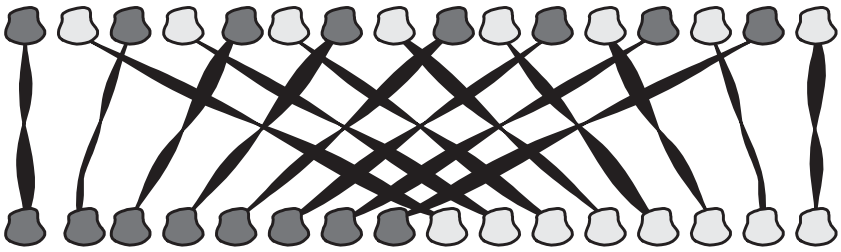
Chapter

5 Fundamental Techniques



Contents

A popular television network broadcasts two different shows about carpentry. In one show, the host builds furniture using specialized power tools, and in the other the host builds furniture using general-purpose hand tools. The specialized tools, used in the first show, are good at the jobs they are intended for, but none of them is very versatile. The tools in the second show are fundamental, however, because they can be used effectively for a wide variety of different tasks.

These two television shows provide an interesting metaphor for data structure and algorithm design. There are some algorithmic tools that are quite specialized. They are good for the problems they are intended to solve, but they are not very versatile. There are other algorithmic tools, however, that are ***fundamental*** in that they can be applied to a wide variety of different data structure and algorithm design problems. Learning to use these fundamental techniques is a craft, and this chapter is dedicated to developing the knowledge for using these techniques effectively.

The fundamental techniques covered in this chapter are the greedy method, divide-and-conquer, and dynamic programming. These techniques are versatile, and examples are given both in this chapter and in other chapters of this book.

The greedy method is used in algorithms for weighted graphs discussed in Chapter 7, as well as a data compression problem presented in Section 9.3. The main idea of this technique, as the name implies, is to make a series of greedy choices in order to construct an optimal solution (or close to optimal solution) for a given problem. In this chapter, we give the general structure for the greedy method and show how it can be applied to knapsack and scheduling problems.

Divide-and-conquer is used in the merge-sort and quick-sort algorithms of Chapter 4. The general idea behind this technique is to solve a given problem by dividing it into a small number of similar subproblems, recursively solve each of the subproblems until they are small enough to solve by brute force, and, after the recursive calls return, merge all the subproblems together to derive a solution to the original problem. In this chapter, we show how to design and analyze general divide-and-conquer algorithms and we give additional applications of this technique to the problems of multiplying big integers and large matrices. We also give a number of techniques for solving divide-and-conquer recurrence equations, including a general master theorem that can be applied to a variety of equations.

The dynamic programming technique might at first seem a bit mysterious, but it is quite powerful. The main idea is to solve a given problem by characterizing its subproblems using a small set of integer indices. The goal of this characterization is to allow an optimal solution to a subproblem to be defined by the combination of (possibly overlapping) solutions to even smaller subproblems. If we can construct such a characterization, which is the hardest step in using the dynamic programming technique, then we can build a rather straightforward algorithm that builds up larger subproblem solutions from smaller ones. This technique underlies the Floyd-Warshall transitive closure algorithm of Chapter 6. In this chapter, we describe the general framework of dynamic programming and give several applications, including to the 0-1 knapsack problem.

5.1 The Greedy Method

The first algorithmic technique we consider in this chapter is the ***greedy method***. We characterize this greedy method design pattern in terms of a general ***greedy-choice*** property, and we give two applications of its use.

The greedy method is applied to optimization problems, that is, problems that involve searching through a set of ***configurations*** to find one that minimizes or maximizes an ***objective function*** defined on these configurations. The general formula of the greedy method could not be simpler. In order to solve a given optimization problem, we proceed by a sequence of choices. The sequence starts from some well-understood starting configuration, and then iteratively makes the decision that seems best from all of those that are currently possible.

This greedy approach does not always lead to an optimal solution. But there are several problems that it does work optimally for, and such problems are said to possess the ***greedy-choice*** property. This is the property that a global optimal configuration can be reached by a series of locally optimal choices (that is, choices that are the best from among the possibilities available at the time), starting from a well-defined configuration.

