Chapter 6

Dynamic Programming

We began our study of algorithmic techniques with greedy algorithms, which in some sense form the most natural approach to algorithm design. Faced with a new computational problem, we've seen that it's not hard to propose multiple possible greedy algorithms; the challenge is then to determine whether any of these algorithms provides a correct solution to the problem in all cases.

The problems we saw in Chapter 4 were all unified by the fact that, in the end, there really was a greedy algorithm that worked. Unfortunately, this is far from being true in general; for most of the problems that one encounters, the real difficulty is not in determining which of several greedy strategies is the right one, but in the fact that there is *no* natural greedy algorithm that works. For such problems, it is important to have other approaches at hand. Divide and conquer can sometimes serve as an alternative approach, but the versions of divide and conquer that we saw in the previous chapter are often not strong enough to reduce exponential brute-force search down to polynomial time. Rather, as we noted in Chapter 5, the applications there tended to reduce a running time that was unnecessarily large, but already polynomial, down to a faster running time.

We now turn to a more powerful and subtle design technique, *dynamic programming*. It will be easier to say exactly what characterizes dynamic programming after we've seen it in action, but the basic idea is drawn from the intuition behind divide and conquer and is essentially the opposite of the greedy strategy: one implicitly explores the space of all possible solutions, by carefully decomposing things into a series of *subproblems*, and then building up correct solutions to larger and larger subproblems. In a way, we can thus view dynamic programming as operating dangerously close to the edge of

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.

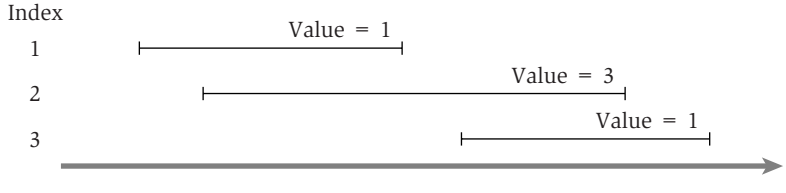

Figure 6.1 A simple instance of weighted interval scheduling.

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.

