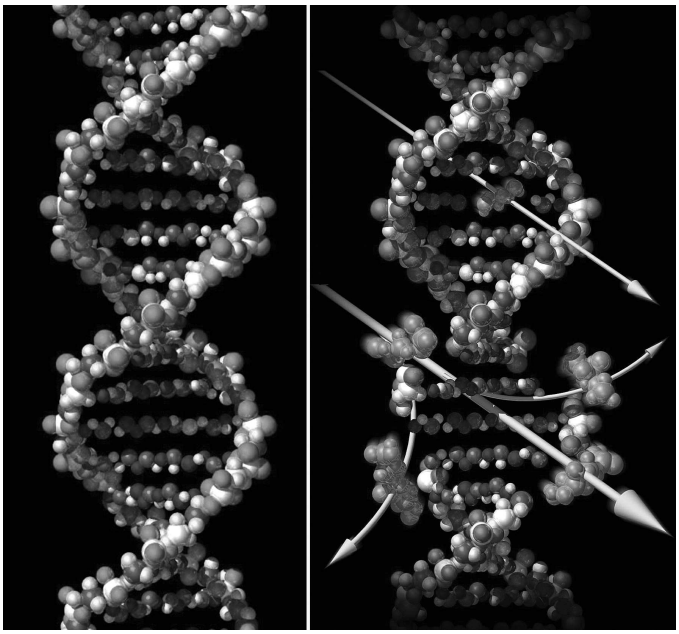
# Chapter 12 Dynamic Programming



Effects of radiation on DNA's double helix, 2003. U.S. Government image. NASA-MSFC.

## Contents

DNA sequences can be viewed as strings of `A`, `C`, `G`, and `T` characters, which represent nucleotides, and finding the similarities between two DNA sequences is an important computation performed in bioinformatics. For instance, when comparing the DNA of different organisms, such alignments can highlight the locations where those organisms have identical DNA patterns. Similarly, places that don't match can show possible mutations between these organisms and a common ancestor, including mutations causing substitutions, insertions, and deletions of nucleotides. Computing the best way to align to DNA strings, therefore, is useful for identifying regions of similarity and difference. For instance, one simple way is to identify a longest common ***subsequence*** of each string, that is, a longest string that can be defined by selecting characters from each string in their order in the respective strings, but not necessarily in a way that is contiguous. (See Figure 12.1.)

From an algorithmic perspective, such similarity computations can appear quite challenging at first. For instance, the most obvious solution for finding the best match between two strings of length $n$ is to try all possible ways of defining subsequences of each string, test if they are the same, and output the one that is longest. Unfortunately, however, there are $2^n$ possible subsequences of each string; hence, this algorithm would run in $O(n2^{2n})$ time, which makes this algorithm impractical.

In this chapter, we discuss the ***dynamic programming*** technique, which is one of the few algorithmic techniques that can take problems, such as this, that seem to require exponential time and produce polynomial-time algorithms to solve them. For example, we show how to solve this problem of finding a longest common subsequence between two strings in time proportional to the product of their lengths, rather than the exponential time of the straightforward method mentioned above.

Moreover, the algorithms that result from applications of the dynamic programming technique are usually quite simple—often needing little more than a few lines of code to describe some nested loops for filling in a table. We demonstrate this effectiveness and simplicity by showing how the dynamic programming technique can be applied to several different types of problems, including matrix-chain products, telescope scheduling, game strategies, the above-mentioned longest common subsequence problem, and the 0-1 knapsack problem. In addition to the topics we discuss in this chapter, dynamic programming is also used for other problems mentioned elsewhere, including maximum subarray-sum (Section 1.3), transitive closure (Section 13.4.2), and all-pairs shortest paths (Section 14.5).

```
A: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
     | ||  || || | ||||||||||||
     G TC  GT CG G AAGCCGGCCGAA
   GTCGT CGGAA GCCG  GC C G AA
   ||||| ||||| ||||  || | | ||
B:  GTCGTTCGGAATGCCGTTGCTCTGTAA
```

**Figure 12.1:** Two DNA sequences, `A` and `B`, and their alignment in terms of a longest subsequence, `GTCGTCGGAAGCCGGCCGAA`, that is common to these two strings.

## 12.1 Matrix Chain-Products

Rather than starting out with an explanation of the general components of the dynamic programming technique, we start out instead by giving a classic, concrete example. Suppose we are given a collection of $n$ two-dimensional matrices for which we wish to compute the product

$$A = A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1},$$

where $A_i$ is a $d_i \times d_{i+1}$ matrix, for $i = 0, 1, 2, \ldots, n-1$. In the standard matrix multiplication algorithm (which is the one we will use), to multiply a $d \times e$-matrix $B$ times an $e \times f$-matrix $C$, we compute the $(i, j)$ entry of the product as follows (see Figure 12.2):

$$\sum_{k=0}^{e-1} B[i, k] \cdot C[k, j].$$

Thus, the computation of a single entry of the product matrix $B \cdot C$ takes $e$ (scalar) multiplications. Overall, the computation of all the entries takes $def$ (scalar) multiplications.

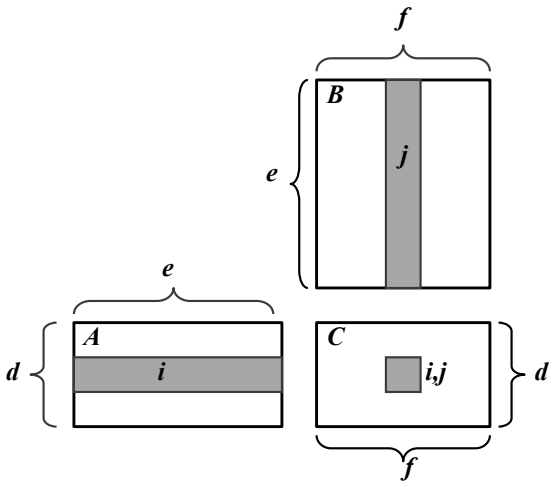



**Figure 12.2:** Multiplication of a $d \times e$ matrix, $A$, and an $e \times f$ matrix, $B$, which produces a $d \times f$ matrix, $C$.

This definition implies that matrix multiplication is associative, that is, it implies that $B \cdot (C \cdot D) = (B \cdot C) \cdot D$. Thus, we can parenthesize the expression for $A$ any way we wish and we will end up with the same answer. We will not necessarily perform the same number of scalar multiplications in each parenthesization, however, as is illustrated in the following example.

