15 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, and 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 leads 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 14, particularly Section 14.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We first examine, in Section 15.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We'll arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that an optimal solution can result from always making greedy choices. Section 15.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 15.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. Finally, Section 15.4 shows that in order to decide which blocks to replace when a miss occurs in a cache, the "furthest-in-future" strategy is optimal if the sequence of block accesses is known in advance.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that you can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 21), Dijkstra's algorithm for shortest paths from a single source (Section 22.3), and a greedy set-covering heuristic (Section 35.3). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read this chapter and Chapter 21 independently of each other, you might find it useful to read them together.

15.1 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. Imagine that you are in charge of scheduling a conference room. You are presented with a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to reserve the conference room, and the room 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 $0 \le s_i < f_i < \infty$. 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 $s_i \ge f_j$ or $s_j \ge f_i$. (Assume that if your staff needs time to change over the room from one activity to the next, the changeover time is built into the intervals.) In the ***activity-selection problem***, your goal is to select a maximum-size subset of mutually compatible activities. Assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n \,. \tag{15.1}$$

(We'll see later the advantage that this assumption provides.) For example, consider the set of activities in Figure 15.1. 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, and another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We'll see how to solve this problem, proceeding in several steps. First we'll explore a dynamic-programming solution, in which you consider several choices when determining which subproblems to use in an optimal solution. We'll then observe that you need to consider only one choice—the greedy choice—and that when you make the greedy choice, only one subproblem remains. Based on these observations, we'll develop a recursive greedy algorithm to solve the activity-selection problem. Finally, we'll complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. Although the steps we 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.

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

Figure 15.1 A set $\{a_1, a_2, \ldots, a_{11}\}$ of activities. Activity a_i has start time s_i and finish time f_i.

The optimal substructure of the activity-selection problem

Let's verify that the activity-selection problem exhibits optimal substructure. 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 you want 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, you 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 an optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If you could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then you could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. You 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 you can solve the activity-selection problem by dynamic programming. Let's denote the size of an optimal solution for the set S_{ij} by $c[i, j]$. Then, the dynamic-programming approach gives the recurrence

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

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

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max \{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset . \end{cases} \tag{15.2}$$

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

Making the greedy choice

What if you could choose an activity to add to an optimal solution without having to first solve all the subproblems? That could save you from having to consider all the choices inherent in recurrence (15.2). In fact, for the activity-selection problem, you need to consider only one choice: the greedy choice.

What is the greedy choice for the activity-selection problem? Intuition suggests that you should choose an activity that leaves the resource available for as many other activities as possible. Of the activities you end up choosing, one of them must be the first one to finish. Intuition says, therefore, choose the activity in S with the earliest finish time, since that leaves 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 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 15.1-3 asks you to explore other possibilities.

Once you make the greedy choice, you have only one remaining subproblem to solve: finding activities that start after a_1 finishes. Why don't you have to consider activities that finish before a_1 starts? Because $s_1 < f_1$, and because f_1 is the earliest finish time of any activity, 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 you make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.[1] Optimal substructure says that if a_1 belongs to an 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 this intuition correct? Is the greedy choice—in which you choose the first activity to finish—always part of some optimal solution? The following theorem shows that it is.

---

[1] We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. The context will make it clear whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem 15.1
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 belongs to 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 compatible, which follows because the activities in A_k are compatible, 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. ■

Although you might be able to solve the activity-selection problem with dynamic programming, Theorem 15.1 says that you don't need to. Instead, you can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain. Moreover, because you always choose the activity with the earliest finish time, the finish times of the activities that you choose must strictly increase. You 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 that it constructs 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 you know you can bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, let's see a straightforward, recursive procedure to solve the activity-selection problem. The procedure RECURSIVE-ACTIVITY-SELECTOR on the following page 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 the size n of the original problem. It returns a maximum-

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

size set of mutually compatible activities in S_k. The procedure assumes that the n input activities are already ordered by monotonically increasing finish time, according to equation (15.1). If not, you can first sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily. In order to start, 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 ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3       m = m + 1
4  if m ≤ n
5       return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Figure 15.2 shows how the algorithm operates on the activities in Figure 15.1. 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 $a_{k+1}, a_{k+2}, \ldots, a_n$, until it finds the first activity a_m that is compatible with a_k, which means that $s_m \geq f_k$. 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 the procedure has examined all activities in S_k without finding one that is compatible with a_k. In this case, $S_k = \emptyset$, and so line 6 returns $\emptyset$.

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 $\Theta(n)$. To see why, observe that 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

The recursive procedure can be converted to an iterative one because the procedure RECURSIVE-ACTIVITY-SELECTOR is almost "tail recursive" (see Problem 7-5): 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.

