brute-force search: although it's systematically working through the exponentially large set of possible solutions to the problem, it does this without ever examining them all explicitly. It is because of this careful balancing act that dynamic programming can be a tricky technique to get used to; it typically takes a reasonable amount of practice before one is fully comfortable with it.

With this in mind, we now turn to a first example of dynamic programming: the Weighted Interval Scheduling Problem that we defined back in Section 1.2. We are going to develop a dynamic programming algorithm for this problem in two stages: first as a recursive procedure that closely resembles brute-force search; and then, by reinterpreting this procedure, as an iterative algorithm that works by building up solutions to larger and larger subproblems.

6.1 Weighted Interval Scheduling: A Recursive Procedure

We have seen that a particular greedy algorithm produces an optimal solution to the Interval Scheduling Problem, where the goal is to accept as large a set of nonoverlapping intervals as possible. The Weighted Interval Scheduling Problem is a strictly more general version, in which each interval has a certain *value* (or *weight*), and we want to accept a set of maximum value.

Designing a Recursive Algorithm

Since the original Interval Scheduling Problem is simply the special case in which all values are equal to 1, we know already that most greedy algorithms will not solve this problem optimally. But even the algorithm that worked before (repeatedly choosing the interval that ends earliest) is no longer optimal in this more general setting, as the simple example in Figure 6.1 shows.

Indeed, no natural greedy algorithm is known for this problem, which is what motivates our switch to dynamic programming. As discussed above, we will begin our introduction to dynamic programming with a recursive type of algorithm for this problem, and then in the next section we'll move to a more iterative method that is closer to the style we use in the rest of this chapter.

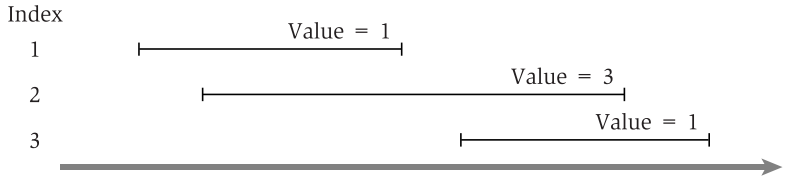

Figure 6.1 A simple instance of weighted interval scheduling.

We use the notation from our discussion of Interval Scheduling in Section 1.2. We have n requests labeled $1, \ldots, n$, with each request i specifying a start time s_i and a finish time f_i. Each interval i now also has a *value*, or *weight* v_i. Two intervals are *compatible* if they do not overlap. The goal of our current problem is to select a subset $S \subseteq \{1, \ldots, n\}$ of mutually compatible intervals, so as to maximize the sum of the values of the selected intervals, $\sum_{i \in S} v_i$.

Let's suppose that the requests are sorted in order of nondecreasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$. We'll say a request i comes *before* a request j if $i < j$. This will be the natural left-to-right order in which we'll consider intervals. To help in talking about this order, we define $p(j)$, for an interval j, to be the largest index $i < j$ such that intervals i and j are disjoint. In other words, i is the leftmost interval that ends before j begins. We define $p(j) = 0$ if no request $i < j$ is disjoint from j. An example of the definition of $p(j)$ is shown in Figure 6.2.

Now, given an instance of the Weighted Interval Scheduling Problem, let's consider an optimal solution $\mathcal{O}$, ignoring for now that we have no idea what it is. Here's something completely obvious that we can say about $\mathcal{O}$: either interval n (the last one) belongs to $\mathcal{O}$, or it doesn't. Suppose we explore both sides of this dichotomy a little further. If $n \in \mathcal{O}$, then clearly no interval indexed strictly between $p(n)$ and n can belong to $\mathcal{O}$, because by the definition of $p(n)$, we know that intervals $p(n) + 1, p(n) + 2, \ldots, n - 1$ all overlap interval n. Moreover, if $n \in \mathcal{O}$, then $\mathcal{O}$ must include an *optimal* solution to the problem consisting of requests $\{1, \ldots, p(n)\}$—for if it didn't, we could replace $\mathcal{O}$'s choice of requests from $\{1, \ldots, p(n)\}$ with a better one, with no danger of overlapping request n.

