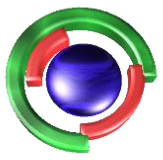
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs



Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Introduction to The Design & Analysis of Algorithms

2ND EDITION

Anany Levitin

Villanova University

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

https://hemanthrajhemu.github.io

2.6 Empirical Analysis of Algorithms 84
Exercises 2.6 90

2.7 Algorithm Visualization 91
Summary 95

3 Brute Force 97

3.1 Selection Sort and Bubble Sort 98
Selection Sort 98
Bubble Sort 100
Exercises 3.1 102

3.2 Sequential Search and Brute-Force String Matching 103
Sequential Search 103
Brute-Force String Matching 104
Exercises 3.2 105

3.3 Closest-Pair and Convex-Hull Problems by Brute Force 107
Closest-Pair Problem 107
Convex-Hull Problem 108
Exercises 3.3 112

3.4 Exhaustive Search 114
Traveling Salesman Problem 114
Knapsack Problem 115
Assignment Problem 116
Exercises 3.4 119
Summary 120

4 Divide-and-Conquer 123

4.1 Mergesort 125
Exercises 4.1 128

4.2 Quicksort 129
Exercises 4.2 134

4.3 Binary Search 135
Exercises 4.3 138

https://hemanthrajhemu.github.io

4.4 Binary Tree Traversals and Related Properties 139
Exercises 4.4 142

4.5 Multiplication of Large Integers and Strassen's Matrix Multiplication 144
Multiplication of Large Integers 144
Strassen's Matrix Multiplication 146
Exercises 4.5 148

4.6 Closest-Pair and Convex-Hull Problems by Divide-and-Conquer 149
Closest-Pair Problem 149
Convex-Hull Problem 150
Exercises 4.6 154
Summary 155

5 Decrease-and-Conquer 157

5.1 Insertion Sort 160
Exercises 5.1 163

5.2 Depth-First Search and Breadth-First Search 164
Depth-First Search 165
Breadth-First Search 167
Exercises 5.2 170

5.3 Topological Sorting 172
Exercises 5.3 176

5.4 Algorithms for Generating Combinatorial Objects 177
Generating Permutations 178
Generating Subsets 180
Exercises 5.4 181

5.5 Decrease-by-a-Constant-Factor Algorithms 183
Fake-Coin Problem 183
Multiplication à la Russe 184
Josephus Problem 185
Exercises 5.5 187

5.6 Variable-Size-Decrease Algorithms 188
Computing a Median and the Selection Problem 188

https://hemanthrajhemu.github.io

Interpolation Search 190
Searching and Insertion in a Binary Search Tree 191
The Game of Nim 192
Exercises 5.6 **194**
Summary **195**

6 Transform-and-Conquer **197**

6.1 Presorting **198**
Exercises 6.1 **201**

6.2 Gaussian Elimination **203**
LU Decomposition and Other Applications 208
Computing a Matrix Inverse 209
Computing a Determinant 210
Exercises 6.2 **212**

6.3 Balanced Search Trees **214**
AVL Trees 214
2-3 Trees 218
Exercises 6.3 **222**

6.4 Heaps and Heapsort **223**
Notion of the Heap 223
Heapsort 227
Exercises 6.4 **229**

6.5 Horner's Rule and Binary Exponentiation **230**
Horner's Rule 231
Binary Exponentiation 233
Exercises 6.5 **236**

6.6 Problem Reduction **237**
Computing the Least Common Multiple 238
Counting Paths in a Graph 239
Reduction of Optimization Problems 240
Linear Programming 241
Reduction to Graph Problems 244
Exercises 6.6 **245**
Summary **247**

https://hemanthrajhemu.github.io

4

Divide-and-Conquer

Whatever man prays for, he prays for a miracle. Every prayer reduces itself to this—Great God, grant that twice two be not four.

—Ivan Turgenev (1818–1883), Russian novelist and short-story writer

Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

The divide-and-conquer technique is diagrammed in Figure 4.1, which depicts the case of dividing a problem into two smaller subproblems, by far the most widely occurring case (at least for divide-and-conquer algorithms designed to be executed on a single-processor computer).

As an example, let us consider the problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed (by applying the same method, i.e., recursively), we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

Is this an efficient way to compute the sum of n numbers? A moment of reflection (why could it be more efficient than the brute-force summation?), a small example of summing, say, four numbers by this algorithm, a formal analysis

https://hemanthrajhemu.github.io

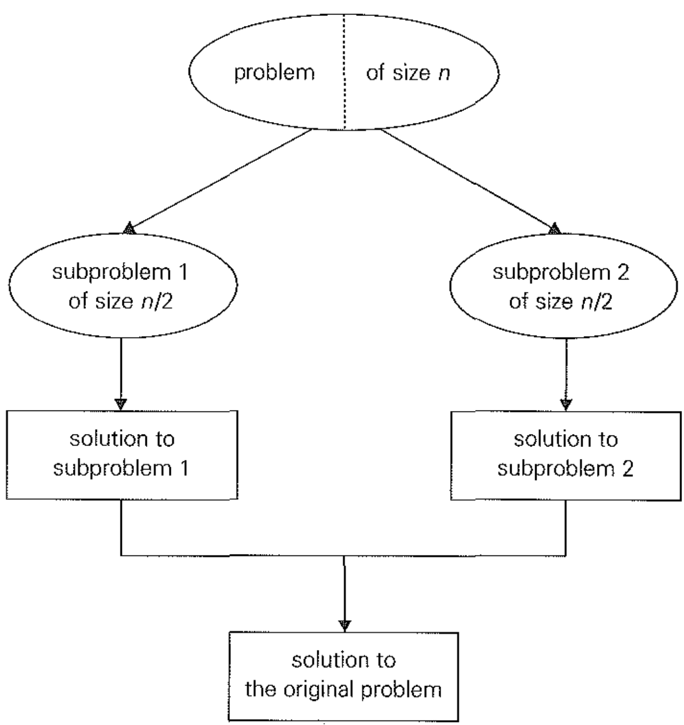


FIGURE 4.1 Divide-and-conquer technique (typical case)

(which follows), and common sense (we do not compute sums this way, do we?) all lead to a negative answer to this question.

Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution. But often our prayers to the Goddess of Algorithmics—see the chapter's epigraph—*are* answered, and the time spent on executing the divide-and-conquer plan turns out to be smaller than solving a problem by a different method. In fact, the divide-and-conquer approach yields some of the most important and efficient algorithms in computer science. We discuss a few classic examples of such algorithms in this chapter. Though we consider only sequential algorithms here, it is worth keeping in mind that the divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

As mentioned above, in the most typical case of divide-and-conquer, a problem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.). Assuming

https://hemanthrajhemu.github.io

that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n), \quad \textbf{(4.1)}$$

where $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. (For the summation example, $a = b = 2$ and $f(n) = 1$.) Recurrence (4.1) is called the ***general divide-and-conquer recurrence***. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (see Appendix B).

THEOREM *(Master Theorem)* If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation (4.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

(Analogous results hold for the O and Ω notations, too.)

For example, the recurrence equation for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Note that we were able to find the solution's efficiency class without going through the drudgery of solving the recurrence. But, of course, this approach can only establish a solution's order of growth to within an unknown multiplicative constant, while solving a recurrence equation with a specific initial condition yields an exact answer (at least for n's that are powers of b).

4.1 Mergesort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2\rfloor - 1]$ and $A[\lfloor n/2\rfloor..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

https://hemanthrajhemu.github.io

ALGORITHM *Mergesort*($A[0..n-1]$)

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)
```

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge*($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$)

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure 4.2.

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced

https://hemanthrajhemu.github.io

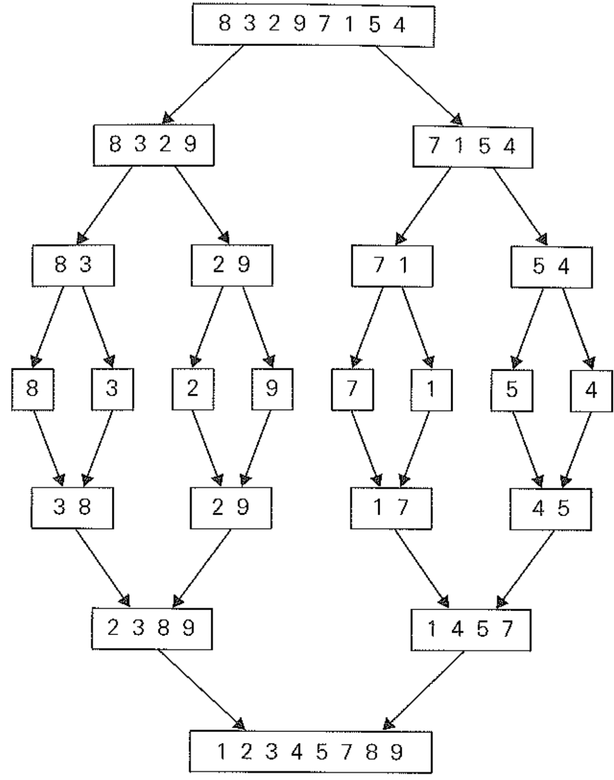


FIGURE 4.2 Example of mergesort operation

by one. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Hence, according to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$ (why?). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1.$$

The number of key comparisons made by mergesort in the worst case comes very close to the theoretical minimum[1] that any general comparison-based sorting algorithm can have. The principal shortcoming of mergesort is the linear amount

1. As we shall see in Section 11.2, this theoretical minimum is $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$.

https://hemanthrajhemu.github.io

of extra storage the algorithm requires. Though merging can be done in place, the resulting algorithm is quite complicated and, since it has a significantly larger multiplicative constant, the in-place mergesort is of theoretical interest only.

Exercises 4.1

1. **a.** Write a pseudocode for a divide-and-conquer algorithm for finding a position of the largest element in an array of n numbers.
   b. What will be your algorithm's output for arrays with several elements of the largest value?
   c. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.
   d. How does this algorithm compare with the brute-force algorithm for this problem?
2. **a.** Write a pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.
   b. Set up and solve (for $n = 2^k$) a recurrence relation for the number of key comparisons made by your algorithm.
   c. How does this algorithm compare with the brute-force algorithm for this problem?
3. **a.** Write a pseudocode for a divide-and-conquer algorithm for the exponentiation problem of computing a^n where $a > 0$ and n is a positive integer.
   b. Set up and solve a recurrence relation for the number of multiplications made by this algorithm.
   c. How does this algorithm compare with the brute-force algorithm for this problem?
4. We mentioned in Chapter 2, that logarithm bases are irrelevant in most contexts arising in the analysis of an algorithm's efficiency class. Is this true for both assertions of the Master Theorem that include logarithms?
5. Find the order of growth for solutions of the following recurrences.
   a. $T(n) = 4T(n/2) + n, \ T(1) = 1$
   b. $T(n) = 4T(n/2) + n^2, \ T(1) = 1$
   c. $T(n) = 4T(n/2) + n^3, \ T(1) = 1$
6. Apply mergesort to sort the list E, X, A, M, P, L, E in alphabetical order.
7. Is mergesort a stable sorting algorithm?
8. **a.** Solve the recurrence relation for the number of key comparisons made by mergesort in the worst case. (You may assume that $n = 2^k$.)

https://hemanthrajhemu.github.io

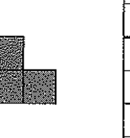
b. Set up a recurrence relation for the number of key comparisons made by mergesort on best-case inputs and solve it for $n = 2^k$.

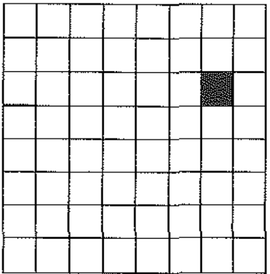
c. Set up a recurrence relation for the number of key moves made by the version of mergesort given in Section 4.1. Does taking the number of key moves into account change the algorithm's efficiency class?

9. Let $A[0..n-1]$ be an array of n distinct real numbers. A pair $(A[i], A[j])$ is said to be an ***inversion*** if these numbers are out of order, i.e., $i < j$ but $A[i] > A[j]$. Design an $O(n \log n)$ algorithm for counting the number of inversions.

10. One can implement mergesort without a recursion by starting with merging adjacent elements of a given array, then merging sorted pairs, and so on. Implement this bottom-up version of mergesort in the language of your choice.

11. *Tromino puzzle* A tromino is an L-shaped tile formed by 1-by-1 adjacent squares. The problem is to cover any 2^n-by-2^n chessboard with one missing square (anywhere on the board) with trominos. Trominos should cover all the squares except the missing one with no overlaps.





Design a divide-and-conquer algorithm for this problem.

4.2 Quicksort

Quicksort is another important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. Specifically, it rearranges elements of a given array $A[0..n-1]$ to achieve its ***partition***, a situation where all the elements before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$$

https://hemanthrajhemu.github.io

Obviously, after a partition has been achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays of the elements preceding and following $A[s]$ independently (e.g., by the same method).

```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right indices
    //       l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ←Partition(A[l..r]) //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

A partition of $A[0..n-1]$ and, more generally, of its subarray $A[l..r]$ $(0 \leq l < r \leq n-1)$ can be achieved by the following algorithm. First, we select an element with respect to whose value we are going to divide the subarray. Because of its guiding role, we call this element the ***pivot***. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of the subarray: one is left-to-right and the other right-to-left, each comparing the subarray's elements with the pivot. The left-to-right scan, denoted below by index i, starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:

		$\rightarrow i$		$j \leftarrow$	
p	all are $\leq p$	$\geq p$	. . .	$\leq p$	all are $\geq p$

https://hemanthrajhemu.github.io

If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the array after exchanging the pivot with $A[j]$:

p	all are $\leq p$	$\leq p$	$\geq p$	all are $\geq p$

($j \leftarrow\!\!\rightarrow i$; the pivot p is exchanged with the element $\leq p$ at position j)

Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the array partitioned, with the split position $s = i = j$:

p	all are $\leq p$	$= p$	all are $\geq p$

($\rightarrow i = j \leftarrow$)

We can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Here is a pseudocode implementing this partitioning procedure.