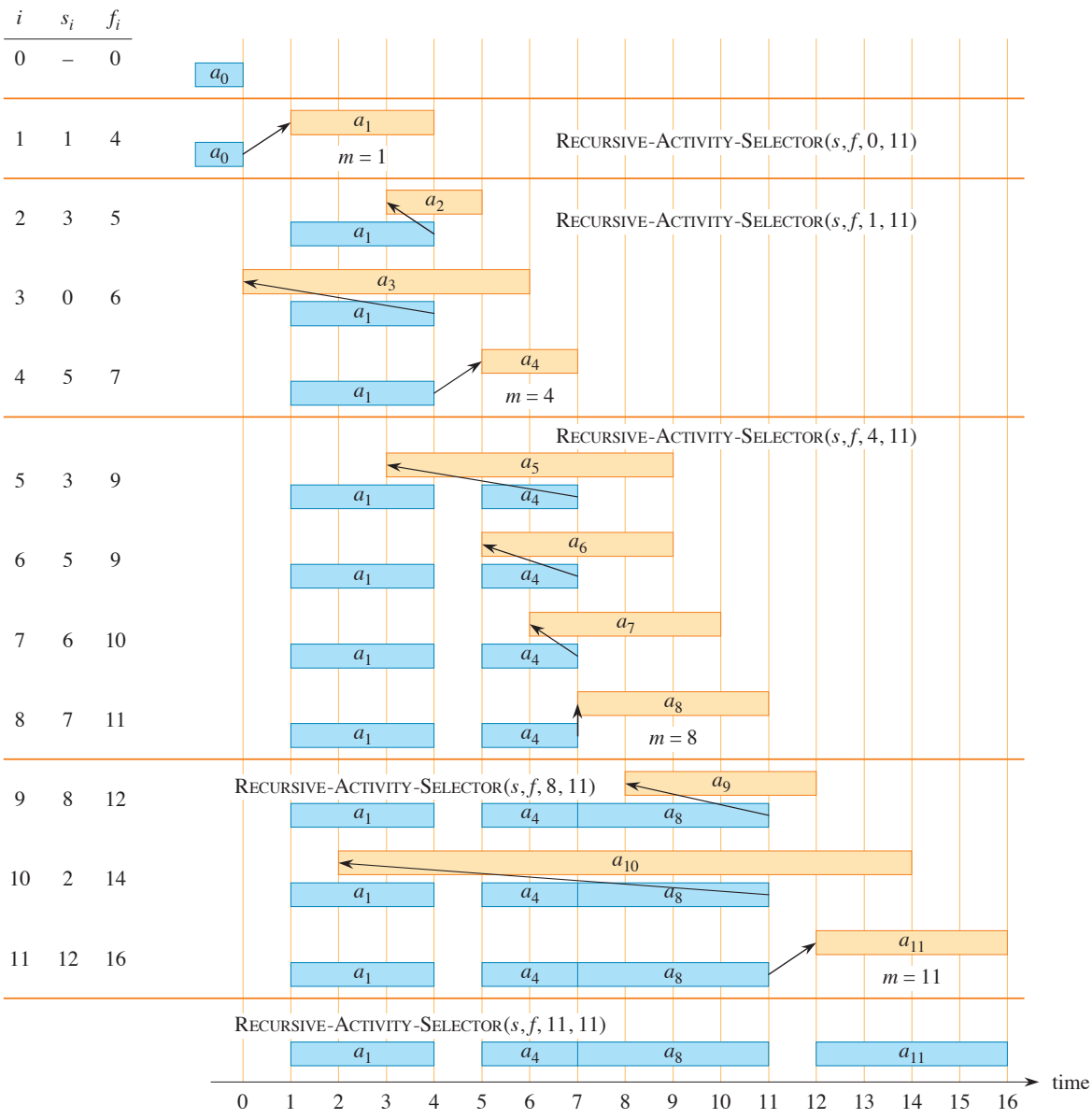

Figure 15.2 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities from Figure 15.1. Activities considered in each recursive call appear between horizontal lines. The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, 11)$, selects activity a_1. In each recursive call, the activities that have already been selected are blue, and the activity shown in tan 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, too, 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, n)
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4      if s[m] ≥ f[k]          // is a_m in S_k?
5          A = A ∪ {a_m}       // yes, so choose it
6          k = m               // and continue from there
7  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 the procedure considers the activities in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A. That is,

$$f_k = \max\{f_i : a_i \in A\} \ . \tag{15.3}$$

Lines 1–2 select activity a_1, initialize A to contain just this activity, and initialize k to index this activity. The **for** loop of lines 3–6 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 (15.3) to check (in line 4) 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 5–6 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 initial 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

15.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (15.2). 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 (15.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

15.1-2

Suppose that instead of always selecting the first activity to finish, you 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.

15.1-3

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.

15.1-4

You are given a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. You 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***. It is modeled by 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.)

15.1-5

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, the goal is 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.

15.2 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 the choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as 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 15.1 to develop a greedy algorithm was a bit more involved than is typical. It consisted of the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (15.2), but bypassed developing a recursive algorithm based solely on this recurrence.)
3. Show that if you 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.

These steps highlighted in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, the first cut at the activity-selection problem defined the subproblems S_{ij}, where both i and j varied. We then found that if you always make the greedy choice, you can restrict the subproblems to be of the form S_k.

An alternative approach is to fashion optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, start by dropping the second subscript and defining subproblems of the form S_k. Then prove 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, you can design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which you 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 you combine an

optimal solution to the subproblem with the greedy choice you have made, you arrive at an optimal solution to the original problem.

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

How can you 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 you can demonstrate that the problem has these properties, then you are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the ***greedy-choice property***: you can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when you are considering which choice to make, you 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, you make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, you typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. (Alternatively, you can solve them top down, but memoizing. Of course, even though the code works top down, you still must solve the subproblems before making a choice.) In a greedy algorithm, you 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 top down, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, you need to prove that a greedy choice at each step yields a globally optimal solution. Typically, as in the case of Theorem 15.1, 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.

You can usually make the greedy choice more efficiently than when you have to consider a wider set of choices. For example, in the activity-selection problem, assuming that the activities were already sorted in monotonically increasing order by finish times, each activity needed to be examined just once. By preprocessing

the input or by using an appropriate data structure (often a priority queue), you often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

As we saw in Chapter 14, 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 whether dynamic programming applies, and it's also essential for greedy algorithms. As an example of optimal substructure, recall how Section 15.1 demonstrated 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 you know which activity to use as a_k, you can 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}. This observation of optimal substructure gave rise to the recurrence (15.2) that describes the value of an optimal solution.