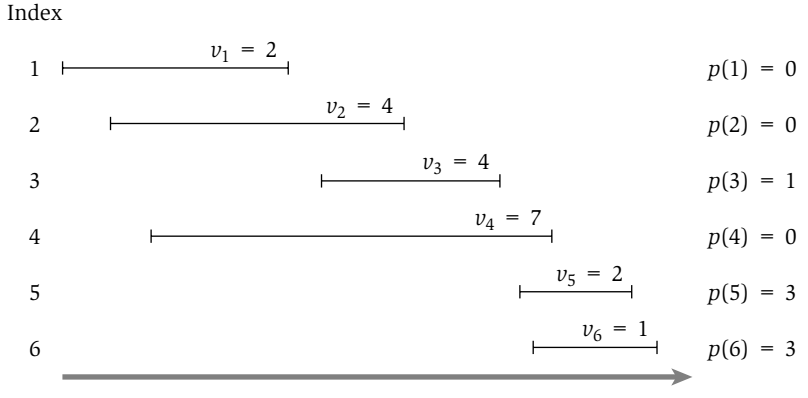

Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

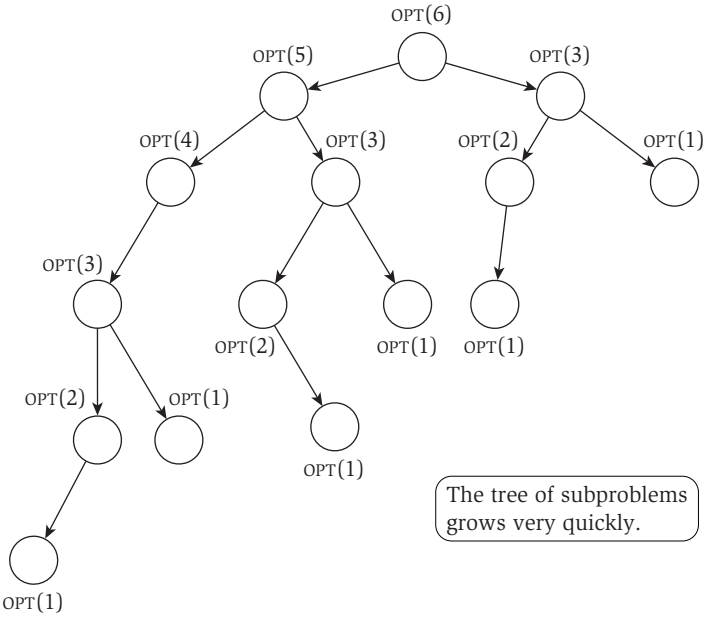

Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

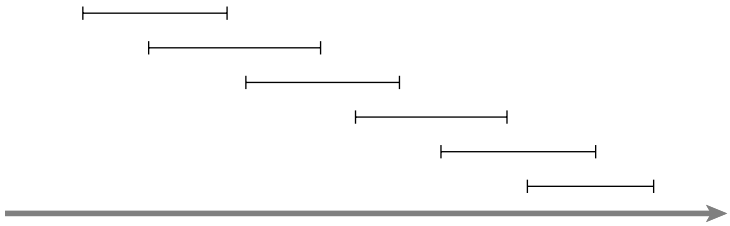

Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

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.

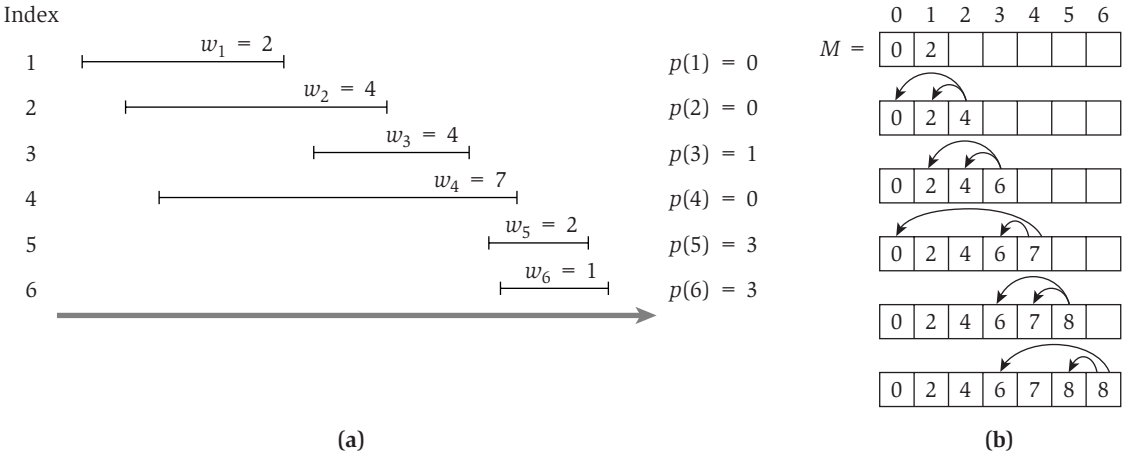

Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

(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.

6.3 Segmented Least Squares: Multi-way Choices

We now discuss a different type of problem, which illustrates a slightly more complicated style of dynamic programming. In the previous section, we developed a recurrence based on a fundamentally *binary* choice: either the interval n belonged to an optimal solution or it didn't. In the problem we consider here, the recurrence will involve what might be called "multi-way choices": at each step, we have a polynomial number of possibilities to consider for the structure of the optimal solution. As we'll see, the dynamic programming approach adapts to this more general situation very naturally.

As a separate issue, the problem developed in this section is also a nice illustration of how a clean algorithmic definition can formalize a notion that initially seems too fuzzy and nonintuitive to work with mathematically.

The Problem

Often when looking at scientific or statistical data, plotted on a two-dimensional set of axes, one tries to pass a "line of best fit" through the data, as in Figure 6.6.

