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.

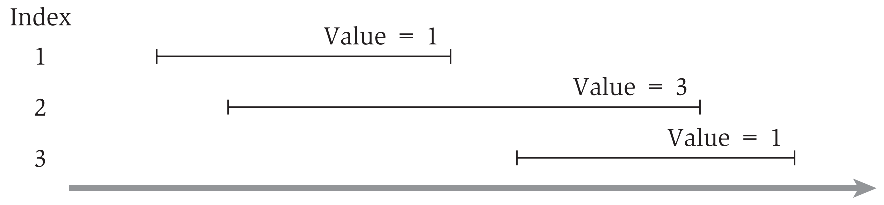


Figure 6.1 A simple instance of weighted interval scheduling.

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: $f_1 \le f_2 \le \cdots \le f_n$. 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.

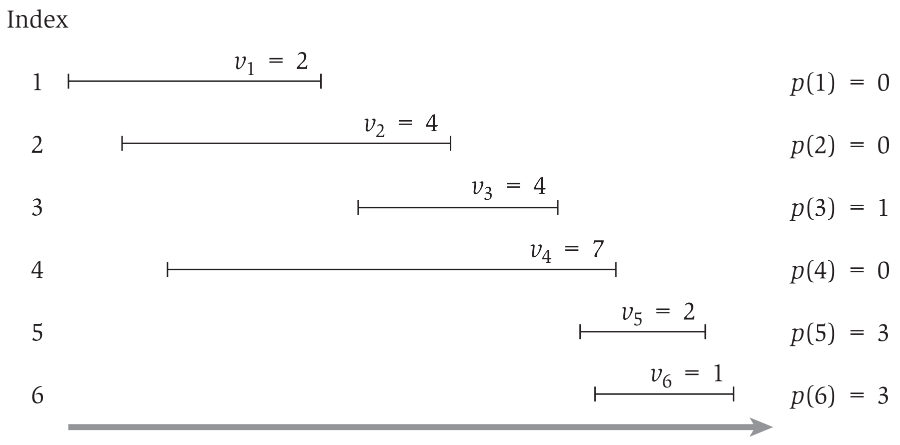


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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* OPT(j) *for each* $j = 1, 2, \ldots, n$.

Proof. By definition OPT$(0) = 0$. Now, take some $j > 0$, and suppose by way of induction that Compute-Opt(i) correctly computes OPT(i) for all $i < j$. By the induction hypothesis, we know that Compute-Opt$(p(j)) =$ OPT$(p(j))$ and Compute-Opt$(j-1) =$ 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

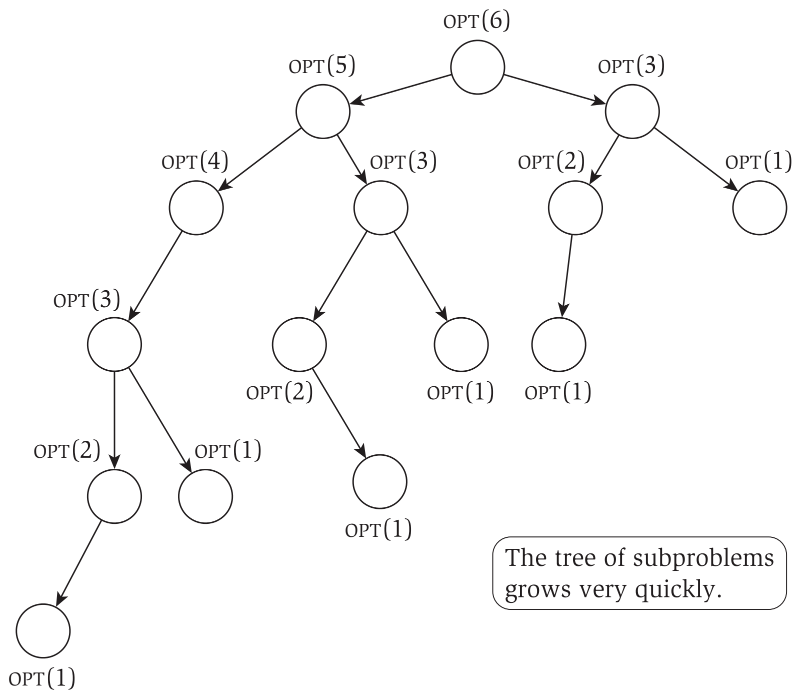


Figure 6.3 The tree of subproblems called by Compute-Opt on the problem instance of Figure 6.2.