5.1.1 The Fractional Knapsack Problem

Consider the ***fractional knapsack*** problem, where we are given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, and we wish to find the maximum-benefit subset that does not exceed a given weight W. If we are restricted to entirely accepting or rejecting each item, then we would have the 0-1 version of this problem (for which we give a dynamic programming solution in Section 5.3.3). Let us now allow ourselves to take arbitrary fractions of some elements, however. The motivation for this fractional knapsack problem is that we are going on a trip and we have a single knapsack that can carry items that together have weight at most W. In addition, we are allowed to break items into fractions arbitrarily. That is, we can take an amount x_i of each item i such that

$$0 \leq x_i \leq w_i \text{ for each } i \in S \quad \text{and} \quad \sum_{i \in S} x_i \leq W.$$

The total benefit of the items taken is determined by the objective function

$$\sum_{i \in S} b_i (x_i / w_i).$$

Consider, for example, a student who is going to an outdoor sporting event and must fill a knapsack full of foodstuffs to take along. Each candidate foodstuff is something that can be easily divided into fractions, such as soda pop, potato chips, popcorn, and pizza.

Algorithm FractionalKnapsack(S,W):
Input: Set S of items, such that each item $i \in S$ has a positive benefit b_i and a positive weight w_i; positive maximum total weight W
Output: Amount x_i of each item $i \in S$ that maximizes the total benefit while not exceeding the maximum total weight W

for each item $i \in S$ **do**
  $x_i \leftarrow 0$
  $v_i \leftarrow b_i/w_i$ {***value index*** of item i}
$w \leftarrow 0$ {total weight}
while $w < W$ **do**
  remove from S an item i with highest value index {greedy choice}
  $a \leftarrow \min\{w_i, W - w\}$ {more than $W - w$ causes a weight overflow}
  $x_i \leftarrow a$
  $w \leftarrow w + a$

Algorithm 5.1: A greedy algorithm for the fractional knapsack problem.

This is one place where greed is good, for we can solve the fractional knapsack problem using the greedy approach shown in Algorithm 5.1.

The FractionalKnapsack algorithm can be implemented in $O(n \log n)$ time, where n is the number of items in S. Specifically, we use a heap-based priority queue (Section 2.4.3) to store the items of S, where the key of each item is its value index. With this data structure, each greedy choice, which removes the item with greatest value index, takes $O(\log n)$ time.

To see that the fractional knapsack problem satisfies the greedy-choice property, suppose that there are two items i and j such that

$$x_i < w_i, \quad x_j > 0, \quad \text{and} \quad v_i < v_j.$$

Let

$$y = \min\{w_i - x_i, x_j\}.$$

We could then replace an amount y of item j with an equal amount of item i, thus increasing the total benefit without changing the total weight. Therefore, we can correctly compute optimal amounts for the items by greedily choosing items with the largest value index. This leads to the following theorem.

Theorem 5.1: *Given a collection S of n items, such that each item i has a benefit b_i and weight w_i, we can construct a maximum-benefit subset of S, allowing for fractional amounts, that has a total weight W in $O(n \log n)$ time.*

This theorem shows how efficiently we can solve the fractional version of the knapsack problem. The all-or-nothing, or "0-1" version of the knapsack problem does not satisfy the greedy choice property, however, and solving this version of the problem is much harder, as we explore in Sections 5.3.3 and 13.3.4.

5.1.2 Task Scheduling

Let us consider another optimization problem. Suppose we are given a set T of n ***tasks***, such that each task i has a ***start time***, s_i, and a finish time, f_i (where $s_i < f_i$). Task i must start at time s_i and it is guaranteed to be finished by time f_i. Each task has to be performed on a ***machine*** and each machine can execute only one task at a time. Two tasks i and j are ***nonconflicting*** if $f_i \leq s_j$ or $f_j \leq s_i$. Two tasks can be scheduled to be executed on the same machine only if they are nonconflicting.

The ***task scheduling*** problem we consider here is to schedule all the tasks in T on the fewest machines possible in a nonconflicting way. Alternatively, we can think of the tasks as meetings that we must schedule in as few conference rooms as possible. (See Figure 5.2.)

