```
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 $\sum_{i \in S} w_i \leq W$ 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

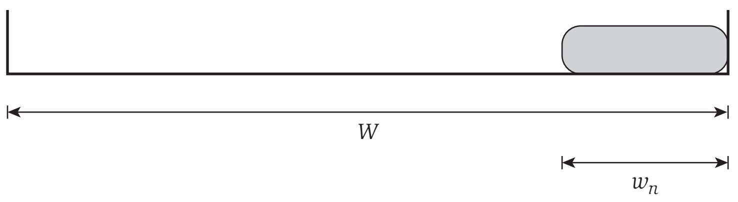


Figure 6.10 After item n is included in the solution, a weight of w_n is used up and there is $W - w_n$ available weight left.

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 $0 \le w \le W$. 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 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 $M[0 \ldots n, 0 \ldots W]$
  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 $M[n, W]$
```

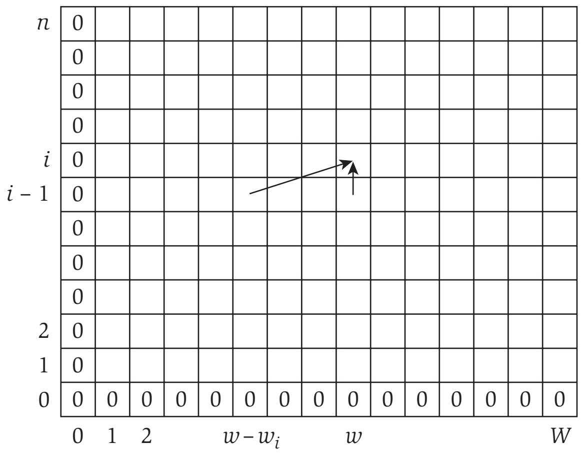


Figure 6.11 The two-dimensional table of OPT values. The leftmost column and bottom row is always 0. The entry for OPT(i, w) is computed from the two other entries OPT$(i-1, w)$ and OPT$(i-1, w-w_i)$, as indicated by the arrows.

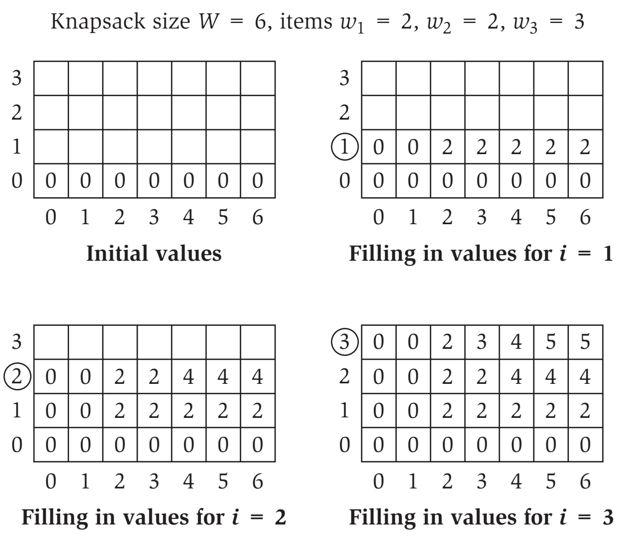
Using (6.8) one can immediately prove by induction that the returned value $M[n, W]$ is the optimum solution value for the requests $1, \ldots, n$ and available weight W.

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 $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

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

Initial values

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

Filling in values for $i = 1$

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
	0	1	2	3	4	5	6

Filling in values for $i = 2$

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
	0	1	2	3	4	5	6

Filling in values for $i = 3$

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.