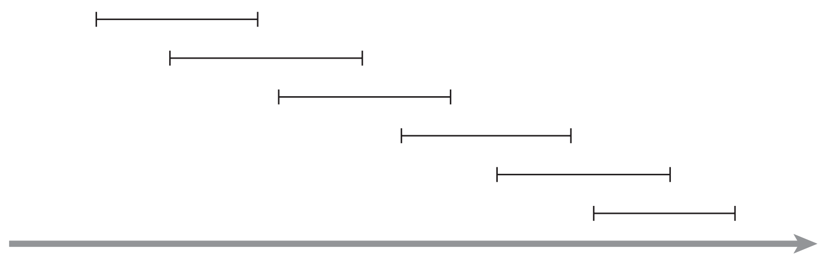


Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

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 $v_j + M[p(j)] \geq M[j-1]$ 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 $j = 1, 2, \ldots, n$
    $M[j] = \max(v_j + M[p(j)], M[j-1])$
  Endfor
```

Analyzing the Algorithm

By exact analogy with the proof of (6.3), we can prove by induction on j that this algorithm writes 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.

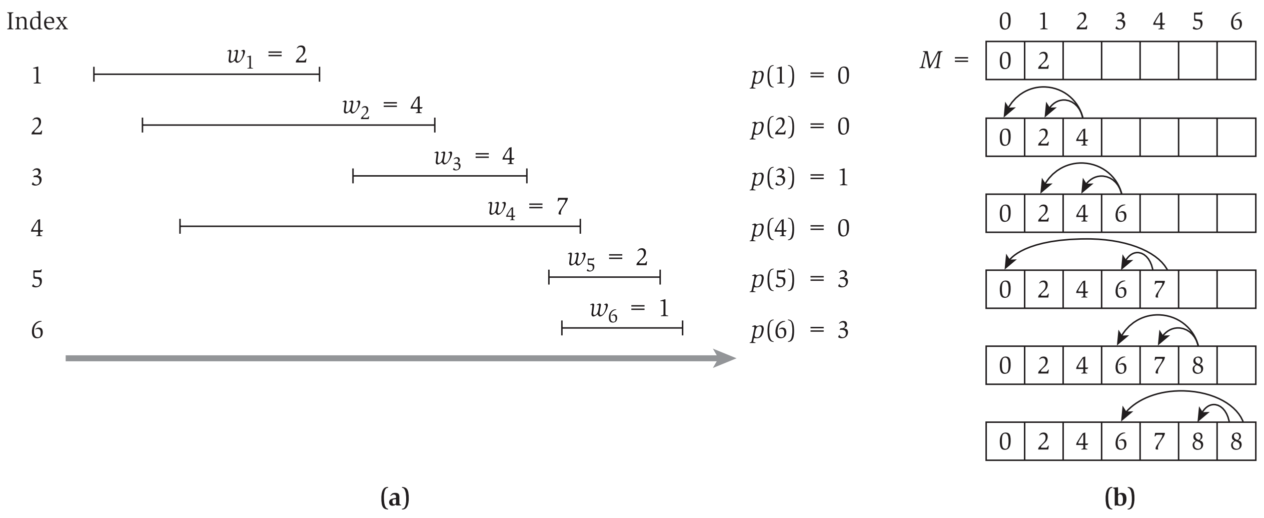


Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

(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:

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

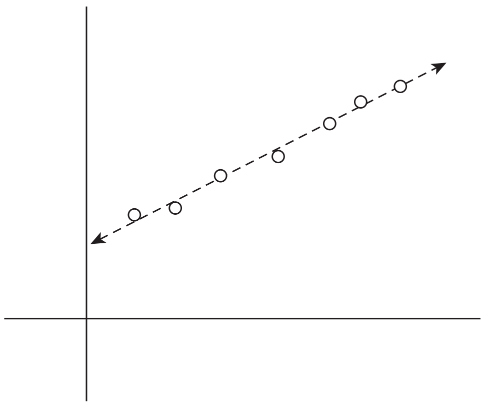


Figure 6.6 A "line of best fit."

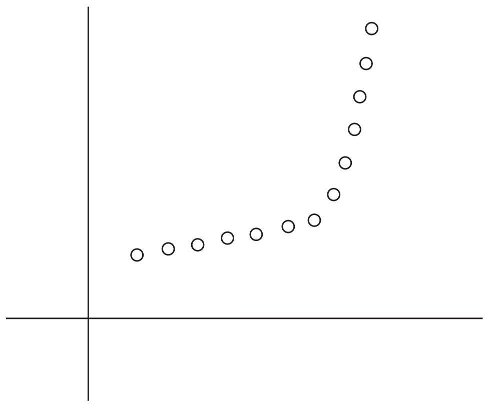


Figure 6.7 A set of points that lie approximately on two lines.

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.

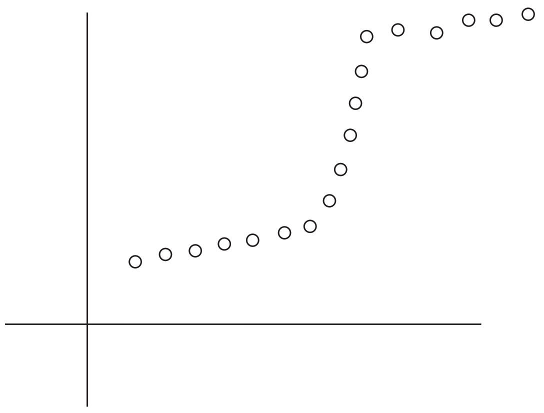


Figure 6.8 A set of points that lie approximately on three lines.

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

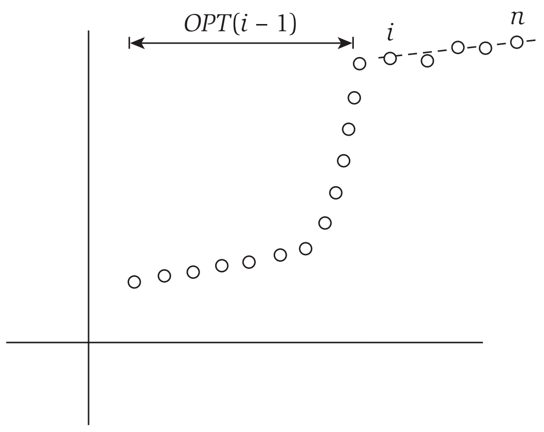