You will usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, you have the luxury of assuming that you arrived at a subproblem by having made the greedy choice in the original problem. All you 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 subtle differences between the two techniques, let's investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem*** is the following. A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. Which items should the thief take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind. The thief 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, if the most valuable load weighing at most W pounds includes item j, then the remaining load must be the most valuable load weighing at most $W - w_j$ pounds that the thief can take from the $n - 1$ original items excluding item j. For the comparable fractional problem, if if the most valuable load weighing at most W pounds includes weight w of item j, then the remaining load must be the most valuable load weighing at most $W - w$ pounds that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Although the problems are similar, a greedy strategy works to solve the fractional knapsack problem, but not the 0-1 problem. To solve the fractional problem, 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 the thief can still carry more, then the thief takes as much as possible of the item with the next greatest value per pound, and so forth, until reaching the weight limit W. Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. You are asked to prove that the fractional knapsack problem has the greedy-choice property in Exercise 15.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 15.3(a). This example has three items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth \$60. Item 2 weighs 20 pounds and is worth \$100. Item 3 weighs 30 pounds and is worth \$120. Thus, the value per pound of item 1 is \$6 per pound, which is greater than the value per pound of either item 2 (\$5 per pound) or item 3 (\$4 per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 15.3(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 15.3(c). Taking item 1 doesn't work in the 0-1 problem, because the thief is unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the load. In the 0-1 problem, when you consider whether to include an item in the knapsack, you must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before you can make the choice. The problem formulated in this way gives rise to many overlapping sub-

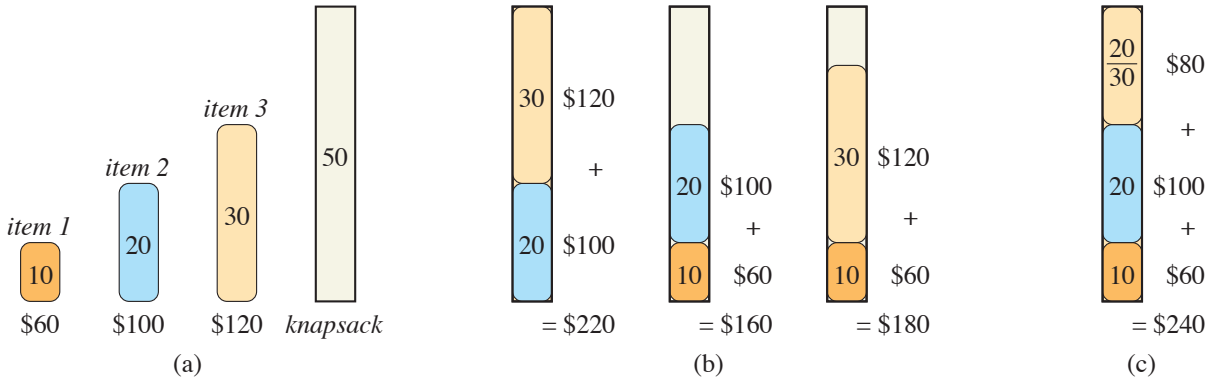

Figure 15.3 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though 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.

problems—a hallmark of dynamic programming, and indeed, as Exercise 15.2-2 asks you to show, you can use dynamic programming to solve the 0-1 problem.

Exercises

15.2-1

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

15.2-2

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

15.2-3

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.

15.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. The professor 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 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. The professor has an official North Dakota state map, which shows all the places along U.S. 2 to refill water and the distances between these locations.

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

15.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ 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.

★ ***15.2-6***

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

15.2-7

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 ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an algorithm that maximizes your payoff. Prove that your algorithm maximizes the payoff, and state its running time, omitting the time for reordering the sets.

15.3 Huffman codes

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

Suppose that you have a 100,000-character data file that you wish to store compactly and you know that the 6 distinct characters in the file occur with the frequencies given by Figure 15.4. The character `a` occurs 45,000 times, the character `b` occurs 13,000 times, and so on.

You 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 15.4 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. With each character represented by a 3-bit codeword, encoding the file requires 300,000 bits. With the variable-length code shown, the encoding requires only 224,000 bits.

in which each character is represented by a unique binary string, which we call a ***codeword***. If you use a ***fixed-length code***, you need $\lceil \lg n \rceil$ bits to represent $n \geq 2$ characters. For 6 characters, therefore, you need 3 bits: a = 000, b = 001, c = 010, d = 011, e = 100, and f = 101. This method requires 300,000 bits to encode the entire file. Can you do better?

A ***variable-length code*** can do considerably better than a fixed-length code. The idea is simple: give frequent characters short codewords and infrequent characters long codewords. Figure 15.4 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-free codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix-free codes***. Although we won't prove it here, a prefix-free 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-free codes.

Encoding is always simple for any binary character code: just concatenate the codewords representing each character of the file. For example, with the variable-length prefix-free code of Figure 15.4, the 4-character file face has the encoding $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where "$\cdot$" denotes concatenation.

Prefix-free 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. You can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 100011001101 parses uniquely as $100 \cdot 0 \cdot 1100 \cdot 1101$, which decodes to cafe.

