

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>

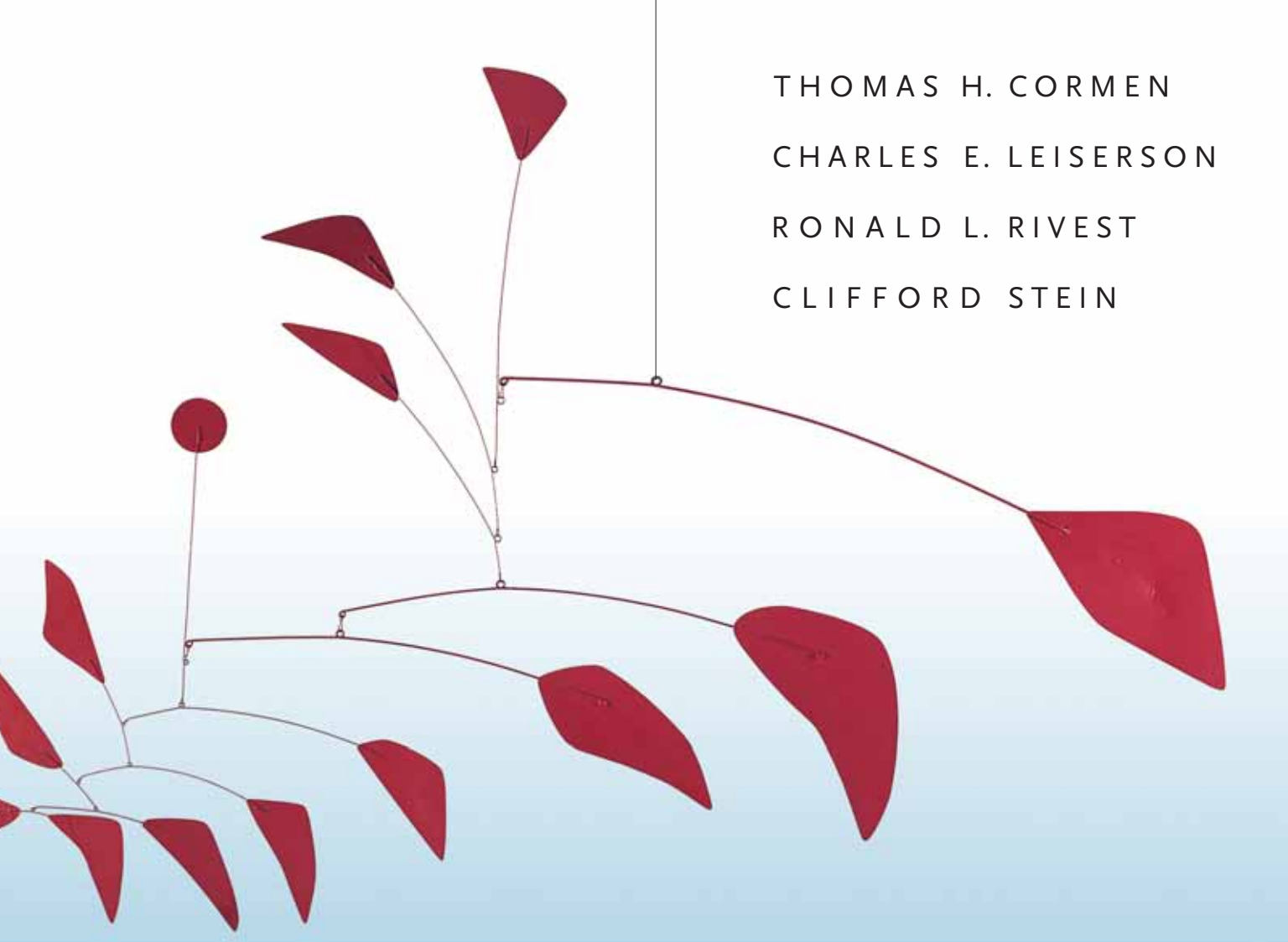
Gain Access to All Study Materials according to
VTU, Currently for CSE – Computer Science
Engineering...

Join Telegram to get Instant Updates: <https://bit.ly/2GKiHnJ>

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/



THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

INTRODUCTION TO

ALGORITHMS

<https://hemanthrajhenu.github.io>

THIRD EDITION

- 12 Binary Search Trees 286**
 - 12.1 What is a binary search tree? 286
 - 12.2 Querying a binary search tree 289
 - 12.3 Insertion and deletion 294
 - ★ 12.4 Randomly built binary search trees 299
- 13 Red-Black Trees 308**
 - 13.1 Properties of red-black trees 308
 - 13.2 Rotations 312
 - 13.3 Insertion 315
 - 13.4 Deletion 323
- 14 Augmenting Data Structures 339**
 - 14.1 Dynamic order statistics 339
 - 14.2 How to augment a data structure 345
 - 14.3 Interval trees 348

IV Advanced Design and Analysis Techniques

- Introduction 357**
- 15 Dynamic Programming 359**
 - 15.1 Rod cutting 360
 - 15.2 Matrix-chain multiplication 370
 - 15.3 Elements of dynamic programming 378
 - 15.4 Longest common subsequence 390
 - 15.5 Optimal binary search trees 397
- 16 Greedy Algorithms 414**
 - 16.1 An activity-selection problem 415
 - 16.2 Elements of the greedy strategy 423
 - 16.3 Huffman codes 428
 - ★ 16.4 Matroids and greedy methods 437
 - ★ 16.5 A task-scheduling problem as a matroid 443
- 17 Amortized Analysis 451**
 - 17.1 Aggregate analysis 452
 - 17.2 The accounting method 456
 - 17.3 The potential method 459
 - 17.4 Dynamic tables 463

- 30 Polynomials and the FFT 898**
 - 30.1 Representing polynomials 900
 - 30.2 The DFT and FFT 906
 - 30.3 Efficient FFT implementations 915
- 31 Number-Theoretic Algorithms 926**
 - 31.1 Elementary number-theoretic notions 927
 - 31.2 Greatest common divisor 933
 - 31.3 Modular arithmetic 939
 - 31.4 Solving modular linear equations 946
 - 31.5 The Chinese remainder theorem 950
 - 31.6 Powers of an element 954
 - 31.7 The RSA public-key cryptosystem 958
 - ★ 31.8 Primality testing 965
 - ★ 31.9 Integer factorization 975
- 32 String Matching 985**
 - 32.1 The naive string-matching algorithm 988
 - 32.2 The Rabin-Karp algorithm 990
 - 32.3 String matching with finite automata 995
 - ★ 32.4 The Knuth-Morris-Pratt algorithm 1002
- 33 Computational Geometry 1014**
 - 33.1 Line-segment properties 1015
 - 33.2 Determining whether any pair of segments intersects 1021
 - 33.3 Finding the convex hull 1029
 - 33.4 Finding the closest pair of points 1039
- 34 NP-Completeness 1048**
 - 34.1 Polynomial time 1053
 - 34.2 Polynomial-time verification 1061
 - 34.3 NP-completeness and reducibility 1067
 - 34.4 NP-completeness proofs 1078
 - 34.5 NP-complete problems 1086
- 35 Approximation Algorithms 1106**
 - 35.1 The vertex-cover problem 1108
 - 35.2 The traveling-salesman problem 1111
 - 35.3 The set-covering problem 1117
 - 35.4 Randomization and linear programming 1123
 - 35.5 The subset-sum problem 1128

16 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A *greedy algorithm* always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 15, particularly Section 15.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We shall first examine, in Section 16.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We shall arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that we can always make greedy choices to arrive at an optimal solution. Section 16.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 16.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. In Section 16.4, we investigate some of the theory underlying combinatorial structures called “matroids,” for which a greedy algorithm always produces an optimal solution. Finally, Section 16.5 applies matroids to solve a problem of scheduling unit-time tasks with deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that we can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 23), Dijkstra’s algorithm for shortest paths from a single source (Chapter 24), and Chvátal’s greedy set-covering heuristic (Chapter 35). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read

this chapter and Chapter 23 independently of each other, you might find it useful to read them together.

16.1 An activity-selection problem

Our first example is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Suppose we have a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed *activities* that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a *start time* s_i and a *finish time* f_i , where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are *compatible* if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. In the *activity-selection problem*, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \dots \leq f_{n-1} \leq f_n. \quad (16.1)$$

(We shall see later the advantage that this assumption provides.) For example, consider the following set S of activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We shall solve this problem in several steps. We start by thinking about a dynamic-programming solution, in which we consider several choices when determining which subproblems to use in an optimal solution. We shall then observe that we need to consider only one choice—the greedy choice—and that when we make the greedy choice, only one subproblem remains. Based on these observations, we shall develop a recursive greedy algorithm to solve the activity-scheduling problem. We shall complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. Although the steps we shall go through in this section are slightly more involved than is typical when developing a greedy algorithm, they illustrate the relationship between greedy algorithms and dynamic programming.

The optimal substructure of the activity-selection problem

We can easily verify that the activity-selection problem exhibits optimal substructure. Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij} , and suppose further that such a maximum set is A_{ij} , which includes some activity a_k . By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts). Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj} . If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then we could use A'_{kj} , rather than A_{kj} , in a solution to the subproblem for S_{ij} . We would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik} .

This way of characterizing optimal substructure suggests that we might solve the activity-selection problem by dynamic programming. If we denote the size of an optimal solution for the set S_{ij} by $c[i, j]$, then we would have the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

Of course, if we did not know that an optimal solution for the set S_{ij} includes activity a_k , we would have to examine all activities in S_{ij} to find which one to choose, so that

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset . \end{cases} \quad (16.2)$$

We could then develop a recursive algorithm and memoize it, or we could work bottom-up and fill in table entries as we go along. But we would be overlooking another important characteristic of the activity-selection problem that we can use to great advantage.

Making the greedy choice

What if we could choose an activity to add to our optimal solution without having to first solve all the subproblems? That could save us from having to consider all the choices inherent in recurrence (16.2). In fact, for the activity-selection problem, we need consider only one choice: the greedy choice.

What do we mean by the greedy choice for the activity-selection problem? Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible. Now, of the activities we end up choosing, one of them must be the first one to finish. Our intuition tells us, therefore, to choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible. (If more than one activity in S has the earliest finish time, then we can choose any such activity.) In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1 . Choosing the first activity to finish is not the only way to think of making a greedy choice for this problem; Exercise 16.1-3 asks you to explore other possibilities.

If we make the greedy choice, we have only one remaining subproblem to solve: finding activities that start after a_1 finishes. Why don't we have to consider activities that finish before a_1 starts? We have that $s_1 < f_1$, and f_1 is the earliest finish time of any activity, and therefore no activity can have a finish time less than or equal to s_1 . Thus, all activities that are compatible with activity a_1 must start after a_1 finishes.

Furthermore, we have already established that the activity-selection problem exhibits optimal substructure. Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If we make the greedy choice of activity a_1 , then S_1 remains as the only subproblem to solve.¹ Optimal substructure tells us that if a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1 .

One big question remains: is our intuition correct? Is the greedy choice—in which we choose the first activity to finish—always part of some optimal solution? The following theorem shows that it is.

¹We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. It will always be clear from the context whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem 16.1

Consider any nonempty subproblem S_k , and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k .

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k , and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k . If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j . The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k , and it includes a_m . ■

Thus, we see that although we might be able to solve the activity-selection problem with dynamic programming, we don't need to. (Besides, we have not yet examined whether the activity-selection problem even has overlapping subproblems.) Instead, we can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain. Moreover, because we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase. We can consider each activity just once overall, in monotonically increasing order of finish times.

An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm. Instead, it can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen. Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Now that we have seen how to bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, we can write a straightforward, recursive procedure to solve the activity-selection problem. The procedure RECURSIVE-ACTIVITY-SELECTOR takes the start and finish times of the activities, represented as arrays s and f ,² the index k that defines the subproblem S_k it is to solve, and

²Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather than subscripts.

the size n of the original problem. It returns a maximum-size set of mutually compatible activities in S_k . We assume that the n input activities are already ordered by monotonically increasing finish time, according to equation (16.1). If not, we can sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily. In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S . The initial call, which solves the entire problem, is `RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$)`.

`RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)`

```

1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$       // find the first activity in  $S_k$  to finish
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6  else return  $\emptyset$ 
```

Figure 16.1 shows the operation of the algorithm. In a given recursive call `RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)`, the **while** loop of lines 2–3 looks for the first activity in S_k to finish. The loop examines $a_{k+1}, a_{k+2}, \dots, a_n$, until it finds the first activity a_m that is compatible with a_k ; such an activity has $s_m \geq f_k$. If the loop terminates because it finds such an activity, line 5 returns the union of $\{a_m\}$ and the maximum-size subset of S_m returned by the recursive call `RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)`. Alternatively, the loop may terminate because $m > n$, in which case we have examined all activities in S_k without finding one that is compatible with a_k . In this case, $S_k = \emptyset$, and so the procedure returns \emptyset in line 6.

Assuming that the activities have already been sorted by finish times, the running time of the call `RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$)` is $\Theta(n)$, which we can see as follows. Over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular, activity a_i is examined in the last call made in which $k < i$.

An iterative greedy algorithm

We easily can convert our recursive procedure to an iterative one. The procedure `RECURSIVE-ACTIVITY-SELECTOR` is almost “tail recursive” (see Problem 7-4): it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form; in fact, some compilers for certain programming languages perform this task automatically. As written, `RECURSIVE-ACTIVITY-SELECTOR` works for subproblems S_k , i.e., subproblems that consist of the last activities to finish.

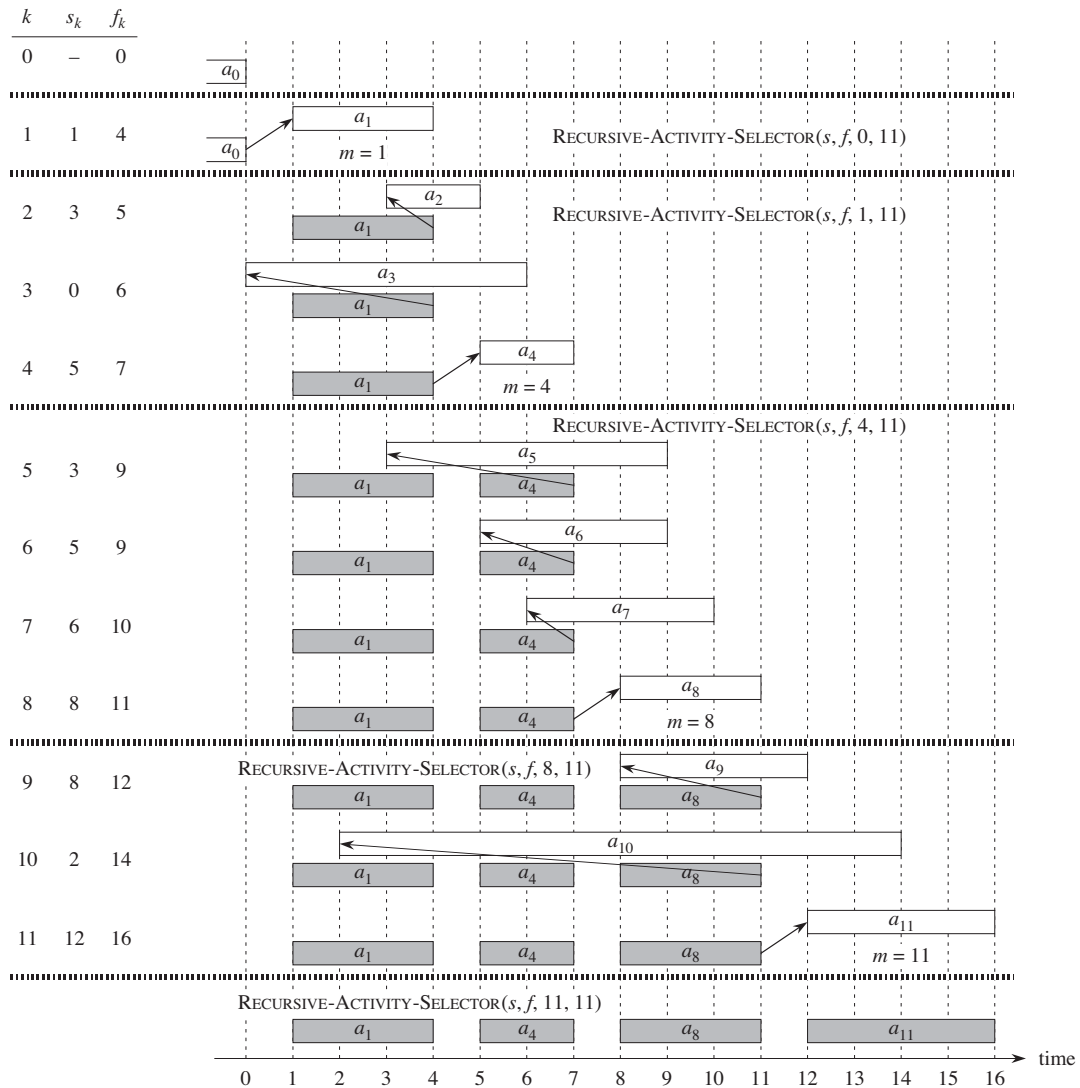


Figure 16.1 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities given earlier. Activities considered in each recursive call appear between horizontal lines. The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, 11$), selects activity a_1 . In each recursive call, the activities that have already been selected are shaded, and the activity shown in white is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, RECURSIVE-ACTIVITY-SELECTOR($s, f, 11, 11$), returns \emptyset . The resulting set of selected activities is $\{a_1, a_4, a_8, a_{11}\}$.

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR. It also assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set A and returns this set when it is done.

GREEDY-ACTIVITY-SELECTOR(s, f)

```

1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 

```

The procedure works as follows. The variable k indexes the most recent addition to A , corresponding to the activity a_k in the recursive version. Since we consider the activities in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A . That is,

$$f_k = \max \{f_i : a_i \in A\} . \quad (16.3)$$

Lines 2–3 select activity a_1 , initialize A to contain just this activity, and initialize k to index this activity. The **for** loop of lines 4–7 finds the earliest activity in S_k to finish. The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities; such an activity is the earliest in S_k to finish. To see whether activity a_m is compatible with every activity currently in A , it suffices by equation (16.3) to check (in line 5) that its start time s_m is not earlier than the finish time f_k of the activity most recently added to A . If activity a_m is compatible, then lines 6–7 add activity a_m to A and set k to m . The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the call RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$).

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Exercises

16.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.

Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

16.1-2

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

16.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

16.1-4

Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the *interval-graph coloring problem*. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

16.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i . The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

16.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in Section 16.1 to develop a greedy algorithm was a bit more involved than is typical. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (16.2), but we bypassed developing a recursive algorithm based on this recurrence.)
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

In going through these steps, we saw in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, in the activity-selection problem, we first defined the subproblems S_{ij} , where both i and j varied. We then found that if we always made the greedy choice, we could restrict the subproblems to be of the form S_k .

Alternatively, we could have fashioned our optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, we could have started by dropping the second subscript and defining subproblems of the form S_k . Then, we could have proven that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k . More generally, we design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

We shall use this more direct process in later sections of this chapter. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

How can we tell whether a greedy algorithm will solve a particular optimization problem? No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients. If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the *greedy-choice property*: we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. (Alternatively, we can solve them top down, but memoizing. Of course, even though the code works top down, we still must solve the subproblems before making a choice.) In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, we must prove that a greedy choice at each step yields a globally optimal solution. Typically, as in the case of Theorem 16.1, the proof examines a globally optimal solution to some subproblem. It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

We can usually make the greedy choice more efficiently than when we have to consider a wider set of choices. For example, in the activity-selection problem, as-

suming that we had already sorted the activities in monotonically increasing order of finish times, we needed to examine each activity just once. By preprocessing the input or by using an appropriate data structure (often a priority queue), we often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. As an example of optimal substructure, recall how we demonstrated in Section 16.1 that if an optimal solution to subproblem S_{ij} includes an activity a_k , then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj} . Given this optimal substructure, we argued that if we knew which activity to use as a_k , we could construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the subproblems S_{ik} and S_{kj} . Based on this observation of optimal substructure, we were able to devise the recurrence (16.2) that described the value of an optimal solution.

We usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, we have the luxury of assuming that we arrived at a subproblem by having made the greedy choice in the original problem. All we really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

The *0-1 knapsack problem* is the following. A thief robbing a store finds n items. The i th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W . Which items should he take? (We call this the 0-1 knapsack problem because for each item, the thief must either

take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

In the *fractional knapsack problem*, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j . For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j .

Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy. To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W . Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. We leave the proof that the fractional knapsack problem has the greedy-choice property as Exercise 16.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 16.2(a). This example has 3 items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is 6 dollars per pound, which is greater than the value per pound of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 16.2(b), however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two possible solutions that take item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 16.2(c). Taking item 1 doesn't work in the 0-1 problem because the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the

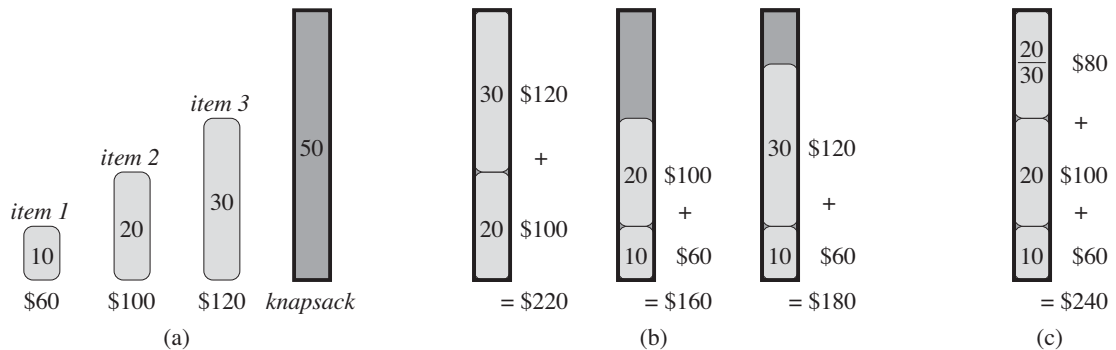


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

choice. The problem formulated in this way gives rise to many overlapping sub-problems—a hallmark of dynamic programming, and indeed, as Exercise 16.2-2 asks you to show, we can use dynamic programming to solve the 0-1 problem.

Exercises

16.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

16.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

16.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

16.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana.

The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations.

The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

16.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \dots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

16.2-6 ★

Show how to solve the fractional knapsack problem in $O(n)$ time.

16.2-7

Suppose you are given two sets A and B , each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i th element of set A , and let b_i be the i th element of set B . You then receive a payoff of $\prod_{i=1}^n a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

16.3 Huffman codes

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by Figure 16.3. That is, only 6 different characters appear, and the character `a` occurs 45,000 times.

We have many options for how to represent such a file of information. Here, we consider the problem of designing a *binary character code* (or *code* for short)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

in which each character is represented by a unique binary string, which we call a *codeword*. If we use a *fixed-length code*, we need 3 bits to represent 6 characters: a = 000, b = 001, . . . , f = 101. This method requires 300,000 bits to code the entire file. Can we do better?

A *variable-length code* can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Figure 16.3 shows such a code; here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called *prefix codes*.³ Although we won’t prove it here, a prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.

Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code of Figure 16.3, we code the 3-character file abc as $0 \cdot 101 \cdot 100 = 0101100$, where “ \cdot ” denotes concatenation.

Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original char-

³Perhaps “prefix-free codes” would be a better name, but the term “prefix codes” is standard in the literature.

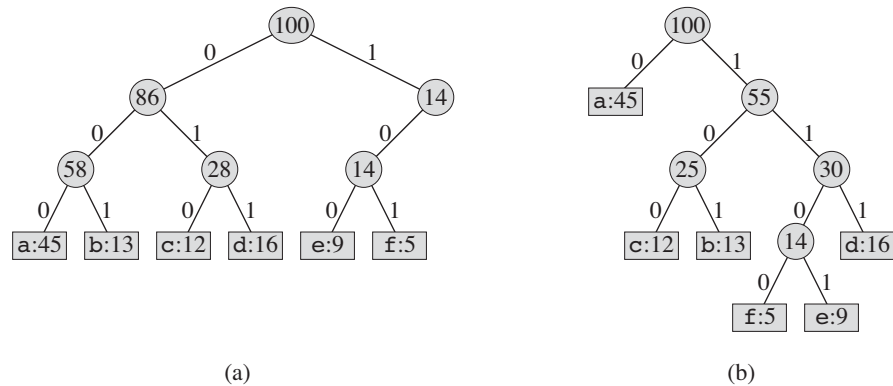


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code $a = 000, \dots, f = 101$. **(b)** The tree corresponding to the optimal prefix code $a = 0, b = 101, \dots, f = 1100$.

acter, and repeat the decoding process on the remainder of the encoded file. In our example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to **aabe**.

The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child.” Figure 16.4 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 16.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 16.4(a), is not a full binary tree: it contains codewords beginning 10..., but none beginning 11.... Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3).

Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file. For each character c in the alphabet C , let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth

of c 's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c . The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.\text{freq} \cdot d_T(c), \quad (16.4)$$

which we define as the *cost* of the tree T .

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a *Huffman code*. In line with our observations in Section 16.2, its proof of correctness relies on the greedy-choice property and optimal substructure. Rather than demonstrating that these properties hold and then developing pseudocode, we present the pseudocode first. Doing so will help clarify how the algorithm makes greedy choices.

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.\text{freq}$ giving its frequency. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ “merging” operations to create the final tree. The algorithm uses a min-priority queue Q , keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

HUFFMAN(C)

```

1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.\text{left} = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.\text{right} = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.\text{freq} = x.\text{freq} + y.\text{freq}$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // return the root of the tree
```

For our example, Huffman’s algorithm proceeds as shown in Figure 16.5. Since the alphabet contains 6 letters, the initial queue size is $n = 6$, and 5 merge steps build the tree. The final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

Line 2 initializes the min-priority queue Q with the characters in C . The **for** loop in lines 3–8 repeatedly extracts the two nodes x and y of lowest frequency

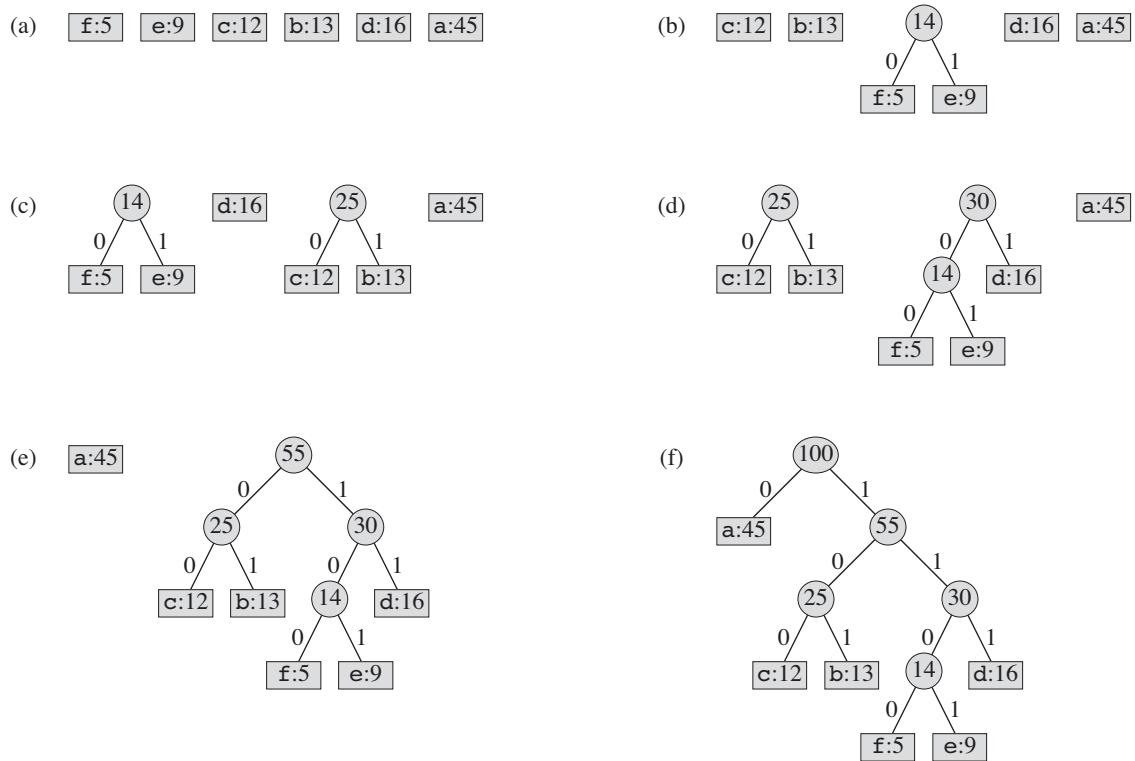


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. (a) The initial set of $n = 6$ nodes, one for each letter. (b)–(e) Intermediate stages. (f) The final tree.

from the queue, replacing them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 7. The node z has x as its left child and y as its right child. (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, line 9 returns the one node left in the queue, which is the root of the code tree.

Although the algorithm would produce the same result if we were to excise the variables x and y —assigning directly to $z.left$ and $z.right$ in lines 5 and 6, and changing line 7 to $z.freq = z.left.freq + z.right.freq$ —we shall use the node

names x and y in the proof of correctness. Therefore, we find it convenient to leave them in.

To analyze the running time of Huffman's algorithm, we assume that Q is implemented as a binary min-heap (see Chapter 6). For a set C of n characters, we can initialize Q in line 2 in $O(n)$ time using the BUILD-MIN-HEAP procedure discussed in Section 6.3. The **for** loop in lines 3–8 executes exactly $n - 1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$. We can reduce the running time to $O(n \lg \lg n)$ by replacing the binary min-heap with a van Emde Boas tree (see Chapter 20).

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.

Let a and b be two characters that are sibling leaves of maximum depth in T . Without loss of generality, we assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \leq a.freq$ and $y.freq \leq b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$. However, if we had $x.freq = b.freq$, then we would also have $a.freq = b.freq = x.freq = y.freq$ (see Exercise 16.3-1), and the lemma would be trivially true. Thus, we will assume that $x.freq \neq b.freq$, which means that $x \neq b$.

