
Lecture Notes

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

Published by the MIT Press. Copyright © 2022 by The Massachusetts Institute of Technology. All rights reserved.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The MIT Press, including, but not limited to, network or other electronic storage or transmission, or broadcast for distance learning.

Contents

Revision History	<i>R-1</i>
Preface	<i>P-1</i>
Chapter 2: Getting Started	
Lecture Notes	<i>2-1</i>
Chapter 3: Characterizing Running Times	
Lecture Notes	<i>3-1</i>
Chapter 4: Divide-and-Conquer	
Lecture Notes	<i>4-1</i>
Chapter 5: Probabilistic Analysis and Randomized Algorithms	
Lecture Notes	<i>5-1</i>
Chapter 6: Heapsort	
Lecture Notes	<i>6-1</i>
Chapter 7: Quicksort	
Lecture Notes	<i>7-1</i>
Chapter 8: Sorting in Linear Time	
Lecture Notes	<i>8-1</i>
Chapter 9: Medians and Order Statistics	
Lecture Notes	<i>9-1</i>
Chapter 10: Elementary Data Structures	
Lecture Notes	<i>10-1</i>
Chapter 11: Hash Tables	
Lecture Notes	<i>11-1</i>
Chapter 12: Binary Search Trees	
Lecture Notes	<i>12-1</i>
Chapter 13: Red-Black Trees	
Lecture Notes	<i>13-1</i>
Chapter 14: Dynamic Programming	
Lecture Notes	<i>14-1</i>
Chapter 15: Greedy Algorithms	
Lecture Notes	<i>15-1</i>
Chapter 16: Amortized Analysis	
Lecture Notes	<i>16-1</i>
Chapter 17: Augmenting Data Structures	
Lecture Notes	<i>17-1</i>
Chapter 19: Data Structures for Disjoint Sets	
Lecture Notes	<i>19-1</i>

Chapter 20: Elementary Graph Algorithms

Lecture Notes 20-1

Chapter 21: Minimum Spanning Trees

Lecture Notes 21-1

Chapter 22: Single-Source Shortest Paths

Lecture Notes 22-1

Chapter 23: All-Pairs Shortest Paths

Lecture Notes 23-1

Chapter 24: Maximum Flow

Lecture Notes 24-1

Chapter 25: Matchings in Bipartite Graphs

Lecture Notes 25-1

Chapter 30: Polynomials and the FFT

Lecture Notes 30-1

Chapter 32: String Matching

Lecture Notes 32-1

Chapter 35: Approximation Algorithms

Lecture Notes 35-1

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 contains lecture notes 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 for every chapter. Future revisions of this document may include additional material.

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.

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 `clrscod4e` package for \LaTeX 2 ϵ 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/clrscod4e.sty and its documentation at https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/11599/clrscod4e.pdf. Make sure to use the `clrscod4e` package, not the `clrscod` or `clrscod3e` 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 $\langle a_1, a_2, \dots, a_n \rangle$.

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

The sequences are typically stored in arrays.

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

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

Expressing algorithms

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

English is sometimes the best way.

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

- Pseudocode is similar to C, C++, 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)

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 $A[j] > key$

$A[j + 1] = A[j]$

$j = j - 1$

$A[j + 1] = key$

cost *times*

c_1 n

c_2 $n - 1$

0 $n - 1$

c_4 $n - 1$

c_5 $\sum_{i=2}^n t_i$

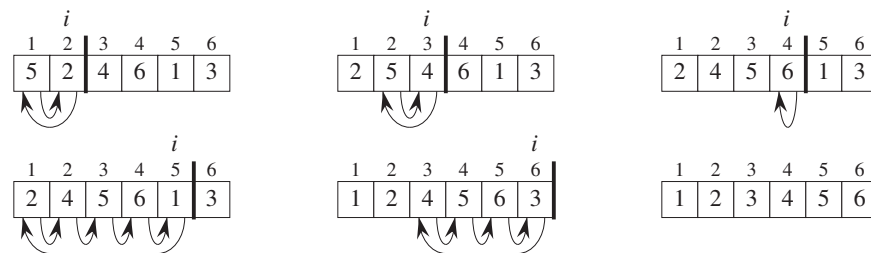
c_6 $\sum_{i=2}^n (t_i - 1)$

c_7 $\sum_{i=2}^n (t_i - 1)$

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 $x \rightarrow attr$ 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 k th line takes time c_k , which is a constant. (Since the third line is a comment, it takes no time.)
- For $i = 2, 3, \dots, n$, 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.

$$\begin{aligned} T(n) = & c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^n t_i + c_6 \sum_{i=2}^n (t_i - 1) \\ & + c_7 \sum_{i=2}^n (t_i - 1) + c_8(n-1) . \end{aligned}$$

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 $A[i] \leq \text{key}$ upon the first time the **while** loop test is run (when $i = i - 1$).
- All t_i are 1.
- Running time is

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- 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 $A[i] > \text{key}$ in while loop test.
- Have to compare key with all elements to the left of the i th position \Rightarrow compare with $i - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $i - 1$ tests $\Rightarrow t_i = i$.
- $\sum_{i=2}^n t_i = \sum_{i=2}^n i$ 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 $\frac{n(n+1)}{2}$.
- Since $\sum_{i=2}^n i = \left(\sum_{i=1}^n i \right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

[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 $\sum_{i=2}^n (i - 1) = \sum_{l=1}^{n-1} l = \frac{n(n-1)}{2}$.
- Running time is

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

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

Worst-case and average-case analysis

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

Reasons

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.

- Why not analyze the average case? Because it's often about as bad as the worst case.

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

On average, the key in $A[i]$ 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 $\text{MERGE}(A, p, q, r)$.

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

$\text{MERGE-SORT}(A, p, r)$

```

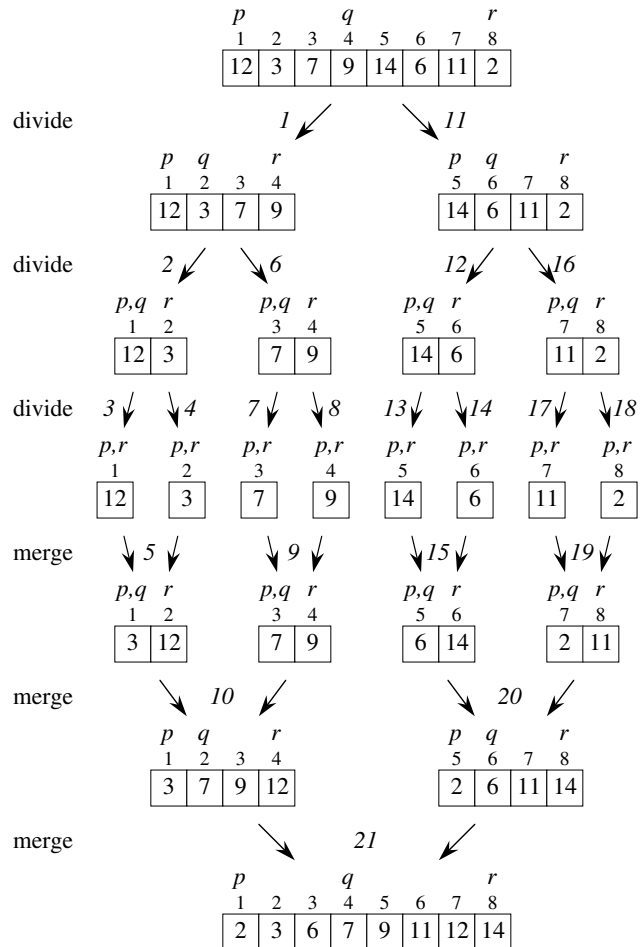
if  $p \geq r$                                 // zero or one element?
    return
 $q = \lfloor (p + r)/2 \rfloor$                       // midpoint of  $A[p:r]$ 
 $\text{MERGE-SORT}(A, p, q)$                       // recursively sort  $A[p:q]$ 
 $\text{MERGE-SORT}(A, q + 1, r)$                   //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

Initial call: $\text{MERGE-SORT}(A, 1, n)$

[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.]

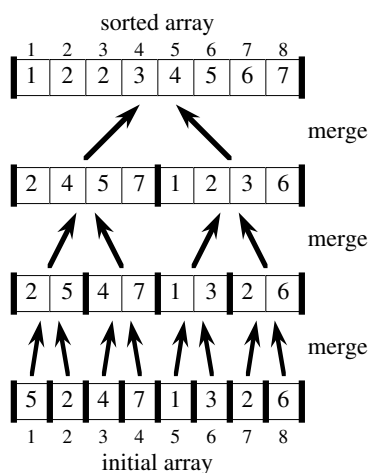
Example

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($A, 1, 8$).]



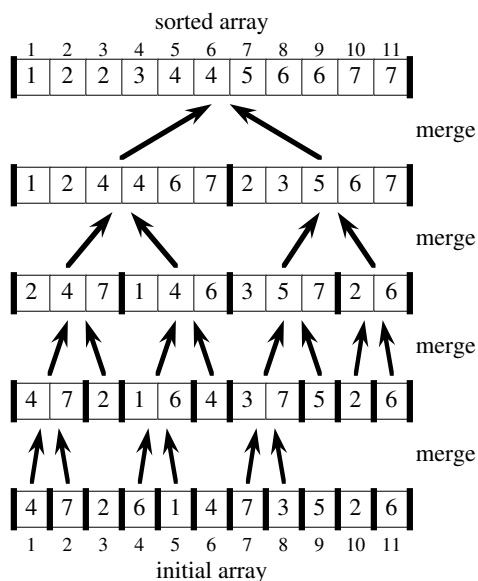
Example

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

- $p \leq q < r$.
- 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 $\leq n$ 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 procedure, 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

```

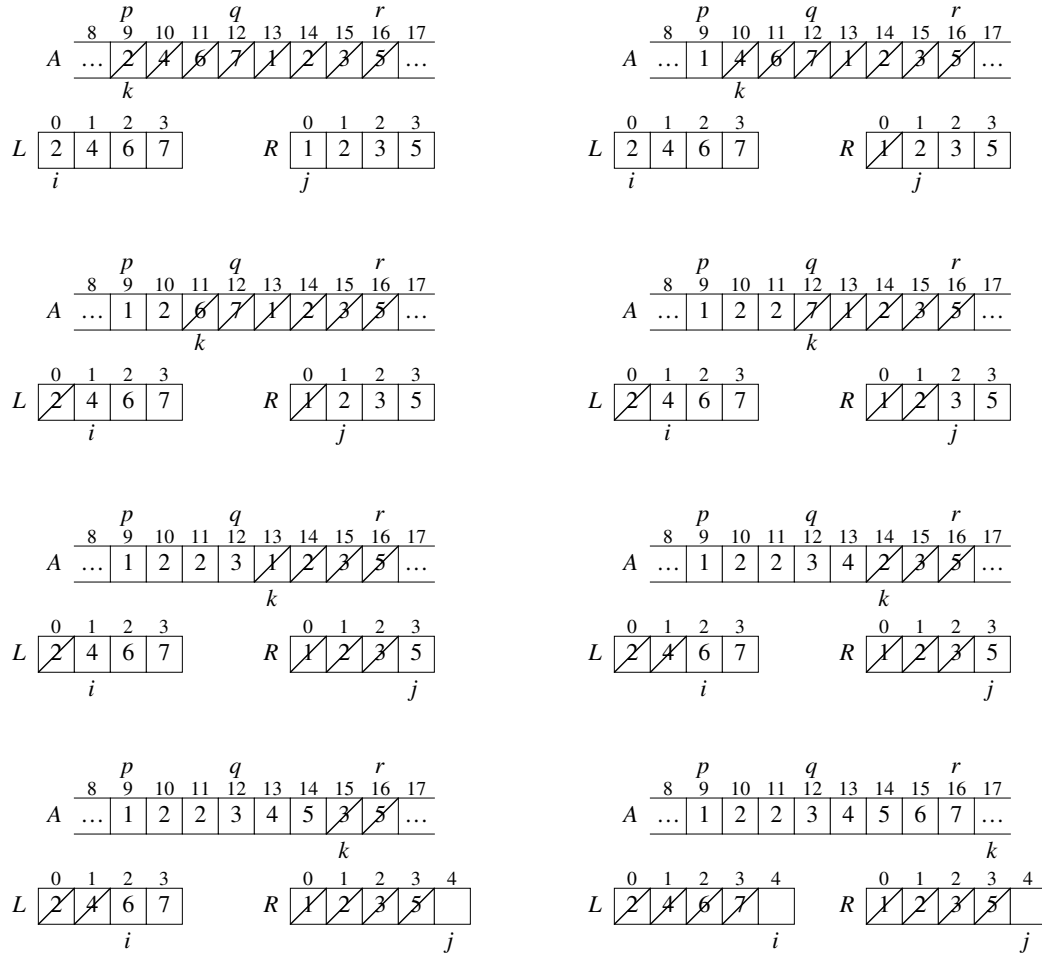
MERGE( $A, p, q, r$ )
   $n_L = q - p + 1$       // length of  $A[p : q]$ 
   $n_R = r - q$           // length of  $A[q + 1 : r]$ 
  let  $L[0 : n_L - 1]$  and  $R[0 : n_R - 1]$  be new arrays
  for  $i = 0$  to  $n_L - 1$  // copy  $A[p : q]$  into  $L[0 : n_L - 1]$ 
     $L[i] = A[p + i]$ 
  for  $j = 0$  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  $L[i] \leq R[j]$ 
       $A[k] = L[i]$ 
       $i = i + 1$ 
    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, $n \leq n_0$ 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

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

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 $n \geq 2$, time for merge sort steps:

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

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

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

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

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

Solving the merge-sort recurrence

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

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

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

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

- Each level except the bottom has cost c_2n .
 - The top level has cost c_2n .
 - The next level down has 2 subproblems, each contributing cost $c_2n/2$.
 - The next level has 4 subproblems, each contributing cost $c_2n/4$.
 - 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 $c_2n \Rightarrow c_2n \lg n$. The bottom level has n leaves in the recursion tree, each costing $c_1 \Rightarrow c_1n$.
- Total cost is $c_2n \lg n + c_1n$. Ignore low-order term of c_1n and constant coefficient $c_2 \Rightarrow \Theta(n \lg n)$.

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:

$$\begin{array}{lll} O & \approx & \leq \\ \Omega & \approx & \geq \\ \Theta & \approx & = \\ o & \approx & < \\ \omega & \approx & > \end{array}$$

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 .

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

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  $A[j] > key$ 
         $A[j + 1] = A[j]$ 
         $j = j - 1$ 
     $A[j + 1] = key$ 

```

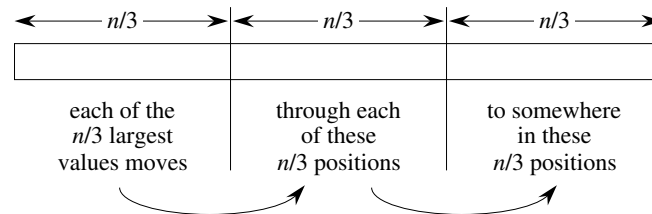
First, show that INSERTION-SORT runs in $O(n^2)$ time, regardless of the input:

- 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.



- 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 $A[n/3 + 1 : 2n/3]$.

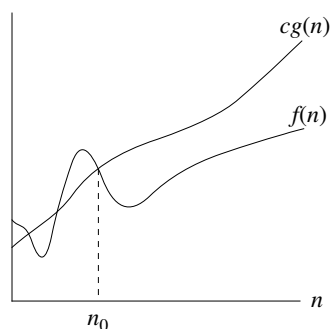
Because at least $n/3$ values must pass through at least $n/3$ positions, the line $A[j + 1] = A[j]$ executes at least $(n/3)(n/3) = n^2/9$ times, which is $\Omega(n^2)$. For this input, INSERTION-SORT takes time $\Omega(n^2)$.

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

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



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

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

Example

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

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

$$n^2$$

$$n^2 + n$$

$$n^2 + 1000n$$

$$1000n^2 + 1000n$$

Also,

$$n$$

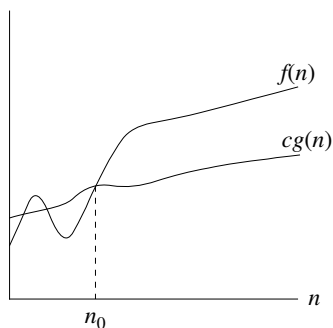
$$n/1000$$

$$n^{1.99999}$$

$$n^2 / \lg \lg \lg n$$

Ω -notation

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



$g(n)$ 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,

$$n^3$$

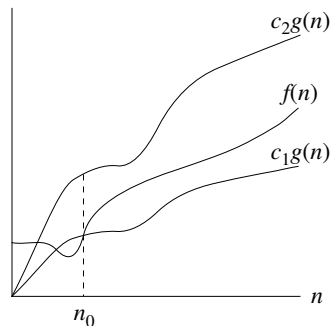
$$n^{2.00001}$$

$$n^2 \lg \lg \lg n$$

$$2^{2^n}$$

Θ -notation

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$.



$g(n)$ 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:

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

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

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:

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) \\ = \Theta(n^2) .$$

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: $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

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

 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: $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$.

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

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:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)).$$

$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).$$

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 \leq , Ω to \geq , 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 $m \leq n \Rightarrow f(m) \leq f(n)$.
- $f(n)$ is *monotonically decreasing* if $m \geq n \Rightarrow f(m) \geq f(n)$.
- $f(n)$ 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:

$$a^{-1} = 1/a,$$

$$(a^m)^n = a^{mn},$$

$$a^m a^n = a^{m+n}.$$

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

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

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

A suprisingly useful inequality: for all real x ,

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

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

Logarithms

Notations:

$$\lg n = \log_2 n \quad (\text{binary logarithm}) ,$$

$$\ln n = \log_e n \quad (\text{natural logarithm}) ,$$

$$\lg^k n = (\lg n)^k \quad (\text{exponentiation}) ,$$

$$\lg \lg n = \lg(\lg n) \quad (\text{composition}) .$$

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 \Rightarrow the expression is strictly increasing as a increases.
- Hold a constant \Rightarrow 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:

$$a = b^{\log_b a} ,$$

$$\log_c(ab) = \log_c a + \log_c b ,$$

$$\log_b a^n = n \log_b a ,$$

$$\log_b a = \frac{\log_c a}{\log_c b} ,$$

$$\log_b(1/a) = -\log_b a ,$$

$$\log_b a = \frac{1}{\log_a b} ,$$

$$a^{\log_b c} = c^{\log_b a} .$$

[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 $\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0$, substitute $\lg n$ for n and 2^a for a :

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

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

Factorials

$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$. 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 $\lg(n!) = \Theta(n \lg n)$.

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: $T(n) \leq 2T(n/2) + \Theta(n)$ 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 $i, j = 1, 2, \dots, n$.

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

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21},$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22},$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21},$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.$$

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 recurrence 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, \dots, 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, \dots, 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, \dots, S_{10} , recursively compute P_1, P_2, \dots, 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 \Rightarrow 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.]

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

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 algorithm 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 $T(n) \leq 2T(n/2) + cn$ 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: $T(n) \leq dn \lg n$ 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:

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

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

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

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

Substitution:

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

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 $T(n) \leq cn$ for $n \geq n_0$, where we choose c, n_0 :

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

But this doesn't say that $T(n) \leq cn$ 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: $T(n) \leq cn - d$, where $d \geq 0$ is a constant:

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

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 $T(n) \leq cn - d - (d - \Theta(1))$, 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 $T(n) = O(n)$:

$$\begin{aligned} T(n) &\leq 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n) . \quad \Leftarrow \text{wrong!} \end{aligned}$$

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

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

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

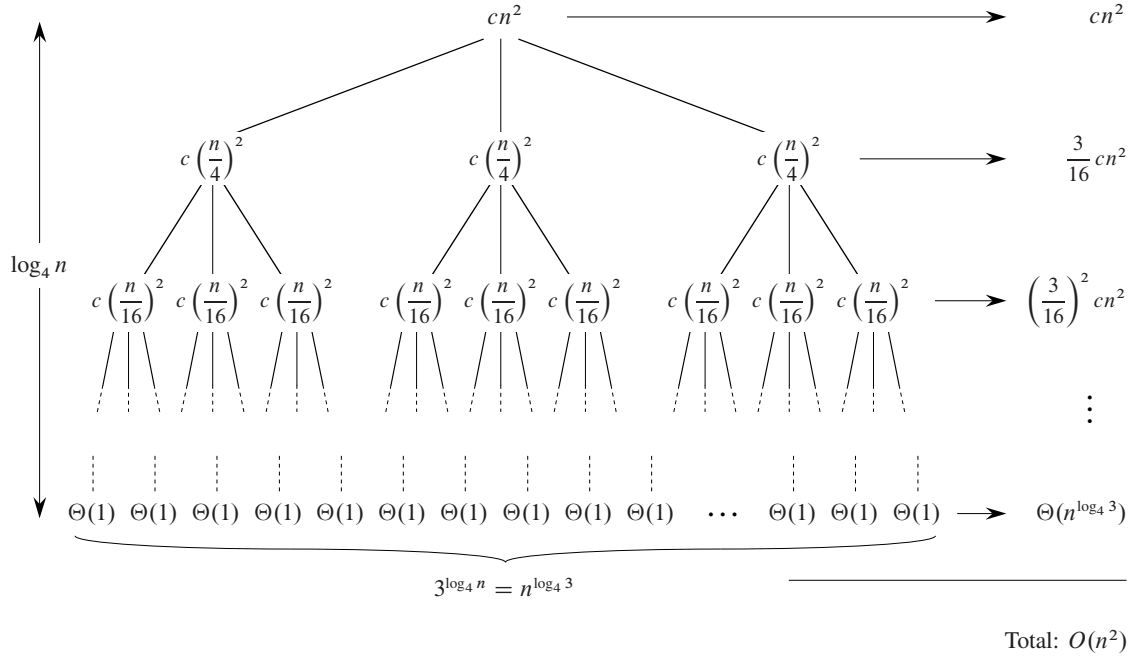
Recursion trees

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 $T(n) = 3T(n/4) + cn^2$:



[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 $T(n) \leq dn^2$ for constant $d > 0$:

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

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

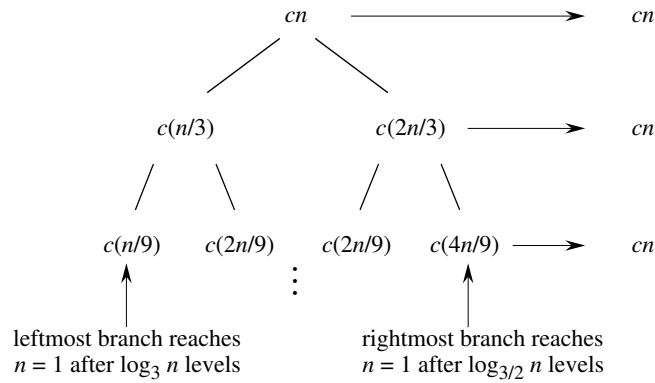
That gives an upper bound of $O(n^2)$. The lower bound of $\Omega(n^2)$ is obvious because the recurrence contains a $\Theta(n^2)$ term. Hence, $T(n) = \Theta(n^2)$.

Irregular example

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

For upper bound, rewrite as $T(n) \leq T(n/3) + T(2n/3) + cn$; for lower bound, as $T(n) \geq T(n/3) + T(2n/3) + cn$.

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 $\leq cn$.
- Lower bound guess: $\geq dn \log_3 n = \Omega(n \lg n)$ for some positive constant d .
- Upper bound guess: $\leq dn \log_{3/2} n = O(n \lg n)$ for some positive constant d .
- Then *prove* by substitution.

1. Upper bound:

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

Substitution:

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

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

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: $T(n) \geq dn \lg n$.

Substitution: Same as for the upper bound, but replacing \leq by \geq . End up needing

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

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 $a \geq 1$, $b > 1$, 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 $a', a'' \geq 0$ 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 $a', a'' \geq 0$ satisfying $a = a' + a''$.

[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 $n^{\lg 2} = n$ 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]

- $T(n) = 5T(n/2) + \Theta(n^2)$
 $n^{\lg 5}$ vs. n^2
 Since $\lg 5 - \epsilon = 2$ for some constant $\epsilon > 0$, use case 1 $\Rightarrow T(n) = \Theta(n^{\lg 5})$
- $T(n) = 27T(n/3) + \Theta(n^3 \lg n)$
 $n^{\lg 3} = n^3$ vs. $n^3 \lg n$
 Use case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n)$
- $T(n) = 5T(n/2) + \Theta(n^3)$
 $n^{\lg 5}$ 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 $\Rightarrow T(n) = \Theta(n^3)$
- $T(n) = 27T(n/3) + \Theta(n^3/\lg n)$
 $n^{\lg 3} = n^3$ 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 $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.]

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: $rank(i)$ is a unique integer in the range 1 to n .
- The ordered list $\langle rank(1), rank(2), \dots, rank(n) \rangle$ is a permutation of the candidate numbers $\langle 1, 2, \dots, n \rangle$.
- 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**.

- $\text{RANDOM}(a, b)$ returns an integer r , where $a \leq r \leq b$ 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**

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

Lemma

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

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

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

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 $X_H = I\{H\}$. X_H counts the number of heads in one flip.
- Since $\Pr\{H\} = 1/2$, lemma says that $E[X_H] = 1/2$.

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 $E[X] = \sum_{k=0}^n k \cdot \Pr\{X = k\}$. In fact, this is what the book does in equation (C.41).
- Instead, use indicator random variables.
- For $i = 1, 2, \dots, n$, define $X_i = I\{\text{the } i\text{th flip results in event } H\}$.
- Then $X = \sum_{i=1}^n X_i$.
- Lemma says that $E[X_i] = \Pr\{H\} = 1/2$ for $i = 1, 2, \dots, n$.
- Expected number of heads is $E[X] = E[\sum_{i=1}^n X_i]$.
- **Problem:** We want $E[\sum_{i=1}^n X_i]$. We have only the individual expectations $E[X_1], E[X_2], \dots, E[X_n]$.
- **Solution:** Linearity of expectation (equation (C.24)) says that the expectation of the sum equals the sum of the expectations. Thus,

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

- 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.]

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, \dots, X_n , where

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

Useful properties:

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

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 $1, 2, \dots, i-1$.
- Assumption that the candidates arrive in random order \Rightarrow candidates $1, 2, \dots, i$ arrive in random order \Rightarrow any one of these first i candidates is equally likely to be the best one so far.
- Thus, $\Pr\{\text{candidate } i \text{ is the best so far}\} = 1/i$.
- Which, by the lemma, implies $E[X_i] = 1/i$.

Now compute $E[X]$:

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

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 $1/n$.

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 i th 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 i th iteration, each possible $(i - 1)$ -permutation appears in $A[1 : i - 1]$ with probability $(n - i + 1)!/n!$. Will show that after the i th 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 $\pi = \langle x_1, x_2, \dots, x_i \rangle$. It consists of an $(i - 1)$ -permutation $\pi' = \langle x_1, x_2, \dots, x_{i-1} \rangle$, 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 i th 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)

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  $\lfloor i/2 \rfloor$ 
```

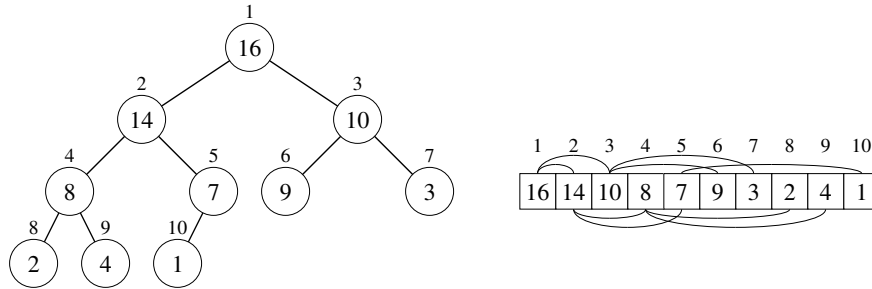
```
LEFT( $i$ )  
  return  $2i$ 
```

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

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

Example

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


```

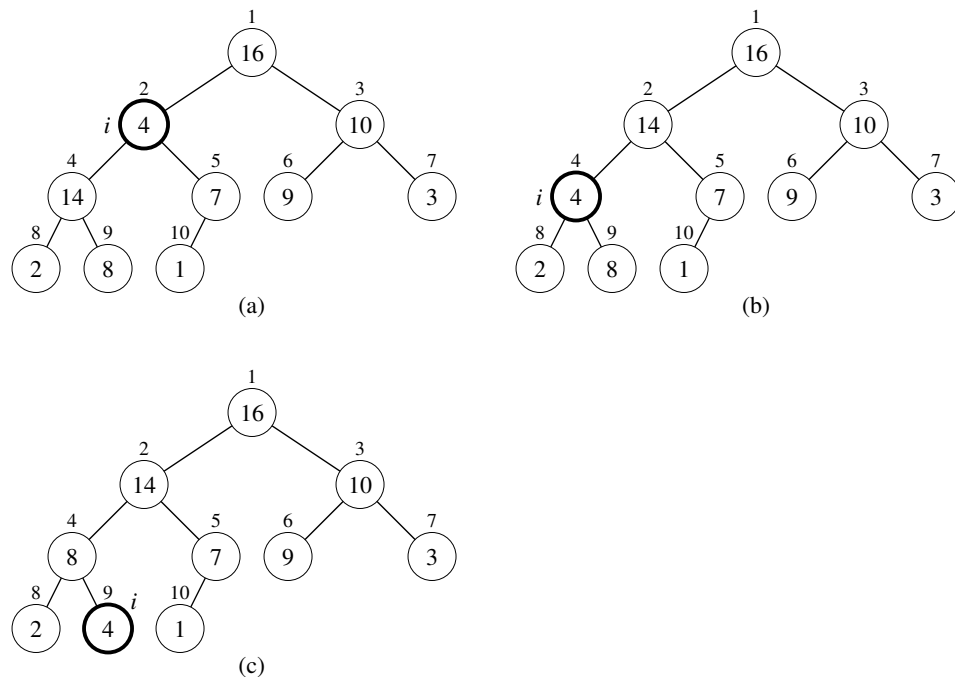
MAX-HEAPIFY( $A, i$ )
   $l = \text{LEFT}(i)$ 
   $r = \text{RIGHT}(i)$ 
  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
     $\text{largest} = l$ 
  else  $\text{largest} = i$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
     $\text{largest} = r$ 
  if  $\text{largest}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

```

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.