ALGORITHM *Partition*$(A[l..r])$

```
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//          this function's value
p ← A[l]
i ← l;   j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Note that index i can go out of the subarray bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array $A[0..n-1]$ a "sentinel" that would prevent index i from advancing beyond position n. The more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Figure 4.3.

https://hemanthrajhemu.github.io

0	1	2	3	4	5	6	7
5	*i* 3	1	9	8	2	4	*j* 7
5	3	1	*i* 9	8	2	*j* 4	7
5	3	1	*i* 4	8	2	*j* 9	7
5	3	1	4	*i* 8	*j* 2	9	7
5	3	1	4	*i* 2	*j* 8	9	7
5	3	1	4	*j* 2	*i* 8	9	7
2	3	1	4	**5**	8	9	7
2	*i* 3	1	*j* 4				
2	*i* 3	*j* 1	4				
2	*i* 1	*j* 3	4				
2	*j* 1	*i* 3	4				
1	**2**	3	4				
1							
		3	*ij* 4				
		j **3**	*i* 4				
			4				
					8	*i* 9	*j* 7
					8	*i* 7	*j* 9
					8	*j* 7	*i* 9
					7	**8**	9
					7		
							9

(a)

(b)

FIGURE 4.3 Example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to *Quicksort* with input values l and r of subarray bounds and split position s of a partition obtained.

We start our discussion of quicksort's efficiency by noting that the number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over, n if they coincide (why?). If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case will satisfy the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

https://hemanthrajhemu.github.io

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, while the size of the other will be just one less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved! Indeed, if $A[0..n-1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0:

$j \leftarrow$	$\rightarrow i$		
$A[0]$	$A[1]$	. . .	$A[n-1]$

So, after making $n+1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will find itself with the strictly increasing array $A[1..n-1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n-2..n-1]$ has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Thus, the question about the utility of quicksort comes to its average-case behavior. Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. Assuming that the partition split can happen in each position s $(0 \le s \le n-1)$ with the same probability $1/n$, we get the following recurrence relation

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Though solving this recurrence is easier than one might expect, it is still much trickier than the worst- and best-case analyses, and we will leave it for the exercises. Its solution turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n.$$

Thus, on the average, quicksort makes only 38% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it runs faster than mergesort (and heapsort, another $n \log n$ algorithm that we discuss in Chapter 6) on randomly ordered arrays, justifying the name given to the algorithm by its inventor, the prominent British computer scientist C.A.R. Hoare.[2]

2. The young Hoare invented his algorithm while trying to sort words of a Russian dictionary for a machine translation project from Russian to English. Says Hoare, "My first thought on how to do

https://hemanthrajhemu.github.io

Given the importance of quicksort, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are: better pivot selection methods (such as the ***median-of-three partitioning*** that uses as a pivot the median of the leftmost, rightmost, and the middle element of the array); switching to a simpler sort on smaller subfiles; and recursion elimination (so-called nonrecursive quicksort). According to R. Sedgewick [Sed98], the world's leading expert on quicksort, these improvements in combination can cut the running time of the algorithm by 20%–25%.

We should also point out that the idea of partitioning can be useful in applications other than sorting. In particular, it underlines a fast algorithm for the important ***selection problem*** discussed in Section 5.6.

Exercises 4.2

1. Apply quicksort to sort the list

$$E,\ X,\ A,\ M,\ P,\ L,\ E$$

in alphabetical order. Draw the tree of the recursive calls made.

2. For the partitioning procedure outlined in Section 4.2:

a. Prove that if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p.

b. Prove that when the scanning indices stop, j cannot point to an element more than one position to the left of the one pointed to by i.

c. Why is it worth stopping the scans after encountering an element equal to the pivot?

3. Is quicksort a stable sorting algorithm?

4. Give an example of an array of n elements for which the sentinel mentioned in the text is actually needed. What should be its value? Also explain why a single sentinel suffices for any input.

5. For the version of quicksort given in the text:

a. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?

b. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

6. a. For quicksort with the median-of-three pivot selection, are increasing arrays the worst-case input, the best-case input, or neither?

---

this was bubblesort and, by an amazing stroke of luck, my second thought was Quicksort." It is hard to disagree with his overall assessment: "I have been very lucky. What a wonderful way to start a career in Computing, by discovering a new sorting algorithm!" [Hoa96]

https://hemanthrajhemu.github.io

b. Answer the same question for decreasing arrays.

7. Solve the average-case recurrence for quicksort.

8. Design an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements. Your algorithm should be both time- and space-efficient.

9. The ***Dutch flag problem*** is to rearrange any array of characters R, W, and B (red, white, and blue are the colors of the Dutch national flag) so that all the R's come first, the W's come next, and the B's come last. Design a linear in-place algorithm for this problem.

10. Implement quicksort in the language of your choice. Run your program on a sample of inputs to verify the theoretical assertions about the algorithm's efficiency.

11. *Nuts and bolts* You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average-case efficiency in $\Theta(n \log n)$. [Raw91], p. 293

4.3 Binary Search

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the array's middle element $A[m]$. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$:

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}. \end{array}$$

As an example, let us apply binary search to searching for $K = 70$ in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

https://hemanthrajhemu.github.io

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is a pseudocode for this nonrecursive version.