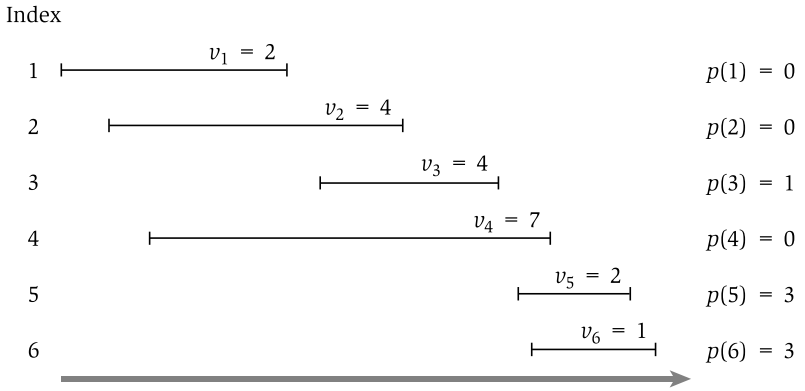

Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

On the other hand, if $n \notin \mathcal{O}$, then $\mathcal{O}$ is simply equal to the optimal solution to the problem consisting of requests $\{1, \ldots, n-1\}$. This is by completely analogous reasoning: we're assuming that $\mathcal{O}$ does not include request n; so if it does not choose the optimal set of requests from $\{1, \ldots, n-1\}$, we could replace it with a better one.

All this suggests that finding the optimal solution on intervals $\{1, 2, \ldots, n\}$ involves looking at the optimal solutions of smaller problems of the form $\{1, 2, \ldots, j\}$. Thus, for any value of j between 1 and n, let $\mathcal{O}_j$ denote the optimal solution to the problem consisting of requests $\{1, \ldots, j\}$, and let $\text{OPT}(j)$ denote the value of this solution. (We define $\text{OPT}(0) = 0$, based on the convention that this is the optimum over an empty set of intervals.) The optimal solution we're seeking is precisely $\mathcal{O}_n$, with value $\text{OPT}(n)$. For the optimal solution $\mathcal{O}_j$ on $\{1, 2, \ldots, j\}$, our reasoning above (generalizing from the case in which $j = n$) says that either $j \in \mathcal{O}_j$, in which case $\text{OPT}(j) = v_j + \text{OPT}(p(j))$, or $j \notin \mathcal{O}_j$, in which case $\text{OPT}(j) = \text{OPT}(j-1)$. Since these are precisely the two possible choices ($j \in \mathcal{O}_j$ or $j \notin \mathcal{O}_j$), we can further say that

(6.1) $$\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)).$$

And how do we decide whether n belongs to the optimal solution $\mathcal{O}_j$? This too is easy: it belongs to the optimal solution if and only if the first of the options above is at least as good as the second; in other words,

(6.2) *Request j belongs to an optimal solution on the set $\{1, 2, \ldots, j\}$ if and only if*

$$v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1).$$

These facts form the first crucial component on which a dynamic programming solution is based: a recurrence equation that expresses the optimal solution (or its value) in terms of the optimal solutions to smaller subproblems.

Despite the simple reasoning that led to this point, (6.1) is already a significant development. It directly gives us a recursive algorithm to compute $\text{OPT}(n)$, assuming that we have already sorted the requests by finishing time and computed the values of $p(j)$ for each j.

