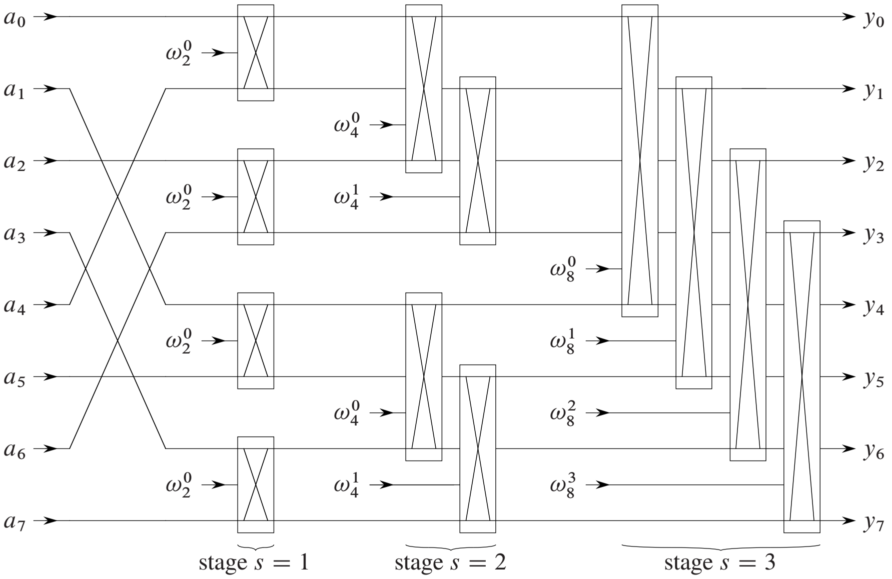
Therefore, can start with the elements of the initial vector a, rearranged as in the leaves of the tree, and then feed them directly into a circuit that follows the schema described in the previous figure.

The order in which the leaves appear in the tree is a ***bit-reversal permutation***: Let $\text{rev}(k)$ be the $\lg n$-bit integer formed by reversing the bits of the binary representation of k. Vector element a_k moves to position $\text{rev}(k)$.

Example: In the above tree, the leaves appear in the order $0, 4, 2, 6, 1, 5, 3, 7$. This sequence in binary is $000, 100, 010, 110, 001, 101, 011, 111$.
Can obtain it by reversing the bits of each number in the sequence $0, 1, 2, 3, 4, 6, 7$ or, in binary, $000, 001, 010, 011, 100, 101, 110, 111$.

Generally, at the top level of the tree, indices whose low-order bit is 0 go into the left subtree and indices whose low-order bit is 1 go into the right subtree.
Strip off the low-order bit at each level, and continue this process down the tree, until reaching the order given by the bit-reversal permutation at the leaves.

The full FFT circuit



The above figure shows the entire FFT circuit for $n = 8$.

- Begins with a bit-reversal permutation of the inputs.
- Then, there are $\lg n$ stages, each consisting of $n/2$ butterflies executed in parallel, which is possible since the butterfly operations at each level of recursion are independent.
- Assuming each butterfly circuit has constant depth, the full circuit has depth $\Theta(\lg n)$.
- For $s = 1, 2, \ldots, \lg n$, stage s consists of $n/2^s$ groups of butterflies, with 2^{s-1} butterflies per group.
- The twiddle factors in stage s are $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, where $m = 2^s$.

Solutions for Chapter 30: Polynomials and the FFT

Solution to Exercise 30.1-1

We are given that $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 6x + 3$. Using the equations (30.1) and (30.2), calculate the coefficients as follows:

- $c_0 = a_0b_0 = -10 \cdot 3 = -30$.
- $c_1 = a_0b_1 + a_1b_0 = -10 \cdot -6 + 1 \cdot 3 = 63$.
- $c_2 = a_0b_2 + a_1b_1 + a_2b_0 = -10 \cdot 0 + 1 \cdot -6 + -1 \cdot 3 = -9$.
- $c_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 = -10 \cdot 8 + 1 \cdot 0 + -1 \cdot -6 + 7 \cdot 3 = -80 + 6 + 21 = -53$.
- $c_4 = a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 = -10 \cdot 0 + 1 \cdot 8 + -1 \cdot 0 + 7 \cdot -6 + 0 \cdot 3 = -34$.
- $c_5 = a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0 = -10 \cdot 0 + 1 \cdot 0 + -1 \cdot 8 + 7 \cdot 0 + 0 \cdot -6 + 0 \cdot 3 = -8$.
- $c_6 = a_0b_6 + a_1b_5 + a_2b_4 + a_3b_3 + a_4b_2 + a_5b_1 + a_6b_0 = -10 \cdot 0 + 1 \cdot 0 + -1 \cdot 0 + 7 \cdot 8 + 0 \cdot 0 + 0 \cdot -6 + 0 \cdot 3 = 56$.

Therefore, the product $C(x)$ is $56x^6 - 8x^5 - 34x^4 - 53x^3 - 9x^2 + 63x - 30$.

Solution to Exercise 30.1-4

Assume for sake of contradiction that we have a set S of $n - 1$ point-value pairs that specify a unique polynomial of degree-bound n.

By Theorem 30.1, S specifies a unique polynomial $A(x)$ of degree-bound $n - 1$.

Choose a point x' not equal to the x value in any of the $n - 1$ point-value pairs in S. Let $y' = A(x')$, and $S' = S \cup \{(x', y')\}$. By Theorem 30.1, S' specifies a unique polynomial $A'(x)$ of degree-bound n.

Now, choose $y'' \neq y'$. Let $S'' = S \cup \{(x', y'')\}$. Again, by Theorem 30.1, S'' specifies a unique polynomial $A''(x)$ of degree-bound n.

We have $A(x') = A'(x') \neq A''(x')$. Therefore, the $n - 1$ point-value pairs in S fail to specify a unique polynomial of degree-bound n.

Solution to Exercise 30.1-7

Construct a polynomial $A(x) = \sum_{j=0}^{10n-1} a_j x^j$ of degree-bound $10n$, where $a_j = 1$ if $j \in A$, otherwise $a_j = 0$.

Construct $B(x)$ in the same way: $B(x) = \sum_{j=0}^{10n-1} b_j x^j$, where $b_j = 1$ if $j \in B$, otherwise $b_j = 0$.

Multiply $A(x)$ and $B(x)$ to get the product $C(x)$ in time $O(n \lg n)$, using the method described in Section 30.1.

To find the elements of C, look at each term $c_j x^j$ in $C(x)$. If c_j is positive, then $j \in C$. The number of times each element of C is realized as a sum of elements in A and B is c_j, since the number of multiplications that led to the coefficient c_j capture every possible way that elements in A and B could sum up to an element in C.

Solution to Exercise 30.2-1

By the cancellation lemma,

$$\begin{aligned}
\omega_n^{n/2} &= \omega_{2n}^{2n/2} \\
&= \omega_{2n}^{n} \\
&= \omega_2 \\
&= e^{2\pi i/2} \\
&= e^{\pi i} \\
&= -1 .
\end{aligned}$$

Solution to Exercise 30.2-2

Computing each term y_k, we have

- $y_0 = \sum_{j=0}^{3} a_j \omega_4^{0j} = 0 + 1 + 2 + 3 = 6.$
- $y_1 = \sum_{j=0}^{3} a_j \omega_4^{1j} = 0 + 1 \cdot w_4^1 + 2 \cdot w_4^2 + 3 \cdot w_4^3 = 1i + 2 - 3i = -2i + 2.$
- $y_2 = \sum_{j=0}^{3} a_j \omega_4^{2j} = 0 + 1 \cdot w_4^2 + 2 \cdot w_4^4 + 3 \cdot w_4^6 = 1(-1) + 2(1) + 3(-1) = -2.$
- $y_3 = \sum_{j=0}^{3} a_j \omega_4^{3j} = 0 + 1 \cdot w_4^3 + 2 \cdot w_4^6 + 3 \cdot w_4^9 = -1i + 2(-1) + 3i = 2i - 2.$

Therefore, the DFT is $(6, -2i + 2, -2, 2i - 2)$.

Solution to Exercise 30.2-7

From the hint, we know that the polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$. Therefore, let $P(x) = (x - z_0)(x - z_1) \cdots (x - z_{n-1})$.

In order to perform the multiplications in $O(n \lg^2 n)$ time, use a divide-and-conquer approach.

At each recursive step, divide the input in the form of $(x - z_0)(x - z_1) \cdots (x - z_{n-1})$ in half and recursively multiply each half. To combine the two halves, multiply them in $\Theta(n \lg n)$ time using the FFT multiplication method. The output will be a polynomial in coefficient form.

The recurrence for the running time of this approach is $T(n) = 2T(n/2) + \Theta(n \lg n)$, which is $O(n \lg^2 n)$ by case 2 of the master theorem.

Solution to Exercise 30.3-3

M is a $b \times b$ matrix with 1s on the antidiagonal and 0s everywhere else:

$$\begin{pmatrix} 0 & \cdots & 0 & 0 & 1 \\ 0 & \cdots & 0 & 1 & 0 \\ 0 & \cdots & 1 & 0 & 0 \\ \vdots & \iddots & \vdots & \vdots & \vdots \\ 1 & \cdots & 0 & 0 & 0 \end{pmatrix}$$

Solution to Exercise 30.3-4

```
BIT-REVERSE-PERMUTATION($a, n$)
  $b = \lg n$
  for $j = 0$ to $n - 1$
      $r =$ BIT-REVERSE-OF($j, b$)
      if $j < r$
          exchange $a_j$ with $a_r$
```

The test for $j < r$ ensures that we don't exchange a_j with a_r and then later exchange them again, which would nullify the original exchange.

Lecture Notes for Chapter 32: String Matching

Chapter 32 overview

[These notes cover Sections 32.1–32.3 and 32.5, but not the Knuth-Morris-Pratt algorithm in Section 32.4. The treatment is lighter than in Introduction to Algorithms*. The discussion of string matching with finite automata is adapted from* Algorithms Unlocked*.]*

Given a ***pattern*** and a ***text***, we want to find all occurrences of the pattern in the text. Many applications, including searching for text in a document, finding web pages relevant to queries, and searching for patterns in DNA sequences.

Formally:

Input: The text is an array $T[1:n]$, and the pattern is an array $P[1:m]$, where $m \le n$. The elements of P and T are drawn from a finite ***alphabet*** Σ. Examples: $\Sigma = \{0, 1\}$, Σ is the ASCII characters, or $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ for DNA matching. Call the elements of P and T ***characters***.

Output: All amounts that we have to shift P to match it with characters of T. Say that P ***occurs with shift s in T*** if $0 \le s \le n-m$ and $T[s+1:s+m] = P[1:m]$. (Need $s \le n - m$ so that the pattern doesn't run off the end of the text.) If P occurs with shift s in T, then s is a ***valid shift***; otherwise, s is an ***invalid shift***. We want to find all valid shifts.

Example: $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, $T = \texttt{GTAACAGTAAACG}$, $P = \texttt{AAC}$. *[Leave this example on the board.]* Then P occurs in T with shifts 2 and 9.

How long does it take to check whether P occurs in T with a given shift amount? Need to check each character of P against the corresponding position in T, taking $\Theta(m)$ time in the worst case. (Assumes that checking each character takes constant time.) Can stop once we find a mismatch, so matching time is $O(m)$ in any case.

Some string-matching algorithms require preprocessing, which we'll account for separately from the matching time.

Naive string-matching algorithm

Just try each shift.

```
NAIVE-STRING-MATCHER$(T, P, n, m)$
  for $s = 0$ to $n - m$
      if $P[1:m] == T[s+1:s+m]$
          print “Pattern occurs with shift” $s$
```

Time: Tries $n-m+1$ shift amounts, each taking $O(m)$ time, so $O((n-m+1)m)$. This bound is tight in the worst case, such as when the text is n `A`s and the pattern is m `A`s. No preprocessing needed.

This algorithm is not efficient. It throws away valuable information. Example: For T and P above, when we look at shift amount $s = 2$, we see all the characters in $T[3:5]$ = `AAC`. But at the next shift, for $s = 3$, we look at $T[4]$ and $T[5]$ again. Since we've already seen these characters, we'd like to avoid having to look at them again.

Rabin-Karp algorithm

Based on two ideas:

1. Represent the pattern P and each length-m substring of T as an integer (that fits in one computer word). Like a hash value, but much simpler. If the integers representing the pattern and the substring of T are equal, we (might) have a match. *[We'll first see a simple version of Rabin-Karp in which if the integers are equal, then there definitely is a match. Then we'll see a more realistic version that can have false positives.]*
2. Slide along T, looking in succession at the length-m substrings $T[1:m]$, $T[2:m+1]$, $T[3:m+2], \ldots, T[n-m+1:n]$. Design the integer representation so that as we slide along T, we can update the integer value for shift $s+1$ in constant time, given the integer value for shift s.