As Figure 16.6 shows, we exchange the positions in T of a and x to produce a tree T' , and then we exchange the positions in T' of b and y to produce a tree T''

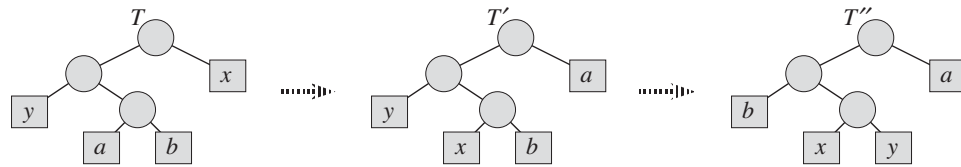


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T , leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T . Assuming that $x \neq b$, swapping leaves a and x produces tree T' , and then swapping leaves b and y produces tree T'' . Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (16.4), the difference in cost between T and T' is

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

because both $a.\text{freq} - x.\text{freq}$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.\text{freq} - x.\text{freq}$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T . Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Lemma 16.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 16.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix codes has the optimal-substructure property.

Lemma 16.3

Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C , except that $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C' . Then the tree T , obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C .

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T' , by considering the component costs in equation (16.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

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

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

or, equivalently,

$$B(T') = B(T) - x.freq - y.freq.$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix code for C . Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 16.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then

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

yielding a contradiction to the assumption that T' represents an optimal prefix code for C' . Thus, T must represent an optimal prefix code for the alphabet C . ■

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

Proof Immediate from Lemmas 16.2 and 16.3. ■

Exercises**16.3-1**

Explain why, in the proof of Lemma 16.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

16.3-2

Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

16.3-3

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

16.3-4

Prove that we can also express the total cost of a tree for a code as the sum, over all internal nodes, of the combined frequencies of the two children of the node.

16.3-5

Prove that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

16.3-6

Suppose we have an optimal prefix code on a set $C = \{0, 1, \dots, n-1\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

16.3-7

Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

16.3-8

Suppose that a data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

16.3-9

Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit. (*Hint:* Compare the number of possible files with the number of possible encoded files.)

★ 16.4 Matroids and greedy methods

In this section, we sketch a beautiful theory about greedy algorithms. This theory describes many situations in which the greedy method yields optimal solutions. It involves combinatorial structures known as “matroids.” Although this theory does not cover all cases for which a greedy method applies (for example, it does not cover the activity-selection problem of Section 16.1 or the Huffman-coding problem of Section 16.3), it does cover many cases of practical interest. Furthermore, this theory has been extended to cover many applications; see the notes at the end of this chapter for references.

Matroids

A *matroid* is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions.

1. S is a finite set.
2. \mathcal{I} is a nonempty family of subsets of S , called the *independent* subsets of S , such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. We say that \mathcal{I} is *hereditary* if it satisfies this property. Note that the empty set \emptyset is necessarily a member of \mathcal{I} .
3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there exists some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the *exchange property*.

The word “matroid” is due to Hassler Whitney. He was studying *matric matroids*, in which the elements of S are the rows of a given matrix and a set of rows is independent if they are linearly independent in the usual sense. As Exercise 16.4-2 asks you to show, this structure defines a matroid.

As another example of matroids, consider the *graphic matroid* $M_G = (S_G, \mathcal{I}_G)$ defined in terms of a given undirected graph $G = (V, E)$ as follows:

- The set S_G is defined to be E , the set of edges of G .
- If A is a subset of E , then $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

The graphic matroid M_G is closely related to the minimum-spanning-tree problem, which Chapter 23 covers in detail.

Theorem 16.5

If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

Proof Clearly, $S_G = E$ is a finite set. Furthermore, \mathcal{I}_G is hereditary, since a subset of a forest is a forest. Putting it another way, removing edges from an acyclic set of edges cannot create cycles.

Thus, it remains to show that M_G satisfies the exchange property. Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$. That is, A and B are acyclic sets of edges, and B contains more edges than A does.

We claim that a forest $F = (V_F, E_F)$ contains exactly $|V_F| - |E_F|$ trees. To see why, suppose that F consists of t trees, where the i th tree contains v_i vertices and e_i edges. Then, we have

$$\begin{aligned} |E_F| &= \sum_{i=1}^t e_i \\ &= \sum_{i=1}^t (v_i - 1) \quad (\text{by Theorem B.2}) \\ &= \sum_{i=1}^t v_i - t \\ &= |V_F| - t, \end{aligned}$$

which implies that $t = |V_F| - |E_F|$. Thus, forest G_A contains $|V| - |A|$ trees, and forest G_B contains $|V| - |B|$ trees.

Since forest G_B has fewer trees than forest G_A does, forest G_B must contain some tree T whose vertices are in two different trees in forest G_A . Moreover, since T is connected, it must contain an edge (u, v) such that vertices u and v are in different trees in forest G_A . Since the edge (u, v) connects vertices in two different trees in forest G_A , we can add the edge (u, v) to forest G_A without creating a cycle. Therefore, M_G satisfies the exchange property, completing the proof that M_G is a matroid. ■

Given a matroid $M = (S, \mathcal{I})$, we call an element $x \notin A$ an *extension* of $A \in \mathcal{I}$ if we can add x to A while preserving independence; that is, x is an extension of A if $A \cup \{x\} \in \mathcal{I}$. As an example, consider a graphic matroid M_G . If A is an independent set of edges, then edge e is an extension of A if and only if e is not in A and the addition of e to A does not create a cycle.

If A is an independent subset in a matroid M , we say that A is *maximal* if it has no extensions. That is, A is maximal if it is not contained in any larger independent subset of M . The following property is often useful.

Theorem 16.6

All maximal independent subsets in a matroid have the same size.

Proof Suppose to the contrary that A is a maximal independent subset of M and there exists another larger maximal independent subset B of M . Then, the exchange property implies that for some $x \in B - A$, we can extend A to a larger independent set $A \cup \{x\}$, contradicting the assumption that A is maximal. ■

As an illustration of this theorem, consider a graphic matroid M_G for a connected, undirected graph G . Every maximal independent subset of M_G must be a free tree with exactly $|V| - 1$ edges that connects all the vertices of G . Such a tree is called a *spanning tree* of G .

We say that a matroid $M = (S, \mathcal{I})$ is *weighted* if it is associated with a weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$. The weight function w extends to subsets of S by summation:

$$w(A) = \sum_{x \in A} w(x)$$

for any $A \subseteq S$. For example, if we let $w(e)$ denote the weight of an edge e in a graphic matroid M_G , then $w(A)$ is the total weight of the edges in edge set A .

Greedy algorithms on a weighted matroid

Many problems for which a greedy approach provides optimal solutions can be formulated in terms of finding a maximum-weight independent subset in a weighted matroid. That is, we are given a weighted matroid $M = (S, \mathcal{I})$, and we wish to find an independent set $A \in \mathcal{I}$ such that $w(A)$ is maximized. We call such a subset that is independent and has maximum possible weight an *optimal* subset of the matroid. Because the weight $w(x)$ of any element $x \in S$ is positive, an optimal subset is always a maximal independent subset—it always helps to make A as large as possible.

For example, in the *minimum-spanning-tree problem*, we are given a connected undirected graph $G = (V, E)$ and a length function w such that $w(e)$ is the (positive) length of edge e . (We use the term “length” here to refer to the original edge weights for the graph, reserving the term “weight” to refer to the weights in the associated matroid.) We wish to find a subset of the edges that connects all of the vertices together and has minimum total length. To view this as a problem of finding an optimal subset of a matroid, consider the weighted matroid M_G with weight function w' , where $w'(e) = w_0 - w(e)$ and w_0 is larger than the maximum length of any edge. In this weighted matroid, all weights are positive and an optimal subset is a spanning tree of minimum total length in the original graph. More specifically, each maximal independent subset A corresponds to a spanning tree

with $|V| - 1$ edges, and since

$$\begin{aligned}
 w'(A) &= \sum_{e \in A} w'(e) \\
 &= \sum_{e \in A} (w_0 - w(e)) \\
 &= (|V| - 1)w_0 - \sum_{e \in A} w(e) \\
 &= (|V| - 1)w_0 - w(A)
 \end{aligned}$$

for any maximal independent subset A , an independent subset that maximizes the quantity $w'(A)$ must minimize $w(A)$. Thus, any algorithm that can find an optimal subset A in an arbitrary matroid can solve the minimum-spanning-tree problem.

Chapter 23 gives algorithms for the minimum-spanning-tree problem, but here we give a greedy algorithm that works for any weighted matroid. The algorithm takes as input a weighted matroid $M = (S, \mathcal{I})$ with an associated positive weight function w , and it returns an optimal subset A . In our pseudocode, we denote the components of M by $M.S$ and $M.\mathcal{I}$ and the weight function by w . The algorithm is greedy because it considers in turn each element $x \in S$, in order of monotonically decreasing weight, and immediately adds it to the set A being accumulated if $A \cup \{x\}$ is independent.

GREEDY(M, w)

```

1   $A = \emptyset$ 
2  sort  $M.S$  into monotonically decreasing order by weight  $w$ 
3  for each  $x \in M.S$ , taken in monotonically decreasing order by weight  $w(x)$ 
4      if  $A \cup \{x\} \in M.\mathcal{I}$ 
5           $A = A \cup \{x\}$ 
6  return  $A$ 

```

Line 4 checks whether adding each element x to A would maintain A as an independent set. If A would remain independent, then line 5 adds x to A . Otherwise, x is discarded. Since the empty set is independent, and since each iteration of the **for** loop maintains A 's independence, the subset A is always independent, by induction. Therefore, GREEDY always returns an independent subset A . We shall see in a moment that A is a subset of maximum possible weight, so that A is an optimal subset.

The running time of GREEDY is easy to analyze. Let n denote $|S|$. The sorting phase of GREEDY takes time $O(n \lg n)$. Line 4 executes exactly n times, once for each element of S . Each execution of line 4 requires a check on whether or not the set $A \cup \{x\}$ is independent. If each such check takes time $O(f(n))$, the entire algorithm runs in time $O(n \lg n + nf(n))$.

We now prove that GREEDY returns an optimal subset.

Lemma 16.7 (Matroids exhibit the greedy-choice property)

Suppose that $M = (S, \mathcal{I})$ is a weighted matroid with weight function w and that S is sorted into monotonically decreasing order by weight. Let x be the first element of S such that $\{x\}$ is independent, if any such x exists. If x exists, then there exists an optimal subset A of S that contains x .

Proof If no such x exists, then the only independent subset is the empty set and the lemma is vacuously true. Otherwise, let B be any nonempty optimal subset. Assume that $x \notin B$; otherwise, letting $A = B$ gives an optimal subset of S that contains x .

No element of B has weight greater than $w(x)$. To see why, observe that $y \in B$ implies that $\{y\}$ is independent, since $B \in \mathcal{I}$ and \mathcal{I} is hereditary. Our choice of x therefore ensures that $w(x) \geq w(y)$ for any $y \in B$.

Construct the set A as follows. Begin with $A = \{x\}$. By the choice of x , set A is independent. Using the exchange property, repeatedly find a new element of B that we can add to A until $|A| = |B|$, while preserving the independence of A . At that point, A and B are the same except that A has x and B has some other element y . That is, $A = B - \{y\} \cup \{x\}$ for some $y \in B$, and so

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B). \end{aligned}$$

Because set B is optimal, set A , which contains x , must also be optimal. ■

We next show that if an element is not an option initially, then it cannot be an option later.

Lemma 16.8

Let $M = (S, \mathcal{I})$ be any matroid. If x is an element of S that is an extension of some independent subset A of S , then x is also an extension of \emptyset .

Proof Since x is an extension of A , we have that $A \cup \{x\}$ is independent. Since \mathcal{I} is hereditary, $\{x\}$ must be independent. Thus, x is an extension of \emptyset . ■

Corollary 16.9

Let $M = (S, \mathcal{I})$ be any matroid. If x is an element of S such that x is not an extension of \emptyset , then x is not an extension of any independent subset A of S .

Proof This corollary is simply the contrapositive of Lemma 16.8. ■

Corollary 16.9 says that any element that cannot be used immediately can never be used. Therefore, GREEDY cannot make an error by passing over any initial elements in S that are not an extension of \emptyset , since they can never be used.

Lemma 16.10 (Matroids exhibit the optimal-substructure property)

Let x be the first element of S chosen by GREEDY for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathcal{I}')$, where

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \mathcal{I}\}, \\ \mathcal{I}' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\}, \end{aligned}$$

and the weight function for M' is the weight function for M , restricted to S' . (We call M' the *contraction* of M by the element x .)

Proof If A is any maximum-weight independent subset of M containing x , then $A' = A - \{x\}$ is an independent subset of M' . Conversely, any independent subset A' of M' yields an independent subset $A = A' \cup \{x\}$ of M . Since we have in both cases that $w(A) = w(A') + w(x)$, a maximum-weight solution in M containing x yields a maximum-weight solution in M' , and vice versa. ■

Theorem 16.11 (Correctness of the greedy algorithm on matroids)

If $M = (S, \mathcal{I})$ is a weighted matroid with weight function w , then GREEDY(M, w) returns an optimal subset.

Proof By Corollary 16.9, any elements that GREEDY passes over initially because they are not extensions of \emptyset can be forgotten about, since they can never be useful. Once GREEDY selects the first element x , Lemma 16.7 implies that the algorithm does not err by adding x to A , since there exists an optimal subset containing x . Finally, Lemma 16.10 implies that the remaining problem is one of finding an optimal subset in the matroid M' that is the contraction of M by x . After the procedure GREEDY sets A to $\{x\}$, we can interpret all of its remaining steps as acting in the matroid $M' = (S', \mathcal{I}')$, because B is independent in M' if and only if $B \cup \{x\}$ is independent in M , for all sets $B \in \mathcal{I}'$. Thus, the subsequent operation of GREEDY will find a maximum-weight independent subset for M' , and the overall operation of GREEDY will find a maximum-weight independent subset for M . ■

Exercises**16.4-1**

Show that (S, \mathcal{I}_k) is a matroid, where S is any finite set and \mathcal{I}_k is the set of all subsets of S of size at most k , where $k \leq |S|$.

16.4-2 ★

Given an $m \times n$ matrix T over some field (such as the reals), show that (S, \mathcal{I}) is a matroid, where S is the set of columns of T and $A \in \mathcal{I}$ if and only if the columns in A are linearly independent.

16.4-3 ★

Show that if (S, \mathcal{I}) is a matroid, then (S, \mathcal{I}') is a matroid, where

$$\mathcal{I}' = \{A' : S - A' \text{ contains some maximal } A \in \mathcal{I}\} .$$

That is, the maximal independent sets of (S, \mathcal{I}') are just the complements of the maximal independent sets of (S, \mathcal{I}) .

16.4-4 ★

Let S be a finite set and let S_1, S_2, \dots, S_k be a partition of S into nonempty disjoint subsets. Define the structure (S, \mathcal{I}) by the condition that $\mathcal{I} = \{A : |A \cap S_i| \leq 1 \text{ for } i = 1, 2, \dots, k\}$. Show that (S, \mathcal{I}) is a matroid. That is, the set of all sets A that contain at most one member of each subset in the partition determines the independent sets of a matroid.

16.4-5

Show how to transform the weight function of a weighted matroid problem, where the desired optimal solution is a *minimum-weight* maximal independent subset, to make it a standard weighted-matroid problem. Argue carefully that your transformation is correct.

★ 16.5 A task-scheduling problem as a matroid

An interesting problem that we can solve using matroids is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a deadline, along with a penalty paid if the task misses its deadline. The problem looks complicated, but we can solve it in a surprisingly simple manner by casting it as a matroid and using a greedy algorithm.

A *unit-time task* is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a

schedule for S is a permutation of S specifying the order in which to perform these tasks. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

The problem of *scheduling unit-time tasks with deadlines and penalties for a single processor* has the following inputs:

- a set $S = \{a_1, a_2, \dots, a_n\}$ of n unit-time tasks;
- a set of n integer *deadlines* d_1, d_2, \dots, d_n , such that each d_i satisfies $1 \leq d_i \leq n$ and task a_i is supposed to finish by time d_i ; and
- a set of n nonnegative weights or *penalties* w_1, w_2, \dots, w_n , such that we incur a penalty of w_i if task a_i is not finished by time d_i , and we incur no penalty if a task finishes by its deadline.

We wish to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

Consider a given schedule. We say that a task is *late* in this schedule if it finishes after its deadline. Otherwise, the task is *early* in the schedule. We can always transform an arbitrary schedule into *early-first form*, in which the early tasks precede the late tasks. To see why, note that if some early task a_i follows some late task a_j , then we can switch the positions of a_i and a_j , and a_i will still be early and a_j will still be late.

Furthermore, we claim that we can always transform an arbitrary schedule into *canonical form*, in which the early tasks precede the late tasks and we schedule the early tasks in order of monotonically increasing deadlines. To do so, we put the schedule into early-first form. Then, as long as there exist two early tasks a_i and a_j finishing at respective times k and $k + 1$ in the schedule such that $d_j < d_i$, we swap the positions of a_i and a_j . Since a_j is early before the swap, $k + 1 \leq d_j$. Therefore, $k + 1 < d_i$, and so a_i is still early after the swap. Because task a_j is moved earlier in the schedule, it remains early after the swap.

The search for an optimal schedule thus reduces to finding a set A of tasks that we assign to be early in the optimal schedule. Having determined A , we can create the actual schedule by listing the elements of A in order of monotonically increasing deadlines, then listing the late tasks (i.e., $S - A$) in any order, producing a canonical ordering of the optimal schedule.

We say that a set A of tasks is *independent* if there exists a schedule for these tasks such that no tasks are late. Clearly, the set of early tasks for a schedule forms an independent set of tasks. Let \mathcal{I} denote the set of all independent sets of tasks.

Consider the problem of determining whether a given set A of tasks is independent. For $t = 0, 1, 2, \dots, n$, let $N_t(A)$ denote the number of tasks in A whose deadline is t or earlier. Note that $N_0(A) = 0$ for any set A .

Lemma 16.12

For any set of tasks A , the following statements are equivalent.

1. The set A is independent.
2. For $t = 0, 1, 2, \dots, n$, we have $N_t(A) \leq t$.
3. If the tasks in A are scheduled in order of monotonically increasing deadlines, then no task is late.

Proof To show that (1) implies (2), we prove the contrapositive: if $N_t(A) > t$ for some t , then there is no way to make a schedule with no late tasks for set A , because more than t tasks must finish before time t . Therefore, (1) implies (2). If (2) holds, then (3) must follow: there is no way to “get stuck” when scheduling the tasks in order of monotonically increasing deadlines, since (2) implies that the i th largest deadline is at least i . Finally, (3) trivially implies (1). ■

Using property 2 of Lemma 16.12, we can easily compute whether or not a given set of tasks is independent (see Exercise 16.5-2).

The problem of minimizing the sum of the penalties of the late tasks is the same as the problem of maximizing the sum of the penalties of the early tasks. The following theorem thus ensures that we can use the greedy algorithm to find an independent set A of tasks with the maximum total penalty.

Theorem 16.13

If S is a set of unit-time tasks with deadlines, and \mathcal{I} is the set of all independent sets of tasks, then the corresponding system (S, \mathcal{I}) is a matroid.

Proof Every subset of an independent set of tasks is certainly independent. To prove the exchange property, suppose that B and A are independent sets of tasks and that $|B| > |A|$. Let k be the largest t such that $N_t(B) \leq N_t(A)$. (Such a value of t exists, since $N_0(A) = N_0(B) = 0$.) Since $N_n(B) = |B|$ and $N_n(A) = |A|$, but $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$ for all j in the range $k + 1 \leq j \leq n$. Therefore, B contains more tasks with deadline $k + 1$ than A does. Let a_i be a task in $B - A$ with deadline $k + 1$. Let $A' = A \cup \{a_i\}$.

We now show that A' must be independent by using property 2 of Lemma 16.12. For $0 \leq t \leq k$, we have $N_t(A') = N_t(A) \leq t$, since A is independent. For $k < t \leq n$, we have $N_t(A') \leq N_t(B) \leq t$, since B is independent. Therefore, A' is independent, completing our proof that (S, \mathcal{I}) is a matroid. ■

By Theorem 16.11, we can use a greedy algorithm to find a maximum-weight independent set of tasks A . We can then create an optimal schedule having the tasks in A as its early tasks. This method is an efficient algorithm for scheduling

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 16.7 An instance of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor.

unit-time tasks with deadlines and penalties for a single processor. The running time is $O(n^2)$ using GREEDY, since each of the $O(n)$ independence checks made by that algorithm takes time $O(n)$ (see Exercise 16.5-2). Problem 16-4 gives a faster implementation.

Figure 16.7 demonstrates an example of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor. In this example, the greedy algorithm selects, in order, tasks a_1 , a_2 , a_3 , and a_4 , then rejects a_5 (because $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$) and a_6 (because $N_4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$), and finally accepts a_7 . The final optimal schedule is

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle,$$

which has a total penalty incurred of $w_5 + w_6 = 50$.

Exercises

16.5-1

Solve the instance of the scheduling problem given in Figure 16.7, but with each penalty w_i replaced by $80 - w_i$.

16.5-2

Show how to use property 2 of Lemma 16.12 to determine in time $O(|A|)$ whether or not a given set A of tasks is independent.

Problems

16-1 Coin changing

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

- Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

- b. Suppose that the available coins are in the denominations that are powers of c , i.e., the denominations are c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.
- c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n .
- d. Give an $O(nk)$ -time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

16-2 Scheduling to minimize average completion time

Suppose you are given a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the **completion time** of task a_i , that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n) \sum_{i=1}^n c_i$. For example, suppose there are two tasks, a_1 and a_2 , with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1 . Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$. If task a_1 runs first, however, then $c_1 = 3$, $c_2 = 8$, and the average completion time is $(3 + 8)/2 = 5.5$.

- a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.
- b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its **release time** r_i . Suppose also that we allow **preemption**, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $r_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. In this scenario, a_i 's completion time is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

16-3 Acyclic subgraphs

- a. The **incidence matrix** for an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix M such that $M_{ve} = 1$ if edge e is incident on vertex v , and $M_{ve} = 0$ otherwise. Argue that a set of columns of M is linearly independent over the field of integers modulo 2 if and only if the corresponding set of edges is acyclic. Then, use the result of Exercise 16.4-2 to provide an alternate proof that (E, \mathcal{I}) of part (a) is a matroid.
- b. Suppose that we associate a nonnegative weight $w(e)$ with each edge in an undirected graph $G = (V, E)$. Give an efficient algorithm to find an acyclic subset of E of maximum total weight.
- c. Let $G(V, E)$ be an arbitrary directed graph, and let (E, \mathcal{I}) be defined so that $A \in \mathcal{I}$ if and only if A does not contain any directed cycles. Give an example of a directed graph G such that the associated system (E, \mathcal{I}) is not a matroid. Specify which defining condition for a matroid fails to hold.
- d. The **incidence matrix** for a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix M such that $M_{ve} = -1$ if edge e leaves vertex v , $M_{ve} = 1$ if edge e enters vertex v , and $M_{ve} = 0$ otherwise. Argue that if a set of columns of M is linearly independent, then the corresponding set of edges does not contain a directed cycle.
- e. Exercise 16.4-2 tells us that the set of linearly independent sets of columns of any matrix M forms a matroid. Explain carefully why the results of parts (d) and (e) are not contradictory. How can there fail to be a perfect correspondence between the notion of a set of edges being acyclic and the notion of the associated set of columns of the incidence matrix being linearly independent?

16-4 Scheduling variations

Consider the following algorithm for the problem from Section 16.5 of scheduling unit-time tasks with deadlines and penalties. Let all n time slots be initially empty, where time slot i is the unit-length slot of time that finishes at time i . We consider the tasks in order of monotonically decreasing penalty. When considering task a_j , if there exists a time slot at or before a_j 's deadline d_j that is still empty, assign a_j to the latest such slot, filling it. If there is no such slot, assign task a_j to the latest of the as yet unfilled slots.

- a. Argue that this algorithm always gives an optimal answer.
- b. Use the fast disjoint-set forest presented in Section 21.3 to implement the algorithm efficiently. Assume that the set of input tasks has already been sorted into

monotonically decreasing order by penalty. Analyze the running time of your implementation.

16-5 Off-line caching

Modern computers use a cache to store a small amount of data in a fast memory. Even though a program may access large amounts of data, by storing a small subset of the main memory in the *cache*—a small but faster memory—overall access time can greatly decrease. When a computer program executes, it makes a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of n memory requests, where each request is for a particular data element. For example, a program that accesses 4 distinct elements $\{a, b, c, d\}$ might make the sequence of requests $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Let k be the size of the cache. When the cache contains k elements and the program requests the $(k + 1)$ st element, the system must decide, for this and each subsequent request, which k elements to keep in the cache. More precisely, for each request r_i , the cache-management algorithm checks whether element r_i is already in the cache. If it is, then we have a *cache hit*; otherwise, we have a *cache miss*. Upon a cache miss, the system retrieves r_i from the main memory, and the cache-management algorithm must decide whether to keep r_i in the cache. If it decides to keep r_i and the cache already holds k elements, then it must evict one element to make room for r_i . The cache-management algorithm evicts data with the goal of minimizing the number of cache misses over the entire sequence of requests.

Typically, caching is an on-line problem. That is, we have to make decisions about which data to keep in the cache without knowing the future requests. Here, however, we consider the off-line version of this problem, in which we are given in advance the entire sequence of n requests and the cache size k , and we wish to minimize the total number of cache misses.

We can solve this off-line problem by a greedy strategy called *furthest-in-future*, which chooses to evict the item in the cache whose next access in the request sequence comes furthest in the future.

- a. Write pseudocode for a cache manager that uses the furthest-in-future strategy. The input should be a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of requests and a cache size k , and the output should be a sequence of decisions about which data element (if any) to evict upon each request. What is the running time of your algorithm?
- b. Show that the off-line caching problem exhibits optimal substructure.
- c. Prove that furthest-in-future produces the minimum possible number of cache misses.

Chapter notes

Much more material on greedy algorithms and matroids can be found in Lawler [224] and Papadimitriou and Steiglitz [271].

The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [101], though the theory of matroids dates back to a 1935 article by Whitney [355].

Our proof of the correctness of the greedy algorithm for the activity-selection problem is based on that of Gavril [131]. The task-scheduling problem is studied in Lawler [224]; Horowitz, Sahni, and Rajasekaran [181]; and Brassard and Bratley [54].

Huffman codes were invented in 1952 [185]; Lelewer and Hirschberg [231] surveys data-compression techniques known as of 1987.

An extension of matroid theory to greedoid theory was pioneered by Korte and Lovász [216, 217, 218, 219], who greatly generalize the theory presented here.

The straightforward method of adding two polynomials of degree n takes $\Theta(n)$ time, but the straightforward method of multiplying them takes $\Theta(n^2)$ time. In this chapter, we shall show how the fast Fourier transform, or FFT, can reduce the time to multiply polynomials to $\Theta(n \lg n)$.

The most common use for Fourier transforms, and hence the FFT, is in signal processing. A signal is given in the *time domain*: as a function mapping time to amplitude. Fourier analysis allows us to express the signal as a weighted sum of phase-shifted sinusoids of varying frequencies. The weights and phases associated with the frequencies characterize the signal in the *frequency domain*. Among the many everyday applications of FFT's are compression techniques used to encode digital video and audio information, including MP3 files. Several fine books delve into the rich area of signal processing; the chapter notes reference a few of them.

Polynomials

A *polynomial* in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum:

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

We call the values a_0, a_1, \dots, a_{n-1} the *coefficients* of the polynomial. The coefficients are drawn from a field F , typically the set \mathbb{C} of complex numbers. A polynomial $A(x)$ has *degree* k if its highest nonzero coefficient is a_k ; we write that $\text{degree}(A) = k$. Any integer strictly greater than the degree of a polynomial is a *degree-bound* of that polynomial. Therefore, the degree of a polynomial of degree-bound n may be any integer between 0 and $n - 1$, inclusive.

We can define a variety of operations on polynomials. For *polynomial addition*, if $A(x)$ and $B(x)$ are polynomials of degree-bound n , their *sum* is a poly-

mial $C(x)$, also of degree-bound n , such that $C(x) = A(x) + B(x)$ for all x in the underlying field. That is, if

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

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j,$$

where $c_j = a_j + b_j$ for $j = 0, 1, \dots, n-1$. For example, if we have the polynomials $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$, then $C(x) = 4x^3 + 7x^2 - 6x + 4$.

For **polynomial multiplication**, if $A(x)$ and $B(x)$ are polynomials of degree-bound n , their **product** $C(x)$ is a polynomial of degree-bound $2n-1$ such that $C(x) = A(x)B(x)$ for all x in the underlying field. You probably have multiplied polynomials before, by multiplying each term in $A(x)$ by each term in $B(x)$ and then combining terms with equal powers. For example, we can multiply $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$ as follows:

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

Another way to express the product $C(x)$ is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j, \tag{30.1}$$

where

$$c_j = \sum_{k=0}^j a_k b_{j-k}. \tag{30.2}$$

Note that $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$, implying that if A is a polynomial of degree-bound n_a and B is a polynomial of degree-bound n_b , then C is a polynomial of degree-bound $n_a + n_b - 1$. Since a polynomial of degree-bound k is also a polynomial of degree-bound $k + 1$, we will normally say that the product polynomial C is a polynomial of degree-bound $n_a + n_b$.

Chapter outline

Section 30.1 presents two ways to represent polynomials: the coefficient representation and the point-value representation. The straightforward methods for multiplying polynomials—equations (30.1) and (30.2)—take $\Theta(n^2)$ time when we represent polynomials in coefficient form, but only $\Theta(n)$ time when we represent them in point-value form. We can, however, multiply polynomials using the coefficient representation in only $\Theta(n \lg n)$ time by converting between the two representations. To see why this approach works, we must first study complex roots of unity, which we do in Section 30.2. Then, we use the FFT and its inverse, also described in Section 30.2, to perform the conversions. Section 30.3 shows how to implement the FFT quickly in both serial and parallel models.