```
ALGORITHM  BinarySearch(A[0..n − 1], K)
    //Implements nonrecursive binary search
    //Input: An array A[0..n − 1] sorted in ascending order and
    //       a search key K
    //Output: An index of the array's element that is equal to K
    //        or −1 if there is no such element
    l ← 0; r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] return m
        else if K < A[m] r ← m − 1
        else l ← m + 1
    return −1
```

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array. Moreover, for the sake of simplicity, we will count the so-called three-way comparisons. This assumes that after one comparison of K with $A[m]$, the algorithm can determine whether K is smaller, equal to, or larger than $A[m]$.

How many such comparisons does the algorithm make on an array of n elements? The answer obviously depends not only on n but also on the specifics of a particular instance of the problem. Let us find the number of key comparisons in the worst case $C_{worst}(n)$. The worst-case inputs include all arrays that do not contain a given search key (and, in fact, some cases of successful searches as well). Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1. \qquad \textbf{(4.2)}$$

https://hemanthrajhemu.github.io

(Stop and convince yourself that $n/2$ must be, indeed, rounded down and that the initial condition must be written as specified.)

As we discussed in Section 2.4, the standard way of solving recurrences such as recurrence (4.2) is to assume that $n = 2^k$ and solve the resulting recurrence by backward substitutions or another method. We leave this as a straightforward exercise to obtain the solution

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1. \tag{4.3}$$

Actually, one can prove that the solution given by formula (4.3) for $n = 2^k$ can be tweaked to get a solution valid for an arbitrary positive integer n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil. \tag{4.4}$$

Let us verify by substitution that $C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$ indeed satisfies equation (4.2) for any positive even number n. (You are asked to do this for odd n's in Exercises 4.3). If n is positive and even, $n = 2i$ where $i > 0$. The left-hand side of equation (4.2) for $n = 2i$ is

$$\begin{aligned} C_{worst}(n) &= \lfloor \log_2 n \rfloor + 1 = \lfloor \log_2 2i \rfloor + 1 = \lfloor \log_2 2 + \log_2 i \rfloor + 1 \\ &= (1 + \lfloor \log_2 i \rfloor) + 1 = \lfloor \log_2 i \rfloor + 2. \end{aligned}$$

The right-hand side of equation (4.2) for $n = 2i$ is

$$\begin{aligned} C_{worst}(\lfloor n/2 \rfloor) + 1 &= C_{worst}(\lfloor 2i/2 \rfloor) + 1 = C_{worst}(i) + 1 \\ &= (\lfloor \log_2 i \rfloor + 1) + 1 = \lfloor \log_2 i \rfloor + 2. \end{aligned}$$

Since both expressions are the same, we proved the assertion.

Formula (4.4) deserves attention. First, it implies that the worst-case efficiency of binary search is in $\Theta(\log n)$. (Incidentally, we could get this fact by applying the Master Theorem, but this approach would not give us the value of the multiplicative constant.) Second, it is the answer we should have fully expected: since the algorithm simply reduces the size of the remaining array by about half on each iteration, the number of such iterations needed to reduce the initial size n to the final size 1 has to be about $\log_2 n$. Third, to reiterate the point made in Section 2.1, the logarithmic function grows so slowly that its values remain small even for very large values of n. In particular, according to formula (4.4), it will take no more than $\lfloor \log_2 10^3 \rfloor + 1 = 10$ three-way comparisons to find an element of a given value (or establish that there is no such element) in any sorted array of 1000 elements, and it will take no more than $\lfloor \log_2 10^6 \rfloor + 1 = 20$ comparisons to do this for any sorted array of size one million!

What can we say about the average-case efficiency of binary search? A sophisticated analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{avg}(n) \approx \log_2 n.$$

https://hemanthrajhemu.github.io

(More accurate formulas for the average number of comparisons in a successful and an unsuccessful search are $C_{avg}^{yes}(n) \approx \log_2 n - 1$ and $C_{avg}^{no}(n) \approx \log_2(n+1)$, respectively.)

Though binary search is an optimal searching algorithm if we restrict our operations only to comparisons between keys (see Section 10.2), there are searching algorithms (see interpolation search in Section 5.6 and hashing in Section 7.3) with a better average-case efficiency, and one of them (hashing) does not even require the array to be sorted! These algorithms do require some special calculations in addition to key comparisons, however. Finally, the idea behind binary search has several applications beyond searching (see, e.g., [Ben00]). In addition, it can be applied to solving nonlinear equations in one unknown; we discuss this continuous analogue of binary search, called the method of bisection, in Section 12.4.

Before we leave this section, one other remark about binary search needs to be made. Binary search is sometimes presented as the quintessential example of a divide-and-conquer algorithm. This interpretation is flawed because, in fact, binary search is a very atypical case of divide-and-conquer. Indeed, according to the definition given at the beginning of this chapter, the divide-and-conquer technique divides a problem into *several* subproblems, each of which needs to be solved. That is not the case for binary search, where only one of the two subproblems needs to be solved. Therefore, if binary search is to be considered as a divide-and-conquer algorithm, it should be looked on as a degenerate case of this technique. As a matter of fact, binary search fits much better into the class of decrease-by-half algorithms, which we discuss in Section 5.5. Why then is this discussion of binary search in this chapter? Partly because of tradition and partly because a bad example can sometimes make a point that a good example cannot.

Exercises 4.3

1. a. What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98

b. List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.

c. Find the average number of key comparisons made by binary search in a successful search in this array. (Assume that each key is searched for with the same probability.)

d. Find the average number of key comparisons made by binary search in an unsuccessful search in this array. (Assume that searches for keys in each of the 14 intervals formed by the array's elements are equally likely.)

https://hemanthrajhemu.github.io

2. Solve the recurrence $C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1$ for $n > 1$, $C_{worst}(1) = 1$, for $n = 2^k$ by backward substitutions.

3. **a.** Prove the equality

$$\lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil \quad \text{for } n \geq 1.$$

   b. Prove that $C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$ satisfies equation (4.2) for every positive odd integer n.

4. Estimate how many times faster an average successful search will be in a sorted array of 100,000 elements if it is done by binary search versus sequential search.

5. Sequential search can be used with about the same efficiency whether a list is implemented as an array or as a linked list. Is it also true for binary search? (Of course, we assume that a list is sorted for binary search.)

6. How can one use binary search for range searching, i.e., for finding all the elements in a sorted array whose values fall between two given values L and U (inclusively), $L \leq U$? What is the worst-case efficiency of this algorithm?

7. Write a pseudocode for a recursive version of binary search.

8. Design a version of binary search that uses only two-way comparisons such as $\leq$ and $=$. Implement your algorithm in the language of your choice and carefully debug it (such programs are notorious for being prone to bugs).

9. Analyze the time efficiency of the two-way comparison version designed in Problem 8.

10. *Picture guessing* A version of the popular problem-solving task involves presenting people with an array of 42 pictures—seven rows of six pictures each—and asking them to identify the target picture by asking questions that can be answered yes or no. Further, people are then required to identify the picture with as few questions as possible. Suggest the most efficient algorithm for this problem and indicate the largest number of questions that may be necessary.

4.4 Binary Tree Traversals and Related Properties

In this section, we see how the divide-and-conquer technique can be applied to binary trees. A ***binary tree*** T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root. We usually think of a binary tree as a special case of an ordered tree (Figure 4.4). (This standard interpretation was an alternative definition of a binary tree in Section 1.4.)

https://hemanthrajhemu.github.io

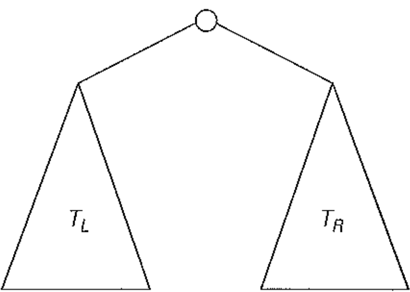


FIGURE 4.4 Standard representation of a binary tree

Since the definition itself divides a binary tree into two smaller structures of the same type, the left subtree and the right subtree, many problems about binary trees can be solved by applying the divide-conquer technique. As an example, let us consider a recursive algorithm for computing the height of a binary tree. Recall that the height is defined as the length of the longest path from the root to a leaf. Hence, it can be computed as the maximum of the heights of the root's left and right subtrees plus 1. (We add 1 to account for the extra level of the root.) Also note that it is convenient to define the height of the empty tree as -1. Thus, we have the following recursive algorithm.

ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_L), Height(T_R)\} + 1$

We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree T. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions $A(n(T))$ made by the algorithm are the same. We have the following recurrence relation for $A(n(T))$:

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1 \quad \text{for } n(T) > 0,$$
$$A(0) = 0.$$

Before we solve this recurrence (can you tell what its solution is?), let us note that addition is not the most frequently executed operation of this algorithm. What is? Checking—and this is very typical for binary tree algorithms—that the tree is not empty. For example, for the empty tree, the comparison $T = \varnothing$ is executed once but there are no additions, and for a single-node tree, the comparison and addition numbers are three and one, respectively.

https://hemanthrajhemu.github.io

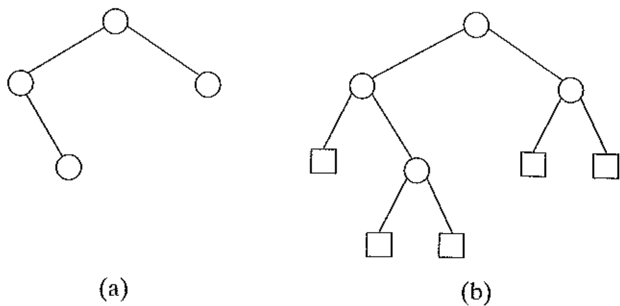


FIGURE 4.5 (a) Binary tree. (b) Its extension. Internal nodes are shown as circles; external nodes are shown as squares.

It helps in the analysis of tree algorithms to draw the tree's extension by replacing the empty subtrees by special nodes. The extra nodes (shown by little squares in Figure 4.5) are called ***external***; the original nodes (shown by little circles) are called ***internal***. By definition, the extension of the empty binary tree is a single external node.

It is easy to see that the height algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal and external node. Thus, to ascertain the algorithm's efficiency, we need to know how many external nodes an extended binary tree with n internal nodes can have. Checking Figure 4.5 and a few similar examples, it is easy to hypothesize that the number of external nodes x is always one more than the number of internal nodes n:

$$x = n + 1. \tag{4.5}$$

To prove this formula, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equation (4.5).

Note that equation (4.5) also applies to any nonempty ***full binary tree***, in which, by definition, every node has either zero or two children: for a full binary tree, n and x denote the numbers of parental nodes and leaves, respectively.

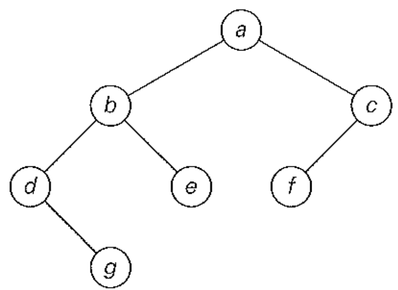
Returning to algorithm *Height*, the number of comparisons to check whether the tree is empty is

$$C(n) = n + x = 2n + 1,$$

while the number of additions is

$$A(n) = n.$$

https://hemanthrajhemu.github.io



Preorder: a, b, d, g, e, c, f
Inorder: d, g, b, e, a, f, c
Postorder: g, d, e, b, f, c, a

FIGURE 4.6 Binary tree and its traversals

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ just by the timing of the root's visit:

In the ***preorder traversal***, the root is visited before the left and right subtrees are visited (in that order).

In the ***inorder traversal***, the root is visited after visiting its left subtree but before visiting the right subtree.

In the ***postorder traversal***, the root is visited after visiting the left and right subtrees (in that order).

These traversals are illustrated in Figure 4.6. Their pseudocodes are quite straightforward, repeating the descriptions given above. (These traversals are also a standard feature of data structures textbooks.) As to their efficiency analysis, it is identical to the above analysis of the *Height* algorithm because a recursive call is made for each node of an extended binary tree.

Finally, we should note that, obviously, not all questions about binary trees require traversals of both left and right subtrees. For example, the find and insert operations for a binary search tree require processing only one of the two subtrees. Hence, they should be considered not as applications of divide-and-conquer but rather as examples of the variable-size decrease technique discussed in Section 5.6.

Exercises 4.4

1. Design a divide-and-conquer algorithm for computing the number of levels in a binary tree. (In particular, the algorithm must return 0 and 1 for the empty and single-node trees, respectively.) What is the efficiency class of your algorithm?

https://hemanthrajhemu.github.io

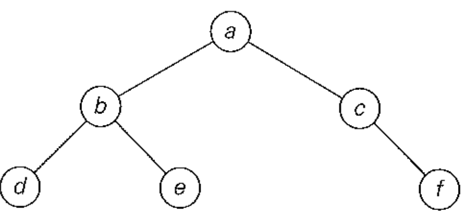
2. The following algorithm seeks to compute the number of leaves in a binary tree.

ALGORITHM *LeafCounter(T)*

//Computes recursively the number of leaves in a binary tree
//Input: A binary tree T
//Output: The number of leaves in T
if $T = \varnothing$ **return** 0
else return $LeafCounter(T_L) + LeafCounter(T_R)$

Is this algorithm correct? If it is, prove it; if it is not, make an appropriate correction.

3. Prove equality (4.5) by mathematical induction.
4. Traverse the following binary tree
   a. in preorder. **b.** in inorder. **c.** in postorder.



5. Write a pseudocode for one of the classic traversal algorithms (preorder, inorder, and postorder) for binary trees. Assuming that your algorithm is recursive, find the number of recursive calls made.
6. Which of the three classic traversal algorithms yields a sorted list if applied to a binary search tree? Prove this property.
7. **a.** Draw a binary tree with ten nodes labeled 0, 1, 2, . . . , 9 in such a way that the inorder and postorder traversals of the tree yield the following lists: 9, 3, 1, 0, 4, 2, 7, 6, 8, 5 (inorder) and 9, 1, 4, 0, 3, 6, 7, 5, 8, 2 (postorder).
   b. Give an example of two permutations of the same n labels $0, 1, 2, \dots, n-1$ that cannot be inorder and postorder traversal lists of the same binary tree.
   c. Design an algorithm that constructs a binary tree for which two given lists of n labels $0, 1, 2, \dots, n-1$ are generated by the inorder and postorder traversals of the tree. Your algorithm should also identify inputs for which the problem has no solution.
8. The ***internal path length*** I of an extended binary tree is defined as the sum of the lengths of the paths—taken over all internal nodes—from the root to

https://hemanthrajhemu.github.io

each internal node. Similarly, the ***external path length*** E of an extended binary tree is defined as the sum of the lengths of the paths—taken over all external nodes—from the root to each external node. Prove that $E = I + 2n$ where n is the number of internal nodes in the tree.

9. Write a program for computing the internal path length of an extended binary tree. Use it to investigate empirically the average number of key comparisons for searching in a randomly generated binary search tree.

10. *Chocolate bar puzzle* Given an n-by-m chocolate bar, you need to break it into nm 1-by-1 pieces. You can break a bar only in a straight line, and only one bar can be broken at a time. Design an algorithm that solves the problem with the minimum number of bar breaks. What is this minimum number? Justify your answer by using properties of a binary tree.

4.5 Multiplication of Large Integers and Strassen's Matrix Multiplication

In this section, we examine two surprising algorithms for seemingly straightforward tasks: multiplying two numbers and multiplying two square matrices. Both seek to decrease the total number of multiplications performed at the expense of a slight increase in the number of additions. Both do this by exploiting the divide-and-conquer idea.

Multiplication of Large Integers

Some applications, notably modern cryptology, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers. In this section, we outline an interesting algorithm for multiplying such numbers. Obviously, if we use the classic pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad a shorter number with leading zeros to equal their lengths.) Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, it turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \text{ and } 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

https://hemanthrajhemu.github.io

Now let us multiply them:

$$23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$
$$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0.$$

The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit integers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$, and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$
$$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$
$$= c_2 10^n + c_1 10^{n/2} + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

If $n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when

https://hemanthrajhemu.github.io

n becomes one. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ will be

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2}) \\ &= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

You should keep in mind that for moderately large integers, this algorithm will probably run longer than the classic one. Brassard and Bratley ([Bra96], pp. 70–71) report that in their experiments the divide-and-conquer algorithm started to outperform the pen-and-pencil method on integers over 600 digits long. If you program in an object-oriented language such as Java, C++, or Smalltalk, you should also be aware that these languages have special classes for dealing with large integers.

Strassen's Matrix Multiplication

Now that we have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers, we should not be surprised that a similar feat can be accomplished for multiplying matrices. Such an algorithm was published by V. Strassen in 1969 [Str69]. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2-by-2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm (see Example 3, Section 2.3). This is accomplished by using the following formulas:

$$\begin{aligned} \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} &= \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} \\ &= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}, \end{aligned}$$

https://hemanthrajhemu.github.io

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$
$$m_2 = (a_{10} + a_{11}) * b_{00}$$
$$m_3 = a_{00} * (b_{01} - b_{11})$$
$$m_4 = a_{11} * (b_{10} - b_{00})$$
$$m_5 = (a_{00} + a_{01}) * b_{11}$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Thus, to multiply two 2-by-2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. These numbers should not lead us to multiplying 2-by-2 matrices by Strassen's algorithm. Its importance stems from its *asymptotic* superiority as matrix order n goes to infinity.

Let A and B be two n-by-n matrices where n is a power of two. (If n is not a power of two, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four $n/2$-by-$n/2$ submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. If the seven products of $n/2$-by-$n/2$ matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Let us evaluate the asymptotic efficiency of this algorithm. If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two n-by-n matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \qquad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \cdots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

which is smaller than n^3 required by the brute-force algorithm.

Since this saving in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions $A(n)$ made by

https://hemanthrajhemu.github.io

Strassen's algorithm. To multiply two matrices of order $n > 1$, the algorithm needs to multiply seven matrices of order $n/2$ and make 18 additions of matrices of size $n/2$; when $n = 1$, no additions are made since two numbers are simply multiplied. These observations yield the following recurrence relation:

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0.$$

Though one can obtain a closed-form solution to this recurrence (see Problem 8), here we simply establish the solution's order of growth. According to the Master Theorem stated in the beginning of the chapter, $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. This puts Strassen's algorithm in $\Theta(n^{\log_2 7})$, which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.

Since the time of Strassen's discovery, several other algorithms for multiplying two n-by-n matrices of real numbers in $O(n^\alpha)$ time with progressively smaller constants α have been invented. The fastest algorithm so far is that of Coopersmith and Winograd [Coo87], with its efficiency in $O(n^{2.376})$. The decreasing values of the exponents have been obtained at the expense of increasing complexity of these algorithms. Because of large multiplicative constants, none of them is of practical value. However, they are interesting from a theoretical point of view. Although these algorithms get closer and closer to the best theoretical lower bound known for matrix multiplication, which is n^2 multiplications, the gap between this bound and the best available algorithm remains unresolved. It is also worth mentioning that matrix multiplication is known to be computationally equivalent to some other important problems such as solving systems of linear equations.

Exercises 4.5

1. What are the smallest and largest numbers of digits the product of two decimal n-digit integers can have?

2. Compute $2101 * 1130$ by applying the divide-and-conquer algorithm outlined in the text.

3. **a.** Prove the equality $a^{\log_b c} = c^{\log_b a}$, which was used twice in Section 4.5.

   b. Why is $n^{\log_2 3}$ better than $3^{\log_2 n}$ as a closed-form formula for $M(n)$?

4. **a.** Why did we not include multiplications by 10^n in the multiplication count $M(n)$ of the large-integer multiplication algorithm?

   b. In addition to assuming that n is a power of 2, we made, for the sake of simplicity, another, more subtle, assumption in setting up a recurrence relation for $M(n)$, which is not always true (it does not change the final answer, however). What is this assumption?

https://hemanthrajhemu.github.io

5. How many one-digit additions are made by the pen-and-pencil algorithm in multiplying two n-digit integers? (You may disregard potential carries.)
6. Verify the formulas underlying Strassen's algorithm for multiplying 2-by-2 matrices.
7. Apply Strassen's algorithm to compute

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

exiting the recursion when $n = 2$, i.e., computing the products of 2-by-2 matrices by the brute-force algorithm.
8. Solve the recurrence for the number of additions required by Strassen's algorithm. (Assume that n is a power of 2.)
9. V. Pan [Pan78] has discovered a divide-and-conquer matrix multiplication algorithm that is based on multiplying two 70-by-70 matrices using 143,640 multiplications. Find the asymptotic efficiency of Pan's algorithm (you may ignore additions) and compare it with that of Strassen's algorithm.