Figure 6.9 A possible solution: a single line segment fits points $p_i, p_{i+1}, \ldots, p_n$, and then an optimal solution is found for the remaining points $p_1, p_2, \ldots, p_{i-1}$.

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 $M[0 \ldots n]$
  Set $M[0] = 0$
  For all pairs $i \le j$
    Compute the least squares error $e_{i,j}$ for the segment $p_i, \ldots, p_j$
  Endfor
  For $j = 1, 2, \ldots, n$
    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 $\{p_i, \ldots, p_j\}$ 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 $\sum_{i \in S} w_i \le W$ 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

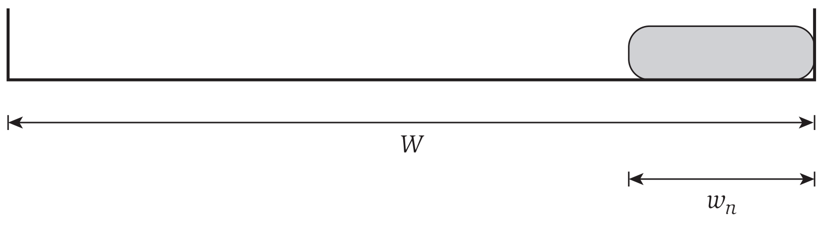


Figure 6.10 After item n is included in the solution, a weight of w_n is used up and there is $W - w_n$ available weight left.

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 $0 \le w \le W$. 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 $\sum_{j \in S} w_j \le w$. 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 nth 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 $M[0 \ldots n, 0 \ldots W]$
  Initialize $M[0, w] = 0$ for each $w = 0, 1, \ldots, W$
  For $i = 1, 2, \ldots, n$
     For $w = 0, \ldots, W$
        Use the recurrence (6.8) to compute $M[i, w]$
     Endfor
  Endfor
  Return $M[n, W]$
```