This chapter uses complex numbers extensively, and within this chapter we use the symbol i exclusively to denote $\sqrt{-1}$.

30.1 Representing polynomials

The coefficient and point-value representations of polynomials are in a sense equivalent; that is, a polynomial in point-value form has a unique counterpart in coefficient form. In this section, we introduce the two representations and show how to combine them so that we can multiply two degree-bound n polynomials in $\Theta(n \lg n)$ time.

Coefficient representation

A *coefficient representation* of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n is a vector of coefficients $a = (a_0, a_1, \dots, a_{n-1})$. In matrix equations in this chapter, we shall generally treat vectors as column vectors.

The coefficient representation is convenient for certain operations on polynomials. For example, the operation of *evaluating* the polynomial $A(x)$ at a given point x_0 consists of computing the value of $A(x_0)$. We can evaluate a polynomial in $\Theta(n)$ time using *Horner's rule*:

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

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

Now, consider multiplying two degree-bound n polynomials $A(x)$ and $B(x)$ represented in coefficient form. If we use the method described by equations (30.1) and (30.2), multiplying polynomials takes time $\Theta(n^2)$, since we must multiply each coefficient in the vector a by each coefficient in the vector b . The operation of multiplying polynomials in coefficient form seems to be considerably more difficult than that of evaluating a polynomial or adding two polynomials. The resulting coefficient vector c , given by equation (30.2), is also called the **convolution** of the input vectors a and b , denoted $c = a \otimes b$. Since multiplying polynomials and computing convolutions are fundamental computational problems of considerable practical importance, this chapter concentrates on efficient algorithms for them.

Point-value representation

A **point-value representation** of a polynomial $A(x)$ of degree-bound n is a set of n **point-value pairs**

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and

$$y_k = A(x_k) \tag{30.3}$$

for $k = 0, 1, \dots, n-1$. A polynomial has many different point-value representations, since we can use any set of n distinct points x_0, x_1, \dots, x_{n-1} as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straightforward, since all we have to do is select n distinct points x_0, x_1, \dots, x_{n-1} and then evaluate $A(x_k)$ for $k = 0, 1, \dots, n-1$. With Horner's method, evaluating a polynomial at n points takes time $\Theta(n^2)$. We shall see later that if we choose the points x_k cleverly, we can accelerate this computation to run in time $\Theta(n \lg n)$.

The inverse of evaluation—determining the coefficient form of a polynomial from a point-value representation—is **interpolation**. The following theorem shows that interpolation is well defined when the desired interpolating polynomial must have a degree-bound equal to the given number of point-value pairs.

Theorem 30.1 (Uniqueness of an interpolating polynomial)

For any set $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \dots, n-1$.

Proof The proof relies on the existence of the inverse of a certain matrix. Equation (30.3) is equivalent to the matrix equation

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \quad (30.4)$$

The matrix on the left is denoted $V(x_0, x_1, \dots, x_{n-1})$ and is known as a Vandermonde matrix. By Problem D-1, this matrix has determinant

$$\prod_{0 \leq j < k \leq n-1} (x_k - x_j),$$

and therefore, by Theorem D.5, it is invertible (that is, nonsingular) if the x_k are distinct. Thus, we can solve for the coefficients a_j uniquely given the point-value representation:

$$a = V(x_0, x_1, \dots, x_{n-1})^{-1} y. \quad \blacksquare$$

The proof of Theorem 30.1 describes an algorithm for interpolation based on solving the set (30.4) of linear equations. Using the LU decomposition algorithms of Chapter 28, we can solve these equations in time $O(n^3)$.

A faster algorithm for n -point interpolation is based on **Lagrange's formula**:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}. \quad (30.5)$$

You may wish to verify that the right-hand side of equation (30.5) is a polynomial of degree-bound n that satisfies $A(x_k) = y_k$ for all k . Exercise 30.1-5 asks you how to compute the coefficients of A using Lagrange's formula in time $\Theta(n^2)$.

Thus, n -point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation.¹ The algorithms described above for these problems take time $\Theta(n^2)$.

The point-value representation is quite convenient for many operations on polynomials. For addition, if $C(x) = A(x) + B(x)$, then $C(x_k) = A(x_k) + B(x_k)$ for any point x_k . More precisely, if we have a point-value representation for A ,

¹Interpolation is a notoriously tricky problem from the point of view of numerical stability. Although the approaches described here are mathematically correct, small differences in the inputs or round-off errors during computation can cause large differences in the result.

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\} ,$$

and for B ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{n-1}, y'_{n-1})\}$$

(note that A and B are evaluated at the *same* n points), then a point-value representation for C is

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \dots, (x_{n-1}, y_{n-1} + y'_{n-1})\} .$$

Thus, the time to add two polynomials of degree-bound n in point-value form is $\Theta(n)$.

Similarly, the point-value representation is convenient for multiplying polynomials. If $C(x) = A(x)B(x)$, then $C(x_k) = A(x_k)B(x_k)$ for any point x_k , and we can pointwise multiply a point-value representation for A by a point-value representation for B to obtain a point-value representation for C . We must face the problem, however, that $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$; if A and B are of degree-bound n , then C is of degree-bound $2n$. A standard point-value representation for A and B consists of n point-value pairs for each polynomial. When we multiply these together, we get n point-value pairs, but we need $2n$ pairs to interpolate a unique polynomial C of degree-bound $2n$. (See Exercise 30.1-4.) We must therefore begin with “extended” point-value representations for A and for B consisting of $2n$ point-value pairs each. Given an extended point-value representation for A ,

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{2n-1}, y_{2n-1})\} ,$$

and a corresponding extended point-value representation for B ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{2n-1}, y'_{2n-1})\} ,$$

then a point-value representation for C is

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \dots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\} .$$

Given two input polynomials in extended point-value form, we see that the time to multiply them to obtain the point-value form of the result is $\Theta(n)$, much less than the time required to multiply polynomials in coefficient form.

Finally, we consider how to evaluate a polynomial given in point-value form at a new point. For this problem, we know of no simpler approach than converting the polynomial to coefficient form first, and then evaluating it at the new point.

Fast multiplication of polynomials in coefficient form

Can we use the linear-time multiplication method for polynomials in point-value form to expedite polynomial multiplication in coefficient form? The answer hinges

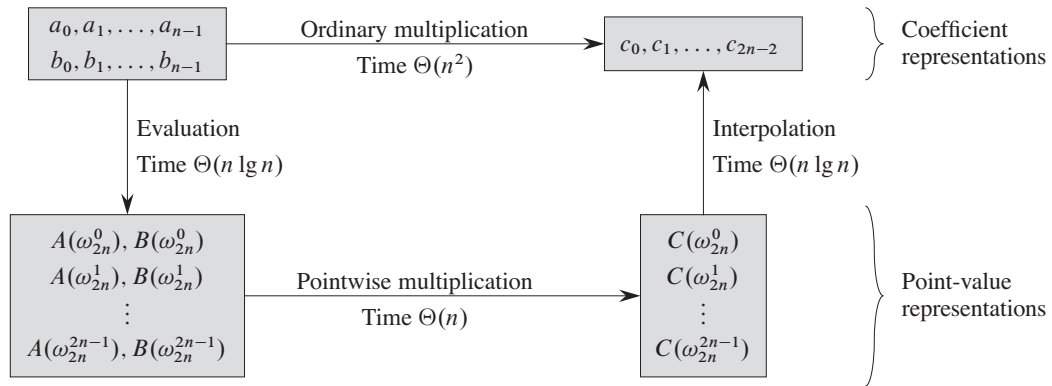


Figure 30.1 A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, while those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$ th roots of unity.

on whether we can convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice versa (interpolate).

We can use any points we want as evaluation points, but by choosing the evaluation points carefully, we can convert between representations in only $\Theta(n \lg n)$ time. As we shall see in Section 30.2, if we choose “complex roots of unity” as the evaluation points, we can produce a point-value representation by taking the discrete Fourier transform (or DFT) of a coefficient vector. We can perform the inverse operation, interpolation, by taking the “inverse DFT” of point-value pairs, yielding a coefficient vector. Section 30.2 will show how the FFT accomplishes the DFT and inverse DFT operations in $\Theta(n \lg n)$ time.

Figure 30.1 shows this strategy graphically. One minor detail concerns degree-bounds. The product of two polynomials of degree-bound n is a polynomial of degree-bound $2n$. Before evaluating the input polynomials A and B , therefore, we first double their degree-bounds to $2n$ by adding n high-order coefficients of 0. Because the vectors have $2n$ elements, we use “complex $(2n)$ th roots of unity,” which are denoted by the ω_{2n} terms in Figure 30.1.

Given the FFT, we have the following $\Theta(n \lg n)$ -time procedure for multiplying two polynomials $A(x)$ and $B(x)$ of degree-bound n , where the input and output representations are in coefficient form. We assume that n is a power of 2; we can always meet this requirement by adding high-order zero coefficients.

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each.

2. *Evaluate*: Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ by applying the FFT of order $2n$ on each polynomial. These representations contain the values of the two polynomials at the $(2n)$ th roots of unity.
3. *Pointwise multiply*: Compute a point-value representation for the polynomial $C(x) = A(x)B(x)$ by multiplying these values together pointwise. This representation contains the value of $C(x)$ at each $(2n)$ th root of unity.
4. *Interpolate*: Create the coefficient representation of the polynomial $C(x)$ by applying the FFT on $2n$ point-value pairs to compute the inverse DFT.

Steps (1) and (3) take time $\Theta(n)$, and steps (2) and (4) take time $\Theta(n \lg n)$. Thus, once we show how to use the FFT, we will have proven the following.

Theorem 30.2

We can multiply two polynomials of degree-bound n in time $\Theta(n \lg n)$, with both the input and output representations in coefficient form. ■

Exercises

30.1-1

Multiply the polynomials $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 6x + 3$ using equations (30.1) and (30.2).

30.1-2

Another way to evaluate a polynomial $A(x)$ of degree-bound n at a given point x_0 is to divide $A(x)$ by the polynomial $(x - x_0)$, obtaining a quotient polynomial $q(x)$ of degree-bound $n - 1$ and a remainder r , such that

$$A(x) = q(x)(x - x_0) + r .$$

Clearly, $A(x_0) = r$. Show how to compute the remainder r and the coefficients of $q(x)$ in time $\Theta(n)$ from x_0 and the coefficients of A .

30.1-3

Derive a point-value representation for $A^{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$ from a point-value representation for $A(x) = \sum_{j=0}^{n-1} a_jx^j$, assuming that none of the points is 0.

30.1-4

Prove that n distinct point-value pairs are necessary to uniquely specify a polynomial of degree-bound n , that is, if fewer than n distinct point-value pairs are given, they fail to specify a unique polynomial of degree-bound n . (*Hint*: Using Theorem 30.1, what can you say about a set of $n - 1$ point-value pairs to which you add one more arbitrarily chosen point-value pair?)

30.1-5

Show how to use equation (30.5) to interpolate in time $\Theta(n^2)$. (*Hint*: First compute the coefficient representation of the polynomial $\prod_j (x - x_j)$ and then divide by $(x - x_k)$ as necessary for the numerator of each term; see Exercise 30.1-2. You can compute each of the n denominators in time $O(n)$.)

30.1-6

Explain what is wrong with the “obvious” approach to polynomial division using a point-value representation, i.e., dividing the corresponding y values. Discuss separately the case in which the division comes out exactly and the case in which it doesn’t.

30.1-7

Consider two sets A and B , each having n integers in the range from 0 to $10n$. We wish to compute the *Cartesian sum* of A and B , defined by

$$C = \{x + y : x \in A \text{ and } y \in B\} .$$

Note that the integers in C are in the range from 0 to $20n$. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B . Show how to solve the problem in $O(n \lg n)$ time. (*Hint*: Represent A and B as polynomials of degree at most $10n$.)

30.2 The DFT and FFT

In Section 30.1, we claimed that if we use complex roots of unity, we can evaluate and interpolate polynomials in $\Theta(n \lg n)$ time. In this section, we define complex roots of unity and study their properties, define the DFT, and then show how the FFT computes the DFT and its inverse in $\Theta(n \lg n)$ time.

Complex roots of unity

A *complex n th root of unity* is a complex number ω such that

$$\omega^n = 1 .$$

There are exactly n complex n th roots of unity: $e^{2\pi ik/n}$ for $k = 0, 1, \dots, n - 1$. To interpret this formula, we use the definition of the exponential of a complex number:

$$e^{iu} = \cos(u) + i \sin(u) .$$

Figure 30.2 shows that the n complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane. The value

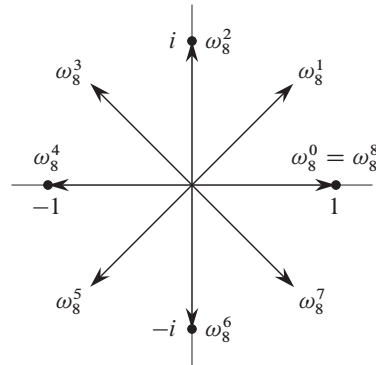


Figure 30.2 The values of $\omega_8^0, \omega_8^1, \dots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity.

$$\omega_n = e^{2\pi i/n} \quad (30.6)$$

is the *principal n th root of unity*;² all other complex n th roots of unity are powers of ω_n .

The n complex n th roots of unity,

$$\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1},$$

form a group under multiplication (see Section 31.3). This group has the same structure as the additive group $(\mathbb{Z}_n, +)$ modulo n , since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Similarly, $\omega_n^{-1} = \omega_n^{n-1}$. The following lemmas furnish some essential properties of the complex n th roots of unity.

Lemma 30.3 (Cancellation lemma)

For any integers $n \geq 0$, $k \geq 0$, and $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k. \quad (30.7)$$

Proof The lemma follows directly from equation (30.6), since

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

■

²Many other authors define ω_n differently: $\omega_n = e^{-2\pi i/n}$. This alternative definition tends to be used for signal-processing applications. The underlying mathematics is substantially the same with either definition of ω_n .

Corollary 30.4

For any even integer $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1.$$

Proof The proof is left as Exercise 30.2-1. ■

Lemma 30.5 (Halving lemma)

If $n > 0$ is even, then the squares of the n complex n th roots of unity are the $n/2$ complex $(n/2)$ th roots of unity.

Proof By the cancellation lemma, we have $(\omega_n^k)^2 = \omega_{n/2}^k$, for any nonnegative integer k . Note that if we square all of the complex n th roots of unity, then we obtain each $(n/2)$ th root of unity exactly twice, since

$$\begin{aligned} (\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k} \omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2. \end{aligned}$$

Thus, ω_n^k and $\omega_n^{k+n/2}$ have the same square. We could also have used Corollary 30.4 to prove this property, since $\omega_n^{n/2} = -1$ implies $\omega_n^{k+n/2} = -\omega_n^k$, and thus $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$. ■

As we shall see, the halving lemma is essential to our divide-and-conquer approach for converting between coefficient and point-value representations of polynomials, since it guarantees that the recursive subproblems are only half as large.

Lemma 30.6 (Summation lemma)

For any integer $n \geq 1$ and nonzero integer k not divisible by n ,

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

Proof Equation (A.5) applies to complex values as well as to reals, and so we have

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

Because we require that k is not divisible by n , and because $\omega_n^k = 1$ only when k is divisible by n , we ensure that the denominator is not 0. ■

The DFT

Recall that we wish to evaluate a polynomial

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

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$ (that is, at the n complex n th roots of unity).³ We assume that A is given in coefficient form: $a = (a_0, a_1, \dots, a_{n-1})$. Let us define the results y_k , for $k = 0, 1, \dots, n-1$, by

$$\begin{aligned}
y_k &= A(\omega_n^k) \\
&= \sum_{j=0}^{n-1} a_j \omega_n^{kj}.
\end{aligned} \tag{30.8}$$

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

The FFT

By using a method known as the **fast Fourier transform (FFT)**, which takes advantage of the special properties of the complex roots of unity, we can compute $\text{DFT}_n(a)$ in time $\Theta(n \lg n)$, as opposed to the $\Theta(n^2)$ time of the straightforward method. We assume throughout that n is an exact power of 2. Although strategies

³The length n is actually what we referred to as $2n$ in Section 30.1, since we double the degree-bound of the given polynomials prior to evaluation. In the context of polynomial multiplication, therefore, we are actually working with complex $(2n)$ th roots of unity.

for dealing with non-power-of-2 sizes are known, they are beyond the scope of this book.

The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of $A(x)$ separately to define the two new polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ of degree-bound $n/2$:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}, \\ A^{[1]}(x) &= a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}. \end{aligned}$$

Note that $A^{[0]}$ contains all the even-indexed coefficients of A (the binary representation of the index ends in 0) and $A^{[1]}$ contains all the odd-indexed coefficients (the binary representation of the index ends in 1). It follows that

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2), \quad (30.9)$$

so that the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ reduces to

1. evaluating the degree-bound $n/2$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ at the points

$$(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2, \quad (30.10)$$

and then

2. combining the results according to equation (30.9).

By the halving lemma, the list of values (30.10) consists not of n distinct values but only of the $n/2$ complex $(n/2)$ th roots of unity, with each root occurring exactly twice. Therefore, we recursively evaluate the polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ at the $n/2$ complex $(n/2)$ th roots of unity. These subproblems have exactly the same form as the original problem, but are half the size. We have now successfully divided an n -element DFT_n computation into two $n/2$ -element $\text{DFT}_{n/2}$ computations. This decomposition is the basis for the following recursive FFT algorithm, which computes the DFT of an n -element vector $a = (a_0, a_1, \dots, a_{n-1})$, where n is a power of 2.

RECURSIVE-FFT(a)

```

1   $n = a.length$            //  $n$  is a power of 2
2  if  $n == 1$ 
3      return  $a$ 
4   $\omega_n = e^{2\pi i/n}$ 
5   $\omega = 1$ 
6   $a^{[0]} = (a_0, a_2, \dots, a_{n-2})$ 
7   $a^{[1]} = (a_1, a_3, \dots, a_{n-1})$ 
8   $y^{[0]} = \text{RECURSIVE-FFT}(a^{[0]})$ 
9   $y^{[1]} = \text{RECURSIVE-FFT}(a^{[1]})$ 
10 for  $k = 0$  to  $n/2 - 1$ 
11      $y_k = y_k^{[0]} + \omega y_k^{[1]}$ 
12      $y_{k+(n/2)} = y_k^{[0]} - \omega y_k^{[1]}$ 
13      $\omega = \omega \omega_n$ 
14 return  $y$            //  $y$  is assumed to be a column vector

```

The RECURSIVE-FFT procedure works as follows. Lines 2–3 represent the basis of the recursion; the DFT of one element is the element itself, since in this case

$$\begin{aligned}
 y_0 &= a_0 \omega_1^0 \\
 &= a_0 \cdot 1 \\
 &= a_0 .
 \end{aligned}$$

Lines 6–7 define the coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. Lines 4, 5, and 13 guarantee that ω is updated properly so that whenever lines 11–12 are executed, we have $\omega = \omega_n^k$. (Keeping a running value of ω from iteration to iteration saves time over computing ω_n^k from scratch each time through the **for** loop.) Lines 8–9 perform the recursive $\text{DFT}_{n/2}$ computations, setting, for $k = 0, 1, \dots, n/2 - 1$,

$$\begin{aligned}
 y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k) , \\
 y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) ,
 \end{aligned}$$

or, since $\omega_{n/2}^k = \omega_n^{2k}$ by the cancellation lemma,

$$\begin{aligned}
 y_k^{[0]} &= A^{[0]}(\omega_n^{2k}) , \\
 y_k^{[1]} &= A^{[1]}(\omega_n^{2k}) .
 \end{aligned}$$

Lines 11–12 combine the results of the recursive $\text{DFT}_{n/2}$ calculations. For $y_0, y_1, \dots, y_{n/2-1}$, line 11 yields

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \end{aligned} \quad (\text{by equation (30.9)}) .$$

For $y_{n/2}, y_{n/2+1}, \dots, y_{n-1}$, letting $k = 0, 1, \dots, n/2 - 1$, line 12 yields

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && (\text{since } \omega_n^{k+(n/2)} = -\omega_n^k) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && (\text{since } \omega_n^{2k+n} = \omega_n^{2k}) \\ &= A(\omega_n^{k+(n/2)}) && (\text{by equation (30.9)}) . \end{aligned}$$

Thus, the vector y returned by `RECURSIVE-FFT` is indeed the DFT of the input vector a .

Lines 11 and 12 multiply each value $y_k^{[1]}$ by ω_n^k , for $k = 0, 1, \dots, n/2 - 1$. Line 11 adds this product to $y_k^{[0]}$, and line 12 subtracts it. Because we use each factor ω_n^k in both its positive and negative forms, we call the factors ω_n^k *twiddle factors*.

To determine the running time of procedure `RECURSIVE-FFT`, we note that exclusive of the recursive calls, each invocation takes time $\Theta(n)$, where n is the length of the input vector. The recurrence for the running time is therefore

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

Thus, we can evaluate a polynomial of degree-bound n at the complex n th roots of unity in time $\Theta(n \lg n)$ using the fast Fourier transform.

Interpolation at the complex roots of unity

We now complete the polynomial multiplication scheme by showing how to interpolate the complex roots of unity by a polynomial, which enables us to convert from point-value form back to coefficient form. We interpolate by writing the DFT as a matrix equation and then looking at the form of the matrix inverse.

From equation (30.4), we can write the DFT as the matrix product $y = V_n a$, where V_n is a Vandermonde matrix containing the appropriate powers of ω_n :

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

The (k, j) entry of V_n is ω_n^{kj} , for $j, k = 0, 1, \dots, n-1$. The exponents of the entries of V_n form a multiplication table.

For the inverse operation, which we write as $a = \text{DFT}_n^{-1}(y)$, we proceed by multiplying y by the matrix V_n^{-1} , the inverse of V_n .

Theorem 30.7

For $j, k = 0, 1, \dots, n-1$, the (j, k) entry of V_n^{-1} is ω_n^{-kj}/n .

Proof We show that $V_n^{-1}V_n = I_n$, the $n \times n$ identity matrix. Consider the (j, j') entry of $V_n^{-1}V_n$:

$$\begin{aligned} [V_n^{-1}V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n. \end{aligned}$$

This summation equals 1 if $j' = j$, and it is 0 otherwise by the summation lemma (Lemma 30.6). Note that we rely on $-(n-1) \leq j' - j \leq n-1$, so that $j' - j$ is not divisible by n , in order for the summation lemma to apply. ■

Given the inverse matrix V_n^{-1} , we have that $\text{DFT}_n^{-1}(y)$ is given by

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

for $j = 0, 1, \dots, n-1$. By comparing equations (30.8) and (30.11), we see that by modifying the FFT algorithm to switch the roles of a and y , replace ω_n by ω_n^{-1} , and divide each element of the result by n , we compute the inverse DFT (see Exercise 30.2-4). Thus, we can compute DFT_n^{-1} in $\Theta(n \lg n)$ time as well.

We see that, by using the FFT and the inverse FFT, we can transform a polynomial of degree-bound n back and forth between its coefficient representation and a point-value representation in time $\Theta(n \lg n)$. In the context of polynomial multiplication, we have shown the following.

Theorem 30.8 (Convolution theorem)

For any two vectors a and b of length n , where n is a power of 2,

$$a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)),$$

where the vectors a and b are padded with 0s to length $2n$ and \cdot denotes the componentwise product of two $2n$ -element vectors. ■

Exercises**30.2-1**

Prove Corollary 30.4.

30.2-2

Compute the DFT of the vector $(0, 1, 2, 3)$.

30.2-3

Do Exercise 30.1-1 by using the $\Theta(n \lg n)$ -time scheme.

30.2-4

Write pseudocode to compute DFT_n^{-1} in $\Theta(n \lg n)$ time.

30.2-5

Describe the generalization of the FFT procedure to the case in which n is a power of 3. Give a recurrence for the running time, and solve the recurrence.

30.2-6 ★

Suppose that instead of performing an n -element FFT over the field of complex numbers (where n is even), we use the ring \mathbb{Z}_m of integers modulo m , where $m = 2^{tn/2} + 1$ and t is an arbitrary positive integer. Use $\omega = 2^t$ instead of ω_n as a principal n th root of unity, modulo m . Prove that the DFT and the inverse DFT are well defined in this system.

30.2-7

Given a list of values z_0, z_1, \dots, z_{n-1} (possibly with repetitions), show how to find the coefficients of a polynomial $P(x)$ of degree-bound $n + 1$ that has zeros only at z_0, z_1, \dots, z_{n-1} (possibly with repetitions). Your procedure should run in time $O(n \lg^2 n)$. (*Hint*: The polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$.)

30.2-8 ★

The *chirp transform* of a vector $a = (a_0, a_1, \dots, a_{n-1})$ is the vector $y = (y_0, y_1, \dots, y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ and z is any complex number. The

DFT is therefore a special case of the chirp transform, obtained by taking $z = \omega_n$. Show how to evaluate the chirp transform in time $O(n \lg n)$ for any complex number z . (*Hint*: Use the equation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2} \right) \left(z^{-(k-j)^2/2} \right)$$

to view the chirp transform as a convolution.)

30.3 Efficient FFT implementations

Since the practical applications of the DFT, such as signal processing, demand the utmost speed, this section examines two efficient FFT implementations. First, we shall examine an iterative version of the FFT algorithm that runs in $\Theta(n \lg n)$ time but can have a lower constant hidden in the Θ -notation than the recursive version in Section 30.2. (Depending on the exact implementation, the recursive version may use the hardware cache more efficiently.) Then, we shall use the insights that led us to the iterative implementation to design an efficient parallel FFT circuit.

An iterative FFT implementation

We first note that the **for** loop of lines 10–13 of RECURSIVE-FFT involves computing the value $\omega_n^k y_k^{[1]}$ twice. In compiler terminology, we call such a value a *common subexpression*. We can change the loop to compute it only once, storing it in a temporary variable t .

```

for  $k = 0$  to  $n/2 - 1$ 
     $t = \omega y_k^{[1]}$ 
     $y_k = y_k^{[0]} + t$ 
     $y_{k+(n/2)} = y_k^{[0]} - t$ 
     $\omega = \omega \omega_n$ 

```

The operation in this loop, multiplying the twiddle factor $\omega = \omega_n^k$ by $y_k^{[1]}$, storing the product into t , and adding and subtracting t from $y_k^{[0]}$, is known as a *butterfly operation* and is shown schematically in Figure 30.3.

We now show how to make the FFT algorithm iterative rather than recursive in structure. In Figure 30.4, we have arranged the input vectors to the recursive calls in an invocation of RECURSIVE-FFT in a tree structure, where the initial call is for $n = 8$. The tree has one node for each call of the procedure, labeled

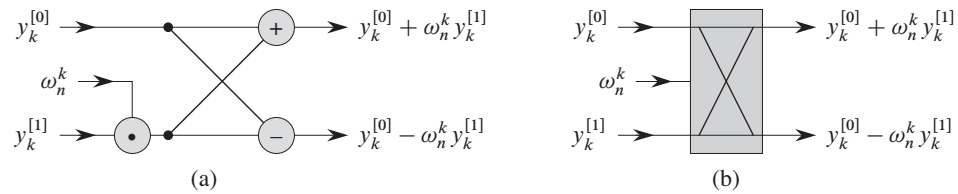


Figure 30.3 A butterfly operation. (a) The two input values enter from the left, the twiddle factor ω_n^k is multiplied by $y_k^{[1]}$, and the sum and difference are output on the right. (b) A simplified drawing of a butterfly operation. We will use this representation in a parallel FFT circuit.

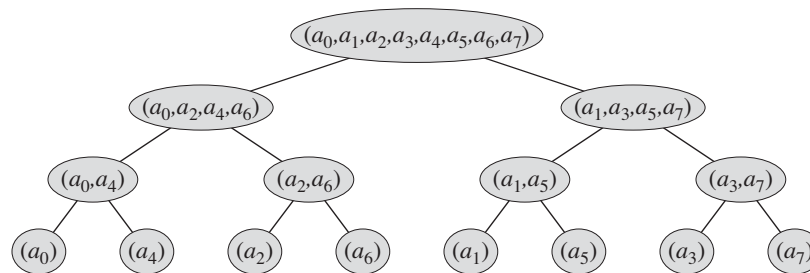


Figure 30.4 The tree of input vectors to the recursive calls of the RECURSIVE-FFT procedure. The initial invocation is for $n = 8$.

by the corresponding input vector. Each RECURSIVE-FFT invocation makes two recursive calls, unless it has received a 1-element vector. The first call appears in the left child, and the second call appears in the right child.

Looking at the tree, we observe that if we could arrange the elements of the initial vector a into the order in which they appear in the leaves, we could trace the execution of the RECURSIVE-FFT procedure, but bottom up instead of top down. First, we take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT. The vector then holds $n/2$ 2-element DFTs. Next, we take these $n/2$ DFTs in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two 2-element DFTs with one 4-element DFT. The vector then holds $n/4$ 4-element DFTs. We continue in this manner until the vector holds two $(n/2)$ -element DFTs, which we combine using $n/2$ butterfly operations into the final n -element DFT.