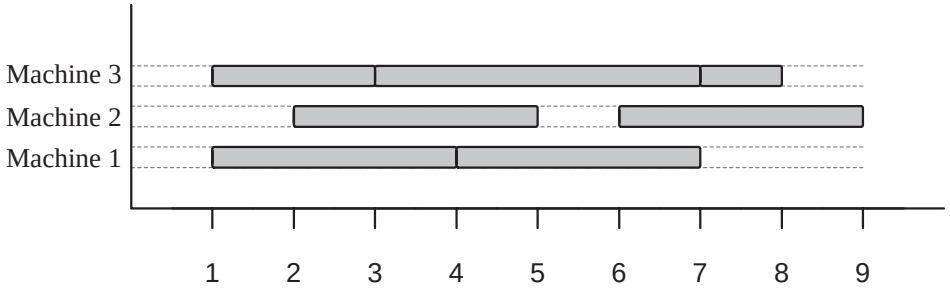


Figure 5.2: An illustration of a solution to the task scheduling problem, for tasks whose collection of pairs of start times and finish times is $\{(1,3), (1,4), (2,5), (3,7), (4,7), (6,9), (7,8)\}$.

In Algorithm 5.3, we describe a simple greedy algorithm for this problem.

Algorithm TaskSchedule(T):
Input: A set T of tasks, such that each task has a start time s_i and a finish time f_i
Output: A nonconflicting schedule of the tasks in T using a minimum number of machines

```
m ← 0          {optimal number of machines}
while T ≠ ∅ do
    remove from T the task i with smallest start time s_i
    if there is a machine j with no task conflicting with task i then
        schedule task i on machine j
    else
        m ← m + 1          {add a new machine}
        schedule task i on machine m
```

Algorithm 5.3: A greedy algorithm for the task scheduling problem.

Correctness of Greedy Task Scheduling

In the algorithm TaskSchedule, we begin with no machines and we consider the tasks in a greedy fashion, ordered by their start times. For each task i, if we have a machine that can handle task i, then we schedule i on that machine. Otherwise, we allocate a new machine, schedule i on it, and repeat this greedy selection process until we have considered all the tasks in T.

The fact that the above TaskSchedule algorithm works correctly is established by the following theorem.

Theorem 5.2: *Given a set of n tasks specified by their start and finish times, Algorithm* TaskSchedule *produces a schedule of the tasks with the minimum number of machines in $O(n \log n)$ time.*

Proof: We can show that the above simple greedy algorithm, TaskSchedule, finds an optimal schedule on the minimum number of machines by a simple contradiction argument.

So, suppose the algorithm does not work. That is, suppose the algorithm finds a nonconflicting schedule using k machines but there is a nonconflicting schedule that uses only $k-1$ machines. Let k be the last machine allocated by our algorithm, and let i be the first task scheduled on k. By the structure of the algorithm, when we scheduled i, each of the machines 1 through $k-1$ contained tasks that conflict with i. Since they conflict with i and because we consider tasks ordered by their start times, all the tasks currently conflicting with task i must have start times less than or equal to s_i, the start time of i, and have finish times after s_i. In other words, these tasks not only conflict with task i, they all conflict with each other. But this means we have k tasks in our set T that conflict with each other, which implies it is impossible for us to schedule all the tasks in T using only $k-1$ machines. Therefore, k is the minimum number of machines needed to schedule all the tasks in T.

We leave as a simple exercise (R-5.2) the job of showing how to implement the Algorithm TaskSchedule in $O(n \log n)$ time. ■

We consider several other applications of the greedy method in this book, including two problems in string compression (Section 9.3), where the greedy approach gives rise to a construction known as Huffman coding, and graph algorithms (Section 7.3), where the greedy approach is used to solve shortest path and minimum spanning tree problems.

The next technique we discuss is the divide-and-conquer technique, which is a general methodology for using recursion to design efficient algorithms.

5.2 Divide-and-Conquer

The ***divide-and-conquer*** technique involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then "merging" or "marrying" the solutions to the subproblem(s) to produce a solution to the original problem.

We can model the divide-and-conquer approach by using a parameter n to denote the size of the original problem, and let $S(n)$ denote this problem. We solve the problem $S(n)$ by solving a collection of k subproblems $S(n_1)$, $S(n_2)$, ..., $S(n_k)$, where $n_i < n$ for $i = 1, \ldots, k$, and then merging the solutions to these subproblems.

For example, in the classic merge-sort algorithm (Section 4.1), $S(n)$ denotes the problem of sorting a sequence of n numbers. Merge-sort solves problem $S(n)$ by dividing it into two subproblems $S(\lfloor n/2 \rfloor)$ and $S(\lceil n/2 \rceil)$, recursively solving these two subproblems, and then merging the resulting sorted sequences into a single sorted sequence that yields a solution to $S(n)$. The merging step takes $O(n)$ time. This, the total running time of the merge-sort algorithm is $O(n \log n)$.

