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.

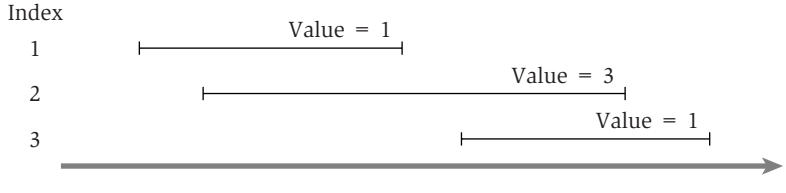


Figure 6.1 A simple instance of weighted interval scheduling.

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.

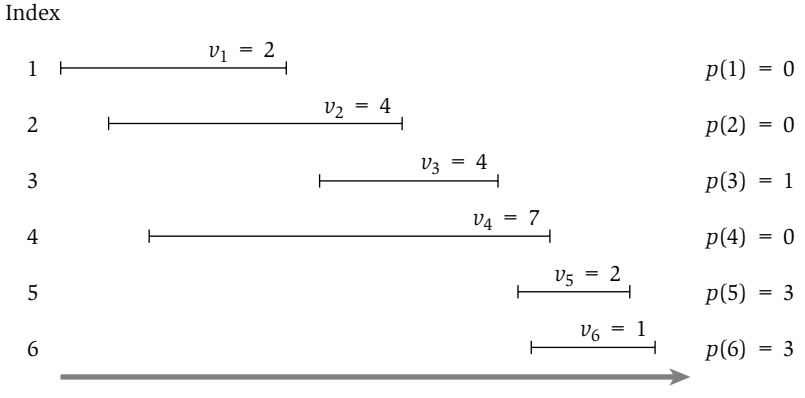


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

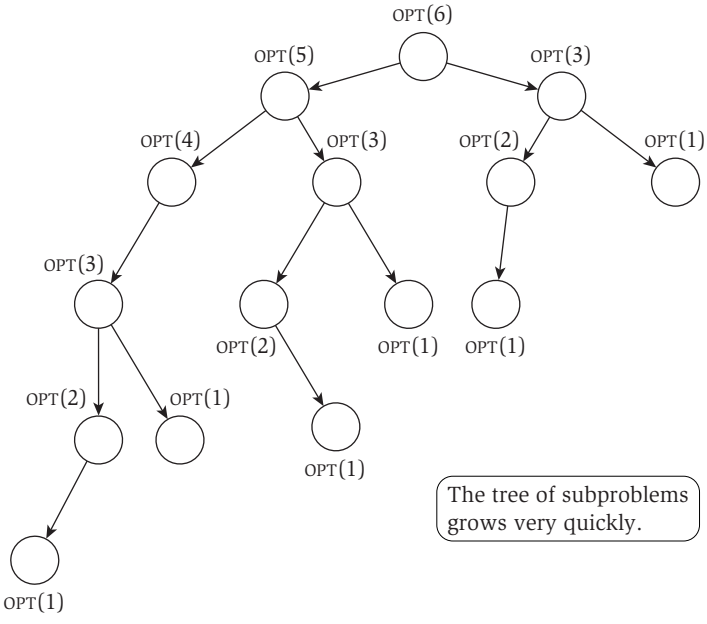


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

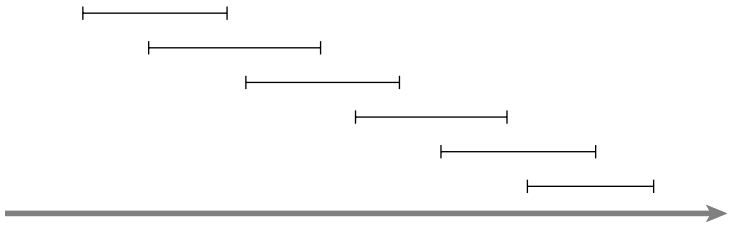


Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

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.

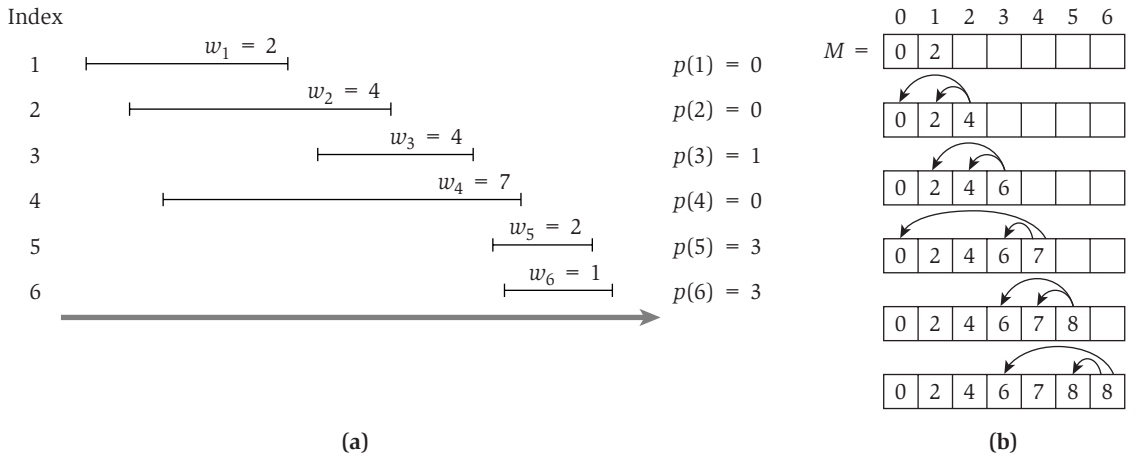


Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

(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:

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

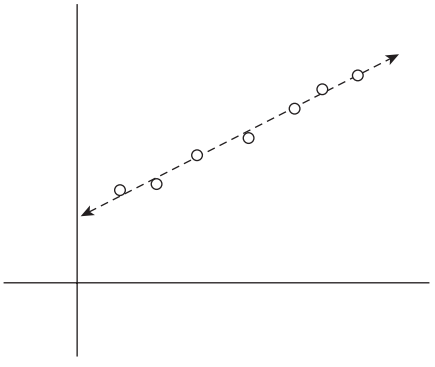


Figure 6.6 A "line of best fit."

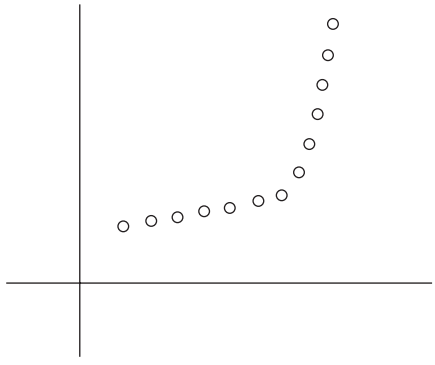


Figure 6.7 A set of points that lie approximately on two lines.

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.

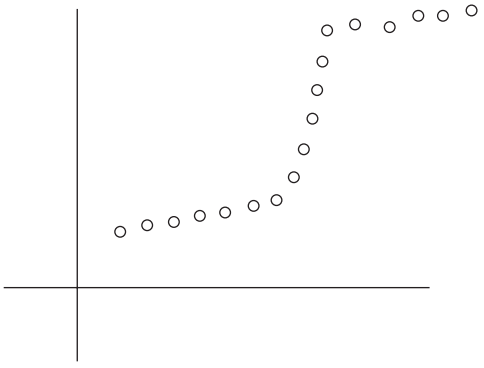


Figure 6.8 A set of points that lie approximately on three lines.

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 $p_1, \ldots, p_{i-1}$.

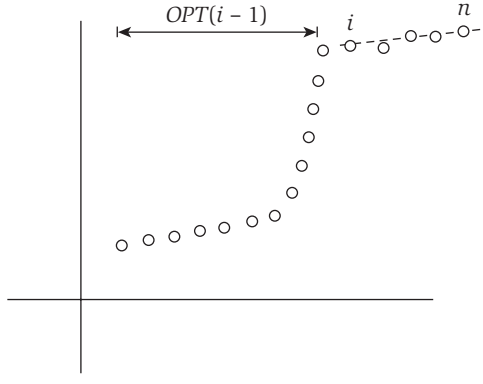


Figure 6.9 A possible solution: a single line segment fits points $p_i, p_{i+1}, \ldots, p_n$, and then an optimal solution is found for the remaining points $p_1, p_2, \ldots, p_{i-1}$.

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

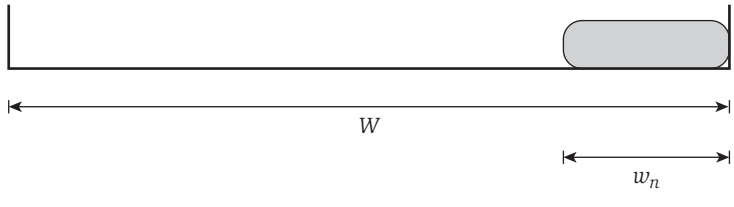


Figure 6.10 After item n is included in the solution, a weight of w_n is used up and there is $W - w_n$ available weight left.

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]
```

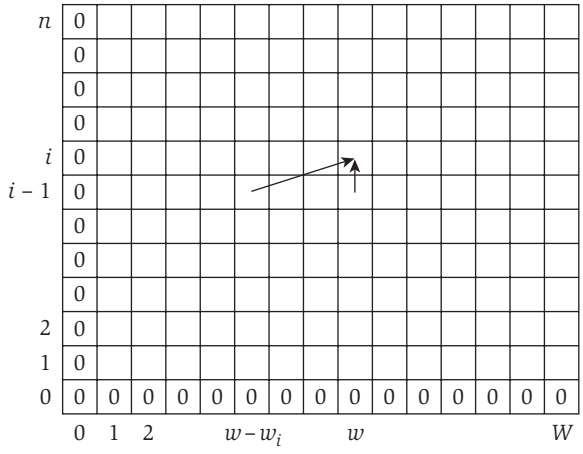


Figure 6.11 The two-dimensional table of OPT values. The leftmost column and bottom row is always 0. The entry for $\text{OPT}(i, w)$ is computed from the two other entries $\text{OPT}(i-1, w)$ and $\text{OPT}(i-1, w-w_i)$, as indicated by the arrows.

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.

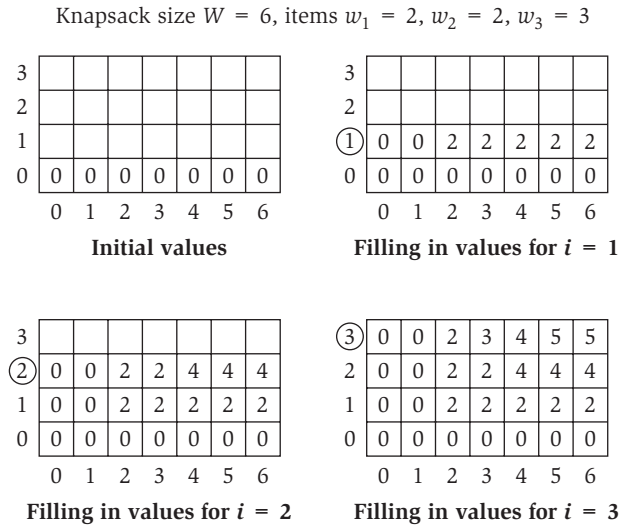


Figure 6.12 The iterations of the algorithm on a sample instance of the Subset Sum Problem.

(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.

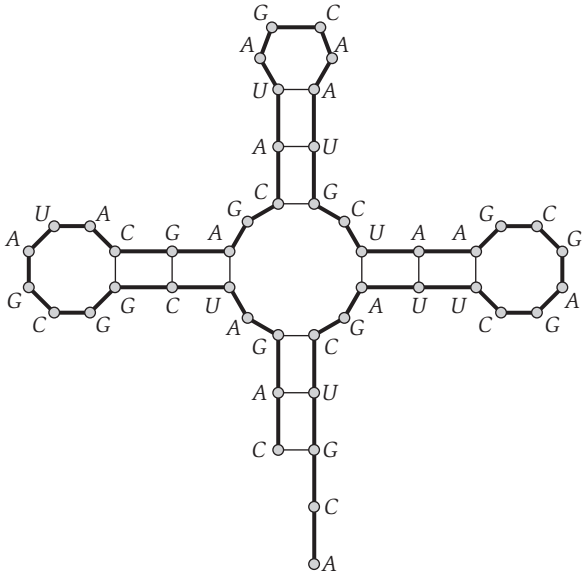


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

The Problem

As one learns in introductory biology classes, Watson and Crick posited that double-stranded DNA is "zipped" together by complementary base-pairing. Each strand of DNA can be viewed as a string of *bases*, where each base is drawn from the set $\{A, C, G, T\}$.[2] The bases A and T pair with each other, and the bases C and G pair with each other; it is these A-T and C-G pairings that hold the two strands together.

Now, single-stranded RNA molecules are key components in many of the processes that go on inside a cell, and they follow more or less the same structural principles. However, unlike double-stranded DNA, there's no "second strand" for the RNA to stick to; so it tends to loop back and form base pairs with itself, resulting in interesting shapes like the one depicted in Figure 6.13. The set of pairs (and resulting shape) formed by the RNA molecule through this process is called the *secondary structure*, and understanding the secondary structure is essential for understanding the behavior of the molecule.

[2] Adenine, cytosine, guanine, and thymine, the four basic units of DNA.

For our purposes, a single-stranded RNA molecule can be viewed as a sequence of n symbols (bases) drawn from the alphabet $\{A, C, G, U\}$.[3] Let $B = b_1 b_2 \cdots b_n$ be a single-stranded RNA molecule, where each $b_i \in \{A, C, G, U\}$. To a first approximation, one can model its secondary structure as follows. As usual, we require that A pairs with U, and C pairs with G; we also require that each base can pair with at most one other base—in other words, the set of base pairs forms a *matching*. It also turns out that secondary structures are (again, to a first approximation) "knot-free," which we will formalize as a kind of *noncrossing* condition below.

Thus, concretely, we say that a *secondary structure on* B is a set of pairs $S = \{(i, j)\}$, where $i, j \in \{1, 2, \ldots, n\}$, that satisfies the following conditions.

(i) *(No sharp turns.)* The ends of each pair in S are separated by at least four intervening bases; that is, if $(i, j) \in S$, then $i < j - 4$.

(ii) The elements of any pair in S consist of either $\{A, U\}$ or $\{C, G\}$ (in either order).

(iii) S is a matching: no base appears in more than one pair.

(iv) *(The noncrossing condition.)* If (i, j) and (k, ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$. (See Figure 6.14 for an illustration.)

Note that the RNA secondary structure in Figure 6.13 satisfies properties (i) through (iv). From a structural point of view, condition (i) arises simply because the RNA molecule cannot bend too sharply; and conditions (ii) and (iii) are the fundamental Watson-Crick rules of base-pairing. Condition (iv) is the striking one, since it's not obvious why it should hold in nature. But while there are sporadic exceptions to it in real molecules (via so-called *pseudo-knotting*), it does turn out to be a good approximation to the spatial constraints on real RNA secondary structures.

Now, out of all the secondary structures that are possible for a single RNA molecule, which are the ones that are likely to arise under physiological conditions? The usual hypothesis is that a single-stranded RNA molecule will form the secondary structure with the optimum total free energy. The correct model for the free energy of a secondary structure is a subject of much debate; but a first approximation here is to assume that the free energy of a secondary structure is proportional simply to the *number* of base pairs that it contains.

Thus, having said all this, we can state the basic RNA secondary structure prediction problem very simply: We want an efficient algorithm that takes

[3] Note that the symbol T from the alphabet of DNA has been replaced by a U, but this is not important for us here.

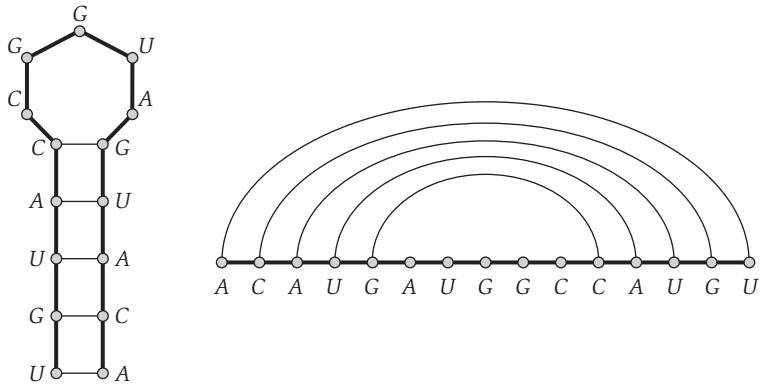


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

a single-stranded RNA molecule $B = b_1 b_2 \cdots b_n$ and determines a secondary structure S with the maximum possible number of base pairs.

Designing and Analyzing the Algorithm

A First Attempt at Dynamic Programming The natural first attempt to apply dynamic programming would presumably be based on the following subproblems: We say that OPT(j) is the maximum number of base pairs in a secondary structure on $b_1 b_2 \cdots b_j$. By the no-sharp-turns condition above, we know that OPT(j) $= 0$ for $j \leq 5$; and we know that OPT(n) is the solution we're looking for.

The trouble comes when we try writing down a recurrence that expresses OPT(j) in terms of the solutions to smaller subproblems. We can get partway there: in the optimal secondary structure on $b_1 b_2 \cdots b_j$, it's the case that either

- j is not involved in a pair; or
- j pairs with t for some $t < j - 4$.

In the first case, we just need to consult our solution for OPT($j - 1$). The second case is depicted in Figure 6.15(a); because of the noncrossing condition, we now know that no pair can have one end between 1 and $t - 1$ and the other end between $t + 1$ and $j - 1$. We've therefore effectively isolated two new subproblems: one on the bases $b_1 b_2 \cdots b_{t-1}$, and the other on the bases $b_{t+1} \cdots b_{j-1}$. The first is solved by OPT($t - 1$), but the second is not on our list of subproblems, because it does not begin with b_1.

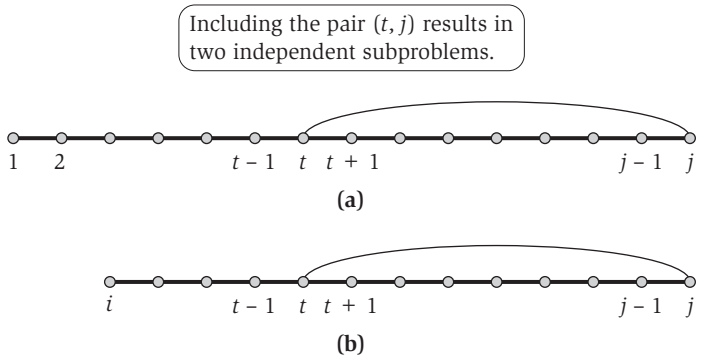


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

This is the insight that makes us realize we need to add a variable. We need to be able to work with subproblems that do not begin with b_1; in other words, we need to consider subproblems on $b_i b_{i+1} \cdots b_j$ for all choices of $i \leq j$.

Dynamic Programming over Intervals Once we make this decision, our previous reasoning leads straight to a successful recurrence. Let $\text{OPT}(i, j)$ denote the maximum number of base pairs in a secondary structure on $b_i b_{i+1} \cdots b_j$. The no-sharp-turns condition lets us initialize $\text{OPT}(i, j) = 0$ whenever $i \geq j - 4$. (For notational convenience, we will also allow ourselves to refer to $\text{OPT}(i, j)$ even when $i > j$; in this case, its value is 0.)

Now, in the optimal secondary structure on $b_i b_{i+1} \cdots b_j$, we have the same alternatives as before:

- j is not involved in a pair; or
- j pairs with t for some $t < j - 4$.

In the first case, we have $\text{OPT}(i, j) = \text{OPT}(i, j - 1)$. In the second case, depicted in Figure 6.15(b), we recur on the two subproblems $\text{OPT}(i, t - 1)$ and $\text{OPT}(t + 1, j - 1)$; as argued above, the noncrossing condition has isolated these two subproblems from each other.

We have therefore justified the following recurrence.

(6.13) $\text{OPT}(i, j) = \max(\text{OPT}(i, j - 1), \max(1 + \text{OPT}(i, t - 1) + \text{OPT}(t + 1, j - 1)))$, *where the* max *is taken over* t *such that* b_t *and* b_j *are an allowable base pair (under conditions (i) and (ii) from the definition of a secondary structure).*

Now we just have to make sure we understand the proper order in which to build up the solutions to the subproblems. The form of (6.13) reveals that we're always invoking the solution to subproblems on *shorter* intervals: those

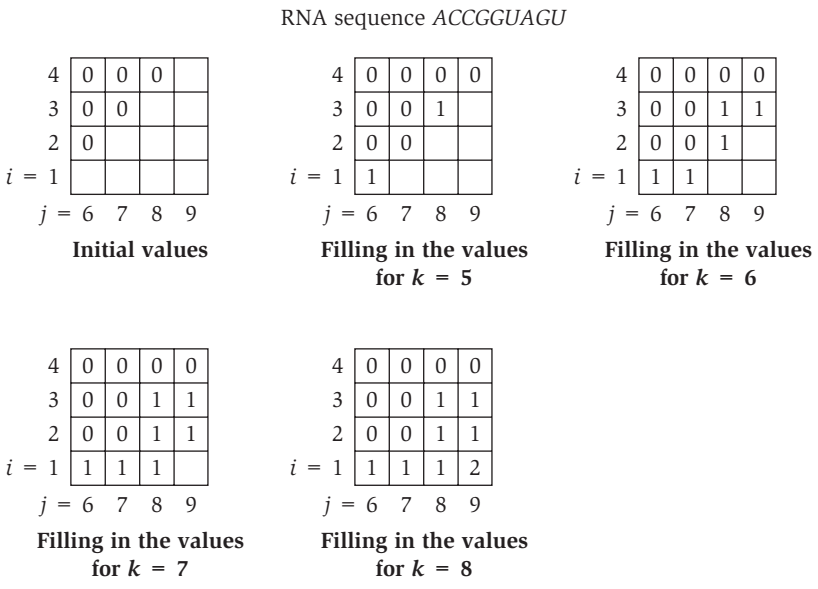


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

for which $k = j - i$ is smaller. Thus things will work without any trouble if we build up the solutions in order of increasing interval length.