This is a foundational problem in statistics and numerical analysis, formulated as follows. Suppose our data consists of a set P of n points in the plane, denoted $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$; and suppose $x_1 < x_2 < \cdots < x_n$. Given a line L defined by the equation $y = ax + b$, we say that the *error* of L with respect to P is the sum of its squared "distances" to the points in P:

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

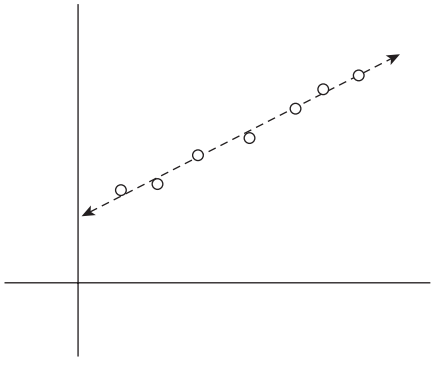

Figure 6.6 A "line of best fit."

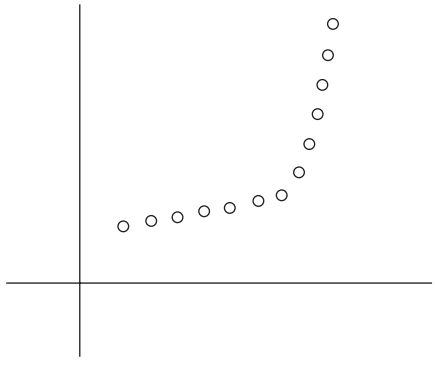

Figure 6.7 A set of points that lie approximately on two lines.

A natural goal is then to find the line with minimum error; this turns out to have a nice closed-form solution that can be easily derived using calculus. Skipping the derivation here, we simply state the result: The line of minimum error is $y = ax + b$, where

$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \quad \text{and} \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}.$$

Now, here's a kind of issue that these formulas weren't designed to cover. Often we have data that looks something like the picture in Figure 6.7. In this case, we'd like to make a statement like: "The points lie roughly on a sequence of two lines." How could we formalize this concept?

Essentially, any single line through the points in the figure would have a terrible error; but if we use two lines, we could achieve quite a small error. So we could try formulating a new problem as follows: Rather than seek a single line of best fit, we are allowed to pass an arbitrary *set* of lines through the points, and we seek a set of lines that minimizes the error. But this fails as a good problem formulation, because it has a trivial solution: if we're allowed to fit the points with an arbitrarily large set of lines, we could fit the points perfectly by having a different line pass through each pair of consecutive points in P.

At the other extreme, we could try "hard-coding" the number two into the problem; we could seek the best fit using at most two lines. But this too misses a crucial feature of our intuition: We didn't start out with a preconceived idea that the points lay approximately on two lines; we concluded that from looking at the picture. For example, most people would say that the points in Figure 6.8 lie approximately on three lines.

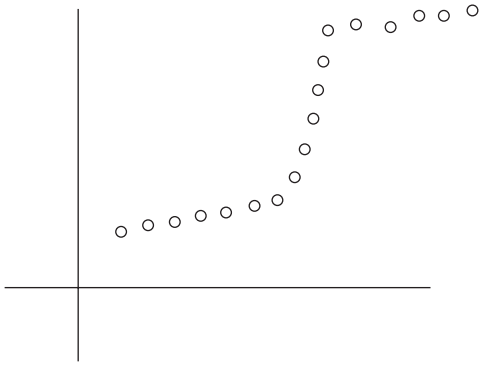

Figure 6.8 A set of points that lie approximately on three lines.

Thus, intuitively, we need a problem formulation that requires us to fit the points well, using as few lines as possible. We now formulate a problem—the *Segmented Least Squares Problem*—that captures these issues quite cleanly. The problem is a fundamental instance of an issue in data mining and statistics known as *change detection*: Given a sequence of data points, we want to identify a few points in the sequence at which a discrete *change* occurs (in this case, a change from one linear approximation to another).

Formulating the Problem As in the discussion above, we are given a set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, with $x_1 < x_2 < \cdots < x_n$. We will use p_i to denote the point (x_i, y_i). We must first partition P into some number of segments. Each *segment* is a subset of P that represents a contiguous set of x-coordinates; that is, it is a subset of the form $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for some indices $i \leq j$. Then, for each segment S in our partition of P, we compute the line minimizing the error with respect to the points in S, according to the formulas above.