To keep things simple at first, suppose that Σ is the digits 0 to 9. Then view a string of k characters as a k-digit decimal number, so that the string `31415` is represented by 31,415. So a substring of the text matches the pattern if and only if they have the same integer represenation. *[Here, we're assuming not only that the alphabet is the digits* 0 *to* 9*, but also that we can fit the integer representation into a single computer word, or a constant number of words. We'll see later how to get around both assumptions.]*

We need to compute the integer representation p for the pattern P, which won't change, and the integer representation for each length-m substring of the text T, which will change for each shift. For shift s, denote by t_s the integer representation for $T[s+1:s+m]$. If $p = t_s$, then pattern P occurs in text T with shift s.

If we treat each character as its corresponding integer value, we can compute p and t_0 in $\Theta(m)$ time using Horner's rule:

```
COMPUTE-REP(S, m)
  r = 0
  for i = 1 to m
      r = 10 · r + S[i]
  return r
```

$p = \text{COMPUTE-REP}(P, m)$
$t_0 = \text{COMPUTE-REP}(T, m)$

Given t_s, compute t_{s+1} by subtracting out the value contributed by $T[s+1]$, multiplying what's left by 10, and adding in the value contributed by $T[s+m+1]$. In t_s, $T[s+1]$ contributes $T[s+1] \cdot 10^{m-1}$. In t_{s+1}, $T[s+m+1]$ contributes just itself. Therefore,

$$t_{s+1} = 10 \cdot (t_s - T[s+1] \cdot 10^{m-1}) + T[s+m+1] ,$$

which takes $\Theta(1)$ time.

Example: Suppose that T begins with `314159` and $m = 5$. Then $t_0 = 31{,}415$ and

$$\begin{aligned} t_1 &= 10 \cdot (t_0 - T[1] \cdot 10^4) + T[6] \\ &= 10 \cdot (31{,}415 - 3 \cdot 10{,}000) + 9 \\ &= 14{,}159 . \end{aligned}$$

So—assuming that the characters are the digits 0 to 9 and the integer representation of a length-m string can fit in a word—the preprocessing time is $\Theta(m)$ (to compute the integer representation of the pattern and the first m text characters), and the matching time is $\Theta(n - m + 1)$ (the number of shifts).

What if the alphabet has a wider range? What if m is large? Either of these conditions could cause the integer representation of length-m string to exceed the capacity of a computer word.

If the alphabet has size d, then map each character to a unique integer in the set $\{0, 1, \ldots, d-1\}$ and replace 10 in the previous equations by d. If the integer representation doesn't fit in a computer word, then do something like hashing: take everything modulo q, for some large q such that dq just fits within one word.

```
COMPUTE-REP-MOD-Q(S, m, d, q)
  r = 0
  for i = 1 to m
      r = (d · r + S[i]) mod q
  return r
```

$p = \text{COMPUTE-REP-MOD-Q}(P, m, d, q)$
$t_0 = \text{COMPUTE-REP-MOD-Q}(T, m, d, q)$

Then

$t_{s+1} = (d \cdot (t_s - T[s+1] \cdot h) + T[s+m+1]) \bmod q$,

where $h = d^{m-1} \bmod q$.

Problem: Could get a ***spurious hit*** (a false positive): $p = t_s$ but $P[1:m] \neq T[s+1:s+m]$. Why? Because the integer representations of two different numbers could have the same value when taken modulo q.

Solution: When $p = t_s$, have to check to see whether $P[1:m] = T[s+1:s+m]$.

```
RABIN-KARP(T, P, n, m, d, q)
  $h = d^{m-1} \bmod q$
  $p = 0$
  $t_0 = 0$
  $p$ = COMPUTE-REP-MOD-Q($P, m, d, q$)    // preprocessing
  $t_0$ = COMPUTE-REP-MOD-Q($T, m, d, q$)    // preprocessing
  for $s = 0$ to $n - m$                     // matching—try all possible shifts
      if $p == t_s$                          // a hit?
          if $P[1:m] == T[s+1:s+m]$          // valid shift?
              print "Pattern occurs with shift" $s$
      if $s < n - m$
          $t_{s+1} = \big(d(t_s - T[s+1]h) + T[s+m+1]\big) \bmod q$
```

Takes $O(m)$ time whenever $p = t_s$. (Using O instead of Θ because we might not need to look at all m characters to discover a mismatch.) Because we could have $p = t_s$ for all shift amounts, worst-case matching time is $\Theta((n-m+1)m)$: no better than the naive method. (Using Θ instead of O because we might have to check all m characters each time.)

If v valid shifts, matching time is $O((n-m+1)+vm) = O(n+m)$, plus time for processing spurious hits. But expect that in realistic cases, spurious hits are rare. Think of reducing values modulo q as like a random mapping from Σ^* to $\{0, 1, \ldots, q-1\}$. Expected number of spurious hits is then $O(n/q)$ because probability that a random t_s is equivalent to p, modulo q, is about $1/q$. Then the expected matching time for RABIN-KARP is $O(m)+O(m(v+n/q))$. If $v = O(1)$ and $q \geq m$, this works out to $O(n)$, since $m \leq n$.

String matching with finite automata

Efficient: examines each text character exactly one time. Preprocessing time will be $\Theta(m^3 |\Sigma|)$ but matching time will be just $\Theta(n)$.

A ***finite automaton (FA)*** is a set of states and a way to go from state to state based on a sequence of input characters. An FA starts in a given state and consumes characters one at a time. Based on the state it's in and the character just consumed, it moves to a new state.

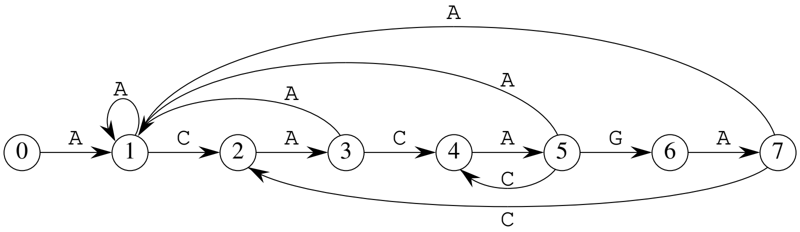
Formally: A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function $Q \times \Sigma \to Q$, called the ***transition function*** of M.

The FA begins in state q_0 and reads the characters of the input string one at a time. When in state q and reading character a, it moves to state $\delta(q, a)$. If the current state is in A, then the FA has ***accepted*** the string read so far. An input that is not accepted is ***rejected***.

For string matching, the FA has $m + 1$ states, numbered 0 to m. The FA starts in state 0. When it's in state k, the k most recent text characters read match the first k characters of the pattern. When the FA gets to state m, it has found a match.

Example: Use the alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ for DNA sequencing. If $P =$ ACACAGA with $m = 7$, then the FA is *[leave this figure on the board]*



The horizontal spine has edges labeled with P. Whenever P occurs in the text, the FA moves right, to the next state along the spine. When it reaches the rightmost state, it has found a match.

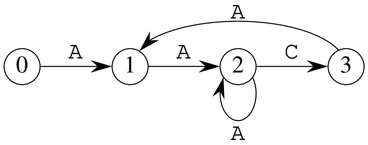
The transition function δ is defined for all states $q \in Q$ and all characters $a \in \Sigma$. Missing arrows are assumed to be transitions to state 0.

When a character from the text does not make progress toward a match, the FA moves to a lower-numbered state or stays in the same state. The only arrows pointing to the right are along the spine.

We'll see how to compute the transition function δ later. Assuming that we have it, here's how to perform string matching.

```
FA-MATCHER($T, \delta, n, m$)
  $q = 0$
  for $i = 1$ to $n$
      $q = \delta(q, T[i])$
      if $q == m$
          print "Pattern occurs with shift" $i - m$
```

Example: P = AAC, T = GTAACAGTAAACG. FA is



Sequence of states, input characters, and output shifts:

i		1	2	3	4	5	6	7	8	9	10	11	12	13
character		G	T	A	A	C	A	G	T	A	A	A	C	G
state	0	0	0	1	2	3	1	0	0	1	2	2	3	0
output shift						2							9	

The matching time is just $\Theta(n)$.

How do we build the finite automaton? We already know that it has states 0 through m, the start state is 0, and the only accepting state is m. The hard part is the transition function δ. The idea:

> When the FA is in state k, the k most recent characters read from the text are the first k characters in the pattern.

For the FA for pattern ACACAGA, why is $\delta(5, \text{C}) = 4$? If the FA gets to state 5, then the 5 most recent characters read are ACACA. (See the spine of the FA.) If the next character read is C, then it doesn't match, so the FA cannot go to state 6. But it doesn't have to go back to state 0, either, because now the 4 most recently read characters are ACAC: the first 4 characters of the pattern. So the FA moves to state 4.

A ***prefix*** $P[:i]$ of a string P is a substring consisting of the first i characters of P. *[Students who are used to programming in Python think of $P[:i]$ as including characters indexed* 0 *to* $i - 1$*. Here,* $P[:i] = P[1:i]$*, so that it includes* $P[i]$*.]* A ***suffix*** is a substring consisting of characters from the end. For a string X and a character a, denote concatenating a onto X by Xa.

So, in state k, we've most recently read $P[:k]$ in the text. We look at the next character a of the text, so now we've read $P[:k]a$. How long a prefix of P have we just read?

> Find the longest prefix of P that is also a suffix of $P[:k]a$. Then $\delta(k, a)$ should be the length of this longest prefix.

In the example: How to determine that $\delta(5, \text{C}) = 4$?

- Take the prefix $P[:5]$ = ACACA.
- Concatenate C, giving ACACAC.
- Want the longest prefix of the pattern ACACAGA that is also a suffix of ACACAC.
- Because the length of ACACAC is 6 and the suffix can't be longer than that, start by looking at $P[:6]$. Work our way down to shorter and shorter prefixes until we find a prefix that is also a suffix of ACACAC.
- $P[:6]$ is ACACAG. Not a suffix of ACACAC.

- $P[:5]$ is ACACA. Not a suffix of ACACAC.
- $P[:4]$ is ACAC. This is a suffix of ACACAC, so stop and set $\delta(5, \texttt{C}) = 4$.

We're guaranteed to stop, because $P[:0]$ is the empty string, and the empty string is a suffix of any string. So if we don't find a suffix of $P[:k]a$ from among $P[:k]$, $P[:k-1]$, $P[:k-2], \ldots, P[:1]$, then set $\delta(k, a) = 0$.

```
COMPUTE-TRANSITION-FUNCTION($P, \Sigma, m$)
  for $q = 0$ to $m$
      for each character $a \in \Sigma$
          $k = \min\{m, q + 1\}$
          while $P[:k]$ is not a suffix of $P[:q]a$
              $k = k - 1$
          $\delta(q, a) = k$
  return $\delta$
```

The maximum value of k is $\min\{m, q + 1\}$. If we set $\delta(q, a) = q + 1$, then $P[:q + 1]$ is a suffix of $P[:q]a$, and so we have found the next character in the pattern. We cap k at m, since we can't go to a state numbered higher than m.

Preprocessing time:

- The outermost **for** loop iterates $m + 1$ times.
- The middle loop iterates $|\Sigma|$ times.
- The innermost **while** loop iterates at most $m + 1$ times.
- Each suffix check in the **while** loop test examines at most m characters of the pattern (since $k \leq m$). Therefore, each suffix check takes $O(m)$ time.
- Total preprocessing time: $O(m^3 |\Sigma|)$.

Therefore, with a finite automaton, we can perform string matching with preprocessing time $O(m^3 |\Sigma|)$ and matching time $\Theta(n)$. It's possible to get the preprocessing time down to $O(m |\Sigma|)$.

Section 32.4 presents the Knuth-Morris-Pratt (KMP) algorithm, which maintains the $\Theta(n)$ matching time of using an FA but avoids computing the transition function δ. It gets the preprocessing time down to $\Theta(m)$.

Suffix arrays

Different goal from the string-matching algorithms in the preceding sections. Can use a suffix array to find all occurrences of a pattern in a text, but that's not the most efficient way. But can use suffix arrays to solve additional problems.

Definition of a suffix array

- For a text $T[1:n]$, denote the suffix $T[i:n]$ by $T[i:]$.
- Suppose that you sort all n suffixes lexicographically and that $T[j:]$ is the ith suffix in the sorted order.
- Then, denoting the suffix array for T by $SA[1:n]$, have $SA[i] = j$.

Example: Using the text $T =$ `ratatat` *[same example as in the textbook]* , have the suffixes

j	$T[j:]$
1	`ratatat`
2	`atatat`
3	`tatat`
4	`atat`
5	`tat`
6	`at`
7	`t`

so that

i	$SA[i]$	$rank[i]$	$LCP[i]$	suffix $T[SA[i]:]$
1	6	4	0	`at`
2	4	3	2	`atat`
3	2	7	4	`atatat`
4	1	2	0	`ratatat`
5	7	6	0	`t`
6	5	1	1	`tat`
7	3	5	3	`tatat`

[Omit the rank and LCP columns for now, and add them in later.]

How to search for pattern with a suffix array

- All occurrences of the pattern appear in consecutive entries of the suffix array.
- Find the length-m pattern using binary search on the suffix array.
  Time: $O(m \lg n)$. Factor of m is to compare suffixes with the pattern.
- If the pattern is found, search backward and forward from that spot until finding a suffix that does not begin with the pattern. (Or run off the end of the suffix array.)
- ***Time:*** If k occurrences, $O(m \lg n + km)$.