```
Initialize OPT(i, j) = 0 whenever i ≥ j − 4
For k = 5, 6, ..., n − 1
  For i = 1, 2, ... n − k
    Set j = i + k
    Compute OPT(i, j) using the recurrence in (6.13)
  Endfor
Endfor
Return OPT(1, n)
```

As an example of this algorithm executing, we consider the input *ACCGGUAGU*, a subsequence of the sequence in Figure 6.14. As with the Knapsack Problem, we need two dimensions to depict the array M: one for the left endpoint of the interval being considered, and one for the right endpoint. In the figure, we only show entries corresponding to $[i, j]$ pairs with $i < j - 4$, since these are the only ones that can possibly be nonzero.

It is easy to bound the running time: there are $O(n^2)$ subproblems to solve, and evaluating the recurrence in (6.13) takes time $O(n)$ for each. Thus the running time is $O(n^3)$.

As always, we can recover the secondary structure itself (not just its value) by recording how the minima in (6.13) are achieved and tracing back through the computation.

6.6 Sequence Alignment

For the remainder of this chapter, we consider two further dynamic programming algorithms that each have a wide range of applications. In the next two sections we discuss *sequence alignment*, a fundamental problem that arises in comparing strings. Following this, we turn to the problem of computing shortest paths in graphs when edges have costs that may be negative.

The Problem

Dictionaries on the Web seem to get more and more useful: often it seems easier to pull up a bookmarked online dictionary than to get a physical dictionary down from the bookshelf. And many online dictionaries offer functions that you can't get from a printed one: if you're looking for a definition and type in a word it doesn't contain—say, *ocurrance*—it will come back and ask, "Perhaps you mean *occurrence*?" How does it do this? Did it truly know what you had in mind?

Let's defer the second question to a different book and think a little about the first one. To decide what you probably meant, it would be natural to search the dictionary for the word most "similar" to the one you typed in. To do this, we have to answer the question: How should we define similarity between two words or strings?

Intuitively, we'd like to say that *ocurrance* and *occurrence* are similar because we can make the two words identical if we add a *c* to the first word and change the *a* to an *e*. Since neither of these changes seems so large, we conclude that the words are quite similar. To put it another way, we can *nearly* line up the two words letter by letter:

```
o-currance
occurrence
```

The hyphen (-) indicates a *gap* where we had to add a letter to the second word to get it to line up with the first. Moreover, our lining up is not perfect in that an *e* is lined up with an *a*.

We want a model in which similarity is determined roughly by the number of gaps and mismatches we incur when we line up the two words. Of course, there are many possible ways to line up the two words; for example, we could have written

```
o-curr-ance
occurre-nce
```

which involves three gaps and no mismatches. Which is better: one gap and one mismatch, or three gaps and no mismatches?

This discussion has been made easier because we know roughly what the correspondence ought to look like. When the two strings don't look like English words—for example, abbbaabbbbaab and ababaaabbbbbab—it may take a little work to decide whether they can be lined up nicely or not:

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

Dictionary interfaces and spell-checkers are not the most computationally intensive application for this type of problem. In fact, determining similarities among strings is one of the central computational problems facing molecular biologists today.

Strings arise very naturally in biology: an organism's *genome*—its full set of genetic material—is divided up into giant linear DNA molecules known as *chromosomes,* each of which serves conceptually as a one-dimensional chemical storage device. Indeed, it does not obscure reality very much to think of it as an enormous linear *tape*, containing a string over the alphabet $\{A, C, G, T\}$. The string of symbols encodes the instructions for building protein molecules; using a chemical mechanism for reading portions of the chromosome, a cell can construct proteins that in turn control its metabolism.

Why is similarity important in this picture? To a first approximation, the sequence of symbols in an organism's genome can be viewed as determining the properties of the organism. So suppose we have two strains of bacteria, X and Y, which are closely related evolutionarily. Suppose further that we've determined that a certain substring in the DNA of X codes for a certain kind of toxin. Then, if we discover a very "similar" substring in the DNA of Y, we might be able to hypothesize, before performing any experiments at all, that this portion of the DNA in Y codes for a similar kind of toxin. This use of computation to guide decisions about biological experiments is one of the hallmarks of the field of *computational biology*.

All this leaves us with the same question we asked initially, while typing badly spelled words into our online dictionary: How should we define the notion of *similarity* between two strings?

In the early 1970s, the two molecular biologists Needleman and Wunsch proposed a definition of similarity, which, basically unchanged, has become

the standard definition in use today. Its position as a standard was reinforced by its simplicity and intuitive appeal, as well as through its independent discovery by several other researchers around the same time. Moreover, this definition of similarity came with an efficient dynamic programming algorithm to compute it. In this way, the paradigm of dynamic programming was independently discovered by biologists some twenty years after mathematicians and computer scientists first articulated it.

The definition is motivated by the considerations we discussed above, and in particular by the notion of "lining up" two strings. Suppose we are given two strings X and Y, where X consists of the sequence of symbols $x_1x_2 \cdots x_m$ and Y consists of the sequence of symbols $y_1y_2 \cdots y_n$. Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ as representing the different positions in the strings X and Y, and consider a matching of these sets; recall that a *matching* is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching M of these two sets is an *alignment* if there are no "crossing" pairs: if $(i, j), (i', j') \in M$ and $i < i'$, then $j < j'$. Intuitively, an alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another. Thus, for example,

```
stop-
-tops
```

corresponds to the alignment $\{(2, 1), (3, 2), (4, 3)\}$.

Our definition of similarity will be based on finding the *optimal* alignment between X and Y, according to the following criteria. Suppose M is a given alignment between X and Y.

- First, there is a parameter $\delta > 0$ that defines a *gap penalty*. For each position of X or Y that is not matched in M—it is a *gap*—we incur a cost of δ.
- Second, for each pair of letters p, q in our alphabet, there is a *mismatch cost* of α_{pq} for lining up p with q. Thus, for each $(i, j) \in M$, we pay the appropriate mismatch cost $\alpha_{x_i y_j}$ for lining up x_i with y_j. One generally assumes that $\alpha_{pp} = 0$ for each letter p—there is no mismatch cost to line up a letter with another copy of itself—although this will not be necessary in anything that follows.
- The *cost* of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost.

The process of minimizing this cost is often referred to as *sequence alignment* in the biology literature. The quantities δ and $\{\alpha_{pq}\}$ are external parameters that must be plugged into software for sequence alignment; indeed, a lot of work goes into choosing the settings for these parameters. From our point of

view, in designing an algorithm for sequence alignment, we will take them as given. To go back to our first example, notice how these parameters determine which alignment of *ocurrance* and *occurrence* we should prefer: the first is strictly better if and only if $\delta + \alpha_{ae} < 3\delta$.

Designing the Algorithm

We now have a concrete numerical definition for the similarity between strings X and Y: it is the minimum cost of an alignment between X and Y. The lower this cost, the more similar we declare the strings to be. We now turn to the problem of computing this minimum cost, and an optimal alignment that yields it, for a given pair of strings X and Y.

One of the approaches we could try for this problem is dynamic programming, and we are motivated by the following basic dichotomy.

- In the optimal alignment M, either $(m, n) \in M$ or $(m, n) \notin M$. (That is, either the last symbols in the two strings are matched to each other, or they aren't.)

By itself, this fact would be too weak to provide us with a dynamic programming solution. Suppose, however, that we compound it with the following basic fact.

(6.14) *Let M be any alignment of X and Y. If $(m, n) \notin M$, then either the m^{th} position of X or the n^{th} position of Y is not matched in M.*

Proof. Suppose by way of contradiction that $(m, n) \notin M$, and there are numbers $i < m$ and $j < n$ so that $(m, j) \in M$ and $(i, n) \in M$. But this contradicts our definition of *alignment*: we have $(i, n), (m, j) \in M$ with $i < m$, but $n > i$ so the pairs (i, n) and (m, j) cross. ■

There is an equivalent way to write (6.14) that exposes three alternative possibilities, and leads directly to the formulation of a recurrence.

(6.15) *In an optimal alignment M, at least one of the following is true:*

(i) $(m, n) \in M$*; or*

(ii) the m^{th} position of X is not matched; or

(iii) the n^{th} position of Y is not matched.

Now, let $\text{OPT}(i, j)$ denote the minimum cost of an alignment between $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$. If case (i) of (6.15) holds, we pay $\alpha_{x_m y_n}$ and then align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_{n-1}$; we get $\text{OPT}(m, n) = \alpha_{x_m y_n} + \text{OPT}(m-1, n-1)$. If case (ii) holds, we pay a gap cost of δ since the m^{th} position of X is not matched, and then we align $x_1x_2 \cdots x_{m-1}$ as well as

possible with $y_1 y_2 \cdots y_n$. In this way, we get $\text{OPT}(m, n) = \delta + \text{OPT}(m-1, n)$. Similarly, if case (iii) holds, we get $\text{OPT}(m, n) = \delta + \text{OPT}(m, n-1)$.

Using the same argument for the subproblem of finding the minimum-cost alignment between $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$, we get the following fact.

(6.16) *The minimum alignment costs satisfy the following recurrence for $i \geq 1$ and $j \geq 1$:*

$$\text{OPT}(i, j) = \min[\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)].$$

Moreover, (i, j) is in an optimal alignment M for this subproblem if and only if the minimum is achieved by the first of these values.

We have maneuvered ourselves into a position where the dynamic programming algorithm has become clear: We build up the values of $\text{OPT}(i, j)$ using the recurrence in (6.16). There are only $O(mn)$ subproblems, and $\text{OPT}(m, n)$ is the value we are seeking.

We now specify the algorithm to compute the value of the optimal alignment. For purposes of initialization, we note that $\text{OPT}(i, 0) = \text{OPT}(0, i) = i\delta$ for all i, since the only way to line up an i-letter word with a 0-letter word is to use i gaps.