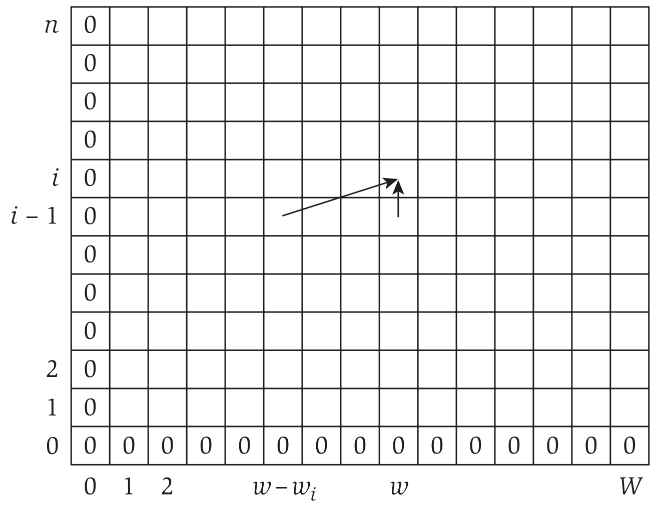


Figure 6.11 The two-dimensional table of OPT values. The leftmost column and bottom row is always 0. The entry for OPT(i, w) is computed from the two other entries OPT$(i-1, w)$ and OPT$(i-1, w-w_i)$, as indicated by the arrows.

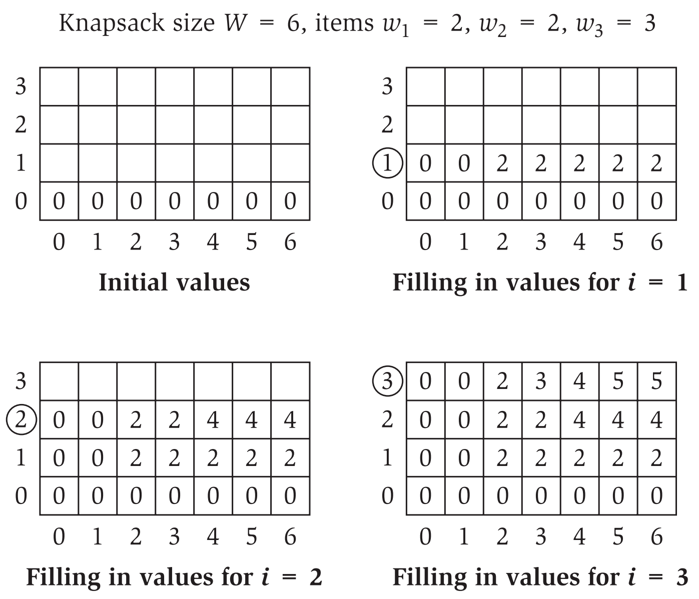
Using (6.8) one can immediately prove by induction that the returned value $M[n, W]$ is the optimum solution value for the requests $1, \ldots, n$ and available weight W.

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 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.



Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

Initial values

	0	1	2	3	4	5	6
3							
2							
1							
0	0	0	0	0	0	0	0

Filling in values for $i = 1$

	0	1	2	3	4	5	6
3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for $i = 2$

	0	1	2	3	4	5	6
3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for $i = 3$

	0	1	2	3	4	5	6
3	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

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.