LCP array

LCP = longest common prefix

$LCP[i]$ equals the length of the longest common prefix between the ith and $(i-1)$st suffixes in the sorted order ($T[SA[i]:]$ and $T[SA[i-1]:]$).

$LCP[SA[1]] = 0$, since no suffix precedes $T[SA[1]:]$ lexicographically.

[Now add in the LCP column in the `ratatat` *example.]*

How to find a longest repeated substring

Note: Not guaranteed to be unique.

Find the maximum value in the LCP array.

If $LCP[i]$ contains the maximum value, then $T[SA[i]:SA[i] + LCP[i] - 1]$ is a longest repeated substring.

Example: For `ratatat`, maximum *LCP* value is 4, appearing in *LCP*[3].
Longest repeated substring is $T[SA[3] : SA[3] + LCP[3] - 1] = T[2:5] =$ `atat`.

Computing the suffix array

Will see an $O(n \lg n)$-time algorithm. *[Problem 32-2 walks through a $\Theta(n)$-time algorithm.]*

Idea: Make several passes over the text, where each pass lexicographically sort substrings whose length is twice the length in the previous pass.
By the $\lceil \lg n \rceil$th pass, sorting all the suffixes.
⇒ Have the information needed to construct the suffix array.

Since $\lceil \lg n \rceil$ passes, if each pass used an $O(n \lg n)$-time sorting algorithm, total time would be $O(n \lg^2 n)$.
Instead, only the first pass uses an $O(n \lg n)$-time sort.
All others use radix sort, running in $\Theta(n)$ time per pass.

Observation: Consider strings $s_1 = s_1' \, s_1''$ (concatenation) and $s_2 = s_2' \, s_2''$.
Suppose $s_1' < s_2'$ (lexicographically).
Then, regardless of s_1'' and s_2'', must have $s_1 < s_2$.

Idea: Don't represent substrings directly. Instead use integer ***ranks***.
$s_1 < s_2$ if and only if rank of $s_1 <$ rank of s_2.
Identical substrings have equal ranks.

Initial substrings are just one character long. For their ranks, use their character codes (such as ASCII or Unicode).
Assume that there is a function ord that maps a character to its character code.

Substrings longer than one character will have positive ranks $\leq n$.
Empty substring always has rank 0.

Use objects internally to keep track of substrings and ranks: Create and sort array *substr-rank*$[1:n]$ of objects with attributes

- *left-rank*: rank of the left part of the substring,
- *right-rank*: rank of the right part of the substring,
- *index*: index into the text T of where the substring starts.

Will show pseudocode and then walk through an example.

```
COMPUTE-SUFFIX-ARRAY($T, n$)
  allocate arrays $substr\text{-}rank[1:n]$, $rank[1:n]$, and $SA[1:n]$
  for $i = 1$ to $n$
      $substr\text{-}rank[i].left\text{-}rank = \text{ord}(T[i])$
      if $i < n$
          $substr\text{-}rank[i].right\text{-}rank = \text{ord}(T[i+1])$
      else $substr\text{-}rank[i].right\text{-}rank = 0$
      $substr\text{-}rank[i].index = i$
  sort the array substr-rank into monotonically increasing order based
      on the left-rank attributes, using the right-rank attributes to break ties;
      if still a tie, the order does not matter
  $l = 2$
  while $l < n$
      MAKE-RANKS($substr\text{-}rank, rank, n$)
      for $i = 1$ to $n$
          $substr\text{-}rank[i].left\text{-}rank = rank[i]$
          if $i + l \le n$
              $substr\text{-}rank[i].right\text{-}rank = rank[i+l]$
          else $substr\text{-}rank[i].right\text{-}rank = 0$
          $substr\text{-}rank[i].index = i$
      sort the array substr-rank into monotonically increasing order based
          on the left-rank attributes, using the right-rank attributes
          to break ties; if still a tie, the order does not matter
      $l = 2l$
  for $i = 1$ to $n$
      $SA[i] = substr\text{-}rank[i].index$
  return $SA$
```

```
MAKE-RANKS($substr\text{-}rank, rank, n$)
  $r = 1$
  $rank[substr\text{-}rank[1].index] = r$
  for $i = 2$ to $n$
      if $substr\text{-}rank[i].left\text{-}rank \ne substr\text{-}rank[i-1].left\text{-}rank$
              or $substr\text{-}rank[i].right\text{-}rank \ne substr\text{-}rank[i-1].right\text{-}rank$
          $r = r + 1$
      $rank[substr\text{-}rank[i].index] = r$
```

For `ratatat`, and using ASCII codes for the ord function, here is the *substr-rank* array after the first **for** loop, for substrings of length 2:

i	*left-rank*	*right-rank*	*index*	substring
1	114	97	1	`ra`
2	97	116	2	`at`
3	116	97	3	`ta`
4	97	116	4	`at`
5	116	97	5	`ta`
6	97	116	6	`at`
7	116	0	7	`t`

After the first sorting step:

i	*left-rank*	*right-rank*	*index*	substring
1	97	116	2	`at`
2	97	116	4	`at`
3	97	116	6	`at`
4	114	97	1	`ra`
5	116	0	7	`t`
6	116	97	3	`ta`
7	116	97	5	`ta`

Going into each iteration of the **while** loop, variable l gives an upper bound on lengths of substrings sorted so far. Once $l \geq n$, can stop. Some substrings might have fewer than l characters, if they cut off at the end of the text.

In each iteration, MAKE-RANKS gives each substring its rank in the sorted order. Each rank is an integer from 1 up to the number of unique length-l substrings. MAKE-RANKS starts the ranks at $r = 1$ and increments r each time it sees a new substring.

Within the **while** loop, the **for** loop revises the *substr-rank* array:

- *substr-rank*$[i]$.*left-rank* gets the value of *rank*$[i]$, which is the rank of the substring $T[i : i + l - 1]$.
- *substr-rank*$[i]$.*right-rank* gets the value of *rank*$[i + l]$, which is the rank of the substring $T[i + l : i + 2l - 1]$. But if $i + l > n$, then there is no such substring, and *substr-rank*$[i]$.*right-rank* gets 0. Note also that it's possible that $i + 2l - 1 > n$, in which case the rank is for $T[i + l : n]$.
- *substr-rank*$[i]$.*index* gets the value i.

Then the *substr-rank* array is sorted, based on the *left-rank* attribute and breaking ties by the *right-rank* attribute.
After sorting, the length of sorted substrings has doubled.

Best shown by continuing the example for `ratatat`.
For the **while** loop iteration with $l = 2$:

After calling MAKE-RANKS		After the **for** loop					After sorting				
i	*rank*	i	*left-rank*	*right-rank*	*index*	substring	i	*left-rank*	*right-rank*	*index*	substring
1	2	1	2	4	1	`rata`	1	1	0	6	`at`
2	1	2	1	1	2	`atat`	2	1	1	2	`atat`
3	4	3	4	4	3	`tata`	3	1	1	4	`atat`
4	1	4	1	1	4	`atat`	4	2	4	1	`rata`
5	4	5	4	3	5	`tat`	5	3	0	7	`t`
6	1	6	1	0	6	`at`	6	4	3	5	`tat`
7	3	7	3	0	7	`t`	7	4	4	3	`tata`

To understand what MAKE-RANKS does:

- The sorted substrings of length at most 2 are

  1. `at`, appearing at indices 2, 4, 6,
  2. `ra`, appearing at index 1,
  3. `t`, appearing at index 7,
  4. `ta`, appearing at indices 3, 5.

- So *rank*[2] = *rank*[4] = *rank*[6] = 1, *rank*[1] = 2, *rank*[7] = 3, and *rank*[3] = *rank*[5] = 4.

To understand what the **for** loop does:

- Concatenating adjacent substrings of length at most 2, it creates substrings of length at most 4.
- *substr-rank*$[i]$ = *rank*$[i]$, the rank of the left substring.
- *substr-rank*$[i]$ = *rank*$[i + 2]$, the rank of the right substring, unless $i + 2 > n$, in which case it's 0.

To understand what the sorting step does:

- Sorts the substrings of length at most 4.
- Sorts based first on *left-rank*. If two substrings of length at most 2 have different values in *left-rank*, the lower one comes first.
- If tied in *left-rank*, sorts based on *right-rank*. If two substrings of length at most 2 are tied in *left-rank* but have different values in *right-rank*, the one with lower *right-rank* comes first.
- If tied in both *left-rank* and *right-rank*, then either substring can come first.

In general, change 2 to l, and change 4 to $2l$. Doubling l at the end of each **while** loop iteration sets up for the next iteration.
For $l = 4$:

After calling MAKE-RANKS

i	*rank*
1	3
2	2
3	6
4	2
5	5
6	1
7	4

After the **for** loop

i	*left-rank*	*right-rank*	*index*	substring
1	3	5	1	`ratatat`
2	2	1	2	`atatat`
3	6	4	3	`tatat`
4	2	0	4	`atat`
5	5	0	5	`tat`
6	1	0	6	`at`
7	4	0	7	`t`

After sorting

i	*left-rank*	*right-rank*	*index*	substring
1	1	0	6	`at`
2	2	0	4	`atat`
3	2	1	2	`atatat`
4	3	5	1	`ratatat`
5	4	0	7	`t`
6	5	0	5	`tat`
7	6	4	3	`tatat`

Once the length of the longest sorted substring (i.e., the entire text) is $\geq n$, can stop.

At that point, the value of *substr-rank*$[i]$.*index* gives *SA*$[i]$.

Running time

- First **for** loop: $\Theta(n)$.
- First sorting step: $O(n \lg n)$, using heapsort or merge sort.
- **while** loop makes $\lceil \lg n \rceil - 1$ iterations.
- Each call of MAKE-RANKS: $\Theta(n)$.
- Inner **for** loop: $\Theta(n)$.
- Inner sorting step: $O(n \lg n)$.
- Total **while** loop time: $O(n \lg^2 n)$.
- Final **for** loop: $\Theta(n)$.
- Total time: $O(n \lg^2 n)$.

Observation: Values of *left-rank* and *right-rank* being sorted within the **while** loop are always integers in the range 0 to n.

- Can use radix sort by first sorting based on *right-rank* and then sorting based on *left-rank*.
- Reduces sorting time within the **while** loop to $\Theta(n)$.
- Which reduces total **while** loop time to $O(n \lg n)$.
- So that total time is $O(n \lg n)$.

For certain inputs, can stop the **while** loop early (Exercise 32.5-2).

Computing the *LCP* array

Use array *rank* that is the inverse of the *SA* array, like the final *rank* array in COMPUTE-SUFFIX-ARRAY.
$SA[i] = j \Rightarrow rank[j] = i$, so that $rank[SA[i]] = i$.

$rank[i]$ gives the position of $T[i:]$ in the final sorted order of suffixes.

[Now go back and fill in the rank values in the first example.]

To compute the *LCP* array, need to determine where in the sorted order a suffix appears, but with first character removed.

- Use the *rank* array.
- Consider the ith small suffix, $T[SA[i]:]$.
- Dropping the first character gives $T[SA[i]+1:]$: the suffix starting at index $SA[i]$ in the text.
- Location of this suffix in the sorted order is given by $rank[SA[i]+1]$.
- ***Example:*** For suffix `atat`, starting at index 4 in the text, where is `tat`, starting at index 5, in the sorted order?
  - `atat` appears in position 2 of *SA*, so that $SA[2] = 4$.
  - $rank[SA[2]+1] = rank[4+1] = rank[5] = 6$.
  - And `tat` appears at position 6 in the sorted order: $SA[6] = 5$.

Lemma

Consider suffixes $T[i-1:]$ and $T[i:]$, with ranks $rank[i-1]$ and $rank[i]$, respectively. If $LCP[rank[i-1]] = l > 1$, then $T[i:]$ ($T[i-1:]$ with first character removed) has $LCP[rank[i]] \geq l-1$.

[Proof omitted.]

```
COMPUTE-LCP($T, SA, n$)
  allocate arrays $rank[1:n]$ and $LCP[1:n]$
  for $i = 1$ to $n$
      $rank[SA[i]] = i$               // by definition
  $LCP[1] = 0$                        // also by definition
  $l = 0$                             // initialize length of LCP
  for $i = 1$ to $n$
      if $rank[i] > 1$
          $j = SA[rank[i] - 1]$       // $T[j:]$ precedes $T[i:]$ lexicographically
          $m = \max\{i, j\}$
          while $m + l \le n$ and $T[i + l] == T[j + l]$
              $l = l + 1$             // next character is in common prefix
          $LCP[rank[i]] = l$          // length of LCP of $T[j:]$ and $T[i:]$
          if $l > 0$
              $l = l - 1$             // peel off first character of common prefix
  return $LCP$
```

First **for** loop fills in the *rank* array per its definition. Then set $LCP[1] = 0$, per definition.