```
Alignment(X,Y)
  Array A[0...m, 0...n]
  Initialize A[i, 0] = iδ for each i
  Initialize A[0, j] = jδ for each j
  For j = 1, ..., n
     For i = 1, ..., m
         Use the recurrence (6.16) to compute A[i, j]
     Endfor
  Endfor
  Return A[m, n]
```

As in previous dynamic programming algorithms, we can trace back through the array A, using the second part of fact (6.16), to construct the alignment itself.

Analyzing the Algorithm

The correctness of the algorithm follows directly from (6.16). The running time is $O(mn)$, since the array A has $O(mn)$ entries, and at worst we spend constant time on each.

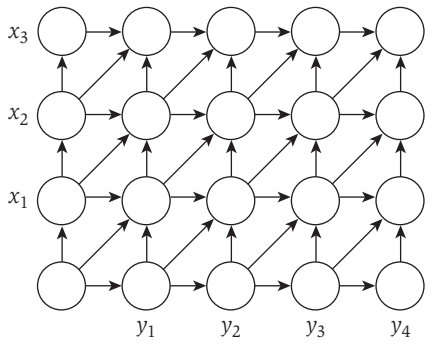


Figure 6.17 A graph-based picture of sequence alignment.

There is an appealing pictorial way in which people think about this sequence alignment algorithm. Suppose we build a two-dimensional $m \times n$ grid graph G_{XY}, with the rows labeled by symbols in the string X, the columns labeled by symbols in Y, and directed edges as in Figure 6.17.

We number the rows from 0 to m and the columns from 0 to n; we denote the node in the i^{th} row and the j^{th} column by the label (i, j). We put *costs* on the edges of G_{XY}: the cost of each horizontal and vertical edge is δ, and the cost of the diagonal edge from $(i-1, j-1)$ to (i, j) is $\alpha_{x_i y_j}$.

The purpose of this picture now emerges: the recurrence in (6.16) for $\text{OPT}(i, j)$ is precisely the recurrence one gets for the minimum-cost path in G_{XY} from $(0, 0)$ to (i, j). Thus we can show

(6.17) *Let $f(i, j)$ denote the minimum cost of a path from $(0, 0)$ to (i, j) in G_{XY}. Then for all i, j, we have $f(i, j) = \text{OPT}(i, j)$.*

Proof. We can easily prove this by induction on $i + j$. When $i + j = 0$, we have $i = j = 0$, and indeed $f(i, j) = \text{OPT}(i, j) = 0$.

Now consider arbitrary values of i and j, and suppose the statement is true for all pairs (i', j') with $i' + j' < i + j$. The last edge on the shortest path to (i, j) is either from $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$. Thus we have

$$\begin{aligned} f(i, j) &= \min[\alpha_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)] \\ &= \min[\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)] \\ &= \text{OPT}(i, j), \end{aligned}$$

where we pass from the first line to the second using the induction hypothesis, and we pass from the second to the third using (6.16). ■

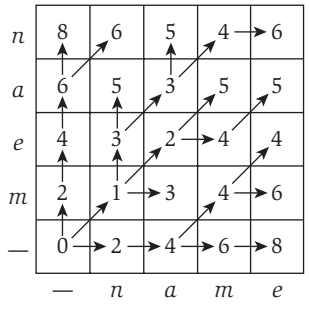


Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

Thus the value of the optimal alignment is the length of the shortest path in G_{XY} from $(0, 0)$ to (m, n). (We'll call any path in G_{XY} from $(0, 0)$ to (m, n) a *corner-to-corner path*.) Moreover, the diagonal edges used in a shortest path correspond precisely to the pairs used in a minimum-cost alignment. These connections to the Shortest-Path Problem in the graph G_{XY} do not directly yield an improvement in the running time for the sequence alignment problem; however, they do help one's intuition for the problem and have been useful in suggesting algorithms for more complex variations on sequence alignment.

For an example, Figure 6.18 shows the value of the shortest path from $(0, 0)$ to each node (i, j) for the problem of aligning the words *mean* and *name*. For the purpose of this example, we assume that $\delta = 2$; matching a vowel with a different vowel, or a consonant with a different consonant, costs 1; while matching a vowel and a consonant with each other costs 3. For each cell in the table (representing the corresponding node), the arrow indicates the last step of the shortest path leading to that node—in other words, the way that the minimum is achieved in (6.16). Thus, by following arrows backward from node $(4, 4)$, we can trace back to construct the alignment.

6.7 Sequence Alignment in Linear Space via Divide and Conquer

In the previous section, we showed how to compute the optimal alignment between two strings X and Y of lengths m and n, respectively. Building up the two-dimensional m-by-n array of optimal solutions to subproblems, OPT$(\cdot, \cdot)$, turned out to be equivalent to constructing a graph G_{XY} with mn nodes laid out in a grid and looking for the cheapest path between opposite corners. In either of these ways of formulating the dynamic programming algorithm, the running time is $O(mn)$, because it takes constant time to determine the value in each of the mn cells of the array OPT; and the space requirement is $O(mn)$ as well, since it was dominated by the cost of storing the array (or the graph G_{XY}).

The Problem

The question we ask in this section is: Should we be happy with $O(mn)$ as a space bound? If our application is to compare English words, or even English sentences, it is quite reasonable. In biological applications of sequence alignment, however, one often compares very long strings against one another; and in these cases, the $\Theta(mn)$ space requirement can potentially be a more severe problem than the $\Theta(mn)$ time requirement. Suppose, for example, that we are comparing two strings of 100,000 symbols each. Depending on the underlying processor, the prospect of performing roughly 10 billion primitive

operations might be less cause for worry than the prospect of working with a single 10-gigabyte array.

Fortunately, this is not the end of the story. In this section we describe a very clever enhancement of the sequence alignment algorithm that makes it work in $O(mn)$ time using only $O(m+n)$ space. In other words, we can bring the space requirement down to linear while blowing up the running time by at most an additional constant factor. For ease of presentation, we'll describe various steps in terms of paths in the graph G_{XY}, with the natural equivalence back to the sequence alignment problem. Thus, when we seek the pairs in an optimal alignment, we can equivalently ask for the edges in a shortest corner-to-corner path in G_{XY}.

The algorithm itself will be a nice application of divide-and-conquer ideas. The crux of the technique is the observation that, if we divide the problem into several recursive calls, then the space needed for the computation can be reused from one call to the next. The way in which this idea is used, however, is fairly subtle.

Designing the Algorithm

We first show that if we only care about the *value* of the optimal alignment, and not the alignment itself, it is easy to get away with linear space. The crucial observation is that to fill in an entry of the array A, the recurrence in (6.16) only needs information from the current column of A and the previous column of A. Thus we will "collapse" the array A to an $m \times 2$ array B: as the algorithm iterates through values of j, entries of the form $B[i,0]$ will hold the "previous" column's value $A[i,j-1]$, while entries of the form $B[i,1]$ will hold the "current" column's value $A[i,j]$.

```
Space-Efficient-Alignment(X,Y)
  Array B[0...m, 0...1]
  Initialize B[i, 0] = iδ for each i (just as in column 0 of A)
  For j = 1, ..., n
     B[0, 1] = jδ (since this corresponds to entry A[0, j])
     For i = 1, ..., m
          B[i, 1] = min[α_{x_i y_j} + B[i − 1, 0],
                        δ + B[i − 1, 1],  δ + B[i, 0]]
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
          Update B[i, 0] = B[i, 1] for each i
  Endfor
```

It is easy to verify that when this algorithm completes, the array entry $B[i, 1]$ holds the value of $\text{OPT}(i, n)$ for $i = 0, 1, \ldots, m$. Moreover, it uses $O(mn)$ time and $O(m)$ space. The problem is: where is the alignment itself? We haven't left enough information around to be able to run a procedure like `Find-Alignment`. Since B at the end of the algorithm only contains the last two columns of the original dynamic programming array A, if we were to try tracing back to get the path, we'd run out of information after just these two columns. We could imagine getting around this difficulty by trying to "predict" what the alignment is going to be in the process of running our space-efficient procedure. In particular, as we compute the values in the j^{th} column of the (now implicit) array A, we could try hypothesizing that a certain entry has a very small value, and hence that the alignment that passes through this entry is a promising candidate to be the optimal one. But this promising alignment might run into big problems later on, and a different alignment that currently looks much less attractive could turn out to be the optimal one.

There is, in fact, a solution to this problem—we will be able to recover the alignment itself using $O(m + n)$ space—but it requires a genuinely new idea. The insight is based on employing the divide-and-conquer technique that we've seen earlier in the book. We begin with a simple alternative way to implement the basic dynamic programming solution.

A Backward Formulation of the Dynamic Program Recall that we use $f(i, j)$ to denote the length of the shortest path from $(0, 0)$ to (i, j) in the graph G_{XY}. (As we showed in the initial sequence alignment algorithm, $f(i, j)$ has the same value as $\text{OPT}(i, j)$.) Now let's define $g(i, j)$ to be the length of the shortest path from (i, j) to (m, n) in G_{XY}. The function g provides an equally natural dynamic programming approach to sequence alignment, except that we build it up in reverse: we start with $g(m, n) = 0$, and the answer we want is $g(0, 0)$. By strict analogy with (6.16), we have the following recurrence for g.

(6.18) *For $i < m$ and $j < n$ we have*

$$g(i, j) = \min[\alpha_{x_{i+1}y_{j+1}} + g(i + 1, j + 1), \delta + g(i, j + 1), \delta + g(i + 1, j)].$$

This is just the recurrence one obtains by taking the graph G_{XY}, "rotating" it so that the node (m, n) is in the lower left corner, and using the previous approach. Using this picture, we can also work out the full dynamic programming algorithm to build up the values of g, *backward* starting from (m, n). Similarly, there is a space-efficient version of this backward dynamic programming algorithm, analogous to `Space-Efficient-Alignment`, which computes the value of the optimal alignment using only $O(m + n)$ space. We will refer to

this backward version, naturally enough, as `Backward-Space-Efficient-Alignment`.

Combining the Forward and Backward Formulations So now we have symmetric algorithms which build up the values of the functions f and g. The idea will be to use these two algorithms in concert to find the optimal alignment. First, here are two basic facts summarizing some relationships between the functions f and g.

(6.19) *The length of the shortest corner-to-corner path in G_{XY} that passes through (i, j) is $f(i, j) + g(i, j)$.*

Proof. Let ℓ_{ij} denote the length of the shortest corner-to-corner path in G_{XY} that passes through (i, j). Clearly, any such path must get from $(0, 0)$ to (i, j) and then from (i, j) to (m, n). Thus its length is at least $f(i, j) + g(i, j)$, and so we have $\ell_{ij} \geq f(i, j) + g(i, j)$. On the other hand, consider the corner-to-corner path that consists of a minimum-length path from $(0, 0)$ to (i, j), followed by a minimum-length path from (i, j) to (m, n). This path has length $f(i, j) + g(i, j)$, and so we have $\ell_{ij} \leq f(i, j) + g(i, j)$. It follows that $\ell_{ij} = f(i, j) + g(i, j)$. ■

(6.20) *Let k be any number in $\{0, \ldots, n\}$, and let q be an index that minimizes the quantity $f(q, k) + g(q, k)$. Then there is a corner-to-corner path of minimum length that passes through the node (q, k).*

Proof. Let ℓ^* denote the length of the shortest corner-to-corner path in G_{XY}. Now fix a value of $k \in \{0, \ldots, n\}$. The shortest corner-to-corner path must use *some* node in the k^{th} column of G_{XY}—let's suppose it is node (p, k)—and thus by (6.19)

$$\ell^* = f(p, k) + g(p, k) \geq \min_q f(q, k) + g(q, k).$$

Now consider the index q that achieves the minimum in the right-hand side of this expression; we have

$$\ell^* \geq f(q, k) + g(q, k).$$

By (6.19) again, the shortest corner-to-corner path using the node (q, k) has length $f(q, k) + g(q, k)$, and since ℓ^* is the minimum length of *any* corner-to-corner path, we have

$$\ell^* \leq f(q, k) + g(q, k).$$

It follows that $\ell^* = f(q, k) + g(q, k)$. Thus the shortest corner-to-corner path using the node (q, k) has length ℓ^*, and this proves (6.20). ■

Using (6.20) and our space-efficient algorithms to compute the *value* of the optimal alignment, we will proceed as follows. We divide G_{XY} along its center column and compute the value of $f(i, n/2)$ and $g(i, n/2)$ for each value of i, using our two space-efficient algorithms. We can then determine the minimum value of $f(i, n/2) + g(i, n/2)$, and conclude via (6.20) that there is a shortest corner-to-corner path passing through the node $(i, n/2)$. Given this, we can search for the shortest path recursively in the portion of G_{XY} between $(0, 0)$ and $(i, n/2)$ and in the portion between $(i, n/2)$ and (m, n). The crucial point is that we apply these recursive calls sequentially and reuse the working space from one call to the next. Thus, since we only work on one recursive call at a time, the total space usage is $O(m + n)$. The key question we have to resolve is whether the running time of this algorithm remains $O(mn)$.

In running the algorithm, we maintain a globally accessible list P which will hold nodes on the shortest corner-to-corner path as they are discovered. Initially, P is empty. P need only have $m + n$ entries, since no corner-to-corner path can use more than this many edges. We also use the following notation: $X[i : j]$, for $1 \leq i \leq j \leq m$, denotes the substring of X consisting of $x_i x_{i+1} \cdots x_j$; and we define $Y[i : j]$ analogously. We will assume for simplicity that n is a power of 2; this assumption makes the discussion much cleaner, although it can be easily avoided.

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1 : n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2 + 1 : n])
  Let q be the index minimizing f(q, n/2) + g(q, n/2)
  Add (q, n/2) to global list P
  Divide-and-Conquer-Alignment(X[1 : q],Y[1 : n/2])
  Divide-and-Conquer-Alignment(X[q + 1 : n],Y[n/2 + 1 : n])
  Return P
```

As an example of the first level of recursion, consider Figure 6.19. If the *minimizing index* q turns out to be 1, we get the two subproblems pictured.

Analyzing the Algorithm

The previous arguments already establish that the algorithm returns the correct answer and that it uses $O(m + n)$ space. Thus, we need only verify the following fact.

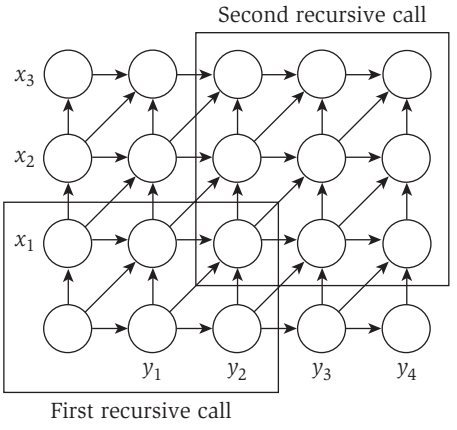


Figure 6.19 The first level of recurrence for the space-efficient `Divide-and-Conquer-Alignment`. The two boxed regions indicate the input to the two recursive cells.

(6.21) *The running time of* `Divide-and-Conquer-Alignment` *on strings of length m and n is $O(mn)$.*

Proof. Let $T(m, n)$ denote the maximum running time of the algorithm on strings of length m and n. The algorithm performs $O(mn)$ work to build up the arrays B and B'; it then runs recursively on strings of size q and $n/2$, and on strings of size $m - q$ and $n/2$. Thus, for some constant c, and some choice of index q, we have

$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$
$$T(m, 2) \leq cm$$
$$T(2, n) \leq cn.$$

This recurrence is more complex than the ones we've seen in our earlier applications of divide-and-conquer in Chapter 5. First of all, the running time is a function of two variables (m and n) rather than just one; also, the division into subproblems is not necessarily an "even split," but instead depends on the value q that is found through the earlier work done by the algorithm.