**Example 12.1:** *Let $B$ be a $2 \times 10$-matrix, let $C$ be a $10 \times 50$-matrix, and let $D$ be a $50 \times 20$-matrix. Computing $B \cdot (C \cdot D)$ requires $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10400$ multiplications, whereas computing $(B \cdot C) \cdot D$ requires $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ multiplications.*

The ***matrix chain-product*** problem is to determine the parenthesization of the expression defining the product $A$ that minimizes the total number of scalar multiplications performed. Of course, one way to solve this problem is to simply enumerate all the possible ways of parenthesizing the expression for $A$ and determine the number of multiplications performed by each one. Unfortunately, the set of all different parenthesizations of the expression for $A$ is equal in number to the set of all different binary trees that have $n$ external nodes. This number is exponential in $n$. Thus, this straightforward ("brute force") algorithm runs in exponential time, for there are an exponential number of ways to parenthesize an associative arithmetic expression (the number is equal to the $n$th ***Catalan number***, which is $\Omega(4^n/n^{3/2})$).

## Defining Subproblems

We can improve the performance achieved by the brute force algorithm significantly, however, by making a few observations about the nature of the matrix chain-product problem. The first observation is that the problem can be split into ***subproblems***. In this case, we can define a number of different subproblems, each of which is to compute the best parenthesization for some subexpression $A_i \cdot A_{i+1} \cdots A_j$. As a concise notation, we use $N_{i,j}$ to denote the minimum number of multiplications needed to compute this subexpression. Thus, the original matrix chain-product problem can be characterized as that of computing the value of $N_{0,n-1}$. This observation is important, but we need one more in order to apply the dynamic programming technique.

## Characterizing Optimal Solutions

The other important observation we can make about the matrix chain-product problem is that it is possible to characterize an optimal solution to a particular subproblem in terms of optimal solutions to its subproblems. We call this property the ***subproblem optimality*** condition.

In the case of the matrix chain-product problem, we observe that, no matter how we parenthesize a subexpression, there has to be some final matrix multiplication that we perform. That is, a full parenthesization of a subexpression $A_i \cdot A_{i+1} \cdots A_j$ has to be of the form $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, for some $k \in \{i, i+1, \ldots, j-1\}$. Moreover, for whichever $k$ is the right one, the products $(A_i \cdots A_k)$ and $(A_{k+1} \cdots A_j)$ must also be solved optimally. If this were not so, then there would be a global optimal that had one of these subproblems solved suboptimally. But this is impossible, since we could then reduce the total number

of multiplications by replacing the current subproblem solution by an optimal solution for the subproblem. This observation implies a way of explicitly defining the optimization problem for $N_{i,j}$ in terms of other optimal subproblem solutions. Namely, we can compute $N_{i,j}$ by considering each place $k$ where we could put the final multiplication and taking the minimum over all such choices.

### Designing a Dynamic Programming Algorithm

The above discussion implies that we can characterize the optimal subproblem solution $N_{i,j}$ as

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\},$$

where we note that $N_{i,i} = 0$, since no work is needed for a subexpression comprising a single matrix. That is, $N_{i,j}$ is the minimum, taken over all possible places to perform the final multiplication, of the number of multiplications needed to compute each subexpression plus the number of multiplications needed to perform the final matrix multiplication.

The equation for $N_{i,j}$ looks similar to the recurrence equations we derive for divide-and-conquer algorithms, but this is only a superficial resemblance, for there is an aspect of the equation for $N_{i,j}$ that makes it difficult to use divide-and-conquer to compute $N_{i,j}$. In particular, there is a ***sharing of subproblems*** going on that prevents us from dividing the problem into completely independent subproblems (as we would need to do to apply the divide-and-conquer technique). We can, nevertheless, use the equation for $N_{i,j}$ to derive an efficient algorithm by computing $N_{i,j}$ values in a bottom-up fashion, and storing intermediate values in a table of $N_{i,j}$ values. We can begin simply enough by assigning $N_{i,i} = 0$ for $i = 0, 1, \ldots, n-1$. We can then apply the general equation for $N_{i,j}$ to compute $N_{i,i+1}$ values, since they depend only on $N_{i,i}$ and $N_{i+1,i+1}$ values, which are available. Given the $N_{i,i+1}$ values, we can then compute the $N_{i,i+2}$ values, and so on. Therefore, we can build $N_{i,j}$ values up from previously computed values until we can finally compute the value of $N_{0,n-1}$, which is the number that we are searching for. The details of this ***dynamic programming*** solution are given in Algorithm 12.3.

### Analyzing the Matrix Chain-Product Algorithm

Thus, we can compute $N_{0,n-1}$ with an algorithm that consists primarily of three nested for-loops. The outside loop is executed $n$ times. The loop inside is executed at most $n$ times. And the inner-most loop is also executed at most $n$ times. Therefore, the total running time of this algorithm is $O(n^3)$.

**Theorem 12.2:** *Given a chain-product of $n$ two-dimensional matrices, we can compute a parenthesization of this chain that achieves the minimum number of scalar multiplications in $O(n^3)$ time.*

**Algorithm** MatrixChain$(d_0, \ldots, d_n)$:
*Input:* Sequence $d_0, \ldots, d_n$ of integers
*Output:* For $i, j = 0, \ldots, n-1$, the minimum number of multiplications $N_{i,j}$ needed to compute the product $A_i \cdot A_{i+1} \cdots A_j$, where $A_k$ is a $d_k \times d_{k+1}$ matrix

**for** $i \leftarrow 0$ to $n-1$ **do**
  $N_{i,i} \leftarrow 0$
**for** $b \leftarrow 1$ to $n-1$ **do**
  **for** $i \leftarrow 0$ to $n-b-1$ **do**
    $j \leftarrow i+b$
    $N_{i,j} \leftarrow +\infty$
    **for** $k \leftarrow i$ to $j-1$ **do**
      $N_{i,j} \leftarrow \min\{N_{i,j},\ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$.

**Algorithm 12.3:** Dynamic programming algorithm for the matrix chain-product problem.

**Proof:** We have shown above how we can compute the optimal ***number*** of scalar multiplications. But how do we recover the actual parenthesization?

The method for computing the parenthesization itself is is actually quite straightforward. We modify the algorithm for computing $N_{i,j}$ values so that any time we find a new minimum value for $N_{i,j}$, we store, with $N_{i,j}$, the index $k$ that allowed us to achieve this minimum. ■

In Figure 12.4, we illustrate the way the dynamic programming solution to the matrix chain-product problem fills in the array $N$.

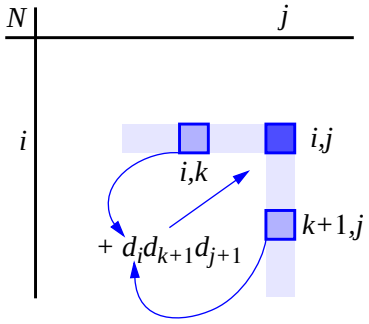



**Figure 12.4:** Illustration of the way the matrix chain-product dynamic-programming algorithm fills in the array $N$.

Now that we have worked through a complete example of the use of the dynamic programming method, we discus in the next section the general aspects of the dynamic programming technique as it can be applied to other problems.