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j – 1))
  Endif
```

The correctness of the algorithm follows directly by induction on j:

(6.3) `Compute-Opt`(j) *correctly computes* $\text{OPT}(j)$ *for each* $j = 1, 2, \ldots, n$.

Proof. By definition $\text{OPT}(0) = 0$. Now, take some $j > 0$, and suppose by way of induction that `Compute-Opt`(i) correctly computes $\text{OPT}(i)$ for all $i < j$. By the induction hypothesis, we know that `Compute-Opt`$(p(j)) = \text{OPT}(p(j))$ and `Compute-Opt`$(j-1) = \text{OPT}(j-1)$; and hence from (6.1) it follows that

$$\begin{aligned}\text{OPT}(j) &= \max(v_j + \texttt{Compute-Opt}(p(j)), \texttt{Compute-Opt}(j-1)) \\ &= \texttt{Compute-Opt}(j). \quad \blacksquare\end{aligned}$$

Unfortunately, if we really implemented the algorithm `Compute-Opt` as just written, it would take exponential time to run in the worst case. For example, see Figure 6.3 for the tree of calls issued for the instance of Figure 6.2: the tree widens very quickly due to the recursive branching. To take a more extreme example, on a nicely layered instance like the one in Figure 6.4, where $p(j) = j - 2$ for each $j = 2, 3, 4, \ldots, n$, we see that `Compute-Opt`(j) generates separate recursive calls on problems of sizes $j-1$ and $j-2$. In other words, the total number of calls made to `Compute-Opt` on this instance will grow

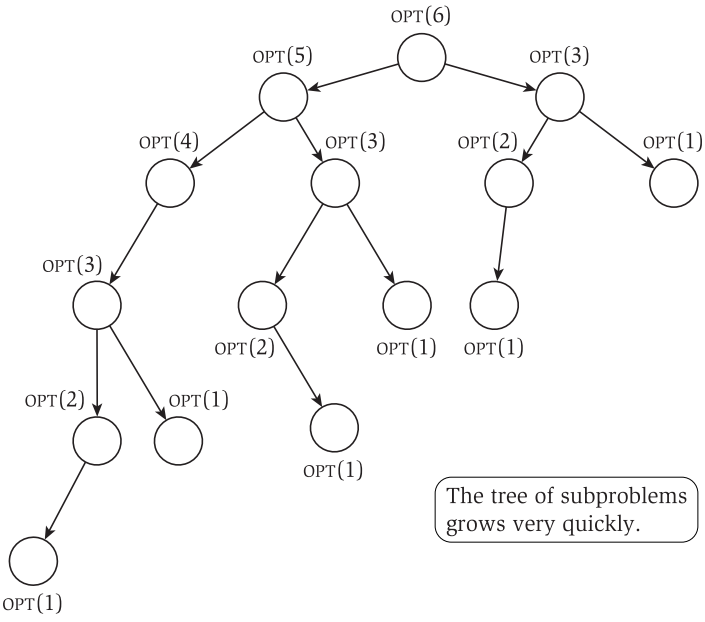

Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

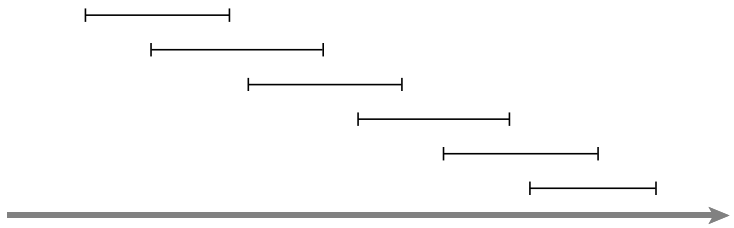

Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

like the Fibonacci numbers, which increase exponentially. Thus we have not achieved a polynomial-time solution.

Memoizing the Recursion

In fact, though, we're not so far from having a polynomial-time algorithm. A fundamental observation, which forms the second crucial component of a dynamic programming solution, is that our recursive algorithm Compute-Opt is really only solving $n+1$ different subproblems: Compute-Opt(0), Compute-Opt(1), ..., Compute-Opt(n). The fact that it runs in exponential time as written is simply due to the spectacular redundancy in the number of times it issues each of these calls.

How could we eliminate all this redundancy? We could store the value of Compute-Opt in a globally accessible place the first time we compute it and then simply use this precomputed value in place of all future recursive calls. This technique of saving values that have already been computed is referred to as *memoization*.

We implement the above strategy in the more "intelligent" procedure M-Compute-Opt. This procedure will make use of an array $M[0 \ldots n]$; $M[j]$ will start with the value "empty," but will hold the value of Compute-Opt(j) as soon as it is first determined. To determine OPT(n), we invoke M-Compute-Opt(n).