To turn this bottom-up approach into code, we use an array $A[0..n-1]$ that initially holds the elements of the input vector a in the order in which they appear

in the leaves of the tree of Figure 30.4. (We shall show later how to determine this order, which is known as a bit-reversal permutation.) Because we have to combine DFTs on each level of the tree, we introduce a variable s to count the levels, ranging from 1 (at the bottom, when we are combining pairs to form 2-element DFTs) to $\lg n$ (at the top, when we are combining two $(n/2)$ -element DFTs to produce the final result). The algorithm therefore has the following structure:

```

1  for  $s = 1$  to  $\lg n$ 
2      for  $k = 0$  to  $n - 1$  by  $2^s$ 
3          combine the two  $2^{s-1}$ -element DFTs in
                 $A[k \dots k + 2^{s-1} - 1]$  and  $A[k + 2^{s-1} \dots k + 2^s - 1]$ 
                into one  $2^s$ -element DFT in  $A[k \dots k + 2^s - 1]$ 

```

We can express the body of the loop (line 3) as more precise pseudocode. We copy the **for** loop from the RECURSIVE-FFT procedure, identifying $y^{[0]}$ with $A[k \dots k + 2^{s-1} - 1]$ and $y^{[1]}$ with $A[k + 2^{s-1} \dots k + 2^s - 1]$. The twiddle factor used in each butterfly operation depends on the value of s ; it is a power of ω_m , where $m = 2^s$. (We introduce the variable m solely for the sake of readability.) We introduce another temporary variable u that allows us to perform the butterfly operation in place. When we replace line 3 of the overall structure by the loop body, we get the following pseudocode, which forms the basis of the parallel implementation we shall present later. The code first calls the auxiliary procedure BIT-REVERSE-COPY(a, A) to copy vector a into array A in the initial order in which we need the values.

```

ITERATIVE-FFT( $a$ )
1  BIT-REVERSE-COPY( $a, A$ )
2   $n = a.length$            //  $n$  is a power of 2
3  for  $s = 1$  to  $\lg n$ 
4       $m = 2^s$ 
5       $\omega_m = e^{2\pi i/m}$ 
6      for  $k = 0$  to  $n - 1$  by  $m$ 
7           $\omega = 1$ 
8          for  $j = 0$  to  $m/2 - 1$ 
9               $t = \omega A[k + j + m/2]$ 
10              $u = A[k + j]$ 
11              $A[k + j] = u + t$ 
12              $A[k + j + m/2] = u - t$ 
13              $\omega = \omega \omega_m$ 
14  return  $A$ 

```

How does BIT-REVERSE-COPY get the elements of the input vector a into the desired order in the array A ? The order in which the leaves appear in Figure 30.4

is a **bit-reversal permutation**. That is, if we let $\text{rev}(k)$ be the $\lg n$ -bit integer formed by reversing the bits of the binary representation of k , then we want to place vector element a_k in array position $A[\text{rev}(k)]$. In Figure 30.4, for example, the leaves appear in the order 0, 4, 2, 6, 1, 5, 3, 7; this sequence in binary is 000, 100, 010, 110, 001, 101, 011, 111, and when we reverse the bits of each value we get the sequence 000, 001, 010, 011, 100, 101, 110, 111. To see that we want a bit-reversal permutation in general, we note that at the top level of the tree, indices whose low-order bit is 0 go into the left subtree and indices whose low-order bit is 1 go into the right subtree. Stripping off the low-order bit at each level, we continue this process down the tree, until we get the order given by the bit-reversal permutation at the leaves.

Since we can easily compute the function $\text{rev}(k)$, the BIT-REVERSE-COPY procedure is simple:

BIT-REVERSE-COPY(a, A)

```

1   $n = a.length$ 
2  for  $k = 0$  to  $n - 1$ 
3       $A[\text{rev}(k)] = a_k$ 

```

The iterative FFT implementation runs in time $\Theta(n \lg n)$. The call to BIT-REVERSE-COPY(a, A) certainly runs in $O(n \lg n)$ time, since we iterate n times and can reverse an integer between 0 and $n - 1$, with $\lg n$ bits, in $O(\lg n)$ time. (In practice, because we usually know the initial value of n in advance, we would probably code a table mapping k to $\text{rev}(k)$, making BIT-REVERSE-COPY run in $\Theta(n)$ time with a low hidden constant. Alternatively, we could use the clever amortized reverse binary counter scheme described in Problem 17-1.) To complete the proof that ITERATIVE-FFT runs in time $\Theta(n \lg n)$, we show that $L(n)$, the number of times the body of the innermost loop (lines 8–13) executes, is $\Theta(n \lg n)$. The **for** loop of lines 6–13 iterates $n/m = n/2^s$ times for each value of s , and the innermost loop of lines 8–13 iterates $m/2 = 2^{s-1}$ times. Thus,

$$\begin{aligned}
 L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
 &= \sum_{s=1}^{\lg n} \frac{n}{2} \\
 &= \Theta(n \lg n).
 \end{aligned}$$

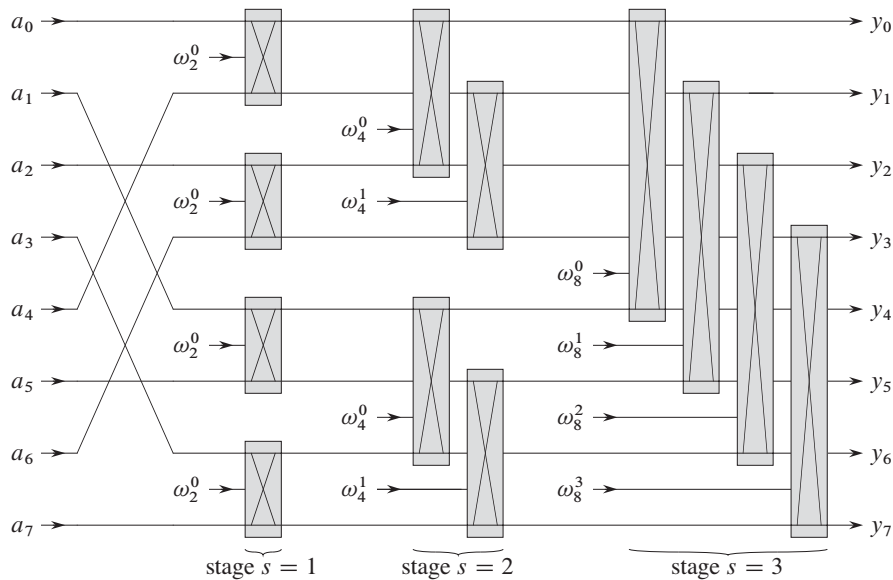


Figure 30.5 A circuit that computes the FFT in parallel, here shown on $n = 8$ inputs. Each butterfly operation takes as input the values on two wires, along with a twiddle factor, and it produces as outputs the values on two wires. The stages of butterflies are labeled to correspond to iterations of the outermost loop of the ITERATIVE-FFT procedure. Only the top and bottom wires passing through a butterfly interact with it; wires that pass through the middle of a butterfly do not affect that butterfly, nor are their values changed by that butterfly. For example, the top butterfly in stage 2 has nothing to do with wire 1 (the wire whose output is labeled y_1); its inputs and outputs are only on wires 0 and 2 (labeled y_0 and y_2 , respectively). This circuit has depth $\Theta(\lg n)$ and performs $\Theta(n \lg n)$ butterfly operations altogether.

A parallel FFT circuit

We can exploit many of the properties that allowed us to implement an efficient iterative FFT algorithm to produce an efficient parallel algorithm for the FFT. We will express the parallel FFT algorithm as a circuit. Figure 30.5 shows a parallel FFT circuit, which computes the FFT on n inputs, for $n = 8$. The circuit begins with a bit-reverse permutation of the inputs, followed by $\lg n$ stages, each stage consisting of $n/2$ butterflies executed in parallel. The *depth* of the circuit—the maximum number of computational elements between any output and any input that can reach it—is therefore $\Theta(\lg n)$.

The leftmost part of the parallel FFT circuit performs the bit-reverse permutation, and the remainder mimics the iterative ITERATIVE-FFT procedure. Because each iteration of the outermost **for** loop performs $n/2$ independent butterfly operations, the circuit performs them in parallel. The value of s in each iteration within

ITERATIVE-FFT corresponds to a stage of butterflies shown in Figure 30.5. For $s = 1, 2, \dots, \lg n$, stage s consists of $n/2^s$ groups of butterflies (corresponding to each value of k in ITERATIVE-FFT), with 2^{s-1} butterflies per group (corresponding to each value of j in ITERATIVE-FFT). The butterflies shown in Figure 30.5 correspond to the butterfly operations of the innermost loop (lines 9–12 of ITERATIVE-FFT). Note also that the twiddle factors used in the butterflies correspond to those used in ITERATIVE-FFT: in stage s , we use $\omega_m^0, \omega_m^1, \dots, \omega_m^{m/2-1}$, where $m = 2^s$.

Exercises

30.3-1

Show how ITERATIVE-FFT computes the DFT of the input vector $(0, 2, 3, -1, 4, 5, 7, 9)$.

30.3-2

Show how to implement an FFT algorithm with the bit-reversal permutation occurring at the end, rather than at the beginning, of the computation. (*Hint*: Consider the inverse DFT.)

30.3-3

How many times does ITERATIVE-FFT compute twiddle factors in each stage? Rewrite ITERATIVE-FFT to compute twiddle factors only 2^{s-1} times in stage s .

30.3-4 ★

Suppose that the adders within the butterfly operations of the FFT circuit sometimes fail in such a manner that they always produce a zero output, independent of their inputs. Suppose that exactly one adder has failed, but that you don't know which one. Describe how you can identify the failed adder by supplying inputs to the overall FFT circuit and observing the outputs. How efficient is your method?

Problems

30-1 Divide-and-conquer multiplication

- Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (*Hint*: One of the multiplications is $(a + b) \cdot (c + d)$.)
- Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n in $\Theta(n^{\lg 3})$ time. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.

- c. Show how to multiply two n -bit integers in $O(n^{\lg 3})$ steps, where each step operates on at most a constant number of 1-bit values.

30-2 Toeplitz matrices

A **Toeplitz matrix** is an $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1, j-1}$ for $i = 2, 3, \dots, n$ and $j = 2, 3, \dots, n$.

- a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?
- b. Describe how to represent a Toeplitz matrix so that you can add two $n \times n$ Toeplitz matrices in $O(n)$ time.
- c. Give an $O(n \lg n)$ -time algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n . Use your representation from part (b).
- d. Give an efficient algorithm for multiplying two $n \times n$ Toeplitz matrices. Analyze its running time.

30-3 Multidimensional fast Fourier transform

We can generalize the 1-dimensional discrete Fourier transform defined by equation (30.8) to d dimensions. The input is a d -dimensional array $A = (a_{j_1, j_2, \dots, j_d})$ whose dimensions are n_1, n_2, \dots, n_d , where $n_1 n_2 \cdots n_d = n$. We define the d -dimensional discrete Fourier transform by the equation

$$y_{k_1, k_2, \dots, k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1, j_2, \dots, j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

for $0 \leq k_1 < n_1, 0 \leq k_2 < n_2, \dots, 0 \leq k_d < n_d$.

- a. Show that we can compute a d -dimensional DFT by computing 1-dimensional DFTs on each dimension in turn. That is, we first compute n/n_1 separate 1-dimensional DFTs along dimension 1. Then, using the result of the DFTs along dimension 1 as the input, we compute n/n_2 separate 1-dimensional DFTs along dimension 2. Using this result as the input, we compute n/n_3 separate 1-dimensional DFTs along dimension 3, and so on, through dimension d .
- b. Show that the ordering of dimensions does not matter, so that we can compute a d -dimensional DFT by computing the 1-dimensional DFTs in any order of the d dimensions.

- c. Show that if we compute each 1-dimensional DFT by computing the fast Fourier transform, the total time to compute a d -dimensional DFT is $O(n \lg n)$, independent of d .

30-4 Evaluating all derivatives of a polynomial at a point

Given a polynomial $A(x)$ of degree-bound n , we define its t th derivative by

$$A^{(t)}(x) = \begin{cases} A(x) & \text{if } t = 0, \\ \frac{d}{dx}A^{(t-1)}(x) & \text{if } 1 \leq t \leq n-1, \\ 0 & \text{if } t \geq n. \end{cases}$$

From the coefficient representation $(a_0, a_1, \dots, a_{n-1})$ of $A(x)$ and a given point x_0 , we wish to determine $A^{(t)}(x_0)$ for $t = 0, 1, \dots, n-1$.

- a. Given coefficients b_0, b_1, \dots, b_{n-1} such that

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j,$$

show how to compute $A^{(t)}(x_0)$, for $t = 0, 1, \dots, n-1$, in $O(n)$ time.

- b. Explain how to find b_0, b_1, \dots, b_{n-1} in $O(n \lg n)$ time, given $A(x_0 + \omega_n^k)$ for $k = 0, 1, \dots, n-1$.
- c. Prove that

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right),$$

where $f(j) = a_j \cdot j!$ and

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{if } -(n-1) \leq l \leq 0, \\ 0 & \text{if } 1 \leq l \leq n-1. \end{cases}$$

- d. Explain how to evaluate $A(x_0 + \omega_n^k)$ for $k = 0, 1, \dots, n-1$ in $O(n \lg n)$ time. Conclude that we can evaluate all nontrivial derivatives of $A(x)$ at x_0 in $O(n \lg n)$ time.

30-5 Polynomial evaluation at multiple points

We have seen how to evaluate a polynomial of degree-bound n at a single point in $O(n)$ time using Horner's rule. We have also discovered how to evaluate such a polynomial at all n complex roots of unity in $O(n \lg n)$ time using the FFT. We shall now show how to evaluate a polynomial of degree-bound n at n arbitrary points in $O(n \lg^2 n)$ time.

To do so, we shall assume that we can compute the polynomial remainder when one such polynomial is divided by another in $O(n \lg n)$ time, a result that we state without proof. For example, the remainder of $3x^3 + x^2 - 3x + 1$ when divided by $x^2 + x + 2$ is

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5.$$

Given the coefficient representation of a polynomial $A(x) = \sum_{k=0}^{n-1} a_k x^k$ and n points x_0, x_1, \dots, x_{n-1} , we wish to compute the n values $A(x_0), A(x_1), \dots, A(x_{n-1})$. For $0 \leq i \leq j \leq n-1$, define the polynomials $P_{ij}(x) = \prod_{k=i}^j (x - x_k)$ and $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$. Note that $Q_{ij}(x)$ has degree at most $j - i$.

- a. Prove that $A(x) \bmod (x - z) = A(z)$ for any point z .
- b. Prove that $Q_{kk}(x) = A(x_k)$ and that $Q_{0,n-1}(x) = A(x)$.
- c. Prove that for $i \leq k \leq j$, we have $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ and $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.
- d. Give an $O(n \lg^2 n)$ -time algorithm to evaluate $A(x_0), A(x_1), \dots, A(x_{n-1})$.

30-6 FFT using modular arithmetic

As defined, the discrete Fourier transform requires us to compute with complex numbers, which can result in a loss of precision due to round-off errors. For some problems, the answer is known to contain only integers, and by using a variant of the FFT based on modular arithmetic, we can guarantee that the answer is calculated exactly. An example of such a problem is that of multiplying two polynomials with integer coefficients. Exercise 30.2-6 gives one approach, using a modulus of length $\Omega(n)$ bits to handle a DFT on n points. This problem gives another approach, which uses a modulus of the more reasonable length $O(\lg n)$; it requires that you understand the material of Chapter 31. Let n be a power of 2.

- a. Suppose that we search for the smallest k such that $p = kn + 1$ is prime. Give a simple heuristic argument why we might expect k to be approximately $\ln n$. (The value of k might be much larger or smaller, but we can reasonably expect to examine $O(\lg n)$ candidate values of k on average.) How does the expected length of p compare to the length of n ?

Let g be a generator of \mathbb{Z}_p^* , and let $w = g^k \bmod p$.

- b.* Argue that the DFT and the inverse DFT are well-defined inverse operations modulo p , where w is used as a principal n th root of unity.
- c.* Show how to make the FFT and its inverse work modulo p in time $O(n \lg n)$, where operations on words of $O(\lg n)$ bits take unit time. Assume that the algorithm is given p and w .
- d.* Compute the DFT modulo $p = 17$ of the vector $(0, 5, 3, 7, 7, 2, 1, 6)$. Note that $g = 3$ is a generator of \mathbb{Z}_{17}^* .

Chapter notes

Van Loan's book [343] provides an outstanding treatment of the fast Fourier transform. Press, Teukolsky, Vetterling, and Flannery [283, 284] have a good description of the fast Fourier transform and its applications. For an excellent introduction to signal processing, a popular FFT application area, see the texts by Oppenheim and Schaffer [266] and Oppenheim and Willsky [267]. The Oppenheim and Schaffer book also shows how to handle cases in which n is not an integer power of 2.

Fourier analysis is not limited to 1-dimensional data. It is widely used in image processing to analyze data in 2 or more dimensions. The books by Gonzalez and Woods [146] and Pratt [281] discuss multidimensional Fourier transforms and their use in image processing, and books by Tolimieri, An, and Lu [338] and Van Loan [343] discuss the mathematics of multidimensional fast Fourier transforms.

Cooley and Tukey [76] are widely credited with devising the FFT in the 1960s. The FFT had in fact been discovered many times previously, but its importance was not fully realized before the advent of modern digital computers. Although Press, Teukolsky, Vetterling, and Flannery attribute the origins of the method to Runge and König in 1924, an article by Heideman, Johnson, and Burrus [163] traces the history of the FFT as far back as C. F. Gauss in 1805.

Frigo and Johnson [117] developed a fast and flexible implementation of the FFT, called FFTW ("fastest Fourier transform in the West"). FFTW is designed for situations requiring multiple DFT computations on the same problem size. Before actually computing the DFTs, FFTW executes a "planner," which, by a series of trial runs, determines how best to decompose the FFT computation for the given problem size on the host machine. FFTW adapts to use the hardware cache efficiently, and once subproblems are small enough, FFTW solves them with optimized, straight-line code. Furthermore, FFTW has the unusual advantage of taking $\Theta(n \lg n)$ time for any problem size n , even when n is a large prime.

Although the standard Fourier transform assumes that the input represents points that are uniformly spaced in the time domain, other techniques can approximate the FFT on “nonequispaced” data. The article by Ware [348] provides an overview.

Number theory was once viewed as a beautiful but largely useless subject in pure mathematics. Today number-theoretic algorithms are used widely, due in large part to the invention of cryptographic schemes based on large prime numbers. These schemes are feasible because we can find large primes easily, and they are secure because we do not know how to factor the product of large primes (or solve related problems, such as computing discrete logarithms) efficiently. This chapter presents some of the number theory and related algorithms that underlie such applications.

Section 31.1 introduces basic concepts of number theory, such as divisibility, modular equivalence, and unique factorization. Section 31.2 studies one of the world's oldest algorithms: Euclid's algorithm for computing the greatest common divisor of two integers. Section 31.3 reviews concepts of modular arithmetic. Section 31.4 then studies the set of multiples of a given number a , modulo n , and shows how to find all solutions to the equation $ax \equiv b \pmod{n}$ by using Euclid's algorithm. The Chinese remainder theorem is presented in Section 31.5. Section 31.6 considers powers of a given number a , modulo n , and presents a repeated-squaring algorithm for efficiently computing $a^b \pmod{n}$, given a , b , and n . This operation is at the heart of efficient primality testing and of much modern cryptography. Section 31.7 then describes the RSA public-key cryptosystem. Section 31.8 examines a randomized primality test. We can use this test to find large primes efficiently, which we need to do in order to create keys for the RSA cryptosystem. Finally, Section 31.9 reviews a simple but effective heuristic for factoring small integers. It is a curious fact that factoring is one problem people may wish to be intractable, since the security of RSA depends on the difficulty of factoring large integers.

Size of inputs and cost of arithmetic computations

Because we shall be working with large integers, we need to adjust how we think about the size of an input and about the cost of elementary arithmetic operations.

In this chapter, a “large input” typically means an input containing “large integers” rather than an input containing “many integers” (as for sorting). Thus,

we shall measure the size of an input in terms of the *number of bits* required to represent that input, not just the number of integers in the input. An algorithm with integer inputs a_1, a_2, \dots, a_k is a **polynomial-time algorithm** if it runs in time polynomial in $\lg a_1, \lg a_2, \dots, \lg a_k$, that is, polynomial in the lengths of its binary-encoded inputs.

In most of this book, we have found it convenient to think of the elementary arithmetic operations (multiplications, divisions, or computing remainders) as primitive operations that take one unit of time. By counting the number of such arithmetic operations that an algorithm performs, we have a basis for making a reasonable estimate of the algorithm's actual running time on a computer. Elementary operations can be time-consuming, however, when their inputs are large. It thus becomes convenient to measure how many **bit operations** a number-theoretic algorithm requires. In this model, multiplying two β -bit integers by the ordinary method uses $\Theta(\beta^2)$ bit operations. Similarly, we can divide a β -bit integer by a shorter integer or take the remainder of a β -bit integer when divided by a shorter integer in time $\Theta(\beta^2)$ by simple algorithms. (See Exercise 31.1-12.) Faster methods are known. For example, a simple divide-and-conquer method for multiplying two β -bit integers has a running time of $\Theta(\beta^{\lg 3})$, and the fastest known method has a running time of $\Theta(\beta \lg \beta \lg \lg \beta)$. For practical purposes, however, the $\Theta(\beta^2)$ algorithm is often best, and we shall use this bound as a basis for our analyses.

We shall generally analyze algorithms in this chapter in terms of both the number of arithmetic operations and the number of bit operations they require.

31.1 Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory concerning the set $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ of integers and the set $\mathbb{N} = \{0, 1, 2, \dots\}$ of natural numbers.

Divisibility and divisors

The notion of one integer being divisible by another is key to the theory of numbers. The notation $d \mid a$ (read “ d **divides** a ”) means that $a = kd$ for some integer k . Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \leq |a|$. If $d \mid a$, then we also say that a is a **multiple** of d . If d does not divide a , we write $d \nmid a$.

If $d \mid a$ and $d \geq 0$, we say that d is a **divisor** of a . Note that $d \mid a$ if and only if $-d \mid a$, so that no generality is lost by defining the divisors to be nonnegative, with the understanding that the negative of any divisor of a also divides a . A

divisor of a nonzero integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Every positive integer a is divisible by the *trivial divisors* 1 and a . The nontrivial divisors of a are the *factors* of a . For example, the factors of 20 are 2, 4, 5, and 10.

Prime and composite numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is a *prime number* or, more simply, a *prime*. Primes have many special properties and play a critical role in number theory. The first 20 primes, in order, are

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71 .

Exercise 31.1-2 asks you to prove that there are infinitely many primes. An integer $a > 1$ that is not prime is a *composite number* or, more simply, a *composite*. For example, 39 is composite because $3 \mid 39$. We call the integer 1 a *unit*, and it is neither prime nor composite. Similarly, the integer 0 and all negative integers are neither prime nor composite.

The division theorem, remainders, and modular equivalence

Given an integer n , we can partition the integers into those that are multiples of n and those that are not multiples of n . Much number theory is based upon refining this partition by classifying the nonmultiples of n according to their remainders when divided by n . The following theorem provides the basis for this refinement. We omit the proof (but see, for example, Niven and Zuckerman [265]).

Theorem 31.1 (Division theorem)

For any integer a and any positive integer n , there exist unique integers q and r such that $0 \leq r < n$ and $a = qn + r$. ■

The value $q = \lfloor a/n \rfloor$ is the *quotient* of the division. The value $r = a \bmod n$ is the *remainder* (or *residue*) of the division. We have that $n \mid a$ if and only if $a \bmod n = 0$.

We can partition the integers into n equivalence classes according to their remainders modulo n . The *equivalence class modulo n* containing an integer a is

$$[a]_n = \{a + kn : k \in \mathbb{Z}\} .$$

For example, $[3]_7 = \{\dots, -11, -4, 3, 10, 17, \dots\}$; we can also denote this set by $[-4]_7$ and $[10]_7$. Using the notation defined on page 54, we can say that writing $a \in [b]_n$ is the same as writing $a \equiv b \pmod{n}$. The set of all such equivalence classes is

$$\mathbb{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\} . \quad (31.1)$$

When you see the definition

$$\mathbb{Z}_n = \{0, 1, \dots, n - 1\} , \quad (31.2)$$

you should read it as equivalent to equation (31.1) with the understanding that 0 represents $[0]_n$, 1 represents $[1]_n$, and so on; each class is represented by its smallest nonnegative element. You should keep the underlying equivalence classes in mind, however. For example, if we refer to -1 as a member of \mathbb{Z}_n , we are really referring to $[n - 1]_n$, since $-1 \equiv n - 1 \pmod{n}$.

Common divisors and greatest common divisors

If d is a divisor of a and d is also a divisor of b , then d is a **common divisor** of a and b . For example, the divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30, and so the common divisors of 24 and 30 are 1, 2, 3, and 6. Note that 1 is a common divisor of any two integers.

An important property of common divisors is that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (a + b) \text{ and } d \mid (a - b) . \quad (31.3)$$

More generally, we have that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (ax + by) \quad (31.4)$$

for any integers x and y . Also, if $a \mid b$, then either $|a| \leq |b|$ or $b = 0$, which implies that

$$a \mid b \text{ and } b \mid a \text{ implies } a = \pm b . \quad (31.5)$$

The **greatest common divisor** of two integers a and b , not both zero, is the largest of the common divisors of a and b ; we denote it by $\gcd(a, b)$. For example, $\gcd(24, 30) = 6$, $\gcd(5, 7) = 1$, and $\gcd(0, 9) = 9$. If a and b are both nonzero, then $\gcd(a, b)$ is an integer between 1 and $\min(|a|, |b|)$. We define $\gcd(0, 0)$ to be 0; this definition is necessary to make standard properties of the gcd function (such as equation (31.9) below) universally valid.

The following are elementary properties of the gcd function:

$$\gcd(a, b) = \gcd(b, a) , \quad (31.6)$$

$$\gcd(a, b) = \gcd(-a, b) , \quad (31.7)$$

$$\gcd(a, b) = \gcd(|a|, |b|) , \quad (31.8)$$

$$\gcd(a, 0) = |a| , \quad (31.9)$$

$$\gcd(a, ka) = |a| \quad \text{for any } k \in \mathbb{Z} . \quad (31.10)$$

The following theorem provides an alternative and useful characterization of $\gcd(a, b)$.

Theorem 31.2

If a and b are any integers, not both zero, then $\gcd(a, b)$ is the smallest positive element of the set $\{ax + by : x, y \in \mathbb{Z}\}$ of linear combinations of a and b .

Proof Let s be the smallest positive such linear combination of a and b , and let $s = ax + by$ for some $x, y \in \mathbb{Z}$. Let $q = \lfloor a/s \rfloor$. Equation (3.8) then implies

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(-qy), \end{aligned}$$

and so $a \bmod s$ is a linear combination of a and b as well. But, since $0 \leq a \bmod s < s$, we have that $a \bmod s = 0$, because s is the smallest positive such linear combination. Therefore, we have that $s \mid a$ and, by analogous reasoning, $s \mid b$. Thus, s is a common divisor of a and b , and so $\gcd(a, b) \geq s$. Equation (31.4) implies that $\gcd(a, b) \mid s$, since $\gcd(a, b)$ divides both a and b and s is a linear combination of a and b . But $\gcd(a, b) \mid s$ and $s > 0$ imply that $\gcd(a, b) \leq s$. Combining $\gcd(a, b) \geq s$ and $\gcd(a, b) \leq s$ yields $\gcd(a, b) = s$. We conclude that s is the greatest common divisor of a and b . ■

Corollary 31.3

For any integers a and b , if $d \mid a$ and $d \mid b$, then $d \mid \gcd(a, b)$.

Proof This corollary follows from equation (31.4), because $\gcd(a, b)$ is a linear combination of a and b by Theorem 31.2. ■

Corollary 31.4

For all integers a and b and any nonnegative integer n ,

$$\gcd(an, bn) = n \gcd(a, b).$$

Proof If $n = 0$, the corollary is trivial. If $n > 0$, then $\gcd(an, bn)$ is the smallest positive element of the set $\{anx + bny : x, y \in \mathbb{Z}\}$, which is n times the smallest positive element of the set $\{ax + by : x, y \in \mathbb{Z}\}$. ■

Corollary 31.5

For all positive integers n , a , and b , if $n \mid ab$ and $\gcd(a, n) = 1$, then $n \mid b$.

Proof We leave the proof as Exercise 31.1-5. ■

Relatively prime integers

Two integers a and b are *relatively prime* if their only common divisor is 1, that is, if $\gcd(a, b) = 1$. For example, 8 and 15 are relatively prime, since the divisors of 8 are 1, 2, 4, and 8, and the divisors of 15 are 1, 3, 5, and 15. The following theorem states that if two integers are each relatively prime to an integer p , then their product is relatively prime to p .

Theorem 31.6

For any integers a , b , and p , if both $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, then $\gcd(ab, p) = 1$.

Proof It follows from Theorem 31.2 that there exist integers x , y , x' , and y' such that

$$\begin{aligned} ax + py &= 1, \\ bx' + py' &= 1. \end{aligned}$$

Multiplying these equations and rearranging, we have

$$ab(xx') + p(ybx' + y'ax + pyy') = 1.$$

Since 1 is thus a positive linear combination of ab and p , an appeal to Theorem 31.2 completes the proof. ■

Integers n_1, n_2, \dots, n_k are *pairwise relatively prime* if, whenever $i \neq j$, we have $\gcd(n_i, n_j) = 1$.

Unique factorization

An elementary but important fact about divisibility by primes is the following.

Theorem 31.7

For all primes p and all integers a and b , if $p \mid ab$, then $p \mid a$ or $p \mid b$ (or both).

Proof Assume for the purpose of contradiction that $p \mid ab$, but that $p \nmid a$ and $p \nmid b$. Thus, $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, since the only divisors of p are 1 and p , and we assume that p divides neither a nor b . Theorem 31.6 then implies that $\gcd(ab, p) = 1$, contradicting our assumption that $p \mid ab$, since $p \mid ab$ implies $\gcd(ab, p) = p$. This contradiction completes the proof. ■

A consequence of Theorem 31.7 is that we can uniquely factor any composite integer into a product of primes.

Theorem 31.8 (Unique factorization)

There is exactly one way to write any composite integer a as a product of the form

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r},$$

where the p_i are prime, $p_1 < p_2 < \cdots < p_r$, and the e_i are positive integers.

Proof We leave the proof as Exercise 31.1-11. ■

As an example, the number 6000 is uniquely factored into primes as $2^4 \cdot 3 \cdot 5^3$.

Exercises**31.1-1**

Prove that if $a > b > 0$ and $c = a + b$, then $c \bmod a = b$.

31.1-2

Prove that there are infinitely many primes. (*Hint*: Show that none of the primes p_1, p_2, \dots, p_k divide $(p_1 p_2 \cdots p_k) + 1$.)