So how should we go about solving such a recurrence? One way is to try guessing the form by considering a special case of the recurrence, and then using partial substitution to fill out the details of this guess. Specifically, suppose that we were in a case in which $m = n$, and in which the split point q were exactly in the middle. In this (admittedly restrictive) special case, we could write the function $T(\cdot)$ in terms of the single variable n, set $q = n/2$ (since we're assuming a perfect bisection), and have

$$T(n) \leq 2T(n/2) + cn^2.$$

This is a useful expression, since it's something that we solved in our earlier discussion of recurrences at the outset of Chapter 5. Specifically, this recurrence implies $T(n) = O(n^2)$.

So when $m = n$ and we get an even split, the running time grows like the square of n. Motivated by this, we move back to the fully general recurrence for the problem at hand and guess that $T(m, n)$ grows like the product of m and n. Specifically, we'll guess that $T(m, n) \leq kmn$ for some constant k, and see if we can prove this by induction. To start with the base cases $m \leq 2$ and $n \leq 2$, we see that these hold as long as $k \geq c/2$. Now, assuming $T(m', n') \leq km'n'$ holds for pairs (m', n') with a smaller product, we have

$$\begin{aligned} T(m, n) &\leq cmn + T(q, n/2) + T(m - q, n/2) \\ &\leq cmn + kqn/2 + k(m - q)n/2 \\ &= cmn + kqn/2 + kmn/2 - kqn/2 \\ &= (c + k/2)mn. \end{aligned}$$

Thus the inductive step will work if we choose $k = 2c$, and this completes the proof. ■

6.8 Shortest Paths in a Graph

For the final three sections, we focus on the problem of finding shortest paths in a graph, together with some closely related issues.

The Problem

Let $G = (V, E)$ be a directed graph. Assume that each edge $(i, j) \in E$ has an associated *weight* c_{ij}. The weights can be used to model a number of different things; we will picture here the interpretation in which the weight c_{ij} represents a *cost* for going directly from node i to node j in the graph.

Earlier we discussed Dijkstra's Algorithm for finding shortest paths in graphs with positive edge costs. Here we consider the more complex problem in which we seek shortest paths when costs may be negative. Among the motivations for studying this problem, here are two that particularly stand out. First, negative costs turn out to be crucial for modeling a number of phenomena with shortest paths. For example, the nodes may represent agents in a financial setting, and c_{ij} represents the cost of a transaction in which we buy from agent i and then immediately sell to agent j. In this case, a path would represent a succession of transactions, and edges with negative costs would represent transactions that result in profits. Second, the algorithm that we develop for dealing with edges of negative cost turns out, in certain crucial ways, to be more flexible and *decentralized* than Dijkstra's Algorithm. As a consequence, it has important applications for the design of distributed

routing algorithms that determine the most efficient path in a communication network.

In this section and the next two, we will consider the following two related problems.

- Given a graph G with weights, as described above, decide if G has a negative cycle—that is, a directed cycle C such that

$$\sum_{ij \in C} c_{ij} < 0.$$

- If the graph has no negative cycles, find a path P from an origin node s to a destination node t with minimum total cost:

$$\sum_{ij \in P} c_{ij}$$

  should be as small as possible for any s-t path. This is generally called both the *Minimum-Cost Path Problem* and the *Shortest-Path Problem*.

In terms of our financial motivation above, a negative cycle corresponds to a profitable sequence of transactions that takes us back to our starting point: we buy from i_1, sell to i_2, buy from i_2, sell to i_3, and so forth, finally arriving back at i_1 with a net profit. Thus negative cycles in such a network can be viewed as good *arbitrage opportunities*.

It makes sense to consider the minimum-cost s-t path problem under the assumption that there are no negative cycles. As illustrated by Figure 6.20, if there is a negative cycle C, a path P_s from s to the cycle, and another path P_t from the cycle to t, then we can build an s-t path of arbitrarily negative cost: we first use P_s to get to the negative cycle C, then we go around C as many times as we want, and then we use P_t to get from C to the destination t.

Designing and Analyzing the Algorithm

A Few False Starts Let's begin by recalling Dijkstra's Algorithm for the Shortest-Path Problem when there are no negative costs. That method

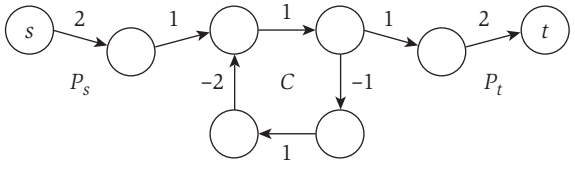
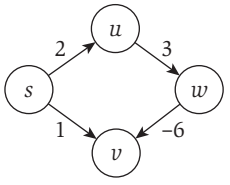
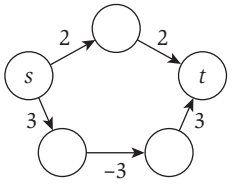


Figure 6.20 In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle C many times).



(a)



(b)

Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest s-t path.

computes a shortest path from the origin s to every other node v in the graph, essentially using a greedy algorithm. The basic idea is to maintain a set S with the property that the shortest path from s to each node in S is known. We start with $S = \{s\}$—since we know the shortest path from s to s has cost 0 when there are no negative edges—and we add elements greedily to this set S. As our first greedy step, we consider the minimum-cost edge leaving node s, that is, $\min_{i \in V} c_{si}$. Let v be a node on which this minimum is obtained. A key observation underlying Dijkstra's Algorithm is that the shortest path from s to v is the single-edge path $\{s, v\}$. Thus we can immediately add the node v to the set S. The path $\{s, v\}$ is clearly the shortest to v if there are no negative edge costs: any other path from s to v would have to start on an edge out of s that is at least as expensive as edge (s, v).

The above observation is no longer true if we can have negative edge costs. As suggested by the example in Figure 6.21(a), a path that starts on an expensive edge, but then compensates with subsequent edges of negative cost, can be cheaper than a path that starts on a cheap edge. This suggests that the Dijkstra-style greedy approach will not work here.

Another natural idea is to first modify the costs c_{ij} by adding some large constant M to each; that is, we let $c'_{ij} = c_{ij} + M$ for each edge $(i, j) \in E$. If the constant M is large enough, then all modified costs are nonnegative, and we can use Dijkstra's Algorithm to find the minimum-cost path subject to costs c'. However, this approach fails to find the correct minimum-cost paths with respect to the original costs c. The problem here is that changing the costs from c to c' changes the minimum-cost path. For example (as in Figure 6.21(b)), if a path P consisting of three edges is only slightly cheaper than another path P' that has two edges, then after the change in costs, P' will be cheaper, since we only add $2M$ to the cost of P' while adding $3M$ to the cost of P.

A Dynamic Programming Approach We will try to use dynamic programming to solve the problem of finding a shortest path from s to t when there are negative edge costs but no negative cycles. We could try an idea that has worked for us so far: subproblem i could be to find a shortest path using only the first i nodes. This idea does not immediately work, but it can be made to work with some effort. Here, however, we will discuss a simpler and more efficient solution, the *Bellman-Ford Algorithm*. The development of dynamic programming as a general algorithmic technique is often credited to the work of Bellman in the 1950's; and the Bellman-Ford Shortest-Path Algorithm was one of the first applications.

The dynamic programming solution we develop will be based on the following crucial observation.

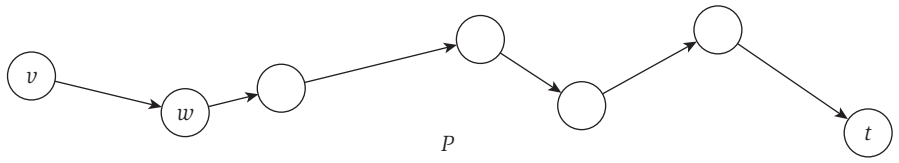


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

(6.22) *If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most $n - 1$ edges.*

Proof. Since every cycle has nonnegative cost, the shortest path P from s to t with the fewest number of edges does not repeat any vertex v. For if P did repeat a vertex v, we could remove the portion of P between consecutive visits to v, resulting in a path of no greater cost and fewer edges. ■

Let's use $\text{OPT}(i, v)$ to denote the minimum cost of a v-t path using at most i edges. By (6.22), our original problem is to compute $\text{OPT}(n-1, s)$. (We could instead design an algorithm whose subproblems correspond to the minimum cost of an s-v path using at most i edges. This would form a more natural parallel with Dijkstra's Algorithm, but it would not be as natural in the context of the routing protocols we discuss later.)

We now need a simple way to express $\text{OPT}(i, v)$ using smaller subproblems. We will see that the most natural approach involves the consideration of many different options; this is another example of the principle of "multi-way choices" that we saw in the algorithm for the Segmented Least Squares Problem.

Let's fix an optimal path P representing $\text{OPT}(i, v)$ as depicted in Figure 6.22.

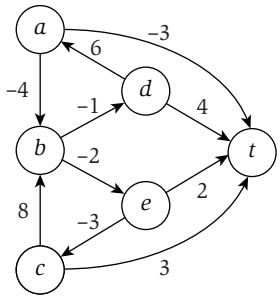
- If the path P uses at most $i - 1$ edges, then $\text{OPT}(i, v) = \text{OPT}(i - 1, v)$.
- If the path P uses i edges, and the first edge is (v, w), then $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i - 1, w)$.

This leads to the following recursive formula.

(6.23) *If $i > 0$ then*

$$\text{OPT}(i, v) = \min(\text{OPT}(i-1, v), \min_{w \in V}(\text{OPT}(i-1, w) + c_{vw})).$$

Using this recurrence, we get the following dynamic programming algorithm to compute the value $\text{OPT}(n-1, s)$.



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0 ... n − 1, V]
  Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For v ∈ V in any order
      Compute M[i, v] using the recurrence (6.23)
    Endfor
  Endfor
  Return M[n − 1, s]
```

The correctness of the method follows directly by induction from (6.23). We can bound the running time as follows. The table M has n^2 entries; and each entry can take $O(n)$ time to compute, as there are at most n nodes $w \in V$ we have to consider.

(6.24) *The* `Shortest-Path` *method correctly computes the minimum cost of an s-t path in any graph that has no negative cycles, and runs in $O(n^3)$ time.*

Given the table M containing the optimal values of the subproblems, the shortest path using at most i edges can be obtained in $O(in)$ time, by tracing back through smaller subproblems.

As an example, consider the graph in Figure 6.23(a), where the goal is to find a shortest path from each node to t. The table in Figure 6.23(b) shows the array M, with entries corresponding to the values $M[i, v]$ from the algorithm. Thus a single row in the table corresponds to the shortest path from a particular node to t, as we allow the path to use an increasing number of edges. For example, the shortest path from node d to t is updated four times, as it changes from d-t, to d-a-t, to d-a-b-e-t, and finally to d-a-b-e-c-t.

Extensions: Some Basic Improvements to the Algorithm

An Improved Running-Time Analysis We can actually provide a better running-time analysis for the case in which the graph G does not have too many edges. A directed graph with n nodes can have close to n^2 edges, since there could potentially be an edge between each pair of nodes, but many graphs are much sparser than this. When we work with a graph for which the number of edges m is significantly less than n^2, we've already seen in a number of cases earlier in the book that it can be useful to write the running-time in terms of both m and n; this way, we can quantify our speed-up on graphs with relatively fewer edges.

If we are a little more careful in the analysis of the method above, we can improve the running-time bound to $O(mn)$ without significantly changing the algorithm itself.

(6.25) *The* `Shortest-Path` *method can be implemented in* $O(mn)$ *time.*

Proof. Consider the computation of the array entry $M[i, v]$ according to the recurrence (6.23); we have

$$M[i, v] = \min(M[i-1, v], \min_{w \in V}(M[i-1, w] + c_{vw})).$$

We assumed it could take up to $O(n)$ time to compute this minimum, since there are n possible nodes w. But, of course, we need only compute this minimum over all nodes w for which v has an edge to w; let us use n_v to denote this number. Then it takes time $O(n_v)$ to compute the array entry $M[i, v]$. We have to compute an entry for every node v and every index $0 \leq i \leq n-1$, so this gives a running-time bound of

$$O\left(n \sum_{v \in V} n_v\right).$$

In Chapter 3, we performed exactly this kind of analysis for other graph algorithms, and used (3.9) from that chapter to bound the expression $\sum_{v \in V} n_v$ for undirected graphs. Here we are dealing with directed graphs, and n_v denotes the number of edges *leaving* v. In a sense, it is even easier to work out the value of $\sum_{v \in V} n_v$ for the directed case: each edge leaves exactly one of the nodes in V, and so each edge is counted exactly once by this expression. Thus we have $\sum_{v \in V} n_v = m$. Plugging this into our expression

$$O\left(n \sum_{v \in V} n_v\right)$$

for the running time, we get a running-time bound of $O(mn)$. ■

Improving the Memory Requirements We can also significantly improve the memory requirements with only a small change to the implementation. A common problem with many dynamic programming algorithms is the large space usage, arising from the M array that needs to be stored. In the Bellman-Ford Algorithm as written, this array has size n^2; however, we now show how to reduce this to $O(n)$. Rather than recording $M[i, v]$ for each value i, we will use and update a single value $M[v]$ for each node v, the length of the shortest path from v to t that we have found so far. We still run the algorithm for

iterations $i = 1, 2, \ldots, n-1$, but the role of i will now simply be as a counter; in each iteration, and for each node v, we perform the update

$$M[v] = \min(M[v], \min_{w \in V}(c_{vw} + M[w])).$$

We now observe the following fact.

(6.26) *Throughout the algorithm $M[v]$ is the length of some path from v to t, and after i rounds of updates the value $M[v]$ is no larger than the length of the shortest path from v to t using at most i edges.*

Given (6.26), we can then use (6.22) as before to show that we are done after $n-1$ iterations. Since we are only storing an M array that indexes over the nodes, this requires only $O(n)$ working memory.

Finding the Shortest Paths One issue to be concerned about is whether this space-efficient version of the algorithm saves enough information to recover the shortest paths themselves. In the case of the Sequence Alignment Problem in the previous section, we had to resort to a tricky divide-and-conquer method to recover the solution from a similar space-efficient implementation. Here, however, we will be able to recover the shortest paths much more easily.

To help with recovering the shortest paths, we will enhance the code by having each node v maintain the first node (after itself) on its path to the destination t; we will denote this first node by $first[v]$. To maintain $first[v]$, we update its value whenever the distance $M[v]$ is updated. In other words, whenever the value of $M[v]$ is reset to the minimum $\min_{w \in V}(c_{vw} + M[w])$, we set $first[v]$ to the node w that attains this minimum.

Now let P denote the directed "pointer graph" whose nodes are V, and whose edges are $\{(v, first[v])\}$. The main observation is the following.

(6.27) *If the pointer graph P contains a cycle C, then this cycle must have negative cost.*

Proof. Notice that if $first[v] = w$ at any time, then we must have $M[v] \geq c_{vw} + M[w]$. Indeed, the left- and right-hand sides are equal after the update that sets $first[v]$ equal to w; and since $M[w]$ may decrease, this equation may turn into an inequality.

Let $v_1, v_2, \ldots, v_k$ be the nodes along the cycle C in the pointer graph, and assume that (v_k, v_1) is the last edge to have been added. Now, consider the values right before this last update. At this time we have $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$ for all $i = 1, \ldots, k-1$, and we also have $M[v_k] > c_{v_k v_1} + M[v_1]$ since we are about to update $M[v_k]$ and change $first[v_k]$ to v_1. Adding all these inequalities, the $M[v_i]$ values cancel, and we get $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$: a negative cycle, as claimed. ■

Now note that if G has no negative cycles, then (6.27) implies that the pointer graph P will never have a cycle. For a node v, consider the path we get by following the edges in P, from v to $first[v] = v_1$, to $first[v_1] = v_2$, and so forth. Since the pointer graph has no cycles, and the sink t is the only node that has no outgoing edge, this path must lead to t. We claim that when the algorithm terminates, this is in fact a shortest path in G from v to t.

(6.28) *Suppose G has no negative cycles, and consider the pointer graph P at the termination of the algorithm. For each node v, the path in P from v to t is a shortest v-t path in G.*

Proof. Consider a node v and let $w = first[v]$. Since the algorithm terminated, we must have $M[v] = c_{vw} + M[w]$. The value $M[t] = 0$, and hence the length of the path traced out by the pointer graph is exactly $M[v]$, which we know is the shortest-path distance. ■

Note that in the more space-efficient version of Bellman-Ford, the path whose length is $M[v]$ after i iterations can have substantially more edges than i. For example, if the graph is a single path from s to t, and we perform updates in the reverse of the order the edges appear on the path, then we get the final shortest-path values in just one iteration. This does not always happen, so we cannot claim a worst-case running-time improvement, but it would be nice to be able to use this fact opportunistically to speed up the algorithm on instances where it does happen. In order to do this, we need a stopping signal in the algorithm—something that tells us it's safe to terminate before iteration $n - 1$ is reached.

Such a stopping signal is a simple consequence of the following observation: If we ever execute a complete iteration i in which *no* $M[v]$ value changes, then no $M[v]$ value will ever change again, since future iterations will begin with exactly the same set of array entries. Thus it is safe to stop the algorithm. Note that it is not enough for a *particular* $M[v]$ value to remain the same; in order to safely terminate, we need for all these values to remain the same for a single iteration.

6.9 Shortest Paths and Distance Vector Protocols

One important application of the Shortest-Path Problem is for routers in a communication network to determine the most efficient path to a destination. We represent the network using a graph in which the nodes correspond to routers, and there is an edge between v and w if the two routers are connected by a direct communication link. We define a cost c_{vw} representing the delay on the link (v, w); the Shortest-Path Problem with these costs is to determine the path with minimum delay from a source node s to a destination t. Delays are

naturally nonnegative, so one could use Dijkstra's Algorithm to compute the shortest path. However, Dijkstra's shortest-path computation requires global knowledge of the network: it needs to maintain a set S of nodes for which shortest paths have been determined, and make a global decision about which node to add next to S. While routers can be made to run a protocol in the background that gathers enough global information to implement such an algorithm, it is often cleaner and more flexible to use algorithms that require only local knowledge of neighboring nodes.

If we think about it, the Bellman-Ford Algorithm discussed in the previous section has just such a "local" property. Suppose we let each node v maintain its value $M[v]$; then to update this value, v needs only obtain the value $M[w]$ from each neighbor w, and compute

$$\min_{w \in V}(c_{vw} + M[w])$$

based on the information obtained.

We now discuss an improvement to the Bellman-Ford Algorithm that makes it better suited for routers and, at the same time, a faster algorithm in practice. Our current implementation of the Bellman-Ford Algorithm can be thought of as a *pull-based* algorithm. In each iteration i, each node v has to contact each neighbor w, and "pull" the new value $M[w]$ from it. If a node w has not changed its value, then there is no need for v to get the value again; however, v has no way of knowing this fact, and so it must execute the pull anyway.

This wastefulness suggests a symmetric *push-based* implementation, where values are only transmitted when they change. Specifically, each node w whose distance value $M[w]$ changes in an iteration informs all its neighbors of the new value in the next iteration; this allows them to update their values accordingly. If $M[w]$ has not changed, then the neighbors of w already have the current value, and there is no need to "push" it to them again. This leads to savings in the running time, as not all values need to be pushed in each iteration. We also may terminate the algorithm early, if no value changes during an iteration. Here is a concrete description of the push-based implementation.