- 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 .

```

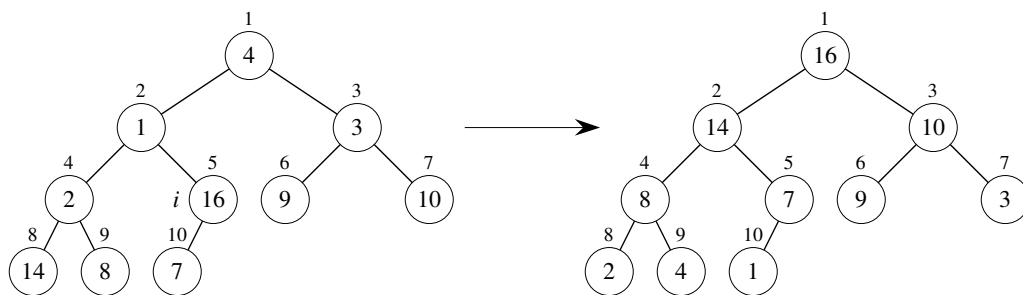
BUILD-MAX-HEAP( $A, n$ )
   $A.heap-size = n$ 
  for  $i = \lfloor n/2 \rfloor$  downto 1
    MAX-HEAPIFY( $A, i$ )
  
```

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.

- $A.heap-size$ 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 $i + 1$, $i + 2, \dots, n$ is root of a max-heap.

Initialization: By Exercise 6.1-8, we know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ 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 $i+1, i+2, \dots, n$ 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 $\leq \lceil n/2^{h+1} \rceil$ 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) $(\sum_{k=0}^{\infty} kx^k)$, 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

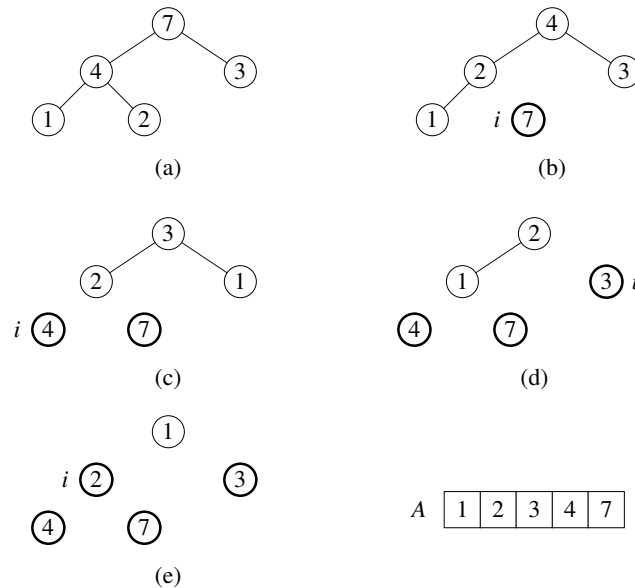
 exchange $A[1]$ with $A[i]$

$A.\text{heap-size} = A.\text{heap-size} - 1$

 MAX-HEAPIFY($A, 1$)

Example

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



Analysis

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

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

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

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:
 - $\text{INSERT}(S, x, k)$: inserts element x with key k into set S .
 - $\text{MAXIMUM}(S)$: returns element of S with largest key.
 - $\text{EXTRACT-MAX}(S)$: removes and returns element of S with largest key.
 - $\text{INCREASE-KEY}(S, x, k)$: 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:
 - $\text{INSERT}(S, x, k)$: inserts element x with key k into set S .
 - $\text{MINIMUM}(S)$: returns element of S with smallest key.
 - $\text{EXTRACT-MIN}(S)$: removes and returns element of S with smallest key.
 - $\text{DECREASE-KEY}(S, x, k)$: 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  $A.heap-size == n$   
    error "heap overflow"  
   $A.heap-size = A.heap-size + 1$   
   $k = x.key$   
   $x.key = -\infty$   
   $A[A.heap-size] = x$   
  map  $x$  to index  $heap-size$  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.

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  $A[q]$ .
     $q = \text{PARTITION}(A, p, r)$ 
    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  $A[j] \leq x$  // 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:

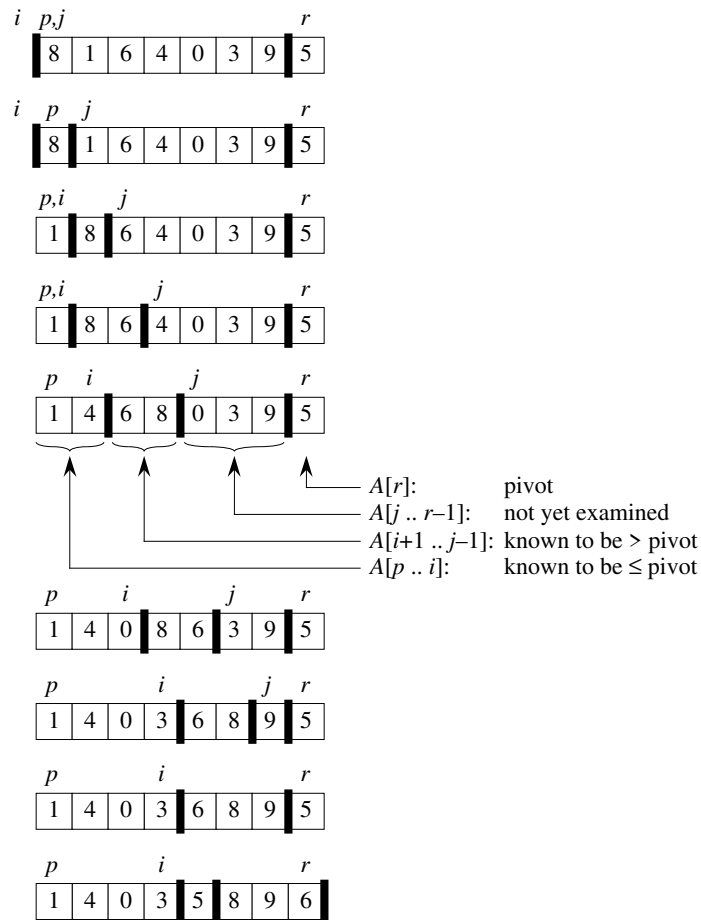
Loop invariant:

1. All entries in $A[p : i]$ are \leq pivot.
2. All entries in $A[i + 1 : j - 1]$ are $>$ pivot.
3. $A[r] = \text{pivot}$.

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 $A[j] \leq \text{pivot}$, then $A[j]$ and $A[i + 1]$ are swapped and then i and j are incremented. If $A[j] > \text{pivot}$, 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: $A[p : i] \leq \text{pivot}$, $A[i + 1 : r - 1] > \text{pivot}$, and $A[r] = \text{pivot}$.

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 $\leq n/2$ elements.
- For an upper bound, get the recurrence

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

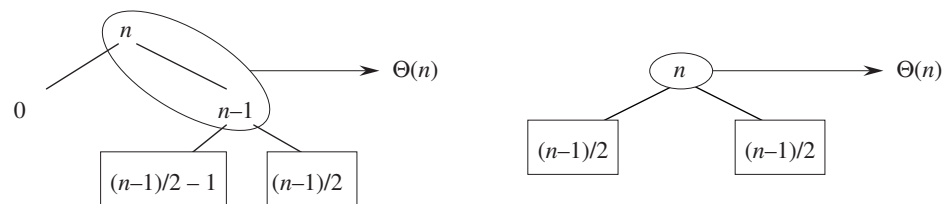
Balanced partitioning

- 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

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

$i = \text{RANDOM}(p, r)$

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 $p < r$

$q = \text{RANDOMIZED-PARTITION}(A, p, r)$

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:

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

- Because PARTITION produces two subproblems, totaling size $n - 1$, q ranges from 0 to $n - 1$.
- **Guess:** $T(n) \leq cn^2$, for some c .
- Substituting our guess into the above recurrence:

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

- 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,

$$\max \{q^2 + (n - q - 1)^2 : 0 \leq q \leq n - 1\} \leq (n - 1)^2 = n^2 - 2n + 1 .$$

- And thus,

$$\begin{aligned} T(n) &\leq cn^2 - c(2n - 1) + \Theta(n) \\ &\leq cn^2 \quad \text{if } c(2n - 1) \geq \Theta(n) . \end{aligned}$$

- 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, \dots, z_n , with z_i being the i th smallest element. (So that the output order is z_1, z_2, \dots, z_n .)
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$ 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 $X_{ij} = \mathbb{I}\{z_i \text{ is compared with } z_j\}$.

(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:

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

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 $E[X]$:

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

Evaluate by using a change in variables ($k = j - i$) and the bound on the harmonic series in equation (A.9):

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

So the expected running time of quicksort, using RANDOMIZED-PARTITION, is $O(n \lg n)$ if all values being sorted are distinct.

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

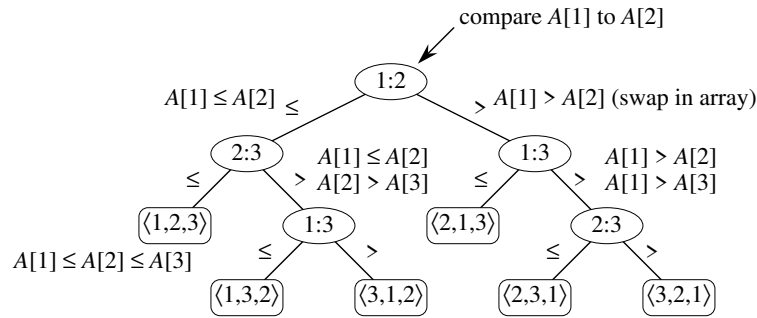
Lower bounds

- $\Omega(n)$ 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 $\{0, 1, \dots, k\}$.

Input: $A[1:n]$, where $A[j] \in \{0, 1, \dots, k\}$ for $j = 1, 2, \dots, n$. 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.

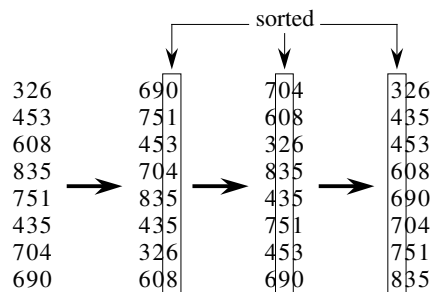
To sort d digits:

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 $1, 2, \dots, i - 1$ are sorted.
- Show that a stable sort on digit i leaves digits $1, \dots, i$ sorted:
 - If two digits in position i are different, ordering by position i is correct, and positions $1, \dots, i - 1$ 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 $0, \dots, k$)
- 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 = $\Theta((b/r)(n + 2^r))$.

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 $b < \lg n$, then choose $r = b \Rightarrow (b/r)(n + 2^r) = \Theta(n)$, which is optimal.

If $b \geq \lg n$, then choosing $r \approx \lg n$ gives $\Theta((b/\lg n)(n + n)) = \Theta(bn/\lg n)$.

- 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 $0 \leq A[i] < 1$ 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 $B[\lfloor n \cdot A[i] \rfloor]$

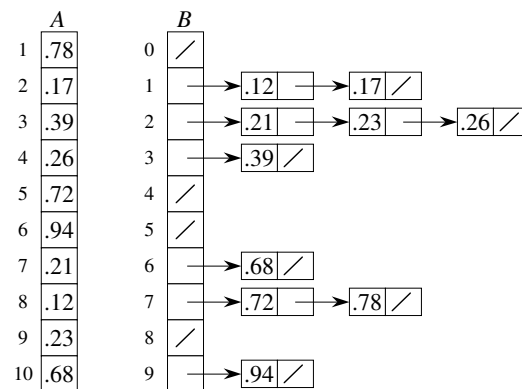
for $i = 0$ **to** $n - 1$

 sort list $B[i]$ with insertion sort

 concatenate lists $B[0], B[1], \dots, B[n-1]$ 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 $A[i] \leq A[j]$. Then $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$. 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:

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \quad (\text{linearity of expectation}) \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad (E[aX] = aE[X]) \end{aligned}$$

Claim

$$E[n_i^2] = 2 - (1/n) \text{ for } i = 0, \dots, n-1.$$

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: $E[n_i] = np = n(1/n) = 1$ and $\text{Var}[n_i] = npq = 1 - 1/n$ (see equations (C.41) and (C.44)). By equation (C.31):

$$\begin{aligned} E[n_i^2] &= \text{Var}[n_i] + E^2[n_i] \\ &= (1 - 1/n) + 1^2 \\ &= 2 - 1/n \end{aligned}$$

■ (claim)

Therefore:

$$\begin{aligned} E[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\ &= \Theta(n) + O(n) \\ &= \Theta(n) \end{aligned}$$

- Again, not a comparison sort. Used a function of key values to index into an array.
- This is a ***probabilistic analysis***—we used probability to analyze an algorithm whose running time depends on the distribution of inputs.
- Different from a ***randomized algorithm***, where we use randomization to *impose* a distribution.
- With bucket sort, if the input isn't drawn from a uniform distribution on $[0, 1)$, 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 $\Theta(n)$.

Lecture Notes for Chapter 9: Medians and Order Statistics

Chapter 9 overview

- ***i th order statistic*** is the i th smallest element of a set of n elements.
- The ***minimum*** is the first order statistic ($i = 1$).
- The ***maximum*** is the n th 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 i th 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 i th 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 i th 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  $i = 1$ 
     $q = \text{RANDOMIZED-PARTITION}(A, p, r)$ 
     $k = q - p + 1$ 
    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 k th element of the subarray $A[p : r]$, where $k = q - p + 1$.
- If the pivot element is the i th smallest element (i.e., $i = k$), return $A[q]$.
- Otherwise, recurse on the subarray containing the i th smallest element.
 - If $i < k$, this subarray is $A[p : q - 1]$, and we want the i th 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 j th 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.
- j th partitioning takes set $A^{(j-1)}$ and produces $A^{(j)}$.
- Assume a 0th “dummy” partitioning that produces $A^{(0)}$.
- j th partitioning is *helpful* if $|A^{(j)}| \leq (3/4)|A^{(j-1)}|$. Not all partitionings are necessarily helpful. Think of a helpful partitioning as a successful Bernoulli trial.

Lemma

A partitioning is helpful with probability $\geq 1/2$.

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:

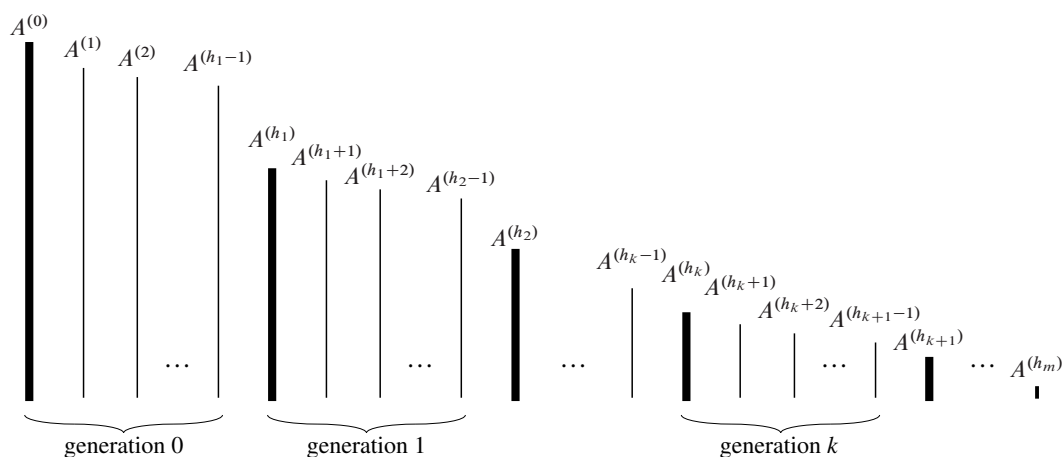
$$\begin{aligned} \frac{2(\lceil n/4 \rceil - 1)}{n} &\leq \frac{2((n/4 + 1) - 1)}{n} \quad (\text{inequality (3.2)}) \\ &= \frac{n/2}{n} \\ &= 1/2. \end{aligned}$$
- Since the pivot has probability $\geq 1/2$ of falling into the middle half, a partitioning is helpful with probability $\geq 1/2$. ■ (lemma)

Theorem

The expected running time of RANDOMIZED-SELECT is $\Theta(n)$.

Proof

- Let the sequence of helpful partitionings be $\langle h_0, h_1, \dots, h_m \rangle$. 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 $k = 1, 2, \dots, m$.
- Iterating gives $n_k \leq (3/4)^k n_0$.
- Break up sets into m “generations.” The sets in generation k are $A^{(h_k)}, A^{(h_{k+1})}, \dots, A^{(h_{k+1}-1)}$, 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 size 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 k th generation, then $|A^{(j)}| \leq |A^{(h_k)}| = n_k \leq (3/4)^k n_0$.
- Define random variable $X_k = h_{k+1} - h_k$ as the number of sets in the k th generation $\Rightarrow k$ th generation includes sets $A^{(h_k)}, A^{(h_k+1)}, \dots, A^{(h_k+X_k-1)}$.
- By previous lemma, a partitioning is helpful with probability $\geq 1/2$. The probability is even higher, since a partitioning is helpful even if the pivot doesn't fall into middle half, but the i th smallest element lies in the smaller side. Just use the $1/2$ lower bound $\Rightarrow E[X_k] \leq 2$ for $k = 0, 1, \dots, m-1$ (by equation (C.36), expectation of a geometric distribution).
- The total running time is dominated by the comparisons during partitioning. The j th partitioning takes $A^{(j-1)}$ and compares the pivot with all the other $|A^{(j-1)}| - 1$ elements $\Rightarrow j$ th partitioning makes $< |A^{(j-1)}|$ comparisons.
- The total number of comparisons is less than

$$\begin{aligned} \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}| &\leq \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(h_k)}| \\ &= \sum_{k=0}^{m-1} X_k |A^{(h_k)}| \\ &\leq \sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0. \end{aligned}$$

- Since $E[X_k] \leq 2$, the expected total number of comparisons is less than

$$\begin{aligned} E \left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0 \right] &= \sum_{k=0}^{m-1} E \left[X_k \left(\frac{3}{4}\right)^k n_0 \right] \quad (\text{linearity of expectation}) \\ &= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k E[X_k] \\ &\leq 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\ &< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\ &= 8n_0 \quad (\text{infinite geometric series}). \end{aligned}$$

- 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 i th 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.
   $x = \text{SELECT}(A, p + 2g, p + 3g - 1, \lceil g/2 \rceil)$ 
   $q = \text{PARTITION-AROUND}(A, p, r, x)$  // 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( $A, q + 1, r, i - k$ )

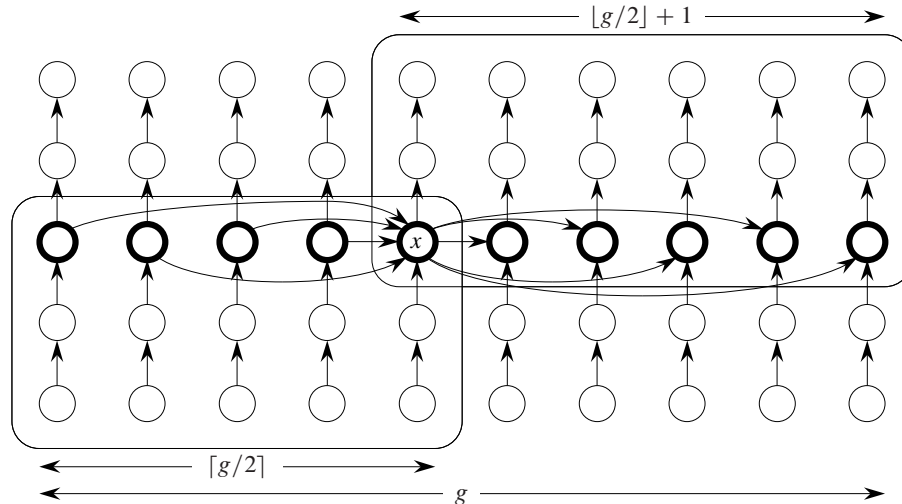
```

The algorithm works with groups of 5 elements. If n is not a multiple of 5, the beginning **while** loop removes $n \bmod 5$ elements from consideration. If $i \leq n \bmod 5$, then the i th iteration of the **while** loop identifies and returns the i th 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 $\langle A[p], A[p + g], A[p + 2g], A[p + 3g], A[p + 4g] \rangle$.
 - 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 i th 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 $g \leq n/5$.
- 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 $T(g) \leq T(n/5)$. 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 $g \leq n/5$ 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 $\geq x$. The lower-left region contains elements known to be $\leq x$. Pivot x is in both regions.
 - The upper-right region contains $\lfloor g/2 \rfloor + 1$ groups \Rightarrow at least $3(\lfloor g/2 \rfloor + 1) \geq 3g/2$ elements are $\geq x$.
 - The lower-left region contains $\lceil g/2 \rceil$ groups \Rightarrow at least $3\lceil g/2 \rceil$ elements are $\leq x$.
 - Either way, the recursive call excludes $\geq 3g/2$ elements, leaving at most $5g - 3g/2 = 7g/2 \leq 7n/10$ elements.
- Get the recurrence $T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$.
- Prove that $T(n) \leq cn$ by substitution for suitably large constant c .
- Assuming that $n \geq 5$ gives

$$\begin{aligned}
 T(n) &\leq c(n/5) + c(7n/10) + \Theta(n) \\
 &\leq 9cn/10 + \Theta(n) \\
 &= cn - cn/10 + \Theta(n) \\
 &\leq cn
 \end{aligned}$$

if $cn/10$ dominates the constant in the $\Theta(n)$ term. Also need to pick c large enough so that $T(n) \leq cn$ for $n \leq 4$ (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.

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 i th 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

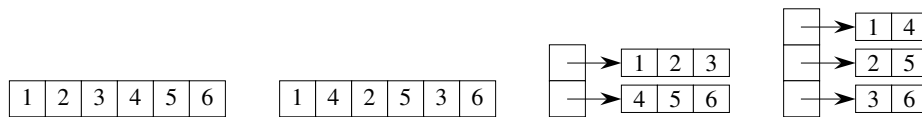
$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}. \quad (*)$$

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

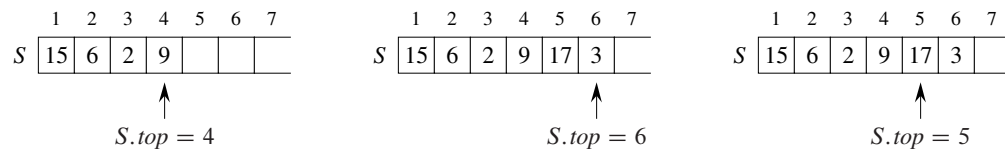
Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified.

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.

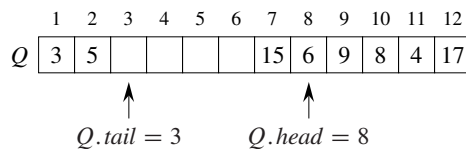
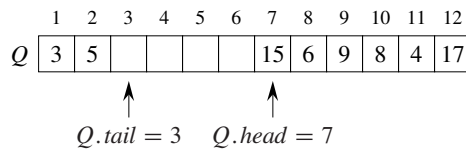
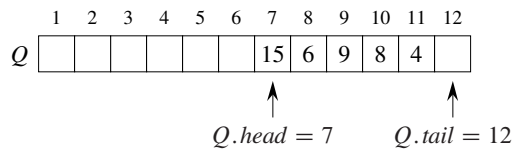
Queues

Inserting into a queue is **enqueueing**, and deleting from a queue is **dequeueing**.

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 a new element will be inserted into the queue.
- Elements reside in $Q.head, Q.head + 1, \dots, Q.tail - 1$, 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:

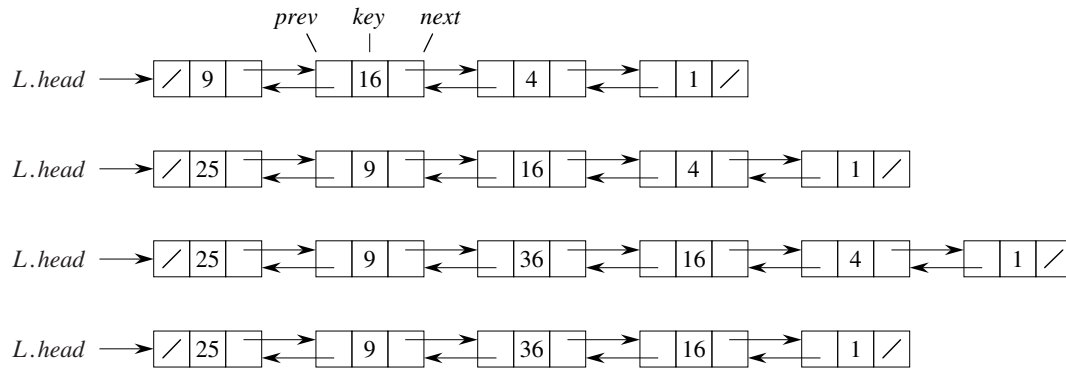
- $x.key$
- $x.next$: the successor of x , NIL if x has no successor so that it's the *tail*
- $x.prev$: the predecessor of x , NIL if x has no predecessor so that it's the *head*

$L.head$ 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 $x \neq \text{NIL}$ 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)

$x.next = L.head$

$x.prev = \text{NIL}$

if $L.head \neq \text{NIL}$

$L.head.prev = x$

$L.head = x$

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  $y.next \neq \text{NIL}$ 
     $y.next.prev = x$ 
   $y.next = x$ 