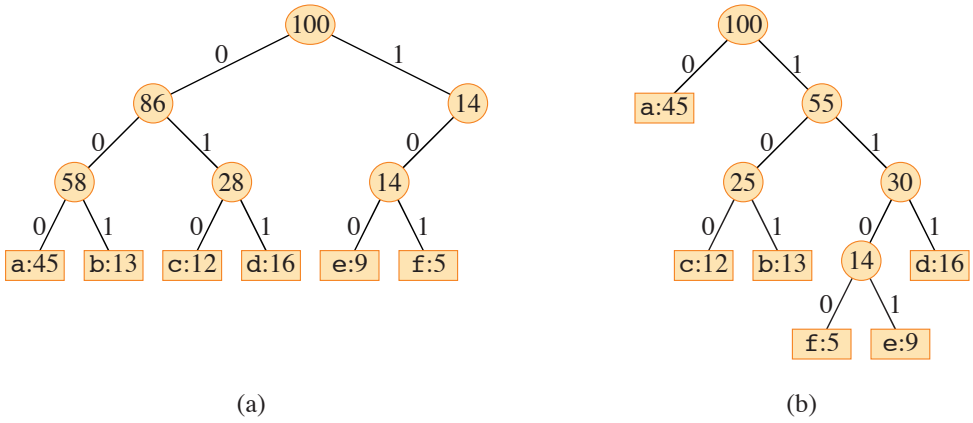

Figure 15.5 Trees corresponding to the coding schemes in Figure 15.4. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. All frequencies are in thousands. **(a)** The tree corresponding to the fixed-length code $\texttt{a} = 000$, $\texttt{b} = 001$, $\texttt{c} = 010$, $\texttt{d} = 011$, $\texttt{e} = 100$, $\texttt{f} = 101$. **(b)** The tree corresponding to the optimal prefix-free code $\texttt{a} = 0$, $\texttt{b} = 101$, $\texttt{c} = 100$, $\texttt{d} = 111$, $\texttt{e} = 1101$, $\texttt{f} = 1100$.

The decoding process needs a convenient representation for the prefix-free code so that you can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. 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 15.5 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 15.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 15.5(a), is not a full binary tree: it contains codewords beginning with 10, but none beginning with 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-free code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3 on page 1175).

Given a tree T corresponding to a prefix-free code, we can 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

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \,, \tag{15.4}$$

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

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix-free code, called a ***Huffman code*** in his honor. In line with our observations in Section 15.2, 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.

The procedure HUFFMAN assumes that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds the tree T corresponding to an 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. The result of merging two objects 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       x = EXTRACT-MIN(Q)
 6       y = EXTRACT-MIN(Q)
 7       z.left = x
 8       z.right = y
 9       z.freq = x.freq + y.freq
10       INSERT(Q, z)
11  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

For our example, Huffman's algorithm proceeds as shown in Figure 15.6. 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-free code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

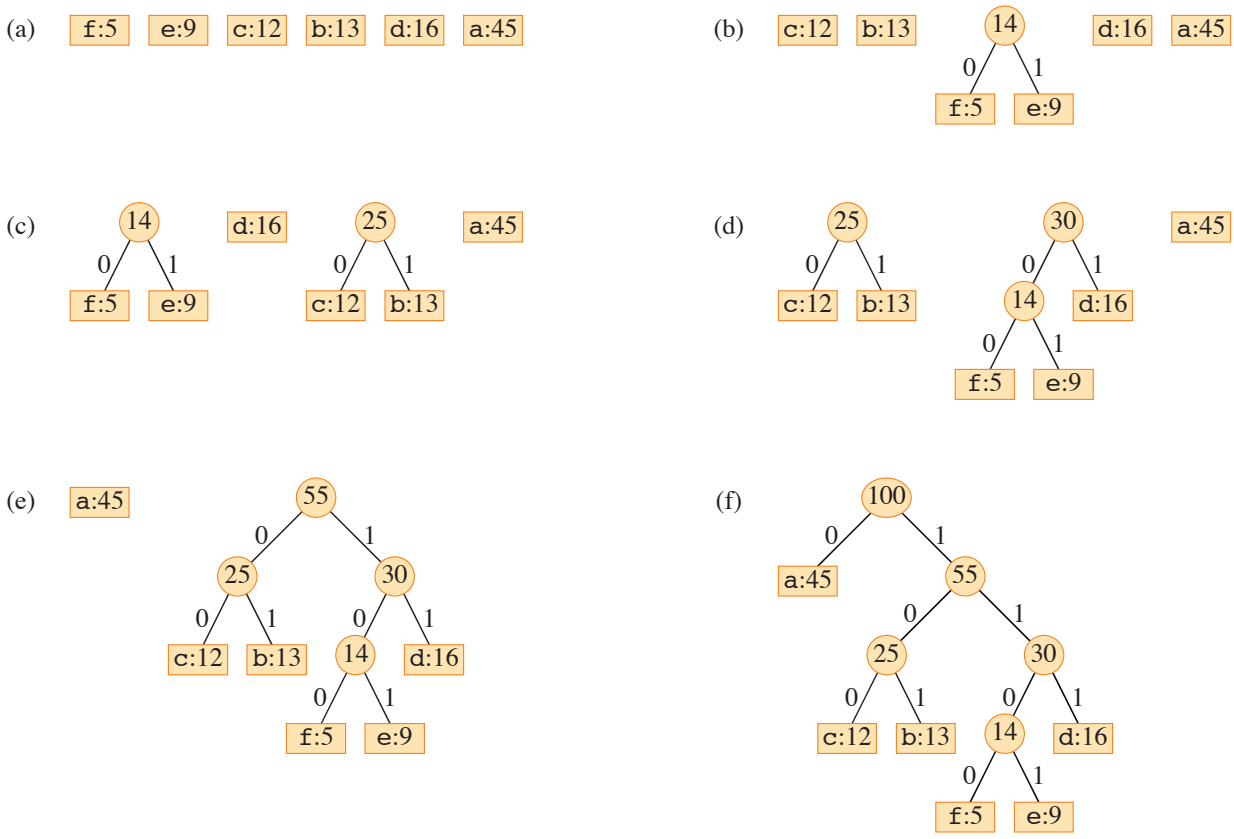

Figure 15.6 The steps of Huffman's algorithm for the frequencies given in Figure 15.4. Each part shows the contents of the queue sorted into increasing order by frequency. Each step merges the two trees with the lowest frequencies. Leaves are shown as rectangles containing a character and its frequency. 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.