10. Practical implementations of Strassen's algorithm usually switch to the brute-force method after matrix sizes become smaller than some "crossover point." Run an experiment to determine such crossover point on your computer system.

4.6 Closest-Pair and Convex-Hull Problems by Divide-and-Conquer

In Section 3.3, we discussed the brute-force approach to solving two classic problems of computational geometry: the closest-pair problem and the convex-hull problem. We saw that the two-dimensional versions of these problems can be solved by brute-force algorithms in $\Theta(n^2)$ and $O(n^3)$ time, respectively. In this section, we discuss more sophisticated and asymptotically more efficient algorithms for these problems, which are based on the divide-and-conquer technique.

Closest-Pair Problem

Let $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$ be a set S of n points in the plane, where n, for simplicity, is a power of two. With no loss of generality, we can assume that the points are ordered in ascending order of their x coordinates. (If they were not, we can sort them in $O(n \log n)$ time, e.g., by mergesort.) We can divide the points

https://hemanthrajhemu.github.io

given into two subsets S_1 and S_2 of $n/2$ points each by drawing a vertical line $x = c$ so that $n/2$ points lie to the left of or on the line itself, and $n/2$ points lie to the right of or on the line. (One way of finding an appropriate value for constant c for doing this is to use the median μ of the x coordinates.)

Following the divide-and-conquer approach, we can find recursively the closest pairs for the left subset S_1 and the right subset S_2. Let d_1 and d_2 be the smallest distances between pairs of points in S_1 and S_2, respectively, and let $d = \min\{d_1, d_2\}$. Unfortunately, d is not necessarily the smallest distance between all pairs of points in S_1 and S_2 because a closer pair of points can lie on the opposite sides of the separating line. So, as a step of combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points in the symmetric vertical strip of width $2d$ since the distance between any other pair of points is greater than d (Figure 4.7a).

Let C_1 and C_2 be the subsets of points in the left and right parts of the strip, respectively. Now, for every point $P(x, y)$ in C_1, we need to inspect points in C_2 that may be closer to P than d. Obviously, such points must have their y coordinates in the interval $[y - d, y + d]$. The critical insight here is an observation that there can be no more than six such points because any pair of points in C_2 is at least d apart from each other. (Recall that $d \le d_2$ where d_2 is the smallest distance between pairs of points to the right of the dividing line.) The worst case is illustrated in Figure 4.7b.

Another important observation is that we can maintain lists of points in C_1 and C_2 sorted in ascending order of their y coordinates. (You can think of these lists as projections of the points on the dividing line.) Moreover, this ordering can be maintained not by resorting points on each iteration but rather by merging two previously sorted lists (see algorithm *Merge* in Section 4.1). We can process the C_1 points sequentially while a pointer into the C_2 list scans an interval of width $2d$ to fetch up to six candidates for computing their distances to a current point P of the C_1 list. The time $M(n)$ for this "merging" of solutions to the smaller subproblems is in $O(n)$.

We have the following recurrence for $T(n)$, the running time of this algorithm on n presorted points:

$$T(n) = 2T(n/2) + M(n).$$

Applying the O version of the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in O(n \log n)$. The possible necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm. In fact, this is the best efficiency class we can achieve because it has been proved that any algorithm for this problem must be in $\Omega(n \log n)$ (see [Pre85], p. 188).

Convex-Hull Problem

Let us revisit the convex-hull problem introduced in Section 3.3: find the smallest convex polygon that contains n given points in the plane. We consider here a

https://hemanthrajhemu.github.io

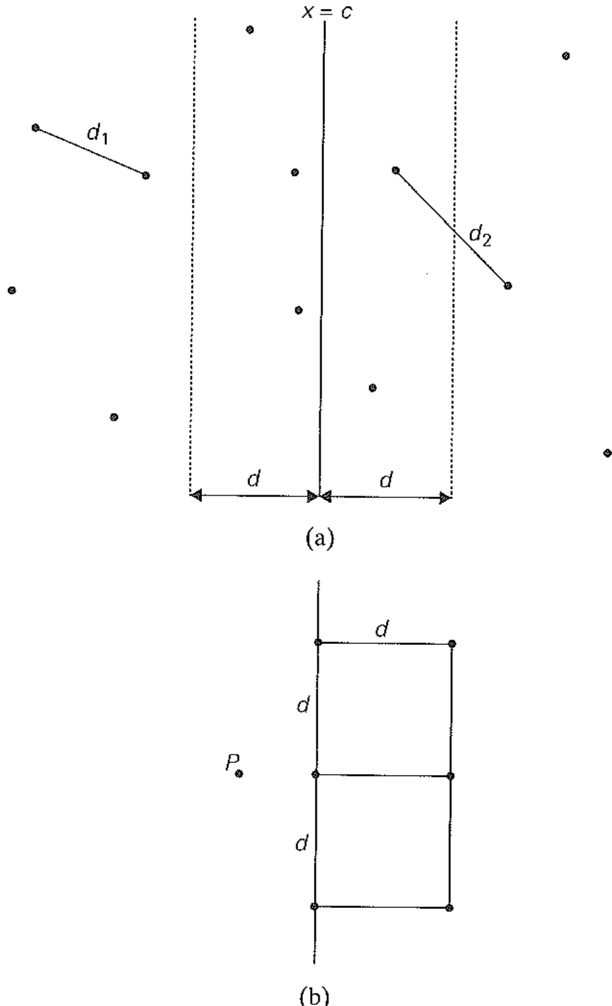


FIGURE 4.7 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem. (b) The six points that may need to be examined for point P.

divide-and-conquer algorithm called ***quickhull*** because of its resemblance to quicksort.

Let $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$ be a set S of $n > 1$ points in the plane. We assume that the points are sorted in increasing order of their x coordinates, with ties resolved by increasing order of the y coordinates of the points involved. It is not difficult to prove the geometrically obvious fact that the leftmost point P_1 and the rightmost point P_n are two distinct extreme points of the set's convex hull

https://hemanthrajhemu.github.io

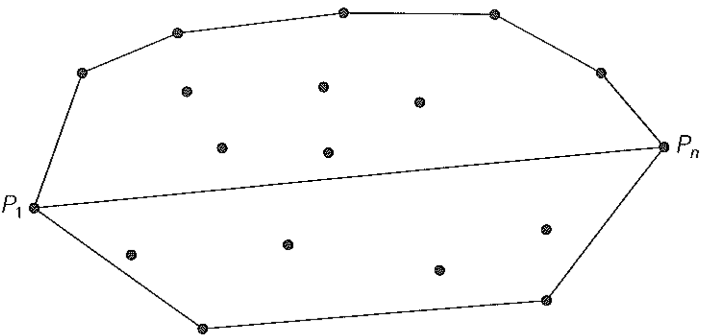


FIGURE 4.8 Upper and lower hulls of a set of points

(Figure 4.8). Let $\overrightarrow{P_1P_n}$ be the straight line through points P_1 and P_n directed from P_1 to P_n. This line separates the points of S into two sets: S_1 is the set of points to the left of or on this line and S_2 is the set of points to the right of or on this line. (We say that point p_3 is to the left of the line $\overrightarrow{p_1p_2}$ directed from point p_1 to point p_2 if $p_1p_2p_3$ forms a counterclockwise cycle. Later, we cite an analytical way to check this condition based on checking the sign of a determinant formed by the coordinates of the three points.) The points of S on the line $\overrightarrow{P_1P_n}$, other than P_1 and P_n, cannot be extreme points of the convex hull and hence are excluded from further consideration.

The boundary of the convex hull of S is made up of two polygonal chains: an "upper" boundary and a "lower" boundary. The "upper" boundary, called the ***upper hull***, is a sequence of line segments with vertices at P_1, some of the points in S_1 (if S_1 is not empty), and P_n. The "lower" boundary, called the ***lower hull***, is a sequence of line segments with vertices at P_1, some of the points in S_2 (if S_2 is not empty) and P_n.

The fact that the convex hull of the entire set S is composed of the upper and lower hulls, which can be constructed independently and in a similar fashion, is a very useful observation that is exploited by several algorithms for this problem.

For concreteness, let us discuss how quickhull proceeds to construct the upper hull; the lower hull can be constructed in the same manner. If S_1 is empty, the upper hull is simply the line segment with the endpoints at P_1 and P_n. If S_1 is not empty, the algorithm identifies vertex P_{max} in S_1, which is the farthest from the line $\overrightarrow{P_1P_n}$ (Figure 4.9) If there is a tie, the point that maximizes the angle $\angle P_{max}P_1P_n$ can be selected. (Note that point P_{max} maximizes the area of the triangle with two vertices at P_1 and P_n and the third at some other point of S_1.) Then the algorithm identifies all the points of set S_1 that are to the left of the line $\overrightarrow{P_1P_{max}}$; these are the points that, along with P_1 and P_{max}, will make up the set $S_{1,1}$. The points of S_1

https://hemanthrajhemu.github.io

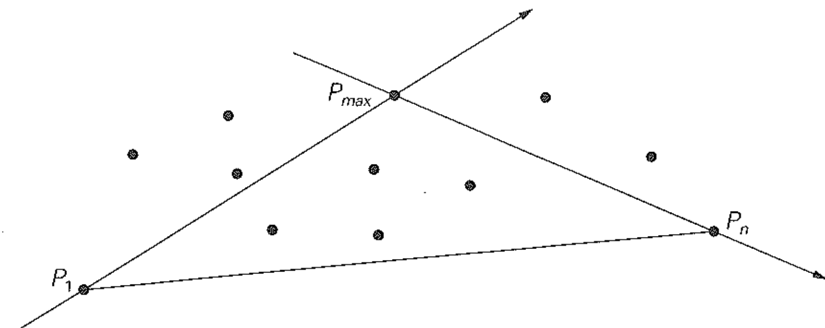


FIGURE 4.9 The idea of quickhull

to the left of the line $\overrightarrow{P_{\max}P_n}$ will make up, along with $P_{\max}$ and P_n, the set $S_{1,2}$. It is not difficult to prove that

- $P_{\max}$ is a vertex of the upper hull;
- the points inside $\triangle P_1 P_{\max} P_n$ cannot be vertices of the upper hull (and hence can be eliminated from further consideration); and
- there are no points to the left of both lines $\overrightarrow{P_1 P_{\max}}$ and $\overrightarrow{P_{\max}P_n}$.

Therefore, the algorithm can continue constructing the upper hulls of $P_1 \cup S_{1,1} \cup P_{\max}$ and $P_{\max} \cup S_{1,2} \cup P_n$ recursively and then simply concatenate them to get the upper hull of the entire set $P_1 \cup S_1 \cup P_n$.

Now we have to figure out how the algorithm's geometric operations can be implemented. Fortunately, we can take advantage of the following very useful fact from analytical geometry: if $p_1 = (x_1, y_1)$, $p_2 = (x_1, y_1)$, and $p_3 = (x_3, y_3)$ are three arbitrary points in the Cartesian plane, then the area of the triangle $\triangle p_1 p_2 p_3$ is equal to one half of the magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1 y_2 + x_3 y_1 + x_2 y_3 - x_3 y_2 - x_2 y_1 - x_1 y_3,$$

while the sign of this expression is positive if and only if the point $p_3 = (x_3, y_3)$ is to the left of the line $\overrightarrow{p_1 p_2}$. Using this formula, we can check in constant time whether a point lies to the left of the line determined by two other points as well as find the distance from the point to the line.

Quickhull has the same $\Theta(n^2)$ worst-case efficiency as quicksort (Problem 8 in the exercises). In the average case, however, we should expect a much better performance. First, the algorithm should benefit from the quicksort-like savings from the on-average balanced split of the problem into two smaller subproblems. Second, a siginificant fraction of the points—namely those inside $\triangle P_1 P_{\max} P_n$ (see

https://hemanthrajhemu.github.io

Figure 4.9)—are eliminated from further processing. Under a natural assumption that points given are chosen randomly from a uniform distribution over some convex region (e.g., a circle or a rectangle), the average-case efficiency of quickhull turns out to be linear [Ove80].

Exercises 4.6

1. **a.** For the one-dimensional version of the closest-pair problem, i.e., for the problem of finding two closest numbers among a given set of n real numbers, design an algorithm that is directly based on the divide-and-conquer technique and determine its efficiency class.

   b. Is it a good algorithm for this problem?

2. Consider the version of the divide-and-conquer two-dimensional closest-pair algorithm in which we simply sort each of the two sets C_1 and C_2 in ascending order of their y coordinates on each recursive call. Assuming that sorting is done by mergesort, set up a recurrence relation for the running time in the worst case and solve it for $n = 2^k$.

3. Implement the divide-and-conquer closest-pair algorithm, outlined in this section, in the language of your choice.

4. Find on the Web a visualization of an algorithm for the closest-pair problem. What algorithm does this visualization represent?

5. The ***Voronoi polygon*** for a point P of a set S of points in the plane is defined to be the perimeter of the set of all points in the plane closer to P than to any other point in S. The union of all the Voronoi polygons of the points in S is called the ***Voronoi diagram*** of S.

   a. What is the Voronoi diagram for a set of three points?

   b. Find on the Web a visualization of an algorithm for generating the Voronoi diagram and study a few examples of such diagrams. Based on your observations, can you tell how the solution to the previous question is generalized to the general case?

6. Explain how one can find point P_{max} in the quickhull algorithm analytically.

7. What is the best-case efficiency of quickhull?

8. Give a specific example of inputs that make the quickhull algorithm run in quadratic time.

9. Implement the quickhull algorithm in the language of your choice.

10. *Shortest path around* There is a fenced area in the two-dimensional Euclidean plane in the shape of a convex polygon with vertices at points $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, ..., $P_n(x_n, y_n)$ (not necessarily in this order). There are two more

https://hemanthrajhemu.github.io

points, $A(x_A, y_A)$ and $B(x_B, y_B)$, such that $x_A < \min\{x_1, x_2, \ldots, x_n\}$ and $x_B > \max\{x_1, x_2, \ldots, x_n\}$. Design a reasonably efficient algorithm for computing the length of the shortest path between A and B. [ORo98], p. 68

SUMMARY

- *Divide-and-conquer* is a general algorithm design technique that solves a problem's instance by dividing it into several smaller instances (ideally, of equal size), solving each of them recursively, and then combining their solutions to get a solution to the original instance of the problem. Many efficient algorithms are based on this technique, although it can be both inapplicable and inferior to simpler algorithmic solutions.

- Running time $T(n)$ of many divide-and-conquer algorithms satisfies the recurrence $T(n) = aT(n/b) + f(n)$. The *Master Theorem* establishes the order of growth of its solutions.

- *Mergesort* is a divide-and-conquer sorting algorithm. It works by dividing an input array into two halves, sorting them recursively, and then *merging* the two sorted halves to get the original array sorted. The algorithm's time efficiency is in $\Theta(n \log n)$ in all cases, with the number of key comparisons being very close to the theoretical minimum. Its principal drawback is a significant extra storage requirement.

- *Quicksort* is a divide-and-conquer sorting algorithm that works by partitioning its input's elements according to their value relative to some preselected element. Quicksort is noted for its superior efficiency among $n \log n$ algorithms for sorting randomly ordered arrays but also for the quadratic worst-case efficiency.

- *Binary search* is a $O(\log n)$ algorithm for searching in sorted arrays. It is an atypical example of an application of the divide-and-conquer technique because it needs to solve just one problem of half the size on each of its iterations.

- The classic traversals of a binary tree—*preorder*, *inorder*, and *postorder*—and similar algorithms that require recursive processing of both left and right subtrees can be considered examples of the divide-and-conquer technique. Their analysis is helped by replacing all the empty subtrees of a given tree with special *external nodes*.

- There is a divide-and-conquer algorithm for multiplying two n-digit integers that requires about $n^{1.585}$ one-digit multiplications.

https://hemanthrajhemu.github.io

- *Strassen's algorithm* needs only seven multiplications to multiply two 2-by-2 matrices but requires more additions than the definition-based algorithm. By exploiting the divide-and-conquer technique, this algorithm can multiply two n-by-n matrices with about $n^{2.807}$ multiplications.
- The divide-and-conquer technique can be successfully applied to two important problems of computational geometry: the closest-pair problem and the convex-hull problem.

https://hemanthrajhemu.github.io

5

Decrease-and-Conquer

Plutarch says that Sertorius, in order to teach his soldiers that perseverance and wit are better than brute force, had two horses brought before them, and set two men to pull out their tails. One of the men was a burly Hercules, who tugged and tugged, but all to no purpose; the other was a sharp, weasel-faced tailor, who plucked one hair at a time, amidst roars of laughter, and soon left the tail quite bare.

—E. Cobham Brewer, *Dictionary of Phrase and Fable*, 1898

The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion). There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 5.1), although reduction-by-two cases do happen occasionally, for example, in algorithms that have to act differently for instances of odd and even sizes.

Consider, as an example, the exponentiation problem of computing a^n for positive integer exponents. The relationship between a solution to an instance of size n and an instance of size $n-1$ is obtained by the obvious formula: $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases} \quad (5.1)$$

https://hemanthrajhemu.github.io

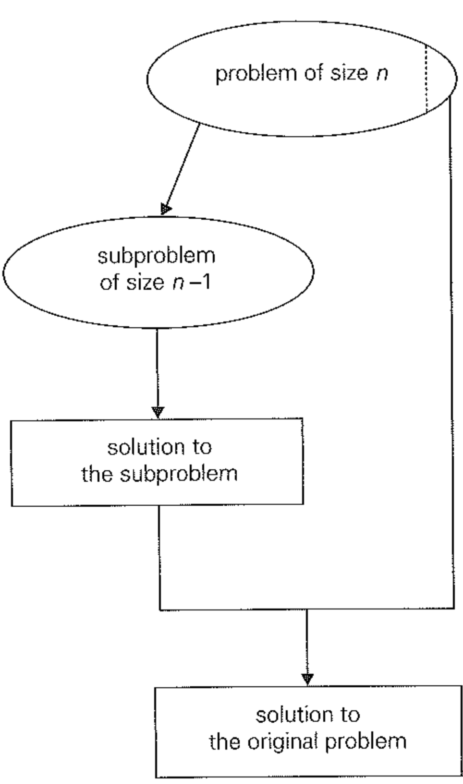


FIGURE 5.1 Decrease (by one)-and-conquer technique

or "bottom up" by multiplying a by itself $n-1$ times. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.) More interesting examples of decrease-by-one algorithms appear in Sections 5.1–5.4.

The ***decrease-by-a-constant-factor*** technique suggests reducing a problem's instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. (Can you give an example of such an algorithm?) The decrease-by-half idea is illustrated in Figure 5.2.

For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size will be to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances of the exponentiation problem with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a. To summarize, we have the following formula:

https://hemanthrajhemu.github.io

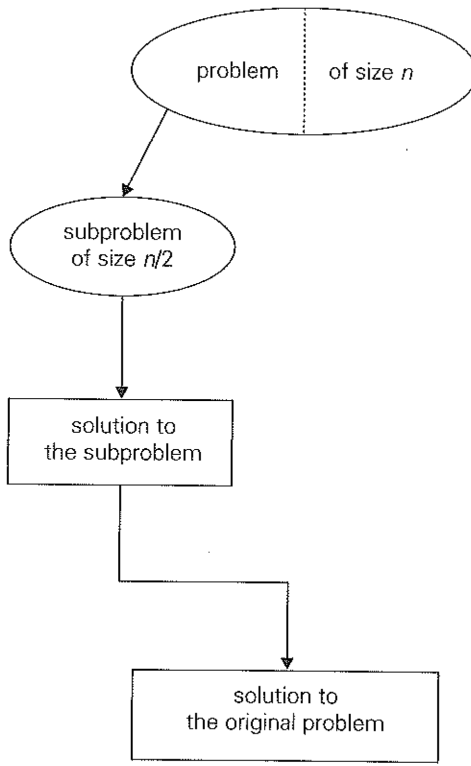


FIGURE 5.2 Decrease (by half)-and-conquer technique

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd and greater than 1} \\ a & \text{if } n = 1. \end{cases} \quad \textbf{(5.2)}$$

If we compute a^n recursively according to formula (5.2) and measure the algorithm's efficiency by the number of multiplications, we should expect the algorithm to be in $O(\log n)$ because, on each iteration, the size is reduced by at least one half at the expense of no more than two multiplications.

Note a difference between this algorithm and the one based on the divide-and-conquer idea of solving two instances of the exponentiation problem of size $n/2$:

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil} & \text{if } n > 1 \\ a & \text{if } n = 1. \end{cases} \quad \textbf{(5.3)}$$

The algorithm based on formula (5.3) is inefficient (why?), whereas the one based on (5.2) is much faster.

https://hemanthrajhemu.github.io

A few other examples of decrease-by-a-constant-factor algorithms are given in Section 5.5 and its exercises. Such algorithms are so efficient, however, that there are few examples of this kind.

Finally, in the ***variable-size-decrease*** variety of decrease-and-conquer, a size reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Though the arguments on the right-hand side are always smaller than those on the left-hand side (at least starting with the second iteration of the algorithm), they are smaller neither by a constant nor by a constant factor. A few other examples of such algorithms appear in Section 5.6.

5.1 Insertion Sort

In this section, we consider an application of the decrease-by-one technique to sorting an array $A[0..n-1]$. Following the technique's idea, we assume that the smaller problem of sorting the array $A[0..n-2]$ has already been solved to give us a sorted array of size $n-1$: $A[0] \leq \ldots \leq A[n-2]$. How can we take advantage of this solution to the smaller problem to get a solution to the original problem by taking into account the element $A[n-1]$? Obviously, all we need is to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there.

There are three reasonable alternatives for doing this. First, we can scan the sorted subarray from left to right until the first element greater than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right before that element. Second, we can scan the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right after that element. These two alternatives are essentially equivalent; usually, it is the second one that is implemented in practice because it is better for sorted and almost-sorted arrays (why?). The resulting algorithm is called ***straight insertion sort*** or simply ***insertion sort***. The third alternative is to use binary search to find an appropriate position for $A[n-1]$ in the sorted portion of the array. The resulting algorithm is called ***binary insertion sort***. We ask you to implement this idea and investigate the efficiency of binary insertion sort in the exercises to this section.

Though insertion sort is clearly based on a recursive idea, it is more efficient to implement this algorithm bottom up, i.e., iteratively. As shown in Figure 5.3, starting with $A[1]$ and ending with $A[n-1]$, $A[i]$ is inserted in its appropriate place among the first i elements of the array that have been already sorted (but, unlike selection sort, are generally not in their final positions).

Here is a pseudocode of this algorithm.

https://hemanthrajhemu.github.io

$$A[0] \leq \ldots \leq A[j] < A[j+1] \leq \ldots \leq A[i-1] \mid A[i] \ldots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

FIGURE 5.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

The operation of the algorithm is illustrated in Figure 5.4.

The basic operation of the algorithm is the key comparison $A[j] > v$. (Why not $j \geq 0$? Because it will almost certainly be faster than the former in an actual computer implementation. Moreover, it is not germane to the algorithm: a better implementation with a sentinel—see Problem 5 in the exercises—eliminates it altogether.)

The number of key comparisons in this algorithm obviously depends on the nature of the input. In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$. Since $v = A[i]$, it happens if and only if

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

FIGURE 5.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

https://hemanthrajhemu.github.io

$A[j] > A[i]$ for $j = i - 1, \ldots, 0$. (Note that we are using the fact that on the ith iteration of insertion sort all the elements preceding $A[i]$ are the first i elements in the input, albeit in the sorted order.) Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), ..., $A[n-2] > A[n-1]$ (for $i = n - 1$). In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort (see Section 3.1).

In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i - 1] \le A[i]$ for every $i = 1, \ldots, n - 1$, i.e., if the input array is already sorted in ascending order. (Though it "makes sense" that the best case of an algorithm happens when the problem is already solved, it is not always the case: recall our discussion of quicksort in Chapter 4.) Thus, for sorted arrays, the number of key comparisons is

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

This very good performance in the best case of sorted arrays is not very useful by itself, because we cannot expect such convenient inputs. However, almost-sorted files arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs. For example, while sorting an array by quicksort, we can stop the algorithm's iterations after subarrays become smaller than some predefined size (say, 10 elements). By that time, the entire array is almost sorted and we can finish the job by applying insertion sort to it. This modification typically decreases the total running time of quicksort by about 10%.

A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of element pairs that are out of order (see Problem 8). It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

This twice-as-fast average-case performance coupled with an excellent efficiency on almost-sorted arrays makes insertion sort stand out among its principal competitors among elementary sorting algorithms, selection sort and bubble sort. In addition, its extension named ***shellsort***, after its inventor D. L. Shell [She59], gives us an even better algorithm for sorting moderately large files (see Problem 10).

https://hemanthrajhemu.github.io

Exercises 5.1

1. *Ferrying soldiers* A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?

2. *Alternating glasses* There are $2n$ glasses standing next to each other in a row, the first n of them filled with a soda drink, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of glass moves. [Gar78], p. 7

3. Design a decrease-by-one algorithm for generating the power set of a set of n elements. (The power set of a set S is the set of all the subsets of S, including the empty set and S itself.)
4. Apply insertion sort to sort the list E, X, A, M, P, L, E in alphabetical order.
5. **a.** What sentinel should be put before the first element of an array being sorted to avoid checking the in-bound condition $j \geq 0$ on each iteration of the inner loop of insertion sort?

   b. Will the version with the sentinel be in the same efficiency class as the original version?
6. Is it possible to implement insertion sort for sorting linked lists? Will it have the same $O(n^2)$ efficiency as the array version?
7. Consider the following version of insertion sort.

