fashion, rather than the bottom-up manner typically used in dynamic programming. To solve the subproblem S_{ij}, we choose the activity a_m in S_{ij} with the earliest finish time and add to this solution the set of activities used in an optimal solution to the subproblem S_{mj}. Because we know that, having chosen a_m, we will certainly be using a solution to S_{mj} in our optimal solution to S_{ij}, we do not need to solve S_{mj} *before* solving S_{ij}. To solve S_{ij}, we can *first* choose a_m as the activity in S_{ij} with the earliest finish time and *then* solve S_{mj}.

Note also that there is a pattern to the subproblems that we solve. Our original problem is $S = S_{0,n+1}$. Suppose that we choose a_{m_1} as the activity in $S_{0,n+1}$ with the earliest finish time. (Since we have sorted activities by monotonically increasing finish times and $f_0 = 0$, we must have $m_1 = 1$.) Our next subproblem is $S_{m_1,n+1}$. Now suppose that we choose a_{m_2} as the activity in $S_{m_1,n+1}$ with the earliest finish time. (It is not necessarily the case that $m_2 = 2$.) Our next subproblem is $S_{m_2,n+1}$. Continuing, we see that each subproblem will be of the form $S_{m_i,n+1}$ for some activity number m_i. In other words, each subproblem consists of the last activities to finish, and the number of such activities varies from subproblem to subproblem.

There is also a pattern to the activities that we choose. Because we always choose the activity with the earliest finish time in $S_{m_i,n+1}$, the finish times of the activities chosen over all subproblems will be strictly increasing over time. Moreover, we can consider each activity just once overall, in monotonically increasing order of finish times.

The activity a_m that we choose when solving a subproblem is always the one with the earliest finish time that can be legally scheduled. The activity picked is thus a "greedy" choice in the sense that, intuitively, it leaves as much opportunity as possible for the remaining activities to be scheduled. That is, the greedy choice is the one that maximizes the amount of unscheduled time remaining.

A recursive greedy algorithm

Now that we have seen how to streamline our dynamic-programming solution, and how to treat it as a top-down method, we are ready to see an algorithm that works in a purely greedy, top-down fashion. We give a straightforward, recursive solution as the procedure RECURSIVE-ACTIVITY-SELECTOR. It takes the start and finish times of the activities, represented as arrays s and f, as well as the indices i and n that define the subproblem $S_{i,n+1}$ it is to solve. (The parameter n indexes the last actual activity a_n in the subproblem, and not the fictitious activity a_{n+1}, which is also in the subproblem.) It returns a maximum-size set of mutually compatible activities in $S_{i,n+1}$. We assume that the n input activities are 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. The initial call is RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR($s, f, i, n$)
1  $m \leftarrow i + 1$
2  while $m \le n$ and $s_m < f_i$        ▷ Find the first activity in $S_{i,n+1}$.
3      do $m \leftarrow m + 1$
4  if $m \le n$
5     then return $\{a_m\} \cup$ 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, i, n), the **while** loop of lines 2–3 looks for the first activity in $S_{i,n+1}$. The loop examines $a_{i+1}, a_{i+2}, \ldots, a_n$, until it finds the first activity a_m that is compatible with a_i; such an activity has $s_m \ge f_i$. If the loop terminates because it finds such an activity, the procedure returns in line 5 the union of $\{a_m\}$ and the maximum-size subset of $S_{m,n+1}$ 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 without finding one that is compatible with a_i. In this case, $S_{i,n+1} = \emptyset$, and so the procedure returns $\emptyset$ in line 6.