The HUFFMAN procedure works as follows. Line 2 initializes the min-priority queue Q with the characters in C. The **for** loop in lines 3–10 repeatedly extracts the two nodes x and y of lowest frequency from the queue and replaces 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 9. 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 11 returns the one node left in the queue, which is the root of the code tree.

The algorithm produces the same result without the variables x and y, assigning the values returned by the EXTRACT-MIN calls directly to $z.left$ and $z.right$ in lines 7 and 8, and changing line 9 to $z.freq = z.left.freq + z.right.freq$. We'll use the node names x and y in the proof of correctness, however, so we leave them in.

The running time of Huffman's algorithm depends on how the min-priority queue Q is implemented. Let's assume that it's implemented as a binary min-heap (see Chapter 6). For a set C of n characters, the BUILD-MIN-HEAP procedure discussed in Section 6.3 can initialize Q in line 2 in $O(n)$ time. The **for** loop in lines 3–10 executes exactly $n - 1$ times, and since each heap operation runs in $O(\lg n)$ time, 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 $O(n \lg n)$.

Correctness of Huffman's algorithm

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

Lemma 15.2 (Optimal prefix-free codes have the greedy-choice property)
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-free 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-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. In such a tree, the codewords for x and y have the same length and differ only in the last bit.

Let a and b be any two characters that are sibling leaves of maximum depth in T. Without loss of generality, assume that $a.freq \le b.freq$ and $x.freq \le y.freq$. 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 $x.freq \le a.freq$ and $y.freq \le b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$, but $x.freq = b.freq$ implies that $a.freq = b.freq = x.freq = y.freq$ (see Exercise 15.3-1), and the lemma would be trivially true. Therefore, assume that $x.freq \ne b.freq$, which means that $x \ne b$.

As Figure 15.7 shows, imagine exchanging the positions in T of a and x to produce a tree T', and then exchanging the positions in T' of b and y to produce a

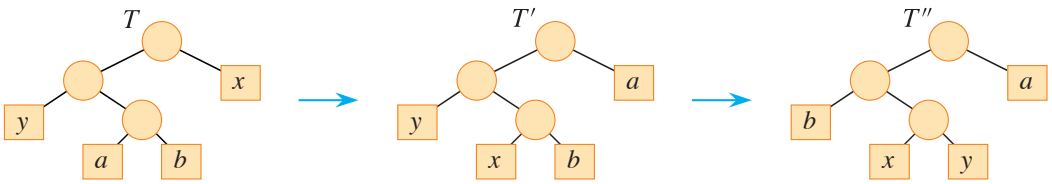

Figure 15.7 An illustration of the key step in the proof of Lemma 15.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies. They appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and 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.

tree T'' 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 (15.4), the difference in cost between T and T' is

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

because both $a.freq - x.freq$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.freq - x.freq$ 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, $B(T'') \leq B(T') \leq B(T)$, 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 15.2 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 15.3-4 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-free codes has the optimal-substructure property.

Lemma 15.3 (Optimal prefix-free codes have the optimal-substructure property)
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 $freq$ for all characters in C' with the same values as in C, along with $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix-free code for 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-free 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 (15.4). 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-free code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 15.2), 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-free code for C'. Thus, T must represent an optimal prefix-free code for the alphabet C. ■

Theorem 15.4
Procedure HUFFMAN produces an optimal prefix-free code.

Proof Immediate from Lemmas 15.2 and 15.3. ■

Exercises

15.3-1
Explain why, in the proof of Lemma 15.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

15.3-2
Prove that a non-full binary tree cannot correspond to an optimal prefix-free code.

15.3-3
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?

15.3-4
Prove that the total cost $B(T)$ of a full binary tree T for a code equals the sum, over all internal nodes, of the combined frequencies of the two children of the node.

15.3-5
Given an optimal prefix-free code on a set C of n characters, you wish to transmit the code itself using as few bits as possible. Show how to represent any optimal prefix-free 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.)

15.3-6
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.

15.3-7
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.

15.3-8
Show that no lossless (invertible) compression scheme can guarantee that for every input file, the corresponding output file is shorter. (*Hint:* Compare the number of possible files with the number of possible encoded files.)

15.4 Offline caching

Computer systems can decrease the time to access data by storing a subset of the main memory in the ***cache***: a small but faster memory. A cache organizes data into ***cache blocks*** typically comprising 32, 64, or 128 bytes. You can also think of main memory as a cache for disk-resident data in a virtual-memory system. Here, the blocks are called ***pages***, and 4096 bytes is a typical size.