```
Push-Based-Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[V]
  Initialize M[t] = 0 and M[v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For w ∈ V in any order
      If M[w] has been updated in the previous iteration then
```

```
      For all edges (v, w) in any order
        M[v] = min(M[v], c_vw + M[w])
        If this changes the value of M[v], then first[v] = w
      Endfor
  Endfor
  If no value changed in this iteration, then end the algorithm
Endfor
Return M[s]
```

In this algorithm, nodes are sent updates of their neighbors' distance values in rounds, and each node sends out an update in each iteration in which it has changed. However, if the nodes correspond to routers in a network, then we do not expect everything to run in lockstep like this; some routers may report updates much more quickly than others, and a router with an update to report may sometimes experience a delay before contacting its neighbors. Thus the routers will end up executing an *asynchronous* version of the algorithm: each time a node w experiences an update to its $M[w]$ value, it becomes "active" and eventually notifies its neighbors of the new value. If we were to watch the behavior of all routers interleaved, it would look as follows.

```
Asynchronous-Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[V]
  Initialize M[t] = 0 and M[v] = ∞ for all other v ∈ V
  Declare t to be active and all other nodes inactive
  While there exists an active node
    Choose an active node w
      For all edges (v, w) in any order
        M[v] = min(M[v], c_vw + M[w])
        If this changes the value of M[v], then
          first[v] = w
          v becomes active
      Endfor
      w becomes inactive
  EndWhile
```

One can show that even this version of the algorithm, with essentially no coordination in the ordering of updates, will converge to the correct values of the shortest-path distances to t, assuming only that each time a node becomes active, it eventually contacts its neighbors.

The algorithm we have developed here uses a single destination t, and all nodes $v \in V$ compute their shortest path to t. More generally, we are

presumably interested in finding distances and shortest paths between all pairs of nodes in a graph. To obtain such distances, we effectively use n separate computations, one for each destination. Such an algorithm is referred to as a *distance vector protocol*, since each node maintains a vector of distances to every other node in the network.

Problems with the Distance Vector Protocol

One of the major problems with the distributed implementation of Bellman-Ford on routers (the protocol we have been discussing above) is that it's derived from an initial dynamic programming algorithm that assumes edge costs will remain constant during the execution of the algorithm. Thus far we've been designing algorithms with the tacit understanding that a program executing the algorithm will be running on a single computer (or a centrally managed set of computers), processing some specified input. In this context, it's a rather benign assumption to require that the input not change while the program is actually running. Once we start thinking about routers in a network, however, this assumption becomes troublesome. Edge costs may change for all sorts of reasons: links can become congested and experience slow-downs; or a link (v, w) may even fail, in which case the cost c_{vw} effectively increases to ∞.

Here's an indication of what can go wrong with our shortest-path algorithm when this happens. If an edge (v, w) is deleted (say the link goes down), it is natural for node v to react as follows: it should check whether its shortest path to some node t used the edge (v, w), and, if so, it should increase the distance using other neighbors. Notice that this increase in distance from v can now trigger increases at v's neighbors, if they were relying on a path through v, and these changes can cascade through the network. Consider the extremely simple example in Figure 6.24, in which the original graph has three edges (s, v), (v, s) and (v, t), each of cost 1.

Now suppose the edge (v, t) in Figure 6.24 is deleted. How does node v react? Unfortunately, it does not have a global map of the network; it only knows the shortest-path distances of each of its neighbors to t. Thus it does

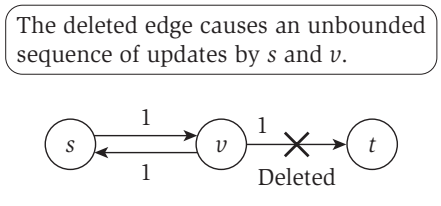


Figure 6.24 When the edge (v, t) is deleted, the distributed Bellman-Ford Algorithm will begin "counting to infinity."

not know that the deletion of (v, t) has eliminated all paths from s to t. Instead, it sees that $M[s] = 2$, and so it updates $M[v] = c_{vs} + M[s] = 3$, assuming that it will use its cost-1 edge to s, followed by the supposed cost-2 path from s to t. Seeing this change, node s will update $M[s] = c_{sv} + M[v] = 4$, based on its cost-1 edge to v, followed by the supposed cost-3 path from v to t. Nodes s and v will continue updating their distance to t until one of them finds an alternate route; in the case, as here, that the network is truly disconnected, these updates will continue indefinitely—a behavior known as the problem of *counting to infinity*.

To avoid this problem and related difficulties arising from the limited amount of information available to nodes in the Bellman-Ford Algorithm, the designers of network routing schemes have tended to move from distance vector protocols to more expressive *path vector protocols*, in which each node stores not just the distance and first hop of their path to a destination, but some representation of the entire path. Given knowledge of the paths, nodes can avoid updating their paths to use edges they know to be deleted; at the same time, they require significantly more storage to keep track of the full paths. In the history of the Internet, there has been a shift from distance vector protocols to path vector protocols; currently, the path vector approach is used in the *Border Gateway Protocol* (BGP) in the Internet core.

* 6.10 Negative Cycles in a Graph

So far in our consideration of the Bellman-Ford Algorithm, we have assumed that the underlying graph has negative edge costs but no negative cycles. We now consider the more general case of a graph that may contain negative cycles.

The Problem

There are two natural questions we will consider.

- How do we decide if a graph contains a negative cycle?
- How do we actually find a negative cycle in a graph that contains one?

The algorithm developed for finding negative cycles will also lead to an improved practical implementation of the Bellman-Ford Algorithm from the previous sections.

It turns out that the ideas we've seen so far will allow us to find negative cycles that have a path reaching a sink t. Before we develop the details of this, let's compare the problem of finding a negative cycle that can reach a given t with the seemingly more natural problem of finding a negative cycle *anywhere* in the graph, regardless of its position related to a sink. It turns out that if we

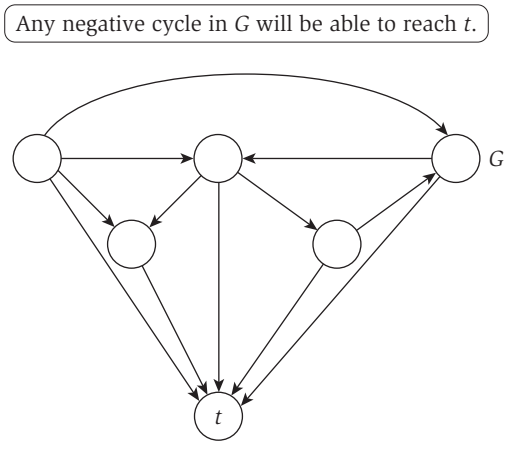


Figure 6.25 The augmented graph.

develop a solution to the first problem, we'll be able to obtain a solution to the second problem as well, in the following way. Suppose we start with a graph G, add a new node t to it, and connect each other node v in the graph to node t via an edge of cost 0, as shown in Figure 6.25. Let us call the new "augmented graph" G'.

(6.29) *The augmented graph G' has a negative cycle C such that there is a path from C to the sink t if and only if the original graph has a negative cycle.*

Proof. Assume G has a negative cycle. Then this cycle C clearly has an edge to t in G', since all nodes have an edge to t.

Now suppose G' has a negative cycle with a path to t. Since no edge leaves t in G', this cycle cannot contain t. Since G' is the same as G aside from the node t, it follows that this cycle is also a negative cycle of G. ■

So it is really enough to solve the problem of deciding whether G has a negative cycle that has a path to a given sink node t, and we do this now.

Designing and Analyzing the Algorithm

To get started thinking about the algorithm, we begin by adopting the original version of the Bellman-Ford Algorithm, which was less efficient in its use of space. We first extend the definitions of $\text{OPT}(i, v)$ from the Bellman-Ford Algorithm, defining them for values $i \geq n$. With the presence of a negative cycle in the graph, (6.22) no longer applies, and indeed the shortest path may

get shorter and shorter as we go around a negative cycle. In fact, for any node v on a negative cycle that has a path to t, we have the following.

(6.30) *If node v can reach node t and is contained in a negative cycle, then*

$$\lim_{i \to \infty} \text{OPT}(i, v) = -\infty.$$

If the graph has no negative cycles, then (6.22) implies following statement.

(6.31) *If there are no negative cycles in G, then $\text{OPT}(i, v) = \text{OPT}(n-1, v)$ for all nodes v and all $i \geq n$.*

But for how large an i do we have to compute the values $\text{OPT}(i, v)$ before concluding that the graph has no negative cycles? For example, a node v may satisfy the equation $\text{OPT}(n, v) = \text{OPT}(n-1, v)$, and yet still lie on a negative cycle. (Do you see why?) However, it turns out that we will be in good shape if this equation holds for all nodes.

(6.32) *There is no negative cycle with a path to t if and only if $\text{OPT}(n, v) = \text{OPT}(n-1, v)$ for all nodes v.*

Proof. Statement (6.31) has already proved the forward direction. For the other direction, we use an argument employed earlier for reasoning about when it's safe to stop the Bellman-Ford Algorithm early. Specifically, suppose $\text{OPT}(n, v) = \text{OPT}(n-1, v)$ for all nodes v. The values of $\text{OPT}(n+1, v)$ can be computed from $\text{OPT}(n, v)$; but all these values are the same as the corresponding $\text{OPT}(n-1, v)$. It follows that we will have $\text{OPT}(n+1, v) = \text{OPT}(n-1, v)$. Extending this reasoning to future iterations, we see that none of the values will ever change again, that is, $\text{OPT}(i, v) = \text{OPT}(n-1, v)$ for all nodes v and all $i \geq n$. Thus there cannot be a negative cycle C that has a path to t; for any node w on this cycle C, (6.30) implies that the values $\text{OPT}(i, w)$ would have to become arbitrarily negative as i increased. ■

Statement (6.32) gives an $O(mn)$ method to decide if G has a negative cycle that can reach t. We compute values of $\text{OPT}(i, v)$ for nodes of G and for values of i up to n. By (6.32), there is no negative cycle if and only if there is some value of $i \leq n$ at which $\text{OPT}(i, v) = \text{OPT}(i-1, v)$ for all nodes v.

So far we have determined whether or not the graph has a negative cycle with a path from the cycle to t, but we have not actually found the cycle. To find a negative cycle, we consider a node v such that $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$: for this node, a path P from v to t of cost $\text{OPT}(n, v)$ must use *exactly* n edges. We find this minimum-cost path P from v to t by tracing back through the subproblems. As in our proof of (6.22), a simple path can only have $n-1$

edges, so P must contain a cycle C. We claim that this cycle C has negative cost.

(6.33) *If G has n nodes and $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$, then a path P from v to t of cost $\text{OPT}(n, v)$ contains a cycle C, and C has negative cost.*

Proof. First observe that the path P must have n edges, as $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$, and so every path using $n-1$ edges has cost greater than that of the path P. In a graph with n nodes, a path consisting of n edges must repeat a node somewhere; let w be a node that occurs on P more than once. Let C be the cycle on P between two consecutive occurrences of node w. If C were not a negative cycle, then deleting C from P would give us a v-t path with fewer than n edges and no greater cost. This contradicts our assumption that $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$, and hence C must be a negative cycle. ■

(6.34) *The algorithm above finds a negative cycle in G, if such a cycle exists, and runs in $O(mn)$ time.*

Extensions: Improved Shortest Paths and Negative Cycle Detection Algorithms

At the end of Section 6.8 we discussed a space-efficient implementation of the Bellman-Ford algorithm for graphs with no negative cycles. Here we implement the detection of negative cycles in a comparably space-efficient way. In addition to the savings in space, this will also lead to a considerable speedup in practice even for graphs with no negative cycles. The implementation will be based on the same pointer graph P derived from the "first edges" $(v, \mathit{first}[v])$ that we used for the space-efficient implementation in Section 6.8. By (6.27), we know that if the pointer graph ever has a cycle, then the cycle has negative cost, and we are done. But if G has a negative cycle, does this guarantee that the pointer graph will ever have a cycle? Furthermore, how much extra computation time do we need for periodically checking whether P has a cycle?

Ideally, we would like to determine whether a cycle is created in the pointer graph P every time we add a new edge (v, w) with $\mathit{first}[v] = w$. An additional advantage of such "instant" cycle detection will be that we will not have to wait for n iterations to see that the graph has a negative cycle: We can terminate as soon as a negative cycle is found. Earlier we saw that if a graph G has no negative cycles, the algorithm can be stopped early if in some iteration the shortest path values $M[v]$ remain the same *for all nodes* v. Instant negative cycle detection will be an analogous early termination rule for graphs that have negative cycles.

Consider a new edge (v, w), with $first[v] = w$, that is added to the pointer graph P. Before we add (v, w) the pointer graph has no cycles, so it consists of paths from each node v to the sink t. The most natural way to check whether adding edge (v, w) creates a cycle in P is to follow the current path from w to the terminal t in time proportional to the length of this path. If we encounter v along this path, then a cycle has been formed, and hence, by (6.27), the graph has a negative cycle. Consider Figure 6.26, for example, where in both (a) and (b) the pointer $first[v]$ is being updated from u to w; in (a), this does not result in a (negative) cycle, but in (b) it does. However, if we trace out the sequence of pointers from v like this, then we could spend as much as $O(n)$ time following the path to t and still not find a cycle. We now discuss a method that does not require an $O(n)$ blow-up in the running time.

We know that before the new edge (v, w) was added, the pointer graph was a directed tree. Another way to test whether the addition of (v, w) creates a cycle is to consider all nodes in the subtree directed toward v. If w is in this subtree, then (v, w) forms a cycle; otherwise it does not. (Again, consider the two sample cases in Figure 6.26.) To be able to find all nodes in the subtree directed toward v, we need to have each node v maintain a list of all other nodes whose selected edges point to v. Given these pointers, we can find the subtree in time proportional to the size of the subtree pointing to v, at most $O(n)$ as before. However, here we will be able to make additional use of the work done. Notice that the current distance value $M[x]$ for all nodes x in the subtree was derived from node v's old value. We have just updated v's distance, and hence we know that the distance values of all these nodes will be updated again. We'll mark each of these nodes x as "dormant," delete the

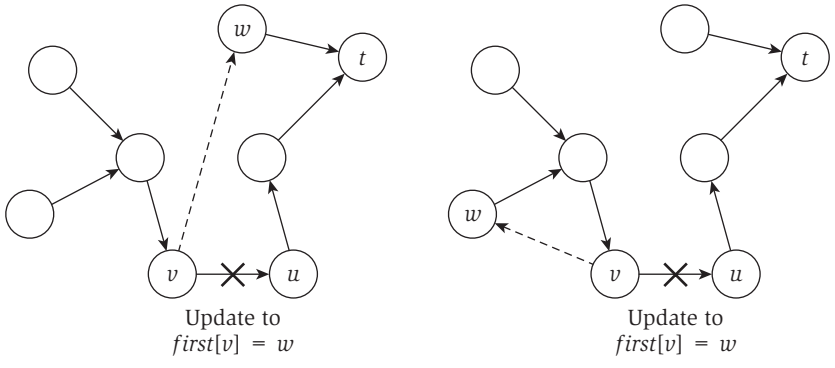


Figure 6.26 Changing the pointer graph P when $first[v]$ is updated from u to w. In (b), this creates a (negative) cycle, whereas in (a) it does not.

edge $(x, first[x])$ from the pointer graph, and not use x for future updates until its distance value changes.

This can save a lot of future work in updates, but what is the effect on the worst-case running time? We can spend as much as $O(n)$ extra time marking nodes dormant after every update in distances. However, a node can be marked dormant only if a pointer had been defined for it at some point in the past, so the time spent on marking nodes dormant is at most as much as the time the algorithm spends updating distances.

Now consider the time the algorithm spends on operations other than marking nodes dormant. Recall that the algorithm is divided into iterations, where iteration $i+1$ processes nodes whose distance has been updated in iteration i. For the original version of the algorithm, we showed in (6.26) that after i iterations, the value $M[v]$ is no larger than the value of the shortest path from v to t using at most i edges. However, with many nodes dormant in each iteration, this may not be true anymore. For example, if the shortest path from v to t using at most i edges starts on edge $e = (v, w)$, and w is dormant in this iteration, then we may not update the distance value $M[v]$, and so it stays at a value higher than the length of the path through the edge (v, w). This seems like a problem—however, in this case, the path through edge (v, w) is not actually the shortest path, so $M[v]$ will have a chance to get updated later to an even smaller value.

So instead of the simpler property that held for $M[v]$ in the original versions of the algorithm, we now have the the following claim.

(6.35) *Throughout the algorithm $M[v]$ is the length of some simple path from v to t; the path has at least i edges if the distance value $M[v]$ is updated in iteration i; and after i iterations, the value $M[v]$ is the length of the shortest path for all nodes v where there is a shortest v-t path using at most i edges.*

Proof. The *first* pointers maintain a tree of paths to t, which implies that all paths used to update the distance values are simple. The fact that updates in iteration i are caused by paths with at least i edges is easy to show by induction on i. Similarly, we use induction to show that after iteration i the value $M[v]$ is the distance on all nodes v where the shortest path from v to t uses at most i edges. Note that nodes v where $M[v]$ is the actual shortest-path distance cannot be dormant, as the value $M[v]$ will be updated in the next iteration for all dormant nodes. ■

Using this claim, we can see that the worst-case running time of the algorithm is still bounded by $O(mn)$: Ignoring the time spent on marking nodes dormant, each iteration is implemented in $O(m)$ time, and there can be at most $n-1$ iterations that update values in the array M without finding

a negative cycle, as simple paths can have at most $n-1$ edges. Finally, the time spent marking nodes dormant is bounded by the time spent on updates. We summarize the discussion with the following claim about the worst-case performance of the algorithm. In fact, as mentioned above, this new version is in practice the fastest implementation of the algorithm even for graphs that do not have negative cycles, or even negative-cost edges.

(6.36) *The improved algorithm outlined above finds a negative cycle in G if such a cycle exists. It terminates immediately if the pointer graph P of first$[v]$ pointers contains a cycle C, or if there is an iteration in which no update occurs to any distance value $M[v]$. The algorithm uses $O(n)$ space, has at most n iterations, and runs in $O(mn)$ time in the worst case.*

Solved Exercises

Solved Exercise 1

Suppose you are managing the construction of billboards on the Stephen Daedalus Memorial Highway, a heavily traveled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$, each in the interval $[0, M]$ (specifying their position along the highway, measured in miles from its western end). If you place a billboard at location x_i, you receive a revenue of $r_i > 0$.

Regulations imposed by the county's Highway Department require that no two of the billboards be within less than or equal to 5 miles of each other. You'd like to place billboards at a subset of the sites so as to maximize your total revenue, subject to this restriction.

Example. Suppose $M = 20$, $n = 4$,

$$\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\},$$

and

$$\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}.$$

Then the optimal solution would be to place billboards at x_1 and x_3, for a total revenue of 10.

Give an algorithm that takes an instance of this problem as input and returns the maximum total revenue that can be obtained from any valid subset of sites. The running time of the algorithm should be polynomial in n.

Solution We can naturally apply dynamic programming to this problem if we reason as follows. Consider an optimal solution for a given input instance; in this solution, we either place a billboard at site x_n or not. If we don't, the optimal solution on sites $x_1, \ldots, x_n$ is really the same as the optimal solution

on sites $x_1, \ldots, x_{n-1}$; if we do, then we should eliminate x_n and all other sites that are within 5 miles of it, and find an optimal solution on what's left. The same reasoning applies when we're looking at the problem defined by just the first j sites, $x_1, \ldots, x_j$: we either include x_j in the optimal solution or we don't, with the same consequences.

Let's define some notation to help express this. For a site x_j, we let $e(j)$ denote the easternmost site x_i that is more than 5 miles from x_j. Since sites are numbered west to east, this means that the sites $x_1, x_2, \ldots, x_{e(j)}$ are still valid options once we've chosen to place a billboard at x_j, but the sites $x_{e(j)+1}, \ldots, x_{j-1}$ are not.

Now, our reasoning above justifies the following recurrence. If we let $\text{OPT}(j)$ denote the revenue from the optimal subset of sites among $x_1, \ldots, x_j$, then we have

$$\text{OPT}(j) = \max(r_j + \text{OPT}(e(j)), \text{OPT}(j-1)).$$

We now have most of the ingredients we need for a dynamic programming algorithm. First, we have a set of n subproblems, consisting of the first j sites for $j = 0, 1, 2, \ldots, n$. Second, we have a recurrence that lets us build up the solutions to subproblems, given by $\text{OPT}(j) = \max(r_j + \text{OPT}(e(j)), \text{OPT}(j-1))$.

To turn this into an algorithm, we just need to define an array M that will store the OPT values and throw a loop around the recurrence that builds up the values $M[j]$ in order of increasing j.