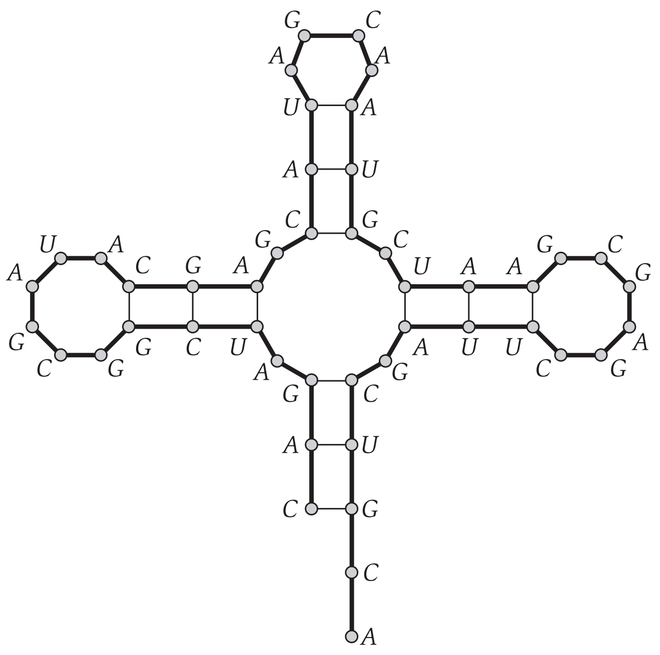


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

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.

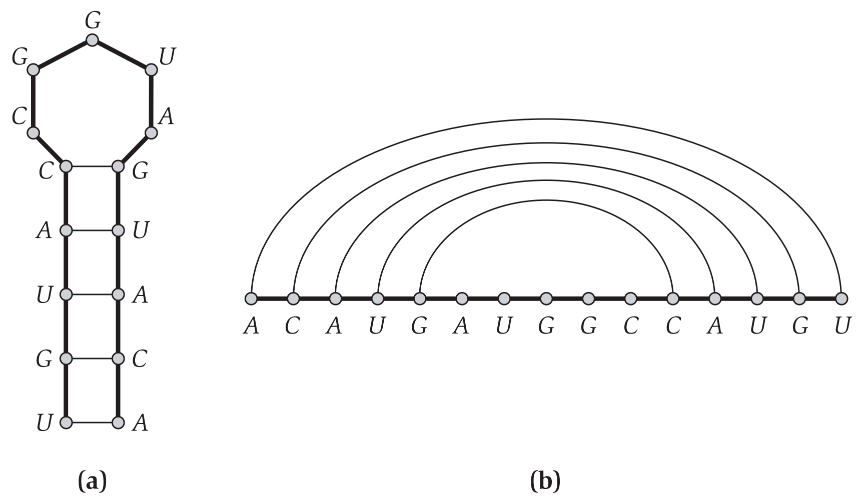


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

a single-stranded RNA molecule $B = b_1b_2 \cdots b_n$ and determines a secondary structure S with the maximum possible number of base pairs.

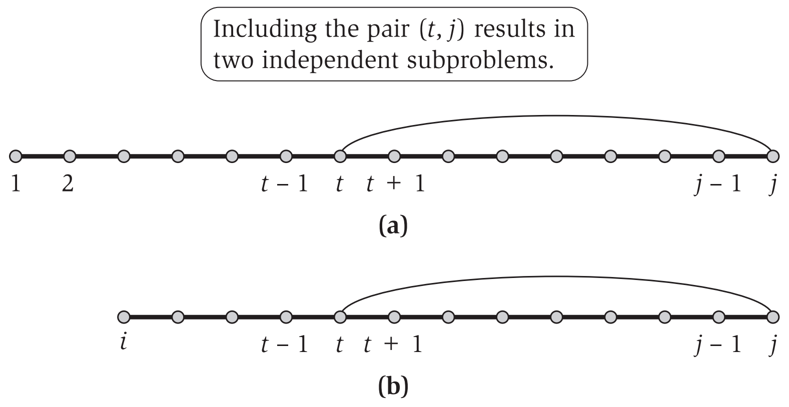
Designing and Analyzing the Algorithm

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 $\text{OPT}(j)$ is the maximum number of base pairs in a secondary structure on $b_1b_2 \cdots b_j$. By the no-sharp-turns condition above, we know that $\text{OPT}(j) = 0$ for $j \le 5$; and we know that $\text{OPT}(n)$ is the solution we're looking for.

The trouble comes when we try writing down a recurrence that expresses $\text{OPT}(j)$ in terms of the solutions to smaller subproblems. We can get partway there: in the optimal secondary structure on $b_1b_2 \cdots b_j$, 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 $\text{OPT}(j-1)$. 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 $b_1b_2 \cdots b_{t-1}$, and the other on the bases $b_{t+1} \cdots b_{j-1}$. The first is solved by $\text{OPT}(t-1)$, but the second is not on our list of subproblems, because it does not begin with b_1.