As a computer program executes, it makes a sequence of memory requests. Say that there are n memory requests, to data in blocks $b_1, b_2, \ldots, b_n$, in that order. The blocks in the access sequence might not be distinct, and indeed, any given block is usually accessed multiple times. For example, a program that accesses four distinct blocks p, q, r, s might make a sequence of requests to blocks $s, q, s, q, q, s, p, p, r, s, s, q, p, r, q$. The cache can hold up to some fixed number k of cache blocks. It starts out empty before the first request. Each request causes at most one block to enter the cache and at most one block to be evicted from the cache. Upon a request for block b_i, any one of three scenarios may occur:

1. Block b_i is already in the cache, due to a previous request for the same block. The cache remains unchanged. This situation is known as a ***cache hit***.
2. Block b_i is not in the cache at that time, but the cache contains fewer than k blocks. In this case, block b_i is placed into the cache, so that the cache contains one more block than it did before the request.
3. Block b_i is not in the cache at that time and the cache is full: it contains k blocks. Block b_i is placed into the cache, but before that happens, some other block in the cache must be evicted from the cache in order to make room.

The latter two situations, in which the requested block is not already in the cache, are called ***cache misses***. The goal is to minimize the number of cache misses or, equivalently, to maximize the number of cache hits, over the entire sequence of n requests. A cache miss that occurs while the cache holds fewer than k blocks—that is, as the cache is first being filled up—is known as a ***compulsory miss***, since no prior decision could have kept the requested block in the cache. When a cache miss occurs and the cache is full, ideally the choice of which block to evict should allow for the smallest possible number of cache misses over the entire sequence of future requests.

Typically, caching is an online problem. That is, the computer has to decide which blocks to keep in the cache without knowing the future requests. Here, however, let's consider the offline version of this problem, in which the computer knows in advance the entire sequence of n requests and the cache size k, with a goal of minimizing the total number of cache misses.

To solve this offline problem, you can use a greedy strategy called ***furthest-in-future***, which chooses to evict the block in the cache whose next access in the request sequence comes furthest in the future. Intuitively, this strategy makes sense: if you're not going to need something for a while, why keep it around? We'll show that the furthest-in-future strategy is indeed optimal by showing that the offline caching problem exhibits optimal substructure and that furthest-in-future has the greedy-choice property.

Now, you might be thinking that since the computer usually doesn't know the sequence of requests in advance, there is no point in studying the offline problem. Actually, there is. In some situations, you do know the sequence of requests in advance. For example, if you view the main memory as the cache and the full set of data as residing on disk (or a solid-state drive), there are algorithms that plan out the entire set of reads and writes in advance. Furthermore, we can use the number of cache misses produced by an optimal algorithm as a baseline for comparing how well online algorithms perform. We'll do just that in Section 27.3.

Offline caching can even model real-world problems. For example, consider a scenario where you know in advance a fixed schedule of n events at known locations. Events may occur at a location multiple times, not necessarily consecutively. You are managing a group of k agents, you need to ensure that you have one agent at each location when an event occurs, and you want to minimize the number of times that agents have to move. Here, the agents are like the blocks, the events are like the requests, and moving an agent is akin to a cache miss.

Optimal substructure of offline caching

To show that the offline problem exhibits optimal substructure, let's define the subproblem (C, i) as processing requests for blocks $b_i, b_{i+1}, \ldots, b_n$ with cache configuration C at the time that the request for block b_i occurs, that is, C is a subset of the set of blocks such that $|C| \le k$. A solution to subproblem (C, i) is a sequence of decisions that specifies which block to evict (if any) upon each request for blocks $b_i, b_{i+1}, \ldots, b_n$. An optimal solution to subproblem (C, i) minimizes the number of cache misses.

Consider an optimal solution S to subproblem (C, i), and let C' be the contents of the cache after processing the request for block b_i in solution S. Let S' be the subsolution of S for the resulting subproblem $(C', i + 1)$. If the request for b_i results in a cache hit, then the cache remains unchanged, so that $C' = C$. If the request for block b_i results in a cache miss, then the contents of the cache change, so that $C' \neq C$. We claim that in either case, S' is an optimal solution to subproblem $(C', i + 1)$. Why? If S' is not an optimal solution to subproblem $(C', i + 1)$, then there exists another solution S'' to subproblem $(C', i + 1)$ that makes fewer cache misses than S'. Combining S'' with the decision of S at the request for

block b_i yields another solution that makes fewer cache misses than S, which contradicts the assumption that S is an optimal solution to subproblem (C, i).

To quantify a recursive solution, we need a little more notation. Let $R_{C,i}$ be the set of all cache configurations that can immediately follow configuration C after processing a request for block b_i. If the request results in a cache hit, then the cache remains unchanged, so that $R_{C,i} = \{C\}$. If the request for b_i results in a cache miss, then there are two possibilities. If the cache is not full ($|C| < k$), then the cache is filling up and the only choice is to insert b_i into the cache, so that $R_{C,i} = \{C \cup \{b_i\}\}$. If the cache is full ($|C| = k$) upon a cache miss, then $R_{C,i}$ contains k potential configurations: one for each candidate block in C that could be evicted and replaced by block b_i. In this case, $R_{C,i} = \{(C - \{x\}) \cup \{b_i\} : x \in C\}$. For example, if $C = \{p, q, r\}$, $k = 3$, and block s is requested, then $R_{C,i} = \{\{p, q, s\}, \{p, r, s\}, \{q, r, s\}\}$.

Let $miss(C, i)$ denote the minimum number of cache misses in a solution for subproblem (C, i). Here is a recurrence for $miss(C, i)$:

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

Greedy-choice property

To prove that the furthest-in-future strategy yields an optimal solution, we need to show that optimal offline caching exhibits the greedy-choice property. Combined with the optimal-substructure property, the greedy-choice property will prove that furthest-in-future produces the minimum possible number of cache misses.