```
ALGORITHM  InsertSort2(A[0..n − 1])
    for i ← 1 to n − 1 do
        j ← i − 1
        while j ≥ 0 and A[j] > A[j + 1] do
            swap(A[j], A[j + 1])
            j ← j − 1
```

   What is its time efficiency? How is it compared to that of the version given in the text?
8. Let $A[0..n-1]$ be an array of n sortable elements. (For simplicity, you can assume that all the elements are distinct.) A pair $(A[i], A[j])$ is called an ***inversion*** if $i < j$ and $A[i] > A[j]$.

https://hemanthrajhemu.github.io

a. What arrays of size n have the largest number of inversions and what is this number? Answer the same questions for the smallest number of inversions.

b. Show that the average-case number of key comparisons in insertion sort is given by the formula

$$C_{avg}(n) \approx \frac{n^2}{4}.$$

9. Binary insertion sort uses binary search to find an appropriate position to insert $A[i]$ among the previously sorted $A[0] \leq \ldots \leq A[i-1]$. Determine the worst-case efficiency class of this algorithm.

10. Shellsort (more accurately Shell's sort) is an important sorting algorithm that works by applying insertion sort to each of several interleaving sublists of a given list. On each pass through the list, the sublists in question are formed by stepping through the list with an increment h_i taken from some predefined decreasing sequence of step sizes, $h_1 > \ldots > h_i > \ldots > 1$, which must end with 1. (The algorithm works for any such sequence, though some sequences are known to yield a better efficiency than others. For example, the sequence 1, 4, 13, 40, 121, . . . , used, of course, in reverse, is known to be among the best for this purpose.)

    a. Apply shellsort to the list

    $$S,\ H,\ E,\ L,\ L,\ S,\ O,\ R,\ T,\ I,\ S,\ U,\ S,\ E,\ F,\ U,\ L$$

    b. Is shellsort a stable sorting algorithm?

    c. Implement shellsort, straight insertion sort, binary insertion sort, mergesort, and quicksort in the language of your choice and compare their performance on random arrays of sizes 10^2, 10^3, 10^4, and 10^5 as well as on increasing and decreasing arrays of these sizes.

5.2 Depth-First Search and Breadth-First Search

In the next two sections of this chapter, we deal with very important graph algorithms that can be viewed as applications of the decrease-by-one technique. We assume familiarity with the notion of a graph, its main varieties (undirected, directed, and weighted graphs), the two principal representations of a graph (adjacency matrix and adjacency lists), and such notions as graph connectivity and acyclicity. If needed, a brief review of this material can be found in Section 1.4.

As pointed out in Section 1.3, graphs are interesting structures with a wide variety of applications. Many graph algorithms require processing vertices or edges of a graph in a systematic fashion. There are two principal algorithms for doing such traversals: ***depth-first search (DFS)*** and ***breadth-first search (BFS)***. In

https://hemanthrajhemu.github.io

addition to doing their main job of visiting vertices and traversing edges of a graph, these algorithms have proved to be very useful in investigating several important properties of a graph.

Depth-First Search

Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. (If there are several such vertices, a tie can be resolved arbitrarily. As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure representing the graph. In our examples, we will always break ties by the alphabetical order of the vertices.) This process continues until a dead end—a vertex with no adjacent unvisited vertices—is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

It is convenient to use a stack to trace the operation of depth-first search. We push a vertex onto the stack when the vertex is reached for the first time (i.e., the visit of the vertex starts), and we pop a vertex off the stack when it becomes a dead end (i.e., the visit of the vertex ends).

It is also very useful to accompany a depth-first search traversal by constructing the so-called ***depth-first search forest***. The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a ***tree edge*** because the set of all such edges forms a forest. The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). Such an edge is called a ***back edge*** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest. Figure 5.5 provides an example of a depth-first search traversal, with the traversal's stack and corresponding depth-first search forest shown as well.

Here is a pseudocode of the depth-first search.

ALGORITHM $DFS(G)$

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal

https://hemanthrajhemu.github.io

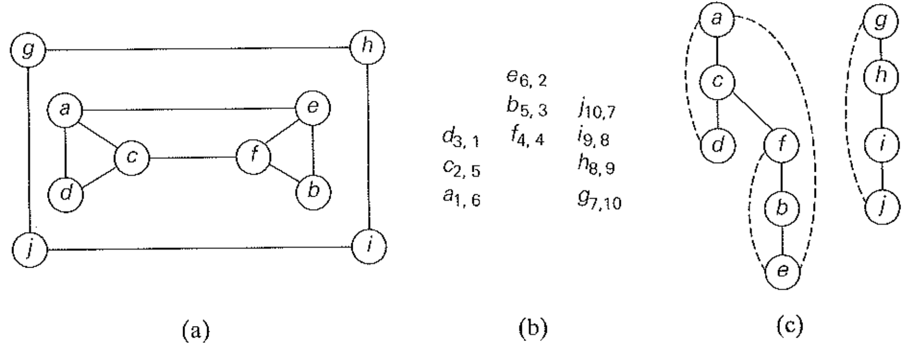


FIGURE 5.5 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto the stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest (with the tree edges shown with solid lines and the back edges shown with dashed lines).