The *penalty* of a partition is defined to be a sum of the following terms.

(i) The number of segments into which we partition P, times a fixed, given multiplier $C > 0$.

(ii) For each segment, the error value of the optimal line through that segment.

Our goal in the Segmented Least Squares Problem is to find a partition of minimum penalty. This minimization captures the trade-offs we discussed earlier. We are allowed to consider partitions into any number of segments; as we increase the number of segments, we reduce the penalty terms in part (ii) of the definition, but we increase the term in part (i). (The multiplier C is provided

with the input, and by tuning C, we can penalize the use of additional lines to a greater or lesser extent.)

There are exponentially many possible partitions of P, and initially it is not clear that we should be able to find the optimal one efficiently. We now show how to use dynamic programming to find a partition of minimum penalty in time polynomial in n.

Designing the Algorithm

To begin with, we should recall the ingredients we need for a dynamic programming algorithm, as outlined at the end of Section 6.2.We want a polynomial number of subproblems, the solutions of which should yield a solution to the original problem; and we should be able to build up solutions to these subproblems using a recurrence. As with the Weighted Interval Scheduling Problem, it helps to think about some simple properties of the optimal solution. Note, however, that there is not really a direct analogy to weighted interval scheduling: there we were looking for a *subset* of n objects, whereas here we are seeking to *partition* n objects.

For segmented least squares, the following observation is very useful: The last point p_n belongs to a single segment in the optimal partition, and that segment begins at some earlier point p_i. This is the type of observation that can suggest the right set of subproblems: if we knew the identity of the *last* segment $p_i, \ldots, p_n$ (see Figure 6.9), then we could remove those points from consideration and recursively solve the problem on the remaining points $p_1, \ldots, p_{i-1}$.

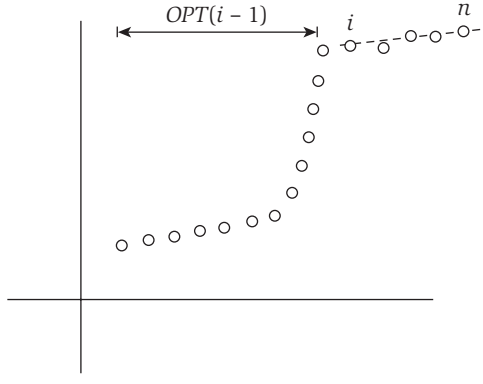

Figure 6.9 A possible solution: a single line segment fits points $p_i, p_{i+1}, \ldots, p_n$, and then an optimal solution is found for the remaining points $p_1, p_2, \ldots, p_{i-1}$.

Suppose we let $\text{OPT}(i)$ denote the optimum solution for the points $p_1, \ldots, p_i$, and we let $e_{i,j}$ denote the minimum error of any line with respect to $p_i, p_{i+1}, \ldots, p_j$. (We will write $\text{OPT}(0) = 0$ as a boundary case.) Then our observation above says the following.

(6.6) *If the last segment of the optimal partition is $p_i, \ldots, p_n$, then the value of the optimal solution is $\text{OPT}(n) = e_{i,n} + C + \text{OPT}(i-1)$.*

Using the same observation for the subproblem consisting of the points $p_1, \ldots, p_j$, we see that to get $\text{OPT}(j)$ we should find the best way to produce a final segment $p_i, \ldots, p_j$—paying the error plus an additive C for this segment—together with an optimal solution $\text{OPT}(i-1)$ for the remaining points. In other words, we have justified the following recurrence.

(6.7) *For the subproblem on the points $p_1, \ldots, p_j$,*

$$\text{OPT}(j) = \min_{1 \le i \le j} (e_{i,j} + C + \text{OPT}(i-1)),$$

and the segment $p_i, \ldots, p_j$ is used in an optimum solution for the subproblem if and only if the minimum is obtained using index i.

The hard part in designing the algorithm is now behind us. From here, we simply build up the solutions $\text{OPT}(i)$ in order of increasing i.

```
Segmented-Least-Squares(n)
  Array M[0 ... n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

By analogy with the arguments for weighted interval scheduling, the correctness of this algorithm can be proved directly by induction, with (6.7) providing the induction step.

And as in our algorithm for weighted interval scheduling, we can trace back through the array M to compute an optimum partition.