```

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  $x.prev \neq \text{NIL}$ 
     $x.prev.next = x.next$ 
  else  $L.head = x.next$ 
  if  $x.next \neq \text{NIL}$ 
     $x.next.prev = x.prev$ 

```

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 $\Theta(n)$.

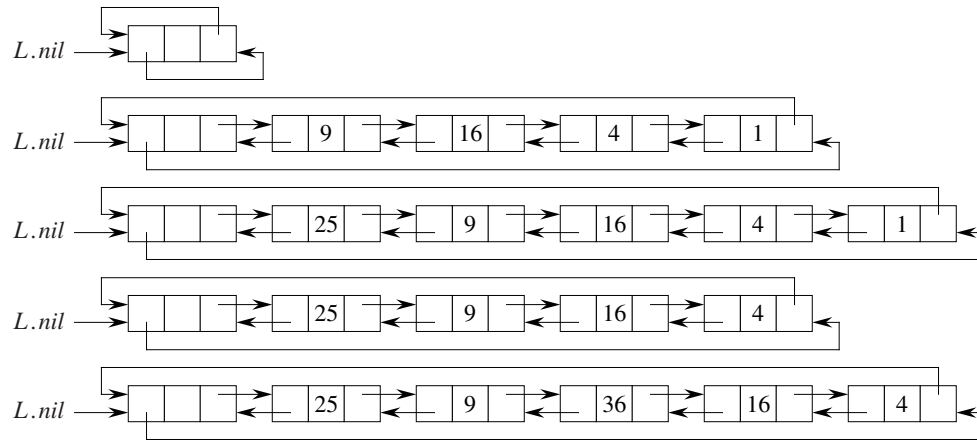
Linked list and array performance

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 k th 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?

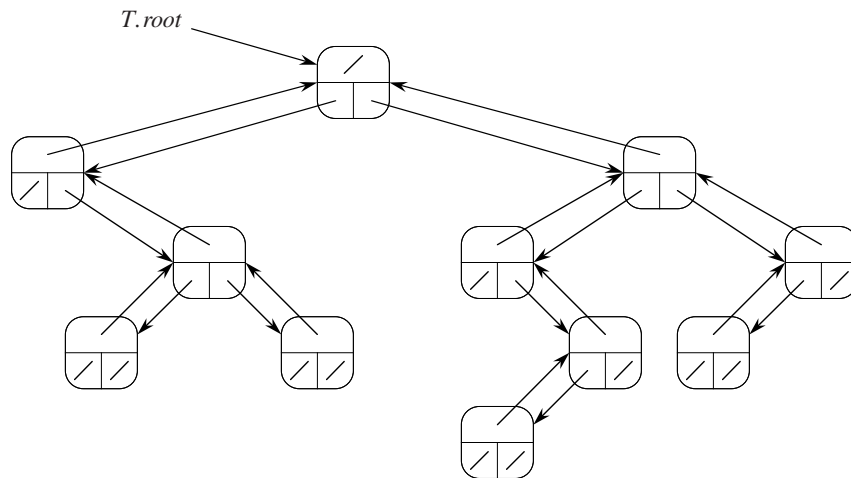
Represent a rooted tree by linked data structures. Each node of a tree is an object with a *key* attribute, like linked lists, and also has attributes that are pointers to other nodes.

Binary trees

Give each node in a binary tree the following attributes: p (parent), $left$, $right$.

If $x.p = \text{NIL}$, x is the root. The root of tree T is $T.root$. If $T.root = \text{NIL}$, then T is empty.

If x has no left child, then $x.left = \text{NIL}$. 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

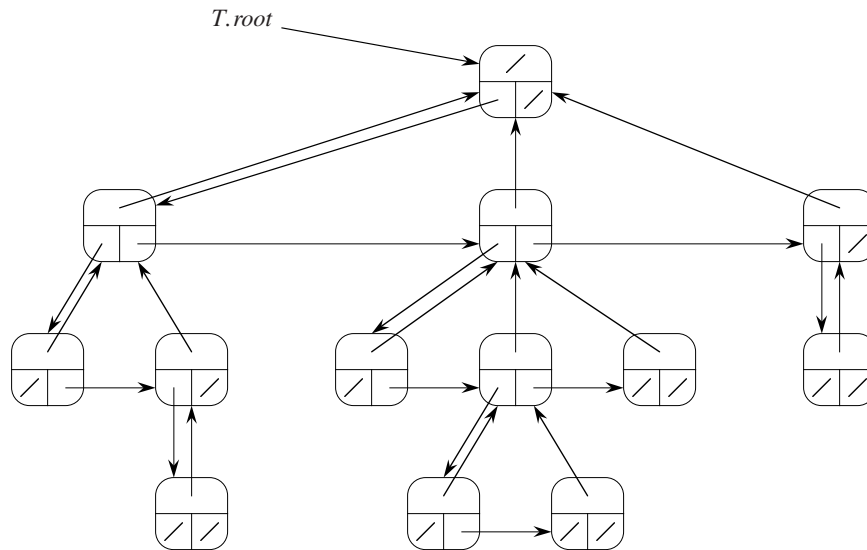
In a binary tree, each node has at most two children. Can extend this representation to let each node have at most k children: replace the *left* and *right* attributes by $child_1, child_2, \dots, child_k$.

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. $x.\text{left-child}$ points to the leftmost child of node x , and
2. $x.\text{right-sibling}$ points to the sibling of x immediately to its right.



Each node x has the attributes $x.p$ (top), $x.\text{left-child}$ (lower left), $x.\text{right-sibling}$ (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.

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 k th 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.

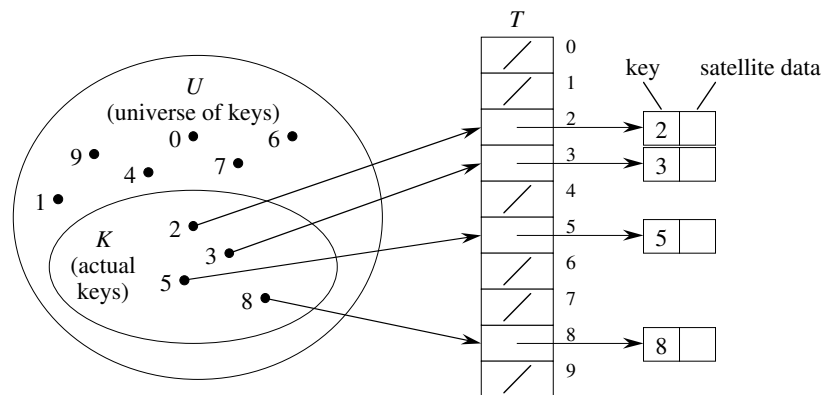
Direct-address tables

Scenario

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \dots, m-1\}$ where m isn't too large.
- No two elements have the same key.

Represent by a **direct-address table**, or array, $T[0 \dots m-1]$:

- 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)

$T[x.key] = \text{NIL}$

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

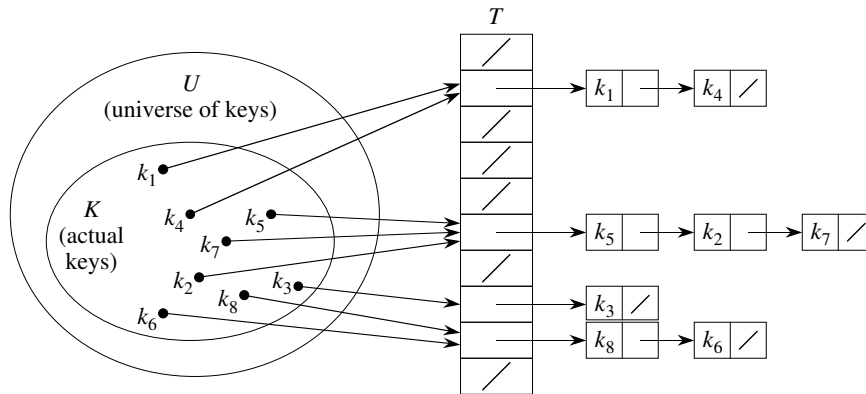
- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing. We'll examine both.

Independent uniform hashing

- Ideally, $h(k)$ would be randomly and independently chosen uniformly from $\{0, 1, \dots, m-1\}$. Once $h(k)$ is chosen, each subsequent evaluation of $h(k)$ must yield the same result.
- Such an ideal 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 $|n/m|$ 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 $j = 0, 1, \dots, m-1$, denote the length of list $T[j]$ by n_j , so that $n = n_0 + n_1 + \dots + n_{m-1}$.
- Expected value of n_j is $E[n_j] = \alpha = n/m$.
- 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 $E[n_{h(k)}] = \alpha$. 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 $i = 1, 2, \dots, n$, let x_i be the i th element inserted into the table, and let $k_i = x_i.\text{key}$.

For each slot q and each pair of distinct keys k_i and k_j , define indicator random variable $X_{ijq} = \mathbf{I}\{\text{the search is for } x_i \text{ and } h(k_i) = h(k_j) = q\}$. $X_{ijq} = 1$ 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

$$\Rightarrow \Pr\{X_{ijq} = 1\} = 1/nm^2$$

$$\Rightarrow \mathbb{E}[X_{ijq}] = 1/nm^2 \text{ (by Lemma 5.1).}$$

For each element x_j , define indicator random variable

$$Y_j = \mathbf{I}\{x_j \text{ appears in a list prior to the element being searched for}\}$$

$$= \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}.$$

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 $\mathbb{E}[Z + 1]$:

$$\begin{aligned} \mathbb{E}[Z + 1] &= \mathbb{E}\left[1 + \sum_{j=1}^n Y_j\right] \\ &= 1 + \mathbb{E}\left[\sum_{j=1}^n \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}\right] \end{aligned}$$

$$\begin{aligned}
&= 1 + \mathbb{E} \left[\sum_{q=0}^{m-1} \sum_{j=1}^n \sum_{i=1}^{j-1} X_{ijq} \right] \\
&= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^n \sum_{i=1}^{j-1} \mathbb{E}[X_{ijq}] \quad (\text{linearity of expectation}) \\
&= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^n \sum_{i=1}^{j-1} \frac{1}{nm^2} \\
&= 1 + m \cdot \binom{n}{2} \cdot \frac{1}{nm^2} \\
&= 1 + \frac{n(n-1)}{2} \cdot \frac{1}{nm} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
\end{aligned}$$

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 .

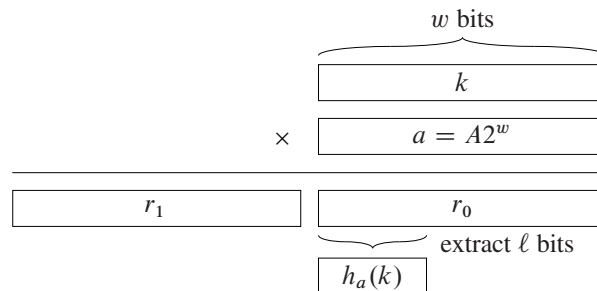
Disadvantage: Slower than division method.

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 $\ell \leq w$.
- Assume that the key k fits into a single word. (k takes w bits.)
- Choose a fixed w -bit positive integer $a = A2^w$ in the range $0 < a < 2^w$.



- Multiply k by a .
- Multiplying two w -bit words \Rightarrow the result is $2w$ bits, $r_12^w + r_0$, 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 $\ell \leq w$, 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 $ka = kA2^w$), 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 $\{0, 1, \dots, m-1\}$. \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 $\{0, 1, \dots, m-1\}$, keys in U , slot numbers in $\{0, 1, \dots, m-1\}$, 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, \dots, k_d and any slots q_1, q_2, \dots, 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 $\{0, 1, \dots, p-1\}$. Assume that $p > m$ (otherwise, just use direct addressing). m need not be prime.
- Denote $\mathbb{Z}_p = \{0, 1, \dots, p-1\}$, $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$.
- 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

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\} .$$

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 $p \Rightarrow a(k_1 - k_2) \not\equiv 0 \pmod{p}$. 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 :

$$a = ((r_1 - r_2)((k_1 - k_2)^{-1} \bmod p)) \bmod p ,$$

$$b = (r_1 - ak_1) \bmod p .$$

$((k_1 - k_2)^{-1} \bmod p)$ 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 $\langle 0, 1, \dots, m-1 \rangle$ (so that all slots are examined if necessary, and so that no slot is examined more than once).
- Thus, the hash function is $h : U \times \underbrace{\{0, 1, \dots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \dots, m-1\}}_{\text{slot number}}$.
- The requirement that the sequence of slots be a permutation of $\langle 0, 1, \dots, m-1 \rangle$ is equivalent to requiring that the **probe sequence** $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ be a permutation of $\langle 0, 1, \dots, m-1 \rangle$.
- 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  $T[q] == \text{NIL}$ 
     $T[q] = k$ 
    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  $T[q] == \text{NIL}$  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 $\langle 0, 1, \dots, m-1 \rangle$ 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 $\{0, 1, \dots, m-1\}$. 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 $U \rightarrow \{0, 1, \dots, m-1\}$.

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 ($14 \bmod 13 = 1$), which is occupied. Second probe is to slot 5 ($((14 \bmod 13) + (1 + (14 \bmod 11))) \bmod 13 = (1 + 4) \bmod 13 = 5$), which is occupied. Third probe is to slot 9 (offset from slot 5 by 4), which is free, so key 14 goes there.

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

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 $\{0, 1, \dots, m-1\}$.

- 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 ($0 \leq i < m$), $h(k, i) = (h'(k) + i) \bmod m$.

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 $0 \leq n < m \Rightarrow 0 \leq \alpha < 1$.
- 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 $1 + \alpha + \alpha^2 + \alpha^3 + \dots = 1/(1 - \alpha)$ (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 $i = 1, 2, \dots$, to be the event that there is an i th probe and that it's to an occupied slot.

$X \geq i$ if and only if probes $1, 2, \dots, i - 1$ are made and are to occupied slots \Rightarrow
 $\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$.

By Exercise C.2-5,

$$\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \dots \cap A_{i-2}\}.$$

Claim

$\Pr\{A_j \mid A_1 \cap A_2 \cap \dots \cap A_{j-1}\} = (n - j + 1)/(m - j + 1)$. 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 $\{0, 1, \dots, m - 1\}$, 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 j th probe is to an occupied slot is $(n - j + 1)/(m - j + 1)$. ■ (claim)

Using this claim,

$$\Pr\{X \geq i\} = \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}}_{i-1 \text{ factors}}.$$

$n < m \Rightarrow (n - j)/(m - j) \leq n/m$ for $j \geq 0$, which implies

$$\begin{aligned} \Pr\{X \geq i\} &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}. \end{aligned}$$

By equation (C.28),

$$\begin{aligned} E[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &= \sum_{i=1}^{n+1} \Pr\{X \geq i\} + \sum_{i>n+1} \Pr\{X \geq i\} \\ &\quad \text{(the } (n+1)\text{st probe must be to an empty slot)} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} + 0 \\ &= \sum_{i=0}^{\infty} \alpha^i \\ &= \frac{1}{1 - \alpha}. \end{aligned} \quad \blacksquare \text{ (theorem)}$$

Interpretation

If α is constant, an unsuccessful search takes $O(1)$ time.

- If $\alpha = 0.5$, then an unsuccessful search takes an average of $1/(1 - 0.5) = 2$ probes.
- If $\alpha = 0.9$, takes an average of $1/(1 - 0.9) = 10$ probes.

Corollary

The expected number of probes to insert is at most $1/(1 - \alpha)$.

Proof Since there is no deletion, insertion uses the same probe sequence as an unsuccessful search. \blacksquare

Theorem

The expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$.

Proof A successful search for key k follows the same probe sequence as when key k was inserted.

By the previous corollary, if k was the $(i + 1)$ st key inserted, then α equaled i/m at the time. Thus, the expected number of probes made in a search for k is at most $1/(1 - i/m) = m/(m - i)$.

That was assuming that k was the $(i + 1)$ st key inserted. We need to average over all n keys:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m - i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m - i}$$

$$\begin{aligned}
&= \frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k} \\
&\leq \frac{1}{\alpha} \int_{m-n}^m (1/x) dx \quad (\text{by inequality (A.19)}) \\
&= \frac{1}{\alpha} \ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}
\end{aligned}$$

■

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 :

$$h(k, i) = q \Rightarrow g(k, q) = i.$$

$$h(k, i) = (h_1(k) + i) \bmod m,$$

$$g(k, q) = (q - h_1(k)) \bmod m.$$

$$h(k, g(k, q)) = q.$$

LINEAR-PROBING-HASH-DELETE(T, q)

```

while TRUE
     $T[q] = \text{NIL}$                                 // 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  $k' == \text{NIL}$ 
            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.]

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

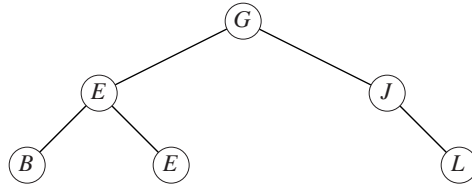
Binary search trees are an important data structure for dynamic sets.

- 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  $x \neq \text{NIL}$ 
  INORDER-TREE-WALK( $x.\text{left}$ )
  print  $\text{key}[x]$ 
  INORDER-TREE-WALK( $x.\text{right}$ )

```

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  $x == \text{NIL}$  or  $k == \text{key}[x]$ 
    return  $x$ 
  if  $k < x.\text{key}$ 
    return TREE-SEARCH( $x.\text{left}, k$ )
  else return TREE-SEARCH( $x.\text{right}, k$ )

```

Initial call is TREE-SEARCH($T.\text{root}, k$).

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  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
    if  $k < x.\text{key}$ 
       $x = x.\text{left}$ 
    else  $x = x.\text{right}$ 
  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 $x = T.\text{root}$.

```

TREE-MINIMUM( $x$ )
  while  $x.\text{left} \neq \text{NIL}$ 
     $x = x.\text{left}$ 
  return  $x$ 

```

TREE-MAXIMUM(x)

```

while  $x.right \neq \text{NIL}$ 
     $x = x.right$ 
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)

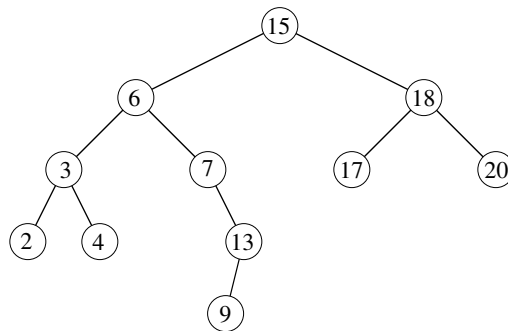
```

if  $x.right \neq \text{NIL}$ 
    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  $y \neq \text{NIL}$  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)

```

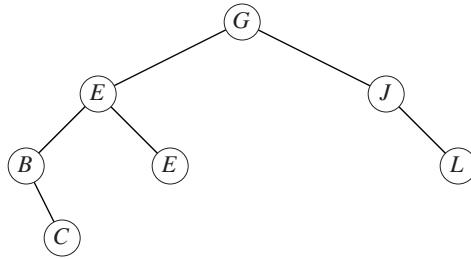
 $x = T.root$            // node being compared with  $z$ 
 $y = NIL$              //  $y$  will be parent of  $z$ 
while  $x \neq NIL$       // descend until reaching a leaf
     $y = x$ 
    if  $z.key < x.key$ 
         $x = x.left$ 
    else  $x = x.right$ 
 $z.p = y$              // found the location—insert  $z$  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, $z.left = NIL$, and $z.right = NIL$.
- 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)

```

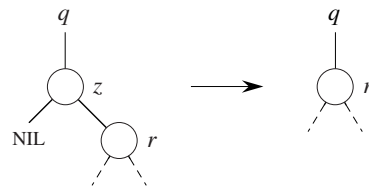
if  $u.p == \text{NIL}$ 
     $T.\text{root} = v$ 
elseif  $u == u.p.\text{left}$ 
     $u.p.\text{left} = v$ 
else  $u.p.\text{right} = v$ 
if  $v \neq \text{NIL}$ 
     $v.p = u.p$ 
  
```

TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v :

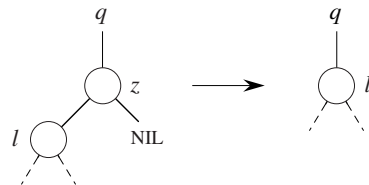
- Makes u 's parent become v 's parent (unless u is the root, in which case it makes v the root).
- u 's parent gets v as either its left or right child, depending on whether u was a left or right child.
- Doesn't update $v.\text{left}$ or $v.\text{right}$, leaving that up to TRANSPLANT's caller.

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.)



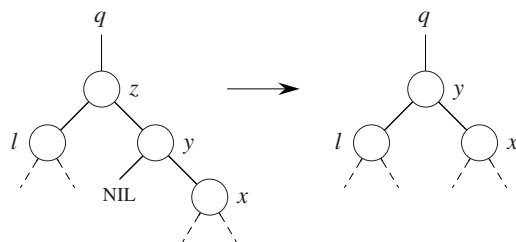
- If z has just one child, and that child is its left child, then replace z by its left child.



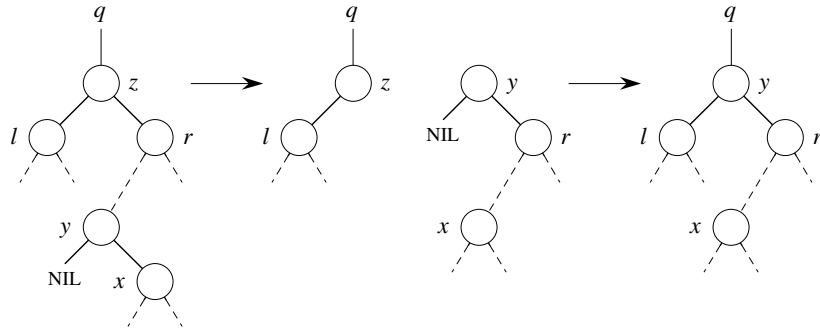
- Otherwise, z has two children. Find z 's successor y . y must lie in z 's right subtree and have no left child.

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)

```

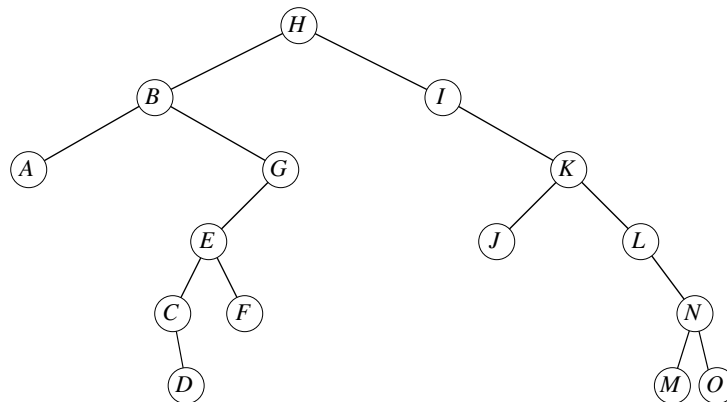
if  $z.left == \text{NIL}$ 
    TRANSPLANT( $T, z, z.right$ )           // replace  $z$  by its right child
elseif  $z.right == \text{NIL}$ 
    TRANSPLANT( $T, z, z.left$ )            // replace  $z$  by its left child
else  $y = \text{TREE-MINIMUM}(z.right)$       //  $y$  is  $z$ 's successor
    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.]

- $\text{TREE-DELETE}(T, I)$ shows the case in which the node deleted has no left child.
- $\text{TREE-DELETE}(T, G)$ shows the case in which the node deleted has a left child but no right child.
- $\text{TREE-DELETE}(T, K)$ shows the case in which the node deleted has both children and its successor is its right child.
- $\text{TREE-DELETE}(T, B)$ 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.]

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 $T.nil$.

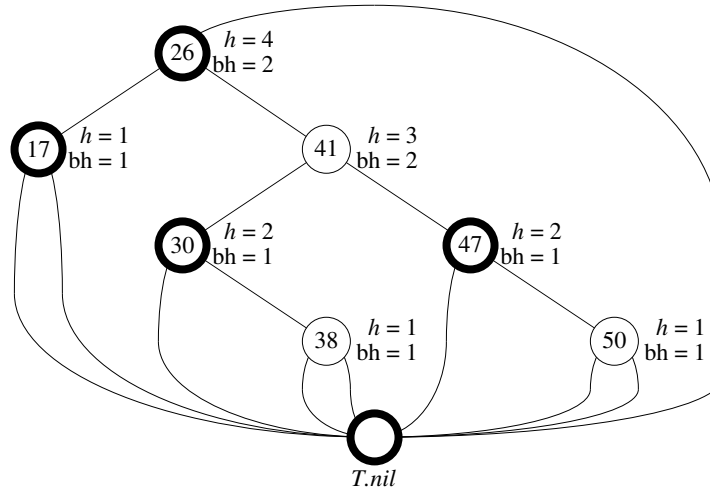
Red-black properties

[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 : $bh(x)$ 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 $\geq 2^{bh(x)} - 1$ internal nodes.

Proof of claim By induction on height of x .

Basis: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow bh(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.

Inductive step: Let the height of x be h and $bh(x) = b$. 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 $\geq 2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ 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?

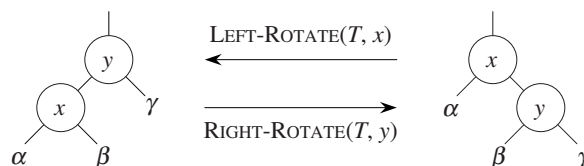
- Red? Might violate property 4.
- Black? Might violate property 5.

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
 $x.p = y$ 

```

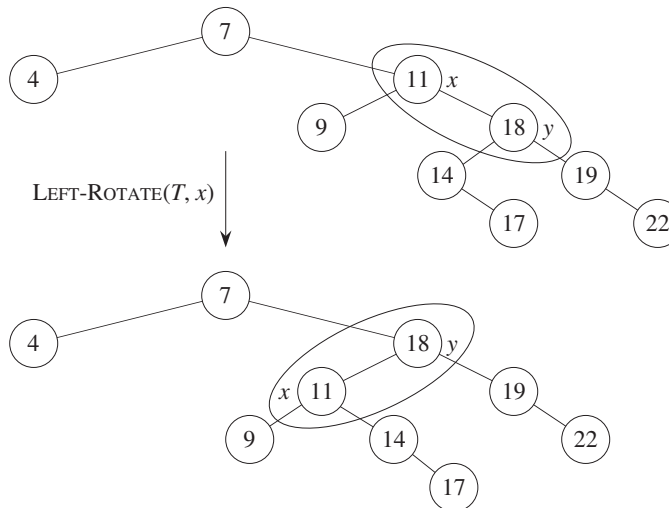
The pseudocode for LEFT-ROTATE assumes that

- $x.right \neq T.nil$, 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  $x \neq T.nil$     // 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?
         $z.p.color = BLACK$ 
         $y.color = BLACK$ 
         $z.p.p.color = RED$ 
         $z = z.p.p$ 
      } case 1
    else
      if  $z == z.p.right$ 
         $z = z.p$ 
        LEFT-ROTATE( $T, z$ )
      } case 2
       $z.p.color = BLACK$ 
       $z.p.p.color = RED$ 
      RIGHT-ROTATE( $T, z.p.p$ )
    } case 3
  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.

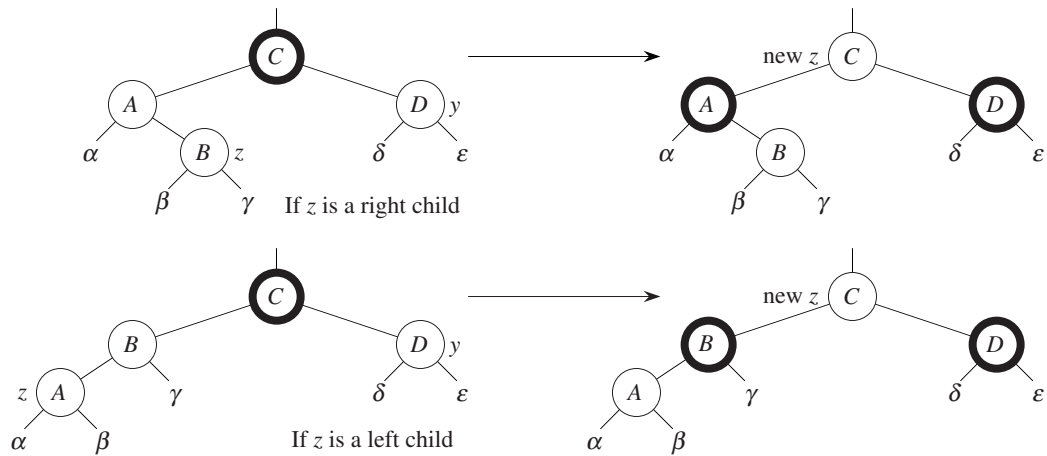
Termination: The loop is guaranteed to terminate. If only case 1 occurs, each iteration moves z toward the root. If case 2 occurs, it falls through into case 3. If case 3 occurs, we'll see that $z.p$ becomes black, so that the loop terminates.

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.

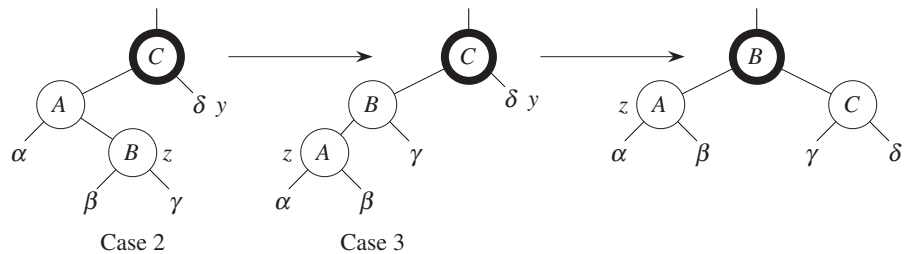
Maintenance: The loop terminates when z is the root (since then $z.p$ is the sentinel $T.nil$, which is black). Upon starting the first iteration of the loop body, the only possible violation is of property 4.

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

- $z.p.p$ (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

- 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 $\Rightarrow O(\lg n)$ 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)