```
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

The brevity of the DFS pseudocode and the ease with which it can be performed by hand may create a wrong impression about the level of sophistication of this algorithm. To appreciate its true power and depth, you should trace the algorithm's action by looking not at a graph's diagram but at its adjacency matrix or adjacency lists. (Try it for the graph in Figure 5.5 or a smaller example.)

How efficient is depth-first search? It is not difficult to see that this algorithm is, in fact, quite efficient since it takes just the time proportional to the size of the data structure used for representing the graph in question. Thus, for the adjacency matrix representation, the traversal's time is in $\Theta(|V|^2)$, and for the adjacency

https://hemanthrajhemu.github.io

list representation, it is in $\Theta(|V| + |E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively.

A DFS forest, which is obtained as a by-product of a DFS traversal, deserves a few comments, too. To begin with, it is not actually a forest. Rather, we can look at it as the given graph with its edges classified by the DFS traversal into two disjoint classes: tree edges and back edges. (No other types are possible for a DFS forest of an undirected graph.) Again, tree edges are edges used by the DFS traversal to reach previously unvisited vertices. If we consider only the edges in this class, we will indeed get a forest. Back edges connect vertices to previously visited vertices other than their immediate predecessors in the traversal. They connect vertices to their ancestors in the forest other than their parents.

A DFS traversal itself and the forest-like representation of a graph it provides have proved to be extremely helpful for the development of efficient algorithms for checking many important properties of graphs.[1] Note that the DFS yields two orderings of vertices: the order in which the vertices are reached for the first time (pushed onto the stack) and the order in which the vertices become dead ends (popped off the stack). These orders are qualitatively different, and various applications can take advantage of either of them.

Important elementary applications of DFS include checking connectivity and checking acyclicity of a graph. Since DFS halts after visiting all the vertices connected by a path to the starting vertex, checking a graph's connectivity can be done as follows. Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the graph's vertices will have been visited. If they have, the graph is connected; otherwise, it is not connected. More generally, we can use DFS for identifying connected components of a graph (how?).

As for checking for a cycle presence in a graph, we can take advantage of the graph's representation in the form of a DFS forest. If the latter does not have back edges, the graph is clearly acyclic. If there is a back edge from some vertex u to its ancestor v (e.g., the back edge from d to a in Figure 5.5c), the graph has a cycle that comprises the path from v to u via a sequence of tree edges in the DFS forest followed by the back edge from u to v.

You will find a few other applications of DFS later in the book, although more sophisticated applications, such as finding articulation points of a graph, are not included. (A vertex of a connected graph is said to be its ***articulation point*** if its removal with all edges incident to it breaks the graph into disjoint pieces.)

Breadth-First Search

If depth-first search is a traversal for the brave (the algorithm goes as far from "home" as it can), breadth-first search is a traversal for the cautious. It proceeds in

1. The discovery of several such applications was an important breakthrough achieved by the two American computer scientists John Hopcroft and Robert Tarjan in the 1970s. For this and other contributions, they subsequently won the Turing Award—the most important prize given in theoretical computer science [Hop87, Tar87].

https://hemanthrajhemu.github.io

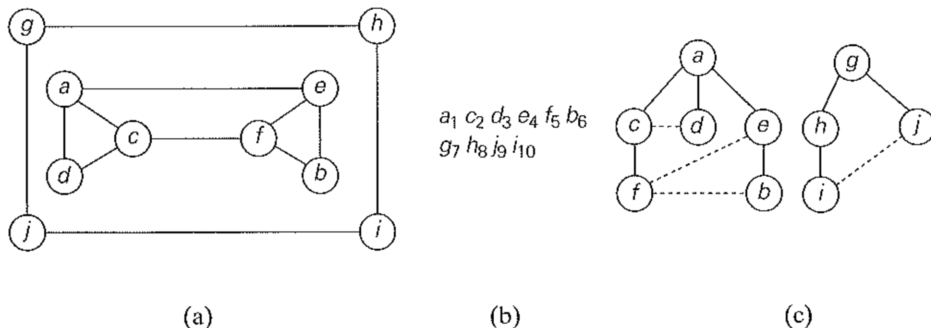


FIGURE 5.6 Example of a BFS traversal. (a) Graph. (b) Traversal's queue, with the numbers indicating the order in which the vertices were visited, i.e., added to (or removed from) the queue. (c) BFS forest (with the tree edges shown with solid lines and the cross edges shown with dotted lines).

a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

It is convenient to use a queue (note the difference from depth-first search!) to trace the operation of breadth-first search. The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called ***breadth-first search forest***. The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a ***tree edge***. If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a ***cross edge***. Figure 5.6 provides an example of a breadth-first search traversal, with the traversal's queue and corresponding breadth-first search forest shown.

Here is a pseudocode of the breadth-first search.