```
Find-Segments(j)
  If j = 0 then
    Output nothing
  Else
    Find an i that minimizes e_{i,j} + C + M[i − 1]
    Output the segment {p_i, ..., p_j} and the result of
           Find-Segments(i − 1)
  Endif
```

Analyzing the Algorithm

Finally, we consider the running time of `Segmented-Least-Squares`. First we need to compute the values of all the least-squares errors $e_{i,j}$. To perform a simple accounting of the running time for this, we note that there are $O(n^2)$ pairs (i, j) for which this computation is needed; and for each pair (i, j), we can use the formula given at the beginning of this section to compute $e_{i,j}$ in $O(n)$ time. Thus the total running time to compute all $e_{i,j}$ values is $O(n^3)$.

Following this, the algorithm has n iterations, for values $j = 1, \ldots, n$. For each value of j, we have to determine the minimum in the recurrence (6.7) to fill in the array entry $M[j]$; this takes time $O(n)$ for each j, for a total of $O(n^2)$. Thus the running time is $O(n^2)$ once all the $e_{i,j}$ values have been determined.[1]

6.4 Subset Sums and Knapsacks: Adding a Variable

We're seeing more and more that issues in scheduling provide a rich source of practically motivated algorithmic problems. So far we've considered problems in which requests are specified by a given interval of time on a resource, as well as problems in which requests have a duration and a deadline but do not mandate a particular interval during which they need to be done.

In this section, we consider a version of the second type of problem, with durations and deadlines, which is difficult to solve directly using the techniques we've seen so far. We will use dynamic programming to solve the problem, but with a twist: the "obvious" set of subproblems will turn out not to be enough, and so we end up creating a richer collection of subproblems. As

[1] In this analysis, the running time is dominated by the $O(n^3)$ needed to compute all $e_{i,j}$ values. But, in fact, it is possible to compute all these values in $O(n^2)$ time, which brings the running time of the full algorithm down to $O(n^2)$. The idea, whose details we will leave as an exercise for the reader, is to first compute $e_{i,j}$ for all pairs (i, j) where $j - i = 1$, then for all pairs where $j - i = 2$, then $j - i = 3$, and so forth. This way, when we get to a particular $e_{i,j}$ value, we can use the ingredients of the calculation for $e_{i,j-1}$ to determine $e_{i,j}$ in constant time.

we will see, this is done by adding a new variable to the recurrence underlying the dynamic program.

The Problem

In the scheduling problem we consider here, we have a single machine that can process jobs, and we have a set of requests $\{1, 2, \ldots, n\}$. We are only able to use this resource for the period between time 0 and time W, for some number W. Each request corresponds to a job that requires time w_i to process. If our goal is to process jobs so as to keep the machine as busy as possible up to the "cut-off" W, which jobs should we choose?

More formally, we are given n items $\{1, \ldots, n\}$, and each has a given nonnegative weight w_i (for $i = 1, \ldots, n$). We are also given a bound W. We would like to select a subset S of the items so that $\sum_{i \in S} w_i \leq W$ and, subject to this restriction, $\sum_{i \in S} w_i$ is as large as possible. We will call this the *Subset Sum Problem*.

This problem is a natural special case of a more general problem called the *Knapsack Problem*, where each request i has both a *value* v_i and a *weight* w_i. The goal in this more general problem is to select a subset of maximum total value, subject to the restriction that its total weight not exceed W. Knapsack problems often show up as subproblems in other, more complex problems. The name *knapsack* refers to the problem of filling a knapsack of capacity W as full as possible (or packing in as much value as possible), using a subset of the items $\{1, \ldots, n\}$. We will use *weight* or *time* when referring to the quantities w_i and W.

Since this resembles other scheduling problems we've seen before, it's natural to ask whether a greedy algorithm can find the optimal solution. It appears that the answer is no—at least, no efficient greedy rule is known that always constructs an optimal solution. One natural greedy approach to try would be to sort the items by decreasing weight—or at least to do this for all items of weight at most W—and then start selecting items in this order as long as the total weight remains below W. But if W is a multiple of 2, and we have three items with weights $\{W/2 + 1, W/2, W/2\}$, then we see that this greedy algorithm will not produce the optimal solution. Alternately, we could sort by *increasing* weight and then do the same thing; but this fails on inputs like $\{1, W/2, W/2\}$.