31.1-3

Prove that if $a \mid b$ and $b \mid c$, then $a \mid c$.

31.1-4

Prove that if p is prime and $0 < k < p$, then $\gcd(k, p) = 1$.

31.1-5

Prove Corollary 31.5.

31.1-6

Prove that if p is prime and $0 < k < p$, then $p \mid \binom{p}{k}$. Conclude that for all integers a and b and all primes p ,

$$(a + b)^p \equiv a^p + b^p \pmod{p}.$$

31.1-7

Prove that if a and b are any positive integers such that $a \mid b$, then

$$(x \bmod b) \bmod a = x \bmod a$$

for any x . Prove, under the same assumptions, that

$$x \equiv y \pmod{b} \text{ implies } x \equiv y \pmod{a}$$

for any integers x and y .

31.1-8

For any integer $k > 0$, an integer n is a ***k*th power** if there exists an integer a such that $a^k = n$. Furthermore, $n > 1$ is a ***nontrivial power*** if it is a k th power for some integer $k > 1$. Show how to determine whether a given β -bit integer n is a nontrivial power in time polynomial in β .

31.1-9

Prove equations (31.6)–(31.10).

31.1-10

Show that the gcd operator is associative. That is, prove that for all integers a , b , and c ,

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c) .$$

31.1-11 ★

Prove Theorem 31.8.

31.1-12

Give efficient algorithms for the operations of dividing a β -bit integer by a shorter integer and of taking the remainder of a β -bit integer when divided by a shorter integer. Your algorithms should run in time $\Theta(\beta^2)$.

31.1-13

Give an efficient algorithm to convert a given β -bit (binary) integer to a decimal representation. Argue that if multiplication or division of integers whose length is at most β takes time $M(\beta)$, then we can convert binary to decimal in time $\Theta(M(\beta) \lg \beta)$. (*Hint*: Use a divide-and-conquer approach, obtaining the top and bottom halves of the result with separate recursions.)

31.2 Greatest common divisor

In this section, we describe Euclid's algorithm for efficiently computing the greatest common divisor of two integers. When we analyze the running time, we shall see a surprising connection with the Fibonacci numbers, which yield a worst-case input for Euclid's algorithm.

We restrict ourselves in this section to nonnegative integers. This restriction is justified by equation (31.8), which states that $\gcd(a, b) = \gcd(|a|, |b|)$.

In principle, we can compute $\gcd(a, b)$ for positive integers a and b from the prime factorizations of a and b . Indeed, if

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \quad (31.11)$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r}, \quad (31.12)$$

with zero exponents being used to make the set of primes p_1, p_2, \dots, p_r the same for both a and b , then, as Exercise 31.2-1 asks you to show,

$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)}. \quad (31.13)$$

As we shall show in Section 31.9, however, the best algorithms to date for factoring do not run in polynomial time. Thus, this approach to computing greatest common divisors seems unlikely to yield an efficient algorithm.

Euclid's algorithm for computing greatest common divisors relies on the following theorem.

Theorem 31.9 (GCD recursion theorem)

For any nonnegative integer a and any positive integer b ,

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

Proof We shall show that $\gcd(a, b)$ and $\gcd(b, a \bmod b)$ divide each other, so that by equation (31.5) they must be equal (since they are both nonnegative).

We first show that $\gcd(a, b) \mid \gcd(b, a \bmod b)$. If we let $d = \gcd(a, b)$, then $d \mid a$ and $d \mid b$. By equation (3.8), $a \bmod b = a - qb$, where $q = \lfloor a/b \rfloor$. Since $a \bmod b$ is thus a linear combination of a and b , equation (31.4) implies that $d \mid (a \bmod b)$. Therefore, since $d \mid b$ and $d \mid (a \bmod b)$, Corollary 31.3 implies that $d \mid \gcd(b, a \bmod b)$ or, equivalently, that

$$\gcd(a, b) \mid \gcd(b, a \bmod b). \quad (31.14)$$

Showing that $\gcd(b, a \bmod b) \mid \gcd(a, b)$ is almost the same. If we now let $d = \gcd(b, a \bmod b)$, then $d \mid b$ and $d \mid (a \bmod b)$. Since $a = qb + (a \bmod b)$, where $q = \lfloor a/b \rfloor$, we have that a is a linear combination of b and $(a \bmod b)$. By equation (31.4), we conclude that $d \mid a$. Since $d \mid b$ and $d \mid a$, we have that $d \mid \gcd(a, b)$ by Corollary 31.3 or, equivalently, that

$$\gcd(b, a \bmod b) \mid \gcd(a, b). \quad (31.15)$$

Using equation (31.5) to combine equations (31.14) and (31.15) completes the proof. ■

Euclid's algorithm

The *Elements* of Euclid (circa 300 B.C.) describes the following gcd algorithm, although it may be of even earlier origin. We express Euclid's algorithm as a recursive program based directly on Theorem 31.9. The inputs a and b are arbitrary nonnegative integers.

```

EUCLID( $a, b$ )
1  if  $b == 0$ 
2    return  $a$ 
3  else return EUCLID( $b, a \bmod b$ )

```

As an example of the running of EUCLID, consider the computation of $\text{gcd}(30, 21)$:

$$\begin{aligned}
 \text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\
 &= \text{EUCLID}(9, 3) \\
 &= \text{EUCLID}(3, 0) \\
 &= 3.
 \end{aligned}$$

This computation calls EUCLID recursively three times.

The correctness of EUCLID follows from Theorem 31.9 and the property that if the algorithm returns a in line 2, then $b = 0$, so that equation (31.9) implies that $\text{gcd}(a, b) = \text{gcd}(a, 0) = a$. The algorithm cannot recurse indefinitely, since the second argument strictly decreases in each recursive call and is always nonnegative. Therefore, EUCLID always terminates with the correct answer.

The running time of Euclid's algorithm

We analyze the worst-case running time of EUCLID as a function of the size of a and b . We assume with no loss of generality that $a > b \geq 0$. To justify this assumption, observe that if $b > a \geq 0$, then EUCLID(a, b) immediately makes the recursive call EUCLID(b, a). That is, if the first argument is less than the second argument, EUCLID spends one recursive call swapping its arguments and then proceeds. Similarly, if $b = a > 0$, the procedure terminates after one recursive call, since $a \bmod b = 0$.

The overall running time of EUCLID is proportional to the number of recursive calls it makes. Our analysis makes use of the Fibonacci numbers F_k , defined by the recurrence (3.22).

Lemma 31.10

If $a > b \geq 1$ and the call EUCLID(a, b) performs $k \geq 1$ recursive calls, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.

Proof The proof proceeds by induction on k . For the basis of the induction, let $k = 1$. Then, $b \geq 1 = F_2$, and since $a > b$, we must have $a \geq 2 = F_3$. Since $b > (a \bmod b)$, in each recursive call the first argument is strictly larger than the second; the assumption that $a > b$ therefore holds for each recursive call.

Assume inductively that the lemma holds if $k - 1$ recursive calls are made; we shall then prove that the lemma holds for k recursive calls. Since $k > 0$, we have $b > 0$, and $\text{EUCLID}(a, b)$ calls $\text{EUCLID}(b, a \bmod b)$ recursively, which in turn makes $k - 1$ recursive calls. The inductive hypothesis then implies that $b \geq F_{k+1}$ (thus proving part of the lemma), and $a \bmod b \geq F_k$. We have

$$\begin{aligned} b + (a \bmod b) &= b + (a - b \lfloor a/b \rfloor) \\ &\leq a, \end{aligned}$$

since $a > b > 0$ implies $\lfloor a/b \rfloor \geq 1$. Thus,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2}. \end{aligned} \quad \blacksquare$$

The following theorem is an immediate corollary of this lemma.

Theorem 31.11 (Lamé's theorem)

For any integer $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, then the call $\text{EUCLID}(a, b)$ makes fewer than k recursive calls. ■

We can show that the upper bound of Theorem 31.11 is the best possible by showing that the call $\text{EUCLID}(F_{k+1}, F_k)$ makes exactly $k - 1$ recursive calls when $k \geq 2$. We use induction on k . For the base case, $k = 2$, and the call $\text{EUCLID}(F_3, F_2)$ makes exactly one recursive call, to $\text{EUCLID}(1, 0)$. (We have to start at $k = 2$, because when $k = 1$ we do not have $F_2 > F_1$.) For the inductive step, assume that $\text{EUCLID}(F_k, F_{k-1})$ makes exactly $k - 2$ recursive calls. For $k > 2$, we have $F_k > F_{k-1} > 0$ and $F_{k+1} = F_k + F_{k-1}$, and so by Exercise 31.1-1, we have $F_{k+1} \bmod F_k = F_{k-1}$. Thus, we have

$$\begin{aligned} \gcd(F_{k+1}, F_k) &= \gcd(F_k, F_{k+1} \bmod F_k) \\ &= \gcd(F_k, F_{k-1}). \end{aligned}$$

Therefore, the call $\text{EUCLID}(F_{k+1}, F_k)$ recurses one time more than the call $\text{EUCLID}(F_k, F_{k-1})$, or exactly $k - 1$ times, meeting the upper bound of Theorem 31.11.

Since F_k is approximately $\phi^k / \sqrt{5}$, where ϕ is the golden ratio $(1 + \sqrt{5})/2$ defined by equation (3.24), the number of recursive calls in EUCLID is $O(\lg b)$. (See

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	—	3	1	0

Figure 31.1 How EXTENDED-EUCLID computes $\text{gcd}(99, 78)$. Each line shows one level of the recursion: the values of the inputs a and b , the computed value $\lfloor a/b \rfloor$, and the values d , x , and y returned. The triple (d, x, y) returned becomes the triple (d', x', y') used at the next higher level of recursion. The call EXTENDED-EUCLID(99, 78) returns $(3, -11, 14)$, so that $\text{gcd}(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

Exercise 31.2-5 for a tighter bound.) Therefore, if we call EUCLID on two β -bit numbers, then it performs $O(\beta)$ arithmetic operations and $O(\beta^3)$ bit operations (assuming that multiplication and division of β -bit numbers take $O(\beta^2)$ bit operations). Problem 31-2 asks you to show an $O(\beta^2)$ bound on the number of bit operations.

The extended form of Euclid's algorithm

We now rewrite Euclid's algorithm to compute additional useful information. Specifically, we extend the algorithm to compute the integer coefficients x and y such that

$$d = \text{gcd}(a, b) = ax + by. \quad (31.16)$$

Note that x and y may be zero or negative. We shall find these coefficients useful later for computing modular multiplicative inverses. The procedure EXTENDED-EUCLID takes as input a pair of nonnegative integers and returns a triple of the form (d, x, y) that satisfies equation (31.16).

EXTENDED-EUCLID(a, b)

```

1  if  $b == 0$ 
2      return  $(a, 1, 0)$ 
3  else  $(d', x', y') = \text{EXTENDED-EUCLID}(b, a \bmod b)$ 
4       $(d, x, y) = (d', y', x' - \lfloor a/b \rfloor y')$ 
5      return  $(d, x, y)$ 

```

Figure 31.1 illustrates how EXTENDED-EUCLID computes $\text{gcd}(99, 78)$.

The EXTENDED-EUCLID procedure is a variation of the EUCLID procedure. Line 1 is equivalent to the test " $b == 0$ " in line 1 of EUCLID. If $b = 0$, then

EXTENDED-EUCLID returns not only $d = a$ in line 2, but also the coefficients $x = 1$ and $y = 0$, so that $a = ax + by$. If $b \neq 0$, EXTENDED-EUCLID first computes (d', x', y') such that $d' = \gcd(b, a \bmod b)$ and

$$d' = bx' + (a \bmod b)y'. \quad (31.17)$$

As for EUCLID, we have in this case $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. To obtain x and y such that $d = ax + by$, we start by rewriting equation (31.17) using the equation $d = d'$ and equation (3.8):

$$\begin{aligned} d &= bx' + (a - b \lfloor a/b \rfloor)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y'). \end{aligned}$$

Thus, choosing $x = y'$ and $y = x' - \lfloor a/b \rfloor y'$ satisfies the equation $d = ax + by$, proving the correctness of EXTENDED-EUCLID.

Since the number of recursive calls made in EUCLID is equal to the number of recursive calls made in EXTENDED-EUCLID, the running times of EUCLID and EXTENDED-EUCLID are the same, to within a constant factor. That is, for $a > b > 0$, the number of recursive calls is $O(\lg b)$.

Exercises

31.2-1

Prove that equations (31.11) and (31.12) imply equation (31.13).

31.2-2

Compute the values (d, x, y) that the call EXTENDED-EUCLID(899, 493) returns.

31.2-3

Prove that for all integers a, k , and n ,

$$\gcd(a, n) = \gcd(a + kn, n).$$

31.2-4

Rewrite EUCLID in an iterative form that uses only a constant amount of memory (that is, stores only a constant number of integer values).

31.2-5

If $a > b \geq 0$, show that the call EUCLID(a, b) makes at most $1 + \log_\phi b$ recursive calls. Improve this bound to $1 + \log_\phi(b / \gcd(a, b))$.

31.2-6

What does EXTENDED-EUCLID(F_{k+1}, F_k) return? Prove your answer correct.

31.2-7

Define the gcd function for more than two arguments by the recursive equation $\gcd(a_0, a_1, \dots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \dots, a_n))$. Show that the gcd function returns the same answer independent of the order in which its arguments are specified. Also show how to find integers x_0, x_1, \dots, x_n such that $\gcd(a_0, a_1, \dots, a_n) = a_0x_0 + a_1x_1 + \dots + a_nx_n$. Show that the number of divisions performed by your algorithm is $O(n + \lg(\max\{a_0, a_1, \dots, a_n\}))$.

31.2-8

Define $\text{lcm}(a_1, a_2, \dots, a_n)$ to be the *least common multiple* of the n integers a_1, a_2, \dots, a_n , that is, the smallest nonnegative integer that is a multiple of each a_i . Show how to compute $\text{lcm}(a_1, a_2, \dots, a_n)$ efficiently using the (two-argument) gcd operation as a subroutine.

31.2-9

Prove that n_1, n_2, n_3 , and n_4 are pairwise relatively prime if and only if

$$\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1.$$

More generally, show that n_1, n_2, \dots, n_k are pairwise relatively prime if and only if a set of $\lceil \lg k \rceil$ pairs of numbers derived from the n_i are relatively prime.

31.3 Modular arithmetic

Informally, we can think of modular arithmetic as arithmetic as usual over the integers, except that if we are working modulo n , then every result x is replaced by the element of $\{0, 1, \dots, n-1\}$ that is equivalent to x , modulo n (that is, x is replaced by $x \bmod n$). This informal model suffices if we stick to the operations of addition, subtraction, and multiplication. A more formal model for modular arithmetic, which we now give, is best described within the framework of group theory.

Finite groups

A **group** (S, \oplus) is a set S together with a binary operation \oplus defined on S for which the following properties hold:

1. **Closure:** For all $a, b \in S$, we have $a \oplus b \in S$.
2. **Identity:** There exists an element $e \in S$, called the *identity* of the group, such that $e \oplus a = a \oplus e = a$ for all $a \in S$.
3. **Associativity:** For all $a, b, c \in S$, we have $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

4. **Inverses:** For each $a \in S$, there exists a unique element $b \in S$, called the *inverse* of a , such that $a \oplus b = b \oplus a = e$.

As an example, consider the familiar group $(\mathbb{Z}, +)$ of the integers \mathbb{Z} under the operation of addition: 0 is the identity, and the inverse of a is $-a$. If a group (S, \oplus) satisfies the *commutative law* $a \oplus b = b \oplus a$ for all $a, b \in S$, then it is an *abelian group*. If a group (S, \oplus) satisfies $|S| < \infty$, then it is a *finite group*.

The groups defined by modular addition and multiplication

We can form two finite abelian groups by using addition and multiplication modulo n , where n is a positive integer. These groups are based on the equivalence classes of the integers modulo n , defined in Section 31.1.

To define a group on \mathbb{Z}_n , we need to have suitable binary operations, which we obtain by redefining the ordinary operations of addition and multiplication. We can easily define addition and multiplication operations for \mathbb{Z}_n , because the equivalence class of two integers uniquely determines the equivalence class of their sum or product. That is, if $a \equiv a' \pmod{n}$ and $b \equiv b' \pmod{n}$, then

$$\begin{aligned} a + b &\equiv a' + b' \pmod{n}, \\ ab &\equiv a'b' \pmod{n}. \end{aligned}$$

Thus, we define addition and multiplication modulo n , denoted $+_n$ and \cdot_n , by

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n, \\ [a]_n \cdot_n [b]_n &= [ab]_n. \end{aligned} \tag{31.18}$$

(We can define subtraction similarly on \mathbb{Z}_n by $[a]_n -_n [b]_n = [a - b]_n$, but division is more complicated, as we shall see.) These facts justify the common and convenient practice of using the smallest nonnegative element of each equivalence class as its representative when performing computations in \mathbb{Z}_n . We add, subtract, and multiply as usual on the representatives, but we replace each result x by the representative of its class, that is, by $x \bmod n$.

Using this definition of addition modulo n , we define the *additive group modulo n* as $(\mathbb{Z}_n, +_n)$. The size of the additive group modulo n is $|\mathbb{Z}_n| = n$. Figure 31.2(a) gives the operation table for the group $(\mathbb{Z}_6, +_6)$.

Theorem 31.12

The system $(\mathbb{Z}_n, +_n)$ is a finite abelian group.

Proof Equation (31.18) shows that $(\mathbb{Z}_n, +_n)$ is closed. Associativity and commutativity of $+_n$ follow from the associativity and commutativity of $+$:

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

\cdot_{15}	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(a)

(b)

Figure 31.2 Two finite groups. Equivalence classes are denoted by their representative elements. (a) The group $(\mathbb{Z}_6, +_6)$. (b) The group $(\mathbb{Z}_{15}^*, \cdot_{15})$.

$$\begin{aligned}
 ([a]_n +_n [b]_n) +_n [c]_n &= [a + b]_n +_n [c]_n \\
 &= [(a + b) + c]_n \\
 &= [a + (b + c)]_n \\
 &= [a]_n +_n [b + c]_n \\
 &= [a]_n +_n ([b]_n +_n [c]_n) ,
 \end{aligned}$$

$$\begin{aligned}
 [a]_n +_n [b]_n &= [a + b]_n \\
 &= [b + a]_n \\
 &= [b]_n +_n [a]_n .
 \end{aligned}$$

The identity element of $(\mathbb{Z}_n, +_n)$ is 0 (that is, $[0]_n$). The (additive) inverse of an element a (that is, of $[a]_n$) is the element $-a$ (that is, $[-a]_n$ or $[n - a]_n$), since $[a]_n +_n [-a]_n = [a - a]_n = [0]_n$. ■

Using the definition of multiplication modulo n , we define the **multiplicative group modulo n** as $(\mathbb{Z}_n^*, \cdot_n)$. The elements of this group are the set \mathbb{Z}_n^* of elements in \mathbb{Z}_n that are relatively prime to n , so that each one has a unique inverse, modulo n :

$$\mathbb{Z}_n^* = \{[a]_n \in \mathbb{Z}_n : \gcd(a, n) = 1\} .$$

To see that \mathbb{Z}_n^* is well defined, note that for $0 \leq a < n$, we have $a \equiv (a + kn) \pmod{n}$ for all integers k . By Exercise 31.2-3, therefore, $\gcd(a, n) = 1$ implies $\gcd(a + kn, n) = 1$ for all integers k . Since $[a]_n = \{a + kn : k \in \mathbb{Z}\}$, the set \mathbb{Z}_n^* is well defined. An example of such a group is

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\} ,$$

where the group operation is multiplication modulo 15. (Here we denote an element $[a]_{15}$ as a ; for example, we denote $[7]_{15}$ as 7.) Figure 31.2(b) shows the group $(\mathbb{Z}_{15}^*, \cdot_{15})$. For example, $8 \cdot 11 \equiv 13 \pmod{15}$, working in \mathbb{Z}_{15}^* . The identity for this group is 1.

Theorem 31.13

The system $(\mathbb{Z}_n^*, \cdot_n)$ is a finite abelian group.

Proof Theorem 31.6 implies that $(\mathbb{Z}_n^*, \cdot_n)$ is closed. Associativity and commutativity can be proved for \cdot_n as they were for $+_n$ in the proof of Theorem 31.12. The identity element is $[1]_n$. To show the existence of inverses, let a be an element of \mathbb{Z}_n^* and let (d, x, y) be returned by EXTENDED-EUCLID(a, n). Then, $d = 1$, since $a \in \mathbb{Z}_n^*$, and

$$ax + ny = 1 \tag{31.19}$$

or, equivalently,

$$ax \equiv 1 \pmod{n}.$$

Thus, $[x]_n$ is a multiplicative inverse of $[a]_n$, modulo n . Furthermore, we claim that $[x]_n \in \mathbb{Z}_n^*$. To see why, equation (31.19) demonstrates that the smallest positive linear combination of x and n must be 1. Therefore, Theorem 31.2 implies that $\gcd(x, n) = 1$. We defer the proof that inverses are uniquely defined until Corollary 31.26. ■

As an example of computing multiplicative inverses, suppose that $a = 5$ and $n = 11$. Then EXTENDED-EUCLID(a, n) returns $(d, x, y) = (1, -2, 1)$, so that $1 = 5 \cdot (-2) + 11 \cdot 1$. Thus, $[-2]_{11}$ (i.e., $[9]_{11}$) is the multiplicative inverse of $[5]_{11}$.

When working with the groups $(\mathbb{Z}_n, +_n)$ and $(\mathbb{Z}_n^*, \cdot_n)$ in the remainder of this chapter, we follow the convenient practice of denoting equivalence classes by their representative elements and denoting the operations $+_n$ and \cdot_n by the usual arithmetic notations $+$ and \cdot (or juxtaposition, so that $ab = a \cdot b$) respectively. Also, equivalences modulo n may also be interpreted as equations in \mathbb{Z}_n . For example, the following two statements are equivalent:

$$\begin{aligned} ax &\equiv b \pmod{n}, \\ [a]_n \cdot_n [x]_n &= [b]_n. \end{aligned}$$

As a further convenience, we sometimes refer to a group (S, \oplus) merely as S when the operation \oplus is understood from context. We may thus refer to the groups $(\mathbb{Z}_n, +_n)$ and $(\mathbb{Z}_n^*, \cdot_n)$ as \mathbb{Z}_n and \mathbb{Z}_n^* , respectively.

We denote the (multiplicative) inverse of an element a by $(a^{-1} \bmod n)$. Division in \mathbb{Z}_n^* is defined by the equation $a/b \equiv ab^{-1} \pmod{n}$. For example, in \mathbb{Z}_{15}^*

we have that $7^{-1} \equiv 13 \pmod{15}$, since $7 \cdot 13 = 91 \equiv 1 \pmod{15}$, so that $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

The size of \mathbb{Z}_n^* is denoted $\phi(n)$. This function, known as *Euler's phi function*, satisfies the equation

$$\phi(n) = n \prod_{p: p \text{ is prime and } p \mid n} \left(1 - \frac{1}{p}\right), \quad (31.20)$$

so that p runs over all the primes dividing n (including n itself, if n is prime). We shall not prove this formula here. Intuitively, we begin with a list of the n remainders $\{0, 1, \dots, n-1\}$ and then, for each prime p that divides n , cross out every multiple of p in the list. For example, since the prime divisors of 45 are 3 and 5,

$$\begin{aligned} \phi(45) &= 45 \left(1 - \frac{1}{3}\right) \left(1 - \frac{1}{5}\right) \\ &= 45 \left(\frac{2}{3}\right) \left(\frac{4}{5}\right) \\ &= 24. \end{aligned}$$

If p is prime, then $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$, and

$$\begin{aligned} \phi(p) &= p \left(1 - \frac{1}{p}\right) \\ &= p - 1. \end{aligned} \quad (31.21)$$

If n is composite, then $\phi(n) < n - 1$, although it can be shown that

$$\phi(n) > \frac{n}{e^\gamma \ln \ln n + \frac{3}{\ln \ln n}} \quad (31.22)$$

for $n \geq 3$, where $\gamma = 0.5772156649\dots$ is *Euler's constant*. A somewhat simpler (but looser) lower bound for $n > 5$ is

$$\phi(n) > \frac{n}{6 \ln \ln n}. \quad (31.23)$$

The lower bound (31.22) is essentially the best possible, since

$$\liminf_{n \rightarrow \infty} \frac{\phi(n)}{n / \ln \ln n} = e^{-\gamma}. \quad (31.24)$$

Subgroups

If (S, \oplus) is a group, $S' \subseteq S$, and (S', \oplus) is also a group, then (S', \oplus) is a *subgroup* of (S, \oplus) . For example, the even integers form a subgroup of the integers under the operation of addition. The following theorem provides a useful tool for recognizing subgroups.

Theorem 31.14 (A nonempty closed subset of a finite group is a subgroup)

If (S, \oplus) is a finite group and S' is any nonempty subset of S such that $a \oplus b \in S'$ for all $a, b \in S'$, then (S', \oplus) is a subgroup of (S, \oplus) .

Proof We leave the proof as Exercise 31.3-3. ■

For example, the set $\{0, 2, 4, 6\}$ forms a subgroup of \mathbb{Z}_8 , since it is nonempty and closed under the operation $+$ (that is, it is closed under $+_8$).

The following theorem provides an extremely useful constraint on the size of a subgroup; we omit the proof.

Theorem 31.15 (Lagrange's theorem)

If (S, \oplus) is a finite group and (S', \oplus) is a subgroup of (S, \oplus) , then $|S'|$ is a divisor of $|S|$. ■

A subgroup S' of a group S is a **proper** subgroup if $S' \neq S$. We shall use the following corollary in our analysis in Section 31.8 of the Miller-Rabin primality test procedure.

Corollary 31.16

If S' is a proper subgroup of a finite group S , then $|S'| \leq |S|/2$. ■

Subgroups generated by an element

Theorem 31.14 gives us an easy way to produce a subgroup of a finite group (S, \oplus) : choose an element a and take all elements that can be generated from a using the group operation. Specifically, define $a^{(k)}$ for $k \geq 1$ by

$$a^{(k)} = \bigoplus_{i=1}^k a = \underbrace{a \oplus a \oplus \cdots \oplus a}_k .$$

For example, if we take $a = 2$ in the group \mathbb{Z}_6 , the sequence $a^{(1)}, a^{(2)}, a^{(3)}, \dots$ is $2, 4, 0, 2, 4, 0, 2, 4, 0, \dots$.

In the group \mathbb{Z}_n , we have $a^{(k)} = ka \bmod n$, and in the group \mathbb{Z}_n^* , we have $a^{(k)} = a^k \bmod n$. We define the **subgroup generated by a** , denoted $\langle a \rangle$ or $(\langle a \rangle, \oplus)$, by

$$\langle a \rangle = \{a^{(k)} : k \geq 1\} .$$

We say that a **generates** the subgroup $\langle a \rangle$ or that a is a **generator** of $\langle a \rangle$. Since S is finite, $\langle a \rangle$ is a finite subset of S , possibly including all of S . Since the associativity of \oplus implies

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)},$$

$\langle a \rangle$ is closed and therefore, by Theorem 31.14, $\langle a \rangle$ is a subgroup of S . For example, in \mathbb{Z}_6 , we have

$$\begin{aligned}\langle 0 \rangle &= \{0\}, \\ \langle 1 \rangle &= \{0, 1, 2, 3, 4, 5\}, \\ \langle 2 \rangle &= \{0, 2, 4\}.\end{aligned}$$

Similarly, in \mathbb{Z}_7^* , we have

$$\begin{aligned}\langle 1 \rangle &= \{1\}, \\ \langle 2 \rangle &= \{1, 2, 4\}, \\ \langle 3 \rangle &= \{1, 2, 3, 4, 5, 6\}.\end{aligned}$$

The **order** of a (in the group S), denoted $\text{ord}(a)$, is defined as the smallest positive integer t such that $a^{(t)} = e$.

Theorem 31.17

For any finite group (S, \oplus) and any $a \in S$, the order of a is equal to the size of the subgroup it generates, or $\text{ord}(a) = |\langle a \rangle|$.

Proof Let $t = \text{ord}(a)$. Since $a^{(t)} = e$ and $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ for $k \geq 1$, if $i > t$, then $a^{(i)} = a^{(j)}$ for some $j < i$. Thus, as we generate elements by a , we see no new elements after $a^{(t)}$. Thus, $\langle a \rangle = \{a^{(1)}, a^{(2)}, \dots, a^{(t)}\}$, and so $|\langle a \rangle| \leq t$. To show that $|\langle a \rangle| \geq t$, we show that each element of the sequence $a^{(1)}, a^{(2)}, \dots, a^{(t)}$ is distinct. Suppose for the purpose of contradiction that $a^{(i)} = a^{(j)}$ for some i and j satisfying $1 \leq i < j \leq t$. Then, $a^{(i+k)} = a^{(j+k)}$ for $k \geq 0$. But this equality implies that $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, a contradiction, since $i + (t - j) < t$ but t is the least positive value such that $a^{(t)} = e$. Therefore, each element of the sequence $a^{(1)}, a^{(2)}, \dots, a^{(t)}$ is distinct, and $|\langle a \rangle| \geq t$. We conclude that $\text{ord}(a) = |\langle a \rangle|$. ■

Corollary 31.18

The sequence $a^{(1)}, a^{(2)}, \dots$ is periodic with period $t = \text{ord}(a)$; that is, $a^{(i)} = a^{(j)}$ if and only if $i \equiv j \pmod{t}$. ■

Consistent with the above corollary, we define $a^{(0)}$ as e and $a^{(i)}$ as $a^{(i \bmod t)}$, where $t = \text{ord}(a)$, for all integers i .