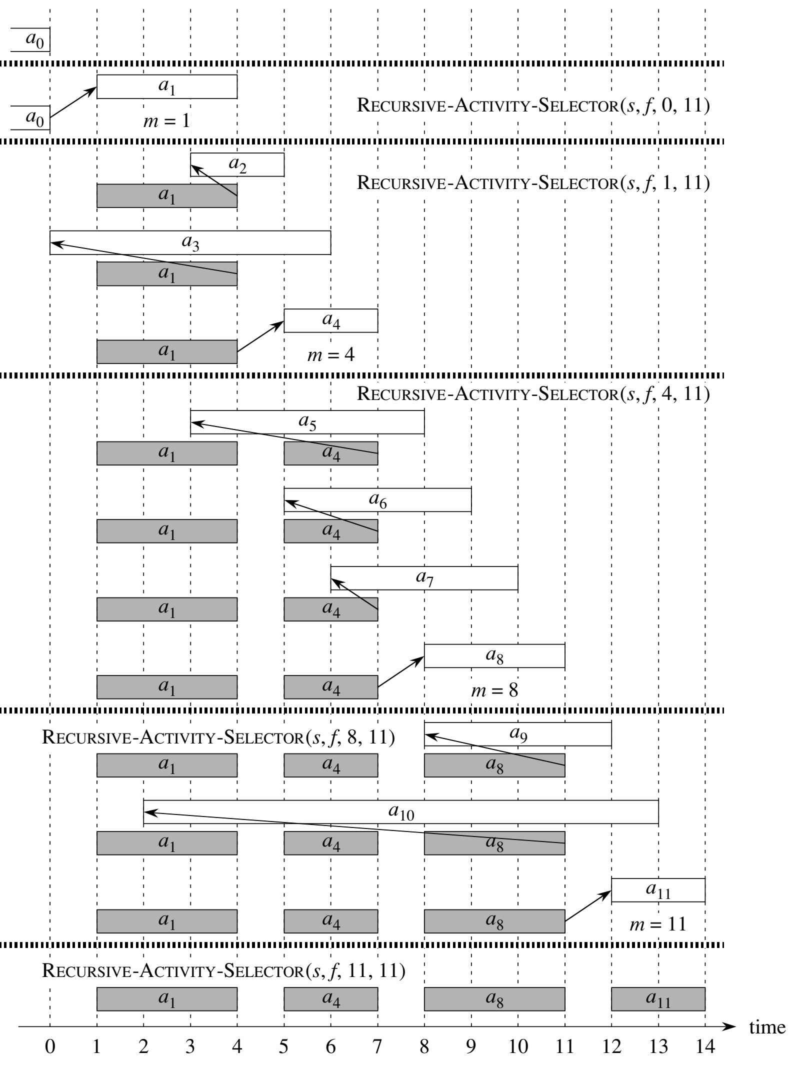
Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ is $\Theta(n)$, which we can see as follows. Over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular,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

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 $S_{i,n+1}$, i.e., subproblems that consist of the last activities to finish.

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.

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	–



a_0
a_1
$m = 1$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
a_2
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
a_3
a_4
$m = 4$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
a_5
a_6
a_7
a_8
$m = 8$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
a_9
a_{10}
a_{11}
$m = 11$
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

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 in the initial call, RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11), activity a_1 is selected. 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 $\{a_1, a_4, a_8, a_{11}\}$.

```
GREEDY-ACTIVITY-SELECTOR($s$, $f$)
1  $n \leftarrow length[s]$
2  $A \leftarrow \{a_1\}$
3  $i \leftarrow 1$
4  for $m \leftarrow 2$ to $n$
5       do if $s_m \geq f_i$
6             then $A \leftarrow A \cup \{a_m\}$
7                  $i \leftarrow m$
8  return $A$
```

The procedure works as follows. The variable i indexes the most recent addition to A, corresponding to the activity a_i in the recursive version. Since the activities are considered in order of monotonically increasing finish time, f_i is always the maximum finish time of any activity in A. That is,

$$f_i = \max\{f_k : a_k \in A\} \ . \tag{16.4}$$

Lines 2–3 select activity a_1, initialize A to contain just this activity, and initialize i to index this activity. The **for** loop of lines 4–7 finds the earliest activity to finish in $S_{i,n+1}$. 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 to finish in $S_{i,n+1}$. To see if activity a_m is compatible with every activity currently in A, it suffices by equation (16.4) to check (line 5) that its start time s_m is not earlier than the finish time f_i 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 i 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 the recurrence (16.3). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset A of 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. De-