The goal of this section is to show how to use dynamic programming to solve this problem. Recall the main principles of dynamic programming: We have to come up with a small number of subproblems so that each subproblem can be solved easily from "smaller" subproblems, and the solution to the original problem can be obtained easily once we know the solutions to all

the subproblems. The tricky issue here lies in figuring out a good set of subproblems.

Designing the Algorithm

A False Start One general strategy, which worked for us in the case of Weighted Interval Scheduling, is to consider subproblems involving only the first i requests. We start by trying this strategy here. We use the notation $\text{OPT}(i)$, analogously to the notation used before, to denote the best possible solution using a subset of the requests $\{1, \ldots, i\}$. The key to our method for the Weighted Interval Scheduling Problem was to concentrate on an optimal solution $\mathcal{O}$ to our problem and consider two cases, depending on whether or not the last request n is accepted or rejected by this optimum solution. Just as in that case, we have the first part, which follows immediately from the definition of $\text{OPT}(i)$.

- If $n \notin \mathcal{O}$, then $\text{OPT}(n) = \text{OPT}(n-1)$.

Next we have to consider the case in which $n \in \mathcal{O}$. What we'd like here is a simple recursion, which tells us the best possible value we can get for solutions that contain the last request n. For Weighted Interval Scheduling this was easy, as we could simply delete each request that conflicted with request n. In the current problem, this is not so simple. Accepting request n does not immediately imply that we have to reject any other request. Instead, it means that for the subset of requests $S \subseteq \{1, \ldots, n-1\}$ that we will accept, we have less available weight left: a weight of w_n is used on the accepted request n, and we only have $W - w_n$ weight left for the set S of remaining requests that we accept. See Figure 6.10.

A Better Solution This suggests that we need more subproblems: To find out the value for $\text{OPT}(n)$ we not only need the value of $\text{OPT}(n-1)$, but we also need to know the best solution we can get using a subset of the first $n-1$ items and total allowed weight $W - w_n$. We are therefore going to use many more subproblems: one for each initial set $\{1, \ldots, i\}$ of the items, and each possible

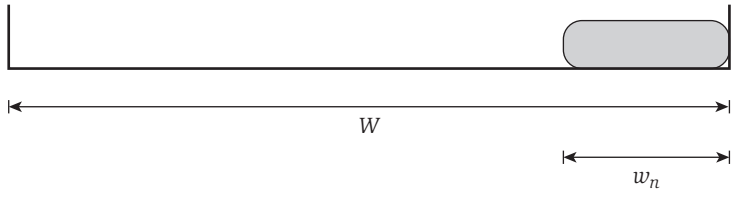

Figure 6.10 After item n is included in the solution, a weight of w_n is used up and there is $W - w_n$ available weight left.

value for the remaining available weight w. Assume that W is an integer, and all requests $i = 1, \ldots, n$ have integer weights w_i. We will have a subproblem for each $i = 0, 1, \ldots, n$ and each integer $0 \leq w \leq W$. We will use $\text{OPT}(i, w)$ to denote the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ with maximum allowed weight w, that is,

$$\text{OPT}(i, w) = \max_S \sum_{j \in S} w_j,$$

where the maximum is over subsets $S \subseteq \{1, \ldots, i\}$ that satisfy $\sum_{j \in S} w_j \leq w$. Using this new set of subproblems, we will be able to express the value $\text{OPT}(i, w)$ as a simple expression in terms of values from smaller problems. Moreover, $\text{OPT}(n, W)$ is the quantity we're looking for in the end. As before, let $\mathcal{O}$ denote an optimum solution for the original problem.

- If $n \notin \mathcal{O}$, then $\text{OPT}(n, W) = \text{OPT}(n-1, W)$, since we can simply ignore item n.
- If $n \in \mathcal{O}$, then $\text{OPT}(n, W) = w_n + \text{OPT}(n-1, W - w_n)$, since we now seek to use the remaining capacity of $W - w_n$ in an optimal way across items $1, 2, \ldots, n-1$.

When the n^{th} item is too big, that is, $W < w_n$, then we must have $\text{OPT}(n, W) = \text{OPT}(n-1, W)$. Otherwise, we get the optimum solution allowing all n requests by taking the better of these two options. Using the same line of argument for the subproblem for items $\{1, \ldots, i\}$, and maximum allowed weight w, gives us the following recurrence.

(6.8) *If $w < w_i$ then $\text{OPT}(i, w) = \text{OPT}(i-1, w)$. Otherwise*

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w), w_i + \text{OPT}(i-1, w - w_i)).$$

As before, we want to design an algorithm that builds up a table of all $\text{OPT}(i, w)$ values while computing each of them at most once.