```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
```

```
  Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
   Return M[j]
 Endif
```

Analyzing the Memoized Version

Clearly, this looks very similar to our previous implementation of the algorithm; however, memoization has brought the running time way down.

(6.4) *The running time of* `M-Compute-Opt`(n) *is* $O(n)$ *(assuming the input intervals are sorted by their finish times).*

Proof. The time spent in a single call to `M-Compute-Opt` is $O(1)$, excluding the time spent in recursive calls it generates. So the running time is bounded by a constant times the number of calls ever issued to `M-Compute-Opt`. Since the implementation itself gives no explicit upper bound on this number of calls, we try to find a bound by looking for a good measure of "progress."

The most useful progress measure here is the number of entries in M that are not "empty." Initially this number is 0; but each time the procedure invokes the recurrence, issuing two recursive calls to `M-Compute-Opt`, it fills in a new entry, and hence increases the number of filled-in entries by 1. Since M has only $n + 1$ entries, it follows that there can be at most $O(n)$ calls to `M-Compute-Opt`, and hence the running time of `M-Compute-Opt`(n) is $O(n)$, as desired. ■

Computing a Solution in Addition to Its Value

So far we have simply computed the *value* of an optimal solution; presumably we want a full optimal set of intervals as well. It would be easy to extend `M-Compute-Opt` so as to keep track of an optimal solution in addition to its value: we could maintain an additional array S so that $S[i]$ contains an optimal set of intervals among $\{1, 2, \ldots, i\}$. Naively enhancing the code to maintain the solutions in the array S, however, would blow up the running time by an additional factor of $O(n)$: while a position in the M array can be updated in $O(1)$ time, writing down a set in the S array takes $O(n)$ time. We can avoid this $O(n)$ blow-up by not explicitly maintaining S, but rather by recovering the optimal solution from values saved in the array M after the optimum value has been computed.

We know from (6.2) that j belongs to an optimal solution for the set of intervals $\{1, \ldots, j\}$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$. Using this observation, we get the following simple procedure, which "traces back" through the array M to find the set of intervals in an optimal solution.

```
Find-Solution(j)
  If j = 0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j − 1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j − 1)
    Endif
  Endif
```

Since `Find-Solution` calls itself recursively only on strictly smaller values, it makes a total of $O(n)$ recursive calls; and since it spends constant time per call, we have

(6.5) *Given the array M of the optimal values of the sub-problems,* `Find-Solution` *returns an optimal solution in $O(n)$ time.*

6.2 Principles of Dynamic Programming: Memoization or Iteration over Subproblems

We now use the algorithm for the Weighted Interval Scheduling Problem developed in the previous section to summarize the basic principles of dynamic programming, and also to offer a different perspective that will be fundamental to the rest of the chapter: iterating over subproblems, rather than computing solutions recursively.

In the previous section, we developed a polynomial-time solution to the Weighted Interval Scheduling Problem by first designing an exponential-time recursive algorithm and then converting it (by memoization) to an efficient recursive algorithm that consulted a global array M of optimal solutions to subproblems. To really understand what is going on here, however, it helps to formulate an essentially equivalent version of the algorithm. It is this new formulation that most explicitly captures the essence of the dynamic programming technique, and it will serve as a general template for the algorithms we develop in later sections.

Designing the Algorithm

The key to the efficient algorithm is really the array M. It encodes the notion that we are using the value of optimal solutions to the subproblems on intervals $\{1, 2, \ldots, j\}$ for each j, and it uses (6.1) to define the value of $M[j]$ based on

values that come earlier in the array. Once we have the array M, the problem is solved: $M[n]$ contains the value of the optimal solution on the full instance, and `Find-Solution` can be used to trace back through M efficiently and return an optimal solution itself.

The point to realize, then, is that we can directly compute the entries in M by an iterative algorithm, rather than using memoized recursion. We just start with $M[0] = 0$ and keep incrementing j; each time we need to determine a value $M[j]$, the answer is provided by (6.1). The algorithm looks as follows.