## 12.2 The General Technique

The dynamic programming technique is used primarily for ***optimization*** problems, where we wish to find the "best" way of doing something. Often the number of different ways of doing that "something" is exponential, so a brute-force search for the best is computationally infeasible for all but the smallest problem sizes. We can apply the dynamic programming technique in such situations, however, if the problem has a certain amount of structure that we can exploit. This structure involves the following three components:

**Simple Subproblems:** There has to be some way of breaking the global optimization problem into subproblems, each having a similar structure to the original problem. Moreover, there should be a simple way of defining subproblems with just a few indices, like $i$, $j$, $k$, and so on.

**Subproblem Optimality:** An optimal solution to the global problem must be a composition of optimal subproblem solutions, using a relatively simple combining operation. We should not be able to find a globally optimal solution that contains suboptimal subproblems.

**Subproblem Overlap:** Optimal solutions to unrelated subproblems can contain subproblems in common. Indeed, such overlap allows us to improve the efficiency of a dynamic programming algorithm by storing solutions to subproblems.

This last property is particularly important for dynamic programming algorithms, because it allows them to take advantage of ***memoization***, which is an optimization that allows us to avoid repeated recursive calls by storing intermediate values. Typically, these intermediate values are indexed by a small set of parameters, and we can store them in an array and look them up as needed.

As an illustration of the power of memoization, consider the Fibonacci series, $f(n)$, defined as

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2). \end{aligned}$$

If we implement this equation literally, as a recursive program, then the running time of our algorithm, $T(n)$, as a function of $n$, has the following behavior:

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ T(n) &= T(n-1) + T(n-2). \end{aligned}$$

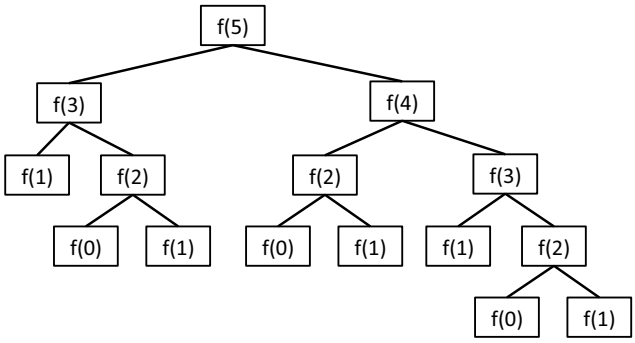
But this implies that

$$T(n) \geq 2T(n-2) = 2^{n/2}.$$

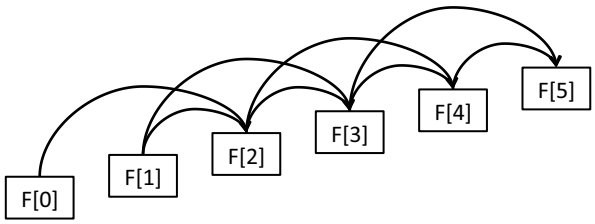
In other words, if we implement this equation recursively as written, then our running time is exponential in $n$. But if we store Fibonacci numbers in an array, $F$, then we can instead calculate the Fibonacci number, $F[n]$, iteratively, as follows:

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
**for** $i = 2$ to $n$ **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$

This algorithm clearly runs in $O(n)$ time, and it illustrates the way memoization can lead to improved performance when subproblems overlap and we use table lookups to avoid repeating recursive calls. (See Figure 12.5.)




(a)




(b)

**Figure 12.5:** The power of memoization. (a) all the function calls needed for a fully recursive definition of the Fibonacci function; (b) the data dependencies in an iterative definition.

## 12.3 Telescope Scheduling

Large, powerful telescopes are precious resources that are typically oversubscribed by the astronomers who request times to use them. This high demand for observation times is especially true, for instance, for the Hubble Space Telescope, which receives thousands of observation requests per month. In this section, we consider a simplified version of the problem of scheduling observations on a telescope, which factors out some details, such as the orientation of the telescope for an observation and who is requesting that observation, but which nevertheless keeps some of the more important aspects of this problem.

The input to this ***telescope scheduling*** problem is a list, $L$, of observation requests, where each request, $i$, consists of the following elements:

- a requested start time, $s_i$, which is the moment when a requested observation should begin
- a finish time, $f_i$, which is the moment when the observation should finish (assuming it begins at its start time)
- a positive numerical benefit, $b_i$, which is an indicator of the scientific gain to be had by performing this observation.

The start and finish times for a observation request are specified by the astronomer requesting the observation; the benefit of a request is determined by an administrator or a review committee for the telescope. To get the benefit, $b_i$, for an observation request, $i$, that observation must be performed by the telescope for the entire time period from the start time, $s_i$, to the finish time, $f_i$. Thus, two requests, $i$ and $j$, ***conflict*** if the time interval $[s_i, f_i]$, intersects the time interval, $[s_j, f_j]$. Given the list, $L$, of observation requests, the optimization problem is to schedule observation requests in a non-conflicting way so as to maximize the total benefit of the observations that are included in the schedule.

There is an obvious exponential-time algorithm for solving this problem, of course, which is to consider all possible subsets of $L$ and choose the one that has the highest total benefit without causing any scheduling conflicts. We can do much better than this, however, by using the dynamic programming technique.

As a first step towards a solution, we need to define subproblems. A natural way to do this is to consider the observation requests according to some ordering, such as ordered by start times, finish times, or benefits. Start times and finish times are essentially symmetric, so we can immediately reduce the choice to that of picking between ordering by finish times and ordering by benefits.

The greedy strategy would be to consider the observation requests ordered by non-increasing benefits, and include each request that doesn't conflict with any chosen before it. This strategy doesn't lead to an optimal solution, however, which we can see after considering a simple example. For instance, suppose we had a list containing just 3 requests—one with benefit 100 that conflicts with two non-conflicting

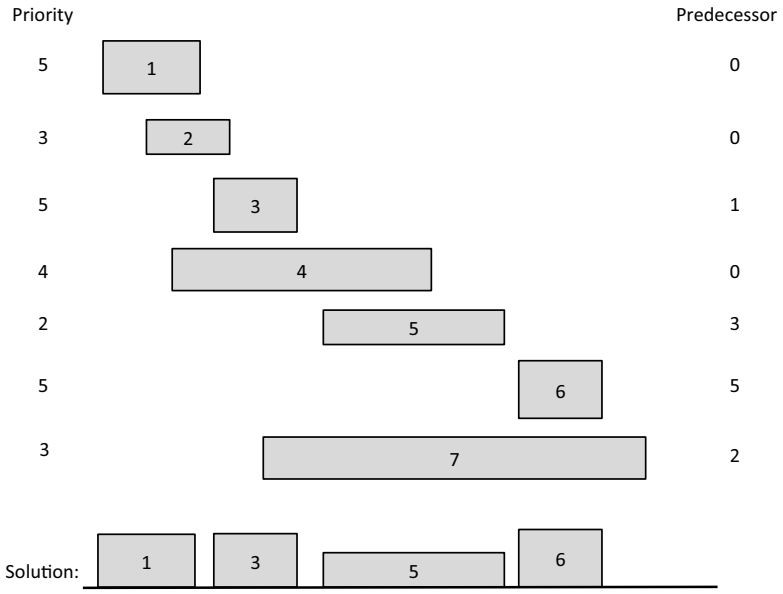



**Figure 12.6:** The telescope scheduling problem. The left and right boundary of each rectangle represent the start and finish times for an observation request. The height of each rectangle represents its benefit. We list each request's benefit on the left and its predecessor on the right. The requests are listed by increasing finish times. The optimal solution has total benefit 17.