```
Subset-Sum(n, W)
  Array M[0...n, 0...W]
  Initialize M[0, w] = 0 for each w = 0, 1, ..., W
  For i = 1, 2, ..., n
     For w = 0, ..., W
        Use the recurrence (6.8) to compute M[i, w]
     Endfor
  Endfor
  Return M[n, W]
```

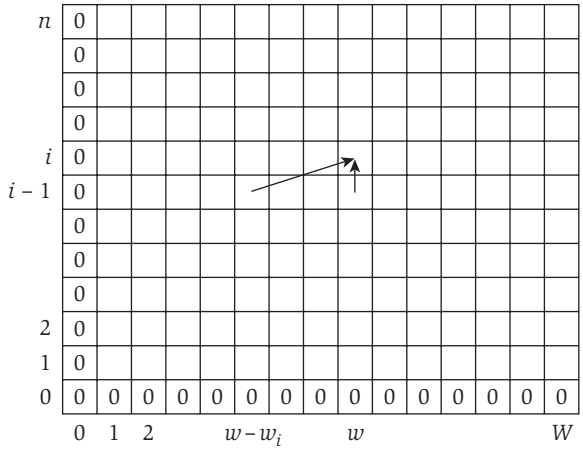

Figure 6.11 The two-dimensional table of OPT values. The leftmost column and bottom row is always 0. The entry for $\text{OPT}(i, w)$ is computed from the two other entries $\text{OPT}(i-1, w)$ and $\text{OPT}(i-1, w-w_i)$, as indicated by the arrows.

Using (6.8) one can immediately prove by induction that the returned value $M[n, W]$ is the optimum solution value for the requests $1, \ldots, n$ and available weight W.

Analyzing the Algorithm

Recall the tabular picture we considered in Figure 6.5, associated with weighted interval scheduling, where we also showed the way in which the array M for that algorithm was iteratively filled in. For the algorithm we've just designed, we can use a similar representation, but we need a two-dimensional table, reflecting the two-dimensional array of subproblems that is being built up. Figure 6.11 shows the building up of subproblems in this case: the value $M[i, w]$ is computed from the two other values $M[i-1, w]$ and $M[i-1, w-w_i]$.

As an example of this algorithm executing, consider an instance with weight limit $W = 6$, and $n = 3$ items of sizes $w_1 = w_2 = 2$ and $w_3 = 3$. We find that the optimal value $\text{OPT}(3, 6) = 5$ (which we get by using the third item and one of the first two items). Figure 6.12 illustrates the way the algorithm fills in the two-dimensional table of OPT values row by row.

Next we will worry about the running time of this algorithm. As before in the case of weighted interval scheduling, we are building up a table of solutions M, and we compute each of the values $M[i, w]$ in $O(1)$ time using the previous values. Thus the running time is proportional to the number of entries in the table.

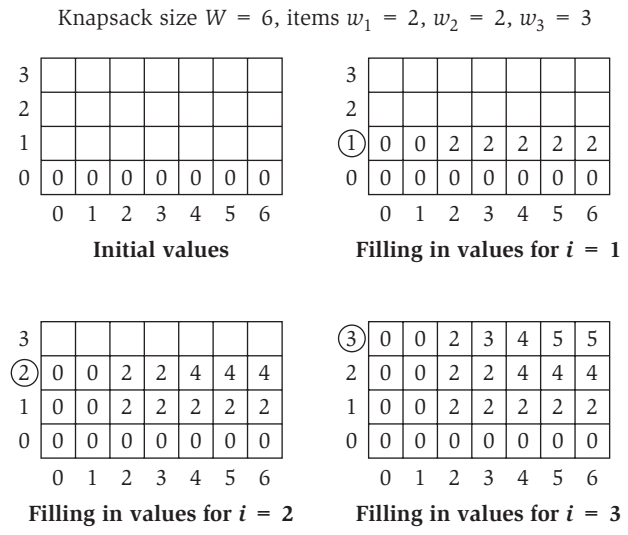

Figure 6.12 The iterations of the algorithm on a sample instance of the Subset Sum Problem.

(6.9) *The* `Subset-Sum`(n, W) *Algorithm correctly computes the optimal value of the problem, and runs in* $O(nW)$ *time.*

Note that this method is not as efficient as our dynamic program for the Weighted Interval Scheduling Problem. Indeed, its running time is not a polynomial function of n; rather, it is a polynomial function of n and W, the largest integer involved in defining the problem. We call such algorithms *pseudo-polynomial*. Pseudo-polynomial algorithms can be reasonably efficient when the numbers $\{w_i\}$ involved in the input are reasonably small; however, they become less practical as these numbers grow large.