```
Initialize M[0] = 0 and M[1] = r_1
For j = 2, 3, ..., n:
  Compute M[j] using the recurrence
Endfor
Return M[n]
```

As with all the dynamic programming algorithms we've seen in this chapter, an optimal *set* of billboards can be found by tracing back through the values in array M.

Given the values $e(j)$ for all j, the running time of the algorithm is $O(n)$, since each iteration of the loop takes constant time. We can also compute all $e(j)$ values in $O(n)$ time as follows. For each site location x_i, we define $x'_i = x_i - 5$. We then merge the sorted list $x_1, \ldots, x_n$ with the sorted list $x'_1, \ldots, x'_n$ in linear time, as we saw how to do in Chapter 2. We now scan through this merged list; when we get to the entry x'_j, we know that anything from this point onward to x_j cannot be chosen together with x_j (since it's within 5 miles), and so we

simply define $e(j)$ to be the largest value of i for which we've seen x_i in our scan.

Here's a final observation on this problem. Clearly, the solution looks very much like that of the Weighted Interval Scheduling Problem, and there's a fundamental reason for that. In fact, our billboard placement problem can be directly encoded as an instance of Weighted Interval Scheduling, as follows. Suppose that for each site x_i, we define an interval with endpoints $[x_i - 5, x_i]$ and weight r_i. Then, given any nonoverlapping set of intervals, the corresponding set of sites has the property that no two lie within 5 miles of each other. Conversely, given any such set of sites (no two within 5 miles), the intervals associated with them will be nonoverlapping. Thus the collections of nonoverlapping intervals correspond precisely to the set of valid billboard placements, and so dropping the set of intervals we've just defined (with their weights) into an algorithm for Weighted Interval Scheduling will yield the desired solution.

Solved Exercise 2

Through some friends of friends, you end up on a consulting visit to the cutting-edge biotech firm Clones 'R' Us (CRU). At first you're not sure how your algorithmic background will be of any help to them, but you soon find yourself called upon to help two identical-looking software engineers tackle a perplexing problem.

The problem they are currently working on is based on the *concatenation* of sequences of genetic material. If X and Y are each strings over a fixed alphabet $\mathcal{S}$, then XY denotes the string obtained by *concatenating* them—writing X followed by Y. CRU has identified a *target sequence* A of genetic material, consisting of m symbols, and they want to produce a sequence that is as similar to A as possible. For this purpose, they have a library $\mathcal{L}$ consisting of k (shorter) sequences, each of length at most n. They can cheaply produce any sequence consisting of copies of the strings in $\mathcal{L}$ concatenated together (with repetitions allowed).

Thus we say that a *concatenation* over $\mathcal{L}$ is any sequence of the form $B_1 B_2 \cdots B_\ell$, where each B_i belongs the set $\mathcal{L}$. (Again, repetitions are allowed, so B_i and B_j could be the same string in $\mathcal{L}$, for different values of i and j.) The problem is to find a concatenation over $\{B_i\}$ for which the sequence alignment cost is as small as possible. (For the purpose of computing the sequence alignment cost, you may assume that you are given a gap cost δ and a mismatch cost α_{pq} for each pair $p, q \in \mathcal{S}$.)

Give a polynomial-time algorithm for this problem.

Solution This problem is vaguely reminiscent of Segmented Least Squares: we have a long sequence of "data" (the string A) that we want to "fit" with shorter segments (the strings in $\mathcal{L}$).

If we wanted to pursue this analogy, we could search for a solution as follows. Let $B = B_1 B_2 \cdots B_\ell$ denote a concatenation over $\mathcal{L}$ that aligns as well as possible with the given string A. (That is, B is an optimal solution to the input instance.) Consider an optimal alignment M of A with B, let t be the first position in A that is matched with some symbol in B_ℓ, and let A_ℓ denote the substring of A from position t to the end. (See Figure 6.27 for an illustration of this with $\ell = 3$.) Now, the point is that in this optimal alignment M, the substring A_ℓ is optimally aligned with B_ℓ; indeed, if there were a way to better align A_ℓ with B_ℓ, we could substitute it for the portion of M that aligns A_ℓ with B_ℓ and obtain a better overall alignment of A with B.

This tells us that we can look at the optimal solution as follows. There's some final piece of A_ℓ that is aligned with one of the strings in $\mathcal{L}$, and for this piece all we're doing is finding the string in $\mathcal{L}$ that aligns with it as well as possible. Having found this optimal alignment for A_ℓ, we can break it off and continue to find the optimal solution for the remainder of A.

Thinking about the problem this way doesn't tell us exactly how to proceed—we don't know how long A_ℓ is supposed to be, or which string in $\mathcal{L}$ it should be aligned with. But this is the kind of thing we can search over in a dynamic programming algorithm. Essentially, we're in about the same spot we were in with the Segmented Least Squares Problem: there we knew that we had to break off some final subsequence of the input points, fit them as well as possible with one line, and then iterate on the remaining input points.

So let's set up things to make the search for A_ℓ possible. First, let $A[x : y]$ denote the substring of A consisting of its symbols from position x to position y, inclusive. Let $c(x, y)$ denote the cost of the optimal alignment of $A[x : y]$ with any string in $\mathcal{L}$. (That is, we search over each string in $\mathcal{L}$ and find the one that

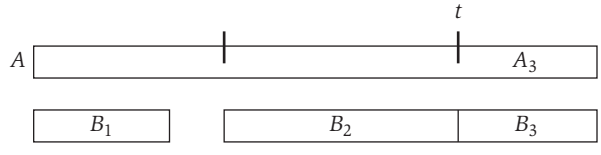


Figure 6.27 In the optimal concatentation of strings to align with A, there is a final string (B_3 in the figure) that aligns with a substring of A (A_3 in the figure) that extends from some position t to the end.

aligns best with $A[x : y]$.) Let $\text{OPT}(j)$ denote the alignment cost of the optimal solution on the string $A[1 : j]$.

The argument above says that an optimal solution on $A[1 : j]$ consists of identifying a final "segment boundary" $t < j$, finding the optimal alignment of $A[t : j]$ with a single string in $\mathcal{L}$, and iterating on $A[1 : t - 1]$. The cost of this alignment of $A[t : j]$ is just $c(t, j)$, and the cost of aligning with what's left is just $\text{OPT}(t - 1)$. This suggests that our subproblems fit together very nicely, and it justifies the following recurrence.

(6.37) $\text{OPT}(j) = \min_{t<j} c(t, j) + \text{OPT}(t - 1)$ *for* $j \geq 1$, *and* $\text{OPT}(0) = 0$.

The full algorithm consists of first computing the quantities $c(t, j)$, for $t < j$, and then building up the values $\text{OPT}(j)$ in order of increasing j. We hold these values in an array M.