requests with benefit 75 each. The greedy algorithm would choose the observation with benefit 100, in this case, whereas we could achieve a total benefit of 150 by taking the two requests with benefit 75 each. So a greedy strategy based on repeatedly choosing a non-conflicting request with maximum benefit won't work.

Let us assume, therefore, that the observation requests in $L$ are sorted by non-decreasing finish times, as shown in Figure 12.6. The idea in this case would be to consider each request according to this ordering. So let us define our set of subproblems in terms of a parameter, $B_i$, which is defined as follows:

$B_i =$ the maximum benefit that can be achieved with the first $i$ requests in $L$.

So, as a boundary condition, we get that $B_0 = 0$.

One nice observation that we can make for this ordering of $L$ by non-decreasing finish times is that, for any request $i$, the set of other requests that conflict with $i$ form a contiguous interval of requests in $L$. Define the ***predecessor***, $\mathrm{pred}(i)$, for each request, $i$, then, to be the largest index, $j < i$, such that request $i$ and $j$ don't conflict. If there is no such index, then define the predecessor of $i$ to be 0. (See Figure 12.6.)

The definition of the predecessor of each request lets us easily reason about the effect that including or not including an observation request, $i$, in a schedule that includes the first $i$ requests in $L$. That is, in a schedule that achieves the optimal

value, $B_i$, for $i \geq 1$, either it includes the observation $i$ or it doesn't; hence, we can reason as follows:

- If the optimal schedule achieving the benefit $B_i$ includes observation $i$, then $B_i = B_{\text{pred}(i)} + b_i$. If this were not the case, then we could get a better benefit by substituting the schedule achieving $B_{\text{pred}(i)}$ for the one we used from among those with indices at most $\text{pred}(i)$.
- On the other hand, if the optimal schedule achieving the benefit $B_i$ does not include observation $i$, then $B_i = B_{i-1}$. If this were not the case, then we could get a better benefit by using the schedule that achieves $B_{i-1}$.

Therefore, we can make the following recursive definition:

$$B_i = \max\{B_{i-1},\ B_{\text{pred}(i)} + b_i\}.$$

Notice that this definition exhibits subproblem overlap. Thus, it is most efficient for us to use memoization when computing $B_i$ values, by storing them in an array, $B$, which is indexed from 0 to $n$. Given the ordering of requests by finish times and an array, $P$, so that $P[i] = \text{pred}(i)$, then we can fill in the array, $B$, using the following simple algorithm:

$B[0] \leftarrow 0$
**for** $i = 1$ to $n$ **do**
    $B[i] \leftarrow \max\{B[i-1],\ B[P[i]] + b_i\}$

After this algorithm completes, the benefit of the optimal solution will be $B[n]$, and, to recover an optimal schedule, we simply need to trace backwards in $B$ from this point. During this trace, if $B[i] = B[i-1]$, then we can assume observation $i$ is not included and move next to consider observation $i-1$. Otherwise, if $B[i] = B[P[i]] + b_i$, then we can assume observation $i$ is included and move next to consider observation $P[i]$.

It is easy to see that the running time of this algorithm is $O(n)$, but it assumes that we are given the list $L$ ordered by finish times and that we are also given the predecessor index for each request $i$. Of course, we can easily sort $L$ by finish times if it is not given to us already sorted according to this ordering. To compute the predecessor of each request, note that it is sufficient that we also have the requests in $L$ sorted by start times. In particular, given a listing of $L$ ordered by finish times and another listing, $L'$, ordered by start times, then a merging of these two lists, as in the merge-sort algorithm (Section 8.1), gives us what we want. The predecessor of request $i$ is literally the index of the predecessor in $L$ of the value, $s_i$, in $L'$. Therefore, we have the following.

**Theorem 12.3:** *Given a list, $L$, of $n$ observation requests, provided in two sorted orders, one by non-decreasing finish times and one by non-decreasing start times, we can solve the telescope scheduling problem for $L$ in $O(n)$ time.*

## 12.4 Game Strategies

There are many types of games, some that are completely random and others where players benefit by employing various kinds of strategies. In this section, we consider two simple games in which dynamic programming can be employed to come up with optimal strategies for playing these games. These are not the only game scenarios where dynamic programming applies, however, as it has been used to analyze strategies for many other games as well, including baseball, American football, and cricket.

### 12.4.1 Coins in a Line

The first game we consider is reported to arise in a problem that is sometimes asked during job interviews at major software and Internet companies (probably because it is so tempting to apply a greedy strategy to this game, whereas the optimal strategy uses dynamic programming).

In this game, which we will call the ***coins-in-a-line game***, an even number, $n$, of coins, of various denominations from various countries, are placed in a line. Two players, who we will call Alice and Bob, take turns removing one of the coins from either end of the remaining line of coins. That is, when it is a player's turn, he or she removes the coin at the left or right end of the line of coins and adds that coin to his or her collection. The player who removes a set of coins with larger total value than the other player wins, where we assume that both Alice and Bob know the value of each coin in some common currency, such as dollars. (See Figure 12.7.)


**Figure 12.7:** The coins-in-a-line game. In this instance, Alice goes first and ultimately ends up with $18 worth of coins. U.S. government images. Credit: U.S. Mint.

### A Dynamic Programming Solution

It is tempting to start thinking of various greedy strategies, such as always choosing the largest-valued coin, minimizing the two remaining choices for the opponent, or even deciding in advance whether it is better to choose all the odd-numbered coins or even-numbered coins. Unfortunately, none of these strategies will consistently lead to an optimal strategy for Alice to play the coins-in-a-line game, assuming that Bob follows an optimal strategy for him.

To design an optimal strategy, we apply the dynamic programming technique. In this case, since Alice and Bob can remove coins from either end of the line, the appropriate way to define subproblems is in terms of a range of indices for the coins, assuming they are initially numbered from 1 to $n$, as in Figure 12.7. Thus, let us define the following indexed parameter:

$$M_{i,j} = \begin{cases} \text{the maximum value of coins taken by Alice, for coins} \\ \text{numbered } i \text{ to } j, \text{ assuming Bob plays optimally.} \end{cases}$$

Therefore, the optimal value for Alice is determined by $M_{1,n}$.