```

 $y = z$ 
 $y\text{-original-color} = y.\text{color}$ 
if  $z.\text{left} == T.\text{nil}$ 
     $x = z.\text{right}$ 
    RB-TRANSPLANT( $T, z, z.\text{right}$ )           // replace  $z$  by its right child
elseif  $z.\text{right} == T.\text{nil}$ 
     $x = z.\text{left}$ 
    RB-TRANSPLANT( $T, z, z.\text{left}$ )           // replace  $z$  by its left child
else  $y = \text{TREE-MINIMUM}(z.\text{right})$        //  $y$  is  $z$ 's successor
     $y\text{-original-color} = y.\text{color}$ 
     $x = y.\text{right}$ 
    if  $y \neq z.\text{right}$                      // is  $y$  farther down the tree?
        RB-TRANSPLANT( $T, y, y.\text{right}$ )    // replace  $y$  by its right child
         $y.\text{right} = z.\text{right}$               //  $z$ 's right child becomes
         $y.\text{right}.p = y$                   //  $y$ 's right child
    else  $x.p = y$                          // in case  $x$  is  $T.\text{nil}$ 
    RB-TRANSPLANT( $T, z, y$ )               // replace  $z$  by its successor  $y$ 
     $y.\text{left} = z.\text{left}$                    // and give  $z$ 's left child to  $y$ ,
     $y.\text{left}.p = y$                        // which had no left child
     $y.\text{color} = z.\text{color}$ 
if  $y\text{-original-color} == \text{BLACK}$          // 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 $x.color = \text{BLACK}$) or **red & black** (if $x.color = \text{RED}$).
 - 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  $x \neq T.root$  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( $T, x.p$ )
       $w = x.p.right$ 
    } case 1
    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( $T, w$ )
         $w = x.p.right$ 
      } case 3
       $w.color = x.p.color$ 
       $x.p.color = BLACK$ 
       $w.right.color = BLACK$ 
      LEFT-ROTATE( $T, x.p$ )
       $x = T.root$ 
    } case 4
    else (same as then part, but with “right” and “left” exchanged)
   $x.color = BLACK$ 

```

Idea

Move the extra black up the tree until

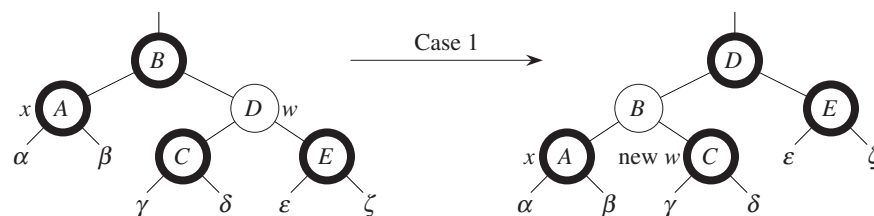
- x points to a red & black node \Rightarrow turn it into a black node,
- x points to the root \Rightarrow just remove the extra black, or
- can perform certain rotations and recolorings and then finish.

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 $x.p$.

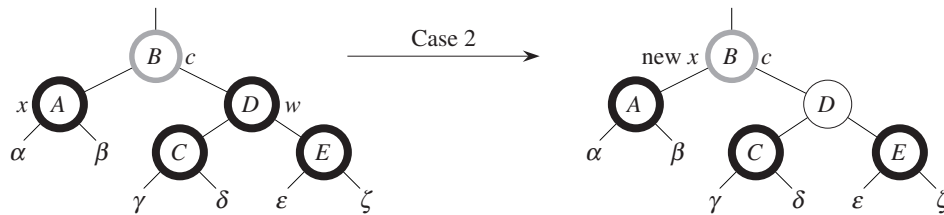
There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which x is a left child.

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 \Rightarrow must be black.
- Go immediately to case 2, 3, or 4.

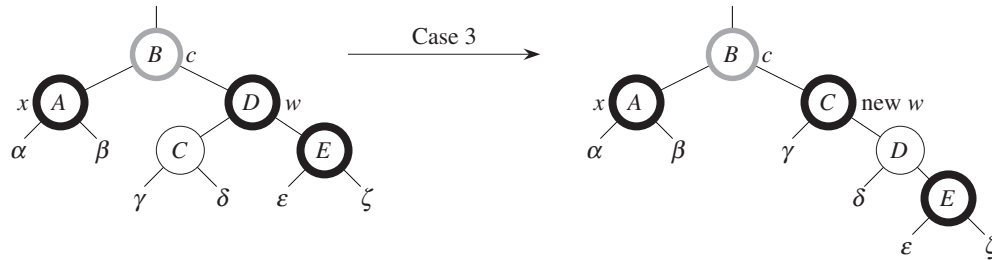
Case 2: w is black and both of w 's children are black



[Node with gray outline is of unknown color, denoted by c .]

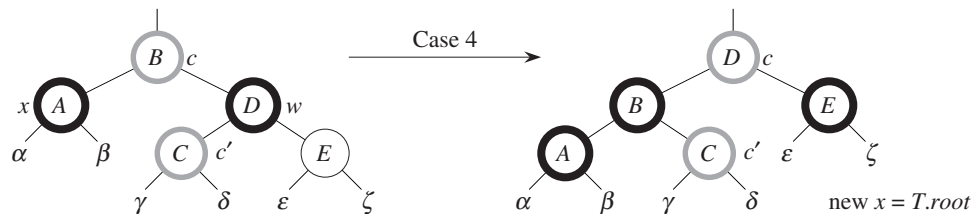
- Take 1 black off x (\Rightarrow singly black) and off w (\Rightarrow red).
- Move that black to $x.p$.
- Do the next iteration with $x.p$ as the new x .
- If entered this case from case 1, then $x.p$ was red \Rightarrow new x is red & black \Rightarrow color attribute of new x is RED \Rightarrow loop terminates. Then new x is made black in the last line.

Case 3: w is black, w 's left child is red, and w 's right child is black



- Make w red and w 's left child black.
- Then right rotate on w .
- New sibling w of x is black with a red right child \Rightarrow case 4.

Case 4: w is black 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.]

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 $i = 1, 2, \dots, n$.

Output: The maximum revenue obtainable for rods whose lengths sum to n , computed as the sum of the prices for the individual rods.

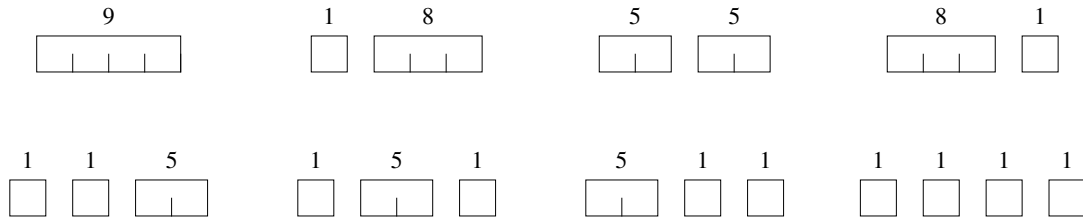
If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

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,

$$r_n = \max \{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1\}.$$

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 :

$$r_n = \max \{p_i + r_{n-i} : 1 \leq i \leq n\} .$$

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

$q = -\infty$

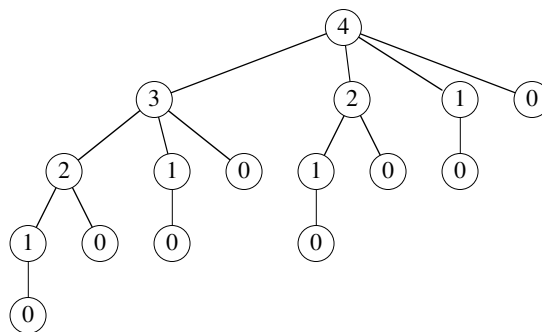
 for $i = 1$ to n

$q = \max \{q, p[i] + \text{CUT-ROD}(p, n - i)\}$

 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” \Rightarrow 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 “memorizing.”]

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

$r[i] = -\infty$

return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)

if $r[n] \geq 0$ // already have a solution for length n ?

return $r[n]$

if $n == 0$

$q = 0$

else $q = -\infty$

for $i = 1$ **to** n // i is the position of the first cut

$q = \max \{q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r)\}$

$r[n] = q$ // 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  $j$ 
     $q = -\infty$ 
    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 $0, 1, \dots, n$. To solve a subproblem of size n , the **for** loop iterates n times \Rightarrow over all recursive calls, total number of iterations forms an arithmetic series. [Actually using aggregate analysis, which Chapter 16 covers.]

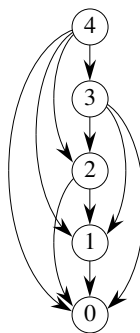
Subproblem graphs

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

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) $\langle A_1, A_2, \dots, A_n \rangle$ 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 $\langle p_0, p_1, p_2, \dots, p_n \rangle$.

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 $k = 1, 2, \dots, n-1$. Then have to split the subproducts. Get

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

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 $A_i A_{i+1} \dots A_j$.

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 $i = 1, 2, \dots, n$.

If $i < j$, then suppose the optimal split is between A_k and A_{k+1} , where $i \leq k < j$. Then $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.

But that's assuming you know the value of k . Have to try all possible values and pick the best, so that

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j : i \leq k < j\} & \text{if } i < j. \end{cases}$$

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 $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.

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 $1 \leq i \leq j \leq n$. 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  $j - 1$ 
         $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
        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 $i = 1, 2, \dots, n$. 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

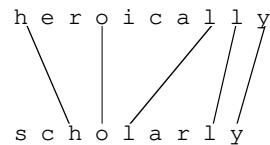
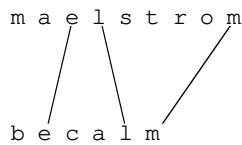
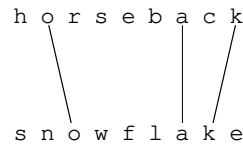
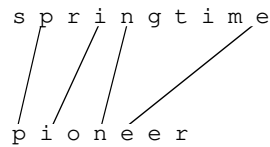
[The textbook has the section on elements of dynamic programming next, but these lecture notes reserve that section for the end of the chapter so that it may refer to two more examples of dynamic programming.]

Problem: Given two sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$. 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:

$X_i = \text{prefix } \langle x_1, \dots, x_i \rangle$

$Y_i = \text{prefix } \langle y_1, \dots, y_i \rangle$

Theorem

Let $Z = \langle z_1, \dots, z_k \rangle$ 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 $Z' = \langle z_1, \dots, z_k, x_m \rangle$. 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.

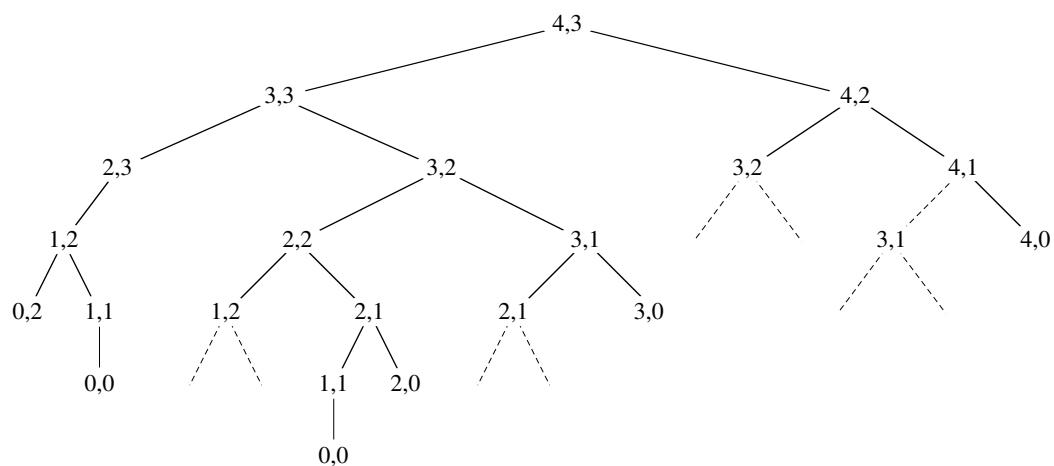
Step 2: Recursively define an optimal solution

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
s	0	0	0	0	0	0	0	0	0	0
p	0	0	0	①	1	1	1	1	1	1
a	0	1	1	1	1	1	②	2	2	2
n	0	1	1	1	1	1	2	2	2	3
k	0	1	1	1	1	1	2	2	2	3
i	0	1	1	1	1	1	2	2	③	3
n	0	1	1	1	1	1	2	2	3	④
g	0	1	1	1	1	1	2	2	3	3
				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 $K = \langle k_1, k_2, \dots, k_n \rangle$ of n distinct keys, sorted ($k_1 < k_2 < \dots < k_n$).
- 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 .

$E[\text{search cost in } T]$

$$= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i$$

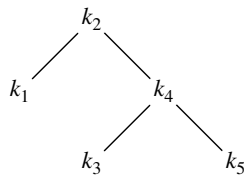
$$\begin{aligned}
&= \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^n p_i \\
&= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i \quad (\text{since probabilities sum to 1}) \quad (*)
\end{aligned}$$

[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.]

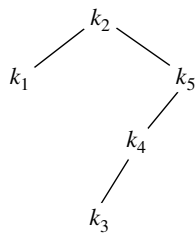
Example

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
		<hr/> 1.15

Therefore, $E[\text{search cost}] = 2.15$.



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
		<hr/> 1.10

Therefore, $E[\text{search cost}] = 2.10$, which turns out to be optimal.

Observations

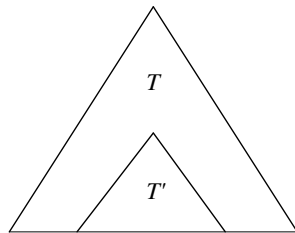
- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.

Build by exhaustive checking?

- Construct each n -node BST.
- For each, put in keys.
- Then compute expected search cost.
- But there are $\Omega(4^n / n^{3/2})$ different BSTs with n nodes.

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, \dots, k_j for some $1 \leq i \leq j \leq n$.

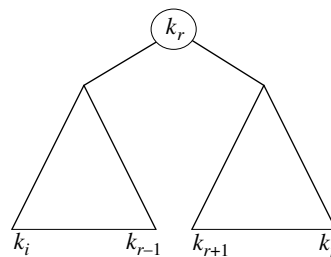


If T is an optimal BST and T contains subtree T' with keys k_i, \dots, k_j , then T' must be an optimal BST for keys k_i, \dots, k_j .

Proof Cut and paste. ■

Use optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblems:

- Given keys k_i, \dots, k_j (the problem).
- One of them, k_r , where $i \leq r \leq j$, must be the root.
- Left subtree of k_r contains k_i, \dots, k_{r-1} .
- Right subtree of k_r contains k_{r+1}, \dots, k_j .



- If
 - you examine all candidate roots k_r , for $i \leq r \leq j$, and
 - you determine all optimal BSTs containing k_i, \dots, k_{r-1} and containing k_{r+1}, \dots, k_j ,
 then you're guaranteed to find an optimal BST for k_i, \dots, k_j .

Step 2: Recursive solution

Subproblem domain:

- Find optimal BST for k_i, \dots, 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, \dots, 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, \dots, k_{r-1} as the left subtree.
- Make an optimal BST with k_{r+1}, \dots, k_j as the right subtree.
- Note: when $r = i$, left subtree is k_i, \dots, k_{i-1} ; when $r = j$, right subtree is k_{j+1}, \dots, k_j . These subtrees 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

$$w(i, j) = \sum_{l=i}^j p_l \quad (\text{refer to equation } (*)).$$

If k_r is the root of an optimal BST for k_i, \dots, 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:

$$e[\underbrace{1:n+1}_{\text{can store}}, \underbrace{0:n}_{\text{can store}}]$$

$$e[n+1, n] \quad e[1, 0]$$

- Will use only entries $e[i, j]$, where $j \geq i - 1$.

- Will also compute

$root[i, j] = \text{root of subtree with keys } k_i, \dots, k_j, \text{ for } 1 \leq i \leq j \leq n.$

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 $1 \leq i \leq n$
- $w[i, j] = w[i, j - 1] + p_j$ for $1 \leq i \leq j \leq n$

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** $n - l + 1$

$j = i + l - 1$

$e[i, j] = \infty$

$w[i, j] = w[i, j - 1] + p_j$

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	
4				0	.2	.7	
5					0	.3	
6						0	

p_i points to the cell $e[2, 1]$.

		<i>j</i>					
<i>w</i>		0	1	2	3	4	5
<i>i</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

		<i>j</i>				
<i>root</i>		1	2	3	4	5
<i>i</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*)

$r = \text{root}[1, n]$

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 $i \leq j$

$t = \text{root}[i, j]$

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 $1 \leq i \leq n$.
- 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 common subsequence

- Space of subproblems for $\langle x_1, \dots, x_i \rangle$ and $\langle y_1, \dots, y_j \rangle$ was just $\langle x_1, \dots, x_{i-1} \rangle$ and $\langle y_1, \dots, y_{j-1} \rangle$.
- 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, \dots, k_j .
- An optimal BST would have root k_r , for some $1 \leq r \leq j$.
- Get subproblems k_1, \dots, k_{r-1} and k_{r+1}, \dots, k_j .
- Unless we could guarantee that $r = j$, so that subproblem with k_{r+1}, \dots, k_j is empty, then this subproblem is *not* of the form k_1, k_2, \dots, 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 $A_i, A_{i+1}, \dots, A_{j-1}$. 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 $x_i \neq y_j$, LCS of X_{i-1} and Y , and LCS of X and Y_{j-1})
 - Optimal binary search tree:
 - 2 subproblems (k_i, \dots, k_{r-1} and k_{r+1}, \dots, k_j)
 - $j - i + 1$ choices for k_r in k_i, \dots, 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) \times (# of choices).

- Rod cutting: $\Theta(n)$ subproblems, $\leq n$ 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 $i = 0, 1, \dots, n$, vertex for subproblem size i has out-degree $i \Rightarrow \# \text{ of edges} = \sum_{i=0}^n i = n(n+1)/2$.
- 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.

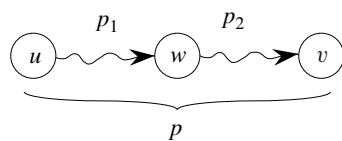
Here are two problems that look similar. In both, we're given an *unweighted, directed graph* $G = (V, E)$.

- 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 $u \rightsquigarrow w$.

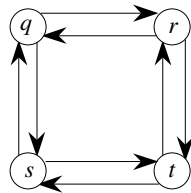
Proof Suppose there exists a shorter path p'_1 going $u \rightsquigarrow w$. Cut out p_1 , replace it with p'_1 , get path $u \rightsquigarrow^{p'_1} w \rightsquigarrow^{p_2} v$ with fewer edges than p . ■

Therefore, can find shortest path $u \rightsquigarrow v$ by considering all intermediate vertices w , then finding shortest paths $u \rightsquigarrow w$ and $w \rightsquigarrow v$.

Same argument applies to p_2 .

Does longest path have optimal substructure?

- It seems like it should.
- It does *not*.



Consider $q \rightarrow r \rightarrow t =$ longest path $q \rightsquigarrow t$. Are its subpaths longest paths?

No!

- Subpath $q \rightsquigarrow r$ is $q \rightarrow r$.
- Longest simple path $q \rightsquigarrow r$ is $q \rightarrow s \rightarrow t \rightarrow r$.
- Subpath $r \rightsquigarrow t$ is $r \rightarrow t$.
- Longest simple path $r \rightsquigarrow t$ is $r \rightarrow q \rightarrow s \rightarrow t$.

Not only isn't there optimal substructure, but can't even assemble a legal solution from solutions to subproblems.

Combine longest simple paths:

$q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$

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 $q \rightsquigarrow r$ and $r \rightsquigarrow t$.
- Longest simple path $q \rightsquigarrow r$ uses s and t .
- Cannot use s and t to solve longest simple path $r \rightsquigarrow t$, since if you do, the path isn't simple.
- But you *have* to use t to find longest simple path $r \rightsquigarrow t$!

- Using resources (vertices) to solve one subproblem renders them unavailable to solve the other subproblem.

[For shortest paths, for a shortest path $u \xrightarrow{p_1} w \xrightarrow{p_2} v$, no vertex other than w can appear in p_1 and p_2 . Otherwise, get a cycle.]

Independent subproblems in our examples:

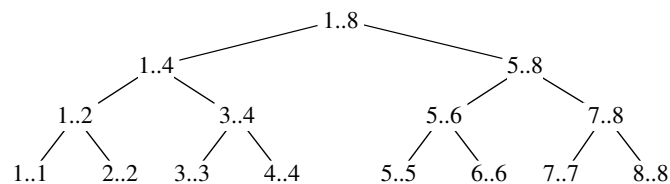
- Rod cutting and longest common subsequence
 - 1 subproblem \Rightarrow automatically independent.
- Matrix-chain multiplication
 - $A_i \cdots A_k$ and $A_{k+1} \cdots A_j \Rightarrow$ independent.
- Optimal binary search tree
 - k_i, \dots, k_{r-1} and $k_{r+1}, \dots, k_j \Rightarrow$ independent.

Overlapping subproblems

These occur when a recursive algorithm revisits the same problem over and over.

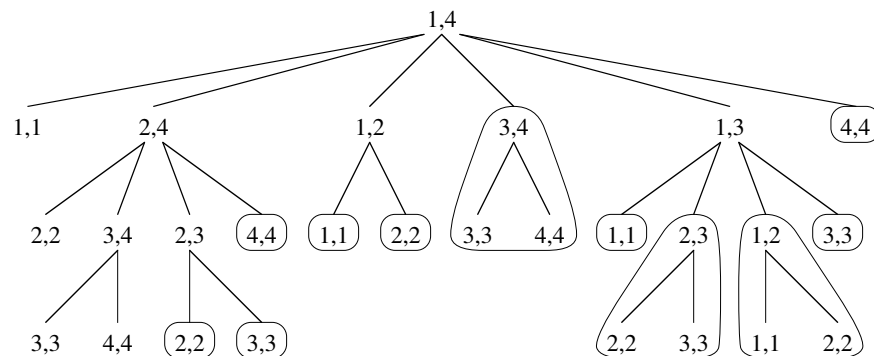
Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

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.

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 $S = \{a_1, \dots, a_n\}$.

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:

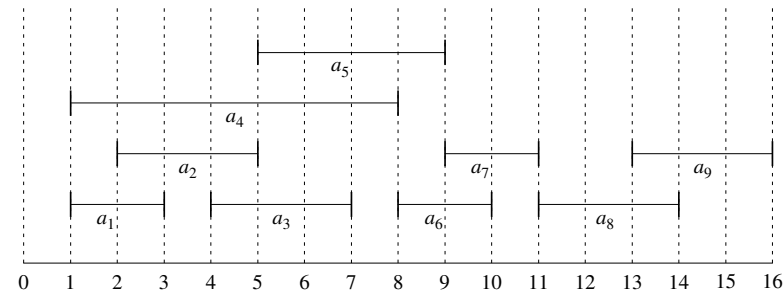
- Schedule room for longest time.
- Maximize income rental fees.

Assume that activities are sorted by finish time: $f_1 \leq f_2 \leq f_3 \leq \dots \leq f_{n-1} \leq f_n$.

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



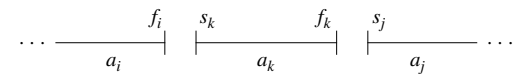
Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

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

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\} \quad [\text{Leave on board}]$$

= activities that start after a_i finishes and finish before a_j starts.



Activities in S_{ij} are compatible with

- all activities that finish by f_i , and
- all activities that start no earlier than s_j .

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

$$A_{ik} = A_{ij} \cap S_{ik} = \text{activities in } A_{ij} \text{ that finish before } a_k \text{ starts,}$$

$$A_{kj} = A_{ij} \cap S_{kj} = \text{activities in } A_{ij} \text{ that start after } a_k \text{ finishes.}$$

$$\text{Then } A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

$$\Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1.$$

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 $f_m \leq f_j$. ■ (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 $m \leq n$ and $s[m] < f[k]$ // find the first activity in S_k to finish

$m = m + 1$

if $m \leq n$

return $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$

else return \emptyset

Initial call

RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$).

Idea

The **while** loop checks $a_{k+1}, a_{k+2}, \dots, a_n$ 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 \Rightarrow optimal solution to problem.

Greedy vs. dynamic programming

The knapsack problem is a good example of the difference.

0-1 knapsack problem

- n items.
- Item i is worth v_i , weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- Have to either take an item or not take it—can't take part of it.

Fractional knapsack problem

Like the 0-1 knapsack problem, but can take fraction of an item.

Both have optimal substructure.

But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.

To solve the fractional problem, rank items by value/weight: v_i/w_i . Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i . 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

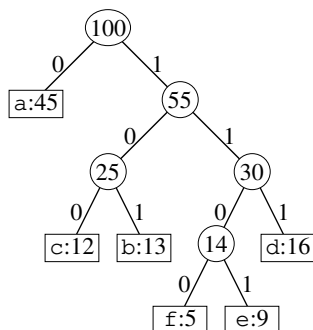
No codeword is also a prefix of any other codeword. [Called “prefix codes” in earlier editions of the book. Changed to “prefix-free codes” in the fourth edition because each codeword is free of prefixes of other codes.] A prefix-free code can always achieve the optimal compression.

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

$x = \text{EXTRACT-MIN}(Q)$

$y = \text{EXTRACT-MIN}(Q)$

$z.\text{left} = x$

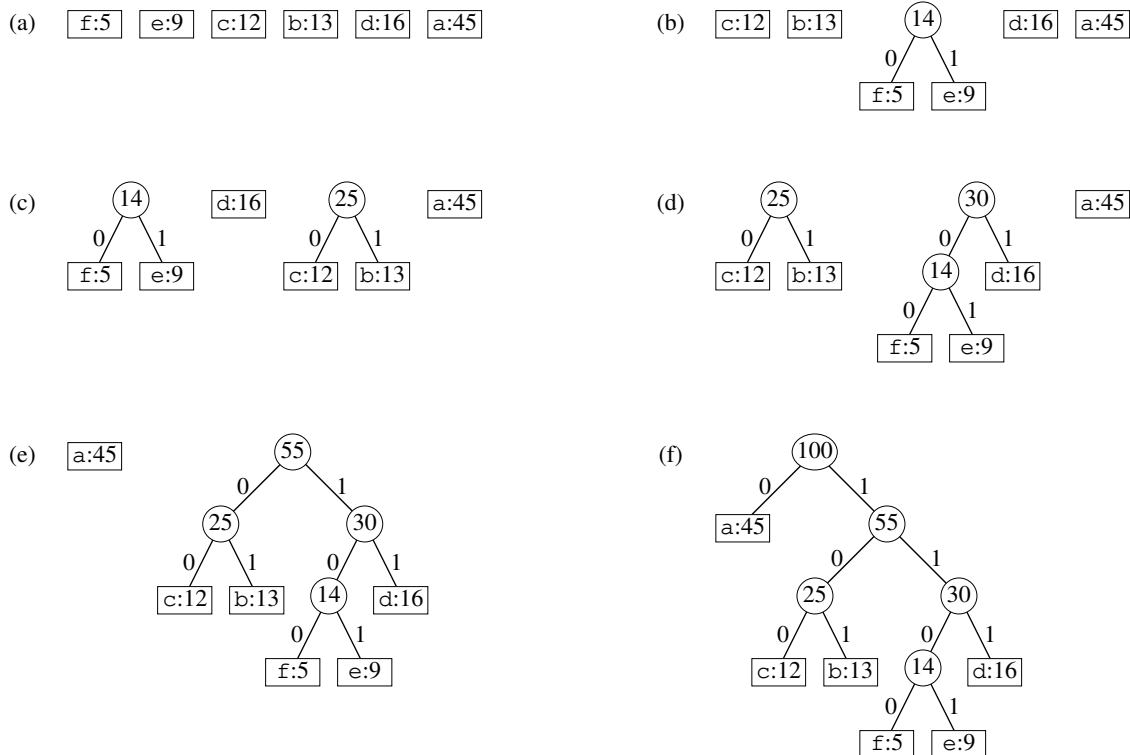
$z.\text{right} = y$

$z.\text{freq} = x.\text{freq} + y.\text{freq}$

$\text{INSERT}(Q, z)$

return $\text{EXTRACT-MIN}(Q)$ // 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.

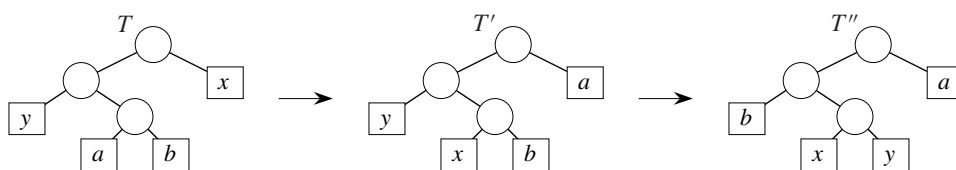
Let a, b be two characters that are sibling leaves of maximum depth in T . Assume wlog that $a.\text{freq} \leq b.\text{freq}$ and $x.\text{freq} \leq y.\text{freq}$. Must have $x.\text{freq} \leq a.\text{freq}$ and $y.\text{freq} \leq b.\text{freq}$.

Could have $x.\text{freq} = a.\text{freq}$ or $y.\text{freq} = b.\text{freq}$. If $x.\text{freq} = b.\text{freq}$, then $a.\text{freq} = b.\text{freq} = x.\text{freq} = y.\text{freq}$ (Exercise 15.3-1), and the lemma is trivially true. So assume that $x.\text{freq} \neq b.\text{freq} \Rightarrow x \neq b$.

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**

$B(T') \leq B(T)$. (Exchanging a and x does not increase the cost.)

Proof of claim

$$\begin{aligned}
 B(T) - B(T') &= \sum_{c \in C} c.\text{freq} \cdot d_T(c) - \sum_{c \in C} c.\text{freq} \cdot d_{T'}(c) \\
 &= x.\text{freq} \cdot d_T(x) + a.\text{freq} \cdot d_T(a) - x.\text{freq} \cdot d_{T'}(x) - a.\text{freq} \cdot d_{T'}(a) \\
 &= x.\text{freq} \cdot d_T(x) + a.\text{freq} \cdot d_T(a) - x.\text{freq} \cdot d_T(a) - a.\text{freq} \cdot d_T(x) \\
 &= (a.\text{freq} - x.\text{freq})(d_T(a) - d_T(x)) \\
 &\geq 0.
 \end{aligned}$$

The last line follows because $x.\text{freq} \leq a.\text{freq}$ and a is a maximum-depth leaf $\Rightarrow d_T(a) \geq d_T(x)$. ■ (claim)

Similarly, $B(T'') \leq B(T')$ because exchanging y and b doesn't increase the cost. Therefore, $B(T'') \leq B(T') \leq B(T)$. T is optimal $\Rightarrow B(T) \leq B(T'') \Rightarrow B(T'') = B(T) \Rightarrow T''$ 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 $z.freq = x.freq + y.freq$. 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 $c \in C - \{x, y\} \Rightarrow d_T(c) = d_{T'}(c) \Rightarrow c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$.
 $d_T(x) = d_T(y) = d_{T'}(z) + 1 \Rightarrow$

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq), \end{aligned}$$

so that $B(T) = B(T') + x.freq + y.freq$, which is equivalent to $B(T') = B(T) - x.freq - y.freq$.

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 $z.freq = x.freq + y.freq$ and call the resulting tree T''' . Then,

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T'), \end{aligned}$$

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, \dots, b_n with cache configuration C at the time b_i is requested. C is a subset of all the blocks, $|C| \leq k$.

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 $\Rightarrow C' \neq C$.

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 \Rightarrow no change in the cache $\Rightarrow R_{C,i} = \{C\}$.
- Compulsory miss \Rightarrow insert b_i into the cache $\Rightarrow R_{C,i} = \{C \cup \{b_i\}\}$.
- Non-compulsory miss (cache is full) \Rightarrow 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 $\text{miss}(C, i)$ = the minimum number of cache misses in a solution for subproblem (C, i) . Recurrence for $\text{miss}(C, i)$:

$$\text{miss}(C, i) = \begin{cases} 0 & \text{if } i = n \text{ and } b_n \in C, \\ 1 & \text{if } i = n \text{ and } b_n \notin C, \\ \text{miss}(C, i + 1) & \text{if } i < n \text{ and } b_i \in C, \\ 1 + \min \{\text{miss}(C', i + 1) : C' \in R_{C,i}\} & \text{if } i < n \text{ and } b_i \notin C. \end{cases}$$

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.

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

- $\text{PUSH}(S, x)$: $O(1)$ each $\Rightarrow O(n)$ for any sequence of n operations.
- $\text{POP}(S)$: $O(1)$ each $\Rightarrow O(n)$ for any sequence of n operations.

- $\text{MULTIPOP}(S, k)$
 - while** not $\text{STACK-EMPTY}(S)$ and $k > 0$
 - $\text{POP}(S)$
 - $k = k - 1$

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 $\leq n$ $\text{PUSHes} \Rightarrow \leq n$ 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 (mod 2^k):

$\text{INCREMENT}(A, k)$

```

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 <u>0</u>	0
1	0 <u>0</u> <u>1</u>	1
2	0 1 <u>0</u>	3
3	<u>0</u> <u>1</u> <u>1</u>	4
4	1 0 <u>0</u>	7
5	1 <u>0</u> <u>1</u>	8
6	1 1 <u>0</u>	10
7	<u>1</u> <u>1</u> <u>1</u>	11
0	0 0 <u>0</u>	14
\vdots	\vdots	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	$\lfloor n/4 \rfloor$
\vdots		
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
\vdots		
$i \geq k$	never	0

$$\begin{aligned}
 \text{Therefore, total \# of flips} &= \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \\
 &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\
 &= n \cdot \frac{1}{1 - 1/2} \\
 &= 2n.
 \end{aligned}$$

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*.