Corollary 31.19

If (S, \oplus) is a finite group with identity e , then for all $a \in S$,

$$a^{(|S|)} = e.$$

Proof Lagrange's theorem (Theorem 31.15) implies that $\text{ord}(a) \mid |S|$, and so $|S| \equiv 0 \pmod{t}$, where $t = \text{ord}(a)$. Therefore, $a^{(|S|)} = a^{(0)} = e$. ■

Exercises

31.3-1

Draw the group operation tables for the groups $(\mathbb{Z}_4, +_4)$ and $(\mathbb{Z}_5^*, \cdot_5)$. Show that these groups are isomorphic by exhibiting a one-to-one correspondence α between their elements such that $a + b \equiv c \pmod{4}$ if and only if $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod{5}$.

31.3-2

List all subgroups of \mathbb{Z}_9 and of \mathbb{Z}_{13}^* .

31.3-3

Prove Theorem 31.14.

31.3-4

Show that if p is prime and e is a positive integer, then

$$\phi(p^e) = p^{e-1}(p-1).$$

31.3-5

Show that for any integer $n > 1$ and for any $a \in \mathbb{Z}_n^*$, the function $f_a : \mathbb{Z}_n^* \rightarrow \mathbb{Z}_n^*$ defined by $f_a(x) = ax \pmod{n}$ is a permutation of \mathbb{Z}_n^* .

31.4 Solving modular linear equations

We now consider the problem of finding solutions to the equation

$$ax \equiv b \pmod{n}, \tag{31.25}$$

where $a > 0$ and $n > 0$. This problem has several applications; for example, we shall use it as part of the procedure for finding keys in the RSA public-key cryptosystem in Section 31.7. We assume that a , b , and n are given, and we wish to find all values of x , modulo n , that satisfy equation (31.25). The equation may have zero, one, or more than one such solution.

Let $\langle a \rangle$ denote the subgroup of \mathbb{Z}_n generated by a . Since $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \pmod{n} : x > 0\}$, equation (31.25) has a solution if and only if $[b] \in \langle a \rangle$. Lagrange's theorem (Theorem 31.15) tells us that $|\langle a \rangle|$ must be a divisor of n . The following theorem gives us a precise characterization of $\langle a \rangle$.

Theorem 31.20

For any positive integers a and n , if $d = \gcd(a, n)$, then

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \dots, ((n/d) - 1)d\} \quad (31.26)$$

in \mathbb{Z}_n , and thus

$$|\langle a \rangle| = n/d .$$

Proof We begin by showing that $d \in \langle a \rangle$. Recall that EXTENDED-EUCLID(a, n) produces integers x' and y' such that $ax' + ny' = d$. Thus, $ax' \equiv d \pmod{n}$, so that $d \in \langle a \rangle$. In other words, d is a multiple of a in \mathbb{Z}_n .

Since $d \in \langle a \rangle$, it follows that every multiple of d belongs to $\langle a \rangle$, because any multiple of a multiple of a is itself a multiple of a . Thus, $\langle a \rangle$ contains every element in $\{0, d, 2d, \dots, ((n/d) - 1)d\}$. That is, $\langle d \rangle \subseteq \langle a \rangle$.

We now show that $\langle a \rangle \subseteq \langle d \rangle$. If $m \in \langle a \rangle$, then $m = ax \pmod{n}$ for some integer x , and so $m = ax + ny$ for some integer y . However, $d \mid a$ and $d \mid n$, and so $d \mid m$ by equation (31.4). Therefore, $m \in \langle d \rangle$.

Combining these results, we have that $\langle a \rangle = \langle d \rangle$. To see that $|\langle a \rangle| = n/d$, observe that there are exactly n/d multiples of d between 0 and $n - 1$, inclusive. ■

Corollary 31.21

The equation $ax \equiv b \pmod{n}$ is solvable for the unknown x if and only if $d \mid b$, where $d = \gcd(a, n)$.

Proof The equation $ax \equiv b \pmod{n}$ is solvable if and only if $[b] \in \langle a \rangle$, which is the same as saying

$$(b \pmod{n}) \in \{0, d, 2d, \dots, ((n/d) - 1)d\} ,$$

by Theorem 31.20. If $0 \leq b < n$, then $b \in \langle a \rangle$ if and only if $d \mid b$, since the members of $\langle a \rangle$ are precisely the multiples of d . If $b < 0$ or $b \geq n$, the corollary then follows from the observation that $d \mid b$ if and only if $d \mid (b \pmod{n})$, since b and $b \pmod{n}$ differ by a multiple of n , which is itself a multiple of d . ■

Corollary 31.22

The equation $ax \equiv b \pmod{n}$ either has d distinct solutions modulo n , where $d = \gcd(a, n)$, or it has no solutions.

Proof If $ax \equiv b \pmod{n}$ has a solution, then $b \in \langle a \rangle$. By Theorem 31.17, $\text{ord}(a) = |\langle a \rangle|$, and so Corollary 31.18 and Theorem 31.20 imply that the sequence $ai \pmod{n}$, for $i = 0, 1, \dots$, is periodic with period $|\langle a \rangle| = n/d$. If $b \in \langle a \rangle$, then b appears exactly d times in the sequence $ai \pmod{n}$, for $i = 0, 1, \dots, n - 1$, since

the length- (n/d) block of values $\langle a \rangle$ repeats exactly d times as i increases from 0 to $n-1$. The indices x of the d positions for which $ax \bmod n = b$ are the solutions of the equation $ax \equiv b \pmod{n}$. ■

Theorem 31.23

Let $d = \gcd(a, n)$, and suppose that $d = ax' + ny'$ for some integers x' and y' (for example, as computed by EXTENDED-EUCLID). If $d \mid b$, then the equation $ax \equiv b \pmod{n}$ has as one of its solutions the value x_0 , where

$$x_0 = x'(b/d) \bmod n .$$

Proof We have

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} \\ &\equiv d(b/d) \pmod{n} && \text{(because } ax' \equiv d \pmod{n}\text{)} \\ &\equiv b \pmod{n} , \end{aligned}$$

and thus x_0 is a solution to $ax \equiv b \pmod{n}$. ■

Theorem 31.24

Suppose that the equation $ax \equiv b \pmod{n}$ is solvable (that is, $d \mid b$, where $d = \gcd(a, n)$) and that x_0 is any solution to this equation. Then, this equation has exactly d distinct solutions, modulo n , given by $x_i = x_0 + i(n/d)$ for $i = 0, 1, \dots, d-1$.

Proof Because $n/d > 0$ and $0 \leq i(n/d) < n$ for $i = 0, 1, \dots, d-1$, the values x_0, x_1, \dots, x_{d-1} are all distinct, modulo n . Since x_0 is a solution of $ax \equiv b \pmod{n}$, we have $ax_0 \bmod n \equiv b \pmod{n}$. Thus, for $i = 0, 1, \dots, d-1$, we have

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n && \text{(because } d \mid a \text{ implies that } ain/d \text{ is a multiple of } n\text{)} \\ &\equiv b \pmod{n} , \end{aligned}$$

and hence $ax_i \equiv b \pmod{n}$, making x_i a solution, too. By Corollary 31.22, the equation $ax \equiv b \pmod{n}$ has exactly d solutions, so that x_0, x_1, \dots, x_{d-1} must be all of them. ■

We have now developed the mathematics needed to solve the equation $ax \equiv b \pmod{n}$; the following algorithm prints all solutions to this equation. The inputs a and n are arbitrary positive integers, and b is an arbitrary integer.

MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)

```

1  ( $d, x', y'$ ) = EXTENDED-EUCLID( $a, n$ )
2  if  $d \mid b$ 
3       $x_0 = x'(b/d) \bmod n$ 
4      for  $i = 0$  to  $d - 1$ 
5          print ( $x_0 + i(n/d)$ ) mod  $n$ 
6  else print “no solutions”

```

As an example of the operation of this procedure, consider the equation $14x \equiv 30 \pmod{100}$ (here, $a = 14$, $b = 30$, and $n = 100$). Calling EXTENDED-EUCLID in line 1, we obtain $(d, x', y') = (2, -7, 1)$. Since $2 \mid 30$, lines 3–5 execute. Line 3 computes $x_0 = (-7)(15) \bmod 100 = 95$. The loop on lines 4–5 prints the two solutions 95 and 45.

The procedure MODULAR-LINEAR-EQUATION-SOLVER works as follows. Line 1 computes $d = \gcd(a, n)$, along with two values x' and y' such that $d = ax' + ny'$, demonstrating that x' is a solution to the equation $ax' \equiv d \pmod{n}$. If d does not divide b , then the equation $ax \equiv b \pmod{n}$ has no solution, by Corollary 31.21. Line 2 checks to see whether $d \mid b$; if not, line 6 reports that there are no solutions. Otherwise, line 3 computes a solution x_0 to $ax \equiv b \pmod{n}$, in accordance with Theorem 31.23. Given one solution, Theorem 31.24 states that adding multiples of (n/d) , modulo n , yields the other $d - 1$ solutions. The **for** loop of lines 4–5 prints out all d solutions, beginning with x_0 and spaced n/d apart, modulo n .

MODULAR-LINEAR-EQUATION-SOLVER performs $O(\lg n + \gcd(a, n))$ arithmetic operations, since EXTENDED-EUCLID performs $O(\lg n)$ arithmetic operations, and each iteration of the **for** loop of lines 4–5 performs a constant number of arithmetic operations.

The following corollaries of Theorem 31.24 give specializations of particular interest.

Corollary 31.25

For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax \equiv b \pmod{n}$ has a unique solution, modulo n . ■

If $b = 1$, a common case of considerable interest, the x we are looking for is a *multiplicative inverse* of a , modulo n .

Corollary 31.26

For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax \equiv 1 \pmod{n}$ has a unique solution, modulo n . Otherwise, it has no solution. ■

Thanks to Corollary 31.26, we can use the notation $a^{-1} \pmod n$ to refer to *the* multiplicative inverse of a , modulo n , when a and n are relatively prime. If $\gcd(a, n) = 1$, then the unique solution to the equation $ax \equiv 1 \pmod n$ is the integer x returned by EXTENDED-EUCLID, since the equation

$$\gcd(a, n) = 1 = ax + ny$$

implies $ax \equiv 1 \pmod n$. Thus, we can compute $a^{-1} \pmod n$ efficiently using EXTENDED-EUCLID.

Exercises

31.4-1

Find all solutions to the equation $35x \equiv 10 \pmod{50}$.

31.4-2

Prove that the equation $ax \equiv ay \pmod n$ implies $x \equiv y \pmod n$ whenever $\gcd(a, n) = 1$. Show that the condition $\gcd(a, n) = 1$ is necessary by supplying a counterexample with $\gcd(a, n) > 1$.

31.4-3

Consider the following change to line 3 of the procedure MODULAR-LINEAR-EQUATION-SOLVER:

$$3 \quad x_0 = x'(b/d) \pmod{(n/d)}$$

Will this work? Explain why or why not.

31.4-4 ★

Let p be prime and $f(x) \equiv f_0 + f_1x + \cdots + f_t x^t \pmod p$ be a polynomial of degree t , with coefficients f_i drawn from \mathbb{Z}_p . We say that $a \in \mathbb{Z}_p$ is a **zero** of f if $f(a) \equiv 0 \pmod p$. Prove that if a is a zero of f , then $f(x) \equiv (x - a)g(x) \pmod p$ for some polynomial $g(x)$ of degree $t - 1$. Prove by induction on t that if p is prime, then a polynomial $f(x)$ of degree t can have at most t distinct zeros modulo p .

31.5 The Chinese remainder theorem

Around A.D. 100, the Chinese mathematician Sun-Tsū solved the problem of finding those integers x that leave remainders 2, 3, and 2 when divided by 3, 5, and 7 respectively. One such solution is $x = 23$; all solutions are of the form $23 + 105k$

for arbitrary integers k . The “Chinese remainder theorem” provides a correspondence between a system of equations modulo a set of pairwise relatively prime moduli (for example, 3, 5, and 7) and an equation modulo their product (for example, 105).

The Chinese remainder theorem has two major applications. Let the integer n be factored as $n = n_1 n_2 \cdots n_k$, where the factors n_i are pairwise relatively prime. First, the Chinese remainder theorem is a descriptive “structure theorem” that describes the structure of \mathbb{Z}_n as identical to that of the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ with componentwise addition and multiplication modulo n_i in the i th component. Second, this description helps us to design efficient algorithms, since working in each of the systems \mathbb{Z}_{n_i} can be more efficient (in terms of bit operations) than working modulo n .

Theorem 31.27 (Chinese remainder theorem)

Let $n = n_1 n_2 \cdots n_k$, where the n_i are pairwise relatively prime. Consider the correspondence

$$a \leftrightarrow (a_1, a_2, \dots, a_k), \quad (31.27)$$

where $a \in \mathbb{Z}_n$, $a_i \in \mathbb{Z}_{n_i}$, and

$$a_i = a \bmod n_i$$

for $i = 1, 2, \dots, k$. Then, mapping (31.27) is a one-to-one correspondence (bijection) between \mathbb{Z}_n and the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$. Operations performed on the elements of \mathbb{Z}_n can be equivalently performed on the corresponding k -tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$a \leftrightarrow (a_1, a_2, \dots, a_k),$$

$$b \leftrightarrow (b_1, b_2, \dots, b_k),$$

then

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \dots, (a_k + b_k) \bmod n_k), \quad (31.28)$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \dots, (a_k - b_k) \bmod n_k), \quad (31.29)$$

$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \dots, a_k b_k \bmod n_k). \quad (31.30)$$

Proof Transforming between the two representations is fairly straightforward. Going from a to (a_1, a_2, \dots, a_k) is quite easy and requires only k “mod” operations.

Computing a from inputs (a_1, a_2, \dots, a_k) is a bit more complicated. We begin by defining $m_i = n/n_i$ for $i = 1, 2, \dots, k$; thus m_i is the product of all of the n_j ’s other than n_i : $m_i = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k$. We next define

$$c_i = m_i(m_i^{-1} \bmod n_i) \quad (31.31)$$

for $i = 1, 2, \dots, k$. Equation (31.31) is always well defined: since m_i and n_i are relatively prime (by Theorem 31.6), Corollary 31.26 guarantees that $m_i^{-1} \bmod n_i$ exists. Finally, we can compute a as a function of a_1, a_2, \dots, a_k as follows:

$$a \equiv (a_1c_1 + a_2c_2 + \dots + a_kc_k) \pmod{n}. \quad (31.32)$$

We now show that equation (31.32) ensures that $a \equiv a_i \pmod{n_i}$ for $i = 1, 2, \dots, k$. Note that if $j \neq i$, then $m_j \equiv 0 \pmod{n_i}$, which implies that $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Note also that $c_i \equiv 1 \pmod{n_i}$, from equation (31.31). We thus have the appealing and useful correspondence

$$c_i \leftrightarrow (0, 0, \dots, 0, 1, 0, \dots, 0),$$

a vector that has 0s everywhere except in the i th coordinate, where it has a 1; the c_i thus form a “basis” for the representation, in a certain sense. For each i , therefore, we have

$$\begin{aligned} a &\equiv a_i c_i && \pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i}, \end{aligned}$$

which is what we wished to show: our method of computing a from the a_i 's produces a result a that satisfies the constraints $a \equiv a_i \pmod{n_i}$ for $i = 1, 2, \dots, k$. The correspondence is one-to-one, since we can transform in both directions. Finally, equations (31.28)–(31.30) follow directly from Exercise 31.1-7, since $x \bmod n_i = (x \bmod n) \bmod n_i$ for any x and $i = 1, 2, \dots, k$. ■

We shall use the following corollaries later in this chapter.

Corollary 31.28

If n_1, n_2, \dots, n_k are pairwise relatively prime and $n = n_1 n_2 \dots n_k$, then for any integers a_1, a_2, \dots, a_k , the set of simultaneous equations

$$x \equiv a_i \pmod{n_i},$$

for $i = 1, 2, \dots, k$, has a unique solution modulo n for the unknown x . ■

Corollary 31.29

If n_1, n_2, \dots, n_k are pairwise relatively prime and $n = n_1 n_2 \dots n_k$, then for all integers x and a ,

$$x \equiv a \pmod{n_i}$$

for $i = 1, 2, \dots, k$ if and only if

$$x \equiv a \pmod{n}. \quad \blacksquare$$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Figure 31.3 An illustration of the Chinese remainder theorem for $n_1 = 5$ and $n_2 = 13$. For this example, $c_1 = 26$ and $c_2 = 40$. In row i , column j is shown the value of a , modulo 65, such that $a \bmod 5 = i$ and $a \bmod 13 = j$. Note that row 0, column 0 contains a 0. Similarly, row 4, column 12 contains a 64 (equivalent to -1). Since $c_1 = 26$, moving down a row increases a by 26. Similarly, $c_2 = 40$ means that moving right by a column increases a by 40. Increasing a by 1 corresponds to moving diagonally downward and to the right, wrapping around from the bottom to the top and from the right to the left.

As an example of the application of the Chinese remainder theorem, suppose we are given the two equations

$$a \equiv 2 \pmod{5},$$

$$a \equiv 3 \pmod{13},$$

so that $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, and $n_2 = m_1 = 13$, and we wish to compute $a \bmod 65$, since $n = n_1 n_2 = 65$. Because $13^{-1} \equiv 2 \pmod{5}$ and $5^{-1} \equiv 8 \pmod{13}$, we have

$$c_1 = 13(2 \bmod 5) = 26,$$

$$c_2 = 5(8 \bmod 13) = 40,$$

and

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &\equiv 52 + 120 \pmod{65} \\ &\equiv 42 \pmod{65}. \end{aligned}$$

See Figure 31.3 for an illustration of the Chinese remainder theorem, modulo 65.

Thus, we can work modulo n by working modulo n directly or by working in the transformed representation using separate modulo n_i computations, as convenient. The computations are entirely equivalent.

Exercises

31.5-1

Find all solutions to the equations $x \equiv 4 \pmod{5}$ and $x \equiv 5 \pmod{11}$.

31.5-2

Find all integers x that leave remainders 1, 2, 3 when divided by 9, 8, 7 respectively.

31.5-3

Argue that, under the definitions of Theorem 31.27, if $\gcd(a, n) = 1$, then

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \dots, (a_k^{-1} \bmod n_k)) .$$

31.5-4

Under the definitions of Theorem 31.27, prove that for any polynomial f , the number of roots of the equation $f(x) \equiv 0 \pmod{n}$ equals the product of the number of roots of each of the equations $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}$, \dots , $f(x) \equiv 0 \pmod{n_k}$.

31.6 Powers of an element

Just as we often consider the multiples of a given element a , modulo n , we consider the sequence of powers of a , modulo n , where $a \in \mathbb{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \dots, \tag{31.33}$$

modulo n . Indexing from 0, the 0th value in this sequence is $a^0 \bmod n = 1$, and the i th value is $a^i \bmod n$. For example, the powers of 3 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	...

whereas the powers of 2 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	...

In this section, let $\langle a \rangle$ denote the subgroup of \mathbb{Z}_n^* generated by a by repeated multiplication, and let $\text{ord}_n(a)$ (the “order of a , modulo n ”) denote the order of a in \mathbb{Z}_n^* . For example, $\langle 2 \rangle = \{1, 2, 4\}$ in \mathbb{Z}_7^* , and $\text{ord}_7(2) = 3$. Using the definition of the Euler phi function $\phi(n)$ as the size of \mathbb{Z}_n^* (see Section 31.3), we now translate Corollary 31.19 into the notation of \mathbb{Z}_n^* to obtain Euler’s theorem and specialize it to \mathbb{Z}_p^* , where p is prime, to obtain Fermat’s theorem.

Theorem 31.30 (Euler’s theorem)

For any integer $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ for all } a \in \mathbb{Z}_n^* . \quad \blacksquare$$

Theorem 31.31 (Fermat's theorem)

If p is prime, then

$$a^{p-1} \equiv 1 \pmod{p} \text{ for all } a \in \mathbb{Z}_p^*.$$

Proof By equation (31.21), $\phi(p) = p - 1$ if p is prime. ■

Fermat's theorem applies to every element in \mathbb{Z}_p except 0, since $0 \notin \mathbb{Z}_p^*$. For all $a \in \mathbb{Z}_p$, however, we have $a^p \equiv a \pmod{p}$ if p is prime.

If $\text{ord}_n(g) = |\mathbb{Z}_n^*|$, then every element in \mathbb{Z}_n^* is a power of g , modulo n , and g is a **primitive root** or a **generator** of \mathbb{Z}_n^* . For example, 3 is a primitive root, modulo 7, but 2 is not a primitive root, modulo 7. If \mathbb{Z}_n^* possesses a primitive root, the group \mathbb{Z}_n^* is **cyclic**. We omit the proof of the following theorem, which is proven by Niven and Zuckerman [265].

Theorem 31.32

The values of $n > 1$ for which \mathbb{Z}_n^* is cyclic are 2, 4, p^e , and $2p^e$, for all primes $p > 2$ and all positive integers e . ■

If g is a primitive root of \mathbb{Z}_n^* and a is any element of \mathbb{Z}_n^* , then there exists a z such that $g^z \equiv a \pmod{n}$. This z is a **discrete logarithm** or an **index** of a , modulo n , to the base g ; we denote this value as $\text{ind}_{n,g}(a)$.

Theorem 31.33 (Discrete logarithm theorem)

If g is a primitive root of \mathbb{Z}_n^* , then the equation $g^x \equiv g^y \pmod{n}$ holds if and only if the equation $x \equiv y \pmod{\phi(n)}$ holds.

Proof Suppose first that $x \equiv y \pmod{\phi(n)}$. Then, $x = y + k\phi(n)$ for some integer k . Therefore,

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} \pmod{n} \\ &\equiv g^y \cdot (g^{\phi(n)})^k \pmod{n} \\ &\equiv g^y \cdot 1^k \pmod{n} \quad (\text{by Euler's theorem}) \\ &\equiv g^y \pmod{n}. \end{aligned}$$

Conversely, suppose that $g^x \equiv g^y \pmod{n}$. Because the sequence of powers of g generates every element of $\langle g \rangle$ and $|\langle g \rangle| = \phi(n)$, Corollary 31.18 implies that the sequence of powers of g is periodic with period $\phi(n)$. Therefore, if $g^x \equiv g^y \pmod{n}$, then we must have $x \equiv y \pmod{\phi(n)}$. ■

We now turn our attention to the square roots of 1, modulo a prime power. The following theorem will be useful in our development of a primality-testing algorithm in Section 31.8.

Theorem 31.34

If p is an odd prime and $e \geq 1$, then the equation

$$x^2 \equiv 1 \pmod{p^e} \tag{31.34}$$

has only two solutions, namely $x = 1$ and $x = -1$.

Proof Equation (31.34) is equivalent to

$$p^e \mid (x - 1)(x + 1).$$

Since $p > 2$, we can have $p \mid (x - 1)$ or $p \mid (x + 1)$, but not both. (Otherwise, by property (31.3), p would also divide their difference $(x + 1) - (x - 1) = 2$.) If $p \nmid (x - 1)$, then $\gcd(p^e, x - 1) = 1$, and by Corollary 31.5, we would have $p^e \mid (x + 1)$. That is, $x \equiv -1 \pmod{p^e}$. Symmetrically, if $p \nmid (x + 1)$, then $\gcd(p^e, x + 1) = 1$, and Corollary 31.5 implies that $p^e \mid (x - 1)$, so that $x \equiv 1 \pmod{p^e}$. Therefore, either $x \equiv -1 \pmod{p^e}$ or $x \equiv 1 \pmod{p^e}$. ■

A number x is a **nontrivial square root of 1, modulo n** , if it satisfies the equation $x^2 \equiv 1 \pmod{n}$ but x is equivalent to neither of the two “trivial” square roots: 1 or -1 , modulo n . For example, 6 is a nontrivial square root of 1, modulo 35. We shall use the following corollary to Theorem 31.34 in the correctness proof in Section 31.8 for the Miller-Rabin primality-testing procedure.

Corollary 31.35

If there exists a nontrivial square root of 1, modulo n , then n is composite.

Proof By the contrapositive of Theorem 31.34, if there exists a nontrivial square root of 1, modulo n , then n cannot be an odd prime or a power of an odd prime. If $x^2 \equiv 1 \pmod{2}$, then $x \equiv 1 \pmod{2}$, and so all square roots of 1, modulo 2, are trivial. Thus, n cannot be prime. Finally, we must have $n > 1$ for a nontrivial square root of 1 to exist. Therefore, n must be composite. ■

Raising to powers with repeated squaring

A frequently occurring operation in number-theoretic computations is raising one number to a power modulo another number, also known as **modular exponentiation**. More precisely, we would like an efficient way to compute $a^b \pmod{n}$, where a and b are nonnegative integers and n is a positive integer. Modular exponentiation is an essential operation in many primality-testing routines and in the RSA public-key cryptosystem. The method of **repeated squaring** solves this problem efficiently using the binary representation of b .

Let $\langle b_k, b_{k-1}, \dots, b_1, b_0 \rangle$ be the binary representation of b . (That is, the binary representation is $k + 1$ bits long, b_k is the most significant bit, and b_0 is the least

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Figure 31.4 The results of MODULAR-EXPONENTIATION when computing $a^b \pmod n$, where $a = 7$, $b = 560 = (1000110000)$, and $n = 561$. The values are shown after each execution of the **for** loop. The final result is 1.

significant bit.) The following procedure computes $a^c \pmod n$ as c is increased by doublings and incrementations from 0 to b .

MODULAR-EXPONENTIATION(a, b, n)

```

1   $c = 0$ 
2   $d = 1$ 
3  let  $\langle b_k, b_{k-1}, \dots, b_0 \rangle$  be the binary representation of  $b$ 
4  for  $i = k$  downto 0
5       $c = 2c$ 
6       $d = (d \cdot d) \pmod n$ 
7      if  $b_i == 1$ 
8           $c = c + 1$ 
9           $d = (d \cdot a) \pmod n$ 
10 return  $d$ 
```

The essential use of squaring in line 6 of each iteration explains the name “repeated squaring.” As an example, for $a = 7$, $b = 560$, and $n = 561$, the algorithm computes the sequence of values modulo 561 shown in Figure 31.4; the sequence of exponents used appears in the row of the table labeled by c .

The variable c is not really needed by the algorithm but is included for the following two-part loop invariant:

Just prior to each iteration of the **for** loop of lines 4–9,

1. The value of c is the same as the prefix $\langle b_k, b_{k-1}, \dots, b_{i+1} \rangle$ of the binary representation of b , and
2. $d = a^c \pmod n$.

We use this loop invariant as follows:

Initialization: Initially, $i = k$, so that the prefix $\langle b_k, b_{k-1}, \dots, b_{i+1} \rangle$ is empty, which corresponds to $c = 0$. Moreover, $d = 1 = a^0 \pmod n$.

Maintenance: Let c' and d' denote the values of c and d at the end of an iteration of the **for** loop, and thus the values prior to the next iteration. Each iteration updates $c' = 2c$ (if $b_i = 0$) or $c' = 2c + 1$ (if $b_i = 1$), so that c will be correct prior to the next iteration. If $b_i = 0$, then $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = a^{c'} \bmod n$. If $b_i = 1$, then $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. In either case, $d = a^c \bmod n$ prior to the next iteration.

Termination: At termination, $i = -1$. Thus, $c = b$, since c has the value of the prefix $\langle b_k, b_{k-1}, \dots, b_0 \rangle$ of b 's binary representation. Hence $d = a^c \bmod n = a^b \bmod n$.

If the inputs a , b , and n are β -bit numbers, then the total number of arithmetic operations required is $O(\beta)$ and the total number of bit operations required is $O(\beta^3)$.

Exercises

31.6-1

Draw a table showing the order of every element in \mathbb{Z}_{11}^* . Pick the smallest primitive root g and compute a table giving $\text{ind}_{11,g}(x)$ for all $x \in \mathbb{Z}_{11}^*$.

31.6-2

Give a modular exponentiation algorithm that examines the bits of b from right to left instead of left to right.

31.6-3

Assuming that you know $\phi(n)$, explain how to compute $a^{-1} \bmod n$ for any $a \in \mathbb{Z}_n^*$ using the procedure MODULAR-EXPONENTIATION.

31.7 The RSA public-key cryptosystem

With a public-key cryptosystem, we can encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode them. A public-key cryptosystem also enables a party to append an unforgeable “digital signature” to the end of an electronic message. Such a signature is the electronic version of a handwritten signature on a paper document. It can be easily checked by anyone, forged by no one, yet loses its validity if any bit of the message is altered. It therefore provides authentication of both the identity of the signer and the contents of the signed message. It is the perfect tool

for electronically signed business contracts, electronic checks, electronic purchase orders, and other electronic communications that parties wish to authenticate.

The RSA public-key cryptosystem relies on the dramatic difference between the ease of finding large prime numbers and the difficulty of factoring the product of two large prime numbers. Section 31.8 describes an efficient procedure for finding large prime numbers, and Section 31.9 discusses the problem of factoring large integers.

Public-key cryptosystems

In a public-key cryptosystem, each participant has both a *public key* and a *secret key*. Each key is a piece of information. For example, in the RSA cryptosystem, each key consists of a pair of integers. The participants “Alice” and “Bob” are traditionally used in cryptography examples; we denote their public and secret keys as P_A, S_A for Alice and P_B, S_B for Bob.

Each participant creates his or her own public and secret keys. Secret keys are kept secret, but public keys can be revealed to anyone or even published. In fact, it is often convenient to assume that everyone’s public key is available in a public directory, so that any participant can easily obtain the public key of any other participant.

The public and secret keys specify functions that can be applied to any message. Let \mathcal{D} denote the set of permissible messages. For example, \mathcal{D} might be the set of all finite-length bit sequences. In the simplest, and original, formulation of public-key cryptography, we require that the public and secret keys specify one-to-one functions from \mathcal{D} to itself. We denote the function corresponding to Alice’s public key P_A by $P_A()$ and the function corresponding to her secret key S_A by $S_A()$. The functions $P_A()$ and $S_A()$ are thus permutations of \mathcal{D} . We assume that the functions $P_A()$ and $S_A()$ are efficiently computable given the corresponding key P_A or S_A .

The public and secret keys for any participant are a “matched pair” in that they specify functions that are inverses of each other. That is,

$$M = S_A(P_A(M)) , \quad (31.35)$$

$$M = P_A(S_A(M)) \quad (31.36)$$

for any message $M \in \mathcal{D}$. Transforming M with the two keys P_A and S_A successively, in either order, yields the message M back.