Let us assume that the values of the coins are stored in an array, $V$, so that coin 1 is of Value $V[1]$, coin 2 is of Value $V[2]$, and so on. To determine a recursive definition for $M_{i,j}$, we note that, given the line of coins from coin $i$ to coin $j$, the choice for Alice at this point is either to take coin $i$ or coin $j$ and thereby gain a coin of value $V[i]$ or $V[j]$. Once that choice is made, play turns to Bob, who we are assuming is playing optimally. Thus, he will make the choice among his possibilities that minimizes the total amount that Alice can get from the coins that remain. In other words, Alice must choose based on the following reasoning:

- If $j = i + 1$, then she should pick the larger of $V[i]$ and $V[j]$, and the game is over.
- Otherwise, if Alice chooses coin $i$, then she gets a total value of
$$\min\{M_{i+1,j-1},\ M_{i+2,j}\} + V[i].$$
- Otherwise, if Alice chooses coin $j$, then she gets a total value of
$$\min\{M_{i,j-2},\ M_{i+1,j-1}\} + V[j].$$

Since these are all the choices that Alice has, and she is trying to maximize her returns, then we get the following recurrence equation, for $j > i+1$, where $j-i+1$ is even:

$$M_{i,j} = \max\left\{\min\{M_{i+1,j-1},\ M_{i+2,j}\} + V[i],\ \min\{M_{i,j-2},\ M_{i+1,j-1}\} + V[j]\right\}.$$

In addition, for $i = 1, 2, \ldots, n-1$, we have the initial conditions

$$M_{i,i+1} = \max\{V[i], V[i+1]\}.$$

We can compute the $M_{i,j}$ values, then, using memoization, by starting with the definitions for the above initial conditions and then computing all the $M_{i,j}$'s where $j - i + 1$ is 4, then for all such values where $j - i + 1$ is 6, and so on. Since there are $O(n)$ iterations in this algorithm and each iteration runs in $O(n)$ time, the total time for this algorithm is $O(n^2)$. Of course, this algorithm simply computes all the relevant $M_{i,j}$ values, including the final value, $M_{1,n}$. To recover the actual game strategy for Alice (and Bob), we simply need to note for each $M_{i,j}$ whether Alice should choose coin $i$ or coin $j$, which we can determine from the definition of $M_{i,j}$. And given this choice, we then know the optimal choice for Bob, and that determines if the next choice for Alice is based on $M_{i+2,j}$, $M_{i+1,j-1}$, or $M_{i,j-2}$. Therefore, we have the following.

**Theorem 12.4:** *Given an even number, $n$, of coins in a line, all of known values, we can determine in $O(n^2)$ time the optimal strategy for the first player, Alice, to maximize her returns in the coins-in-a-line game, assuming Bob plays optimally.*

## 12.4.2 Probabilistic Game Strategies and Backward Induction

In addition to games, like chess and the coins-in-a-line game, which are purely strategic, there are lots of games that involve some combination of strategy and randomness (or events that can be modeled probabilistically), like backgammon and sports. Another application of dynamic programming in the context of games arises in these games, in a way that involves combining probability and optimization. To illustrate this point, we consider in this section a strategic decision that arises in the game of American football, which hereafter we refer to simply as "football."

### Extra Points in Football

After a team scores a touchdown in football, they have have a choice between kicking an extra point, which involves kicking the ball through the goal posts to add 1 point to their score if this is successful, or attempting a two-point conversion, which involves lining up again and advancing the ball into the end zone to add 2 points to their score if this is successful. In professional football teams, extra point attempts are successful with a probability of .98 and two-point conversion have a success probability between .40 and .55, depending on the team.

In addition to these probabilistic considerations, the choice of whether it is better to attempt a two-point conversion or not also depends on the difference in the scores between the two teams and how many possessions are left in the game (a possession is a sequence of plays where one team has control of the ball).

### Developing a Recurrence Equation

Let us characterize the state of a football game in terms of a triple, $(k, d, n)$, where these parameters have the following meanings:

- $k$ is the number of points scored at the end of a possession (0 for no score, 3 for a field goal, and 6 for a touchdown, as we are ignoring safeties and we are counting the effects of extra points after a touchdown separately). Possessions alternate between team A and team B.
- $d$ is the difference in points between team A and team B (which is positive when A is in the lead and negative when B is in the lead).
- $n$ is the number of possessions remaining in the game.

For the sake of this analysis, let us assume that $n$ is a potentially unbounded parameter that is known to the two teams, whereas $k$ is always a constant and $d$ can be considered a constant as well, since no professional football team has come back from a point deficit of $-30$ to win.

We can then define $V_A(k, d, n)$ to be the probability that team A wins the game given that its possession ended with team A scoring $k$ points to now have a score deficit of $d$ and $n$ more possessions remaining in the game. Similarly, define $V_B(k, d, n)$ to be the probability that team A wins the game given that team B's possession ended with team B scoring $k$ points to now cause team A to have a score deficit of $d$ with $n$ more possessions remaining in the game. Thus, team A is trying to maximize $V_A$ and team B is trying to minimize $V_B$.

To derive recursive definitions for $V_A$ and $V_B$, note that at the end of the game, when $n = 0$, the outcome is determined. Thus, $V_A(k, d, 0) = 1$ if and only if $d > 0$, and similarly for $V_B(k, d, 0)$. We assume, based on past performance, that we know the probability that team A or B will score a touchdown or field goal in a possession, and that these probabilities are independent of $k$, $d$, or $n$. Thus, we can determine $V(k, d, n)$, the probability that A wins after completing a possession with no score ($k = 0$) or a field goal ($k = 3$) as follows:

$$\begin{aligned} V_A(0, d, n) \;=\; V_A(3, d, n) \;=\; & \Pr(\text{TD by B}) V_B(6, d-6, n-1) \\ & + \Pr(\text{FG by B}) V_B(3, d-3, n-1) \\ & + \Pr(\text{NS by B}) V_B(0, d, n-1). \end{aligned}$$

The first term quantifies the impact of team B scoring a touchdown (TD) on the next possession, the second term quantifies the impact of team B scoring a field goal (FG) on the next possession, and the third term quantifies the impact of team B having no score (NS) at the end of the next possession. Similar equations hold for $V_B$, with the roles of A and B reversed. For professional football teams, the average probability of a possession ending in a touchdown is .20, the average probability of a possession ending in a field goal is .12; hence, for such an average team, we would take the probability of a possession ending with no score for team B to be .68. The main point of this exercise, however, is to characterize the case when $k = 6$, that is, when a possession ends with a touchdown.