Let c_i = actual cost of i th operation ,

\hat{c}_i = amortized cost of i th operation .

Then require $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$ for *all* sequences of n operations.

Total credit stored = $\sum_{i=1}^n \hat{c}_i - \underbrace{\sum_{i=1}^n c_i}_{\text{had better be}} \geq 0$.

Stack

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	$\min(k, s)$	0

Intuition

When pushing an object, pay \$2.

- \$1 pays for the PUSH.
- \$1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has \$1, which is credit, the credit can never go negative.
- Therefore, total amortized cost, = $O(n)$, is an upper bound on total actual cost.

Binary counter

Charge \$2 to set a bit to 1.

- \$1 pays for setting a bit to 1.
- \$1 is prepayment for flipping it back to 0.
- Have \$1 of credit for every 1 in the counter.
- Therefore, credit ≥ 0 .

Amortized cost of INCREMENT:

- Cost of resetting bits to 0 is paid by credit.
- At most 1 bit is set to 1.
- Therefore, amortized cost $\leq \$2$.
- For n operations, amortized cost = $O(n)$.

Potential method

Like the accounting method, but think of the credit as *potential* stored with the entire data structure.

- Accounting method stores credit with specific objects.
- Potential method stores potential in the data structure as a whole.
- Can release potential to pay for future operations.
- Most flexible of the amortized analysis methods.

Let D_i = data structure after i th operation ,

D_0 = initial data structure ,

c_i = actual cost of i th operation ,

\hat{c}_i = amortized cost of i th operation .

Potential function $\Phi : D_i \rightarrow \mathbb{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i .

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} .$$

increase in potential due to i th operation

$$\begin{aligned} \text{Total amortized cost} &= \sum_{i=1}^n \hat{c}_i \\ &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &\quad \text{(telescoping sum: every term other than } D_0 \text{ and } D_n \\ &\quad \text{is added once and subtracted once)} \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned}$$

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 i th INCREMENT

Suppose i th 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 i th 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 i th 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 $\alpha = \text{num}/\text{size}$, where $\text{num} = \#$ items stored, $\text{size} =$ allocated size.

If $\text{size} = 0$, then $\text{num} = 0$. 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  $T.\text{size} == 0$ 
    allocate  $T.\text{table}$  with 1 slot
     $T.\text{size} = 1$ 
if  $T.\text{num} == T.\text{size}$                                 // expand?
    allocate  $\text{new-table}$  with  $2 \cdot T.\text{size}$  slots
    insert all items in  $T.\text{table}$  into  $\text{new-table}$         //  $T.\text{num}$  elem insertions
    free  $T.\text{table}$ 
     $T.\text{table} = \text{new-table}$ 
     $T.\text{size} = 2 \cdot T.\text{size}$ 
    insert  $x$  into  $T.\text{table}$                                 // 1 elem insertion
     $T.\text{num} = T.\text{num} + 1$ 

```

Initially, $T.\text{num} = T.\text{size} = 0$.

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 i th operation

- If not full, $c_i = 1$.
- If full, have $i - 1$ items in the table at the start of the i th operation. Have to copy all $i - 1$ existing items, then insert i th 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:

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \text{Total cost} &= \sum_{i=1}^n c_i \\ &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} \\ &< n + 2n \\ &= 3n \end{aligned}$$

Therefore, **aggregate analysis** says amortized cost per operation = 3.

Accounting method

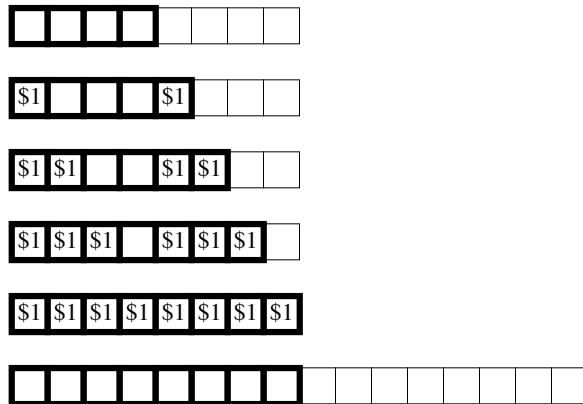
Charge \$3 per insertion of x .

- \$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, $size = num = m$ just before the expansion, and $size = 2m$ and $num = m$ just after the expansion. The next expansion occurs when $size = num = 2m$.

- 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 \Rightarrow 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.

\Rightarrow Over $size/2$ insertions, potential needs to go from 0 to $size$.

\Rightarrow 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 \Rightarrow 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.]

Φ_i = potential after the i th operation ,
 $\Delta\Phi_i$ = change in potential due to the i th operation ,
 $\hat{c}_i = c_i + \Delta\Phi_i$.

When the i th insertion does not trigger expansion

$c_i = 1$ and $\Delta\Phi_i = 2 \Rightarrow \hat{c}_i = c_i + \Delta\Phi_i = 1 + 2 = 3$.

When the i th insertion triggers an expansion

num_i = number of items in the table after the i th operation ,

$size_i$ = size of the table after the i th operation .

Before insertion:

$size_{i-1} = num_{i-1} = i - 1 \Rightarrow$

$\Phi_{i-1} = 2(size_{i-1} - size_{i-1}/2)$

$= size_{i-1}$

$= i - 1$.

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

$\hat{c}_i = c_i + \Delta\Phi_i$

$= i + (3 - i)$

$= 3$,

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.]

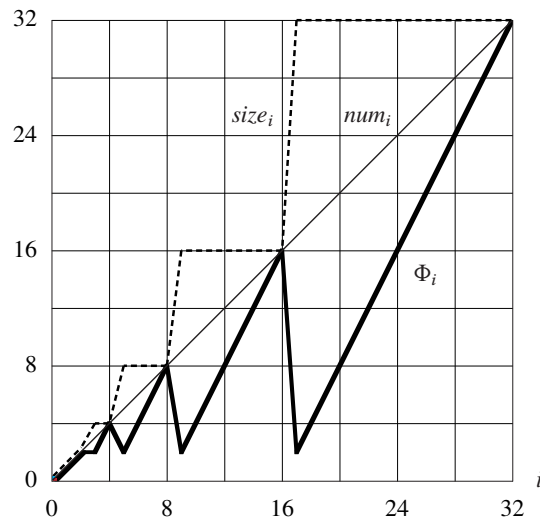


Table expansion and contraction

Now supporting both insertion and deletion from the table.

Want to limit wasted space in the table \Rightarrow 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.

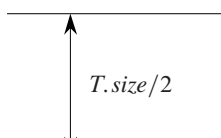
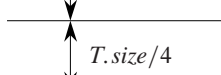
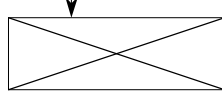
Actual cost of an operation is the number of elementary insertions or deletions.

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 $num = size = n/2$.
- The second $n/2$ operations are the sequence
insert, delete, delete, insert, insert, delete, delete, insert, insert, \dots
- 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 \Rightarrow total cost is $\Theta(n^2) \Rightarrow$ 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 $1/2$) \Rightarrow 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.

$T.num$		α	Φ	$\Delta\Phi$ per operation
$T.size$		1	$T.size$	$\uparrow +2$ per insertion $\downarrow -2$ per deletion
$T.size/2$		$1/2$	0	$\uparrow -1$ per insertion $\downarrow +1$ per deletion
$T.size/4$		$1/4$	$T.size/4$	
0				

As the load factor moves away from $1/2$, the potential builds up to pay for copying all the items \Rightarrow 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 \Rightarrow 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 $\hat{c}_i = c_i + \Delta\Phi_i$.

- Insertion when $\alpha_{i-1} \geq 1/2$: $\Delta\Phi_i = 2 \Rightarrow \hat{c}_i = 1 + 2 = 3$.
- Deletion when $\alpha_i \geq 1/2$: $\Delta\Phi = -2 \Rightarrow \hat{c}_i = 1 - 2 = -1$.
- Insertion when $\alpha_i < 1/2$: $\Delta\Phi = -1 \Rightarrow \hat{c}_i = 1 - 1 = 0$.
- Deletion when $\alpha_{i-1} < 1/2$: $\Delta\Phi = 1 \Rightarrow \hat{c}_i = 1 + 1 = 2$.

When expansion or contraction occurs

An insertion that causes expansion is the same as when the only operation was insertion: $\hat{c}_i = 3$.

For a deletion that causes contraction: $num_{i-1} = size_{i-1}/4$ beforehand \Rightarrow

$$\begin{aligned} \Phi_{i-1} &= size_{i-1}/2 - num_{i-1} \\ &= size_{i-1}/2 - size_{i-1}/4 \\ &= size_{i-1}/4 . \end{aligned}$$

Then the item is deleted, then $num_i = size/2 - 1 \Rightarrow \alpha_i < 1/2 \Rightarrow$

$$\begin{aligned} \Phi_i &= size_i/2 - num_i \\ &= 1 \end{aligned}$$

$$\Rightarrow \Delta \Phi_i = 1 - \text{size}_{i-1}/4.$$

Actual cost $c_i = \text{size}_{i-1}/4$, for deleting 1 item and copying $\text{size}_{i-1}/4 - 1$ items \Rightarrow

$$\begin{aligned}\hat{c}_i &= c_i + \Delta \Phi_i \\ &= \text{size}_{i-1}/4 + (1 - \text{size}_{i-1}/4) \\ &= 1.\end{aligned}$$

When the load factor crosses 1/2

Deletion: $\text{num}_{i-1} = \text{size}_{i-1}/2$ and $\alpha_{i-1} = 1/2$ before.

$\text{num}_i = \text{size}_i/2 - 1$ and $\alpha_i < 1/2$ after.

$$\alpha_{i-1} = 1/2 \Rightarrow \Phi_{i-1} = 0.$$

$$\alpha_i < 1/2 \Rightarrow \Phi_i = \text{size}_i/2 - \text{num}_i = 1.$$

Therefore, $\Delta \Phi_i = 1 - 0 = 1$.

No contraction occurs $\Rightarrow c_i = 1$

$$\Rightarrow \hat{c}_i = 1 + 1 = 2.$$

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 $c_i = 1$

$$\Rightarrow \hat{c}_i = 1 - 1 = 0.$$

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 $O(n)$.

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:

- $\text{OS-SELECT}(x, i)$: return a pointer to the node containing the i th smallest key of the subtree rooted at x .
- $\text{OS-RANK}(T, x)$: return the rank of x in the linear order determined by an inorder walk of T .

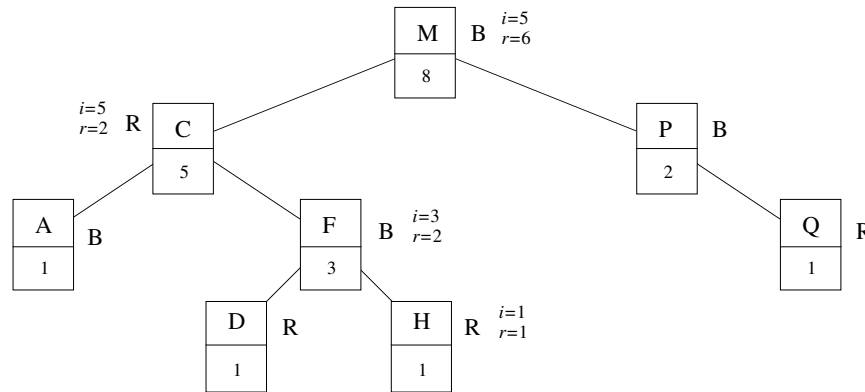
Augment by storing in each node x :

$x.size$ = 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)

$r = x.\text{left.size} + 1$ // rank of x within the subtree rooted at x

if $i == r$

return x

elseif $i < r$

return OS-SELECT($x.\text{left}, i$)

else return OS-SELECT($x.\text{right}, i - r$)

Initial call: OS-SELECT($T.\text{root}, i$)

Try OS-SELECT($T.\text{root}, 5$).

[Values shown in figure above. Returns the node whose key is H.

OS-SELECT($T.\text{root}, 3$) returns the node whose key is D.

OS-SELECT($T.\text{root}, 7$) 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 i th smallest element is in x 's left subtree, and we want the i th smallest element in the subtree.
- If $i > r$, then i th 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 $\Rightarrow O(\lg n)$ time.

Find the rank of an elementOS-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  $y \neq T.root$ 
    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.

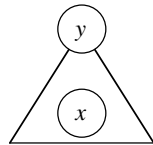
Why does this work?

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 $y = T.root \Rightarrow$ the loop terminates and subtree rooted at y is entire tree. Therefore, r = rank of $x.key$ in entire tree.

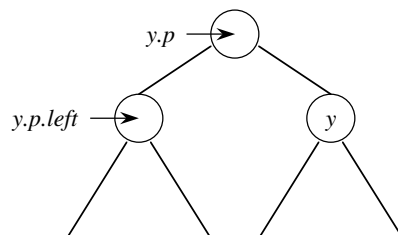
Maintenance: At end of each iteration, set $y = y.p$. So, show that if r = rank of $x.key$ in subtree rooted at y at start of loop body, then r = rank of $x.key$ in subtree rooted at $y.p$ at end of loop body.



[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 \Rightarrow don't change r .
- If y is a right child, all nodes in y 's sibling's subtree precede all nodes in y 's subtree \Rightarrow add size of y 's sibling's subtree, plus 1 for $y.p$, into r .

**Analysis**

y goes up one level in each iteration $\Rightarrow O(\lg n)$ time.

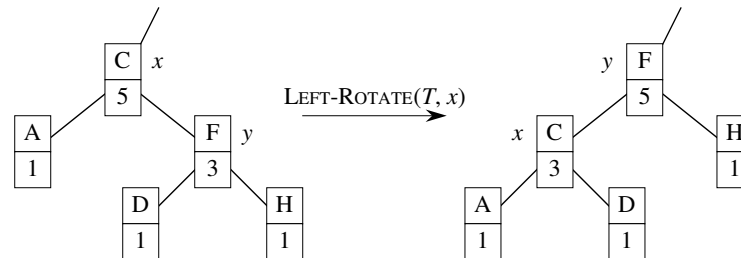
Maintaining subtree sizes

- 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.



$$y.size = x.size$$

$$x.size = x.left.size + x.right.size + 1$$

- 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 $x.p$, $x.p.p$, $x.p.p.p$, ..., $root$).

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

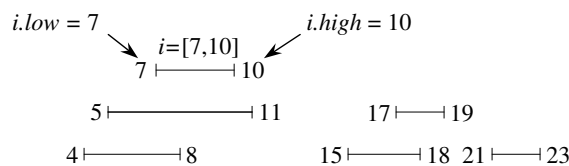
If an update after rotation requires traversing up to the root, then each rotation costs $O(\lg n)$ instead of $O(1)$. Because insertion and deletion in red-black trees bound the number of rotations to a constant, get $O(\lg n)$ time per insertion or deletion in the above theorem.

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 $x.int.high$.

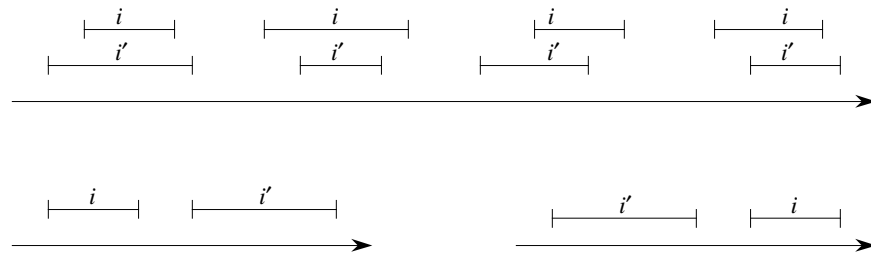
Operations

- 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 $i.low \leq j.high$ and $j.low \leq i.high$.

Interval trichotomy: For intervals i and i' , exactly one of the following holds:

- i and i' overlap.
- i is to the left of i' ($i.high < i'.low$).
- 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.

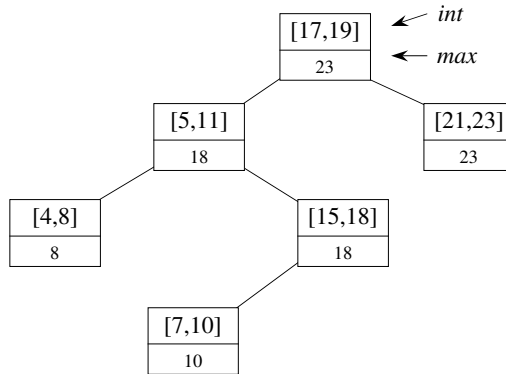
For interval trees

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

$x.max = \text{max endpoint value in subtree rooted at } x$.



[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 $x \neq T.nil$ and i does not overlap $x.int$

if $x.left \neq T.nil$ and $x.left.max \geq i.low$

$x = x.left$ // 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:

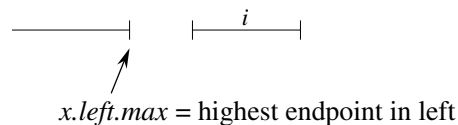
- There is an overlap in right subtree, or
- There is no overlap in either subtree.

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
 - $x.left = T.nil \Rightarrow$ no overlap in left.
 - OR
 - $x.left.max < i.low \Rightarrow$ 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:

$$i.low \leq x.left.max$$

$$= j.high \text{ for some } j \text{ in left subtree .}$$
 - Since there is no overlap in left, i and j don't overlap.
 - Refer back to: no overlap if

$$i.low > j.high \text{ or } j.low > i.high .$$
 - Since $i.low \leq j.high$, must have $j.low > i.high$.
 - Now consider *any* interval k in *right* subtree.
 - Because keys are low endpoint,

$$\underbrace{j.low}_{\text{in left}} \leq \underbrace{k.low}_{\text{in right}} .$$
 - Therefore, $i.high < j.low \leq k.low$.
 - Therefore, $i.high < k.low$.
 - Therefore, i and k do not overlap.

■ (theorem)

Lecture Notes for Chapter 19:

Data Structures for Disjoint Sets

Chapter 19 overview

Disjoint-set data structures

- Also known as “union find.”
- Maintain collection $\mathcal{S} = \{S_1, \dots, S_k\}$ 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

- MAKE-SET(x): make a new set $S_i = \{x\}$, and add S_i to \mathcal{S} .
- UNION(x, y): if $x \in S_x, y \in S_y$, then $\mathcal{S} = \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$.
 - Representative of new set is any member of $S_x \cup S_y$, often the representative of one of S_x and S_y .
 - Destroys S_x and S_y (since sets must be disjoint).
- FIND-SET(x): return representative of set containing x .

Analysis in terms of:

- n = # of elements = # of MAKE-SET operations,
- m = total # of operations.

Analysis

- Since MAKE-SET counts toward total # of operations, $m \geq n$.
- Can have at most $n - 1$ UNION operations, since after $n - 1$ UNIONS, only 1 set remains.
- Assume that the first n operations are MAKE-SET (helpful for analysis, usually not really necessary).

Application

Dynamic connected components.

For a graph $G = (V, E)$, vertices u, v are in same connected component if and only if there's a path between them.

- Connected components partition vertices into equivalence classes.

CONNECTED-COMPONENTS(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.

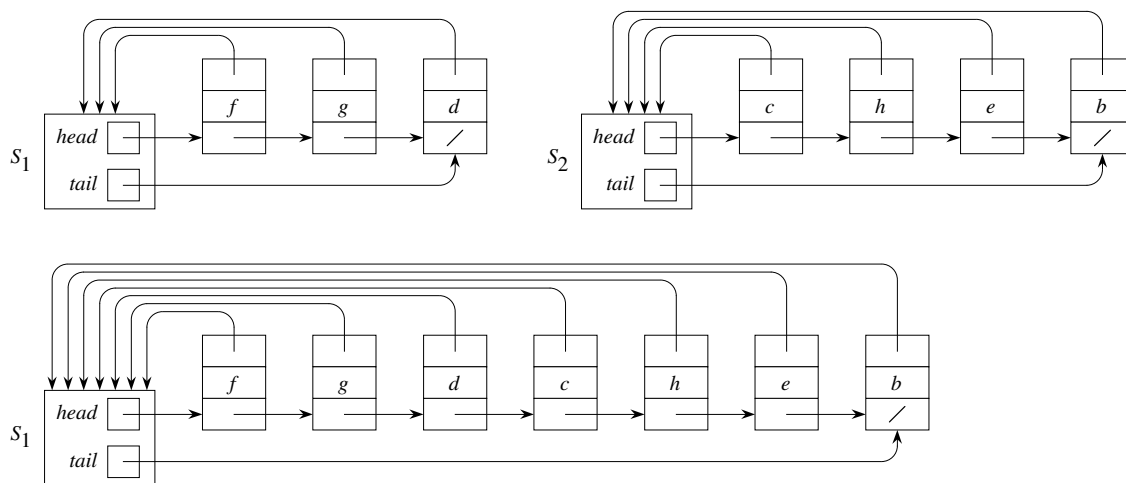
- Each object in the list has attributes for
 - the set member,
 - pointer to the set object, and
 - next.

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 $\text{UNION}(S_1, S_2)$. 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
$\text{UNION}(x_2, x_1)$	1
$\text{UNION}(x_3, x_2)$	2
$\text{UNION}(x_4, x_3)$	3
$\text{UNION}(x_5, x_4)$	4
\vdots	\vdots
$\text{UNION}(x_n, x_{n-1})$	$\frac{n-1}{2}$
	$\Theta(n^2)$ 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.

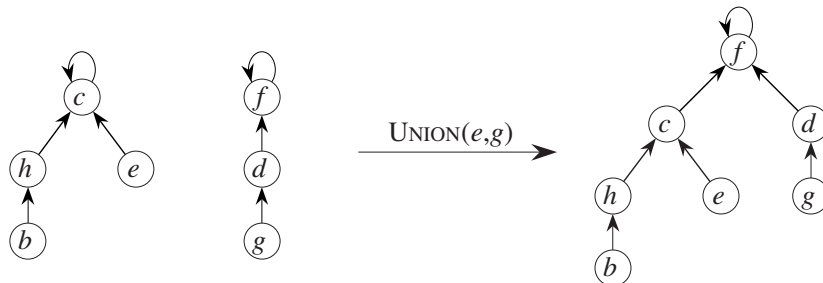
■ (theorem)

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.



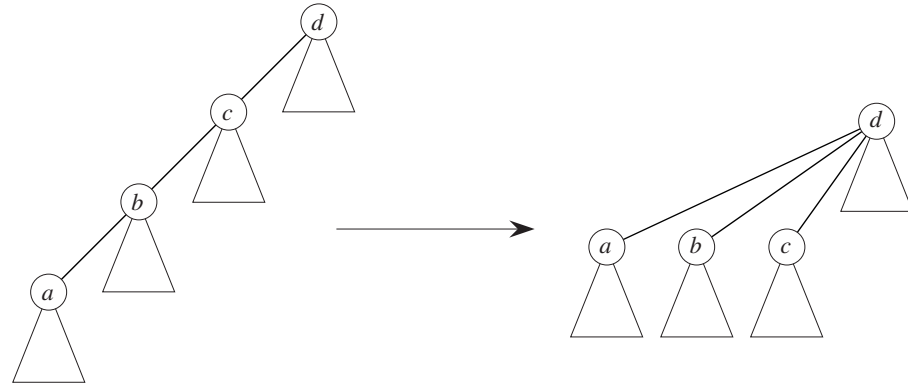
- MAKE-SET: make a single-node tree.
- UNION: make one root a child of the other.
- FIND-SET: follow pointers to the root.

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 $x \neq x.p$ // not the root?
 $x.p = \text{FIND-SET}(x.p)$ // 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.

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

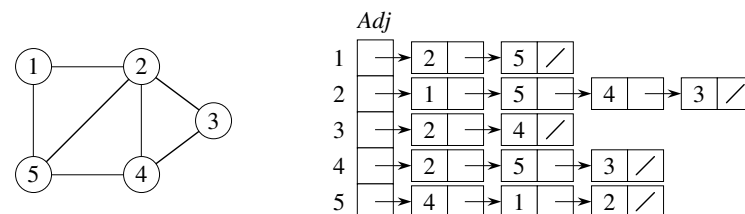
Array Adj of $|V|$ lists, one per vertex.

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: $w : E \rightarrow \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

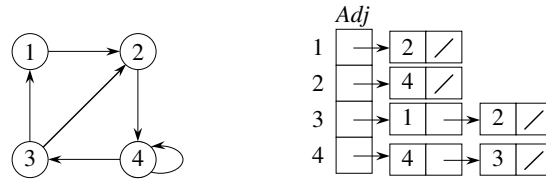
Space: $\Theta(V + E)$.

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 $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u : $\Theta(V)$.

Time: to determine 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 breadth 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  $u \in V - \{s\}$ 
     $u.d = \infty$ 
     $u.\pi = \text{NIL}$ 
   $s.d = 0$ 
   $Q = \emptyset$ 
  ENQUEUE( $Q, s$ )
  while  $Q \neq \emptyset$ 
     $u = \text{DEQUEUE}(Q)$ 
    for each vertex  $v$  in  $G.\text{Adj}[u]$  // 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.

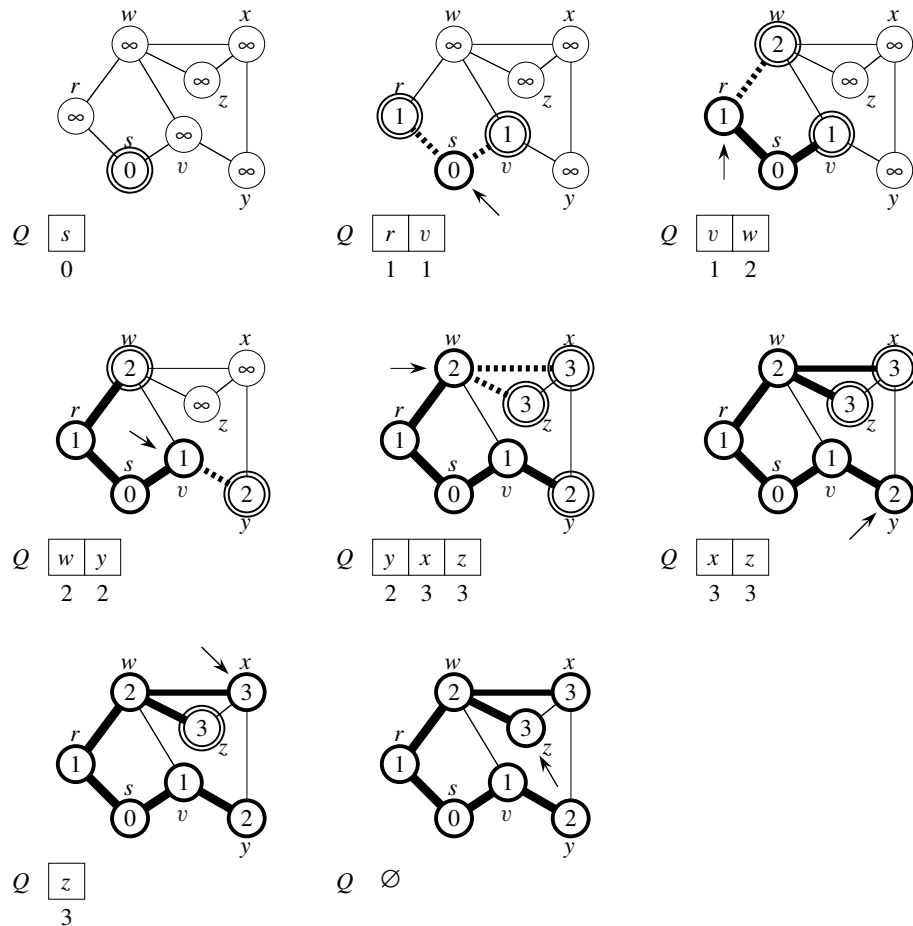
```

[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.