The main **for** loop fills in the rest of the *LCP* array, going by decreasing-length suffixes: fills in going in order of $rank[1], rank[2], \ldots, rank[n]$:

- When considering a suffix $T[i:]$, determine the suffix $T[j:]$ immediately preceding $T[i:]$ in the sorted order.
- LCP of $T[j:]$ and $T[i:]$ has length $\ge l$:
  - True in first iteration of the **for** loop, when $l = 0$.
  - If $LCP[rank[i]]$ is set correctly, then decrementing l and the lemma maintain this property for the next iteration.
- LCP of $T[j:]$ and $T[i:]$ could be $> l$, however.
- The **while** loop increments l for each additional character they have in common.
- Index m is used to ensure that the test $T[i + l] == T[j + l]$ doesn't run off the end of the text.
- When the **while** loop terminates, l is the length of the LCP of $T[j:]$ and $T[i:]$. It then gets filled into the *LCP* array.

Running time

Use aggregate analysis.

- Each of the **for** loops iterates n times, so just need to bound total number of iterations of the **while** loop.
- Each iteration of the **while** loop increases l by 1.
- Test $m + l \le n$ ensures that $l \le n$.
- Initial value of l is 0, and l decreases $\le n - 1$ times in the last line of the **for** loop $\Rightarrow$ l is incremented $< 2n$ times.
- Total time is $\Theta(n)$.

Solutions for Chapter 32: String Matching

Solution to Exercise 32.1-2

As soon as a position j is found such that $P[j] \neq T[s+j]$, set $s = s + j + 1$. Since we assume that all characters in P are unique, the character $P[1]$ does not occur within $T[s+2:s+j]$. The next possible location in T where $P[1]$ could occur is $T[s+j+1]$.

The running time becomes $O(n)$ because each character in T is examined at most once.

Solution to Exercise 32.1-3

For any given shift amount, the ith character of the pattern is compared—that is, at least i characters are compared—only if the first $i-1$ characters match the text. The probability that a pattern character matches a text character is $1/d$, and so the probability that the first $i-1$ characters match is $1/d^{i-1}$.

For a given shift amount, let X be the random variable equaling how many characters of the pattern are compared. We have

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} && \text{(by equation (C.28))} \\
&= \sum_{i=1}^{m} \Pr\{X \geq i\} && \text{(because } \Pr\{X = i\} = 0 \text{ for } i > m\text{)} \\
&= \sum_{i=1}^{m} \frac{1}{d^{i-1}} \\
&= \sum_{i=0}^{m-1} \left(\frac{1}{d}\right)^{i} \\
&= \frac{(1/d)^m - 1}{(1/d) - 1} && \text{(by equation (A.6))} \\
&= \frac{1 - d^{-m}}{1 - d^{-1}} .
\end{aligned}$$

By linearity of expectation and summing over all $n - m + 1$ possible shifts, we get that the expected number of character comparisons equals

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} .$$

Because $d \geq 2$, we have

$$\begin{aligned}\frac{1 - d^{-m}}{1 - d^{-1}} &\leq \frac{1 - d^{-m}}{1 - 1/2} \\ &= 2(1 - d^{-m}) \\ &< 2 ,\end{aligned}$$

and so

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} < 2(n - m + 1) .$$

Solution to Exercise 32.1-4

Decompose the pattern P as $P_1 \diamondsuit P_2 \diamondsuit P_3 \diamondsuit \ldots \diamondsuit P_k$. Look for the first occurrence of P_1 in the text. If found, start looking for the first occurrence of P_2 after the end of the first occurrence of P_1. If P_2 is found, start looking for the first occurrence of P_3 after the end of the first occurrence of P_2, and so on.

Denote the length of subpattern P_j by l_j, so that $l_1 + l_2 + \cdots + l_k = m$. Then the running time is

$$O\left(\sum_{j=1}^{k}(n - m + 1)l_j\right) = O((n - m + 1)m) ,$$

the same as without gap characters.

Solution to Exercise 32.2-2

Let the patterns be $P_1, P_2, \ldots, P_k$.

First, assume that the patterns all have length m. For each pattern P_j, compute its numerical value p_j, taking time $\Theta(km)$. Then, for each of the $n - m + 1$ shift amounts s, compare t_s with each p_j, and if they are equal, compare P_j with the substring $T[s + 1 : s + m]$. The total running time increases by a factor of k, to $O((n - m + 1)km)$, the same as performing k distinct calls of RABIN-KARP-MATCHER.

Now, suppose that the patterns have lengths $l_1, l_2, \ldots, l_k$. Instead of a single value t_s, maintain k values $t_{s,j}$ for $j = 1, 2, \ldots, k$, and compare p_j with $t_{s,j}$. If $r = l_1 + l_2 + \cdots + l_k$ and $w = \min\{l_1, l_2, \ldots, l_k\}$ the running time is $O((n - w + 1)(k + r))$. (Each of $n - w + 1$ iterations might compare a total of r characters, and then each iteration makes k updates to the $t_{s,j}$ values.)

Solution to Exercise 32.2-3

Extending the Rabin-Karp method to a 2-dimensional $m \times m$ pattern is more complicated than a 1-dimensional pattern. As the pattern slides along the $n \times n$ character array, m values leave p and m values enter. Moreover, we must treat vertical and horizontal groups of m values differently.

In the 1-dimensional case, we computed $p = \left(\sum_{j=1}^{m} d^{m-j} P[j]\right) \bmod q$. For the 2-dimensional case, where we have the pattern $P = P[1:m, 1:m]$, we treat the pattern in row major order, so that

$$p = \left(\sum_{i=1}^{m} \sum_{j=1}^{m} d^{m^2-(m(i-1)+j)} P[i,j]\right) \bmod q \; .$$

Similarly, compute

$$t_{0,0} = \left(\sum_{i=1}^{m} \sum_{j=1}^{m} d^{m^2-(m(i-1)+j)} T[i,j]\right) \bmod q \; .$$

When sliding the pattern down by one row, subtract out the top row of the portion of T that is leaving, and add in the new bottom row. This change takes $\Theta(m)$ time. When sliding the pattern right by one column, subtract out the left column of the portion of T that is leaving, and add in the new right column, again taking $\Theta(m)$ time. Similarly, we can slide the pattern up or left as needed.

Since we can slide the pattern in any of the four directions, we can cover the entire text by using a "serpentine" pattern: keep sliding right until reaching the right end of the text, then go down by one row, keep sliding left until reaching the left end of the text, go down another row, keep sliding right, and so on.

It takes $O(m^2)$ time to verify that the pattern matches a square of the text when there's a hit, and $n^2 - m^2 + 1$ shift amounts need to be checked, for a total time of $O(m^2(n^2 - m^2 + 1))$.

Solution to Exercise 32.2-4

If files A and B are the same, then obviously $A(x) = B(x)$. So now suppose that the files differ, but that $A(x) = B(x)$ or, equivalently,

$$\left(\sum_{i=0}^{n-1} (a_i - b_i)x^i\right) \bmod q = 0 \; .$$

By Exercise 31.4-4, the polynomial $\sum_{i=0}^{n-1} (a_i - b_i)x^i$ has at most n distinct zeros modulo q, so that there are at most n values of x such that $A(x) = B(x)$. There are $q > 1000n$ choices for x, and so the probability that $A(x) = B(x)$ is at most $n/q < n/(1000n) = 1/1000$.

Solution to Exercise 32.3-1

Instead of drawing a diagram, here is a table of the state transitions:

state	a	b	*P*
0	1	0	a
1	2	0	a
2	2	3	b
3	4	0	a
4	2	5	a
5	1	0	

On the text string $T = \texttt{aaababaabaababaab}$, it goes through the following sequence:

state 0
state 1
state 2
state 2
state 3
state 4
state 5, prints "Pattern occurs with shift 1"
state 1
state 2
state 3
state 4
state 2
state 3
state 4
state 5, prints "Pattern occurs with shift 9"
state 1
state 2
state 3

Solution to Exercise 32.3-2

Instead of drawing a diagram, here is a table of the state transitions:

state	a	b	P
0	1	0	a
1	1	2	b
2	3	0	a
3	1	4	b
4	3	5	b
5	6	0	a
6	1	7	b
7	3	8	b
8	9	0	a
9	1	10	b
10	11	0	a
11	1	12	b
12	3	13	b
13	14	0	a
14	1	15	b
15	16	8	a
16	1	17	b
17	3	18	b
18	19	0	a
19	1	20	b
20	3	21	b
21	9	0	

Solution to Exercise 32.3-3

In a nonoverlappable pattern, no prefix is a suffix of any other prefix, and so all transitions are to state 0 except for those along the spine.

Solution to Exercise 32.3-4

Since $\sigma(x) = \max\{k : P[:k] \sqsupset x\}$, let's denote by z the longest prefix of P that is a suffix of x. Then $x = wz$ for some string w, possibly empty. Since x is a suffix of y, write y as $tx = twz$ for some string t, also possibly empty. Then, z is a suffix of y, so that $\sigma(y)$ is at least the length of z, which equals $\sigma(x)$. Hence, we have that $\sigma(y) \geq \sigma(x)$.

Solution to Exercise 32.3-5

[This solution refers more deeply to automata theory than the textbook does.]

First, create finite automata M and M' for patterns P and P', with start states q_0 and q'_0, respectively. Next, create a new start state q^*, and add transitions

$\delta(q^*, \varepsilon) = q_0$ and $\delta(q^*, \varepsilon) = q'_0$. We now have a nondeterministic finite automaton. Use the algorithm that converts a nondeterministic finite automaton to a (deterministic) finite automaton, and then use the Myhill-Nerode theorem to minimize the number of states.

Solution to Exercise 32.3-6

As in the solution to Exercise 32.1-4, decompose the pattern P as $P_1 \diamondsuit P_2 \diamondsuit P_3 \diamondsuit \ldots \diamondsuit P_k$. Look for the first occurrence of P_1 in the text. If found, start looking for the first occurrence of P_2 after the end of the first occurrence of P_1. If P_2 is found, start looking for the first occurrence of P_3 after the end of the first occurrence of P_2, and so on.

The way to do so with finite automata is to create k finite automata, $M_1, M_2, \ldots, M_k$, where M_j is for subpattern P_j. Let M_j have states $q_{j,0}, q_{j,1}, \ldots, q_{j,l_j}$, where l_j is the length of P_j. For $j = 1, 2, \ldots, k-1$ and each character $a \in \Sigma$, add a transition $\delta(q_{j,l_j}, a) = \delta(q_{j+1,0}, a)$, so that after matching on P_j in M_j, the automaton goes to the state in M_{j+1} that M_{j+1} would go to upon seeing the next character. The only state in which the entire pattern has been found is q_{k,l_k}, the last state in the last automaton.

Solution to Exercise 32.5-2

Once the *left-rank* values are unique, the sorted order will never change. Have MAKE-RANKS return the highest rank value it assigns. If MAKE-RANKS returns n, COMPUTE-SUFFIX-ARRAY can break out of the **while** loop. An input that would allow COMPUTE-SUFFIX-ARRAY to make just one call of MAKE-RANKS would have n unique characters. An input that would force COMPUTE-SUFFIX-ARRAY to make the maximum number of iterations would be one character repeated n times.

Solution to Exercise 32.5-3

Let `@` be a character that appears in neither T_1 nor T_2. Construct the text T as the concatenation of T_1, `@`, and T_2. Create the suffix array and *LCP* array for T, which takes $O(n \lg n)$ time. Find the entries with the maximum *LCP* value $LCP[i]$ such that either $SA[i-1] \le n_1$ and $SA[i] > n_1$ or $SA[i-1] > n_1$ and $SA[i] \le n_1$. (If boolean values can be compared, the condition can be expressed as $(SA[i-1] \le n_1) \ne (SA[i] > n_1)$. For each such index i, one of the longest substrings appearing in both T_1 and T_2 is $T[SA[i] : SA[i] + LCP[i] - 1]$. It takes $\Theta(n)$ time to find such indices and $\Theta(kl)$ time to produce the common substrings.

Solution to Exercise 32.5-4

`addcbdd` causes the method to fail. In T', there are four suffixes beginning with the longest palindrome `dd`. They appear in the suffix array in the order `dd@ddbcdda`, `dda`, `ddbcdda`, `ddcbdd@ddbcdda`. The trouble is that the first and third of these, `dd@ddbcdda` and `ddbcdda`, match up, as do the second and fourth, `dda` and `ddcbdd@ddbcdda`. The suffixes that match up do not appear in consecutive positions of the suffix array, and therefore Professor Markram's method misses them.

Solution to Problem 32-2

a. First, we show that the order of suffixes of P is the same as the order of its nonempty suffixes. We use lowercase Greek letters to denote a substring, possibly empty, of P, and individual lowercase letters to denote single metacharacters of P. Each metacharacter of P is a triple of characters from T.

If the metacharacters of P are unique, then the result is obvious, because the rank of a suffix is then determined by its first character.

Now, assume that P has duplicate metacharacters, so that it has at least two suffixes with some common prefix. Let these two suffixes be $\alpha b \beta$ and $\alpha c \gamma$, where the common prefix α has at least one metacharacter and, since any two distinct suffixes of a string must differ in at least one character, $b \neq c$. There are three cases, depending on whether each suffix starts in P_1 or in P_2.