```
Set M[0] = 0
For all pairs 1 ≤ t ≤ j ≤ m
  Compute the cost c(t, j) as follows:
  For each string B ∈ ℒ
    Compute the optimal alignment of B with A[t : j]
  Endfor
  Choose the B that achieves the best alignment, and use
    this alignment cost as c(t, j)
Endfor
For j = 1, 2, . . . , n
  Use the recurrence (6.37) to compute M[j]
Endfor
Return M[n]
```

As usual, we can get a concatentation that achieves it by tracing back over the array of OPT values.

Let's consider the running time of this algorithm. First, there are $O(m^2)$ values $c(t, j)$ that need to be computed. For each, we try each string of the k strings $B \in \mathcal{L}$, and compute the optimal alignment of B with $A[t : j]$ in time $O(n(j - t)) = O(mn)$. Thus the total time to compute all $c(t, j)$ values is $O(km^3n)$.

This dominates the time to compute all OPT values: Computing $\text{OPT}(j)$ uses the recurrence in (6.37), and this takes $O(m)$ time to compute the minimum. Summing this over all choices of $j = 1, 2, \ldots, m$, we get $O(m^2)$ time for this portion of the algorithm.

Exercises

1. Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

Consider, for example, the five-node path drawn in Figure 6.28. The *weights* are the numbers drawn inside the nodes.

The goal in this question is to solve the following problem:

Find an independent set in a path G whose total weight is as large as possible.

(a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
  Start with S equal to the empty set
  While some node remains in G
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
  Endwhile
  Return S
```

(b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

```
Let S_1 be the set of all v_i where i is an odd number
Let S_2 be the set of all v_i where i is an even number
(Note that S_1 and S_2 are both independent sets)
Determine which of S_1 or S_2 has greater total weight,
  and return this one
```

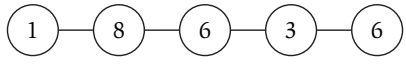


Figure 6.28 A paths with weights on the nodes. The maximum weight of an independent set is 14.

(c) Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

2. Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g., setting up a Web site for a class at the local elementary school) and those that are *high-stress* (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i-1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem. Given sets of values $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For iterations i = 1 to n
  If h_{i+1} > ℓ_i + ℓ_{i+1} then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i+1"
    Continue with iteration i+2
  Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i+1
  Endif
End
```

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an efficient algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$ and returns the *value* of an optimal plan.

3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 6.29 for an example).

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

```
Set w = v_1
Set L = 0
```

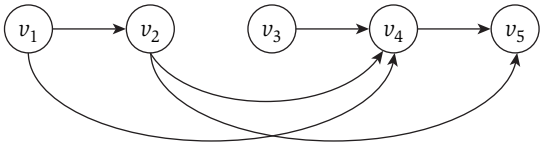


Figure 6.29 The correct answer for this ordered graph is 3: The longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

```
While there is an edge out of the node w
  Choose the edge (w, v_j)
     for which j is as small as possible
  Set w = v_j
  Increase L by 1
end while
Return L as the length of the longest path
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n. (Again, the *length* of a path is the number of edges in the path.)

4. Suppose you're running a lightweight consulting business—just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i, you'll incur an *operating cost* of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month i, and then out of the other city in month $i + 1$, then you incur a fixed *moving cost* of M to switch base offices.

Given a sequence of n months, a *plan* is a sequence of n locations—each one equal to either NY or SF—such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

The problem. Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

Then the plan of minimum cost would be the sequence of locations

$$[NY, NY, SF, SF],$$

with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For i = 1 to n
  If N_i < S_i then
    Output "NY in Month i"
  Else
    Output "SF in Month i"
End
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an example of an instance in which every optimal plan must move (i.e., change locations) at least three times.

Provide a brief explanation, saying why your example has this property.

(c) Give an efficient algorithm that takes values for n, M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the *cost* of an optimal plan.

5. As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the *word segmentation problem*—inferring likely boundaries between consecutive words in the

text. If English were written without spaces, the analogous problem would consist of taking a string like "meetateight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight," or "meet ate ight," or any of a huge number of even less plausible alternatives). How could we automate this process?

A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative "quality" of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters $x = x_1x_2 \cdots x_k$, will return a number $quality(x)$. This number can be either positive or negative; larger numbers correspond to more plausible English words. (So $quality($"*me*"$)$ would be positive, while $quality($"*ght*"$)$ would be negative.)

Given a long string of letters $y = y_1y_2 \cdots y_n$, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The *total quality* of a segmentation is determined by adding up the qualities of each of its blocks. (So we'd get the right answer above provided that $quality($"*meet*"$) + quality($"*at*"$) + quality($"*eight*"$)$ was greater than the total quality of any other segmentation of the string.)

Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. (You can treat a single call to the black box computing $quality(x)$ as a single computational step.)

(*A final note, not necessary for solving the problem:* To achieve better performance, word segmentation software in practice works with a more complex formulation of the problem—for example, incorporating the notion that solutions should not only be reasonable at the word level, but also form coherent phrases and sentences. If we consider the example "theyouthevent," there are at least three valid ways to segment this into common English words, but one constitutes a much more coherent phrase than the other two. If we think of this in the terminology of formal languages, this broader problem is like searching for a segmentation that also can be parsed well according to a grammar for the underlying language. But even with these additional criteria and constraints, dynamic programming approaches lie at the heart of a number of successful segmentation systems.)

6. In a word processor, the goal of "pretty-printing" is to take text with a ragged right margin, like this,

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
```

```
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

and turn it into text whose right margin is as "even" as possible, like this.

```
Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be "even." So suppose our text consists of a sequence of *words*, $W = \{w_1, w_2, \ldots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L. We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of a partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word except the last; and so if $w_j, w_{j+1}, \ldots, w_k$ are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \leq L.$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line—that is, the number of spaces left at the right margin.

Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the *squares* of the slacks of all lines (including the last line) is minimized.

7. As a solved exercise in Chapter 5, we gave an algorithm with $O(n \log n)$ running time for the following problem. We're looking at the price of a given stock over n consecutive days, numbered $i = 1, 2, \ldots, n$. For each day i, we have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) We'd like to know: How should we choose a day i on which to buy the stock and a later day $j > i$ on which to sell it, if we want to maximize the profit per

share, $p(j) - p(i)$? (If there is no way to make money during the n days, we should conclude this instead.)

In the solved exercise, we showed how to find the optimal pair of days i and j in time $O(n \log n)$. But, in fact, it's possible to do better than this. Show how to find the optimal numbers i and j in time $O(n)$.

8. The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots.

Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of n seconds; in the i^{th} second, x_i robots arrive. Based on remote sensing data, you know this sequence $x_1, x_2, \ldots, x_n$ in advance.
- You have at your disposal an *electromagnetic pulse* (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function $f(\cdot)$ so that if j seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the k^{th} second, and it has been j seconds since it was previously used, then it will destroy $\min(x_k, f(j))$ robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the j^{th} second, then it is capable of destroying up to $f(j)$ robots.

The problem. Given the data on robot arrivals $x_1, x_2, \ldots, x_n$, and given the recharging function $f(\cdot)$, choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

Example. Suppose $n = 4$, and the values of x_i and $f(i)$ are given by the following table.

i	**1**	**2**	**3**	**4**
x_i	1	10	10	1
$f(i)$	1	2	4	8

The best solution would be to activate the EMP in the 3^{rd} and the 4^{th} seconds. In the 3^{rd} second, the EMP has gotten to charge for 3 seconds, and so it destroys $\min(10, 4) = 4$ robots; In the 4^{th} second, the EMP has only gotten to charge for 1 second since its last use, and it destroys $\min(1, 1) = 1$ robot. This is a total of 5.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
Schedule-EMP(x_1, ..., x_n)
  Let j be the smallest number for which f(j) ≥ x_n
    (If no such j exists then set j = n)
  Activate the EMP in the n^th second
  If n − j ≥ 1 then
    Continue recursively on the input x_1, ..., x_{n−j}
      (i.e., invoke Schedule-EMP(x_1, ..., x_{n−j}))
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes the data on robot arrivals $x_1, x_2, \ldots, x_n$, and the recharging function $f(\cdot)$, and returns the maximum number of robots that can be destroyed by a sequence of EMP activations.

9. You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i, you're presented with x_i terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s_1 terabytes, on the second day after a reboot, you can process s_2 terabytes, and so on, up to s_n; we assume $s_1 > s_2 > s_3 > \cdots > s_n > 0$. (Of course, on day i you can only process up to x_i terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

The problem. Given the amounts of available data $x_1, x_2, \ldots, x_n$ for the next n days, and given the profile of your system as expressed by $s_1, s_2, \ldots, s_n$ (and starting from a freshly rebooted system on day 1), choose

the days on which you're going to reboot so as to maximize the total amount of data you process.

Example. Suppose $n = 4$, and the values of x_i and s_i are given by the following table.

	Day 1	Day 2	Day 3	Day 4
x	10	1	7	7
s	8	4	2	1

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process $8 + 1 + 2 + 1 = 12$; and other rebooting strategies give you less than 19 as well.)

(a) Give an example of an instance with the following properties.
- There is a "surplus" of data in the sense that $x_i > s_1$ for every i.
- The optimal solution reboots the system at least twice.

In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

(b) Give an efficient algorithm that takes values for $x_1, x_2, \ldots, x_n$ and $s_1, s_2, \ldots, s_n$ and returns the total *number* of terabytes processed by an optimal solution.

10. You're trying to run a large computing job in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available.

Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of n minutes, a *plan* is specified by a choice of A, B, or "*move*" for each minute, with the property that choices A and

B cannot appear in consecutive minutes. For example, if your job is on machine A in minute i, and you want to switch to machine B, then your choice for minute $i+1$ must be *move*, and then your choice for minute $i+2$ can be B. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on A, plus the sum of b_i over all minutes in which the job is on B.

The problem. Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal*.) Note that your plan can start with either of the machines A or B in minute 1.

Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Then the plan of maximum value would be to choose A for minute 1, then *move* for minute 2, and then B for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
In minute 1, choose the machine achieving the larger of a_1, b_1
Set i = 2
While i ≤ n
   What was the choice in minute i − 1?
   If A:
      If b_{i+1} > a_i + a_{i+1} then
         Choose move in minute i and B in minute i + 1
         Proceed to iteration i + 2
      Else
         Choose A in minute i
         Proceed to iteration i + 1
      Endif
   If B: behave as above with roles of A and B reversed
EndWhile
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and returns the *value* of an optimal plan.

11. Suppose you're consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next n weeks, they have a projected *supply* s_i of equipment (measured in pounds), which has to be shipped by an air freight carrier.

Each week's supply can be carried by one of two air freight companies, A or B.

- Company A charges a fixed rate r per pound (so it costs $r \cdot s_i$ to ship a week's supply s_i).
- Company B makes contracts for a fixed amount c per week, independent of the weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A *schedule*, for the PC company, is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks at a time. The *cost* of the schedule is the total amount paid to company A and B, according to the description above.

Give a polynomial-time algorithm that takes a sequence of supply values $s_1, s_2, \ldots, s_n$ and returns a *schedule* of minimum cost.

Example. Suppose $r = 1$, $c = 10$, and the sequence of values is

$$11, 9, 9, 12, 12, 12, 12, 9, 9, 11.$$

Then the optimal schedule would be to choose company A for the first three weeks, then company B for a block of four consecutive weeks, and then company A for the final three weeks.

12. Suppose we want to replicate a file over a collection of n servers, labeled $S_1, S_2, \ldots, S_n$. To place a copy of the file at server S_i results in a *placement cost* of c_i, for an integer $c_i > 0$.

Now, if a user requests the file from server S_i, and no copy of the file is present at S_i, then the servers $S_{i+1}, S_{i+2}, S_{i+3} \ldots$ are searched in order until a copy of the file is finally found, say at server S_j, where $j > i$. This results in an *access cost* of $j - i$. (Note that the lower-indexed servers $S_{i-1}, S_{i-2}, \ldots$ are not consulted in this search.) The access cost is 0 if S_i holds a copy of the file. We will require that a copy of the file be placed at server S_n, so that all such searches will terminate, at the latest, at S_n.

We'd like to place copies of the files at the servers so as to minimize the sum of placement and access costs. Formally, we say that a *configuration* is a choice, for each server S_i with $i = 1, 2, \ldots, n-1$, of whether to place a copy of the file at S_i or not. (Recall that a copy is always placed at S_n.) The *total cost* of a configuration is the sum of all placement costs for servers with a copy of the file, plus the sum of all access costs associated with all n servers.

Give a polynomial-time algorithm to find a configuration of minimum total cost.

13. The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm that trades shares in n different companies. For each pair $i \neq j$, they maintain a trade ratio r_{ij}, meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional; that is, $r_{ij} = \frac{2}{3}$ means that you can trade three shares of i to get two shares of j.

A *trading cycle* for a sequence of shares $i_1, i_2, \ldots, i_k$ consists of successively trading shares in company i_1 for shares in company i_2, then shares in company i_2 for shares i_3, and so on, finally trading shares in i_k back to shares in company i_1. After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an *opportunity cycle*, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

Give a polynomial-time algorithm that finds such an opportunity cycle, if one exists.

14. A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices x and y are connected by an edge if they are able to directly communicate with each other (e.g., by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge (x, y) might disappear as x and y move far apart from each other and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of *maintaining* a path between certain designated nodes. There are

two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (That is, we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

Suppose we have a set of mobile nodes V, and at a particular point in time there is a set E_0 of edges among these nodes. As the nodes move, the set of edges changes from E_0 to E_1, then to E_2, then to E_3, and so on, to an edge set E_b. For $i = 0, 1, 2, \ldots, b$, let G_i denote the graph (V, E_i). So if we were to watch the structure of the network on the nodes V as a "time lapse," it would look precisely like the sequence of graphs $G_0, G_1, G_2, \ldots, G_{b-1}, G_b$. We will assume that each of these graphs G_i is connected.

Now consider two particular nodes $s, t \in V$. For an s-t path P in one of the graphs G_i, we define the *length* of P to be simply the number of edges in P, and we denote this $\ell(P)$. Our goal is to produce a sequence of paths $P_0, P_1, \ldots, P_b$ so that for each i, P_i is an s-t path in G_i. We want the paths to be relatively short. We also do not want there to be too many *changes*—points at which the identity of the path switches. Formally, we define $changes(P_0, P_1, \ldots, P_b)$ to be the number of indices i $(0 \leq i \leq b-1)$ for which $P_i \neq P_{i+1}$.

Fix a constant $K > 0$. We define the *cost* of the sequence of paths $P_0, P_1, \ldots, P_b$ to be

$$cost(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b} \ell(P_i) + K \cdot changes(P_0, P_1, \ldots, P_b).$$

(a) Suppose it is possible to choose a single path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$. Give a polynomial-time algorithm to find the shortest such path.

(b) Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \ldots, P_b$ of minimum cost, where P_i is an s-t path in G_i for $i = 0, 1, \ldots, b$.

15. On most clear days, a group of your friends in the Astronomy Department gets together to plan out the astronomical events they're going to try observing that night. We'll make the following assumptions about the events.

- There are n events, which for simplicity we'll assume occur in sequence separated by exactly one minute each. Thus event j occurs at minute j; if they don't observe this event at exactly minute j, then they miss out on it.

- The sky is mapped according to a one-dimensional coordinate system (measured in degrees from some central baseline); event j will be taking place at coordinate d_j, for some integer value d_j. The telescope starts at coordinate 0 at minute 0.
- The last event, n, is much more important than the others; so it is required that they observe event n.

The Astronomy Department operates a large telescope that can be used for viewing these events. Because it is such a complex instrument, it can only move at a rate of one degree per minute. Thus they do not expect to be able to observe all n events; they just want to observe as many as possible, limited by the operation of the telescope and the requirement that event n must be observed.

We say that a subset S of the events is *viewable* if it is possible to observe each event $j \in S$ at its appointed time j, and the telescope has adequate time (moving at its maximum of one degree per minute) to move between consecutive events in S.

The problem. Given the coordinates of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n. Such a solution will be called *optimal*.

Example. Suppose the one-dimensional coordinates of the events are as shown here.

Event	1	2	3	4	5	6	7	8	9
Coordinate	1	−4	−1	4	5	−4	6	7	−2

Then the optimal solution is to observe events 1, 3, 6, 9. Note that the telescope has time to move from one event in this set to the next, even moving at one degree per minute.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
Mark all events j with |d_n − d_j| > n − j as illegal (as
  observing them would prevent you from observing event n)
Mark all other events as legal
Initialize current position to coordinate 0 at minute 0
While not at end of event sequence
  Find the earliest legal event j that can be reached without
    exceeding the maximum movement rate of the telescope
  Add j to the set S
```

```
  Update current position to be coord. ~d_j at minute j
Endwhile
Output the set S
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for the coordinates $d_1, d_2, \ldots, d_n$ of the events and returns the *size* of an optimal solution.

16. There are many sunny days in Ithaca, New York; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on a rainy day. The ranking officer decides to postpone the picnic and must notify everyone by phone. Here is the mechanism she uses to do this.

Each ROTC person on campus except the ranking officer reports to a unique *superior officer*. Thus the reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we will call v a *direct subordinate* of u. See Figure 6.30, in which A is the ranking officer, B and D are the direct subordinates of A, and C is the direct subordinate of B.

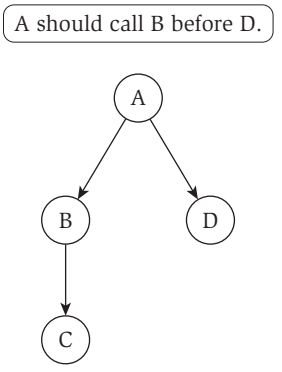


Figure 6.30 A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues this way until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone; for example, in Figure 6.30, A would not be allowed to call C.

We can picture this process as being divided into *rounds*. In one round, each person who has already learned of the postponement can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates. For example, in Figure 6.30, it will take only two rounds if A starts by calling B, but it will take three rounds if A starts by calling D.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds.

17. Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They've defined something called a *rising trend*, as follows.

They have the closing price for a given stock recorded for n days in succession; let these prices be denoted $P[1], P[2], \ldots, P[n]$. A *rising trend* in these prices is a subsequence of the prices $P[i_1], P[i_2], \ldots, P[i_k]$, for days $i_1 < i_2 < \ldots < i_k$, so that

- $i_1 = 1$, and
- $P[i_j] < P[i_{j+1}]$ for each $j = 1, 2, \ldots, k-1$.

Thus a rising trend is a subsequence of the days—beginning on the first day and not necessarily contiguous—so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

Example. Suppose $n = 7$, and the sequence of prices is

$$10, 1, 2, 11, 3, 4, 12.$$

Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend.

(a) Show that the following algorithm does not correctly return the *length* of the longest rising trend, by giving an instance on which it fails to return the correct answer.