$k \quad k \quad k \quad \dots \quad k \quad k+1 \quad k+1 \quad \dots \quad k+1$

- 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  $v.\pi == \text{NIL}$ 
    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:

- $v.d = \text{discovery time}$
- $v.f = \text{finish time}$

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  $u \in G.V$ 
   $u.color = \text{WHITE}$ 
   $u.\pi = \text{NIL}$ 
 $time = 0$ 
for each vertex  $u \in G.V$ 
  if  $u.color == \text{WHITE}$ 
    DFS-VISIT( $G, u$ )

```


- 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 $u \rightsquigarrow v$ consisting of only white vertices. (Except for u , which was *just* colored gray.)

Classification of edges

- **Tree edge:** in the depth-first forest. Found by exploring (u, v) .
- **Back edge:** (u, v) , where u is a descendant of v .
- **Forward edge:** (u, v) , where v is a descendant of u , but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

[Now label the example from above with edge types.]

In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Classify by the first type above that matches.

Theorem

[Proof omitted.]

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.

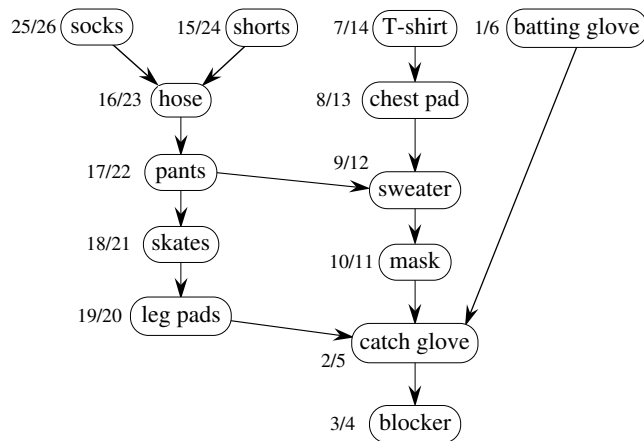
Good for modeling processes and structures that have a **partial order**:

- $a > b$ and $b > c \Rightarrow a > c$.
- But may have a and b such that neither $a > b$ nor $b > a$.

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

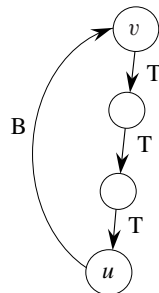
Dag of dependencies for putting on goalie equipment for ice hockey: [Leave on board, but show without discovery and finish times. Will put them in later.]

**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 $v \rightsquigarrow u \rightarrow v$ is a cycle.

\Leftarrow : Show that cycle \Rightarrow back edge.

Suppose G contains cycle c . Let v be the first vertex discovered in c , and let (u, v) be the preceding edge in c . At time $v.d$, 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, $v.f < u.f$. ■

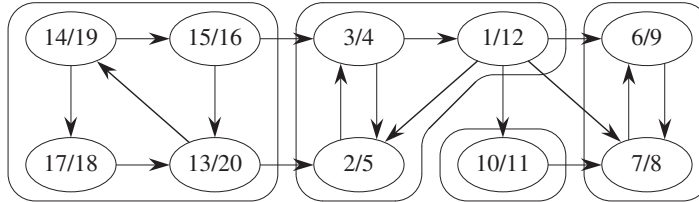
Strongly connected components

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 $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example

[Don't show discovery/finish times yet.]



Algorithm uses $G^T = \text{transpose of } G$.

- $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
- G^T is G with all edges reversed.

Can create G^T in $\Theta(V + E)$ time if using adjacency lists.

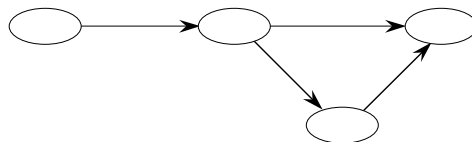
Observation

G and G^T have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^T .)

Component graph

- $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$.
- V^{SCC} has one vertex for each SCC in G .
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G .

For our example:



Lemma

G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G , let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G . Then there cannot also be a path $v' \rightsquigarrow v$ in G .

Proof Suppose there is a path $v' \rightsquigarrow v$ in G . Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G . Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ■ (lemma)

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 $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof $(u, v) \in E^T \Rightarrow (v, u) \in E$. 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 $(G^T)^{\text{SCC}}$ in reverse of topologically sorted order.

[The book has a formal proof.]

Lecture Notes for Chapter 21:

Minimum Spanning Trees

Chapter 21 overview

Problem

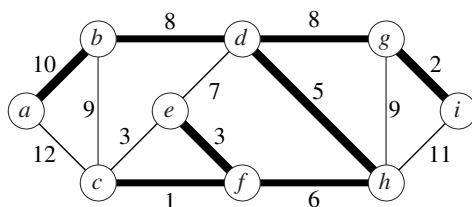
- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- **Goal:** Repair enough (and no more) roads such that
 1. everyone stays connected: can reach every house from all other houses, and
 2. total repair 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

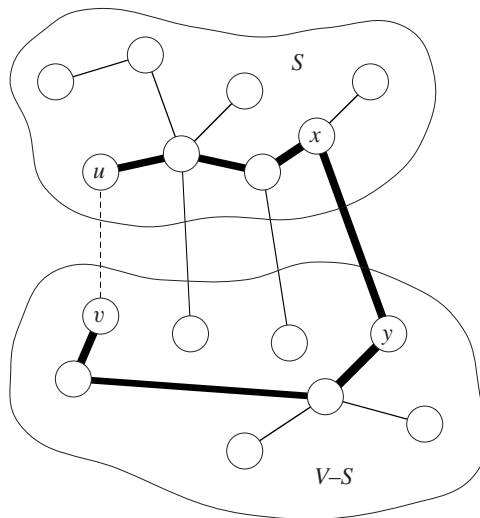
Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A , and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A .

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.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T), \end{aligned}$$

since $w(u, v) \leq w(x, y)$. Since T' is a spanning tree, $w(T') \leq w(T)$, and T is an MST, then T' must be an MST.

Need to show that (u, v) is safe for A :

- $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$.
- $A \cup \{(u, v)\} \subseteq T'$.
- Since T' is an MST, (u, v) is safe for A . ■ (theorem)

So, in **GENERIC-MST**:

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the **for** loop iterates $|V| - 1$ times. Equivalently, after adding $|V| - 1$ safe edges, we're down to just one component.

Corollary

If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component in G_A (i.e., (u, v) is a light edge crossing the cut $(V_C, V - V_C)$), then (u, v) is safe for A .

Proof Set $S = V_C$ in the theorem. ■ (corollary)

This 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. $w : E \rightarrow \mathbb{R}$.

- 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:

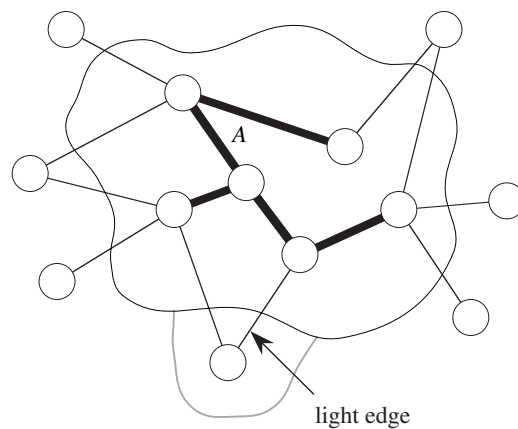
$$O((V + E) \alpha(V)) + O(E \lg E).$$

- 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 $O(E \lg E)$.

- $|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.
- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E \alpha(V))$, which is almost linear.)

Prim's algorithm

- Builds one tree, so A is always a tree.
- Starts from an arbitrary “root” r .
- At each step, find a light edge 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 $Q \neq \emptyset$

$u = \text{EXTRACT-MIN}(Q)$ // 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: $|V|$ EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
 $\leq |E|$ 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.

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 $w : E \rightarrow \mathbb{R}$

Weight of path $p = \langle v_0, v_1, \dots, v_k \rangle$

$$= \sum_{i=1}^k w(v_{i-1}, v_i)$$

= sum of edge weights on path p .

Shortest-path weight u to v :

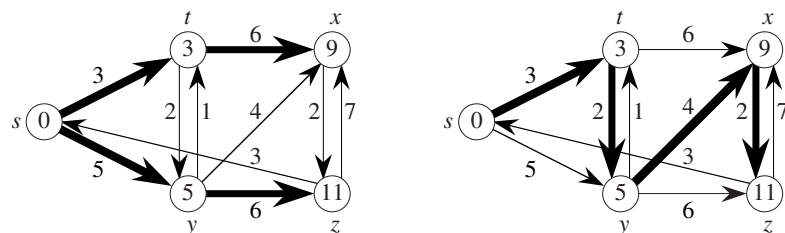
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there exists a path } u \rightsquigarrow v, \\ \infty & \text{otherwise.} \end{cases}$$

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.

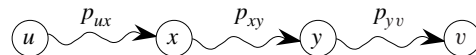
- If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
- But OK if the negative-weight cycle is not reachable from the source.
- Some algorithms work only if there are no negative-weight edges in the graph. We'll be clear when they're allowed and not allowed.

Optimal substructure

Lemma

Any subpath of a shortest path is a shortest path.

Proof Cut-and-paste.



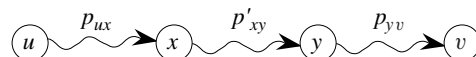
Suppose this path p is a shortest path from u to v .

Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \xrightarrow{p'_{xy}} y$.

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 \Rightarrow we can get a shorter path by omitting the cycle.
- 0-weight: no reason to use them \Rightarrow assume that our solutions won't use them.

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

- $v.d = \delta(s, v)$.
 - 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, $v.\pi = \text{NIL}$.
 - π induces a tree—**shortest-path tree**.
 - We won't prove properties of π in lecture—see text.

Initialization

All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE.

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$

$v.d = \infty$

$v.\pi = \text{NIL}$

$s.d = 0$

Relaxing an edge (u, v)

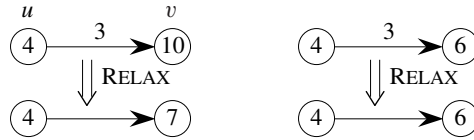
Can the shortest-path estimate for v be improved by going through u and taking (u, v) ?

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 $s \rightsquigarrow v$ is \leq weight of any path $s \rightsquigarrow v$. Path $s \rightsquigarrow u \rightarrow v$ is a path $s \rightsquigarrow v$, and if we use a shortest path $s \rightsquigarrow u$, its weight is $\delta(s, u) + w(u, v)$. ■

Upper-bound property: Always have $v.d \geq \delta(s, v)$ 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,

$$\begin{aligned}
 v.d &< \delta(s, v) \\
 &\leq \delta(s, u) + w(u, v) \quad (\text{triangle inequality}) \\
 &\leq u.d + w(u, v) \quad (v \text{ is first violation}) \\
 \Rightarrow v.d &< u.d + w(u, v) .
 \end{aligned}$$

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 $s \rightsquigarrow u \rightarrow v$ 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 $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k . If the edges of p are relaxed, *in the order*, $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, 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

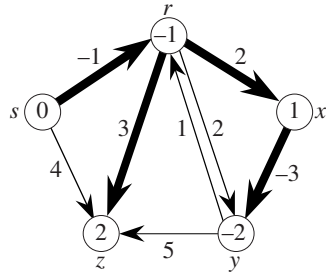
- Allows negative-weight edges.
- Computes $v.d$ and $v.\pi$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s , FALSE otherwise.

```

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 $p = \langle v_0, v_1, \dots, v_k \rangle$ 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) .
- k th 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 $c = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = v_k$, reachable from s .

$$\text{Then } \sum_{i=1}^k w(v_{i-1}, v_i) < 0 .$$

Suppose (for contradiction) that BELLMAN-FORD returns TRUE.

Then $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \dots, k$.

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 $\sum_{i=1}^k v_i \cdot d$ and $\sum_{i=1}^k v_{i-1} \cdot d \Rightarrow$

$$0 \leq \sum_{i=1}^k w(v_{i-1}, v_i) \ .$$

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

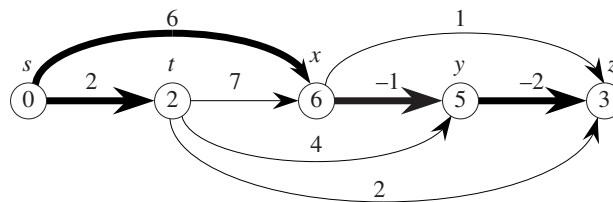
INITIALIZE-SINGLE-SOURCE(G, s)

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.

\Rightarrow Edges on any shortest path are relaxed in order.

\Rightarrow By path-relaxation property, correct. ■

Dijkstra's algorithm

No negative-weight *edges*.

Essentially a weighted version of breadth-first search.

- Instead of a FIFO queue, uses a priority queue.
- Keys are shortest-path weights ($v.d$).
- 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 $Q \neq \emptyset$

$u = \text{EXTRACT-MIN}(Q)$

$S = S \cup \{u\}$

for each vertex v in $G.Adj[u]$

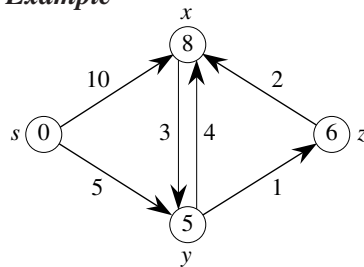
 RELAX(u, v, w)

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 $v \in V$.

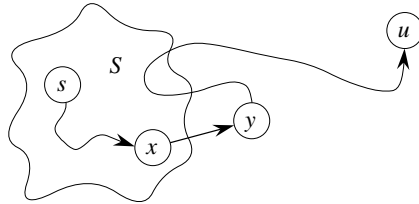
The proof is by induction on the number of iterations of the **while** loop, i.e., on $|S|$. The bases are for $|S| = 0$, so that $S = \emptyset$ and the claim is trivially true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = \delta(s, s) = 0$.

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 $\Rightarrow \delta(s, y) \leq \delta(s, u)$.
- How we chose $u \Rightarrow u.d \leq y.d$ at the time u is extracted from $V - S$.
- Upper-bound property $\Rightarrow \delta(s, u) \leq u.d$.
- $x \in S \Rightarrow x.d = \delta(s, x)$. 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 $\delta(s, y) \leq \delta(s, u) \leq u.d \leq y.d$ 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.

$\leq |E|$ 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 $x_j - x_i \leq b_k$.

- x 's are variables, $1 \leq i, j \leq n$,
- b 's are constants, $1 \leq k \leq m$.

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

$$x_1 - x_2 \leq 5$$

$$x_1 - x_3 \leq 6$$

$$x_2 - x_4 \leq -1$$

$$x_3 - x_4 \leq -2$$

$$x_4 - x_1 \leq -3$$

Solution: $x = (0, -4, -5, -3)$

Also: $x = (5, 1, 0, 2) = [\text{above solution}] + 5$

Lemma

If x is a feasible solution, then so is $x + d$ for any constant d .

Proof x is a feasible solution $\Rightarrow x_j - x_i \leq b_k$ for all i, j, k
 $\Rightarrow (x_j + d) - (x_i + d) \leq b_k$.

■ (lemma)

Example application

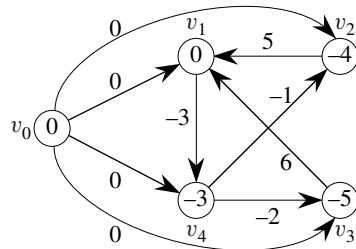
x_i are times when events are to occur.

- 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 $x_j - x_i \geq 0$, which is equivalent to $x_i - x_j \leq 0$, and $x_j - x_i \leq b_k$.
- What if x_j must occur at least b_k time units after x_i ? Get $x_j - x_i \geq b_k$, which is equivalent to $x_i - x_j \leq -b_k$.

Constraint graph

$G = (V, E)$, weighted, directed.

- $V = \{v_0, v_1, v_2, \dots, v_n\}$: one vertex per variable + v_0
- $E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\} \cup \{(v_0, v_1), (v_0, v_2), \dots, (v_0, v_n)\}$
- $w(v_0, v_j) = 0$ for all $j = 1, 2, \dots, n$
- $w(v_i, v_j) = b_k$ if $x_j - x_i \leq b_k$



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

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \dots, \delta(v_0, v_n))$$

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 $\Rightarrow x = (\delta(v_0, v_1), \delta(v_0, v_2), \dots, \delta(v_0, v_n))$ is a feasible solution.

Need to show that $x_j - x_i \leq b_k$ for all constraints. Use

$$x_j = \delta(v_0, v_j)$$

$$x_i = \delta(v_0, v_i)$$

$$b_k = w(v_i, v_j) .$$

By the triangle inequality,

$$\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$$

$$x_j \leq x_i + b_k$$

$$x_j - x_i \leq b_k .$$

Therefore, feasible.

2. Show negative-weight cycles \Rightarrow no feasible solution.

Without loss of generality, let a negative-weight cycle be $c = \langle v_1, v_2, \dots, v_k \rangle$, where $v_1 = v_k$. (v_0 can't be on c , since v_0 has no entering edges.) c corresponds to the constraints

$$x_2 - x_1 \leq w(v_1, v_2) ,$$

$$x_3 - x_2 \leq w(v_2, v_3) ,$$

$$\vdots$$

$$x_{k-1} - x_{k-2} \leq w(v_{k-2}, v_{k-1}) ,$$

$$x_k - x_{k-1} \leq w(v_{k-1}, v_k) .$$

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 \Rightarrow no feasible solution.
BELLMAN-FORD returns TRUE \Rightarrow set $x_i = \delta(v_0, v_i)$ for all $i = 1, 2, \dots, n$.

Lecture Notes for Chapter 23:

All-Pairs Shortest Paths

Chapter 23 overview

Given a directed graph $G = (V, E)$, weight function $w : E \rightarrow \mathbb{R}$, $|V| = n$. Assume that the vertices are numbered $1, 2, \dots, n$.

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:

- $O(V^2 E)$ —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 $i \rightsquigarrow j$ that contains $\leq r$ edges.

- $r = 0$
 \Rightarrow there is a shortest path $i \rightsquigarrow j$ with $\leq r$ edges if and only if $i = j$
 $\Rightarrow l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$
- $r \geq 1$
 $\Rightarrow l_{ij}^{(r)} = \min \left\{ l_{ij}^{(r-1)}, \min \{ l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n \} \right\}$
(k ranges over all possible predecessors of j)
 $= \min \{ l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n \}$ (since $w_{jj} = 0$ for all j).
- 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 $i \rightsquigarrow j$ must be w_{ij} .
 And the math works out, too:

$$\begin{aligned} l_{ij}^{(1)} &= \min \{ l_{ik}^{(0)} + w_{kj} : 1 \leq k \leq n \} \\ &= l_{ii}^{(0)} + w_{ij} \quad (l_{ii}^{(0)} \text{ is the only non-}\infty \text{ among } l_{ik}^{(0)}) \\ &= w_{ij}. \end{aligned}$$

All simple shortest paths contain $\leq n - 1$ edges

$$\Rightarrow \delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \dots$$

Compute a solution bottom-up

Compute $L^{(1)}, L^{(2)}, \dots, L^{(n-1)}$.

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 ∞ .

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 $l_{ij}^{(r)} = \min \{l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n\}$:

$l^{(r-1)} \rightarrow a$

$w \rightarrow b$

$l^{(r)} \rightarrow c$

$\min \rightarrow +$

$+ \rightarrow \cdot$

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 $L^{(1)}, L^{(2)}, L^{(3)}, \dots, L^{(n-2)}$.

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, \dots

If we knew $A^r = A^{n-1}$ for all $r \geq n - 1$, could just finish with A^r , where r is the smallest power of 2 that is $\geq n - 1$ ($r = 2^{\lceil \lg(n-1) \rceil}$).

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

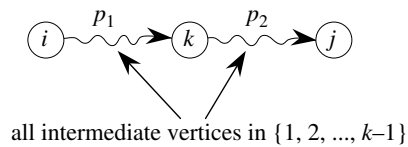
A different dynamic-programming approach.

For path $p = \langle v_1, v_2, \dots, v_l \rangle$, 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 $\{1, 2, \dots, k\}$.

Consider a shortest path $i \overset{p}{\rightsquigarrow} j$ with all intermediate vertices in $\{1, 2, \dots, k\}$:

- If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, 2, \dots, k-1\}$.
- If k is an intermediate vertex:

**Recursive formulation**

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min \{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k \geq 1. \end{cases}$$

Have $d_{ij}^{(0)} = w_{ij}$ because can't have intermediate vertices $\Rightarrow \leq 1$ edge.

Want $D^{(n)} = (d_{ij}^{(n)})$, 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 $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix

for $i = 1$ **to** n

for $j = 1$ **to** n

$d_{ij}^{(k)} = \min \{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$

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 $\Pi^{(k)} = (\pi_{ij}^{(k)})$ for $k = 0, 1, \dots, n$.

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 $i \rightsquigarrow k \rightsquigarrow j$ 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 $G^* = (V, E^*)$.

- $E^* = \{(i, j) : \text{there is a path } i \rightsquigarrow j \text{ in } G\}$.

Could assign weight of 1 to each edge, then run FLOYD-WARSHALL.

- If $d_{ij} < n$, then there is a path $i \rightsquigarrow j$.
- Otherwise, $d_{ij} = \infty$ and there is no path.

Simpler way

Substitute other values and operators in FLOYD-WARSHALL.