Let $p_1$ denote the probability of success for a extra-point attempt and $p_2$ denote the probability of success for a two-point conversion. Then we have

1. $\Pr(\text{Team A wins if it makes an extra point attempt in state } (6, d, n))$
$= p_1 [\Pr(\text{TD by B}) V_B(6, d-5, n-1)$
$+ \Pr(\text{FG by B}) V_B(3, d-2, n-1)$
$+ \Pr(\text{NS by B}) V_B(0, d+1, n-1)]$
$+ (1-p_1) [\Pr(\text{TD by B}) V_B(6, d-6, n-1)$
$+ \Pr(\text{FG by B}) V_B(3, d-3, n-1)$
$+ \Pr(\text{NS by B}) V_B(0, d, n-1)]$.

2. $\Pr(\text{Team A wins if it tries a two-point conversion in state } (6, d, n))$
$= p_2 [\Pr(\text{TD by B}) V_B(6, d-4, n-1)$
$+ \Pr(\text{FG by B}) V_B(3, d-1, n-1)$
$+ \Pr(\text{NS by B}) V_B(0, d+2, n-1)]$
$+ (1-p_2) [\Pr(\text{TD by B}) V_B(6, d-6, n-1)$
$+ \Pr(\text{FG by B}) V_B(3, d-3, n-1)$
$+ \Pr(\text{NS by B}) V_B(0, d, n-1)]$.

The value of $V_A(6, d, n)$ is the maximum of the above two probabilities. Similar bounds hold for $V_B$, except that $V_B(6, d, n)$ is the minimum of the two similarly-defined probabilities. Given our assumptions about $k$ and $d$, these equations imply that we can compute $V(k, d, n)$ in $O(n)$ time, by incrementally increasing the value of $n$ in the above equations and applying memoization. Note that this amounts to reasoning about the game backwards, in that we start with an ending state and use a recurrence equation to reason backward in time. For this reason, this analysis technique is called ***backward induction***.

In the case of the decision we are considering, given known statistics for an average professional football team, the values of $n$ for when it is better to attempt a two-point conversion are shown in Table 12.8.

| **behind by** $(-d)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $n$ **range** | $\emptyset$ | $[0, 15]$ | $\emptyset$ | $\emptyset$ | $[2, 14]$ | $\emptyset$ | $\emptyset$ | $[2, 8]$ | $[4, 9]$ | $[2, 5]$ |

| **ahead by** $(d)$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ **range** | $\emptyset$ | $[1, 7]$ | $[4, 10]$ | $\emptyset$ | $\emptyset$ | $[1, 15]$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

**Table 12.8:** When it is preferential to attempt a two-point conversion after a touchdown, based on $n$, the number of possessions remaining in a game. Each interval indicates the range of values of $n$ for which it is better to make such an attempt.

## 12.5 The Longest Common Subsequence Problem

A common text processing problem, which, we mentioned in the introduction, arises in genetics, is to test the similarity between two text strings. Recall that, in the genetics application, the two strings correspond to two strands of DNA, which could, for example, come from two individuals, who we will consider genetically related if they have a long subsequence common to their respective DNA sequences. There are other applications, as well, such as in software engineering, where the two strings could come from two versions of source code for the same program, and we may wish to determine which changes were made from one version to the next. In addition, the data gathering systems of search engines, which are called ***Web crawlers***, must be able to distinguish between similar Web pages to avoid needless Web page requests. Indeed, determining the similarity between two strings is considered such a common operation that the Unix/Linux operating systems come with a program, called diff, for comparing text files.

### 12.5.1 Problem Definition

There are several different ways we can define the similarity between two strings. Even so, we can abstract a simple, yet common, version of this problem using character strings and their subsequences. Given a string $X$ of size $n$, a ***subsequence*** of $X$ is any string that is of the form

$$X[i_1]X[i_2]\cdots X[i_k], \quad i_j < i_{j+1} \text{ for } j = 1, \ldots, k;$$

that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from $X$. For example, the string $AAAG$ is a subsequence of the string $CGATAATTGAGA$. Note that the concept of ***subsequence*** of a string is different from the one of ***substring*** of a string.

The specific text similarity problem we address here is the ***longest common subsequence*** (LCS) problem. In this problem, we are given two character strings, $X$ of size $n$ and $Y$ of size $m$, over some alphabet and are asked to find a longest string $S$ that is a subsequence of both $X$ and $Y$.

One way to solve the longest common subsequence problem is to enumerate all subsequences of $X$ and take the largest one that is also a subsequence of $Y$. Since each character of $X$ is either in or not in a subsequence, there are potentially $2^n$ different subsequences of $X$, each of which requires $O(m)$ time to determine whether it is a subsequence of $Y$. Thus, the brute-force approach yields an exponential algorithm that runs in $O(2^n m)$ time, which is very inefficient. In this section, we discuss how to use ***dynamic programming*** to solve the longest common subsequence problem much faster than this.

### 12.5.2 Applying Dynamic Programming to the LCS Problem

We can solve the LCS problem much faster than exponential time using dynamic programming. As mentioned above, one of the key components of the dynamic programming technique is the definition of simple subproblems that satisfy the subproblem optimization and subproblem overlap properties.

Recall that in the LCS problem, we are given two character strings, $X$ and $Y$, of length $n$ and $m$, respectively, and are asked to find a longest string $S$ that is a subsequence of both $X$ and $Y$. Since $X$ and $Y$ are character strings, we have a natural set of indices with which to define subproblems—indices into the strings $X$ and $Y$. Let us define a subproblem, therefore, as that of computing the length of the longest common subsequence of $X[0..i]$ and $Y[0..j]$, denoted $L[i, j]$.

This definition allows us to rewrite $L[i, j]$ in terms of optimal subproblem solutions. We consider the following two cases. (See Figure 12.9.)

**Case 1:** $X[i] = Y[j]$. Let $c = X[i] = Y[j]$. We claim that a longest common subsequence of $X[0..i]$ and $Y[0..j]$ ends with $c$. To prove this claim, let us suppose it is not true. There has to be some longest common subsequence $X[i_1]X[i_2]\ldots X[i_k] = Y[j_1]Y[j_2]\ldots Y[j_k]$. If $X[i_k] = c$ or $Y[j_k] = c$, then we get the same sequence by setting $i_k = i$ and $j_k = j$. Alternately, if $X[j_k] \neq c$, then we can get an even longer common subsequence by adding $c$ to the end. Thus, a longest common subsequence of $X[0..i]$ and $Y[0..j]$ ends with $c = X[i] = Y[j]$. Therefore, we set

$$L[i, j] = L[i - 1, j - 1] + 1 \quad \text{if } X[i] = Y[j]. \tag{12.1}$$

**Case 2:** $X[i] \neq Y[j]$. In this case, we cannot have a common subsequence that includes both $X[i]$ and $Y[j]$. That is, a common subsequence can end with $X[i]$, $Y[j]$, or neither, but not both. Therefore, we set

$$L[i, j] = \max\{L[i - 1, j]\,,\, L[i, j - 1]\} \quad \text{if } X[i] \neq Y[j]. \tag{12.2}$$

In order to make Equations 12.1 and 12.2 make sense in the boundary cases when $i = 0$ or $j = 0$, we define $L[i, -1] = 0$ for $i = -1, 0, 1, \ldots, n-1$ and $L[-1, j] = 0$ for $j = -1, 0, 1, \ldots, m - 1$.