Theorem 15.5 (Optimal offline caching has the greedy-choice property)
Consider a subproblem (C, i) when the cache C contains k blocks, so that it is full, and a cache miss occurs. When block b_i is requested, let $z = b_m$ be the block in C whose next access is furthest in the future. (If some block in the cache will never again be referenced, then consider any such block to be block z, and add a dummy request for block $z = b_m = b_{n+1}$.) Then evicting block z upon a request for block b_i is included in some optimal solution for the subproblem (C, i).

Proof Let S be an optimal solution to subproblem (C, i). If S evicts block z upon the request for block b_i, then we are done, since we have shown that some optimal solution includes evicting z.

So now suppose that optimal solution S evicts some other block x when block b_i is requested. We'll construct another solution S' to subproblem (C, i) which, upon

the request for b_i, evicts block z instead of x and induces no more cache misses than S does, so that S' is also optimal. Because different solutions may yield different cache configurations, denote by $C_{S,j}$ the configuration of the cache under solution S just before the request for some block b_j, and likewise for solution S' and $C_{S',j}$. We'll show how to construct S' with the following properties:

1. For $j = i+1, \ldots, m$, let $D_j = C_{S,j} \cap C_{S',j}$. Then, $|D_j| \geq k-1$, so that the cache configurations $C_{S,j}$ and $C_{S',j}$ differ by at most one block. If they differ, then $C_{S,j} = D_j \cup \{z\}$ and $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$.
2. For each request of blocks $b_i, \ldots, b_{m-1}$, if solution S has a cache hit, then solution S' also has a cache hit.
3. For all $j > m$, the cache configurations $C_{S,j}$ and $C_{S',j}$ are identical.
4. Over the sequence of requests for blocks $b_i, \ldots, b_m$, the number of cache misses produced by solution S' is at most the number of cache misses produced by solution S.

We'll prove inductively that these properties hold for each request.

1. We proceed by induction on j, for $j = i+1, \ldots, m$. For the base case, the initial caches $C_{S,i}$ and $C_{S',i}$ are identical. Upon the request for block b_i, solution S evicts x and solution S' evicts z. Thus, cache configurations $C_{S,i+1}$ and $C_{S',i+1}$ differ by just one block, $C_{S,i+1} = D_{i+1} \cup \{z\}$, $C_{S',i+1} = D_{i+1} \cup \{x\}$, and $x \neq z$.

   The inductive step defines how solution S' behaves upon a request for block b_j for $i+1 \leq j \leq m-1$. The inductive hypothesis is that property 1 holds when b_j is requested. Because $z = b_m$ is the block in $C_{S,i}$ whose next reference is furthest in the future, we know that $b_j \neq z$. We consider several scenarios:

   - If $C_{S,j} = C_{S',j}$ (so that $|D_j| = k$), then solution S' makes the same decision upon the request for b_j as S makes, so that $C_{S,j+1} = C_{S',j+1}$.
   - If $|D_j| = k-1$ and $b_j \in D_j$, then both caches already contain block b_j, and both solutions S and S' have cache hits. Therefore, $C_{S,j+1} = C_{S,j}$ and $C_{S',j+1} = C_{S',j}$.
   - If $|D_j| = k-1$ and $b_j \notin D_j$, then because $C_{S,j} = D_j \cup \{z\}$ and $b_j \neq z$, solution S has a cache miss. It evicts either block z or some block $w \in D_j$.
     - If solution S evicts block z, then $C_{S,j+1} = D_j \cup \{b_j\}$. There are two cases, depending on whether $b_j = y$:
       - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Thus, $C_{S,j+1} = C_{S',j+1}$.
       - If $b_j \neq y$, then solution S' has a cache miss. It evicts block y, so that $C_{S',j+1} = D_j \cup \{b_j\}$, and again $C_{S,j+1} = C_{S',j+1}$.

- If solution S evicts some block $w \in D_j$, then $C_{S,j+1} = (D_j - \{w\}) \cup \{b_j, z\}$. Once again, there are two cases, depending on whether $b_j = y$:
  - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Since $w \in D_j$ and w was not evicted by solution S', we have $w \in C_{S',j+1}$. Therefore, $w \notin D_{j+1}$ and $b_j \in D_{j+1}$, so that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$. Thus, $C_{S,j+1} = D_{j+1} \cup \{z\}$, $C_{S',j+1} = D_{j+1} \cup \{w\}$, and because $w \neq z$, property 1 holds when block b_{j+1} is requested. (In other words, block w replaces block y in property 1.)
  - If $b_j \neq y$, then solution S' has a cache miss. It evicts block w, so that $C_{S',j+1} = (D_j - \{w\}) \cup \{b_j, y\}$. Therefore, we have that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$ and so $C_{S,j+1} = D_{j+1} \cup \{z\}$ and $C_{S',j+1} = D_{j+1} \cup \{y\}$.

2. In the above discussion about maintaining property 1, solution S may have a cache hit in only the first two cases, and solution S' has a cache hit in these cases if and only if S does.

3. If $C_{S,m} = C_{S',m}$, then solution S' makes the same decision upon the request for block $z = b_m$ as S makes, so that $C_{S,m+1} = C_{S',m+1}$. If $C_{S,m} \neq C_{S',m}$, then by property 1, $C_{S,m} = D_m \cup \{z\}$ and $C_{S',m} = D_m \cup \{y\}$, where $y \neq z$. In this case, solution S has a cache hit, so that $C_{S,m+1} = C_{S,m} = D_m \cup \{z\}$. Solution S' evicts block y and brings in block z, so that $C_{S',m+1} = D_m \cup \{z\} = C_{S,m+1}$. Thus, regardless of whether or not $C_{S,m} = C_{S',m}$, we have $C_{S,m+1} = C_{S',m+1}$, and starting with the request for block b_{m+1}, solution S' simply makes the same decisions as S.