```
Iterative-Compute-Opt
  M[0] = 0
  For j = 1, 2, ..., n
    M[j] = max(v_j + M[p(j)], M[j − 1])
  Endfor
```

Analyzing the Algorithm

By exact analogy with the proof of (6.3), we can prove by induction on j that this algorithm writes $\text{OPT}(j)$ in array entry $M[j]$; (6.1) provides the induction step. Also, as before, we can pass the filled-in array M to `Find-Solution` to get an optimal solution in addition to the value. Finally, the running time of `Iterative-Compute-Opt` is clearly $O(n)$, since it explicitly runs for n iterations and spends constant time in each.

An example of the execution of `Iterative-Compute-Opt` is depicted in Figure 6.5. In each iteration, the algorithm fills in one additional entry of the array M, by comparing the value of $v_j + M[p(j)]$ to the value of $M[j-1]$.

A Basic Outline of Dynamic Programming

This, then, provides a second efficient algorithm to solve the Weighted Interval Scheduling Problem. The two approaches clearly have a great deal of conceptual overlap, since they both grow from the insight contained in the recurrence (6.1). For the remainder of the chapter, we will develop dynamic programming algorithms using the second type of approach—iterative building up of subproblems—because the algorithms are often simpler to express this way. But in each case that we consider, there is an equivalent way to formulate the algorithm as a memoized recursion.

Most crucially, the bulk of our discussion about the particular problem of selecting intervals can be cast more generally as a rough template for designing dynamic programming algorithms. To set about developing an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problem that satisfies a few basic properties.

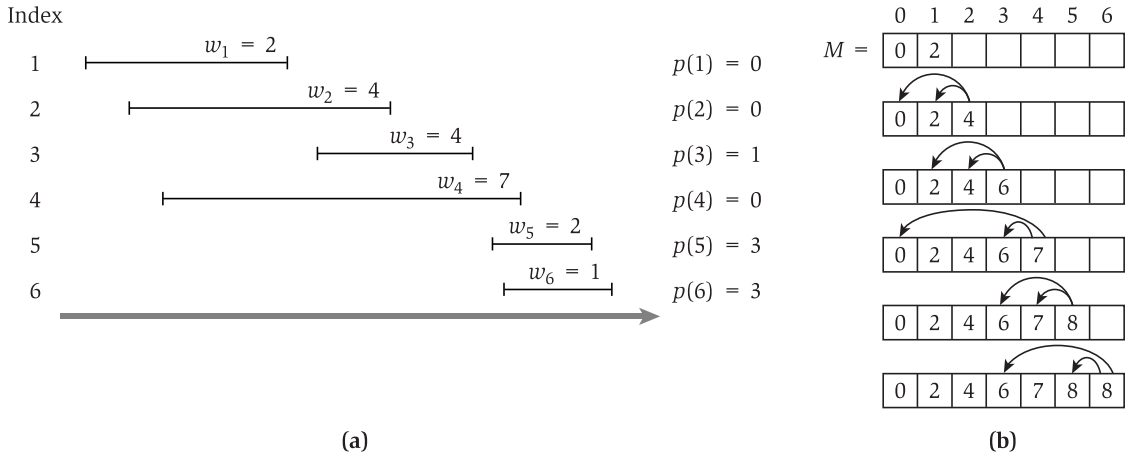

Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

(i) There are only a polynomial number of subproblems.

(ii) The solution to the original problem can be easily computed from the solutions to the subproblems. (For example, the original problem may actually *be* one of the subproblems.)

(iii) There is a natural ordering on subproblems from "smallest" to "largest," together with an easy-to-compute recurrence (as in (6.1) and (6.2)) that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.

Naturally, these are informal guidelines. In particular, the notion of "smaller" in part (iii) will depend on the type of recurrence one has.

We will see that it is sometimes easier to start the process of designing such an algorithm by formulating a set of subproblems that looks natural, and then figuring out a recurrence that links them together; but often (as happened in the case of weighted interval scheduling), it can be useful to first define a recurrence by reasoning about the structure of an optimal solution, and then determine which subproblems will be necessary to unwind the recurrence. This chicken-and-egg relationship between subproblems and recurrences is a subtle issue underlying dynamic programming. It's never clear that a collection of subproblems will be useful until one finds a recurrence linking them together; but it can be difficult to think about recurrences in the absence of the "smaller" subproblems that they build on. In subsequent sections, we will develop further practice in managing this design trade-off.