- Use unweighted adjacency matrix
- $\min \rightarrow \vee$ (OR)
- $+$ $\rightarrow \wedge$ (AND)
- $t_{ij}^{(k)} = \begin{cases} 1 & \text{if there is path } i \rightsquigarrow j \text{ with all intermediate vertices in } \{1, 2, \dots, k\}, \\ 0 & \text{otherwise.} \end{cases}$
- $t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E, \\ 1 & \text{if } i = j \text{ or } (i, j) \in E. \end{cases}$
- $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$.

```

TRANSITIVE-CLOSURE( $G, n$ )
  let  $T^{(0)} = (t_{ij}^{(0)})$  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  $T^{(k)} = (t_{ij}^{(k)})$  be a new  $n \times n$  matrix
    for  $i = 1$  to  $n$ 
      for  $j = 1$  to  $n$ 
         $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
  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 \hat{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 \hat{w} .
2. For all $(u, v) \in E$, $\hat{w}(u, v) \geq 0$.

Property (1) says that it suffices to find shortest paths with \hat{w} . Property (2) says we can do so by running Dijkstra's algorithm from each vertex.

How to come up with \hat{w} ?

Lemma shows it's easy to get property (1):

Lemma (Reweighting doesn't change shortest paths)

Given a directed, weighted graph $G = (V, E)$, $w : E \rightarrow \mathbb{R}$. Let h be any function such that $h : V \rightarrow \mathbb{R}$. For all $(u, v) \in E$, define

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) .$$

Let $p = \langle v_0, v_1, \dots, v_k \rangle$ 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 \hat{w} . (Formally, $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w} = \hat{\delta}(v_0, v_k)$, where $\hat{\delta}$ is the shortest-path weight with \hat{w} .)

Also, G has a negative-weight cycle with w if and only if G has a negative-weight cycle with \hat{w} .

Proof First, show that $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$:

$$\begin{aligned} \hat{w}(p) &= \sum_{i=1}^k \hat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^k (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^k w(v_{i-1}, v_i) + h(v_0) - h(v_k) \quad (\text{sum telescopes}) \\ &= w(p) + h(v_0) - h(v_k) . \end{aligned}$$

Therefore, any path $v_0 \rightsquigarrow^p v_k$ has $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. 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 \hat{w} .

Now show there exists a negative-weight cycle with w if and only if there exists a negative-weight cycle with \hat{w} :

- Let cycle $c = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = v_k$.

- Then

$$\begin{aligned} \hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) \quad (\text{since } v_0 = v_k) . \end{aligned}$$

Therefore, c has a negative-weight cycle with w if and only if it has a negative-weight cycle with \hat{w} . ■ (lemma)

So, now to get property (2), we just need to come up with a function $h : V \rightarrow \mathbb{R}$ such that when we compute $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$, 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

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0.$$

Proof of claim By the triangle inequality,

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

$$h(v) \leq h(u) + w(u, v).$$

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 $(u, v) \in G'.E$

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

 let $D = (d_{uv})$ be a new $n \times n$ matrix

for each vertex $u \in G.V$

 run DIJKSTRA(G, \hat{w}, u) to compute $\hat{\delta}(u, v)$ for all $v \in G.V$

for each vertex $v \in G.V$

$$d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$$

return D

Time

- $\Theta(V + E)$ to compute G' .
- $O(VE)$ to run BELLMAN-FORD.
- $\Theta(E)$ to compute \hat{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: $O(V^2 \lg V + VE)$.

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.

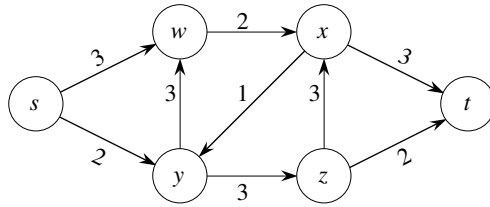
Each edge (u, v) has a **capacity** $c(u, v) \geq 0$.

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 $s \rightsquigarrow v \rightsquigarrow t$ for all $v \in V$, so that each vertex lies on a path from source to sink.

Example: *[Edges are labeled with capacities.]*

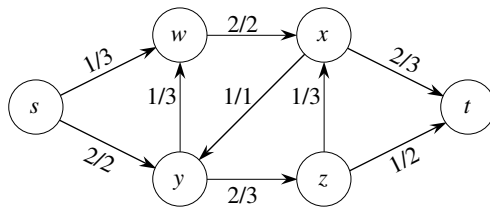
**Flow**

A function $f : V \times V \rightarrow \mathbb{R}$ satisfying

- **Capacity constraint:** For all $u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$,
- **Flow conservation:** For all $u \in V - \{s, t\}$, $\underbrace{\sum_{v \in V} f(v, u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u, v)}_{\text{flow out of } u}$.

$$\text{Equivalently, } \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0.$$

[Add flows to previous example. Edges here are labeled as flow/capacity. Leave on board.]



- Note that all flows are \leq capacities.
- Verify flow conservation by adding up flows at a couple of vertices.
- Note that all flows = 0 is legitimate.

$$\begin{aligned} \text{Value of flow } f &= |f| \\ &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \\ &= \text{flow out of source} - \text{flow into source} . \end{aligned}$$

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, \dots, s_m , create a single **supersource** s with edges (s, s_i) for $i = 1, \dots, m$ 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 $S = \{s, w, y\}$, $T = \{x, z, t\}$.

$$\begin{aligned} f(S, T) &= \underbrace{f(w, x) + f(y, z)}_{\text{from } S \text{ to } T} - \underbrace{f(x, y)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3 . \end{aligned}$$

$$\begin{aligned} c(S, T) &= \underbrace{c(w, x) + c(y, z)}_{\text{from } S \text{ to } T} \\ &= 2 + 3 \\ &= 5 . \end{aligned}$$

Now consider the cut $S = \{s, w, x, y\}$, $T = \{z, t\}$.

$$\begin{aligned} f(S, T) &= \underbrace{f(x, t) + f(y, z)}_{\text{from } S \text{ to } T} - \underbrace{f(z, x)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3 . \end{aligned}$$

$$\begin{aligned} c(S, T) &= \underbrace{c(x, t) + c(y, z)}_{\text{from } S \text{ to } T} \\ &= 3 + 3 \\ &= 6 . \end{aligned}$$

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 :

$$\begin{aligned} |f| &= \sum_{v \in S} \sum_{u \in S} f(u, v) + \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &\quad + \left(\sum_{v \in S} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) \right). \end{aligned}$$

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 \leq capacity of any cut.

[Leave on board.]

Proof Let (S, T) be any cut, f be any flow.

$$\begin{aligned}
 |f| &= f(S, T) && \text{(lemma)} \\
 &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) && \text{(definition of } f(S, T)) \\
 &\leq \sum_{u \in S} \sum_{v \in T} f(u, v) && (f(v, u) \geq 0) \\
 &\leq \sum_{u \in S} \sum_{v \in T} c(u, v) && \text{(capacity constraint)} \\
 &= c(S, T) . && \text{(definition of } c(S, T)) \quad \blacksquare \text{ (corollary)}
 \end{aligned}$$

Therefore, maximum flow \leq 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**,

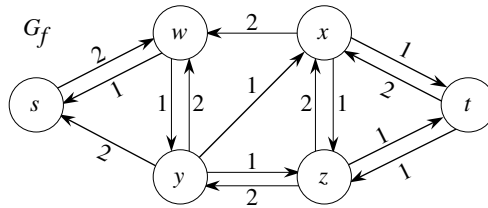
$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E , \\ f(v, u) & \text{if } (v, u) \in E , \\ 0 & \text{otherwise (i.e., } (u, v), (v, u) \notin E \text{)} . \end{cases}$$

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, $|E_f| \leq 2|E|$.

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 $V \times V \rightarrow \mathbb{R}$:

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E, \\ 0 & \text{otherwise} \end{cases}$$

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 .

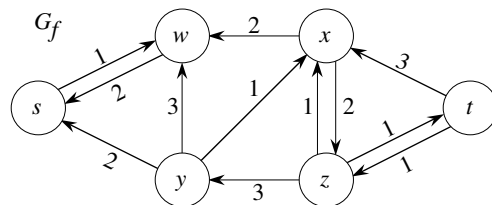
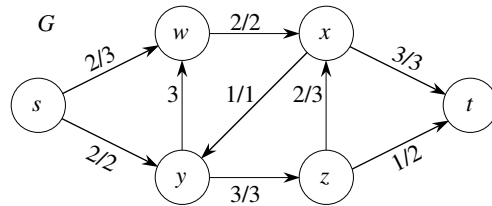
How much more flow can be pushed from s to t along augmenting path p ? That is the **residual capacity** of p :

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is on } p\}.$$

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 $f_p : V \times V \rightarrow \mathbb{R}$:

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p, \\ 0 & \text{otherwise.} \end{cases}$$

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) \Rightarrow (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) \Rightarrow (3): Suppose G_f has no augmenting path. Define

$$S = \{v \in V : \text{there exists a path } s \rightsquigarrow v \text{ in } G_f\},$$

$$T = V - S.$$

Must have $t \in T$; otherwise there is an augmenting path.

Therefore, (S, T) is a cut.

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) \Rightarrow (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 $s \rightsquigarrow t$ in G_f , with all edge weights = 1.

Edmonds-Karp runs in $O(VE^2)$ time.

To prove, need to look at distances to vertices in G_f .

Let $\delta_f(u, v)$ = shortest path distance u to v in G_f , with unit edge weights.

Lemma

For all $v \in V - \{s, t\}$, $\delta_f(s, v)$ increases monotonically with each flow augmentation.

Proof Suppose there exists $v \in V - \{s, t\}$ such that 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 $s \rightsquigarrow u \rightarrow v$, 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

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 \quad (\text{triangle inequality}) \\ &\leq \delta_{f'}(s, u) + 1 \\ &= \delta_{f'}(s, v), \end{aligned}$$

contradicting $\delta_{f'}(s, v) < \delta_f(s, v)$.

■ (claim)

How can $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$?

The augmentation must increase flow v to u .

Since Edmonds-Karp augments along shortest paths, the shortest path s to u in G_f has (v, u) as its last edge.

Therefore,

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 \\ &= \delta_{f'}(s, v) - 2, \end{aligned}$$

contradicting $\delta_{f'}(s, v) < \delta_f(s, v)$.

Therefore, v cannot exist.

■ (lemma)

Theorem

Edmonds-Karp performs $O(VE)$ augmentations.

Proof Suppose p is an augmenting path and $c_f(u, v) = c_f(p)$. Then call (u, v) a **critical** edge in G_f , and it disappears from the residual network after augmenting along p .

≥ 1 edge on any augmenting path is critical.

Will show that each of the $|E|$ edges can become critical $\leq |V|/2$ 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) disappears from the residual network. This edge cannot reappear in the residual network until flow from u to v decreases, which happens only if (v, u) is on an augmenting path in $G_{f'}$: $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$. (f' is flow when this occurs.)

By lemma, $\delta_f(s, v) \leq \delta_{f'}(s, v) \Rightarrow$

$$\begin{aligned} \delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2. \end{aligned}$$

Therefore, from the time (u, v) becomes critical to the next time, distance of u from s increases by ≥ 2 . Initially, distance to u is ≥ 0 , and augmenting path can't have s, u , and t as intermediate vertices.

Therefore, until u becomes unreachable from source, its distance is $\leq |V| - 2$
 \Rightarrow 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.]

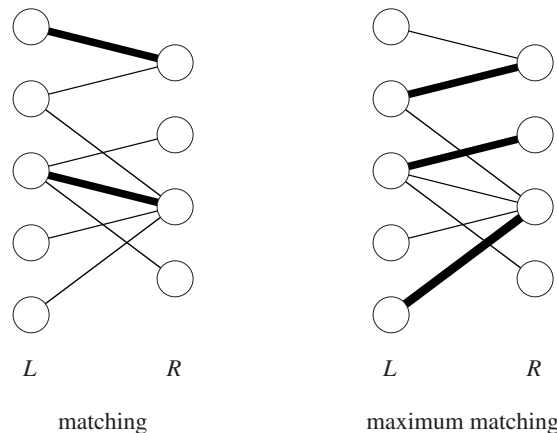
Maximum bipartite matching

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' .)



[Edges in matchings are drawn with heavy lines.]

Problem

Given a bipartite graph (with the partition), find a maximum matching.

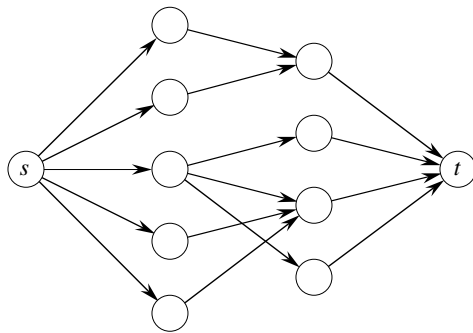
Application

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.]

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 **un-matched**.
- **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.

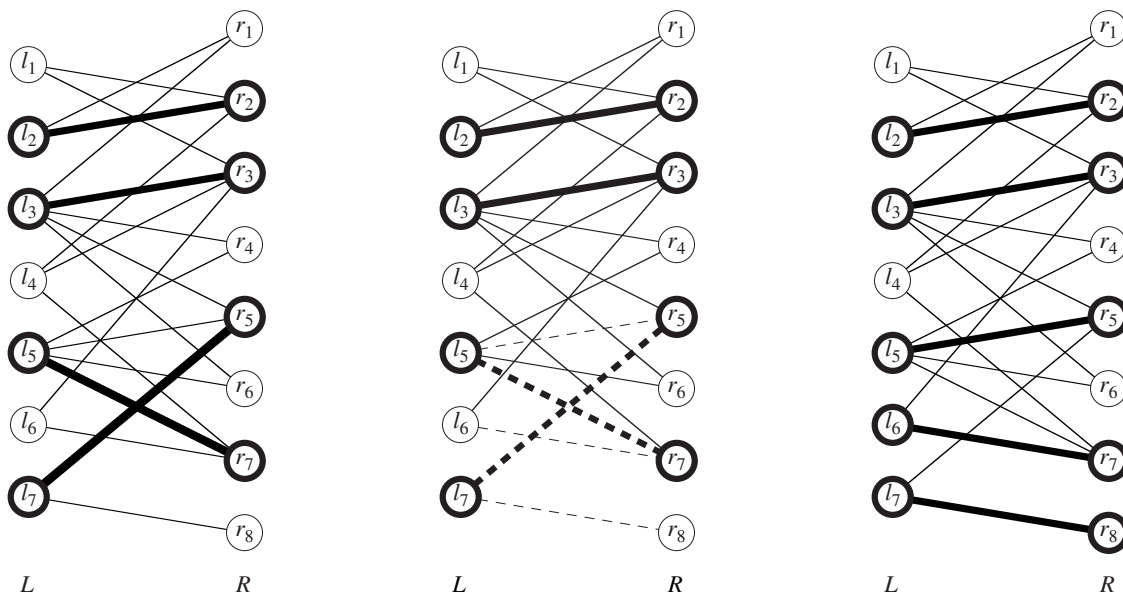
\oplus is commutative and associative. Identity is the empty set.

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, \dots, P_k be vertex-disjoint augmenting paths. Then $M' = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ is a matching with $|M'| = |M| + k$.

Proof Induction on i using the lemma shows that $M \oplus (P_1 \cup P_2 \cup \dots \cup P_{i-1})$ is a matching with $|M| + i - 1$ edges and P_i is an augmenting path with respect to $M \oplus (P_1 \cup P_2 \cup \dots \cup P_{i-1})$. ■

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)

$M = \emptyset$

repeat

 let $\mathcal{P} = \{P_1, P_2, \dots, P_k\}$ be a maximal set of vertex-disjoint
 shortest M -augmenting paths

$M = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$

until $\mathcal{P} == \emptyset$

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

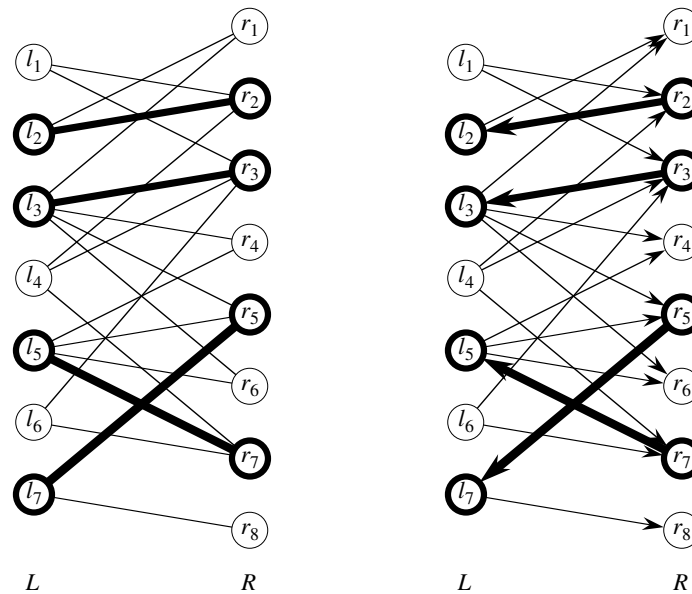
Think of an M -augmenting path P as starting at some unmatched vertex in L , traversing an odd number of edges, and ending at some unmatched vertex in R .

- Edges in P going $L \rightarrow R$ belong to $E - M$.
- Edges in P going $R \rightarrow L$ belong to M .

Create G_M by directing the edges as above: $G_M = (V, E_M)$,

$$E_M = \{(l, r) : l \in L, r \in R, \text{ and } (l, r) \in E - M\} \quad (\text{edges from } L \text{ to } R) \\ \cup \{(r, l) : r \in R, l \in L, \text{ and } (l, r) \in M\} \quad (\text{edges from } R \text{ to } L).$$

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.

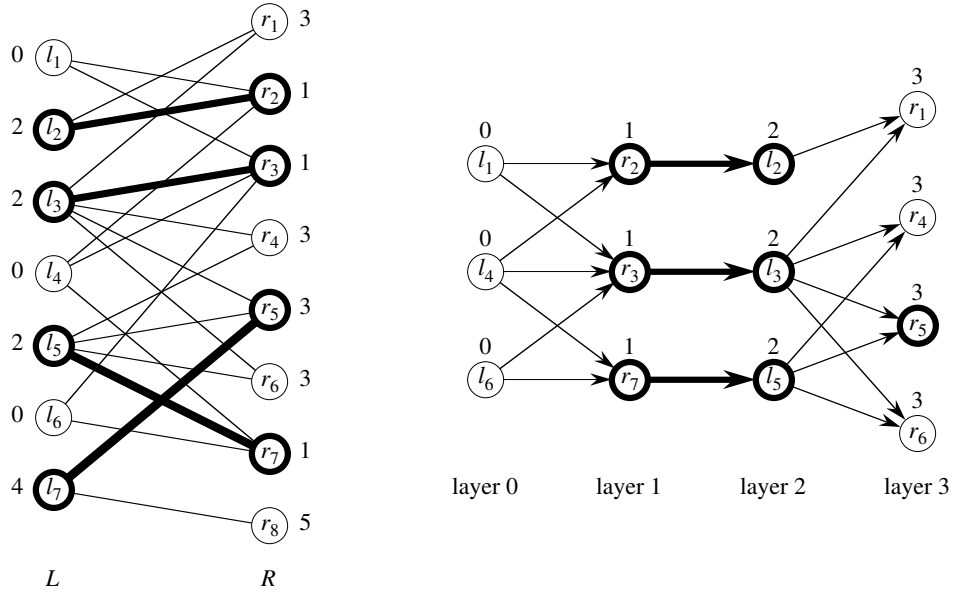
- If q = smallest distance in G_M from any unmatched vertex in L to an unmatched vertex in R , then the last layer is numbered q .
- Vertices with distance $> q$ do not appear in H .

$$E_H = \{(l, r) \in E_M : r.d \leq q \text{ and } r.d = l.d + 1\} \cup \{(r, l) \in E_M : l.d \leq q\},$$

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 .

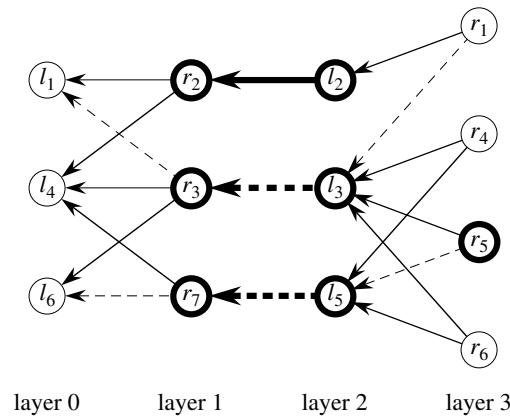


Run BFS on G_M to determine vertex distances, but starting from all unmatched vertices in L . In BFS code, replace the root vertex (s) by the set of unmatched vertices in L .

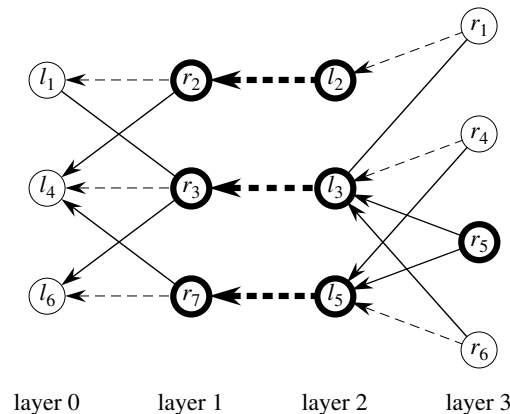
Every path in H from a vertex in layer 0 to an unmatched vertex in layer q corresponds to a shortest M -augmenting path in G . (Use the undirected versions of the directed edges in H .) Every shortest M -augmenting path in G is in H .

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,
- $\mathcal{P} = \{P_1, P_2, \dots, P_k\}$ be a maximal set of vertex-disjoint shortest paths of length q ,
- $M' = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$, 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, \dots, P_k

$\Rightarrow P$ is also an M -augmenting path.

P disjoint from P_1, \dots, 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

$$A = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P.$$

Proof of claim

$$\begin{aligned} A &= M \oplus M' \oplus P \\ &= M \oplus (M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)) \oplus P \\ &= (M \oplus M) \oplus (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P \quad (\text{associativity of } \oplus) \\ &= \emptyset \oplus (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P \quad (X \oplus X = \emptyset \text{ for all } X) \\ &= (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P \quad (\emptyset \oplus X = X \text{ for all } X). \end{aligned}$$

■ (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)

$A = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$ and P shares ≥ 1 edge with some $P_i \in \mathcal{P}$

$\Rightarrow |A| < |P_1 \cup P_2 \cup \dots \cup P_k| + |P|$.

Then,

$$\begin{aligned} kq + q &\leq |A| \\ &< |P_1 \cup P_2 \cup \dots \cup P_k| + |P| \\ &= kq + |P|, \end{aligned}$$

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. \Rightarrow 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 \Rightarrow number of additional iterations before reaching the maximum matching is at most

$$\frac{|V|}{\lceil \sqrt{|V|} \rceil + 1} < \frac{|V|}{\sqrt{|V|}} \\ = \sqrt{|V|}.$$

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:*

$$l.h = \max \{w(l, r) : r \in R\} \text{ for all } l \in L,$$

$$r.h = 0 \text{ for all } r \in R.$$

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. König 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 .

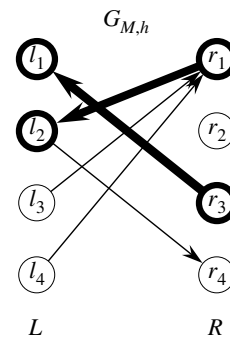
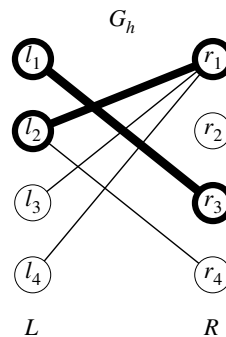
Questions to answer:

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.]

		r_1	r_2	r_3	r_4
h		0	0	0	0
l_1	12	8	5	<u>12</u>	9
l_2	9	<u>9</u>	6	7	<u>9</u>
l_3	9	<u>9</u>	8	8	5
l_4	6	<u>6</u>	2	3	5



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 unmatched vertex in L , ending at an unmatched vertex in R , taking unmatched edges $L \rightarrow R$, and matched edges $R \rightarrow L$. Direct edges accordingly in $G_{M,h} = (V, E_{M,h})$, where

$$E_{M,h} = \{(l, r) : l \in L, r \in R, \text{ and } (l, r) \in E_h - M\} \quad (\text{edges from } L \text{ to } R)$$

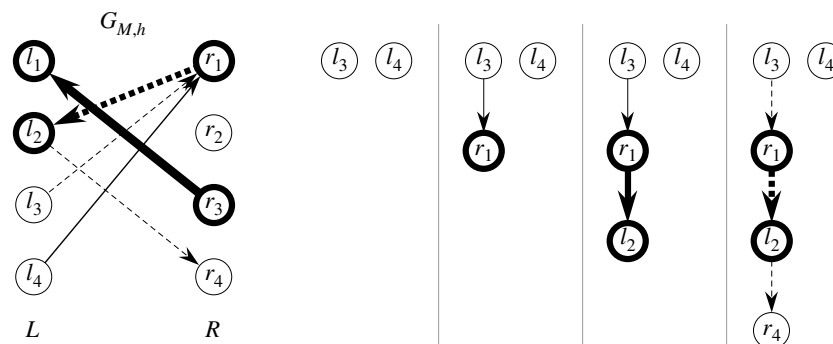
$$\cup \{(r, l) : r \in R, l \in L, \text{ and } (l, r) \in M\} \quad (\text{edges from } R \text{ to } L).$$

[Now show $G_{M,h}$ in the above figure.]

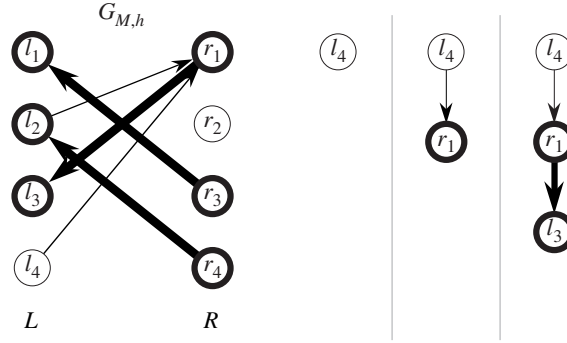
Hungarian algorithm searches for a directed path in $G_{M,h}$ from any unmatched vertex in L to any unmatched vertex in R . Can use any graph-searching method.

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 $r \in F_R$:

$$v.h' = \begin{cases} v.h - \delta & \text{if } v \in F_L, \\ v.h + \delta & \text{if } v \in F_R, \\ v.h & \text{otherwise } (v \in V - V_F). \end{cases}$$

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 definition of δ , ≥ 1 such an edge exists.)
 $l.h' + r.h' = l.h - \delta + r.h$
 $= l.h - (l.h + r.h - w(l, r)) + r.h$
 $= w(l, r)$.

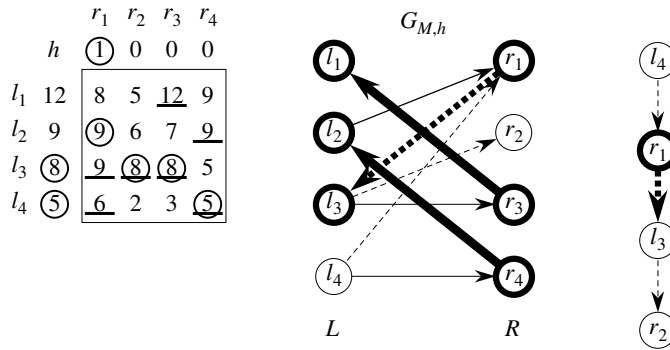
$$\Rightarrow (l, r) \in E_{h'}.$$

$$(l, r) \notin E_h \Rightarrow (l, r) \notin M \Rightarrow (l, r) \text{ is directed } L \rightarrow R \text{ in } E_{M, h'}.$$

■

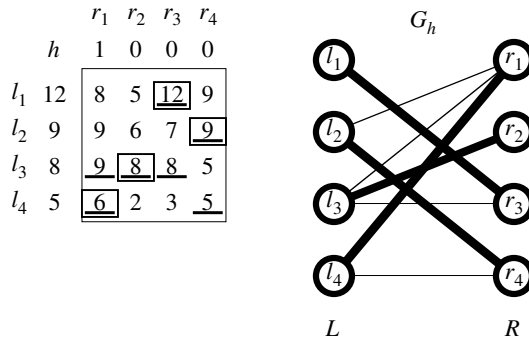
The figure below continues the example. Here, $\delta = 1$, achieved at edges (l_3, r_2) , (l_3, r_3) , and (l_4, r_4) , which enter E_h . Changes are circled in the matrix and h values.

(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 .



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.



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

$P = \text{FIND-AUGMENTING-PATH}(G_{M,h})$

$M = M \oplus P$

 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
             $F_R = F_R \cup \{r\}$ 
         $u = \text{DEQUEUE}(Q)$                 // 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).

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, \dots, 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 $j = 0, 1, \dots, n-1$.

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:

$$\begin{array}{r}
 6x^3 + 7x^2 - 10x + 9 \\
 - 2x^3 \qquad \qquad + 4x - 5 \\
 \hline
 - 30x^3 - 35x^2 + 50x - 45 \quad (\text{multiply } A(x) \text{ by } -5) \\
 24x^4 + 28x^3 - 40x^2 + 36x \quad (\text{multiply } A(x) \text{ by } 4x) \\
 - 12x^6 - 14x^5 + 20x^4 - 18x^3 \quad (\text{multiply } A(x) \text{ by } -2x^3) \\
 \hline
 - 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45
 \end{array}$$

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 $a = (a_0, a_1, \dots, a_{n-1})$.

[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**:

$$A(x_0) = a_0 + x_0 \left(a_1 + x_0 \left(a_2 + \dots + x_0 (a_{n-2} + x_0 (a_{n-1})) \dots \right) \right).$$

Adding two polynomials represented by the coefficient vectors $a = (a_0, a_1, \dots, a_{n-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$ takes $\Theta(n)$ time: just produce the coefficient vector $c = (c_0, c_1, \dots, c_{n-1})$, where $c_j = a_j + b_j$ for $j = 0, 1, \dots, n-1$.

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** $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ such that all of the x_k are distinct and $y_k = A(x_k)$ for $k = 0, 1, \dots, n-1$.

- A polynomial has many different point-value representations, since any set of n distinct points x_0, x_1, \dots, 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, \dots, x_{n-1} and then evaluate $A(x_k)$ for $k = 0, 1, \dots, n-1$. 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 $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ 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 $k = 0, 1, \dots, n-1$. ■

Proof The equations $y_k = A(x_k)$ for $k = 0, 1, \dots, n-1$ 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 $V(x_0, x_1, \dots, x_{n-1})$, 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 $y = (y_0, y_1, \dots, y_{n-1})$, the coefficients $a = (a_0, a_1, \dots, a_{n-1})$ are given by $a = V(x_0, x_1, \dots, x_{n-1})^{-1}y$. ■

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 , $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$, and for B , $\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{n-1}, y'_{n-1})\}$, where A and B are evaluated at the *same* n points, then a point-value representation for C is $\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \dots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$.

\Rightarrow 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 , $\{(x_0, y_0), (x_1, y_1), \dots, (x_{2n-1}, y_{2n-1})\}$, and a corresponding extended point-value representation for B , $\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{2n-1}, y'_{2n-1})\}$, then a point-value representation for C is $\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \dots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}$.

\Rightarrow Multiplying polynomials in point-value form takes just $\Theta(n)$ time.

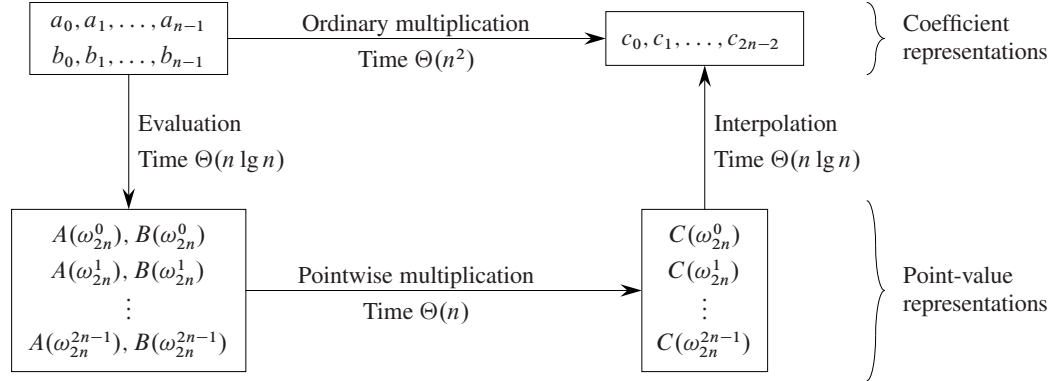
Evaluation at a new point: The simplest approach is to first convert the polynomial from point-value form to coefficient form and then evaluate it at the new point.

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 $2n$.]



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. ■

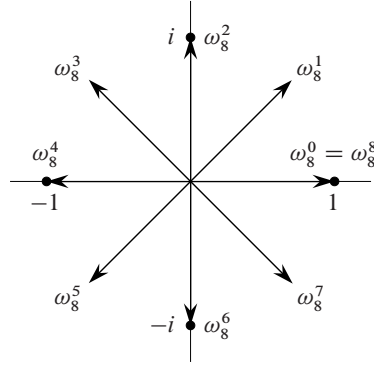
The DFT and FFT

Complex roots of unity

A **complex n th root of unity** is a complex number ω such that $\omega^n = 1$.

- There are exactly n complex n th roots of unity: $e^{2\pi i k/n}$ for $k = 0, 1, \dots, n-1$.
- 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 n th 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 n th roots of unity are powers of ω_n .

The values of $\omega_8^0, \omega_8^1, \dots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity:



The n complex n th roots of unity, $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$, 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 n th 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

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^k \\ &= \omega_n^k. \end{aligned}$$

■

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 n th roots of unity are the $n/2$ complex $(n/2)$ th roots of unity.

Proof [This is the alternative proof given in the textbook.]

$$\omega_n^{n/2} = -1 \Rightarrow \omega_n^{k+n/2} = \omega_n^k \omega_n^{n/2} = -\omega_n^k \Rightarrow (\omega_n^{k+n/2})^2 = (-\omega_n^k)^2 = (\omega_n^k)^2. \quad \blacksquare$$

Lemma (Summation lemma)

For any integer $n \geq 1$ and nonzero integer k not divisible by n , $\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$.

Proof The summation formula for a geometric series applies to complex numbers as well as to reals \Rightarrow

$$\begin{aligned} \sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} \\ &= 0. \end{aligned}$$

$\omega_n^k = 1$ only when k is divisible by n , which the lemma statement prohibits \Rightarrow denominator is not 0. \blacksquare

The DFT

Goal: Evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$ (at the n complex n th roots of unity).
Given in coefficient form: $a = (a_0, a_1, \dots, a_{n-1})$.

Results: y_k , for $k = 0, 1, \dots, n-1$:

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj}. \end{aligned}$$

Vector $y = (y_0, y_1, \dots, y_{n-1})$ is the **discrete Fourier transform (DFT)** of the coefficient vector $a = (a_0, a_1, \dots, a_{n-1})$.

Denote $y = \text{DFT}_n(a)$.

The FFT

Fast Fourier transform (FFT) uses properties of the complex roots of unity to compute $\text{DFT}_n(a)$ 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 $n/2$:

$$A^{\text{even}}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1} ,$$

$$A^{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1} .$$

- 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.

$$A(x) = A^{\text{even}}(x^2) + xA^{\text{odd}}(x^2) ,$$

so that evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ becomes

1. Evaluate the degree-bound $n/2$ polynomials $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ at the points $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$.
2. Combine the results according to $A(x) = A^{\text{even}}(x^2) + xA^{\text{odd}}(x^2)$.

FFT procedure idea:

- By the halving lemma, the list of values $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$ 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 $\text{DFT}_{n/2}$ computations.
- This decomposition is the basis for the FFT procedure, which computes the DFT of an n -element vector $a = (a_0, a_1, \dots, a_{n-1})$, where n is a power of 2.

FFT procedureFFT(a, n)**if** $n == 1$ **return** a

// DFT of 1 element is the element itself

 $\omega_n = e^{2\pi i/n}$ $\omega = 1$ $a^{\text{even}} = (a_0, a_2, \dots, a_{n-2})$ $a^{\text{odd}} = (a_1, a_3, \dots, a_{n-1})$ $y^{\text{even}} = \text{FFT}(a^{\text{even}}, n/2)$ $y^{\text{odd}} = \text{FFT}(a^{\text{odd}}, n/2)$ **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 $a^{\text{even}} = (a_0, a_2, \dots, a_{n-2})$ and $a^{\text{odd}} = (a_1, a_3, \dots, a_{n-1})$ 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 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 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 $y_0, y_1, \dots, y_{n/2-1}$, 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 $y_{n/2}, y_{n/2+1}, \dots, y_{n-1}$, letting $k = 0, 1, \dots, n/2 - 1$, 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 $n/2$.

\Rightarrow 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 $j, k = 0, 1, \dots, n-1$.

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 $j, k = 0, 1, \dots, n-1$, 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

$$\sum_{j=0}^{n-1} \omega_n^{j(k'-k)}/n = \sum_{j=0}^{n-1} \omega_n^0/n$$

$$\begin{aligned}
&= \sum_{j=0}^{n-1} 1/n \\
&= n \cdot (1/n) \\
&= 1.
\end{aligned}$$

■

With the inverse matrix V_n^{-1} defined, $\text{DFT}_n^{-1}(y)$ is given by

$$\begin{aligned}
a_j &= \sum_{k=0}^{n-1} y_k \frac{\omega_n^{-jk}}{n} \\
&= \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}
\end{aligned}$$

for $j = 0, 1, \dots, n-1$.

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 $\Theta(\lg n)$.

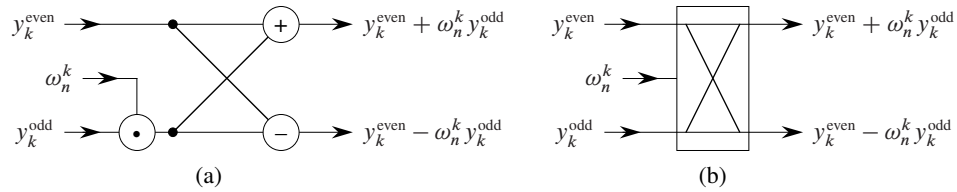
Butterfly operations

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

$$\begin{aligned}
 t &= \omega_n^k y_k^{\text{odd}} \\
 y_k &= y_k^{\text{even}} + t \\
 y_{k+(n/2)} &= y_k^{\text{even}} - t
 \end{aligned}$$

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.



Recursive circuit structure

Recall that the FFT procedure follows the divide-and-conquer strategy:

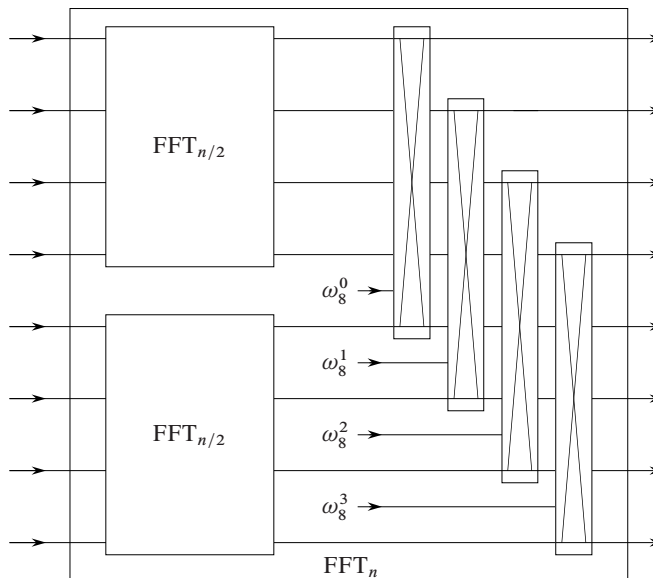
Divide the n -element input vector into its $n/2$ even-indexed and $n/2$ odd-indexed elements.

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 $\omega_n^0, \omega_n^1, \dots, \omega_n^{n/2-1}$.

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.

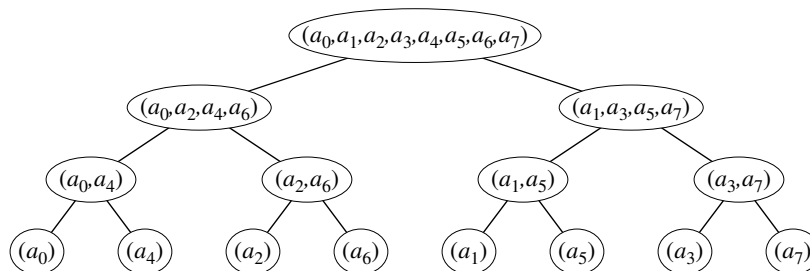


Permuting the inputs

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 $\text{rev}(k)$.

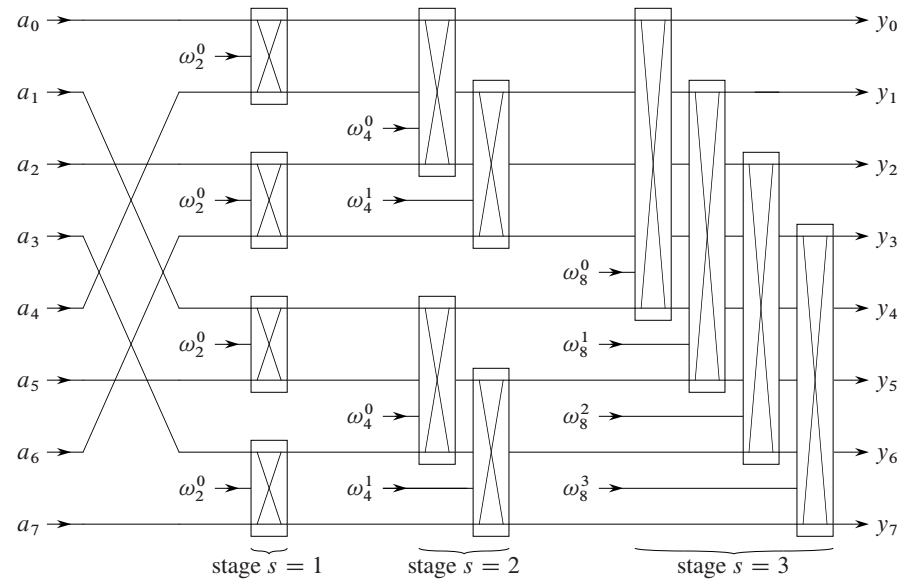
Example: In the above tree, the leaves appear in the order 0, 4, 2, 6, 1, 5, 3, 7. This sequence in binary is 000, 100, 010, 110, 001, 101, 011, 111.

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 $s = 1, 2, \dots, \lg n$, stage s consists of $n/2^s$ groups of butterflies, with 2^{s-1} butterflies per group.
- The twiddle factors in stage s are $\omega_m^0, \omega_m^1, \dots, \omega_m^{m/2-1}$, where $m = 2^s$.

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 $m \leq n$. The elements of P and T are drawn from a finite **alphabet** Σ . Examples: $\Sigma = \{0, 1\}$, Σ is the ASCII characters, or $\Sigma = \{A, C, G, T\}$ 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 $0 \leq s \leq n-m$ and $T[s+1:s+m] = P[1:m]$. (Need $s \leq n-m$ 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: $\Sigma = \{A, C, G, T\}$, $T = \text{GTAACAGTAAACG}$, $P = \text{AAC}$. [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 As and the pattern is m As. 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 $T[3 : 5] = \text{AAC}$. 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 $T[1 : m]$, $T[2 : m + 1]$, $T[3 : m + 2]$, \dots , $T[n - m + 1 : n]$. 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 representation. *[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

$r = 10 \cdot r + S[i]$

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 $\{0, 1, \dots, d - 1\}$ 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

$r = (d \cdot r + S[i]) \bmod q$

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)