- If both suffixes start in P_1, then let z be the metacharacter in P_1 in which at least one $\varnothing$ appears. By the definition of P_1, it must contain such a metacharacter. Rewrite $\alpha b \beta$ as $\alpha b \delta z P_2$, and rewrite $\alpha c \gamma$ as $\alpha c \lambda z P_2$, where b or c could be z, in which case δ or λ is an empty string. With these suffixes rewritten, $\alpha b \beta$ is lexicographically less than $\alpha c \gamma$ if and only if b is lexicographically less than c. The substring P_2 does not affect the comparison, so that the order of the suffixes of P is the same as the order of its nonempty suffixes.
- If both suffixes start in P_2, then they are, by definition, nonempty suffixes. In this case, therefore, the order of the suffixes of P must be the same as the order of its nonempty suffixes.
- If one suffix starts in P_1 and the other starts in P_2, then without loss of generality, let $\alpha b \beta$ start in P_1 and $\alpha c \gamma$ start in P_2. As in the first case, let z be the metacharacter of P_1 containing at least one $\varnothing$. Rewrite $\alpha b \beta$ as $\alpha b \delta z \lambda \alpha c \gamma$, where b could be z, in which case δ is empty. As in the second case, because $\alpha c \gamma$ starts in P_2, it is a nonempty suffix. The nonempty suffix of $\alpha b \beta$ is $\alpha b \delta z$. Since $\alpha b \beta$ is lexicographically less than $\alpha c \gamma$ if and only if b is lexicographically less than c, and since b is part of the nonempty suffix of $\alpha b \beta$, once again the order of the suffixes of P must be the same as the order of its nonempty suffixes.

Since, not counting the $\varnothing$ metacharacters, the nonempty suffixes of P correspond to the sample suffixes of T, the order of suffixes of P gives the order of sample suffixes of T.

b. Use radix sort. Initially, the size of the underlying encoding is independent of n. In the recursive invocations of substep step:SA-P', each metacharacter comprises three integers, each of which is at most n. In either case, you can run counting sort on each of the three-character encodings in $\Theta(n)$ time.

c. Because the ranks r_i are unique, even if two characters in the tuples are equal, the tuples will be unique. Again, use radix sort.

d. First, observe that because $T[j:]$ is a nonsample suffix, it must be the case that $j \bmod 3 = 0$, which in turn means that $(j + 1) \bmod 3 = 1$. Therefore, $r_{j+1} \neq \square$.

Taking a cue from the hint, consider first the case in which $i \bmod 3 = 1$. Since $i \bmod 3 = 1$, we also have $(i + 1) \bmod 3 = 2$, so that $r_{i+1} \neq \square$. Therefore, you can determine whether $T[i:]$ is lexicographically smaller than $T[j:]$ by comparing the tuples $(T[i], r_{i+1})$ and $(T[j], r_{j+1})$.

Now, suppose that $i \bmod 3 = 2$. In this case, $(i+1) \bmod 3 = 0$, so that $r_{i+1} = \square$. But we also have that $(i + 2) \bmod 3 = 1$, so that $r_{i+2} \neq \square$. Therefore, you can determine whether $T[i:]$ is lexicographically smaller than $T[j:]$ by comparing the tuples $(T[i], T[i + 1], r_{i+2})$ and $(T[j], T[j + 1], r_{j+2})$.

Therefore, merging the sorted sets of suffixes can be done in linear time. One set of sorted suffixes contains suffixes $T[i:]$, where $i \bmod 3$ equals either 1 or 2, and the other set contains suffixes Tsj, where $j \bmod 3 = 0$. When merging, determine whether the suffix from the first sorted set has $i \bmod 3 = 1$ or $i \bmod 3 = 2$, and compare with the suffix from the second sorted set according to the above tuple comparisons.

e. Each of the substeps takes $\Theta(n)$ time, except for substep when it has to recursively compute $SA_{P'}$. When this substep recurses, it is on a subproblem of size at most $2n/3 + 2$. Therefore, the recurrence for the running time is $T(n) \leq T(2n/3 + 2) + \Theta(n)$. We can ignore the $+2$ in the argument, and this recurrence falls into case 3 of the master theorem (Theorem 4.1) with the solution $T(n) = O(n)$. (Because the recurrence uses "$\leq$" instead of "$=$" the solution is $O(n)$ rather than $\Theta(n)$. The entire input must be examined, however, so that the running time is $\Theta(n)$.

Solution to Problem 32-3

a. Look again at the sorted cyclic rotations of T'. For the example of $T' =$ `rutabaga$`,

```
$rutabaga
a$rutabag
abaga$rut
aga$rutab
baga$ruta
ga$rutaba
rutabaga$
tabaga$ru
utabaga$r
```

Notice that the first character in each row follows the last character in that row within T', except for the row that equals T' (and therefore ends with `$`.) Now remove everything after `$` in the sorted cyclic rotations of T'. In our example, we get

```
$
a$
abaga$
aga$
baga$
ga$
rutabaga$
tabaga$
utabaga$
```

The rows are the suffixes of T', sorted lexicographically. Now take the suffix array of T', which in this example is $\langle 9, 8, 4, 6, 5, 7, 1, 3, 2 \rangle$. Since the character in the last column of the sorted full rows precedes the character in the first column (with that one exception), $T'[SA[i-1]]$ gives the ith character of the BWT when $SA[i] > 1$. If $SA[i] = 1$, then the "preceding" character of T' is `$`, and so the ith character of the BWT is `$` if $SA[i] = 1$.

In our example, the characters of the BWT are, in order, $T'[8] =$ `a`, $T'[7] =$ `g`, $T'[3] =$ `t`, $T'[5] =$ `b`, $T'[4] =$ `a`, $T'[6] =$ `a`, `$`, $T'[2] =$ `u`, $T'[1] =$ `r`.

Thus it is simple to compute the BWT in $\Theta(n)$ time from the suffix array for T'.

b. MAKE-RANK(bwt, n)

```
let the alphabet have $C$ characters, with ord values 0 to $C - 1$
allocate array $count[0:C-1]$ with an entry for each character in the alphabet
**for** each character $c$ in the alphabet
    $count[\text{ord}(c)] = 0$
**for** $i = 1$ **to** $n$
    $count[\text{ord}(bwt[i])] = count[\text{ord}(bwt[i])] + 1$
**for** $i = 1$ **to** $C - 1$
    $count[i] = count[i] + count[i-1]$
**for** $i = C - 1$ **downto** 1
    $count[i] = count[i-1]$
allocate array $rank[1:n]$
**for** $i = 1$ **to** $n$
    $rank[i] = count[\text{ord}(bwt[i])] + 1$
    $count[\text{ord}(bwt[i])] = count[\text{ord}(bwt[i])] + 1$
**return** $rank$
```

c. INVERSE-BWT$(bwt, rank, n)$

```
use linear search to find the index $i$ of `$` in $bwt$
$inverse = \varepsilon$
**for** $j = 1$ **to** $n$
    prepend $bwt[i]$ to $inverse$ (i.e., $inverse = bwt[i]$ concatenated with $inverse$)
    $i = rank[i]$
**return** $inverse$
```

Lecture Notes for Chapter 35: Approximation Algorithms

Chapter 35 overview

What to do about the optimization versions of NP-complete problems? We cannot just give up, because some of them are too important to ignore.

- Run an optimal, but exponential-time, algorithm on only small inputs.
- Find special cases that can be solved in polynomial time.
- Use an algorithm that produces a solution that might not be optimal, but is close enough: an ***approximation algorithm***.

Performance ratios

When an approximation algorithm produces a solution, how good is the solution? How close to optimal is it?

- The problem could be a maximization problem, or it could be a minimization problem.
- If the input has size n, the solution cost of the approximation algorithm is C, and the optimal solution cost is C^*, then the algorithm has a $\boldsymbol{\rho(n)}$ ***approximation ratio*** if

  $$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \le \rho(n) .$$

  We call the algorithm a $\boldsymbol{\rho(n)}$***-approximation algorithm***.
- Taking both ratios gives us $\rho(n) \ge 1$ regardless of whether it's a maximization or minimization problem. The solution is optimal if and only if $\rho(n) = 1$.

We'll see two approximation algorithms for minimization, each a 2-approximation algorithm. *[So that $\rho(n)$ is a constant.]*

The book covers other types of approximation algorithms, where if you use more time, you can get a tighter approximation.

- An ***approximation scheme*** is an approximation algorithm that takes as input an instance of a problem plus a parameter $\epsilon > 0$ such that for fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation algorithm.

- It's a ***polynomial-time approximation scheme*** if for fixed $\epsilon > 0$, it runs in time polynomial in the input size n.

  But as ϵ decreases (approximation gets tighter), the running time can increase quickly, e.g., $O(n^{2/\epsilon})$.

Vertex cover

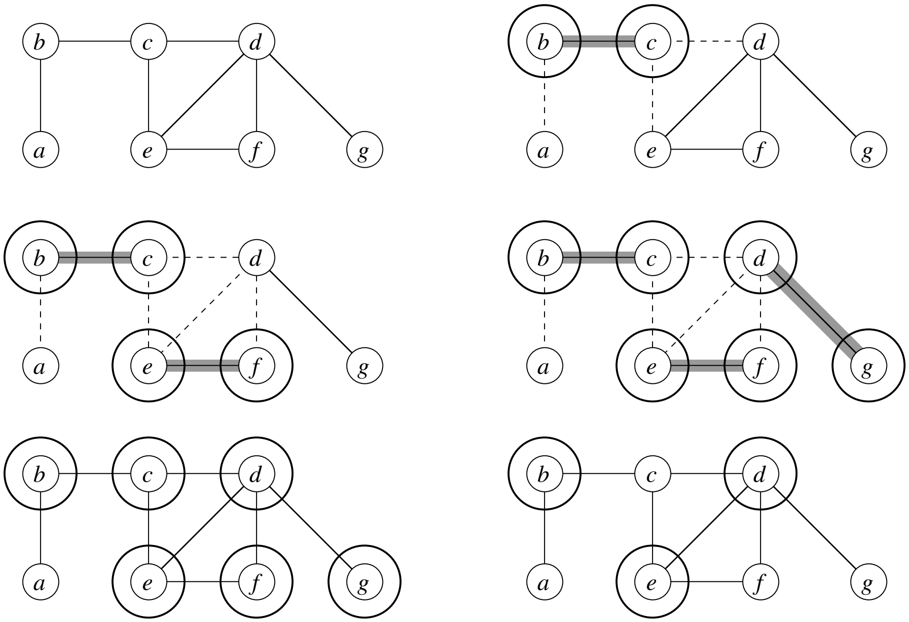
Input: Undirected graph $G = (V, E)$.

Output: A minimum-size subset $V' \subseteq V$ such that each edge in E is incident on at least one vertex in V'.

It's the optimization version of an NP-complete problem.

```
APPROX-VERTEX-COVER(G)
  $C = \emptyset$
  $E' = G.E$
  while $E' \neq \emptyset$
      let $(u, v)$ be an arbitrary edge of $E'$
      $C = C \cup \{u, v\}$
      remove from $E'$ edge $(u, v)$ and every edge incident on either $u$ or $v$
  return $C$
```

Example: Edges considered are shaded. Vertices are circled as they are added to the vertex cover C. Edges are dashed as they are removed from E'.



vertex cover produced by APPROX-VERTEX-COVER

optimal vertex cover

Running time: Using adjacency lists to represent E', $O(V + E)$.

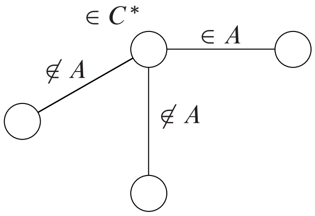
Theorem

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof Already shown that it runs in polynomial time. Need to show that it produces a vertex cover, and that the size of the vertex cover it produces is within a factor of 2 of optimal.

The procedure produces a vertex cover because it loops until every edge in E' has been covered by some vertex in C.

Now let C^* be an optimal vertex cover. Need to show that $|C| \le 2|C^*|$. Let A be the set of edges chosen in the **while** loop. To cover the edges in A, any vertex cover—including C^*—must include at least one endpoint of each edge in A. No two edges in A share an endpoint (once we pick an edge, all other edges incident on its endpoints are removed from E').



For every vertex in C^*, there is at most one edge of A, so that $|C^*| \ge |A|$.

Each time APPROX-VERTEX-COVER chooses an edge, neither of the endpoints can be in C. That means each edge chosen puts two vertices into C, so that $|C| = 2|A|$. Therefore,

$$\begin{aligned} |C| &= 2|A| \\ &\le 2|C^*| . \end{aligned}$$ ■

How did this proof work when we don't even know the size of an optimal vertex cover? We got a lower bound on its size ($|C^*| \ge |A|$) and then found an upper bound on the size of the approximate vertex cover that was within a factor of 2 of the lower bound.

Traveling-salesperson problem

Also known as TSP.

Input: A complete undirected graph $G = (V, E)$ with a nonnegative cost $c(u, v)$ for each edge $(u, v) \in E$.