ALGORITHM $BFS(G)$

//Implements a breadth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal

https://hemanthrajhemu.github.io

```
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;   mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

Breadth-first search has the same efficiency as depth-first search: it is in $\Theta(|V|^2)$ for the adjacency matrix representation and in $\Theta(|V| + |E|)$ for the adjacency list representation. Unlike depth-first search, it yields a single ordering of vertices because the queue is a FIFO (first-in first-out) structure and hence the order in which vertices are added to the queue is the same order in which they are removed from it. As to the structure of a BFS forest of an undirected graph, it can also have two kinds of edges: tree edges and cross edges. Tree edges are the ones used to reach previously unvisited vertices. Cross edges connect vertices to those visited before, but, unlike back edges in a DFS tree, they connect vertices either on the same or adjacent levels of a BFS tree.

Finally, BFS can be used to check connectivity and acyclicity of a graph, essentially in the same manner as DFS can. It is not applicable, however, for several less straightforward applications such as finding articulation points. On the other hand, it can be helpful in some situations where DFS cannot. For example, BFS can be used for finding a path with the fewest number of edges between two given vertices. We start a BFS traversal at one of the two vertices given and stop it as soon as the other vertex is reached. The simple path from the root of the BFS tree to the second vertex is the path sought. For example, path a–b–c–g in the graph in Figure 5.7 has the fewest number of edges among all the paths between vertices a and g. Although the correctness of this application appears to stem immediately from the way BFS operates, a mathematical proof of its validity is not quite elementary (see, e.g., [Cor01]).

Table 5.1 summarizes the main facts about depth-first search and breadth-first search.

https://hemanthrajhemu.github.io

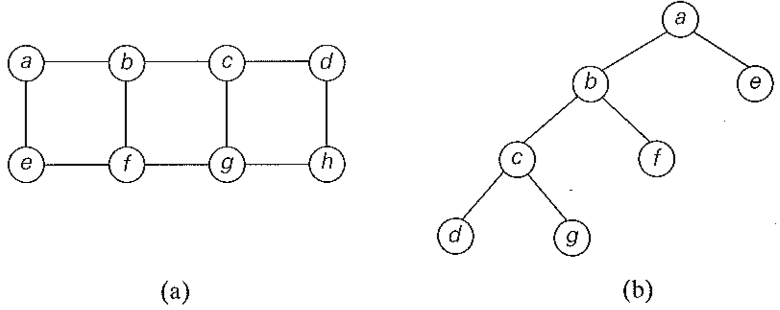


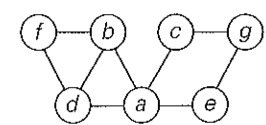
FIGURE 5.7 Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g.

TABLE 5.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	**BFS**
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacent matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacent lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Exercises 5.2

1. Consider the following graph.



https://hemanthrajhemu.github.io

a. Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.)

b. Starting at vertex a and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).

2. If we define sparse graphs as graphs for which $|E| \in O(|V|)$, which implementation of DFS will have a better time efficiency for such graphs, the one that uses the adjacency matrix or the one that uses the adjacency lists?

3. Let G be a graph with n vertices and m edges.

a. True or false: All its DFS forests (for traversals starting at different vertices) will have the same number of trees?

b. True or false: All its DFS forests will have the same number of tree edges and the same number of back edges?

4. Traverse the graph of Problem 1 by breadth-first search and construct the corresponding breadth-first search tree. Start the traversal at vertex a and resolve ties by the vertex alphabetical order.

5. Prove that a cross edge in a BFS tree of an undirected graph can connect vertices only on either the same level or on two adjacent levels of a BFS tree.

6. a. Explain how one can check a graph's acyclicity by using breadth-first search.

b. Does either of the two traversals—DFS or BFS—always find a cycle faster than the other? If you answer yes, indicate which of them is better and explain why it is the case; if you answer no, give two examples supporting your answer.

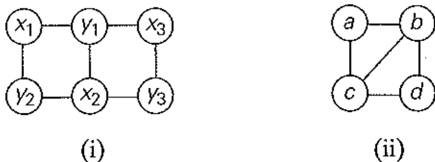
7. Explain how one can identify connected components of a graph by using

a. a depth-first search.

b. a breadth-first search.

8. A graph is said to be ***bipartite*** if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. (One can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called ***2-colorable***). For example, graph (i) is bipartite while graph (ii) is not.

https://hemanthrajhemu.github.io



(i) (ii)

a. Design a DFS-based algorithm for checking whether a graph is bipartite.

b. Design a BFS-based algorithm for checking whether a graph is bipartite.

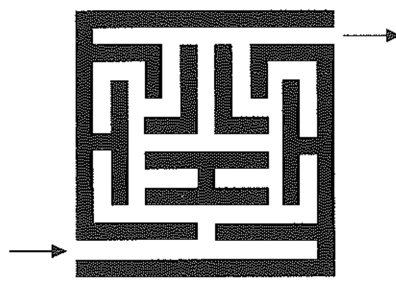
9. Write a program that, for a given graph, outputs

a. vertices of each connected component;

b. its cycle or a message that the graph is acyclic.

10. One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze.

a. Construct such a graph for the following maze.



b. Which traversal—DFS or BFS—would you use if you found yourself in a maze and why?

5.3 Topological Sorting

In this section, we discuss an important problem for directed graphs. Before we pose this problem though, let us review a few basic facts about directed graphs themselves. A ***directed graph***, or ***digraph*** for short, is a graph with directions specified for all its edges (Figure 5.8a is an example). The adjacency matrix and adjacency lists are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be

https://hemanthrajhemu.github.io

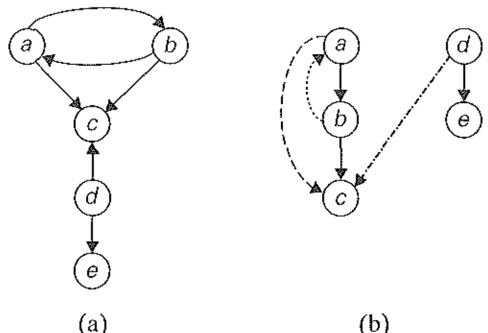


FIGURE 5.8 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs, but the structure of corresponding forests can be more complex. Thus, even for the simple example in Figure 5.8a, the depth-first search forest (Figure 5.8b) exhibits all four types of edges possible in a DFS forest of a directed graph: ***tree edges*** (ab, bc, de), ***back edges*** (ba) from vertices to their ancestors, ***forward edges*** (ac) from vertices to their descendants in the tree other than their children, and ***cross edges*** (dc), which are none of the aforementioned types.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. (A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor.) Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

Directions on a graph's edges lead to new questions about the graph that are either meaningless or trivial for undirected graphs. In this section, we discuss one such problem. As a motivating example, consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements (Figure 5.9). In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex

https://hemanthrajhemu.github.io

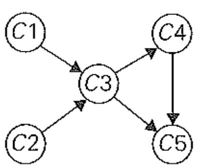


FIGURE 5.9 Digraph representing the prerequisite structure of five courses

where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called ***topological sorting***. It can be posed for an arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting to be possible, a digraph must be a dag. It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible; i.e., if a digraph has no cycles, the topological sorting problem for it has a solution. Moreover, there are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead ends (i.e., are popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Why does the algorithm work? When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices popped off before v. (Otherwise, (u, v) would have been a back edge.) Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

Figure 5.10 illustrates an application of this algorithm to the digraph in Figure 5.9. Note that in Figure 5.10c, we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a con-

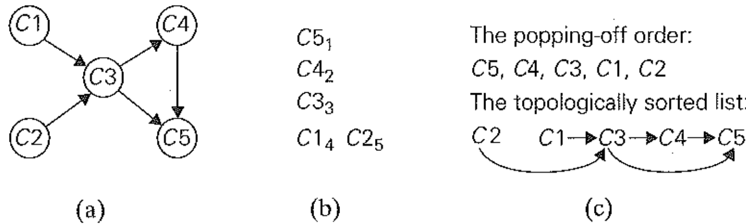


FIGURE 5.10 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

https://hemanthrajhemu.github.io

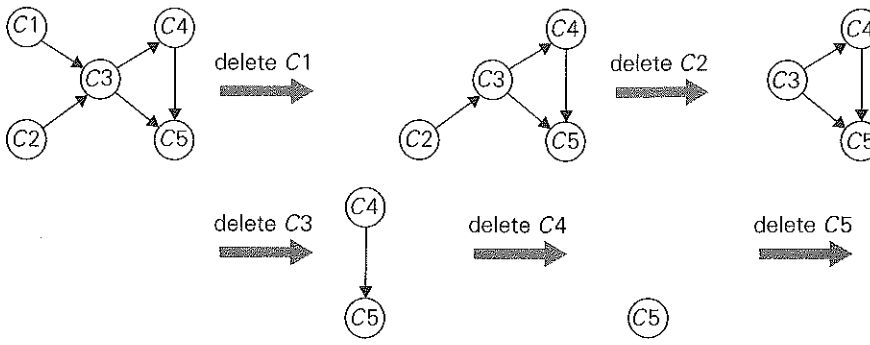


FIGURE 5.11 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

venient way to check visually the correctness of a solution to an instance of the topological sorting problem.

The second algorithm is based on a direct implementation of the decrease (by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there is none, stop because the problem cannot be solved—see Problem 6a.) The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 5.11.

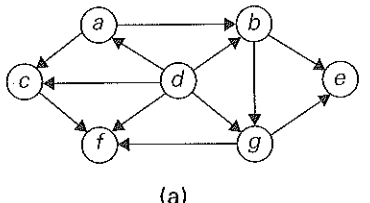
Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

The tiny size of the example we used might create a wrong impression about the topological sorting problem. But imagine a large project—e.g., in construction or research—that involves thousands of interrelated tasks with known prerequisites. The first thing you should do in such a situation is to make sure that the set of given prerequisites is not contradictory. The convenient way of doing this is to solve the topological sorting problem for the project's digraph. Only then can you start thinking about scheduling your tasks to, say, minimize the total completion time of the project. This would require, of course, other algorithms that you can find in general books on operations research or in special ones on so-called CPM (Critical Path Method) and PERT (Program Evaluation and Review Technique) methodologies.

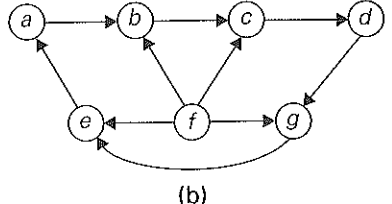
https://hemanthrajhemu.github.io

Exercises 5.3

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs.



(a)



(b)

2. **a.** Prove that the topological sorting problem has a solution for a digraph if and only if it is a dag.

   b. For a digraph with n vertices, what is the largest number of distinct solutions the topological sorting problem can have?

3. **a.** What is the time efficiency of the DFS-based algorithm for topological sorting?

   b. How can one modify the DFS-based algorithm to avoid reversing the vertex ordering generated by DFS?

4. Can one use the order in which vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?

5. Apply the source-removal algorithm to the digraphs of Problem 1.

6. **a.** Prove that a dag must have at least one source.

   b. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency matrix? What is the time efficiency of this operation?

   c. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency lists? What is the time efficiency of this operation?

7. Can you implement the source-removal algorithm for a digraph represented by its adjacency lists so that its running time is in $O(|V| + |E|)$?

8. Implement the two topological sorting algorithms in the language of your choice. Run an experiment to compare their running times.

9. A digraph is called ***strongly connected*** if for any pair of two distinct vertices u and v there exists a directed path from u to v and a directed path from v to u. In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths of the digraph; these subsets are called ***strongly connected components***. There are two DFS-based

https://hemanthrajhemu.github.io

algorithms for identifying strongly connected components. Here is the simpler (but somewhat less efficient) one of the two.

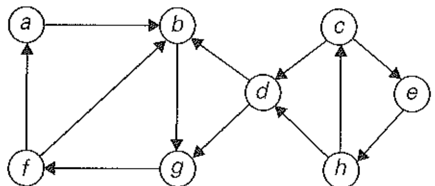
Step 1 Do a DFS traversal of the digraph given and number its vertices in the order that they become dead ends.

Step 2 Reverse the directions of all the edges of the digraph.

Step 3 Do a DFS traversal of the new digraph by starting (and, if necessary, restarting) the traversal at the highest numbered vertex among still unvisited vertices.

The strongly connected components are exactly the subsets of vertices in each DFS tree obtained during the last traversal.

a. Apply this algorithm to the following digraph to determine its strongly connected components.



b. What is the time efficiency class of this algorithm? Give separate answers for the adjacency matrix representation and adjacency list representation of an input graph.

c. How many strongly connected components does a dag have?

10. *Celebrity problem* A celebrity among a group of n people is a person who knows nobody but is known by everybody else. The task is to identify a celebrity by only asking questions to people of the form: "Do you know him/her?" Design an efficient algorithm to identify a celebrity or determine that the group has no such person. How many questions does your algorithm need in the worst case?

5.4 Algorithms for Generating Combinatorial Objects

In this section, we keep our promise to discuss algorithms for generating combinatorial objects. The most important types of combinatorial objects are permutations, combinations, and subsets of a given set. They typically arise in problems that require a consideration of different choices. We already encountered them in Chapter 3 when we discussed exhaustive search. Combinatorial objects are studied in a branch of discrete mathematics called combinatorics. Mathematicians, of

https://hemanthrajhemu.github.io

course, are primarily interested in different counting formulas; we should be grateful for such formulas because they tell us how many items need to be generated. (In particular, they warn us that the number of combinatorial objects typically grows exponentially or even faster as a function of the problem's size.) But our primary interest here lies in algorithms for generating combinatorial objects, not just in counting them.

Generating Permutations

We start with permutations. For simplicity, we assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n; more generally, they can be interpreted as indices of elements in an n-element set $\{a_1, \ldots, a_n\}$. What would the decrease-by-one technique suggest for the problem of generating all $n!$ permutations of $\{1, \ldots, n\}$? The smaller-by-one problem is to generate all $(n-1)!$ permutations. Assuming that the smaller problem is solved, we can get a solution to the larger one by inserting n in each of the n possible positions among elements of every permutation of $n-1$ elements. All the permutations obtained in this fashion will be distinct (why?), and their total number will be $n(n-1)! = n!$. Hence, we will obtain all the permutations of $\{1, \ldots, n\}$.

We can insert n in the previously generated permutations either left to right or right to left. It turns out that it is beneficial to start with inserting n into $12 \ldots (n-1)$ by moving right to left and then switch direction every time a new permutation of $\{1, \ldots, n-1\}$ needs to be processed. An example of applying this approach bottom up for $n = 3$ is given in Figure 5.12.

The advantage of this order stems from the fact that it satisfies the ***minimal-change*** requirement: each permutation can be obtained from its immediate predecessor by exchanging just two elements in it. (Check this for the permutations generated in Figure 5.12.) The minimal-change requirement is beneficial both for the algorithm's speed and for applications using the permutations. For example, in Section 3.4, we needed permutations of cities to solve the traveling salesman problem by exhaustive search. If such permutations are generated by a minimal-change algorithm, we can compute the length of a new tour from the length of its predecessor in constant rather than linear time (how?).

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

FIGURE 5.12 Generating permutations bottom up

https://hemanthrajhemu.github.io

It is possible to get the same ordering of permutations of n elements without explicitly generating permutations for smaller values of n. It can be done by associating a direction with each element k in a permutation. We indicate such a direction by a small arrow written above the element in question, e.g.,

$$\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{4}\,\overleftarrow{1}.$$

The element k is said to be ***mobile*** in such an arrow-marked permutation if its arrow points to a smaller number adjacent to it. For example, for the permutation $\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{4}\,\overleftarrow{1}$, 3 and 4 are mobile while 2 and 1 are not. Using the notion of a mobile element, we can give the following description of the ***Johnson-Trotter algorithm*** for generating permutations.

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k and the adjacent integer k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

Here is an application of this algorithm for $n = 3$, with the largest mobile integer shown in bold:

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2} \quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}} \quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$$

This algorithm is one of the most efficient for generating permutations; it can be implemented to run in time proportional to the number of permutations, i.e., in $\Theta(n!)$. Of course, it is horribly slow for all but very small values of n; however, this is not the algorithm's fault but rather the "fault" of the problem: it simply asks to generate too many items.

One can argue that the permutation ordering generated by the Johnson-Trotter algorithm is not quite natural; e.g., the natural place for permutation $n\ n-1 \ldots 1$ seems to be the last one on the list. This would be the case if permutations were listed in increasing order—also called the ***lexicographic order***—which is the order in which they would be listed in a dictionary if the numbers were interpreted as letters of an alphabet:

123 132 213 231 312 321.

https://hemanthrajhemu.github.io

So how can we generate the permutation following $a_1a_2 \ldots a_{n-1}a_n$ in lexicographic order? If $a_{n-1} < a_n$, we can simply transpose these last two elements. For example, 123 is followed by 132. If $a_{n-1} > a_n$, we have to engage a_{n-2}. If $a_{n-2} < a_{n-1}$, we should rearrange the last three elements by increasing the $(n-2)$th element as little as possible by putting there the next larger than a_{n-2} element chosen from a_{n-1} and a_n and filling positions $n-1$ and n with the remaining two of the three elements a_{n-2}, a_{n-1}, and a_n in increasing order. For example, 132 is followed by 213 while 231 is followed by 312. In general, we scan a current permutation from right to left looking for the first pair of consecutive elements a_i and a_{i+1} such that $a_i < a_{i+1}$ (and, hence, $a_{i+1} > \ldots > a_n$). Then we find the smallest element in the tail that is larger than a_i, i.e., $\min\{a_j \mid a_j > a_i,\ j > i\}$, and put it in position i; the positions from $i+1$ through n are filled with the elements $a_i, a_{i+1}, \ldots, a_n$, from which the element put in the ith position has been eliminated, in increasing order. For example, 163542 would be followed by 164235. We leave writing a complete pseudocode of this algorithm for the exercises.

Generating Subsets

Recall that in Section 3.4 we examined the knapsack problem that asks to find the most valuable subset of items that fits a knapsack of a given capacity. The exhaustive-search approach to solving this problem discussed there was based on generating all subsets of a given set of items. In this section, we discuss algorithms for generating all 2^n subsets of an abstract set $A = \{a_1, \ldots, a_n\}$. (Mathematicians call the set of all subsets of a set its ***power set***.)

The decrease-by-one idea is immediately applicable to this problem, too. All subsets of $A = \{a_1, \ldots, a_n\}$ can be divided into two groups: those that do not contain a_n and those that do. The former group is nothing but all the subsets of $\{a_1, \ldots, a_{n-1}\}$, while each and every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \ldots, a_{n-1}\}$. Thus, once we have a list of all subsets of $\{a_1, \ldots, a_{n-1}\}$, we can get all the subsets of $\{a_1, \ldots, a_n\}$ by adding to the list all its elements with a_n put into each of them. An application of this algorithm to generate all subsets of $\{a_1, a_2, a_3\}$ is illustrated in Figure 5.13.

n	**subsets**							
0	$\emptyset$							
1	$\emptyset$	$\{a_1\}$						
2	$\emptyset$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\emptyset$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

FIGURE 5.13 Generating subsets bottom up

https://hemanthrajhemu.github.io

Similarly to generating permutations, we do not have to generate power sets of smaller sets. A convenient way of solving the problem directly is based on a one-to-one correspondence between all 2^n subsets of an n element set $A = \{a_1, \ldots, a_n\}$ and all 2^n bit strings $b_1, \ldots, b_n$ of length n. The easiest way to establish such a correspondence is to assign to a subset the bit string in which $b_i = 1$ if a_i belongs to the subset and $b_i = 0$ if a_i does not belong to it. (We mentioned this idea of bit vectors in Section 1.4.) For example, the bit string 000 will correspond to the empty subset of a three-element set, 111 will correspond to the set itself, i.e., $\{a_1, a_2, a_3\}$, while 110 will represent $\{a_1, a_2\}$. With this correspondence in place, we can generate all the bit strings of length n by generating successive binary numbers from 0 to $2^n - 1$, padded, when necessary, with an appropriate number of leading 0's. For example, for the case of $n = 3$, we obtain

bit strings	000	001	010	011	100	101	110	111
subsets	$\emptyset$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Note that while the bit strings are generated by this algorithm in the lexicographic order (in the two-symbol alphabet of 0 and 1), the order of the subsets looks anything but natural. For example, we might want the so-called ***squashed order***, in which any subset involving a_j can be listed only after all the subsets involving $a_1, \ldots, a_{j-1}$, as was the case for the list of the three-element set in Figure 5.13. It is easy to adjust the bit string–based algorithm to yield a squashed ordering of the subsets involved (Problem 6).

A more challenging question is whether there exists a minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit. (In the language of subsets, we want every subset to differ from its immediate predecessor by either an addition or a deletion, but not both, of a single element). The answer to this question is yes (Problem 9); for example, for $n = 3$, we can get

000 001 011 010 110 111 101 100.

Such a sequence of bit strings is called the ***binary reflected Gray code***. Gray codes have many interesting properties and a few useful applications; you can read about them in such books as [Bru04].

Exercises 5.4

1. Is it realistic to implement an algorithm that requires generating all permutations of a 25-element set on your computer? What about all the subsets of such a set?

2. Generate all permutations of {1, 2, 3, 4} by

   a. the bottom-up minimal-change algorithm.

https://hemanthrajhemu.github.io

b. the Johnson-Trotter algorithm.

c. the lexicographic-order algorithm.

3. Write a computer program for generating permutations in lexicographic order.

4. Consider the following implementation of the algorithm for generating permutations discovered by B. Heap [Hea63].