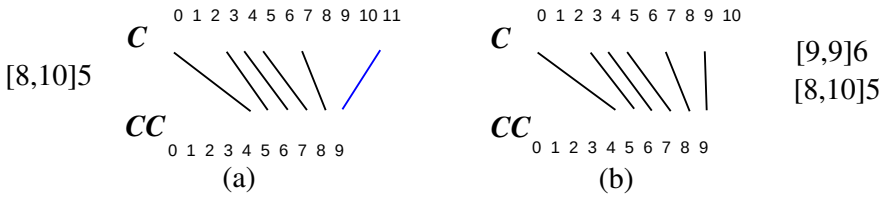



**Figure 12.9:** The two cases for $L[i, j]$: (a) $X[i] = Y[j]$; (b) $X[i] \neq Y[j]$.

## The LCS Algorithm

The above definition of $L[i, j]$ satisfies subproblem optimization, for we cannot have a longest common subsequence without also having longest common subsequences for the subproblems. Also, it uses subproblem overlap, because a subproblem solution $L[i, j]$ can be used in several other problems (namely, the problems $L[i + 1, j]$, $L[i, j + 1]$, and $L[i + 1, j + 1]$).

Turning this definition of $L[i, j]$ into an algorithm is actually quite straightforward. We initialize an $(n + 1) \times (m + 1)$ array, $L$, for the boundary cases when $i = 0$ or $j = 0$. Namely, we initialize $L[i, -1] = 0$ for $i = -1, 0, 1, \ldots, n - 1$ and $L[-1, j] = 0$ for $j = -1, 0, 1, \ldots, m - 1$. (This is a slight abuse of notation, since in reality, we would have to index the rows and columns of $L$ starting with 0.) Then, we iteratively build up values in $L$ until we have $L[n - 1, m - 1]$, the length of a longest common subsequence of $X$ and $Y$. We give a pseudo-code description of how this approach results in a dynamic programming solution to the longest common subsequence (LCS) problem in Algorithm 12.10. Note that the algorithm stores only the $L[i, j]$ values, not the matches.

**Algorithm** LCS$(X, Y)$:
*Input:* Strings $X$ and $Y$ with $n$ and $m$ elements, respectively
*Output:* For $i = 0, \ldots, n - 1$, $j = 0, \ldots, m - 1$, the length $L[i, j]$ of a longest common subsequence of $X[0..i]$ and $Y[0..j]$

```
for i ← −1 to n − 1 do
    L[i, −1] ← 0
for j ← 0 to m − 1 do
    L[−1, j] ← 0
for i ← 0 to n − 1 do
    for j ← 0 to m − 1 do
        if X[i] = Y[j] then
            L[i, j] ← L[i − 1, j − 1] + 1
        else
            L[i, j] ← max{L[i − 1, j] , L[i, j − 1]}
return array L
```

**Algorithm 12.10:** Dynamic programming algorithm for the LCS problem.

## Performance

The running time of Algorithm 12.10 is easy to analyze, for it is dominated by two nested for-loops, with the outer one iterating $n$ times and the inner one iterating $m$ times. Since the if-statement and assignment inside the loop each requires $O(1)$ primitive operations, this algorithm runs in $O(nm)$ time. Thus, the dynamic pro-

gramming technique can be applied to the longest common subsequence problem to improve significantly over the exponential-time brute-force solution to the LCS problem.

Algorithm LCS (12.10) computes the length of the longest common subsequence (stored in $L[n-1, m-1]$), but not the subsequence itself. As shown in the following theorem, a simple postprocessing step can extract the longest common subsequence from the array $L$ returned by the algorithm.

**Theorem 12.5:** *Given a string $X$ of $n$ characters and a string $Y$ of $m$ characters, we can find the longest common subsequence of $X$ and $Y$ in $O(nm)$ time.*

**Proof:** We have already observed that Algorithm LCS computes the ***length*** of a longest common subsequence of the input strings $X$ and $Y$ in $O(nm)$ time. Given the table of $L[i, j]$ values, constructing a longest common subsequence is straightforward. One method is to start from $L[n-1, m-1]$ and work back through the table, reconstructing a longest common subsequence from back to front. At any position $L[i, j]$, we determine whether $X[i] = Y[j]$. If this is true, then we take $X[i]$ as the next character of the subsequence (noting that $X[i]$ is ***before*** the previous character we found, if any), moving next to $L[i-1, j-1]$. If $X[i] \neq Y[j]$, then we move to the larger of $L[i, j-1]$ and $L[i-1, j]$. (See Figure 12.11.) We stop when we reach a boundary entry (with $i = -1$ or $j = -1$). This method constructs a longest common subsequence in $O(n+m)$ additional time. ■

| ***L*** | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 0 | **1** | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 5 | 0 | 1 | 1 | 1 | **2** | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 4 |
| 6 | 0 | 1 | 1 | 2 | 2 | **3** | 3 | 3 | 4 | 4 | 5 | 5 | 5 |
| 7 | 0 | 1 | 1 | 2 | 2 | 3 | **4** | 4 | 4 | 4 | 5 | 5 | 6 |
| 8 | 0 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | **5** | 5 | 5 | 5 | 6 |
| 9 | 0 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | **6** |

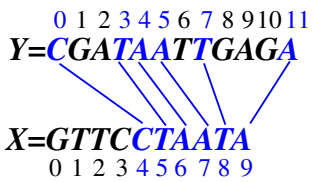



**Figure 12.11:** Illustration of the algorithm for constructing a longest common subsequence from the array $L$.

## 12.6 The 0-1 Knapsack Problem

Suppose a hiker is about to go on a trek through a rain forest carrying a single knapsack. Suppose further that she knows the maximum total weight $W$ that she can carry, and she has a set $S$ of $n$ different useful items that she can potentially take with her, such as a folding chair, a tent, and a copy of this book. Let us assume that each item $i$ has an integer weight $w_i$ and a benefit value $b_i$, which is the utility value that our hiker assigns to item $i$. Her problem, of course, is to optimize the total value of the set $T$ of items that she takes with her, without going over the weight limit $W$. That is, she has the following objective:

$$\text{maximize} \sum_{i \in T} b_i \quad \text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Her problem is an instance of the ***0-1 knapsack problem***. This problem is called a "0-1" problem, because each item must be entirely accepted or rejected. We consider the fractional version of this problem in Section 10.1, and we study how knapsack problems arise in the context of Internet auctions in Exercise R-12.9.