In a public-key cryptosystem, we require that no one but Alice be able to compute the function $S_A()$ in any practical amount of time. This assumption is crucial to keeping encrypted mail sent to Alice private and to knowing that Alice’s digital signatures are authentic. Alice must keep S_A secret; if she does not, she loses her uniqueness and the cryptosystem cannot provide her with unique capabilities. The assumption that only Alice can compute $S_A()$ must hold even though everyone

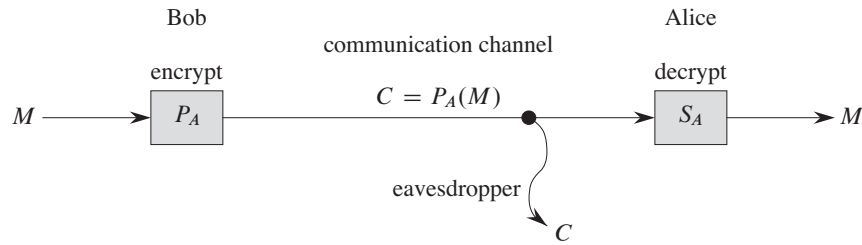


Figure 31.5 Encryption in a public key system. Bob encrypts the message M using Alice’s public key P_A and transmits the resulting ciphertext $C = P_A(M)$ over a communication channel to Alice. An eavesdropper who captures the transmitted ciphertext gains no information about M . Alice receives C and decrypts it using her secret key to obtain the original message $M = S_A(C)$.

knows P_A and can compute $P_A()$, the inverse function to $S_A()$, efficiently. In order to design a workable public-key cryptosystem, we must figure out how to create a system in which we can reveal a transformation $P_A()$ without thereby revealing how to compute the corresponding inverse transformation $S_A()$. This task appears formidable, but we shall see how to accomplish it.

In a public-key cryptosystem, encryption works as shown in Figure 31.5. Suppose Bob wishes to send Alice a message M encrypted so that it will look like unintelligible gibberish to an eavesdropper. The scenario for sending the message goes as follows.

- Bob obtains Alice’s public key P_A (from a public directory or directly from Alice).
- Bob computes the *ciphertext* $C = P_A(M)$ corresponding to the message M and sends C to Alice.
- When Alice receives the ciphertext C , she applies her secret key S_A to retrieve the original message: $S_A(C) = S_A(P_A(M)) = M$.

Because $S_A()$ and $P_A()$ are inverse functions, Alice can compute M from C . Because only Alice is able to compute $S_A()$, Alice is the only one who can compute M from C . Because Bob encrypts M using $P_A()$, only Alice can understand the transmitted message.

We can just as easily implement digital signatures within our formulation of a public-key cryptosystem. (There are other ways of approaching the problem of constructing digital signatures, but we shall not go into them here.) Suppose now that Alice wishes to send Bob a digitally signed response M' . Figure 31.6 shows how the digital-signature scenario proceeds.

- Alice computes her *digital signature* σ for the message M' using her secret key S_A and the equation $\sigma = S_A(M')$.

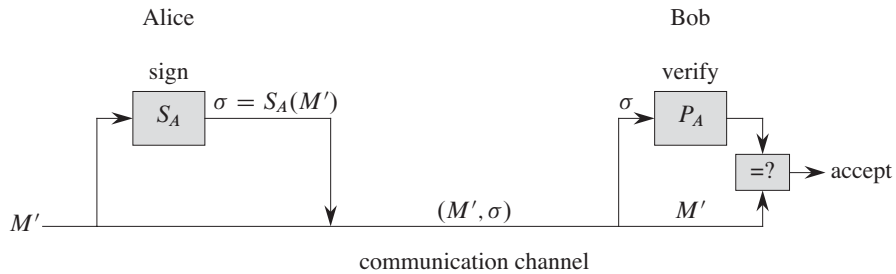


Figure 31.6 Digital signatures in a public-key system. Alice signs the message M' by appending her digital signature $\sigma = S_A(M')$ to it. She transmits the message/signature pair (M', σ) to Bob, who verifies it by checking the equation $M' = P_A(\sigma)$. If the equation holds, he accepts (M', σ) as a message that Alice has signed.

- Alice sends the message/signature pair (M', σ) to Bob.
- When Bob receives (M', σ) , he can verify that it originated from Alice by using Alice's public key to verify the equation $M' = P_A(\sigma)$. (Presumably, M' contains Alice's name, so Bob knows whose public key to use.) If the equation holds, then Bob concludes that the message M' was actually signed by Alice. If the equation fails to hold, Bob concludes either that the message M' or the digital signature σ was corrupted by transmission errors or that the pair (M', σ) is an attempted forgery.

Because a digital signature provides both authentication of the signer's identity and authentication of the contents of the signed message, it is analogous to a handwritten signature at the end of a written document.

A digital signature must be verifiable by anyone who has access to the signer's public key. A signed message can be verified by one party and then passed on to other parties who can also verify the signature. For example, the message might be an electronic check from Alice to Bob. After Bob verifies Alice's signature on the check, he can give the check to his bank, who can then also verify the signature and effect the appropriate funds transfer.

A signed message is not necessarily encrypted; the message can be "in the clear" and not protected from disclosure. By composing the above protocols for encryption and for signatures, we can create messages that are both signed and encrypted. The signer first appends his or her digital signature to the message and then encrypts the resulting message/signature pair with the public key of the intended recipient. The recipient decrypts the received message with his or her secret key to obtain both the original message and its digital signature. The recipient can then verify the signature using the public key of the signer. The corresponding combined process using paper-based systems would be to sign the paper document and

then seal the document inside a paper envelope that is opened only by the intended recipient.

The RSA cryptosystem

In the *RSA public-key cryptosystem*, a participant creates his or her public and secret keys with the following procedure:

1. Select at random two large prime numbers p and q such that $p \neq q$. The primes p and q might be, say, 1024 bits each.
2. Compute $n = pq$.
3. Select a small odd integer e that is relatively prime to $\phi(n)$, which, by equation (31.20), equals $(p - 1)(q - 1)$.
4. Compute d as the multiplicative inverse of e , modulo $\phi(n)$. (Corollary 31.26 guarantees that d exists and is uniquely defined. We can use the technique of Section 31.4 to compute d , given e and $\phi(n)$.)
5. Publish the pair $P = (e, n)$ as the participant's *RSA public key*.
6. Keep secret the pair $S = (d, n)$ as the participant's *RSA secret key*.

For this scheme, the domain \mathcal{D} is the set \mathbb{Z}_n . To transform a message M associated with a public key $P = (e, n)$, compute

$$P(M) = M^e \bmod n. \quad (31.37)$$

To transform a ciphertext C associated with a secret key $S = (d, n)$, compute

$$S(C) = C^d \bmod n. \quad (31.38)$$

These equations apply to both encryption and signatures. To create a signature, the signer applies his or her secret key to the message to be signed, rather than to a ciphertext. To verify a signature, the public key of the signer is applied to it, rather than to a message to be encrypted.

We can implement the public-key and secret-key operations using the procedure MODULAR-EXPONENTIATION described in Section 31.6. To analyze the running time of these operations, assume that the public key (e, n) and secret key (d, n) satisfy $\lg e = O(1)$, $\lg d \leq \beta$, and $\lg n \leq \beta$. Then, applying a public key requires $O(1)$ modular multiplications and uses $O(\beta^2)$ bit operations. Applying a secret key requires $O(\beta)$ modular multiplications, using $O(\beta^3)$ bit operations.

Theorem 31.36 (Correctness of RSA)

The RSA equations (31.37) and (31.38) define inverse transformations of \mathbb{Z}_n satisfying equations (31.35) and (31.36).

Proof From equations (31.37) and (31.38), we have that for any $M \in \mathbb{Z}_n$,
 $P(S(M)) = S(P(M)) = M^{ed} \pmod{n}$.

Since e and d are multiplicative inverses modulo $\phi(n) = (p-1)(q-1)$,
 $ed = 1 + k(p-1)(q-1)$

for some integer k . But then, if $M \not\equiv 0 \pmod{p}$, we have

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M((M \pmod{p})^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M(1)^{k(q-1)} && \pmod{p} \quad (\text{by Theorem 31.31}) \\ &\equiv M && \pmod{p}. \end{aligned}$$

Also, $M^{ed} \equiv M \pmod{p}$ if $M \equiv 0 \pmod{p}$. Thus,

$$M^{ed} \equiv M \pmod{p}$$

for all M . Similarly,

$$M^{ed} \equiv M \pmod{q}$$

for all M . Thus, by Corollary 31.29 to the Chinese remainder theorem,

$$M^{ed} \equiv M \pmod{n}$$

for all M . ■

The security of the RSA cryptosystem rests in large part on the difficulty of factoring large integers. If an adversary can factor the modulus n in a public key, then the adversary can derive the secret key from the public key, using the knowledge of the factors p and q in the same way that the creator of the public key used them. Therefore, if factoring large integers is easy, then breaking the RSA cryptosystem is easy. The converse statement, that if factoring large integers is hard, then breaking RSA is hard, is unproven. After two decades of research, however, no easier method has been found to break the RSA public-key cryptosystem than to factor the modulus n . And as we shall see in Section 31.9, factoring large integers is surprisingly difficult. By randomly selecting and multiplying together two 1024-bit primes, we can create a public key that cannot be “broken” in any feasible amount of time with current technology. In the absence of a fundamental breakthrough in the design of number-theoretic algorithms, and when implemented with care following recommended standards, the RSA cryptosystem is capable of providing a high degree of security in applications.

In order to achieve security with the RSA cryptosystem, however, we should use integers that are quite long—hundreds or even more than one thousand bits

long—to resist possible advances in the art of factoring. At the time of this writing (2009), RSA moduli were commonly in the range of 768 to 2048 bits. To create moduli of such sizes, we must be able to find large primes efficiently. Section 31.8 addresses this problem.

For efficiency, RSA is often used in a “hybrid” or “key-management” mode with fast non-public-key cryptosystems. With such a system, the encryption and decryption keys are identical. If Alice wishes to send a long message M to Bob privately, she selects a random key K for the fast non-public-key cryptosystem and encrypts M using K , obtaining ciphertext C . Here, C is as long as M , but K is quite short. Then, she encrypts K using Bob’s public RSA key. Since K is short, computing $P_B(K)$ is fast (much faster than computing $P_B(M)$). She then transmits $(C, P_B(K))$ to Bob, who decrypts $P_B(K)$ to obtain K and then uses K to decrypt C , obtaining M .

We can use a similar hybrid approach to make digital signatures efficiently. This approach combines RSA with a public *collision-resistant hash function* h —a function that is easy to compute but for which it is computationally infeasible to find two messages M and M' such that $h(M) = h(M')$. The value $h(M)$ is a short (say, 256-bit) “fingerprint” of the message M . If Alice wishes to sign a message M , she first applies h to M to obtain the fingerprint $h(M)$, which she then encrypts with her secret key. She sends $(M, S_A(h(M)))$ to Bob as her signed version of M . Bob can verify the signature by computing $h(M)$ and verifying that P_A applied to $S_A(h(M))$ as received equals $h(M)$. Because no one can create two messages with the same fingerprint, it is computationally infeasible to alter a signed message and preserve the validity of the signature.

Finally, we note that the use of *certificates* makes distributing public keys much easier. For example, assume there is a “trusted authority” T whose public key is known by everyone. Alice can obtain from T a signed message (her certificate) stating that “Alice’s public key is P_A .” This certificate is “self-authenticating” since everyone knows P_T . Alice can include her certificate with her signed messages, so that the recipient has Alice’s public key immediately available in order to verify her signature. Because her key was signed by T , the recipient knows that Alice’s key is really Alice’s.

Exercises

31.7-1

Consider an RSA key set with $p = 11$, $q = 29$, $n = 319$, and $e = 3$. What value of d should be used in the secret key? What is the encryption of the message $M = 100$?

31.7-2

Prove that if Alice's public exponent e is 3 and an adversary obtains Alice's secret exponent d , where $0 < d < \phi(n)$, then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n . (Although you are not asked to prove it, you may be interested to know that this result remains true even if the condition $e = 3$ is removed. See Miller [255].)

31.7-3 ★

Prove that RSA is multiplicative in the sense that

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n}.$$

Use this fact to prove that if an adversary had a procedure that could efficiently decrypt 1 percent of messages from \mathbb{Z}_n encrypted with P_A , then he could employ a probabilistic algorithm to decrypt every message encrypted with P_A with high probability.

★ 31.8 Primality testing

In this section, we consider the problem of finding large primes. We begin with a discussion of the density of primes, proceed to examine a plausible, but incomplete, approach to primality testing, and then present an effective randomized primality test due to Miller and Rabin.

The density of prime numbers

For many applications, such as cryptography, we need to find large “random” primes. Fortunately, large primes are not too rare, so that it is feasible to test random integers of the appropriate size until we find a prime. The *prime distribution function* $\pi(n)$ specifies the number of primes that are less than or equal to n . For example, $\pi(10) = 4$, since there are 4 prime numbers less than or equal to 10, namely, 2, 3, 5, and 7. The prime number theorem gives a useful approximation to $\pi(n)$.

Theorem 31.37 (Prime number theorem)

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \ln n} = 1. \quad \blacksquare$$

The approximation $n / \ln n$ gives reasonably accurate estimates of $\pi(n)$ even for small n . For example, it is off by less than 6% at $n = 10^9$, where $\pi(n) =$

50,847,534 and $n / \ln n \approx 48,254,942$. (To a number theorist, 10^9 is a small number.)

We can view the process of randomly selecting an integer n and determining whether it is prime as a Bernoulli trial (see Section C.4). By the prime number theorem, the probability of a success—that is, the probability that n is prime—is approximately $1 / \ln n$. The geometric distribution tells us how many trials we need to obtain a success, and by equation (C.32), the expected number of trials is approximately $\ln n$. Thus, we would expect to examine approximately $\ln n$ integers chosen randomly near n in order to find a prime that is of the same length as n . For example, we expect that finding a 1024-bit prime would require testing approximately $\ln 2^{1024} \approx 710$ randomly chosen 1024-bit numbers for primality. (Of course, we can cut this figure in half by choosing only odd integers.)

In the remainder of this section, we consider the problem of determining whether or not a large odd integer n is prime. For notational convenience, we assume that n has the prime factorization

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \quad (31.39)$$

where $r \geq 1$, p_1, p_2, \dots, p_r are the prime factors of n , and e_1, e_2, \dots, e_r are positive integers. The integer n is prime if and only if $r = 1$ and $e_1 = 1$.

One simple approach to the problem of testing for primality is **trial division**. We try dividing n by each integer $2, 3, \dots, \lfloor \sqrt{n} \rfloor$. (Again, we may skip even integers greater than 2.) It is easy to see that n is prime if and only if none of the trial divisors divides n . Assuming that each trial division takes constant time, the worst-case running time is $\Theta(\sqrt{n})$, which is exponential in the length of n . (Recall that if n is encoded in binary using β bits, then $\beta = \lceil \lg(n + 1) \rceil$, and so $\sqrt{n} = \Theta(2^{\beta/2})$.) Thus, trial division works well only if n is very small or happens to have a small prime factor. When it works, trial division has the advantage that it not only determines whether n is prime or composite, but also determines one of n 's prime factors if n is composite.

In this section, we are interested only in finding out whether a given number n is prime; if n is composite, we are not concerned with finding its prime factorization. As we shall see in Section 31.9, computing the prime factorization of a number is computationally expensive. It is perhaps surprising that it is much easier to tell whether or not a given number is prime than it is to determine the prime factorization of the number if it is not prime.

Pseudoprimality testing

We now consider a method for primality testing that “almost works” and in fact is good enough for many practical applications. Later on, we shall present a re-

finement of this method that removes the small defect. Let \mathbb{Z}_n^+ denote the nonzero elements of \mathbb{Z}_n :

$$\mathbb{Z}_n^+ = \{1, 2, \dots, n-1\} .$$

If n is prime, then $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$.

We say that n is a **base- a pseudoprime** if n is composite and

$$a^{n-1} \equiv 1 \pmod{n} . \tag{31.40}$$

Fermat's theorem (Theorem 31.31) implies that if n is prime, then n satisfies equation (31.40) for every a in \mathbb{Z}_n^+ . Thus, if we can find any $a \in \mathbb{Z}_n^+$ such that n does *not* satisfy equation (31.40), then n is certainly composite. Surprisingly, the converse *almost* holds, so that this criterion forms an almost perfect test for primality. We test to see whether n satisfies equation (31.40) for $a = 2$. If not, we declare n to be composite by returning COMPOSITE. Otherwise, we return PRIME, guessing that n is prime (when, in fact, all we know is that n is either prime or a base-2 pseudoprime).

The following procedure pretends in this manner to be checking the primality of n . It uses the procedure MODULAR-EXPONENTIATION from Section 31.6. We assume that the input n is an odd integer greater than 2.

PSEUDOPRIME(n)

```

1  if MODULAR-EXPONENTIATION(2,  $n-1$ ,  $n$ )  $\not\equiv$  1 (mod  $n$ )
2      return COMPOSITE           // definitely
3  else return PRIME             // we hope!
```

This procedure can make errors, but only of one type. That is, if it says that n is composite, then it is always correct. If it says that n is prime, however, then it makes an error only if n is a base-2 pseudoprime.

How often does this procedure err? Surprisingly rarely. There are only 22 values of n less than 10,000 for which it errs; the first four such values are 341, 561, 645, and 1105. We won't prove it, but the probability that this program makes an error on a randomly chosen β -bit number goes to zero as $\beta \rightarrow \infty$. Using more precise estimates due to Pomerance [279] of the number of base-2 pseudoprimes of a given size, we may estimate that a randomly chosen 512-bit number that is called prime by the above procedure has less than one chance in 10^{20} of being a base-2 pseudoprime, and a randomly chosen 1024-bit number that is called prime has less than one chance in 10^{41} of being a base-2 pseudoprime. So if you are merely trying to find a large prime for some application, for all practical purposes you almost never go wrong by choosing large numbers at random until one of them causes PSEUDOPRIME to return PRIME. But when the numbers being tested for primality are not randomly chosen, we need a better approach for testing primality.

As we shall see, a little more cleverness, and some randomization, will yield a primality-testing routine that works well on all inputs.

Unfortunately, we cannot entirely eliminate all the errors by simply checking equation (31.40) for a second base number, say $a = 3$, because there exist composite integers n , known as *Carmichael numbers*, that satisfy equation (31.40) for all $a \in \mathbb{Z}_n^*$. (We note that equation (31.40) does fail when $\gcd(a, n) > 1$ —that is, when $a \notin \mathbb{Z}_n^*$ —but hoping to demonstrate that n is composite by finding such an a can be difficult if n has only large prime factors.) The first three Carmichael numbers are 561, 1105, and 1729. Carmichael numbers are extremely rare; there are, for example, only 255 of them less than 100,000,000. Exercise 31.8-2 helps explain why they are so rare.

We next show how to improve our primality test so that it won't be fooled by Carmichael numbers.

The Miller-Rabin randomized primality test

The Miller-Rabin primality test overcomes the problems of the simple test PSEUDOPRIME with two modifications:

- It tries several randomly chosen base values a instead of just one base value.
- While computing each modular exponentiation, it looks for a nontrivial square root of 1, modulo n , during the final set of squarings. If it finds one, it stops and returns COMPOSITE. Corollary 31.35 from Section 31.6 justifies detecting composites in this manner.

The pseudocode for the Miller-Rabin primality test follows. The input $n > 2$ is the odd number to be tested for primality, and s is the number of randomly chosen base values from \mathbb{Z}_n^+ to be tried. The code uses the random-number generator RANDOM described on page 117: RANDOM($1, n - 1$) returns a randomly chosen integer a satisfying $1 \leq a \leq n - 1$. The code uses an auxiliary procedure WITNESS such that WITNESS(a, n) is TRUE if and only if a is a “witness” to the compositeness of n —that is, if it is possible using a to prove (in a manner that we shall see) that n is composite. The test WITNESS(a, n) is an extension of, but more effective than, the test

$$a^{n-1} \not\equiv 1 \pmod{n}$$

that formed the basis (using $a = 2$) for PSEUDOPRIME. We first present and justify the construction of WITNESS, and then we shall show how we use it in the Miller-Rabin primality test. Let $n - 1 = 2^t u$ where $t \geq 1$ and u is odd; i.e., the binary representation of $n - 1$ is the binary representation of the odd integer u followed by exactly t zeros. Therefore, $a^{n-1} \equiv (a^u)^{2^t} \pmod{n}$, so that we can

compute $a^{n-1} \bmod n$ by first computing $a^u \bmod n$ and then squaring the result t times successively.

WITNESS(a, n)

```

1  let  $t$  and  $u$  be such that  $t \geq 1$ ,  $u$  is odd, and  $n - 1 = 2^t u$ 
2   $x_0 = \text{MODULAR-EXPONENTIATION}(a, u, n)$ 
3  for  $i = 1$  to  $t$ 
4       $x_i = x_{i-1}^2 \bmod n$ 
5      if  $x_i == 1$  and  $x_{i-1} \neq 1$  and  $x_{i-1} \neq n - 1$ 
6          return TRUE
7  if  $x_t \neq 1$ 
8      return TRUE
9  return FALSE

```

This pseudocode for WITNESS computes $a^{n-1} \bmod n$ by first computing the value $x_0 = a^u \bmod n$ in line 2 and then squaring the result t times in a row in the **for** loop of lines 3–6. By induction on i , the sequence x_0, x_1, \dots, x_t of values computed satisfies the equation $x_i \equiv a^{2^i u} \pmod{n}$ for $i = 0, 1, \dots, t$, so that in particular $x_t \equiv a^{n-1} \pmod{n}$. After line 4 performs a squaring step, however, the loop may terminate early if lines 5–6 detect that a nontrivial square root of 1 has just been discovered. (We shall explain these tests shortly.) If so, the algorithm stops and returns TRUE. Lines 7–8 return TRUE if the value computed for $x_t \equiv a^{n-1} \pmod{n}$ is not equal to 1, just as the PSEUDOPRIME procedure returns COMPOSITE in this case. Line 9 returns FALSE if we haven't returned TRUE in lines 6 or 8.

We now argue that if WITNESS(a, n) returns TRUE, then we can construct a proof that n is composite using a as a witness.

If WITNESS returns TRUE from line 8, then it has discovered that $x_t = a^{n-1} \bmod n \neq 1$. If n is prime, however, we have by Fermat's theorem (Theorem 31.31) that $a^{n-1} \equiv 1 \pmod{n}$ for all $a \in \mathbb{Z}_n^+$. Therefore, n cannot be prime, and the equation $a^{n-1} \bmod n \neq 1$ proves this fact.

If WITNESS returns TRUE from line 6, then it has discovered that x_{i-1} is a nontrivial square root of 1, modulo n , since we have that $x_{i-1} \not\equiv \pm 1 \pmod{n}$ yet $x_i \equiv x_{i-1}^2 \equiv 1 \pmod{n}$. Corollary 31.35 states that only if n is composite can there exist a nontrivial square root of 1 modulo n , so that demonstrating that x_{i-1} is a nontrivial square root of 1 modulo n proves that n is composite.

This completes our proof of the correctness of WITNESS. If we find that the call WITNESS(a, n) returns TRUE, then n is surely composite, and the witness a , along with the reason that the procedure returns TRUE (did it return from line 6 or from line 8?), provides a proof that n is composite.

At this point, we briefly present an alternative description of the behavior of WITNESS as a function of the sequence $X = \langle x_0, x_1, \dots, x_t \rangle$, which we shall find useful later on, when we analyze the efficiency of the Miller-Rabin primality test. Note that if $x_i = 1$ for some $0 \leq i < t$, WITNESS might not compute the rest of the sequence. If it were to do so, however, each value $x_{i+1}, x_{i+2}, \dots, x_t$ would be 1, and we consider these positions in the sequence X as being all 1s. We have four cases:

1. $X = \langle \dots, d \rangle$, where $d \neq 1$: the sequence X does not end in 1. Return TRUE in line 8; a is a witness to the compositeness of n (by Fermat's Theorem).
2. $X = \langle 1, 1, \dots, 1 \rangle$: the sequence X is all 1s. Return FALSE; a is not a witness to the compositeness of n .
3. $X = \langle \dots, -1, 1, \dots, 1 \rangle$: the sequence X ends in 1, and the last non-1 is equal to -1 . Return FALSE; a is not a witness to the compositeness of n .
4. $X = \langle \dots, d, 1, \dots, 1 \rangle$, where $d \neq \pm 1$: the sequence X ends in 1, but the last non-1 is not -1 . Return TRUE in line 6; a is a witness to the compositeness of n , since d is a nontrivial square root of 1.

We now examine the Miller-Rabin primality test based on the use of WITNESS. Again, we assume that n is an odd integer greater than 2.

MILLER-RABIN(n, s)

```

1  for  $j = 1$  to  $s$ 
2       $a = \text{RANDOM}(1, n - 1)$ 
3      if WITNESS( $a, n$ )
4          return COMPOSITE           // definitely
5  return PRIME                       // almost surely
```

The procedure MILLER-RABIN is a probabilistic search for a proof that n is composite. The main loop (beginning on line 1) picks up to s random values of a from \mathbb{Z}_n^+ (line 2). If one of the a 's picked is a witness to the compositeness of n , then MILLER-RABIN returns COMPOSITE on line 4. Such a result is always correct, by the correctness of WITNESS. If MILLER-RABIN finds no witness in s trials, then the procedure assumes that this is because no witnesses exist, and therefore it assumes that n is prime. We shall see that this result is likely to be correct if s is large enough, but that there is still a tiny chance that the procedure may be unlucky in its choice of a 's and that witnesses do exist even though none has been found.

To illustrate the operation of MILLER-RABIN, let n be the Carmichael number 561, so that $n - 1 = 560 = 2^4 \cdot 35$, $t = 4$, and $u = 35$. If the procedure chooses $a = 7$ as a base, Figure 31.4 in Section 31.6 shows that WITNESS computes $x_0 \equiv a^{35} \equiv 241 \pmod{561}$ and thus computes the sequence

$X = \langle 241, 298, 166, 67, 1 \rangle$. Thus, WITNESS discovers a nontrivial square root of 1 in the last squaring step, since $a^{280} \equiv 67 \pmod{n}$ and $a^{560} \equiv 1 \pmod{n}$. Therefore, $a = 7$ is a witness to the compositeness of n , WITNESS(7, n) returns TRUE, and MILLER-RABIN returns COMPOSITE.

If n is a β -bit number, MILLER-RABIN requires $O(s\beta)$ arithmetic operations and $O(s\beta^3)$ bit operations, since it requires asymptotically no more work than s modular exponentiations.

Error rate of the Miller-Rabin primality test

If MILLER-RABIN returns PRIME, then there is a very slim chance that it has made an error. Unlike PSEUDOPRIME, however, the chance of error does not depend on n ; there are no bad inputs for this procedure. Rather, it depends on the size of s and the “luck of the draw” in choosing base values a . Moreover, since each test is more stringent than a simple check of equation (31.40), we can expect on general principles that the error rate should be small for randomly chosen integers n . The following theorem presents a more precise argument.

Theorem 31.38

If n is an odd composite number, then the number of witnesses to the compositeness of n is at least $(n - 1)/2$.

Proof The proof shows that the number of nonwitnesses is at most $(n - 1)/2$, which implies the theorem.

We start by claiming that any nonwitness must be a member of \mathbb{Z}_n^* . Why? Consider any nonwitness a . It must satisfy $a^{n-1} \equiv 1 \pmod{n}$ or, equivalently, $a \cdot a^{n-2} \equiv 1 \pmod{n}$. Thus, the equation $ax \equiv 1 \pmod{n}$ has a solution, namely a^{n-2} . By Corollary 31.21, $\gcd(a, n) \mid 1$, which in turn implies that $\gcd(a, n) = 1$. Therefore, a is a member of \mathbb{Z}_n^* ; all nonwitnesses belong to \mathbb{Z}_n^* .

To complete the proof, we show that not only are all nonwitnesses contained in \mathbb{Z}_n^* , they are all contained in a proper subgroup B of \mathbb{Z}_n^* (recall that we say B is a *proper* subgroup of \mathbb{Z}_n^* when B is subgroup of \mathbb{Z}_n^* but B is not equal to \mathbb{Z}_n^*). By Corollary 31.16, we then have $|B| \leq |\mathbb{Z}_n^*|/2$. Since $|\mathbb{Z}_n^*| \leq n - 1$, we obtain $|B| \leq (n - 1)/2$. Therefore, the number of nonwitnesses is at most $(n - 1)/2$, so that the number of witnesses must be at least $(n - 1)/2$.

We now show how to find a proper subgroup B of \mathbb{Z}_n^* containing all of the nonwitnesses. We break the proof into two cases.

Case 1: There exists an $x \in \mathbb{Z}_n^*$ such that

$$x^{n-1} \not\equiv 1 \pmod{n}.$$

In other words, n is not a Carmichael number. Because, as we noted earlier, Carmichael numbers are extremely rare, case 1 is the main case that arises “in practice” (e.g., when n has been chosen randomly and is being tested for primality).

Let $B = \{b \in \mathbb{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}$. Clearly, B is nonempty, since $1 \in B$. Since B is closed under multiplication modulo n , we have that B is a subgroup of \mathbb{Z}_n^* by Theorem 31.14. Note that every nonwitness belongs to B , since a nonwitness a satisfies $a^{n-1} \equiv 1 \pmod{n}$. Since $x \in \mathbb{Z}_n^* - B$, we have that B is a proper subgroup of \mathbb{Z}_n^* .

Case 2: For all $x \in \mathbb{Z}_n^*$,

$$x^{n-1} \equiv 1 \pmod{n}. \quad (31.41)$$

In other words, n is a Carmichael number. This case is extremely rare in practice. However, the Miller-Rabin test (unlike a pseudo-primality test) can efficiently determine that Carmichael numbers are composite, as we now show.