```
ALGORITHM HeapPermute(n)
    //Implements Heap's algorithm for generating permutations
    //Input: A positive integer n and a global array A[1..n]
    //Output: All permutations of elements of A
    if n = 1
        write A
    else
        for i ← 1 to n do
            HeapPermute(n − 1)
            if n is odd
                swap A[1] and A[n]
            else swap A[i] and A[n]
```

a. Trace the algorithm by hand for $n = 2$, 3, and 4.

b. Prove correctness of Heap's algorithm.

c. What is the time efficiency of *HeapPermute*?

5. Generate all the subsets of a four-element set $A = \{a_1, a_2, a_3, a_4\}$ by each of the two algorithms outlined in this section.

6. What simple trick would make the bit string–based algorithm generate subsets in squashed order?

7. Write a pseudocode for a recursive algorithm for generating all 2^n bit strings of length n.

8. Write a nonrecursive algorithm for generating 2^n bit strings of length n that implements bit strings as arrays and does not use binary additions.

9. **a.** Use the decrease-by-one technique to generate a Gray code for $n = 4$.

b. Design a general decrease-by-one algorithm for generating a Gray code of order n.

10. Design a decrease-and-conquer algorithm for generating all combinations of k items chosen from n, i.e., all k-element subsets of a given n-element set. Is your algorithm a minimal-change algorithm?

https://hemanthrajhemu.github.io

11. *Gray code and the Tower of Hanoi*

a. Show that the disk moves made in the classic recursive algorithm for the Tower of Hanoi puzzle can be used for generating the binary reflected Gray code.

b. Show how the binary reflected Gray code can be used for solving the Tower of Hanoi puzzle.

5.5 Decrease-by-a-Constant-Factor Algorithms

You may recall from the introduction to this chapter that decrease-by-a-constant-factor is the second major variety of decrease-and-conquer. You have already encountered examples of this design technique in this book: binary search (Section 4.3) and exponentiation by squaring (introduction to Section 5.1). In this section, you will find a few other examples of algorithms based on the decrease-by-a-constant-factor idea. We should not expect a wealth of examples of this kind, however, because these algorithms are usually logarithmic and, being very fast, do not happen often; a reduction by a factor other than two is especially rare.

Fake-Coin Problem

Of several versions of the fake-coin identification problem, we consider the one that best illustrates the decrease-by-a-constant-factor strategy. Among n identically looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much. The problem is to design an efficient algorithm for detecting the fake coin. An easier version of the problem—the one we discuss here—assumes that it is known whether the fake coin is lighter or heavier than the genuine one.[2] (We assume that the fake coin is lighter.)

The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin apart if n is odd, and put the two piles on the scale. If the piles weigh the same, the coin put aside must be fake; otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin. Note that even though we divide the coins into two subsets, after one weighing we are left to solve a single problem of half the

2. A much more challenging version assumes no additional information about the relative weights of the fake and genuine coins or even the presence of the fake coin among n given coins. We pursue this more difficult version in the exercises to Section 11.2.

https://hemanthrajhemu.github.io

original size. Therefore, according to our classification of the design techniques, it is a decrease (by half)-and-conquer rather than a divide-and-conquer algorithm.

We can easily set up a recurrence relation for the number of weighings $W(n)$ needed by this algorithm in the worst case:

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \qquad W(1) = 0.$$

This recurrence should look familiar to you. Indeed, it is almost identical to the one for the worst-case number of comparisons in binary search. (The difference is in the initial condition.) This similarity is not really surprising, since both algorithms are based on the same technique of halving an instance size. The solution to the recurrence for the number of weighings is also very similar to the one we had for binary search: $W(n) = \lfloor \log_2 n \rfloor$.

This stuff should look elementary by now, if not outright boring. But wait: the interesting point here is the fact that this algorithm is not the most efficient solution. We would be better off dividing the coins not into two but into *three* piles of about $n/3$ coins each. (Details of a precise formulation are developed in the exercises. Do not miss it! If your instructor forgets, demand the instructor to assign Problem 3.) After weighing two of the piles, we can reduce the instance size by a factor of three. Accordingly, we should expect the number of weighings to be about $\log_3 n$, which is smaller than $\log_2 n$. (Can you tell by what factor?)

Multiplication à la Russe

Now we consider a nonorthodox algorithm for multiplying two positive integers called ***multiplication à la russe***, or the ***Russian peasant method***. Let n and m be positive integers whose product we want to compute, and let us measure the instance size by the value of n. Now, if n is even, an instance of half the size has to deal with $n/2$, and we have an obvious formula relating the solution to the problem's larger instance to the solution to the smaller one:

$$n \cdot m = \frac{n}{2} \cdot 2m.$$

If n is odd, we need only a slight adjustment of this formula:

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m.$$

Using these formulas and the trivial case of $1 \cdot m = m$ to stop, we can compute product $n \cdot m$ either recursively or iteratively. An example of computing $50 \cdot 65$ with this algorithm is given in Figure 5.14. Note that all the extra addends shown in parentheses in Figure 5.14a are in the rows that have odd values in the first column. Therefore we can find the product by simply adding all the elements in the m column that have an odd number in the n column (Figure 5.14b).

Also note that the algorithm involves just the simple operations of halving, doubling, and adding—a feature that might be attractive, for example, to those

https://hemanthrajhemu.github.io

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1, 040	
1	2, 080	(+1040)
	2, 080	+(130 + 1040) = 3, 250

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1, 040	1, 040
1	2, 080	2, 080
		3, 250

(b)

FIGURE 5.14 Computing $50 \cdot 65$ by multiplication à la russe

who do not want to memorize the table of multiplications. It is this feature of the algorithm that most probably made it attractive to Russian peasants who, according to Western visitors, used it widely in the nineteenth century, and for whom the method is named. (In fact, the algorithm's idea was used by Egyptian mathematicians as early as 1650 B.C. [Cha98], p. 16.) It also leads to very fast hardware implementation since doubling and halving of binary numbers can be performed using shifts, which are among the most basic operations at the machine level.

Josephus Problem

Our last example is the ***Josephus problem***, named for Flavius Josephus, a famous Jewish historian who participated in and chronicled the Jewish revolt of 66–70 C.E. against the Romans. Josephus, as a general, managed to hold the fortress of Jotapata for 47 days, but after the fall of the city he took refuge with 40 diehards in a nearby cave. There, the rebels voted to perish rather than surrender. Josephus proposed that each man in turn should dispatch his neighbor, the order to be determined by casting lots. Josephus contrived to draw the last lot, and, as one of the two surviving men in the cave, he prevailed upon his intended victim to surrender to the Romans.

So let n people numbered 1 to n stand in a circle. Starting the grim count with person number 1, we eliminate every second person until only one survivor is left. The problem is to determine the survivor's number $J(n)$. For example (Figure 5.15), if n is 6, people in positions 2, 4, and 6 will be eliminated on the first pass through the circle, and people in initial positions 3 and 1 will be eliminated on the second pass, leaving a sole survivor in initial position 5—thus, $J(6) = 5$. To give another example, if n is 7, people in positions 2, 4, 6, and 1 will be eliminated

https://hemanthrajhemu.github.io

1_2 6_1 2_1 5 3_2 4_1

(a)

1_1 7 2_1 6_1 3_2 5_2 4_1

(b)

FIGURE 5.15 Instances of the Josephus problem for (a) $n = 6$ and (b) $n = 7$. Subscript numbers indicate the pass on which the person in that position is eliminated. The solutions are $J(6) = 5$ and $J(7) = 7$, respectively.

on the first pass (it is more convenient to include 1 in the first pass) and people in positions 5 and, for convenience, 3 on the second—thus, $J(7) = 7$.

It is convenient to consider the cases of even and odd n's separately. If n is even, i.e., $n = 2k$, the first pass through the circle yields an instance of exactly the same problem but half its initial size. The only difference is in position numbering; for example, a person in initial position 3 will be in position 2 for the second pass, a person in initial position 5 will be in position 3, and so on (check Figure 5.15a). It is easy to see that to get the initial position of a person, we simply need to multiply his new position by two and subtract one. This relationship will hold, in particular, for the survivor, i.e.,

$$J(2k) = 2J(k) - 1.$$

Let us now consider the case of an odd n ($n > 1$), i.e., $n = 2k + 1$. The first pass eliminates people in all even positions. If we add to this the elimination of the person in position 1 right after that, we are left with an instance of size k. Here, to get the initial position that corresponds to the new position numbering, we have to multiply the new position number by two and add one (check Figure 5.15b). Thus, for odd values of n, we get

$$J(2k + 1) = 2J(k) + 1.$$

Can we get a closed-form solution to the two-case recurrence (subject to the initial condition $J(1) = 1$)? The answer is yes, though getting it requires more ingenuity than just applying backward substitutions. In fact, one way to find a solution is to apply forward substitutions to get, say, the first 15 values of $J(n)$, discern a pattern, and then prove its general validity by mathematical induction. We leave the execution of this plan to the exercises; alternatively, you can look it up in [Gra94], whose exposition of the Josephus problem we have been following. Interestingly, the most elegant form of the closed-form answer involves the binary representation of size n: $J(n)$ can be obtained by a one-bit cyclic shift left of n itself! For example, $J(6) = J(110_2) = 101_2 = 5$ and $J(7) = J(111_2) = 111_2 = 7$.

https://hemanthrajhemu.github.io

Exercises 5.5

1. Design a decrease-by-half algorithm for computing $\lfloor \log_2 n \rfloor$ and determine its time efficiency.

2. Consider ***ternary search***—the following algorithm for searching in a sorted array $A[0..n-1]$. If $n = 1$, simply compare the search key K with the single element of the array; otherwise, search recursively by comparing K with $A[\lfloor n/3 \rfloor]$, and if K is larger, compare it with $A[\lfloor 2n/3 \rfloor]$ to determine in which third of the array to continue the search.

   a. What design technique is this algorithm based on?

   b. Set up a recurrence for the number of key comparisons in the worst case. (You may assume that $n = 3^k$.)

   c. Solve the recurrence for $n = 3^k$.

   d. Compare this algorithm's efficiency with that of binary search.

3. **a.** Write a pseudocode for the divide-into-three algorithm for the fake-coin problem. (Make sure that your algorithm handles properly all values of n, not only those that are multiples of 3.)

   b. Set up a recurrence relation for the number of weighings in the divide-into-three algorithm for the fake-coin problem and solve it for $n = 3^k$.

   c. For large values of n, about how many times faster is this algorithm than the one based on dividing coins into two piles? (Your answer should not depend on n.)

4. Apply multiplication à la russe to compute $26 \cdot 47$.

5. **a.** From the standpoint of time efficiency, does it matter whether we multiply n by m or m by n by the multiplication à la russe algorithm?

   b. What is the efficiency class of multiplication à la russe?

6. Write a pseudocode for the multiplication à la russe algorithm.

7. Find $J(40)$—the solution to the Josephus problem for $n = 40$.

8. Prove that the solution to the Josephus problem is 1 for every n that is a power of 2.

9. For the Josephus problem,

   a. compute $J(n)$ for $n = 1, 2, \ldots, 15$.

   b. discern a pattern in the solutions for the first fifteen values of n and prove its general validity.

   c. prove the validity of getting $J(n)$ by a one-bit cyclic shift left of the binary representation of n.

https://hemanthrajhemu.github.io

5.6 Variable-Size-Decrease Algorithms

As mentioned in the introduction to this chapter, in the third principal variety of decrease-and-conquer, the size reduction pattern varies from one iteration of the algorithm to another. Euclid's algorithm for computing the greatest common divisor (Section 1.1) provides a good example of this kind of algorithm. In this section, we encounter a few more examples of this variety.

Computing a Median and the Selection Problem

The ***selection problem*** is the problem of finding the kth smallest element in a list of n numbers. This number is called the kth ***order statistic***. Of course, for $k = 1$ or $k = n$, we can simply scan the list in question to find the smallest or largest element, respectively. A more interesting case of this problem is for $k = \lceil n/2 \rceil$, which asks to find an element that is greater than one half of the list's elements and smaller than the other half. This middle value is called the ***median***, and it is one of the most important quantities in mathematical statistics. Obviously, we can find the kth smallest element in a list by sorting the list first and then selecting the kth element in the output of a sorting algorithm. The time of such an algorithm is determined by the efficiency of the sorting algorithm used. Thus, with a good sorting algorithm such as mergesort, the algorithm's efficiency is in $O(n \log n)$.

You should immediately suspect, however, that sorting the entire list is most likely overkill since the problem asks not to order the entire list but just to find its kth smallest element. Fortunately, we do have a very efficient (on average) algorithm for doing a similar task of partitioning an array's elements into two subsets: the one containing the elements that are less than or equal to some value p pivoting the partition and the other containing the elements that are greater than or equal to p:

$$\underbrace{a_{i_1} \ldots a_{i_{s-1}}}_{\leq p} \quad p \quad \underbrace{a_{i_{s+1}} \ldots a_{i_n}}_{\geq p}.$$

Such partitioning was the principal part of quicksort, discussed in Chapter 4.

How can we take advantage of a list's partition? Let s be the partition's split position. If $s = k$, the pivot p obviously solves the selection problem. (Had we indexed the list starting at 0, it would have been $s = k - 1$, of course.) If $s > k$, the kth smallest element in the entire list can be found as the kth smallest element in the left part of the partitioned array. And if $s < k$, we can proceed by searching for the $(k - s)$th smallest element in its right part. Thus, if we do not solve the problem outright, we reduce its instance to a smaller one, which can be solved by the same approach, i.e., recursively. In fact, the same idea can be implemented without recursion as well. For the nonrecursive version, we need not even adjust the value of k but just continue until $s = k$.

https://hemanthrajhemu.github.io

EXAMPLE Find the median of the following list of nine numbers: 4, 1, 10, 9, 7, 12, 8, 2, 15. Here, $k = \lceil 9/2 \rceil = 5$ and our task is to find the fifth smallest element in the array. As earlier, we assume for the sake of convenience that the elements of the list are indexed from 1 to 9.

We use the same version of array partitioning that we used in our discussion of quicksort in Chapter 4, which selects the first element as a pivot and rearranges elements by two oppositely directed scans of the array:

4	1	10	9	7	12	8	2	15
2	1	**4**	9	7	12	8	10	15

Since $s = 3 < k = 5$, we proceed with the right part of the list:

9	7	12	8	10	15
8	7	**9**	12	10	15

Since $s = 6 > k = 5$, we continue with the left part of the previous sublist:

8	7
7	**8**

Now $s = k = 5$, and hence we can stop: the found median is 8, which is greater than 2, 1, 4, and 7 but smaller than 9, 12, 10, and 15. ▩

How efficient is this algorithm? We should expect it to be more efficient than quicksort in the average case because it has to deal with just a single subarray after a partition while quicksort has to work on two of them. Had the splits always happened in the middle of a remaining array, the recurrence for the number of comparisons would have been

$$C(n) = C(n/2) + (n + 1),$$

whose solution, according to the Master Theorem (see Chapter 4), is in $\Theta(n)$. Although the array's size is actually reduced in an unpredictable fashion from one iteration of the algorithm to another (with some size reductions less than half and some larger), a careful mathematical analysis shows the average-case efficiency to be the same as it would be had the size always been reduced by one half. In other words, the algorithm turns out to be linear in the average case. In the worst case, we have the same embarrassing deterioration of efficiency into $\Theta(n^2)$. Though computer scientists have discovered an algorithm that works in linear time even in the worst case [Blo73], it is too complicated to be recommended for practical applications.

Note also that the partitioning-based algorithm solves a somewhat more general problem of identifying the k smallest and $n - k$ largest elements of a given list, not just the value of its kth smallest element.

https://hemanthrajhemu.github.io

Interpolation Search

As the next example of a variable-size-decrease algorithm, we consider an algorithm for searching in a sorted array called ***interpolation search***. Unlike binary search, which always compares a search key with the middle value of a given sorted array (and hence reduces the problem's instance size by half), interpolation search takes into account the value of the search key in order to find the array's element to be compared with the search key. In a sense, the algorithm mimics the way we search for a name in a telephone book: if we are searching for someone named Brown, we open the book not in the middle but very close to the beginning, unlike our action when searching for someone named, say, Smith.

More precisely, on the iteration dealing with the array's portion between the leftmost element $A[l]$ and the rightmost element $A[r]$, the algorithm assumes that the array's values increase linearly, i.e., along the straight line through the points $(l, A[l])$ and $(r, A[r])$. (The accuracy of this assumption can influence the algorithm's efficiency but not its correctness.) Accordingly, the search key's value v is compared with the element whose index is computed as (the roundoff of) the x coordinate of the point on the straight line through the points $(l, A[l])$ and $(r, A[r])$ whose y coordinate is equal to the search value v (Figure 5.16).

Writing a standard equation for the straight line passing though the points $(l, A[l])$ and $(r, A[r])$, substituting v for y, and solving it for x leads to the following formula:

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor. \quad \textbf{(5.4)}$$

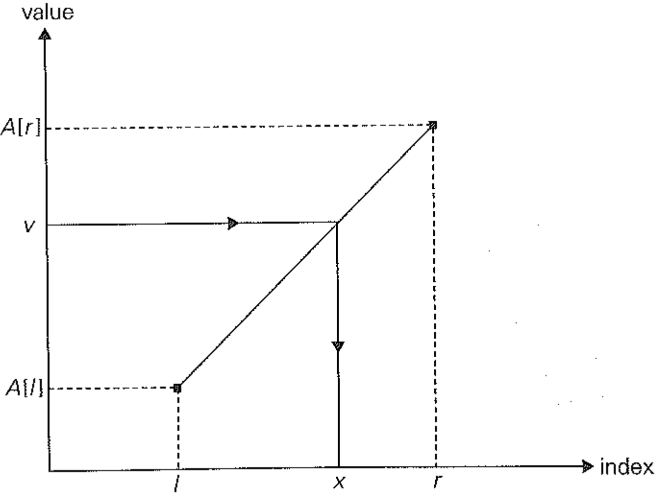


FIGURE 5.16 Index computation in interpolation search

https://hemanthrajhemu.github.io

The logic behind this approach is quite straightforward. We know that the array values are increasing (more accurately, not decreasing) from $A[l]$ to $A[r]$, but we do not know how they do it. Had the array's values increased linearly, which is the simplest manner possible, the index computed by formula (5.4) would be the expected location of the array's element with the value equal to v. Of course, if v is not between $A[l]$ and $A[r]$, formula (5.4) need not be applied (why?).

After comparing v with $A[x]$, the algorithm either stops (if they are equal) or proceeds by searching in the same manner among the elements indexed either between l and $x - 1$ or between $x + 1$ and r, depending on whether $A[x]$ is smaller or larger than v. Thus, the size of the problem's instance is reduced, but we cannot tell a priori by how much.

The analysis of the algorithm's efficiency shows that interpolation search uses fewer than $\log_2 \log_2 n + 1$ key comparisons on the average when searching in a list of n random keys. This function grows so slowly that the number of comparisons will be a very small constant for all practically feasible inputs (see Problem 6). But in the worst case, interpolation search is only linear, which must be considered as a bad performance (why?). As a final assessment of the worthiness of interpolation search versus that of binary search, we can point to an opinion by R. Sedgewick [Sed88] that binary search is probably better for smaller files but interpolation search is worth considering for large files and for applications where comparisons are particularly expensive or access costs are very high. Note that in Section 12.4, we discuss a continuous counterpart of interpolation search, which can be seen as one more example of a variable-size-decrease algorithm.

Searching and Insertion in a Binary Search Tree

As the last example of this section, let us revisit the binary search tree. Recall that this is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that for every node all elements in the left subtree are smaller and all the elements in the right subtree are greater than the element in the subtree's root. When we need to search for an element of a given value (say, v) in such a tree, we do it recursively in the following manner. If the tree is empty, the search ends in failure. If the tree is not empty, we compare v with the tree's root $K(r)$. If they match, a desired element is found and the search can be stopped; if they do not match, we continue with the search in the left subtree of the root if $v < K(r)$ and in the right subtree if $v > K(r)$. Thus, on each iteration of the algorithm, the problem of searching in a binary search tree is reduced to searching in a smaller binary search tree. The most sensible measure of size of a search tree is its height; obviously, the decrease in a tree's height normally changes from one iteration to another of the binary tree search—thus giving us an excellent example of a variable-size-decrease algorithm.

In the worst case of the binary tree search, the tree is severely skewed. This happens, in particular, if a tree is constructed by successive insertions of an increasing or decreasing sequence of keys (Figure 5.17).

https://hemanthrajhemu.github.io

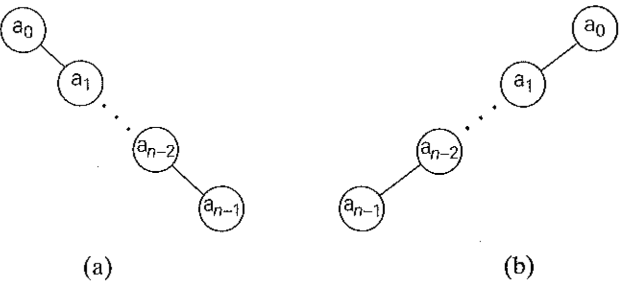


FIGURE 5.17 Binary search trees for (a) an increasing sequence of keys and (b) a decreasing sequence of keys

Obviously, the search for a_{n-1} in such a tree requires n comparisons, making the worst-case efficiency of the search operation fall into $\Theta(n)$. Fortunately, the average-case efficiency turns out to be in $\Theta(\log n)$. More precisely, the number of key comparisons needed for a search in a binary search tree built from n random keys is about $2\ln n \approx 1.39 \log_2 n$. Since the operation of insertion of a new key into a binary search tree is almost identical to that of searching there, it also exemplifies the variable-size-decrease technique and has the same efficiency characteristics as the search operation.

The Game of Nim

There are several well-known games that share the following features. There are two players, who move in turn. No randomness or hidden information is permitted: all players know all information about gameplay. A game is impartial: each player has the same moves available from the same game position. Each of a finite number of available moves leads to a smaller instance of the same game. The game ends with a win by one of the players (there are no ties). The winner is the last player who is able to move.

A prototypical example of such games is ***Nim***. Generally, the game is played with several piles of chips, but we consider the one-pile version first. Thus, there is a single pile of n chips. Two players take turns by removing from the pile at least one and at most m chips; the number of chips taken may vary from one move to another, but both the lower and upper limits stay the same. Who wins the game by taking the last chip, the player moving first or second, if both players make the best moves possible?

Let us call an instance of the game a winning position for the player to move next if that player has a winning strategy, that is, a sequence of moves that results in a victory no matter what moves the opponent makes. Let us call an instance of the game a losing position for the player to move next if every move available for that player leads to a winning position for the opponent. The standard approach to determining which positions are winning and which are losing is to

https://hemanthrajhemu.github.io

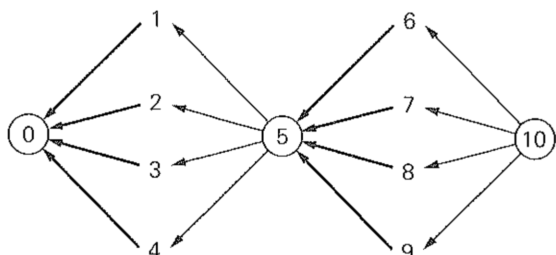


FIGURE 5.18 Figure 5.18 Illustration of one-pile Nim with the maximum number of chips that may be taken on each move $m = 4$. The numbers indicate n, the number of chips in the pile. The losing positions for the player to move are circled. Only winning moves from the winning positions are shown (in bold).

investigate small values of n first. It is logical to consider the instance of $n = 0$ as a losing one for the player to move next because this player is the first one who cannot make a move. Any instance with $1 \le n \le m$ chips is obviously a winning position for the player to move next (why?). The instance with $n = m + 1$ chips is a losing one because taking any allowed number of chips puts the opponent in a winning position. (See an illustration for $m = 4$ in Figure 5.18.) Any instance with $m + 2 \le n \le 2m + 1$ chips is a winning position for the player to move next because there is a move that leaves the opponent with $m + 1$ chips, which is a losing position; $2m + 2 = 2(m + 1)$ chips is the next losing position, and so on. It is not difficult to see the pattern that can be formally proved by mathematical induction: an instance with n chips is a winning position for the player to move next if and only if n is not a multiple of $m + 1$. The winning strategy is to take $n \bmod (m + 1)$ chips on every move; any deviation from this strategy puts the opponent in a winning position.

One-pile Nim has been known for a very long time. It appeared, in particular, as the ***summation game*** in the first published book on recreational mathematics, authored by Claude-Gaspar Bachet, a French aristocrat and mathematician, in 1612: a player picks a positive integer less than, say, ten, and then his opponent and he take turns adding any integer less than ten; the first player to reach 100 exactly is the winner [Dud70].

In general, Nim is played with $I > 1$ piles of chips of sizes $n_1, n_2, \ldots, n_I$. On each move, a player can take any available number of chips, including all of them, from any single pile. The goal is the same—to be the last player able to make a move. Note that for $I = 2$, it is easy to figure out who wins this game and how. Here is a hint: the answer for the game's instances with $n_1 = n_2$ differs from the answer for those with $n_1 \ne n_2$.

A solution to the general case of Nim is quite unexpected because it is based on the binary representation of the pile sizes. Let $b_1, b_2, \ldots, b_I$ be the pile sizes

https://hemanthrajhemu.github.io

in binary. Compute their ***binary digital sum***, defined as the sum of binary digits discarding any carry. (In other words, a binary digit s_i in the sum is 0 if the number of 1's in the ith position in the addends is even, and it is 1 if the number of 1's is odd.) It turns out that an instance of Nim is a winning one (for the player to move next) if and only if its binary digital sum contains at least one 1; consequently, Nim's instance is a loosing instance if and only if its binary digit sum contains only zeros. For example, for the commonly played instance with $n_1 = 3$, $n_2 = 4$, $n_3 = 5$, the binary digital sum is

$$\begin{array}{r} 011 \\ 100 \\ \underline{101} \\ 010 \end{array}$$

Since this sum contains a 1, the instance is a winning one for the payer moving first. To find a winning move from this position, the player needs to change one of the three bit strings so that the new binary digital sum contains only 0's. It is not difficult to see that the only way to accomplish this is to remove two chips from the first pile.

This ingenious solution to the game of Nim was discovered by Harvard mathematics professor C. L. Bouton more than 100 years ago. Since then, mathematicians have developed a much more general theory of such games. An excellent account of this theory, with applications to many specific games, is given in the monograph by E. R. Berlekamp, J. H. Conway, and R. K. Guy [Ber03].

Exercises 5.6

1. **a.** If we measure the size of an instance of the problem of computing the greatest common divisor of m and n by the size of the second parameter n, by how much can the size decrease after one iteration of Euclid's algorithm?

   b. Prove that the size of an instance will always decrease at least by a factor of 2 after two successive iterations of Euclid's algorithm.

2. **a.** Apply the partition-based algorithm to find the median of the list of numbers 9, 12, 5, 17, 20.

   b. Show that the worst-case efficiency of the partition-based algorithm for the selection problem is quadratic.

3. **a.** Write a pseudocode for a nonrecursive implementation of the partition-based algorithm for the selection problem.

   b. Write a pseudocode for a recursive implementation of this algorithm.

4. Derive the formula underlying interpolation search.

5. Give an example of the worst-case input for interpolation search and show that the algorithm is linear in the worst case.

https://hemanthrajhemu.github.io

6. a. Find the smallest value of n for which $\log_2 \log_2 n + 1$ is greater than 6.
   b. Determine which, if any, of the following assertions are true:
      i. $\log \log n \in o(\log n)$ ii. $\log \log n \in \Theta(\log n)$
      iii. $\log \log n \in \Omega(\log n)$
7. a. Outline an algorithm for finding the largest key in a binary search tree. Would you classify your algorithm as a variable-size decrease algorithm?
   b. What is the time efficiency class of your algorithm in the worst case?
8. a. Outline an algorithm for deleting a key from a binary search tree. Would you classify this algorithm as a variable-size decrease algorithm?
   b. What is the time efficiency class of your algorithm in the worst case?

9. *Misere one-pile Nim* Consider the so-called ***misere version*** of the one-pile Nim, in which the player taking the last chip looses the game. All the other conditions of the game remain the same, i.e., the pile contains n chips and on each move a player takes at least one but no more than m chips. Identify the winning and loosing positions (for the player to move next) in this game.

10. a. *Moldy chocolate* Two players take turns by breaking an m-by-n chocolate bar, which has one spoiled 1-by-1 square. Each break must be a single straight line cutting all the way across the bar along the boundaries between the squares. After each break, the player who broke the bar last eats the piece that does not contain the spoiled square. The player left with the spoiled square looses the game. Is it better to go first or second in this game?
    b. Write an interactive program to play this game with the computer. Your program should make a winning move in a winning position and a random legitimate move in a losing position.

11. *Flipping pancakes* There are n pancakes all of different sizes that are stacked on top of each other. You are allowed to slip a flipper under one of the pancakes and flip over the whole sack above the flipper. The purpose is to arrange pancakes according to their size with the biggest at the bottom. (You can see a visualization of this puzzles on the *Interactive Mathematics Miscellany and Puzzles* site [Bog].) Design an algorithm for solving this puzzle.

SUMMARY

- *Decrease-and-conquer* is a general algorithm design technique, based on exploiting a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion).

https://hemanthrajhemu.github.io

- There are three major variations of decrease-and-conquer:
  - *decrease-by-a-constant*, most often by one (e.g., insertion sort);
  - *decrease-by-a-constant-factor*, most often by the factor of two (e.g., binary search);
  - *variable-size-decrease* (e.g., Euclid's algorithm).

- *Insertion sort* is a direct application of the decrease (by one)-and-conquer technique to the sorting problem. It is a $\Theta(n^2)$ algorithm both in the worst and average cases, but it is about twice as fast on average than in the worst case. The algorithm's notable advantage is a good performance on almost-sorted arrays.

- *Depth-first search* (*DFS*) and *breadth-first search* (*BFS*) are two principal graph traversal algorithms. By representing a graph in a form of a depth-first or breadth-first search forest, they help in the investigation of many important properties of the graph. Both algorithms have the same time efficiency: $\Theta(|V|^2)$ for the adjacency matrix representation and $\Theta(|V| + |E|)$ for the adjacency list representation.

- A *digraph* is a graph with directions on its edges. The *topological sorting problem* asks to list vertices of a digraph in an order such that for every edge of the digraph, the vertex it starts at is listed before the vertex it points to. This problem has a solution if and only if a digraph is a *dag* (*directed acyclic graph*), i.e., it has no directed cycles.

- There are two algorithms for solving the topological sorting problem. The first one is based on depth-first search; the second is based on the direct implementation of the decrease-by-one technique.

- Decrease-by-one technique is a natural approach to developing algorithms for generating elementary combinatorial objects. The most efficient type of such algorithms are minimal-change algorithms. However, the number of combinatorial objects grows so fast that even the best algorithms are of practical interest only for very small instances of such problems.

- Identifying a fake coin with a balance scale, *multiplication à la russe*, and the *Josephus problem* are examples of problems that can be solved by *decrease-by-a-constant-factor* algorithms. Two other and more important examples are binary search and exponentiation by squaring.

- For some algorithms based on the decrease-and-conquer technique, the size reduction varies from one iteration of the algorithm to another. Examples of such *variable-size-decrease* algorithms include Euclid's algorithm, the partition-based algorithm for the *selection problem*, *interpolation search*, and searching and insertion in a binary search tree. *Nim* exemplifies games that proceed through a series of diminishing instances of the same game.

https://hemanthrajhemu.github.io