Output: A ***tour*** (a hamiltonian cycle—a cycle that visits every vertex) with minimum total cost.

Also the optimization version of an NP-complete problem.

With the triangle inequality

Assume that the triangle inequality holds:

$$c(u, w) \le c(u, v) + c(v, w)$$

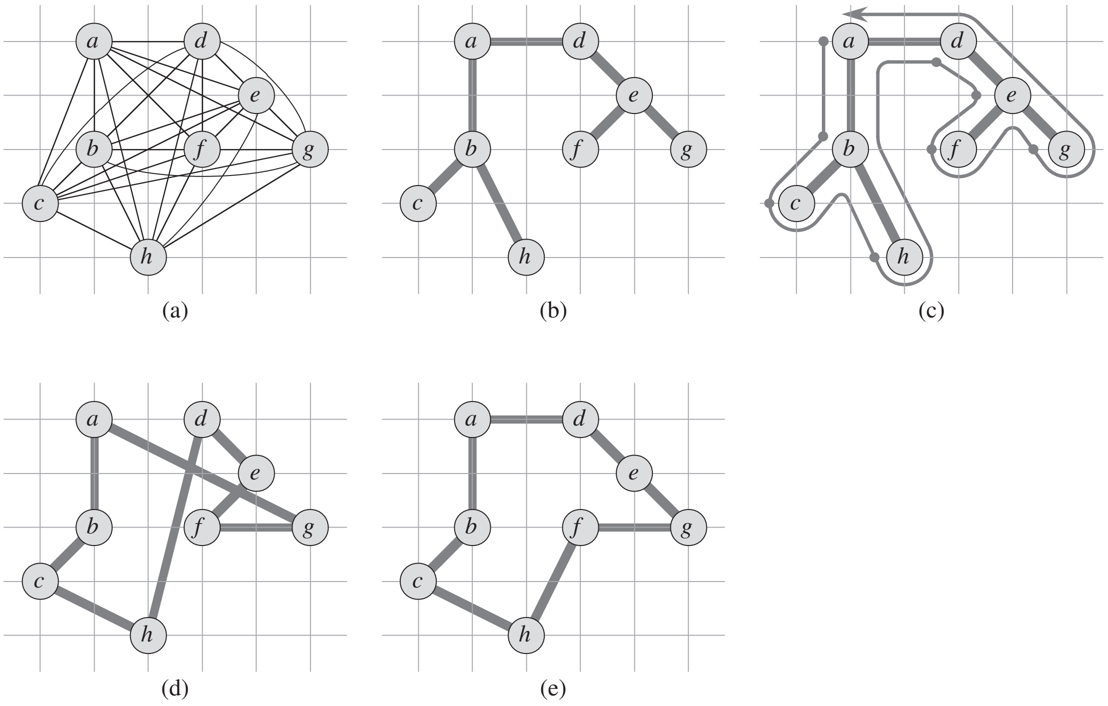
for all $u, v, w \in V$. The triangle inequality does not always hold, but it does when the vertices are points in the plane and the costs are euclidean distances between vertices. (Other cost functions can satisfy the triangle inequality.)

TSP is NP-complete even when the triangle inequality holds. The book shows that if the triangle inequality does not hold, then there is no polynomial-time approximation algorithm with a constant approximation ratio unless P = NP. So we'll concentrate on when the triangle inequality holds.

Idea: Construct a minimum spanning tree, do a preorder walk of the tree, and construct the TSP tour in the order in which each vertex is first visited during the preorder walk.

```
APPROX-TSP($G, c$)
  select a vertex $r \in G.V$ to be a root vertex
  call MST-PRIM($G, c, r$) to construct a minimum spanning tree $T$
  perform a preorder walk of $T$, and make a list $H$ of vertices,
        ordered according to when first visited
  return the list $H$ as the tour
```

Example: Using euclidean distance.



(a) A complete undirected graph.

(b) A minimum spanning tree with root a.

(c) A preorder walk of the spanning tree, with a dot next to each vertex the first time it's visited.

(d) The tour obtained by visiting the vertices in that order. Its total cost is approximately 19.074.

(e) An optimal tour with cost approximately 14.715.

Time: Easy to make Prim's algorithm run in $O(V^2)$ time. The rest is $\Theta(V)$, so total running time is $O(V^2)$.

Theorem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesperson problem with the triangle inequality.

Proof Already shown that it runs in polynomial time. It's obvious that it produces a tour. Need to show that the cost of the tour is at most 2 times the cost of an optimal tour. For any subset $A \subseteq E$ of edges, let $c(A) = \sum_{(u,v)\in A} c(u,v)$ be the total cost of the edges in A.

Let H^* be an optimal tour. Delete any edge from H^*, and get a spanning tree T^* whose cost is no greater than the cost of H^*, since all edge weights are nonnegative. Since the minimum spanning tree T computed in APPROX-TSP has a cost at most that of T^*, have

$$c(T) \le c(T^*) \le c(H^*) .$$

Instead of the preorder walk, think of the ***full walk*** of T, which lists each vertex whenever it's visited, including when returning from a visit to a subtree. In the example above, a full walk of T visits vertices in the order

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

The full walk, call it W, visits every edge exactly twice, so $c(W) = 2c(T)$. *[We're extending the cost notation to allow edges to appear mulitple times—twice in this case—in the summation.]* So now we have

$$c(W) = 2c(T) \le 2c(H^*) .$$

The full walk W is not a tour, since it visits some vertices more than once. Instead, remove vertices from W to make it a tour by leaving only the first instance of each vertex in W. By the triangle inequality, the total length cannot increase by removing a vertex: if the full walk contains vertices u, v, w in order, then by removing v, we have $c(u,w) \le c(u,v) + c(v,w)$. What we get is the tour H, where $c(H) \le c(W)$. And now

$$\begin{aligned} c(H) &\le c(W) \\ &\le 2c(H^*) . \end{aligned}$$

■

[This algorithm is nowhere near the best approximation algorithm for the traveling-salesperson problem. Much better approximations are possible.]

Without the triangle inequality

[This material assumes that students understand that there is a polynomial-time algorithm for the hamiltonian-cycle problem if and only if P = NP. *Otherwise, you need to present some background material on* P = NP.*]*

If we cannot assume that the cost function satisfies the triangle inquality, then we cannot find good approximate tours in polynomial time unless P = NP.

Theorem
If P $\neq$ NP, then for any constant $\rho \geq 1$, there is no polynomial-time ρ-approximation algorithm for the general traveling-salesperson problem.

Proof

Idea: Proof by contradiction. Assume that a polynomial-time ρ-approximation algorithm exists. Given an instance of the hamiltonian-cycle problem, create in polynomial time an instance of TSP that has a small optimal value if the original graph has a hamiltonian cycle, but a high optimal value if the original graph does not have a hamiltonian cycle. Then use the ρ-approximation algorithm to get an approximate bound on the cost of the TSP tour. This bound is low if and only if the original graph has a hamiltonian cycle. Then we have a way to solve the hamiltonian-cycle problem in polynomial time.

Given an undirected graph $G = (V, E)$ as an instance of the hamiltonian-cycle problem, suppose that there is a ρ-approximation algorithm for TSP. Create an instance $G' = (V, E')$ of TSP. G' is a complete graph:

$$E' = \{(u, v) : u, v \in V \text{ and } u \neq v\} \ .$$

Without loss of generality, assume that ρ is an integer, rounding up if necessary. Define integer edge costs in E':

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \ , \\ \rho\,|V| + 1 & \text{if } (u, v) \notin E \ . \end{cases}$$

Easy to create G' and c in polynomial time.

We will show that G has hamiltonian cycle if and only if G' with cost c has a tour with cost $|V|$, which means that G has hamiltonian cycle if and only if the approximation algorithm A would find a tour with cost at most $\rho\,|V|$.

If G has a hamiltonian cycle H, then the edges of H each have cost 1 in G', and so G' has a tour with cost $|V|$. The approximation algorithm would find a tour with cost at most $\rho\,|V|$.

If G does not have a hamiltonian cycle, then any tour of G' must use some edge not in E. This edge has cost $\rho\,|V| + 1$. The tour of G' has at most $|V| - 1$ edges with cost 1, and so the cost of the tour is at least

$$(\rho\,|V| + 1) + (|V| - 1) = \rho\,|V| + |V| \ .$$

Therefore, the lowest tour cost that the approximation algorithm A could give is $\rho\,|V| + |V|$, which is greater than $|V|$.

So if A gives a tour with cost at most $\rho\,|V|$, then G has a hamiltonian cycle, and if A gives a tour with cost at least $\rho\,|V| + |V|$, then G does not contain a hamiltonian cycle. Thus, there is no polynomial-time ρ-approximation algorithm for the general TSP unless P = NP. ■

Set-covering problem

Input: X, a finite set, and $\mathcal{F}$, a family of subsets of X such that every element of X belongs to at least one subset in $\mathcal{F}$: $X = \bigcup_{S \in \mathcal{F}} S$.

Output: $\mathcal{C} \subseteq \mathcal{F}$, where $\mathcal{C}$ covers X.

(A subfamily $\mathcal{C} \subseteq \mathcal{F}$ ***covers*** a set U if $U \subseteq \bigcup_{S \in \mathcal{C}} S$.)

Want to minimize $|\mathcal{C}|$, the number of sets in $\mathcal{C}$.

Idea:

- Corresponding decision problem generalizes the NP-complete vertex-cover problem, therefore NP-hard.
- Approximation algorithm for vertex-cover problem doesn't apply. Need another approach.
- Our approach will have a logarithmic approximation ratio: the size of the approximate solution grows relative to the size of an optimal solution as the problem instance grows. Still useful.

A greedy approximation algorithm

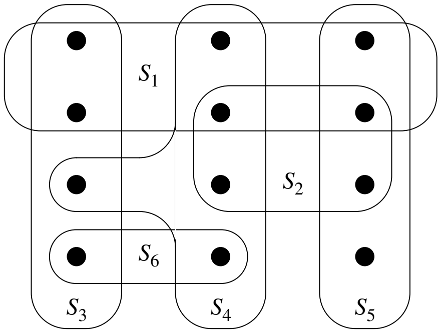
Description:

- After the ith iteration, set U_i contains the remaining uncovered elements. Initially, $U_0 = X$, all the elements. Set $\mathcal{C}$ contains the cover being constructed.
- Repeatedly choose a subset S that covers the most uncovered elements possible. Break ties arbitrarily. Keep going until all elements are covered.
- After choosing S, create U_{i+1} by removing remove elements of S from U_i. Place S into $\mathcal{C}$.
- When the algorithm returns $\mathcal{C}$, the set $\mathcal{C}$ contains a subfamily of $\mathcal{F}$ that covers X.

```
GREEDY-SET-COVER($X, \mathcal{F}$)
  $U_0 = X$
  $\mathcal{C} = \emptyset$
  $i = 0$
  while $U_i \neq \emptyset$
      select $S \in \mathcal{F}$ that maximizes $|S \cap U_i|$
      $U_{i+1} = U_i - S$
      $\mathcal{C} = \mathcal{C} \cup \{S\}$
      $i = i + 1$
  return $\mathcal{C}$
```

The number of iterations of the **while** loop is bounded above by $\min\{|X|, |\mathcal{F}|\} = O(|X| + |F|)$, and the loop body can be easily implemented to run in $O(|X| \cdot |\mathcal{F}|)$ time, so a simple implementation has running time $O(|X| \cdot |\mathcal{F}| \cdot (|X| + |\mathcal{F}|))$.

Example: The figure below shows an instance $(X, \mathcal{F})$ of the set covering. X is the 12 black points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$.



GREEDY-SET-COVER selects S_1, then S_4, then S_5, then S_3 or S_6.

A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$.

[The following theorem and its proof are new in the fourth edition.]

Theorem

GREEDY-SET-COVER is a polynomial-time $O(\lg X)$-approximation algorithm.

Proof Have already shown that the algorithm runs in $O(|X| \cdot |\mathcal{F}| \cdot (|X| + |\mathcal{F}|))$ time, which is polynomial in the input size.

Need to prove the approximation bound.

- Let $\mathcal{C}^*$ be an optimal set cover for the instance $(X, \mathcal{F})$, and $k = |\mathcal{C}^*|$.
- $\mathcal{C}^*$ is also a set cover for each subset U_i created during the algorithm.
  $\Rightarrow$ Any U_i can be covered by k sets.
  $\Rightarrow$ An instance $(U_i, \mathcal{F})$ has an optimal set cover of size $\leq k$.
- If an optimal set cover for instance $(U_i, \mathcal{F})$ has size $\leq k$, then at least one of the sets in $\mathcal{C}$ covers at least $|U_i|/k$ new elements.
  $\Rightarrow$ The set S chosen in each iteration must choose a set covering $\geq |U_i|/k$ new elements.
- These elements are removed when creating U_{i+1}, so
  $$\begin{aligned} |U_{i+1}| &\leq |U_i| - |U_i|/k \\ &= U_i(1 - 1/k) . \end{aligned}$$
- Iterating this inequality gives
  $$\begin{aligned} |U_0| &= X , \\ |U_1| &\leq |U_0|(1 - 1/k) , \\ |U_2| &\leq |U_1|(1 - 1/k) \leq |U_0|(1 - 1/k)^2 . \end{aligned}$$
  In general:
  $$|U_i| \leq |U_0|(1 - 1/k)^i = |X|(1 - 1/k)^i .$$
- The algorithm stops when $|U_i| = 0$
  $\Rightarrow |U_i| < 1$
  $\Rightarrow$ Upper bound on number of iterations is smallest i such that $|U_i| < 1$.