4. By property 2, upon the requests for blocks $b_i, \ldots, b_{m-1}$, whenever solution S has a cache hit, so does S'. Only the request for block $b_m = z$ remains to be considered. If S has a cache miss upon the request for b_m, then regardless of whether S' has a cache hit or a cache miss, we are done: S' has at most the same number of cache misses as S.

   So now suppose that S has a cache hit and S' has a cache miss upon the request for b_m. We'll show that there exists a request for at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ in which the request results in a cache miss for S and a cache hit for S', thereby compensating for what happens upon the request for block b_m. The proof is by contradiction. Assume that no request for blocks $b_{i+1}, \ldots, b_{m-1}$ results in a cache miss for S and a cache hit for S'.

   We start by observing that once the caches $C_{S,j}$ and $C_{S'j}$ are equal for some $j > i$, they remain equal thereafter. Observe also that if $b_m \in C_{S,m}$ and $b_m \notin C_{S',m}$, then $C_{S,m} \neq C_{S',m}$. Therefore, solution S cannot have evicted block z upon the requests for blocks $b_i, \ldots, b_{m-1}$, for if it had, then these two

cache configurations would be equal. The remaining possibility is that upon each of these requests, we had $C_{S,j} = D_j \cup \{z\}$, $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$, and solution S evicted some block $w \in D_j$. Moreover, since none of these requests resulted in a cache miss for S and a cache hit for S', the case of $b_j = y$ never occurred. That is, for every request of blocks $b_{i+1}, \ldots, b_{m-1}$, the requested block b_j was never the block $y \in C_{S',j} - C_{S,j}$. In these cases, after processing the request, we had $C_{S',j+1} = D_{j+1} \cup \{y\}$: the difference between the two caches did not change. Now, let's go back to the request for block b_i, where afterward, we had $C_{S',i+1} = D_{i+1} \cup \{x\}$. Because every succeeding request until requesting block b_m did not change the difference between the caches, we had $C_{S',j} = D_j \cup \{x\}$ for $j = i + 1, \ldots, m$.

By definition, block $z = b_m$ is requested after block x. That means at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ is block x. But for $j = i + 1, \ldots, m$, we have $x \in C_{S',j}$ and $x \notin C_{S,j}$, so that at least one of these requests had a cache hit for S' and a cache miss for S, a contradiction. We conclude that if solution S has a cache hit and solution S' has a cache miss upon the request for block b_m, then some earlier request had the opposite result, and so solution S' produces no more cache misses than solution S. Since S is assumed to be optimal, S' is optimal as well. ■

Along with the optimal-substructure property, Theorem 15.5 tells us that the furthest-in-future strategy yields the minimum number of cache misses.

Exercises

15.4-1

Write pseudocode for a cache manager that uses the furthest-in-future strategy. It should take as input a set C of blocks in the cache, the number of blocks k that the cache can hold, a sequence $b_1, b_2, \ldots, b_n$ of requested blocks, and the index i into the sequence for the block b_i being requested. For each request, it should print out whether a cache hit or cache miss occurs, and for each cache miss, it should also print out which block, if any, is evicted.

15.4-2

Real cache managers do not know the future requests, and so they often use the past to decide which block to evict. The ***least-recently-used***, or ***LRU***, strategy evicts the block that, of all blocks currently in the cache, was the least recently requested. (You can think of LRU as "furthest-in-past.") Give an example of a request sequence in which the LRU strategy is not optimal, by showing that it induces more cache misses than the furthest-in-future strategy does on the same request sequence.

15.4-3

Professor Croesus suggests that in the proof of Theorem 15.5, the last clause in property 1 can change to $C_{S',j} = D_j \cup \{x\}$ or, equivalently, require the block y given in property 1 to always be the block x evicted by solution S upon the request for block b_i. Show where the proof breaks down with this requirement.

15.4-4

This section has assumed that at most one block is placed into the cache whenever a block is requested. You can imagine, however, a strategy in which multiple blocks may enter the cache upon a single request. Show that for every solution that allows multiple blocks to enter the cache upon each request, there is another solution that brings in only one block upon each request and is at least as good.

Problems

15-1 Coin changing

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

a. 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 denominations that are powers of c: the denominations are $c^0, c^1, \ldots, c^k$ 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 using the smallest number of coins, assuming that one of the coins is a penny.

15-2 Scheduling to minimize average completion time

You are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete. 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 that 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 we have $C_2 = 5$, $C_1 = 8$, and the average completion time is $(5+8)/2 = 6.5$. If task a_1 runs first, however, then we have $C_1 = 3$, $C_2 = 8$, 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 nonpreemptively, that is, once task a_i starts, it must run continuously for p_i units of time until it is done. Prove that your algorithm minimizes the average completion time, and analyze 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*** b_i. Suppose also that tasks may be ***preempted***, 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 $b_i = 1$ 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. 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 analyze the running time of your algorithm.

Chapter notes

Much more material on greedy algorithms can be found in Lawler [276] and Papadimitriou and Steiglitz [353]. The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [131].

The proof of correctness of the greedy algorithm for the activity-selection problem is based on that of Gavril [179].

Huffman codes were invented in 1952 [233]. Lelewer and Hirschberg [294] surveys data-compression techniques known as of 1987.

The furthest-in-future strategy was proposed by Belady [41], who suggested it for virtual-memory systems. Alternative proofs that furthest-in-future is optimal appear in articles by Lee et al. [284] and Van Roy [443].