### A Pseudo-Polynomial Time Dynamic Programming Algorithm

We can easily solve the 0-1 knapsack problem in $\Theta(2^n)$ time, of course, by enumerating all subsets of $S$ and selecting the one that has highest total benefit from among all those with total weight not exceeding $W$. This would be an inefficient algorithm, however. Fortunately, we can derive a dynamic programming algorithm for the 0-1 knapsack problem that runs much faster than this in most cases.

As with many dynamic programming problems, one of the hardest parts of designing such an algorithm for the 0-1 knapsack problem is to find a nice characterization for subproblems (so that we satisfy the three properties of a dynamic programming algorithm). To simplify the discussion, number the items in $S$ as $1, 2, \ldots, n$ and define, for each $k \in \{1, 2, \ldots, n\}$, the subset

$$S_k = \{\text{items in } S \text{ labeled } 1, 2, \ldots, k\}.$$

One possibility is for us to define subproblems by using a parameter $k$ so that subproblem $k$ is the best way to fill the knapsack using only items from the set $S_k$. This is a valid subproblem definition, but it is not at all clear how to define an optimal solution for index $k$ in terms of optimal subproblem solutions. Our hope would be that we would be able to derive an equation that takes the best solution using items from $S_{k-1}$ and considers how to add the item $k$ to that. Unfortunately, if we stick with this definition for subproblems, then this approach is fatally flawed. For, as we show in Figure 12.12, if we use this characterization for subproblems, then an optimal solution to the global problem may actually contain a suboptimal subproblem.

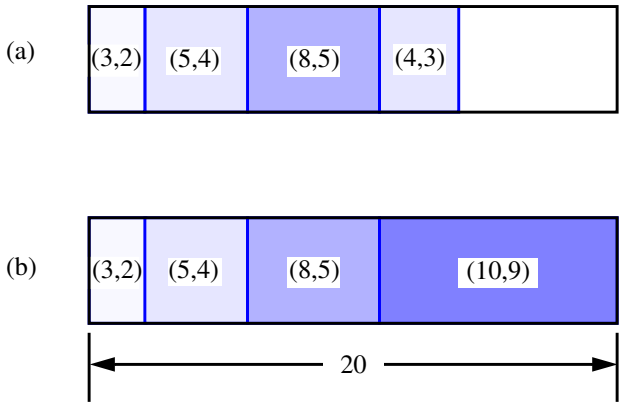



**Figure 12.12:** An example showing that our first approach to defining a knapsack subproblem does not work. The set $S$ consists of five items denoted by the the (***weight***, ***benefit***) pairs $(3,2)$, $(5,4)$, $(8,5)$, $(4,3)$, and $(10,9)$. The maximum total weight is $W = 20$: (a) best solution with the first four items; (b) best solution with the first five items. We shade each item in proportion to its benefit.

One of the reasons that defining subproblems only in terms of an index $k$ is fatally flawed is that there is not enough information represented in a subproblem to provide much help for solving the global optimization problem. We can correct this difficulty, however, by adding a second parameter $w$. Let us therefore formulate each subproblem as that of computing $B[k,w]$, which is defined as the maximum total value of a subset of $S_k$ from among all those subsets having total weight ***at most*** $w$. We have $B[0,w] = 0$ for each $w \leq W$, and we derive the following relationship for the general case

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else.} \end{cases}$$

That is, the best subset of $S_k$ that has total weight at most $w$ is either the best subset of $S_{k-1}$ that has total weight at most $w$ or the best subset of $S_{k-1}$ that has total weight at most $w - w_k$ plus item $k$. Since the best subset of $S_k$ that has total weight $w$ must either contain item $k$ or not, one of these two choices must be the right choice. Thus, we have a subproblem definition that is simple (it involves just two parameters, $k$ and $w$) and satisfies the subproblem optimization condition. Moreover, it has subproblem overlap, for an optimal subset of total weight at most $w$ may be used by many future subproblems.

In deriving an algorithm from this definition, we can make one additional observation, namely, that the definition of $B[k,w]$ is built from $B[k-1,w]$ and possibly $B[k-1,w-w_k]$. Thus, we can implement this algorithm using only a single array $B$, which we update in each of a series of iterations indexed by a parameter $k$ so that at the end of each iteration $B[w] = B[k,w]$. This gives us Algorithm 12.13.

**Algorithm** 01Knapsack$(S, W)$:
***Input:*** Set $S$ of $n$ items, such that item $i$ has positive benefit $b_i$ and positive integer weight $w_i$; positive integer maximum total weight $W$
***Output:*** For $w = 0, \ldots, W$, maximum benefit $B[w]$ of a subset of $S$ with total weight at most $w$

```
for w ← 0 to W do
    B[w] ← 0
for k ← 1 to n do
    for w ← W downto w_k do
        if B[w − w_k] + b_k > B[w] then
            B[w] ← B[w − w_k] + b_k
```

**Algorithm 12.13:** Dynamic programming algorithm for the 0-1 knapsack problem.

The running time of the 01Knapsack algorithm is dominated by the two nested for-loops, where the outer one iterates $n$ times and the inner one iterates at most $W$ times. After it completes we can find the optimal value by locating the value $B[w]$ that is greatest among all $w \leq W$. Thus, we have the following:

**Theorem 12.6:** *Given an integer $W$ and a set $S$ of $n$ items, each of which has a positive benefit and a positive integer weight, we can find the highest benefit subset of $S$ with total weight at most $W$ in $O(nW)$ time.*

**Proof:** We have given Algorithm 12.13 (01Knapsack) for constructing the ***value*** of the maximum-benefit subset of $S$ that has total weight at most $W$ using an array $B$ of benefit values. We can easily convert our algorithm into one that outputs the items in a best subset, however. We leave the details of this conversion as an exercise. ■

In addition to being another useful application of the dynamic programming technique, Theorem 12.6 states something very interesting. Namely, it states that the running time of our algorithm depends on a parameter $W$ that, strictly speaking, is not proportional to the size of the input (the $n$ items, together with their weights and benefits, plus the ***number*** $W$). Assuming that $W$ is encoded in some standard way (such as a binary number), then it takes only $O(\log W)$ bits to encode $W$. Moreover, if $W$ is very large (say $W = 2^n$), then this dynamic programming algorithm would actually be asymptotically slower than the brute force method. Thus, technically speaking, this algorithm is not a polynomial-time algorithm, for its running time is not actually a function of the ***size*** of the input. It is common to refer to an algorithm such as our knapsack dynamic programming algorithm as being a ***pseudo-polynomial time*** algorithm, for its running time depends on the magnitude of a number given in the input, not its encoding size.