- Using inequality (3.14), $1 + x \le e^x$ for all real x.
  Let $x = -1/k \Rightarrow 1 - 1/k \le e^{-1/k} \Rightarrow (1 - 1/k)^k \le (e^{-1/k})^k = e^{-1}$.
- Denote number i of iterations by ck for some $c \ge 0$.
  Want c such that $|X|\,(1 - 1/k)^{ck} \le |X|\,e^{-c} < 1$.
- Multiply both sides by e^c and take natural logarithm of both sides, giving $c > \ln|X|$.
  $\Rightarrow$ Can choose for c any integer $> \ln|X|$.
- Choose $c = \lceil \ln|X| \rceil + 1$.
- $i = ck$ is an upper bound on number of iterations, which equals $|\mathcal{C}|$. $k = |\mathcal{C}^*|$.
  Thus, $|\mathcal{C}| \le i = ck = c\,|\mathcal{C}^*| = |\mathcal{C}^*|\,(\lceil \ln|X| \rceil + 1)$.
  $\Rightarrow |\mathcal{C}| = |\mathcal{C}^*|\,O(\lg X)$. ■

Randomization and linear programming

This section presents two useful techniques for designing approximation algorithms:

1. Randomization, illustrated with an optimization version of 3-CNF satisfiability.
2. Linear programming, illustrated with a weighted version of the vertex-cover problem.

A randomized approximation algorithm for MAX-3-CNF satisfiability

Approximation ratios

- A randomized algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the *expected* cost C of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution: $\max\{C/C^*, C^*/C\} \le \rho(n)$.
- A randomized algorithm that achieves an approximation ratio of $\rho(n)$ is a ***randomized $\rho(n)$-approximation algorithm.***
- A randomized approximation algorithm is like a deterministic approximation algorithm, except that the approximation ratio is for an expected cost.

MAX-3-CNF satisfiability

MAX-3-CNF satisfiability problem:

3-CNF satisfiability (defined in Section 34.4) may or may not be satisfiable. Want to find an assignment of variables that satisfies as many clauses as possible.

Input: An instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ and m clauses.

Output: An assignment of 0 or 1 to the n variables $x_1, x_2, \ldots, x_n$ that maximizes the number of clauses satisfied.

Approximation algorithm

Randomly set each variable to 1 with probability $1/2$ and to 0 with probability $1/2$. Yields a randomized $8/7$-approximation algorithm.

(Require each clause to consist of exactly three literals. Assume that no clause contains a variable and its negation.)

Theorem

Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ is a randomized $8/7$-approximation algorithm.

Proof Suppose that each variable is independently set to 1 with probability $1/2$ and to 0 with probability $1/2$.

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

$Y_i = \mathrm{I}\{\text{clause } i \text{ is satisfied}\}$,

so that $Y_i = 1$ as long as at least one of the literals in the ith clause is set to 1.

Since no literal appears more than once in the same clause, and since no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent.

A clause is not satisfied only if all three of its literals are set to 0, and so $\Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8$. Thus, $\Pr\{\text{clause } i \text{ is satisfied}\} = 1 - 1/8 = 7/8$, and by Lemma 5.1, $\mathrm{E}[Y_i] = 7/8$.

Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Y_m$. Then,

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\
&= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{m} 7/8 \\
&= 7m/8 \,.
\end{aligned}$$

Clearly, m is an upper bound on the number of satisfied clauses, and hence the approximation ratio is at most $m/(7m/8) = 8/7$. ■

Approximating weighted vertex cover using linear programming

Minimum-weight vertex-cover problem:

Input: Undirected graph $G = (V, E)$, where each vertex $v \in V$ has a positive weight $w(v)$.

Output: A minimum-weight subset $V' \subseteq V$ such that each edge in E is incident on at least one vertex in V'.

The weight of a vertex cover V' is defined as $w(V') = \sum_{v \in V'} w(v)$.

Idea: Compute a lower bound on the weight of the minimum-weight vertex cover by solving a linear program. Then "round" the solution to obtain a vertex cover.

Build a 0-1 integer program for finding a minimum-weight vertex cover as follows:

- Associate a variable $x(v)$ for each vertex $v \in V$.
- $x(v)$ is 0 or 1 for each $v \in V$. Put v in the vertex cover if and only if $x(v) = 1$.
- For any edge (u, v), at least one of u and v is in the vertex cover, giving the constraint $x(u) + x(v) \geq 1$.

This idea leads to the following 0-1 integer program, which formulates the optimization version of the NP-hard vertex-cover problem:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v)$$

subject to

$$\begin{aligned} x(u) + x(v) &\geq 1 && \text{for each } (u, v) \in E \\ x(v) &\in \{0, 1\} && \text{for each } v \in V \,. \end{aligned}$$

Now, remove the constraint that $x(v) \in \{0, 1\}$, and replace it by $0 \leq x(v) \leq 1$.

Get the following ***linear-programming relaxation***:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v)$$

subject to

$$\begin{aligned} x(u) + x(v) &\geq 1 && \text{for each } (u, v) \in E \\ x(v) &\leq 1 && \text{for each } v \in V \\ x(v) &\geq 0 && \text{for each } v \in V \,. \end{aligned}$$

Any feasible solution to the 0-1 integer program is also a feasible solution to the linear program.
$\Rightarrow$ The value of an optimal solution to the linear program gives a lower bound on the value of an optimal solution to the 0-1 integer program, and therefore, a lower bound on the optimal weight in the minimum-weight vertex-cover problem.

The following procedure uses the above idea to return an approximate solution to the minimum-weight vertex-cover problem:

- Initialize an empty vertex cover C and solve the linear program.
- Every vertex v will have a value $\bar{x}$ where $0 \leq \bar{x} \leq 1$.
- Add vertices with $\bar{x} \geq 1/2$ to C,"rounding" these fractional variables in order to obtain a solution to the 0-1 integer program.

```
APPROX-MIN-WEIGHT-VC(G, w)
  C = ∅
  compute x̄, an optimal solution to the linear programming relaxation
  for each vertex v ∈ V
      if x̄(v) ≥ 1/2
          C = C ∪ {v}
  return C
```

Theorem

APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the linear program, and because the **for** loop runs in polynomial time, APPROX-MIN-WEIGHT-VC is a polynomial-time algorithm.

Need to show that APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm.

- Let C^* be an optimal solution to the minimum-weight vertex-cover problem, and let z^* be the value of an optimal solution to the linear program.
- Since an optimal vertex cover is a feasible solution to the linear program, z^* must be a lower bound on $w(C^*)$, that is, $z^* \le w(C^*)$.

  Claim: Rounding the fractional values of the variables $\bar{x}(v)$ produces a set C that is a vertex cover and satisfies $w(C) \le 2z^*$.

  Proof of claim:

- - To see that C is a vertex cover, consider any edge $(u, v) \in E$. By the first constraint in the linear program, $x(u) + x(v) \ge 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$. Therefore, at least one of u and v is included in the vertex cover, and so every edge is covered.
  - Now, consider the weight of the cover. We have

    $$\begin{aligned} z^* &= \sum_{v \in V} w(v)\,\bar{x}(v) \\ &\ge \sum_{v \in V : \bar{x}(v) \ge 1/2} w(v)\,\bar{x}(v) \\ &\ge \sum_{v \in V : \bar{x}(v) \ge 1/2} w(v) \cdot \frac{1}{2} \\ &= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\ &= \frac{1}{2} \sum_{v \in C} w(v) \\ &= \frac{1}{2} w(C) \; . \end{aligned}$$

    Combining inequalities $z^* \le w(C^*)$ and and $z^* \ge (1/2)w(C)$ gives

    $$w(C) \le 2z^* \le 2w(C^*) \; ,$$

    and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. ■

The subset-sum problem

Input: Pair (S, t), where S is a set $\{x_1, x_2, \ldots, x_n\}$ of positive integers and t is a positive integer.

Output: A subset of $\{x_1, x_2, \ldots, x_n\}$ whose sum is as large as possible but not larger than t.

Decision problem (seen in Section 34.5.5) asks whether there exists a subset of S that adds up exactly to the target value t. The subset-sum problem is NP-complete.

An exponential-time exact algorithm

Idea:

- Compute, for each subset S' of S, the sum of the elements in S'. Select the subset S' with sum closest to t.
- Returns an optimal solution, but takes exponential time.
- Implement with an iterative procedure. In iteration i, compute the sums of all subsets of $\{x_1, x_2, \ldots, x_i\}$, using as a starting point the sums of all subsets of $\{x_1, x_2, \ldots, x_{i-1}\}$.
- If a subset S' has a sum exceeding t, do not maintain it, since no superset of S' could be an optimal solution.

Notation: If L is a list of positive integers and x is another positive integer, then $L + x$ denotes the list of integers derived from L by increasing each element of L by x.

Example: $L = \langle 1, 2, 3, 5, 9 \rangle \Rightarrow L + 2 = \langle 3, 4, 5, 7, 11 \rangle$.

Also use this notation for sets: $S + x = \{s + x : s \in S\}$.

The procedure EXACT-SUBSET-SUM works as follows:

1. Iteratively compute L_i, the list of sums of all subsets of $\{x_1, \ldots, x_i\}$ that do not exceed t.
2. Use an auxiliary procedure MERGE-LISTS(L, L'), which returns the sorted list that is the merge of its two sorted input lists L and L' with duplicate values removed.
3. Return the maximum value in L_n.

```
EXACT-SUBSET-SUM$(S, n, t)$
  $L_0 = \langle 0 \rangle$
  for $i = 1$ to $n$
      $L_i$ = MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
      remove from $L_i$ every element that is greater than $t$
  return the largest element in $L_n$
```

The length of L_i can be as much as 2^i, and so EXACT-SUBSET-SUM has running time $O(2^n)$. Some special cases exist where it is a polynomial-time algorithm.

A fully polynomial-time approximation scheme

Fully polynomial-time approximation scheme comes from “trimming” each list L_i after it is created:

- If two values in list L are close to each other, do not need to maintain both explicitly (since need only an approximate solution).

- Use trimming parameter δ, where $0 < \delta < 1$.
- ***Trim*** a list L by δ to get result L': Remove as many elements from L as possible so that L' contains an element z that approximates every element y removed from L such that $y/(1+\delta) \le z \le y$.
- The element z "represents" y in L'.
- Trimming can decrease the number of elements kept while keeping a close representative value in the list for each deleted element.

Example: Trim $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$ by $\delta = 0.1$.

Result: $L' = \langle 10, 12, 15, 20, 23, 29 \rangle$.

- Deleted value 11 is represented by 10.
- Deleted values 21 and 22 are represented by 20.
- Deleted value 24 is represented by 23.

Trimming procedure

Input: List $L = \langle y_1, y_2, \ldots, y_m \rangle$ sorted into monotonically increasing order; parameter δ.

Output: A trimmed, sorted list L'.

Scans the elements of L in monotonically increasing order. A number is appended onto the returned list L' only if it is the first element of L or if it cannot be represented by the most recent number placed into L'.

Runs in $\Theta(m)$ time.

```
TRIM($L, \delta$)
  let $m$ be the length of $L$
  $L' = \langle y_1 \rangle$
  $last = y_1$
  for $i = 2$ to $m$
      if $y_i > last \cdot (1 + \delta)$        // $y_i \ge last$ because $L$ is sorted
          append $y_i$ onto the end of $L'$
          $last = y_i$
  return $L'$
```

Approximation scheme for subset-sum

Given TRIM, construct an approximation scheme.

Input: Set $S = \{x_1, x_2, \ldots, x_n\}$ of n integers (in arbitrary order), size $n = |S|$, target integer t, "approximation parameter" ϵ where $0 < \epsilon < 1$.

Output: Value z^*, within a $1 + \epsilon$ factor of the optimal solution.

Procedure overview:

- Initialize list L_0 to be the list containing 0.
- Let P_i be the set of values obtained by summing all 2^i subsets of $\{x_1, x_2, \ldots, x_i\}$.

- The **for** loop computes L_i as a sorted list containing a suitably trimmed version of the set P_i, with all elements larger than t removed.
- Since L_i is created from L_{i-1}, must ensure that the repeated trimming doesn't introduce too much compounded inaccuracy. Therefore use $\epsilon/2n$ for the trimming parameter δ, rather than ϵ.