```
Define i = 1
       L = 1
For j = 2 to n
  If P[j] > P[i] then
    Set i = j.
    Add 1 to L
  Endif
Endfor
```

In your example, give the actual length of the longest rising trend, and say what the algorithm above returns.

(b) Give an efficient algorithm that takes a sequence of prices $P[1], P[2], \ldots, P[n]$ and returns the *length* of the longest rising trend.

18. Consider the sequence alignment problem over a four-letter alphabet $\{z_1, z_2, z_3, z_4\}$, with a given gap cost and given mismatch costs. Assume that each of these parameters is a positive integer.

Suppose you are given two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ and a proposed alignment between them. Give an $O(mn)$ algorithm to decide whether this alignment is the *unique* minimum-cost alignment between A and B.

19. You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters 1,2,5,7,8,9 form 101101—a repetition of x—and the remaining characters 3,4,6 form 000—a repetition of y.

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: We want to make sure s "unravels" into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

20. Suppose it's nearing the end of the semester and you're taking n courses, each with a final project that still has to be done. Each project will be graded on the following scale: It will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the n projects.

You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you'll spend an integer number of hours on each project. To figure out how best to divide up your time, you've come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$ (rough

estimates, of course) for each of your n courses; if you spend $h \leq H$ hours on the project for course i, you'll get a grade of $f_i(h)$. (You may assume that the functions f_i are *nondecreasing*: if $h < h'$, then $f_i(h) \leq f_i(h')$.)

So the problem is: Given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the f_i, is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in n, g, and H; none of these quantities should appear as an exponent in your running time.

21. Some time back, you helped a group of friends who were doing simulations for a computation-intensive investment company, and they've come back to you with a new problem. They're looking at n consecutive days of a given stock, at some point in the past. The days are numbered $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day.

For certain (possibly large) values of k, they want to study what they call *k-shot strategies*. A k-shot strategy is a collection of m pairs of days $(b_1, s_1), \ldots, (b_m, s_m)$, where $0 \leq m \leq k$ and

$$1 \leq b_1 < s_1 < b_2 < s_2 \cdots < b_m < s_m \leq n.$$

We view these as a set of up to k nonoverlapping intervals, during each of which the investors buy 1,000 shares of the stock (on day b_i) and then sell it (on day s_i). The *return* of a given k-shot strategy is simply the profit obtained from the m buy-sell transactions, namely,

$$1{,}000 \sum_{i=1}^{m} p(s_i) - p(b_i).$$

The investors want to assess the value of k-shot strategies by running simulations on their n-day trace of the stock price. Your goal is to design an efficient algorithm that determines, given the sequence of prices, the k-shot strategy with the maximum possible return. Since k may be relatively large in these simulations, your running time should be polynomial in both n and k; it should not contain k in the exponent.

22. To assess how "well-connected" two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the *number* of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or

negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$. Give an efficient algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.)

23. Suppose you are given a directed graph $G = (V, E)$ with costs on the edges c_e for $e \in E$ and a sink t (costs may be negative). Assume that you also have finite values $d(v)$ for $v \in V$. Someone claims that, for each node $v \in V$, the quantity $d(v)$ is the cost of the minimum-cost path from node v to the sink t.

(a) Give a linear-time algorithm (time $O(m)$ if the graph has m edges) that verifies whether this claim is correct.

(b) Assume that the distances are correct, and $d(v)$ is finite for all $v \in V$. Now you need to compute distances to a different sink t'. Give an $O(m \log n)$ algorithm for computing distances $d'(v)$ for all nodes $v \in V$ to the sink node t'. (*Hint:* It is useful to consider a new cost function defined as follows: for edge $e = (v, w)$, let $c'_e = c_e - d(v) + d(w)$. Is there a relation between costs of paths for the two different costs c and c'?)

24. *Gerrymandering* is the practice of carving up electoral districts in very careful ways so as to lead to outcomes that favor a particular political party. Recent court challenges to the practice have argued that through this calculated redistricting, large numbers of voters are being effectively (and intentionally) disenfranchised.

Computers, it turns out, have been implicated as the source of some of the "villainy" in the news coverage on this topic: Thanks to powerful software, gerrymandering has changed from an activity carried out by a bunch of people with maps, pencil, and paper into the industrial-strength process that it is today. Why is gerrymandering a computational problem? There are database issues involved in tracking voter demographics down to the level of individual streets and houses; and there are algorithmic issues involved in grouping voters into districts. Let's think a bit about what these latter issues look like.

Suppose we have a set of n *precincts* $P_1, P_2, \ldots, P_n$, each containing m registered voters. We're supposed to divide these precincts into two *districts*, each consisting of $n/2$ of the precincts. Now, for each precinct, we have information on how many voters are registered to each of two political parties. (Suppose, for simplicity, that every voter is registered to one of these two.) We'll say that the set of precincts is *susceptible* to gerrymandering if it is possible to perform the division into two districts in such a way that the same party holds a majority in both districts.

Give an algorithm to determine whether a given set of precincts is susceptible to gerrymandering; the running time of your algorithm should be polynomial in n and m.

Example. Suppose we have $n = 4$ precincts, and the following information on registered voters.

Precinct	1	2	3	4
Number registered for party A	55	43	60	47
Number registered for party B	45	57	40	53

This set of precincts is susceptible since, if we grouped precincts 1 and 4 into one district, and precincts 2 and 3 into the other, then party A would have a majority in both districts. (Presumably, the "we" who are doing the grouping here are members of party A.) This example is a quick illustration of the basic unfairness in gerrymandering: Although party A holds only a slim majority in the overall population (205 to 195), it ends up with a majority in not one but both districts.

25. Consider the problem faced by a stockbroker trying to sell a large number of shares of stock in a company whose stock price has been steadily falling in value. It is always hard to predict the right moment to sell stock, but owning a lot of shares in a single company adds an extra complication: the mere act of selling many shares in a single day will have an adverse effect on the price.

Since future market prices, and the effect of large sales on these prices, are very hard to predict, brokerage firms use models of the market to help them make such decisions. In this problem, we will consider the following simple model. Suppose we need to sell x shares of stock in a company, and suppose that we have an accurate model of the market: it predicts that the stock price will take the values $p_1, p_2, \ldots, p_n$ over the next n days. Moreover, there is a function $f(\cdot)$ that predicts the effect of large sales: if we sell y shares on a single day, it will permanently decrease the price by $f(y)$ from that day onward. So, if we sell y_1 shares on day 1, we obtain a price per share of $p_1 - f(y_1)$, for a total income of $y_1 \cdot (p_1 - f(y_1))$. Having sold y_1 shares on day 1, we can then sell y_2 shares on day 2 for a price per share of $p_2 - f(y_1) - f(y_2)$; this yields an additional income of $y_2 \cdot (p_2 - f(y_1) - f(y_2))$. This process continues over all n days. (Note, as in our calculation for day 2, that the decreases from earlier days are absorbed into the prices for all later days.)

Design an efficient algorithm that takes the prices $p_1, \ldots, p_n$ and the function $f(\cdot)$ (written as a list of values $f(1), f(2), \ldots, f(x)$) and determines

the best way to sell x shares by day n. In other words, find natural numbers $y_1, y_2, \ldots, y_n$ so that $x = y_1 + \ldots + y_n$, and selling y_i shares on day i for $i = 1, 2, \ldots, n$ maximizes the total income achievable. You should assume that the share value p_i is monotone decreasing, and $f(\cdot)$ is monotone increasing; that is, selling a larger number of shares causes a larger drop in the price. Your algorithm's running time can have a polynomial dependence on n (the number of days), x (the number of shares), and p_1 (the peak price of the stock).

Example Consider the case when $n = 3$; the prices for the three days are $90, 80, 40$; and $f(y) = 1$ for $y \leq 40{,}000$ and $f(y) = 20$ for $y > 40,000$. Assume you start with $x = 100,000$ shares. Selling all of them on day 1 would yield a price of 70 per share, for a total income of 7,000,000. On the other hand, selling 40,000 shares on day 1 yields a price of 89 per share, and selling the remaining 60,000 shares on day 2 results in a price of 59 per share, for a total income of 7,100,000.

26. Consider the following inventory problem. You are running a company that sells some large product (let's assume you sell trucks), and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i. We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are *stored* until the beginning of the next month. You can store at most S trucks, and it costs C to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of K each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $\{d_i\}$, and minimize the costs. In summary:

- There are two parts to the cost: (1) storage—it costs C for every truck on hand that is not needed that month; (2) ordering fees—it costs K for every order placed.
- In each month you need enough trucks to satisfy the demand d_i, but the number left over after satisfying the demand for the month should not exceed the inventory limit S.

Give an algorithm that solves this problem in time that is polynomial in n and S.

27. The owners of an independently operated gas station are faced with the following situation. They have a large underground tank in which they store gas; the tank can hold up to L gallons at one time. Ordering gas is quite expensive, so they want to order relatively rarely. For each order,

they need to pay a fixed price P for delivery in addition to the cost of the gas ordered. However, it costs c to store a gallon of gas for an extra day, so ordering too much ahead increases the storage cost.

They are planning to close for a week in the winter, and they want their tank to be empty by the time they close. Luckily, based on years of experience, they have accurate projections for how much gas they will need each day until this point in time. Assume that there are n days left until they close, and they need g_i gallons of gas for each of the days $i = 1, \ldots, n$. Assume that the tank is empty at the end of day 0. Give an algorithm to decide on which days they should place orders, and how much to order so as to minimize their total cost.

28. Recall the scheduling problem from Section 4.2 in which we sought to minimize the maximum lateness. There are n jobs, each with a deadline d_i and a required processing time t_i, and all jobs are available to be scheduled starting at time s. For a job i to be done, it needs to be assigned a period from $s_i \geq s$ to $f_i = s_i + t_i$, and different jobs should be assigned nonoverlapping intervals. As usual, such an assignment of times will be called a *schedule*.

In this problem, we consider the same setup, but want to optimize a different objective. In particular, we consider the case in which each job must either be done by its deadline or not at all. We'll say that a subset J of the jobs is *schedulable* if there is a schedule for the jobs in J so that each of them finishes by its deadline. Your problem is to select a schedulable subset of maximum possible size and give a schedule for this subset that allows each job to finish by its deadline.

(a) Prove that there is an optimal solution J (i.e., a schedulable set of maximum size) in which the jobs in J are scheduled in increasing order of their deadlines.

(b) Assume that all deadlines d_i and required times t_i are integers. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number of jobs n, and the maximum deadline $D = \max_i d_i$.

29. Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together

with a spanning subtree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

Show that there is function $f(\cdot)$ and a *polynomial function* $p(\cdot)$ so that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(f(k) \cdot p(n))$.

Notes and Further Reading

Richard Bellman is credited with pioneering the systematic study of dynamic programming (Bellman 1957); the algorithm in this chapter for segmented least squares is based on Bellman's work from this early period (Bellman 1961). Dynamic programming has since grown into a technique that is widely used across computer science, operations research, control theory, and a number of other areas. Much of the recent work on this topic has been concerned with *stochastic dynamic programming*: Whereas our problem formulations tended to tacitly assume that all input is known at the outset, many problems in scheduling, production and inventory planning, and other domains involve uncertainty, and dynamic programming algorithms for these problems encode this uncertainty using a probabilistic formulation. The book by Ross (1983) provides an introduction to stochastic dynamic programming.

Many extensions and variations of the Knapsack Problem have been studied in the area of combinatorial optimization. As we discussed in the chapter, the pseudo-polynomial bound arising from dynamic programming can become prohibitive when the input numbers get large; in these cases, dynamic programming is often combined with other heuristics to solve large instances of Knapsack Problems in practice. The book by Martello and Toth (1990) is devoted to computational approaches to versions of the Knapsack Problem.

Dynamic programming emerged as a basic technique in computational biology in the early 1970s, in a flurry of activity on the problem of sequence comparison. Sankoff (2000) gives an interesting historical account of the early work in this period. The books by Waterman (1995) and Gusfield (1997) provide extensive coverage of sequence alignment algorithms (as well as many related algorithms in computational biology); Mathews and Zuker (2004) discuss further approaches to the problem of RNA secondary structure prediction. The space-efficient algorithm for sequence alignment is due to Hirschberg (1975).

The algorithm for the Shortest-Path Problem described in this chapter is based originally on the work of Bellman (1958) and Ford (1956). Many optimizations, motivated both by theoretical and experimental considerations,

have been added to this basic approach to shortest paths; a Web site maintained by Andrew Goldberg contains state-of-the-art code that he has developed for this problem (among a number of others), based on work by Cherkassky, Goldberg and Radzik (1994). The applications of shortest-path methods to Internet routing, and the trade-offs among the different algorithms for networking applications, are covered in books by Bertsekas and Gallager (1992), Keshav (1997), and Stewart (1998).

Notes on the Exercises Exercise 5 is based on discussions with Lillian Lee; Exercise 6 is based on a result of Donald Knuth; Exercise 25 is based on results of Dimitris Bertsimas and Andrew Lo; and Exercise 29 is based on a result of S. Dreyfus and R. Wagner.