To recover an optimal set S of items, we can trace back through the array M by a procedure similar to those we developed in the previous sections.

(6.10) *Given a table* M *of the optimal values of the subproblems, the optimal set* S *can be found in* $O(n)$ *time.*

Extension: The Knapsack Problem

The Knapsack Problem is a bit more complex than the scheduling problem we discussed earlier. Consider a situation in which each item i has a nonnegative weight w_i as before, and also a distinct *value* v_i. Our goal is now to find a

subset S of maximum value $\sum_{i \in S} v_i$, subject to the restriction that the total weight of the set should not exceed W: $\sum_{i \in S} w_i \leq W$.

It is not hard to extend our dynamic programming algorithm to this more general problem. We use the analogous set of subproblems, $\text{OPT}(i, w)$, to denote the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ and maximum available weight w. We consider an optimal solution $\mathcal{O}$, and identify two cases depending on whether or not $n \in \mathcal{O}$.

- If $n \notin \mathcal{O}$, then $\text{OPT}(n, W) = \text{OPT}(n-1, W)$.
- If $n \in \mathcal{O}$, then $\text{OPT}(n, W) = v_n + \text{OPT}(n-1, W - w_n)$.

Using this line of argument for the subproblems implies the following analogue of (6.8).

(6.11) *If $w < w_i$ then $\text{OPT}(i, w) = \text{OPT}(i-1, w)$. Otherwise*

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w - w_i)).$$

Using this recurrence, we can write down a completely analogous dynamic programming algorithm, and this implies the following fact.

(6.12) *The Knapsack Problem can be solved in $O(nW)$ time.*

6.5 RNA Secondary Structure: Dynamic Programming over Intervals

In the Knapsack Problem, we were able to formulate a dynamic programming algorithm by adding a new variable. A different but very common way by which one ends up adding a variable to a dynamic program is through the following scenario. We start by thinking about the set of subproblems on $\{1, 2, \ldots, j\}$, for all choices of j, and find ourselves unable to come up with a natural recurrence. We then look at the larger set of subproblems on $\{i, i+1, \ldots, j\}$ for all choices of i and j (where $i \leq j$), and find a natural recurrence relation on these subproblems. In this way, we have added the second variable i; the effect is to consider a subproblem for every contiguous *interval* in $\{1, 2, \ldots, n\}$.

There are a few canonical problems that fit this profile; those of you who have studied parsing algorithms for context-free grammars have probably seen at least one dynamic programming algorithm in this style. Here we focus on the problem of RNA secondary structure prediction, a fundamental issue in computational biology.