Example: If $S = \{1, 4, 5\}$, then

- $P_1 = \{0, 1\}$.
- $P_2 = \{0, 1, 4, 5\}$.
- $P_3 = \{0, 1, 4, 5, 6, 9, 10\}$.

```
APPROX-SUBSET-SUM$(S, n, t, \epsilon)$
  $L_0 = \langle 0 \rangle$
  for $i = 1$ to $n$
      $L_i =$ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
      $L_i =$ TRIM$(L_i, \epsilon/2n)$
      remove from $L_i$ every element that is greater than $t$
  let $z^*$ be the largest value in $L_n$
  return $z^*$
```

Example: Suppose $S = \langle 104, 102, 201, 101 \rangle$ with $t = 308$ and $\epsilon = 0.40$.

The trimming parameter δ is $\epsilon/8 = 0.05$.

APPROX-SUBSET-SUM initially sets $L_0 = \langle 0 \rangle$.

Then, the procedure proceeds as follows for each iteration i in the three lines in the body of the **for** loop.

$i = 1$:

line 1: $L_1 = \langle 0, 104 \rangle$,

line 2: $L_1 = \langle 0, 104 \rangle$,

line 3: $L_1 = \langle 0, 104 \rangle$.

$i = 2$:

line 1: $L_2 = \langle 0, 102, 104, 206 \rangle$,

line 2: $L_2 = \langle 0, 102, 206 \rangle$,

line 3: $L_2 = \langle 0, 102, 206 \rangle$.

$i = 3$:

line 1: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,

line 2: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,

line 3: $L_3 = \langle 0, 102, 201, 303 \rangle$.

$i = 4$:

line 1: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,

line 2: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,

line 3: $L_4 = \langle 0, 101, 201, 302 \rangle$.

The algorithm returns $z^* = 302$ as its answer. This is well within $\epsilon = 40\%$ of the optimal answer $307 = 104 + 102 + 101$. In fact, it is within 2%.

Theorem

APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

Proof The operations of trimming L_i and removing from L_i every element that is greater than t maintain the property that every element of L_i is also a member of P_i. Therefore, the value z^* returned is indeed the sum of some subset of S.

Let $y^* \in P_n$ denote an optimal solution to the subset-sum problem. Because the procedure ensures that $z^* \le t$ by removing from L_i every element that is greater than t, must have $z^* \le y^*$.
Need to show that $y^*/z^* \le 1 + \epsilon$.
Must also show that the running time of this algorithm is polynomial in both $1/\epsilon$ and the size of the input.

- By Exercise 35.5-2, for every element y in P_i that is at most t, there exists an element $z \in L_i$ such that
$$\frac{y}{(1+\epsilon/2n)^i} \le z \le y \ .$$
- The above inequality must hold for $y^* \in P_n$.
$\Rightarrow$ there exists an element $z \in L_n$ such that
$$\frac{y^*}{(1+\epsilon/2n)^n} \le z \le y^* \ ,$$
and thus
$$\frac{y^*}{z} \le \left(1 + \frac{\epsilon}{2n}\right)^n \ .$$
- Since there exists an element $z \in L_n$ fulfilling the above inequality, the inequality must hold for z^*, which is the largest value in L_n; that is,
$$\frac{y^*}{z^*} \le \left(1 + \frac{\epsilon}{2n}\right)^n \ .$$
- Now, show that $y^*/z^* \le 1 + \epsilon$ by showing that $(1 + \epsilon/2n)^n \le 1 + \epsilon$.
    - $0 < \epsilon < 1 \Rightarrow (\epsilon/2)^2 \le \epsilon/2$.
    - By equation (3.16), $\lim_{n\to\infty}(1 + \epsilon/2n)^n = e^{\epsilon/2}$.
    By Exercise 35.5-3,
    $$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0 \ .$$
    - Therefore, the function $(1 + \epsilon/2n)^n$ increases with n as it approaches its limit of $e^{\epsilon/2}$, and
    $$\begin{aligned}\left(1 + \frac{\epsilon}{2n}\right)^n &\le e^{\epsilon/2} \\ &\le 1 + \epsilon/2 + (\epsilon/2)^2 \\ &\qquad \text{(inequality (3.15), } e^x \le 1 + x + x^2 \text{ when } |x| < 1) \\ &\le 1 + \epsilon \ .\end{aligned}$$
    - Combining the above inequality and $y^*/z^* \le (1 + \epsilon/2n)^n$ completes the analysis of the approximation ratio.

To show that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme, derive a bound on the length of L_i.

- After trimming, must have $z'/z > 1 + \epsilon/2n$ for successive elements z and z' of L_i, so that they differ by a factor of at least $1 + \epsilon/2n$.
- Each list, therefore, contains the value 0, possibly the value 1, and up to

  $$\lfloor \log_{1+\epsilon/2n} t \rfloor$$

  additional values.
- The number of elements in each list L_i is at most

  $$\begin{aligned} &\log_{1+\epsilon/2n} t + 2 \\ &\quad = \frac{\ln t}{\ln(1 + \epsilon/2n)} + 2 \\ &\quad \le \frac{2n(1 + \epsilon/2n)\ln t}{\epsilon} + 2 \\ &\qquad\quad \text{(by inequality (3.23), } x/(1+x) \le \ln(1+x) \text{ with } x = \epsilon/2n) \\ &\quad < \frac{3n \ln t}{\epsilon} + 2\,. \end{aligned}$$

This bound is polynomial in the size of the input—which is the number of bits $\lg t$ needed to represent t plus the number of bits needed to represent the set S, which is in turn polynomial in n—and in $1/\epsilon$.

Since the running time of APPROX-SUBSET-SUM is polynomial in the lengths of the L_i, conclude that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme. ■

Solutions for Chapter 35: Approximation Algorithms

Solution to Exercise 35.1-1

Consider a graph with two vertices and one edge: $G = (\{u, v\}, \{(u, v)\})$. An optimal vertex cover would consist of one vertex, either $\{u\}$ or $\{v\}$. However, APPROX-VERTEX-COVER will always return the vertex cover $\{u, v\}$, which is suboptimal.

Solution to Exercise 35.1-2

In order to prove that the set of edges A picked in line 4 of APPROX-VERTEX-COVER forms a maximal matching in G, we will first show that A forms a matching.

A matching M is a subset of E such that no two edges in M share a common vertex. When an edge (u, v) is picked from E' in line 4, all edges incident on u or v are removed from E'. Therefore, edges in A cannot share a common vertex, and A is a matching.

A maximal matching is a matching that cannot be extended. Assume for sake of contradiction that when we exit the **while** loop of lines 3–6 and are done adding edges to A, there is another edge $(x, y) \in E$ that we could have added to extend the matching A. If we could extend A, then (x, y) does not share endpoints with any edges in A. Then we have $(x, y) \in E'$, but also $E' = \emptyset$, and so we have reached a contradiction. Therefore, A must be a maximal matching.

Solution to Exercise 35.2-1

If G satisfies the triangle inequality, then we have $c(u, w) \leq c(u, v) + c(v, w)$ for all vertices $u, v, w \in V$.

Assume for sake of contradiction that there is an edge (x, y) with $c(x, y) < 0$. Pick an arbitrary vertex $z \in V$. We must have $c(z, y) \leq c(z, x) + c(x, y)$. This inequality leads to $c(z, y) - c(x, y) \leq c(z, x)$. Since $c(x, y) < 0$, we have $c(z, y) - c(x, y) > c(z, y) + c(x, y)$. Then, $c(z, x) > c(z, y) + c(x, y)$. By the triangle inequality, however, we must also have $c(z, x) \leq c(z, y) + c(x, y)$. Therefore we have reached a contradiction and every edge $(u, v) \in E$ must have $c(u, v) \geq 0$.

Solution to Exercise 35.3-1

Repeatedly select the first word with the most undiscovered letters. First, add `thread` to $\mathcal{C}$, since it has six undiscovered letters. Now, the letters `t`, `h`, `r`, `e`, `a`, and `d` have been found, so look for the next word with the most letters aside from these six letters. The first word (in alphabetical order) with the most undiscovered letters (three) is `lost`, so add it to $\mathcal{C}$. Next add `drain`, and finally add `shun`, for a complete set cover $\mathcal{C} = \{\texttt{thread}, \texttt{lost}, \texttt{drain}, \texttt{shun}\}$.

Solution to Exercise 35.3-2

Verifying that the set-covering problem is in NP is straightforward, since it is easy in polynomial time to verify whether a proposed covering indeed covers all of the elements in X. We can then use a reduction from the vertex-cover problem (see Section 35.1) to complete the proof. Given an instance of the vertex-cover problem as a graph $G = (V, E)$, we let $X = E$ and let $\mathcal{F} = \{E_u : u \in V\}$, where E_u is the set of edges incident on vertex u. Selecting set E_u in the set-covering problem is equivalent to selecting vertex u in the original vertex-cover problem. We have therefore reduced the vertex-cover problem to the set-covering problem, so that the set-covering problem is NP-complete.

Solution to Exercise 35.3-4

Consider each set S added into $\mathcal{C}$ as incurring 1 unit of cost, and distribute this cost over the elements in S that are newly covered. For an element x, denote its portion of the cost by c_x. Since every element is eventually covered, we have $c_x > 0$ for all elements x when GREEDY-SET-COVER terminates. Since each iteration of GREEDY-SET-COVER adds one set into $\mathcal{C}$, we have $|\mathcal{C}| = \sum_{x \in X} c_x$. And since every element in X makes it into at least one set in the optimal set cover $\mathcal{C}^*$, we have $\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$.

Now consider any set $S \in \mathcal{F}$. Since each element $x \in S$ has at most 1 unit of cost, we have $\sum_{x \in S} c_x \leq |S|$. Since S is any set in $\mathcal{F}$, we have that $|S| \leq \max\{|S| : S \in \mathcal{F}\}$.

Thus, we have

$$\begin{aligned}
|\mathcal{C}| &= \sum_{x \in X} c_x \\
&\leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x
\end{aligned}$$

$$\begin{aligned} &\le \sum_{S \in \mathcal{C}^*} |S| \\ &\le \sum_{S \in \mathcal{C}^*} \max\{|S| : S \in \mathcal{F}\} \\ &= |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\} \ . \end{aligned}$$

Solution to Exercise 35.4-4

Assume for sake of contradiction that we have an optimal solution x to the linear program in which there is a vertex $u \in V$ that has $x(u) > 1$.

Assume that we have another solution y to the linear program for which $y(u) = 1$, and $y(v) = x(v)$ for all other vertices $v \in V$.

The solution y is feasible since it satisfies all of the constraints of the linear program. Since y is feasible, $y(v) \le 1$ and $y(v) \ge 0$ for each $v \in V$. For every edge (u, v) incident on u, we have $y(u) + y(v) \ge 1$ since $y(u) = 1$.

Thus, we have

$$\begin{aligned} \sum_{v \in V} w(v)x(v) &= w(u)x(u) + \sum_{v \in V - \{u\}} w(u)y(u) \\ &> w(u)y(u) + \sum_{v \in V - \{u\}} w(u)y(u) \\ &= \sum_{v \in V} w(v)y(v) \ , \end{aligned}$$

and so x is not an optimal solution, since y results in a smaller value of the objective function. Therefore, we must have $x(v) \le 1$ for all $v \in V$.

Solution to Problem 35-4

a. Consider a path of four vertices: $G = (V, E)$, with $V = \{a, b, c, d\}$ and $E = \{(a, b), (b, c), (c, d)\}$. The set $\{(b, c)\}$ is a maximal matching, while $\{(a, b), (c, d)\}$ is a maximum matching.

b. The following procedure returns a maximal matching in G.

```
MAXIMAL-MATCHING(G)
  $M = \emptyset$
  for each vertex $v \in G.V$
      $v.matched$ = FALSE
  for each edge $(u, v) \in G.E$
      if $v.matched$ == FALSE and $u.matched$ == FALSE
          $M = M \cup \{(u, v)\}$
          $v.matched$ = TRUE
          $u.matched$ = TRUE
  return $M$
```

The final set M is a maximal matching because for every edge $e' \notin M$, the set $M \cup \{e'\}$ is not a matching.

In order for the running time to be $O(E)$, we must have $|V| = O(E)$, which is the case because G is connected. Then, since the algorithm does a constant amount of work per each edge and each vertex, and G is connected, its running time is $O(V + E) = O(E)$.

c. A vertex cover C must cover each edge in the maximum (or any) matching M. That is, C must contain one endpoint from each edge in M, so that the number of vertices in C must be at least the number of edges in M. More vertices may be necessary to cover the edges not in M, so we have shown a lower bound.

d. This induced subgraph—let's call it I—has no edges, consisting only of isolated vertices.

Assume for sake of contradiction that I had an edge e. Then, we could have added e to M to form a matching which is a proper superset of M, thus contradicting the maximality of M.

e. From part (d), each edge in the graph G is incident to some vertex in T. Therefore, T is a vertex cover for G.

Since no two edges in M share an endpoint and T contains two vertices for each edge in M, the size of T is $|T| = 2\,|M|$.

f. Suppose the greedy algorithm in part (b) finds a maximal matching M'. Then by part (e), a vertex cover for G has size $2\,|M'|$. Now, let M^* be a maximum matching for G. By part (c), we have $2\,|M'| \geq |M^*|$, and so the greedy algorithm in part (b) is a 2-approximation algorithm for finding a maximum matching.

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.