```

     $h = d^{m-1} \bmod q$ 
     $p = 0$ 
     $t_0 = 0$ 
     $p = \text{COMPUTE-REP-MOD-Q}(P, m, d, q)$  // preprocessing
     $t_0 = \text{COMPUTE-REP-MOD-Q}(T, m, d, q)$  // 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  $s < n - m$ 
         $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$ 
```

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 $\{0, 1, \dots, q - 1\}$. 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.

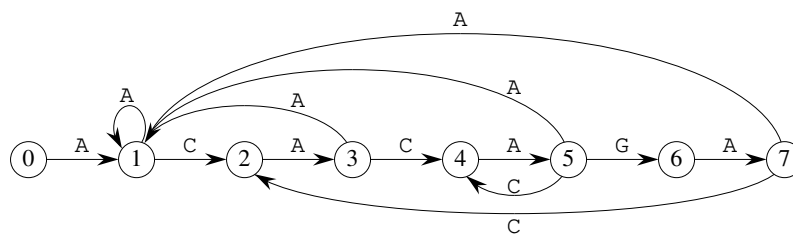
Formally: A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

- 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 $Q \times \Sigma \rightarrow Q$, 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 $\Sigma = \{A, C, G, T\}$ 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, δ, n, m)

$q = 0$

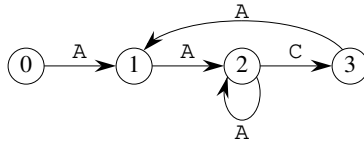
for $i = 1$ **to** n

$q = \delta(q, T[i])$

if $q == m$

 print "Pattern occurs with shift" $i - m$

Example: $P = \text{AAC}$, $T = \text{GTAACAGTAAACG}$. 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 $\delta(5, C) = 4$? 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 $\delta(5, C) = 4$?

- Take the prefix $P[:5] = \text{ACACA}$.
- 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 $\delta(5, c) = 4$.

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 $P[:k]$, $P[:k-1]$, $P[:k-2]$, \dots , $P[:1]$, then set $\delta(k, a) = 0$.

COMPUTE-TRANSITION-FUNCTION(P, Σ, 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 i th suffix in the sorted order.
- Then, denoting the suffix array for T by $SA[1:n]$, have $SA[i] = j$.

Example: Using the text $T = \text{ratatat}$ [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 i th 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 $T[SA[3] : SA[3] + LCP[3] - 1] = T[2 : 5] = \text{atat}$.

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 $s_1 < \text{rank of } s_2$.

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-rank[1 :  $n$ ], rank[1 :  $n$ ], and SA[1 :  $n$ ]
  for  $i = 1$  to  $n$ 
    substr-rank[ $i$ ].left-rank = ord( $T[i]$ )
    if  $i < n$ 
      substr-rank[ $i$ ].right-rank = ord( $T[i + 1]$ )
    else substr-rank[ $i$ ].right-rank = 0
    substr-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-rank, rank,  $n$ )
    for  $i = 1$  to  $n$ 
      substr-rank[ $i$ ].left-rank = rank[ $i$ ]
      if  $i + l \leq n$ 
        substr-rank[ $i$ ].right-rank = rank[ $i + l$ ]
      else substr-rank[ $i$ ].right-rank = 0
      substr-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-rank[ $i$ ].index
  return SA

MAKE-RANKS(substr-rank, rank,  $n$ )
   $r = 1$ 
  rank[substr-rank[1].index] =  $r$ 
  for  $i = 2$  to  $n$ 
    if substr-rank[ $i$ ].left-rank  $\neq$  substr-rank[ $i - 1$ ].left-rank
      or substr-rank[ $i$ ].right-rank  $\neq$  substr-rank[ $i - 1$ ].right-rank
       $r = r + 1$ 
    rank[substr-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	<i>left-rank</i>	<i>right-rank</i>	<i>index</i>	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>i</i>	<i>left-rank</i>	<i>right-rank</i>	<i>index</i>	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>i</i>	<i>rank</i>	<i>i</i>	<i>left-rank</i>	<i>right-rank</i>	<i>index</i>	substring	<i>i</i>	<i>left-rank</i>	<i>right-rank</i>	<i>index</i>	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 $\text{rank}[2] = \text{rank}[4] = \text{rank}[6] = 1$, $\text{rank}[1] = 2$, $\text{rank}[7] = 3$, and $\text{rank}[3] = \text{rank}[5] = 4$.

To understand what the **for** loop does:

- Concatenating adjacent substrings of length at most 2, it creates substrings of length at most 4.
- $\text{substr-rank}[i] = \text{rank}[i]$, the rank of the left substring.
- $\text{substr-rank}[i] = \text{rank}[i + 2]$, 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		After the for loop					After sorting				
i	rank	i	left-rank	right-rank	index	substring	i	left-rank	right-rank	index	substring
1	3	1	3	5	1	ratatat	1	1	0	6	at
2	2	2	2	1	2	atatat	2	2	0	4	atat
3	6	3	6	4	3	tatat	3	2	1	2	atatat
4	2	4	2	0	4	atat	4	3	5	1	ratatat
5	5	5	5	0	5	tat	5	4	0	7	t
6	1	6	1	0	6	at	6	5	0	5	tat
7	4	7	4	0	7	t	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 $\text{substr-rank}[i].\text{index}$ gives $\text{SA}[i]$.

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 i th 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 *tatat*, 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 *tatat* 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  $m + l \leq n$  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 $rank[1], rank[2], \dots, rank[n]$:

- 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 $\geq l$:
 - 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 $m + l \leq n$ ensures that $l \leq n$.
- Initial value of l is 0, and l decreases $\leq n - 1$ times in the last line of the **for** loop $\Rightarrow l$ is incremented $< 2n$ times.
- Total time is $\Theta(n)$.

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 $\rho(n)$ *approximation ratio* if

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n) .$$

We call the algorithm a $\rho(n)$ -*approximation algorithm*.

- Taking both ratios gives us $\rho(n) \geq 1$ 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 $E' \neq \emptyset$

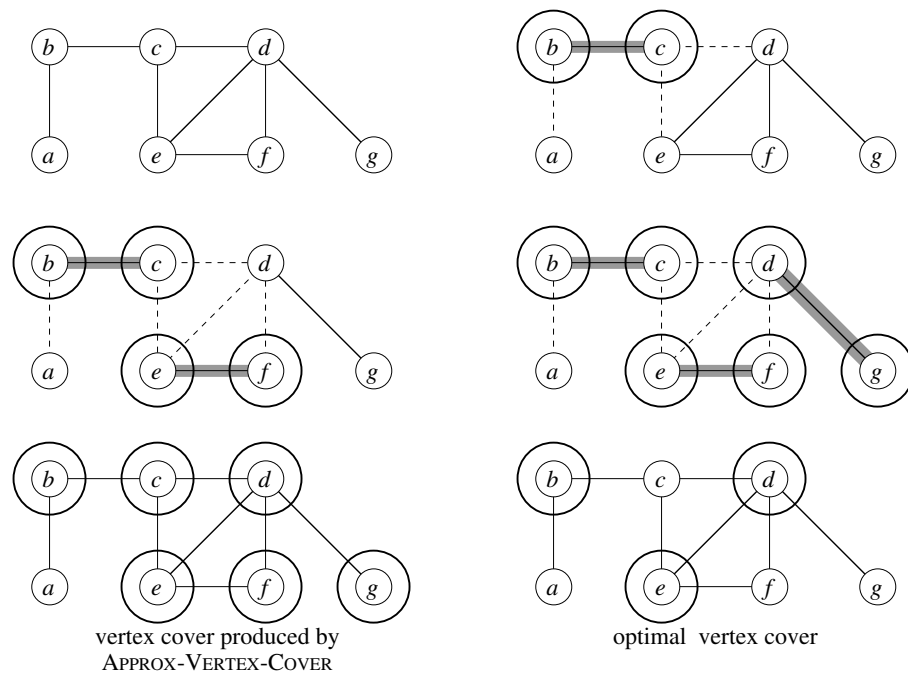
 let (u, v) be an arbitrary edge of E'

$C = C \cup \{u, v\}$

 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' .



Running time: Using adjacency lists to represent E' , $O(V + E)$.

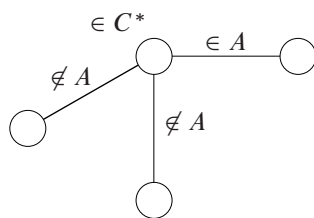
Theorem

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 $|C| \leq 2|C^*|$. 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 $|C^*| \geq |A|$.

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,

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*|. \end{aligned}$$

■

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 ($|C^*| \geq |A|$) 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:

$$c(u, w) \leq c(u, v) + c(v, w)$$

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.

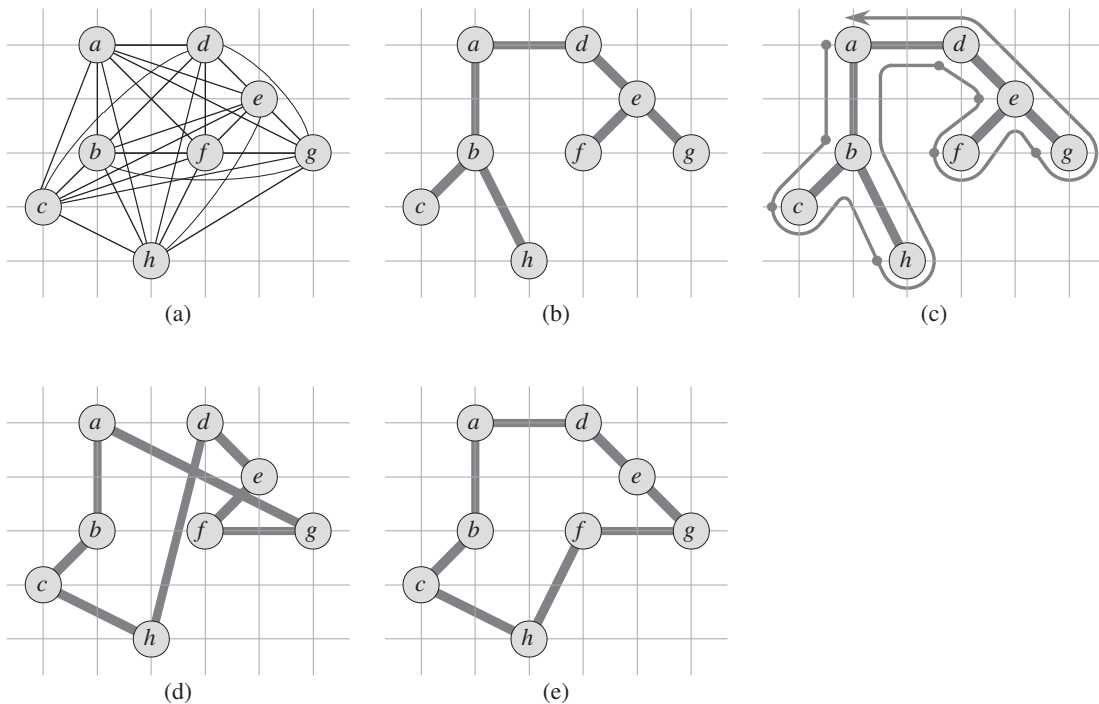
Idea: Construct a minimum spanning tree, do a preorder walk of the tree, and construct the TSP tour in the order in which each vertex is first visited during the preorder walk.

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

$$c(T) \leq c(T^*) \leq c(H^*) .$$

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 multiple times—twice in this case—in the summation.] So now we have

$$c(W) = 2c(T) \leq 2c(H^*) .$$

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 $c(u, w) \leq c(u, v) + c(v, w)$. What we get is the tour H , where $c(H) \leq c(W)$. And now

$$\begin{aligned} c(H) &\leq c(W) \\ &\leq 2c(H^*) . \end{aligned}$$

■

[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 inequality, 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 i th 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 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 $|S \cap U_i|$

$U_{i+1} = U_i - S$

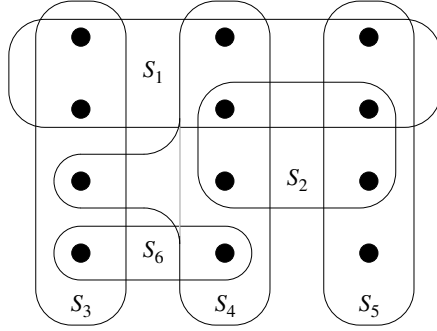
$\mathcal{C} = \mathcal{C} \cup \{S\}$

$i = i + 1$

return \mathcal{C}

The number of iterations of the **while** loop is bounded above by $\min\{|X|, |\mathcal{F}|\} = O(|X| + |\mathcal{F}|)$, 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

$$|U_{i+1}| \leq |U_i| - |U_i|/k$$

$$= |U_i|(1 - 1/k).$$
- Iterating this inequality gives

$$|U_0| = X,$$

$$|U_1| \leq |U_0|(1 - 1/k),$$

$$|U_2| \leq |U_1|(1 - 1/k) \leq |U_0|(1 - 1/k)^2.$$
 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), $1 + x \leq e^x$ for all real x .
Let $x = -1/k \Rightarrow 1 - 1/k \leq e^{-1/k} \Rightarrow (1 - 1/k)^k \leq (e^{-1/k})^k = e^{-1}$.
- Denote number i of iterations by ck for some $c \geq 0$.
Want c such that $|X| (1 - 1/k)^{ck} \leq |X| e^{-c} < 1$.
- 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, $|\mathcal{C}| \leq i = ck = c |\mathcal{C}^*| = |\mathcal{C}^*| (\lceil \ln |X| \rceil + 1)$.
 $\Rightarrow |\mathcal{C}| = |\mathcal{C}^*| O(\lg X)$. ■

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: $\max \{C/C^*, C^*/C\} \leq \rho(n)$.
- 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, \dots, x_n and m clauses.

Output: An assignment of 0 or 1 to the n variables x_1, x_2, \dots, 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, \dots, 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 $i = 1, 2, \dots, m$, define the indicator random variable

$$Y_i = \mathbf{I}\{\text{clause } i \text{ is satisfied}\},$$

so that $Y_i = 1$ as long as at least one of the literals in the i th 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, $E[Y_i] = 7/8$.

Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \dots + Y_m$. Then,

$$\begin{aligned} E[Y] &= E\left[\sum_{i=1}^m Y_i\right] \\ &= \sum_{i=1}^m E[Y_i] \quad (\text{by linearity of expectation}) \\ &= \sum_{i=1}^m 7/8 \\ &= 7m/8. \end{aligned}$$

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:

$$\begin{aligned} &\text{minimize} && \sum_{v \in V} w(v) x(v) \\ &\text{subject to} && \\ &&& x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\ &&& x(v) \in \{0, 1\} \text{ for each } v \in V. \end{aligned}$$

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**:

$$\begin{aligned} &\text{minimize} && \sum_{v \in V} w(v) x(v) \\ &\text{subject to} && \\ &&& x(u) + x(v) \geq 1 \text{ for each } (u, v) \in E \\ &&& x(v) \leq 1 \text{ for each } v \in V \\ &&& x(v) \geq 0 \text{ for each } v \in V. \end{aligned}$$

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)

$C = \emptyset$

compute \bar{x} , an optimal solution to the linear programming relaxation

for each vertex $v \in V$

if $\bar{x}(v) \geq 1/2$

$C = C \cup \{v\}$

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, $z^* \leq w(C^*)$.

Claim: Rounding the fractional values of the variables $\bar{x}(v)$ produces a set C that is a vertex cover and satisfies $w(C) \leq 2z^*$.

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, $x(u) + x(v) \geq 1$, 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

$$\begin{aligned}
 z^* &= \sum_{v \in V} w(v) \bar{x}(v) \\
 &\geq \sum_{v \in V: \bar{x}(v) \geq 1/2} w(v) \bar{x}(v) \\
 &\geq \sum_{v \in V: \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\
 &= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
 &= \frac{1}{2} \sum_{v \in C} w(v) \\
 &= \frac{1}{2} w(C) .
 \end{aligned}$$

Combining inequalities $z^* \leq w(C^*)$ and $z^* \geq (1/2)w(C)$ gives

$$w(C) \leq 2z^* \leq 2w(C^*) ,$$

and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. ■

The subset-sum problem

Input: Pair (S, t) , where S is a set $\{x_1, x_2, \dots, x_n\}$ of positive integers and t is a positive integer.

Output: A subset of $\{x_1, x_2, \dots, x_n\}$ 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 $\{x_1, x_2, \dots, x_i\}$, using as a starting point the sums of all subsets of $\{x_1, x_2, \dots, x_{i-1}\}$.
- 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 $\{x_1, \dots, x_i\}$ 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

$L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$

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 $y/(1 + \delta) \leq z \leq y$.
- 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 $L = \langle y_1, y_2, \dots, y_m \rangle$ 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, δ)

let m be the length of L

$L' = \langle y_1 \rangle$

$last = y_1$

for $i = 2$ **to** m

if $y_i > last \cdot (1 + \delta)$ // $y_i \geq last$ 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 $S = \{x_1, x_2, \dots, x_n\}$ 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 $\{x_1, x_2, \dots, x_i\}$.

- 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, ϵ)

$L_0 = \langle 0 \rangle$

for $i = 1$ **to** n

$L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$

$L_i = \text{TRIM}(L_i, \epsilon/2n)$

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 $z^* \leq t$ by removing from L_i every element that is greater than t , must have $z^* \leq y^*$.

Need to show that $y^*/z^* \leq 1 + \epsilon$.

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

$$\frac{y}{(1 + \epsilon/2n)^i} \leq z \leq y.$$

- The above inequality must hold for $y^* \in P_n$.
 \Rightarrow there exists an element $z \in L_n$ such that

$$\frac{y^*}{(1 + \epsilon/2n)^n} \leq z \leq y^*,$$

and thus

$$\frac{y^*}{z} \leq \left(1 + \frac{\epsilon}{2n}\right)^n.$$

- 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,

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\epsilon}{2n}\right)^n.$$

- Now, show that $y^*/z^* \leq 1 + \epsilon$ by showing that $(1 + \epsilon/2n)^n \leq 1 + \epsilon$.

- $0 < \epsilon < 1 \Rightarrow (\epsilon/2)^2 \leq \epsilon/2$.

- By equation (3.16), $\lim_{n \rightarrow \infty} (1 + \epsilon/2n)^n = e^{\epsilon/2}$.

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

$$\left(1 + \frac{\epsilon}{2n}\right)^n \leq e^{\epsilon/2}$$

$$\leq 1 + \epsilon/2 + (\epsilon/2)^2$$

$$\text{(inequality (3.15), } e^x \leq 1 + x + x^2 \text{ when } |x| < 1)$$

$$\leq 1 + \epsilon.$$

- Combining the above inequality and $y^*/z^* \leq (1 + \epsilon/2n)^n$ 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

$$\begin{aligned} & \log_{1+\epsilon/2n} t + 2 \\ &= \frac{\ln t}{\ln(1 + \epsilon/2n)} + 2 \\ &\leq \frac{2n(1 + \epsilon/2n) \ln t}{\epsilon} + 2 \\ &\quad \text{(by inequality (3.23), } x/(1+x) \leq \ln(1+x) \text{ with } x = \epsilon/2n) \\ &< \frac{3n \ln t}{\epsilon} + 2. \end{aligned}$$

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. ■