Including the pair (t, j) results in two independent subproblems.
1
2
$t-1$
t
$t+1$
$j-1$
j
(a)
i
$t-1$
t
$t+1$
$j-1$
j
(b)

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 $i \le j$.

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 $i \ge j - 4$. (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

RNA sequence *ACCGGUAGU*

4	0	0	0	
3	0	0		
2	0			
i = 1				
j =	6	7	8	9

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
i = 1	1			
j =	6	7	8	9

Filling in the values for k = 5

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
i = 1	1	1		
j =	6	7	8	9

Filling in the values for k = 6

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
i = 1	1	1	1	
j =	6	7	8	9

Filling in the values for k = 7

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
i = 1	1	1	1	2
j =	6	7	8	9

Filling in the values for k = 8

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 $i \geq j - 4$
For $k = 5, 6, \ldots, n - 1$
  For $i = 1, 2, \ldots n - k$
    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 $\alpha_{x_iy_j}$ 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 $A[0 \ldots m, 0 \ldots n]$
  Initialize $A[i,0] = i\delta$ for each $i$
  Initialize $A[0,j] = j\delta$ for each $j$
  For $j = 1, \ldots, n$
     For $i = 1, \ldots, m$
         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.

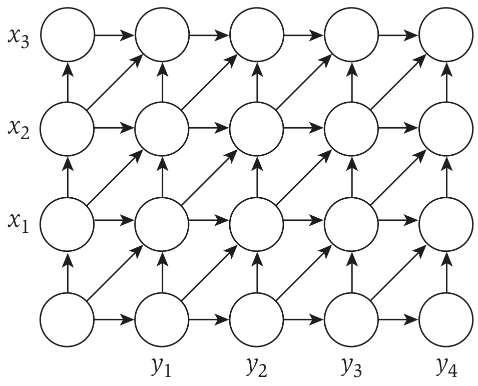


Figure 6.17 A graph-based picture of sequence alignment.

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). ■

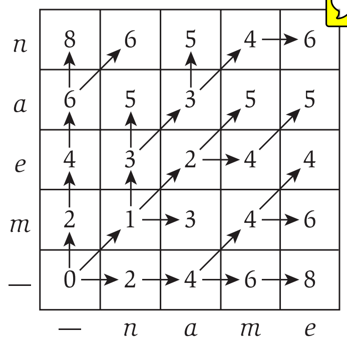


Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

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 $B[0 \ldots m, 0 \ldots 1]$
  Initialize $B[i,0] = i\delta$ for each $i$ (just as in column 0 of $A$)
  For $j = 1, \ldots, n$
     $B[0,1] = j\delta$ (since this corresponds to entry $A[0,j]$)
     For $i = 1, \ldots, m$
        $B[i,1] = \min[\alpha_{x_i y_j} + B[i-1,0],$
                        $\delta + B[i-1,1],\quad \delta + B[i,0]]$
     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 $1 \le i \le j \le m$, 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 $m \le 2$ or $n \le 2$ 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.

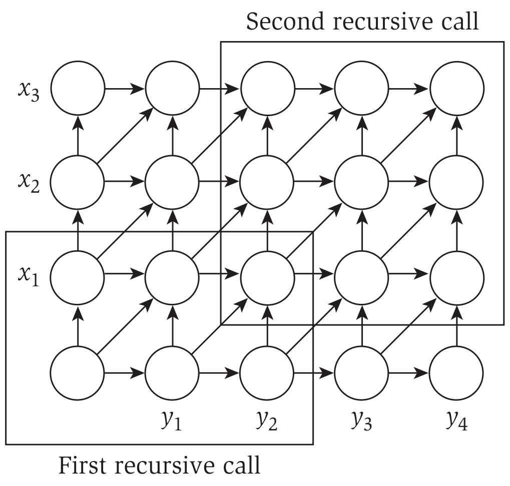


Figure 6.19 The first level of recurrence for the space-efficient `Divide-and-Conquer-Alignment`. The two boxed regions indicate the input to the two recursive cells.

(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. ■