In this case, n cannot be a prime power. To see why, let us suppose to the contrary that $n = p^e$, where p is a prime and $e > 1$. We derive a contradiction as follows. Since we assume that n is odd, p must also be odd. Theorem 31.32 implies that \mathbb{Z}_n^* is a cyclic group: it contains a generator g such that $\text{ord}_n(g) = |\mathbb{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p - 1)p^{e-1}$. (The formula for $\phi(n)$ comes from equation (31.20).) By equation (31.41), we have $g^{n-1} \equiv 1 \pmod{n}$. Then the discrete logarithm theorem (Theorem 31.33, taking $y = 0$) implies that $n - 1 \equiv 0 \pmod{\phi(n)}$, or

$$(p - 1)p^{e-1} \mid p^e - 1.$$

This is a contradiction for $e > 1$, since $(p - 1)p^{e-1}$ is divisible by the prime p but $p^e - 1$ is not. Thus, n is not a prime power.

Since the odd composite number n is not a prime power, we decompose it into a product $n_1 n_2$, where n_1 and n_2 are odd numbers greater than 1 that are relatively prime to each other. (There may be several ways to decompose n , and it does not matter which one we choose. For example, if $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we can choose $n_1 = p_1^{e_1}$ and $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Recall that we define t and u so that $n - 1 = 2^t u$, where $t \geq 1$ and u is odd, and that for an input a , the procedure WITNESS computes the sequence

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \dots, a^{2^{t-1} u} \rangle$$

(all computations are performed modulo n).

Let us call a pair (v, j) of integers *acceptable* if $v \in \mathbb{Z}_n^*$, $j \in \{0, 1, \dots, t\}$, and

$$v^{2^j u} \equiv -1 \pmod{n}.$$

Acceptable pairs certainly exist since u is odd; we can choose $v = n - 1$ and $j = 0$, so that $(n - 1, 0)$ is an acceptable pair. Now pick the largest possible j such that there exists an acceptable pair (v, j) , and fix v so that (v, j) is an acceptable pair. Let

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Since B is closed under multiplication modulo n , it is a subgroup of \mathbb{Z}_n^* . By Theorem 31.15, therefore, $|B|$ divides $|\mathbb{Z}_n^*|$. Every nonwitness must be a member of B , since the sequence X produced by a nonwitness must either be all 1s or else contain a -1 no later than the j th position, by the maximality of j . (If (a, j') is acceptable, where a is a nonwitness, we must have $j' \leq j$ by how we chose j .)

We now use the existence of v to demonstrate that there exists a $w \in \mathbb{Z}_n^* - B$, and hence that B is a proper subgroup of \mathbb{Z}_n^* . Since $v^{2^j u} \equiv -1 \pmod{n}$, we have $v^{2^j u} \equiv -1 \pmod{n_1}$ by Corollary 31.29 to the Chinese remainder theorem. By Corollary 31.28, there exists a w simultaneously satisfying the equations

$$\begin{aligned} w &\equiv v \pmod{n_1}, \\ w &\equiv 1 \pmod{n_2}. \end{aligned}$$

Therefore,

$$\begin{aligned} w^{2^j u} &\equiv -1 \pmod{n_1}, \\ w^{2^j u} &\equiv 1 \pmod{n_2}. \end{aligned}$$

By Corollary 31.29, $w^{2^j u} \not\equiv 1 \pmod{n_1}$ implies $w^{2^j u} \not\equiv 1 \pmod{n}$, and $w^{2^j u} \not\equiv -1 \pmod{n_2}$ implies $w^{2^j u} \not\equiv -1 \pmod{n}$. Hence, we conclude that $w^{2^j u} \not\equiv \pm 1 \pmod{n}$, and so $w \notin B$.

It remains to show that $w \in \mathbb{Z}_n^*$, which we do by first working separately modulo n_1 and modulo n_2 . Working modulo n_1 , we observe that since $v \in \mathbb{Z}_n^*$, we have that $\gcd(v, n) = 1$, and so also $\gcd(v, n_1) = 1$; if v does not have any common divisors with n , then it certainly does not have any common divisors with n_1 . Since $w \equiv v \pmod{n_1}$, we see that $\gcd(w, n_1) = 1$. Working modulo n_2 , we observe that $w \equiv 1 \pmod{n_2}$ implies $\gcd(w, n_2) = 1$. To combine these results, we use Theorem 31.6, which implies that $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$. That is, $w \in \mathbb{Z}_n^*$.

Therefore $w \in \mathbb{Z}_n^* - B$, and we finish case 2 with the conclusion that B is a proper subgroup of \mathbb{Z}_n^* .

In either case, we see that the number of witnesses to the compositeness of n is at least $(n - 1)/2$. ■

Theorem 31.39

For any odd integer $n > 2$ and positive integer s , the probability that MILLER-RABIN(n, s) errs is at most 2^{-s} .

Proof Using Theorem 31.38, we see that if n is composite, then each execution of the **for** loop of lines 1–4 has a probability of at least $1/2$ of discovering a witness x to the compositeness of n . MILLER-RABIN makes an error only if it is so unlucky as to miss discovering a witness to the compositeness of n on each of the s iterations of the main loop. The probability of such a sequence of misses is at most 2^{-s} . ■

If n is prime, MILLER-RABIN always reports PRIME, and if n is composite, the chance that MILLER-RABIN reports PRIME is at most 2^{-s} .

When applying MILLER-RABIN to a large randomly chosen integer n , however, we need to consider as well the prior probability that n is prime, in order to correctly interpret MILLER-RABIN's result. Suppose that we fix a bit length β and choose at random an integer n of length β bits to be tested for primality. Let A denote the event that n is prime. By the prime number theorem (Theorem 31.37), the probability that n is prime is approximately

$$\begin{aligned}\Pr\{A\} &\approx 1/\ln n \\ &\approx 1.443/\beta.\end{aligned}$$

Now let B denote the event that MILLER-RABIN returns PRIME. We have that $\Pr\{\bar{B} \mid A\} = 0$ (or equivalently, that $\Pr\{B \mid A\} = 1$) and $\Pr\{B \mid \bar{A}\} \leq 2^{-s}$ (or equivalently, that $\Pr\{\bar{B} \mid \bar{A}\} > 1 - 2^{-s}$).

But what is $\Pr\{A \mid B\}$, the probability that n is prime, given that MILLER-RABIN has returned PRIME? By the alternate form of Bayes's theorem (equation (C.18)) we have

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{\Pr\{A\} \Pr\{B \mid A\}}{\Pr\{A\} \Pr\{B \mid A\} + \Pr\{\bar{A}\} \Pr\{B \mid \bar{A}\}} \\ &\approx \frac{1}{1 + 2^{-s}(\ln n - 1)}.\end{aligned}$$

This probability does not exceed $1/2$ until s exceeds $\lg(\ln n - 1)$. Intuitively, that many initial trials are needed just for the confidence derived from failing to find a witness to the compositeness of n to overcome the prior bias in favor of n being composite. For a number with $\beta = 1024$ bits, this initial testing requires about

$$\begin{aligned}\lg(\ln n - 1) &\approx \lg(\beta/1.443) \\ &\approx 9\end{aligned}$$

trials. In any case, choosing $s = 50$ should suffice for almost any imaginable application.

In fact, the situation is much better. If we are trying to find large primes by applying MILLER-RABIN to large randomly chosen odd integers, then choosing a small value of s (say 3) is very unlikely to lead to erroneous results, though

we won't prove it here. The reason is that for a randomly chosen odd composite integer n , the expected number of nonwitnesses to the compositeness of n is likely to be very much smaller than $(n - 1)/2$.

If the integer n is not chosen randomly, however, the best that can be proven is that the number of nonwitnesses is at most $(n - 1)/4$, using an improved version of Theorem 31.38. Furthermore, there do exist integers n for which the number of nonwitnesses is $(n - 1)/4$.

Exercises

31.8-1

Prove that if an odd integer $n > 1$ is not a prime or a prime power, then there exists a nontrivial square root of 1 modulo n .

31.8-2 ★

It is possible to strengthen Euler's theorem slightly to the form

$$a^{\lambda(n)} \equiv 1 \pmod{n} \text{ for all } a \in \mathbb{Z}_n^*,$$

where $n = p_1^{e_1} \cdots p_r^{e_r}$ and $\lambda(n)$ is defined by

$$\lambda(n) = \text{lcm}(\phi(p_1^{e_1}), \dots, \phi(p_r^{e_r})). \quad (31.42)$$

Prove that $\lambda(n) \mid \phi(n)$. A composite number n is a Carmichael number if $\lambda(n) \mid n - 1$. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$; here, $\lambda(n) = \text{lcm}(2, 10, 16) = 80$, which divides 560. Prove that Carmichael numbers must be both "square-free" (not divisible by the square of any prime) and the product of at least three primes. (For this reason, they are not very common.)

31.8-3

Prove that if x is a nontrivial square root of 1, modulo n , then $\gcd(x - 1, n)$ and $\gcd(x + 1, n)$ are both nontrivial divisors of n .

★ 31.9 Integer factorization

Suppose we have an integer n that we wish to *factor*, that is, to decompose into a product of primes. The primality test of the preceding section may tell us that n is composite, but it does not tell us the prime factors of n . Factoring a large integer n seems to be much more difficult than simply determining whether n is prime or composite. Even with today's supercomputers and the best algorithms to date, we cannot feasibly factor an arbitrary 1024-bit number.

Pollard's rho heuristic

Trial division by all integers up to R is guaranteed to factor completely any number up to R^2 . For the same amount of work, the following procedure, POLLARD-RHO, factors any number up to R^4 (unless we are unlucky). Since the procedure is only a heuristic, neither its running time nor its success is guaranteed, although the procedure is highly effective in practice. Another advantage of the POLLARD-RHO procedure is that it uses only a constant number of memory locations. (If you wanted to, you could easily implement POLLARD-RHO on a programmable pocket calculator to find factors of small numbers.)

POLLARD-RHO(n)

```

1   $i = 1$ 
2   $x_1 = \text{RANDOM}(0, n - 1)$ 
3   $y = x_1$ 
4   $k = 2$ 
5  while TRUE
6       $i = i + 1$ 
7       $x_i = (x_{i-1}^2 - 1) \bmod n$ 
8       $d = \text{gcd}(y - x_i, n)$ 
9      if  $d \neq 1$  and  $d \neq n$ 
10         print  $d$ 
11     if  $i == k$ 
12          $y = x_i$ 
13          $k = 2k$ 

```

The procedure works as follows. Lines 1–2 initialize i to 1 and x_1 to a randomly chosen value in \mathbb{Z}_n . The **while** loop beginning on line 5 iterates forever, searching for factors of n . During each iteration of the **while** loop, line 7 uses the recurrence

$$x_i = (x_{i-1}^2 - 1) \bmod n \quad (31.43)$$

to produce the next value of x_i in the infinite sequence

$$x_1, x_2, x_3, x_4, \dots, \quad (31.44)$$

with line 6 correspondingly incrementing i . The pseudocode is written using subscripted variables x_i for clarity, but the program works the same if all of the subscripts are dropped, since only the most recent value of x_i needs to be maintained. With this modification, the procedure uses only a constant number of memory locations.

Every so often, the program saves the most recently generated x_i value in the variable y . Specifically, the values that are saved are the ones whose subscripts are powers of 2:

$x_1, x_2, x_4, x_8, x_{16}, \dots$

Line 3 saves the value x_1 , and line 12 saves x_k whenever i is equal to k . The variable k is initialized to 2 in line 4, and line 13 doubles it whenever line 12 updates y . Therefore, k follows the sequence 1, 2, 4, 8, ... and always gives the subscript of the next value x_k to be saved in y .

Lines 8–10 try to find a factor of n , using the saved value of y and the current value of x_i . Specifically, line 8 computes the greatest common divisor $d = \gcd(y - x_i, n)$. If line 9 finds d to be a nontrivial divisor of n , then line 10 prints d .

This procedure for finding a factor may seem somewhat mysterious at first. Note, however, that POLLARD-RHO never prints an incorrect answer; any number it prints is a nontrivial divisor of n . POLLARD-RHO might not print anything at all, though; it comes with no guarantee that it will print any divisors. We shall see, however, that we have good reason to expect POLLARD-RHO to print a factor p of n after $\Theta(\sqrt{p})$ iterations of the **while** loop. Thus, if n is composite, we can expect this procedure to discover enough divisors to factor n completely after approximately $n^{1/4}$ updates, since every prime factor p of n except possibly the largest one is less than \sqrt{n} .

We begin our analysis of how this procedure behaves by studying how long it takes a random sequence modulo n to repeat a value. Since \mathbb{Z}_n is finite, and since each value in the sequence (31.44) depends only on the previous value, the sequence (31.44) eventually repeats itself. Once we reach an x_i such that $x_i = x_j$ for some $j < i$, we are in a cycle, since $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$, and so on. The reason for the name “rho heuristic” is that, as Figure 31.7 shows, we can draw the sequence x_1, x_2, \dots, x_{j-1} as the “tail” of the rho and the cycle x_j, x_{j+1}, \dots, x_i as the “body” of the rho.

Let us consider the question of how long it takes for the sequence of x_i to repeat. This information is not exactly what we need, but we shall see later how to modify the argument. For the purpose of this estimation, let us assume that the function

$$f_n(x) = (x^2 - 1) \bmod n$$

behaves like a “random” function. Of course, it is not really random, but this assumption yields results consistent with the observed behavior of POLLARD-RHO. We can then consider each x_i to have been independently drawn from \mathbb{Z}_n according to a uniform distribution on \mathbb{Z}_n . By the birthday-paradox analysis of Section 5.4.1, we expect $\Theta(\sqrt{n})$ steps to be taken before the sequence cycles.

Now for the required modification. Let p be a nontrivial factor of n such that $\gcd(p, n/p) = 1$. For example, if n has the factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we may take p to be $p_1^{e_1}$. (If $e_1 = 1$, then p is just the smallest prime factor of n , a good example to keep in mind.)

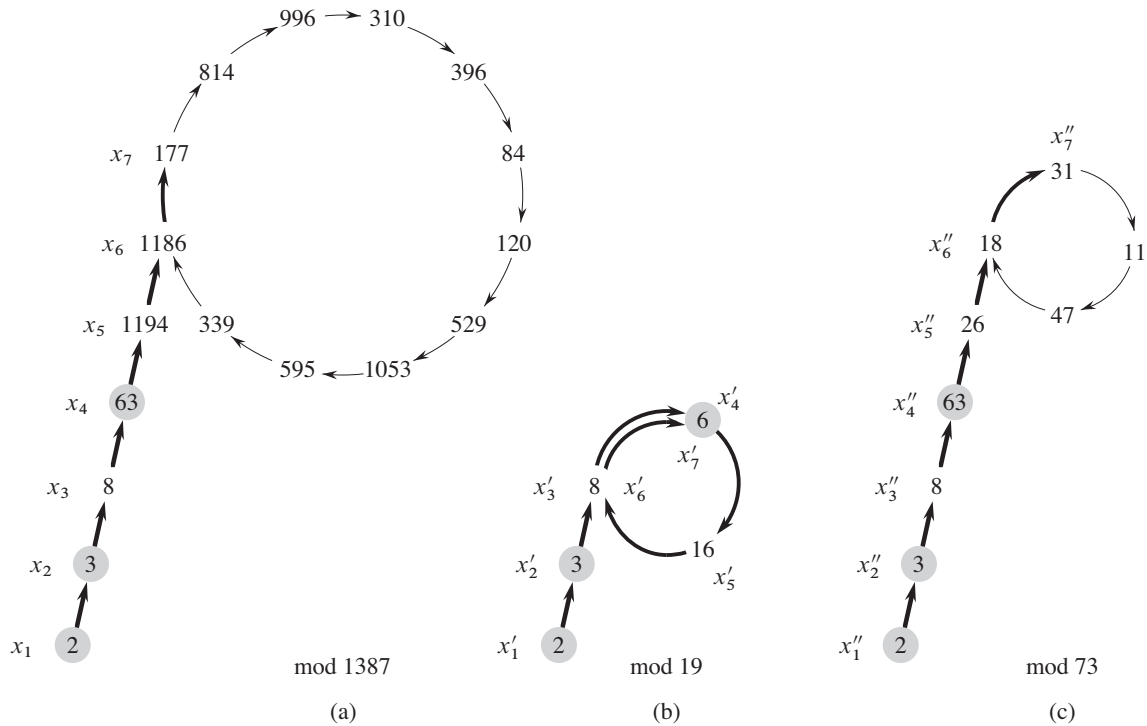


Figure 31.7 Pollard's rho heuristic. **(a)** The values produced by the recurrence $x_{i+1} = (x_i^2 - 1) \bmod 1387$, starting with $x_1 = 2$. The prime factorization of 1387 is $19 \cdot 73$. The heavy arrows indicate the iteration steps that are executed before the factor 19 is discovered. The light arrows point to unreached values in the iteration, to illustrate the “rho” shape. The shaded values are the y values stored by POLLARD-RHO. The factor 19 is discovered upon reaching $x_7 = 177$, when $\gcd(63 - 177, 1387) = 19$ is computed. The first x value that would be repeated is 1186, but the factor 19 is discovered before this value is repeated. **(b)** The values produced by the same recurrence, modulo 19. Every value x_i given in part (a) is equivalent, modulo 19, to the value x'_i shown here. For example, both $x_4 = 63$ and $x_7 = 177$ are equivalent to 6, modulo 19. **(c)** The values produced by the same recurrence, modulo 73. Every value x_i given in part (a) is equivalent, modulo 73, to the value x''_i shown here. By the Chinese remainder theorem, each node in part (a) corresponds to a pair of nodes, one from part (b) and one from part (c).

The sequence $\langle x_i \rangle$ induces a corresponding sequence $\langle x'_i \rangle$ modulo p , where

$$x'_i = x_i \bmod p$$

for all i .

Furthermore, because f_n is defined using only arithmetic operations (squaring and subtraction) modulo n , we can compute x'_{i+1} from x'_i ; the “modulo p ” view of

the sequence is a smaller version of what is happening modulo n :

$$\begin{aligned}
 x'_{i+1} &= x_{i+1} \bmod p \\
 &= f_n(x_i) \bmod p \\
 &= ((x_i^2 - 1) \bmod n) \bmod p \\
 &= (x_i^2 - 1) \bmod p && \text{(by Exercise 31.1-7)} \\
 &= ((x_i \bmod p)^2 - 1) \bmod p \\
 &= ((x'_i)^2 - 1) \bmod p \\
 &= f_p(x'_i).
 \end{aligned}$$

Thus, although we are not explicitly computing the sequence $\langle x'_i \rangle$, this sequence is well defined and obeys the same recurrence as the sequence $\langle x_i \rangle$.

Reasoning as before, we find that the expected number of steps before the sequence $\langle x'_i \rangle$ repeats is $\Theta(\sqrt{p})$. If p is small compared to n , the sequence $\langle x'_i \rangle$ might repeat much more quickly than the sequence $\langle x_i \rangle$. Indeed, as parts (b) and (c) of Figure 31.7 show, the $\langle x'_i \rangle$ sequence repeats as soon as two elements of the sequence $\langle x_i \rangle$ are merely equivalent modulo p , rather than equivalent modulo n .

Let t denote the index of the first repeated value in the $\langle x'_i \rangle$ sequence, and let $u > 0$ denote the length of the cycle that has been thereby produced. That is, t and $u > 0$ are the smallest values such that $x'_{t+i} = x'_{t+u+i}$ for all $i \geq 0$. By the above arguments, the expected values of t and u are both $\Theta(\sqrt{p})$. Note that if $x'_{t+i} = x'_{t+u+i}$, then $p \mid (x_{t+u+i} - x_{t+i})$. Thus, $\gcd(x_{t+u+i} - x_{t+i}, n) > 1$.

Therefore, once POLLARD-RHO has saved as y any value x_k such that $k \geq t$, then $y \bmod p$ is always on the cycle modulo p . (If a new value is saved as y , that value is also on the cycle modulo p .) Eventually, k is set to a value that is greater than u , and the procedure then makes an entire loop around the cycle modulo p without changing the value of y . The procedure then discovers a factor of n when x_i “runs into” the previously stored value of y , modulo p , that is, when $x_i \equiv y \pmod{p}$.

Presumably, the factor found is the factor p , although it may occasionally happen that a multiple of p is discovered. Since the expected values of both t and u are $\Theta(\sqrt{p})$, the expected number of steps required to produce the factor p is $\Theta(\sqrt{p})$.

This algorithm might not perform quite as expected, for two reasons. First, the heuristic analysis of the running time is not rigorous, and it is possible that the cycle of values, modulo p , could be much larger than \sqrt{p} . In this case, the algorithm performs correctly but much more slowly than desired. In practice, this issue seems to be moot. Second, the divisors of n produced by this algorithm might always be one of the trivial factors 1 or n . For example, suppose that $n = pq$, where p and q are prime. It can happen that the values of t and u for p are identical with the values of t and u for q , and thus the factor p is always revealed in the same gcd operation that reveals the factor q . Since both factors are revealed at the same

time, the trivial factor $pq = n$ is revealed, which is useless. Again, this problem seems to be insignificant in practice. If necessary, we can restart the heuristic with a different recurrence of the form $x_{i+1} = (x_i^2 - c) \bmod n$. (We should avoid the values $c = 0$ and $c = 2$ for reasons we will not go into here, but other values are fine.)

Of course, this analysis is heuristic and not rigorous, since the recurrence is not really “random.” Nonetheless, the procedure performs well in practice, and it seems to be as efficient as this heuristic analysis indicates. It is the method of choice for finding small prime factors of a large number. To factor a β -bit composite number n completely, we only need to find all prime factors less than $\lfloor n^{1/2} \rfloor$, and so we expect POLLARD-RHO to require at most $n^{1/4} = 2^{\beta/4}$ arithmetic operations and at most $n^{1/4}\beta^2 = 2^{\beta/4}\beta^2$ bit operations. POLLARD-RHO’s ability to find a small factor p of n with an expected number $\Theta(\sqrt{p})$ of arithmetic operations is often its most appealing feature.

Exercises

31.9-1

Referring to the execution history shown in Figure 31.7(a), when does POLLARD-RHO print the factor 73 of 1387?

31.9-2

Suppose that we are given a function $f : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$ and an initial value $x_0 \in \mathbb{Z}_n$. Define $x_i = f(x_{i-1})$ for $i = 1, 2, \dots$. Let t and $u > 0$ be the smallest values such that $x_{t+i} = x_{t+u+i}$ for $i = 0, 1, \dots$. In the terminology of Pollard’s rho algorithm, t is the length of the tail and u is the length of the cycle of the rho. Give an efficient algorithm to determine t and u exactly, and analyze its running time.

31.9-3

How many steps would you expect POLLARD-RHO to require to discover a factor of the form p^e , where p is prime and $e > 1$?

31.9-4 ★

One disadvantage of POLLARD-RHO as written is that it requires one gcd computation for each step of the recurrence. Instead, we could batch the gcd computations by accumulating the product of several x_i values in a row and then using this product instead of x_i in the gcd computation. Describe carefully how you would implement this idea, why it works, and what batch size you would pick as the most effective when working on a β -bit number n .

Problems
31-1 Binary gcd algorithm

Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the *binary gcd algorithm*, which avoids the remainder computations used in Euclid's algorithm.

- a. Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.
- b. Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.
- c. Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.
- d. Design an efficient binary gcd algorithm for input integers a and b , where $a \geq b$, that runs in $O(\lg a)$ time. Assume that each subtraction, parity test, and halving takes unit time.

31-2 Analysis of bit operations in Euclid's algorithm

- a. Consider the ordinary "paper and pencil" algorithm for long division: dividing a by b , which yields a quotient q and remainder r . Show that this method requires $O((1 + \lg q) \lg b)$ bit operations.
- b. Define $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Show that the number of bit operations performed by EUCLID in reducing the problem of computing $\gcd(a, b)$ to that of computing $\gcd(b, a \bmod b)$ is at most $c(\mu(a, b) - \mu(b, a \bmod b))$ for some sufficiently large constant $c > 0$.
- c. Show that EUCLID(a, b) requires $O(\mu(a, b))$ bit operations in general and $O(\beta^2)$ bit operations when applied to two β -bit inputs.

31-3 Three algorithms for Fibonacci numbers

This problem compares the efficiency of three methods for computing the n th Fibonacci number F_n , given n . Assume that the cost of adding, subtracting, or multiplying two numbers is $O(1)$, independent of the size of the numbers.

- a. Show that the running time of the straightforward recursive method for computing F_n based on recurrence (3.22) is exponential in n . (See, for example, the FIB procedure on page 775.)
- b. Show how to compute F_n in $O(n)$ time using memoization.

- c. Show how to compute F_n in $O(\lg n)$ time using only integer addition and multiplication. (*Hint:* Consider the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

and its powers.)

- d. Assume now that adding two β -bit numbers takes $\Theta(\beta)$ time and that multiplying two β -bit numbers takes $\Theta(\beta^2)$ time. What is the running time of these three methods under this more reasonable cost measure for the elementary arithmetic operations?

31-4 Quadratic residues

Let p be an odd prime. A number $a \in \mathbb{Z}_p^*$ is a **quadratic residue** if the equation $x^2 = a \pmod{p}$ has a solution for the unknown x .

- a. Show that there are exactly $(p-1)/2$ quadratic residues, modulo p .
- b. If p is prime, we define the **Legendre symbol** $\left(\frac{a}{p}\right)$, for $a \in \mathbb{Z}_p^*$, to be 1 if a is a quadratic residue modulo p and -1 otherwise. Prove that if $a \in \mathbb{Z}_p^*$, then

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Give an efficient algorithm that determines whether a given number a is a quadratic residue modulo p . Analyze the efficiency of your algorithm.

- c. Prove that if p is a prime of the form $4k+3$ and a is a quadratic residue in \mathbb{Z}_p^* , then $a^{k+1} \pmod{p}$ is a square root of a , modulo p . How much time is required to find the square root of a quadratic residue a modulo p ?
- d. Describe an efficient randomized algorithm for finding a nonquadratic residue, modulo an arbitrary prime p , that is, a member of \mathbb{Z}_p^* that is not a quadratic residue. How many arithmetic operations does your algorithm require on average?

Chapter notes

Niven and Zuckerman [265] provide an excellent introduction to elementary number theory. Knuth [210] contains a good discussion of algorithms for finding the

greatest common divisor, as well as other basic number-theoretic algorithms. Bach [30] and Riesel [295] provide more recent surveys of computational number theory. Dixon [91] gives an overview of factorization and primality testing. The conference proceedings edited by Pomerance [280] contains several excellent survey articles. More recently, Bach and Shallit [31] have provided an exceptional overview of the basics of computational number theory.

Knuth [210] discusses the origin of Euclid's algorithm. It appears in Book 7, Propositions 1 and 2, of the Greek mathematician Euclid's *Elements*, which was written around 300 B.C. Euclid's description may have been derived from an algorithm due to Eudoxus around 375 B.C. Euclid's algorithm may hold the honor of being the oldest nontrivial algorithm; it is rivaled only by an algorithm for multiplication known to the ancient Egyptians. Shallit [312] chronicles the history of the analysis of Euclid's algorithm.

Knuth attributes a special case of the Chinese remainder theorem (Theorem 31.27) to the Chinese mathematician Sun-Tsü, who lived sometime between 200 B.C. and A.D. 200—the date is quite uncertain. The same special case was given by the Greek mathematician Nichomachus around A.D. 100. It was generalized by Chhin Chiu-Shao in 1247. The Chinese remainder theorem was finally stated and proved in its full generality by L. Euler in 1734.

The randomized primality-testing algorithm presented here is due to Miller [255] and Rabin [289]; it is the fastest randomized primality-testing algorithm known, to within constant factors. The proof of Theorem 31.39 is a slight adaptation of one suggested by Bach [29]. A proof of a stronger result for MILLER-RABIN was given by Monier [258, 259]. For many years primality-testing was the classic example of a problem where randomization appeared to be necessary to obtain an efficient (polynomial-time) algorithm. In 2002, however, Agrawal, Kayal, and Saxena [4] surprised everyone with their deterministic polynomial-time primality-testing algorithm. Until then, the fastest deterministic primality testing algorithm known, due to Cohen and Lenstra [73], ran in time $(\lg n)^{O(\lg \lg n)}$ on input n , which is just slightly superpolynomial. Nonetheless, for practical purposes randomized primality-testing algorithms remain more efficient and are preferred.

The problem of finding large “random” primes is nicely discussed in an article by Beauchemin, Brassard, Crépeau, Goutier, and Pomerance [36].

The concept of a public-key cryptosystem is due to Diffie and Hellman [87]. The RSA cryptosystem was proposed in 1977 by Rivest, Shamir, and Adleman [296]. Since then, the field of cryptography has blossomed. Our understanding of the RSA cryptosystem has deepened, and modern implementations use significant refinements of the basic techniques presented here. In addition, many new techniques have been developed for proving cryptosystems to be secure. For example, Goldwasser and Micali [142] show that randomization can be an effective tool in the design of secure public-key encryption schemes. For signature schemes,

Goldwasser, Micali, and Rivest [143] present a digital-signature scheme for which every conceivable type of forgery is provably as difficult as factoring. Menezes, van Oorschot, and Vanstone [254] provide an overview of applied cryptography.

The rho heuristic for integer factorization was invented by Pollard [277]. The version presented here is a variant proposed by Brent [56].

The best algorithms for factoring large numbers have a running time that grows roughly exponentially with the cube root of the length of the number n to be factored. The general number-field sieve factoring algorithm (as developed by Buhler, Lenstra, and Pomerance [57] as an extension of the ideas in the number-field sieve factoring algorithm by Pollard [278] and Lenstra et al. [232] and refined by Coppersmith [77] and others) is perhaps the most efficient such algorithm in general for large inputs. Although it is difficult to give a rigorous analysis of this algorithm, under reasonable assumptions we can derive a running-time estimate of $L(1/3, n)^{1.902+o(1)}$, where $L(\alpha, n) = e^{(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$.

The elliptic-curve method due to Lenstra [233] may be more effective for some inputs than the number-field sieve method, since, like Pollard's rho method, it can find a small prime factor p quite quickly. With this method, the time to find p is estimated to be $L(1/2, p)^{\sqrt{2}+o(1)}$.