As with the merge-sort algorithm, the general divide-and-conquer technique can be used to build algorithms that have fast running times.

5.2.1 Divide-and-Conquer Recurrence Equations

To analyze the running time of a divide-and-conquer algorithm we utilize a ***recurrence equation*** (Section 1.1.4). That is, we let a function $T(n)$ denote the running time of the algorithm on an input of size n, and characterize $T(n)$ using an equation that relates $T(n)$ to values of the function T for problem sizes smaller than n. In the case of the merge-sort algorithm, we get the recurrence equation

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2, \end{cases}$$

for some constant $b > 0$, taking the simplifying assumption that n is a power of 2. In fact, throughout this section, we take the simplifying assumption that n is an appropriate power, so that we can avoid using floor and ceiling functions. Every asymptotic statement we make about recurrence equations will still be true, even if we relax this assumption, but justifying this fact formally involves long and boring proofs. As we observed above, we can show that $T(n)$ is $O(n \log n)$ in this case. In general, however, we will possibly get a recurrence equation that is more challenging to solve than this one. Thus, it is useful to develop some general ways of solving the kinds of recurrence equations that arise in the analysis of divide-and-conquer algorithms.

The Iterative Substitution Method

One way to solve a divide-and-conquer recurrence equation is to use the ***iterative substitution*** method, which is more colloquially known as the "plug-and-chug" method. In using this method, we assume that the problem size n is fairly large and we then substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, performing such a substitution with the merge-sort recurrence equation yields the equation

$$\begin{aligned} T(n) &= 2(2T(n/2^2)+b(n/2))+bn \\ &= 2^2T(n/2^2)+2bn. \end{aligned}$$

Plugging the general equation for T in again yields the equation

$$\begin{aligned} T(n) &= 2^2(2T(n/2^3)+b(n/2^2))+2bn \\ &= 2^3T(n/2^3)+3bn. \end{aligned}$$

The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general closed-form equation (with T only appearing on the left-hand side). In the case of the merge-sort recurrence equation, the general form is

$$T(n) = 2^iT(n/2^i)+ibn.$$

Note that the general form of this equation shifts to the base case, $T(n) = b$, when $n = 2^i$, that is, when $i = \log n$, which implies

$$T(n) = bn+bn\log n.$$

In other words, $T(n)$ is $O(n\log n)$. In a general application of the iterative substitution technique, we hope that we can determine a general pattern for $T(n)$ and that we can also figure out when the general form of $T(n)$ shifts to the base case.

From a mathematical point of view, there is one point in the use of the iterative substitution technique that involves a bit of a logical "jump." This jump occurs at the point where we try to characterize the general pattern emerging from a sequence of substitutions. Often, as was the case with the merge-sort recurrence equation, this jump is quite reasonable. Other times, however, it may not be so obvious what a general form for the equation should look like. In these cases, the jump may be more dangerous. To be completely safe in making such a jump, we must fully justify the general form of the equation, possibly using induction. Combined with such a justification, the iterative substitution method is completely correct and an often useful way of characterizing recurrence equations. By the way, the colloquialism "plug-and-chug," used to describe the iterative substitution method, comes from the way this method involves "plugging" in the recursive part of an equation for $T(n)$ and then often "chugging" through a considerable amount of algebra in order to get this equation into a form where we can infer a general pattern.

The Recursion Tree

Another way of characterizing recurrence equations is to use the ***recursion tree*** method. Like the iterative substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach. In using the recursion tree method, we draw a tree R where each node represents a different substitution of the recurrence equation. Thus, each node in R has a value of the argument n of the function $T(n)$ associated with it. In addition, we associate an ***overhead*** with each node v in R, defined as the value of the nonrecursive part of the recurrence equation for v. For divide-and-conquer recurrences, the overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v. The recurrence equation is then solved by summing the overheads associated with all the nodes of R. This is commonly done by first summing values across the levels of R and then summing up these partial sums for all the levels of R.

Example 5.3: *Consider the following recurrence equation:*

$$T(n) = \begin{cases} b & \text{if } n < 3 \\ 3T(n/3) + bn & \text{if } n \geq 3. \end{cases}$$

This is the recurrence equation that we get, for example, by modifying the merge-sort algorithm so that we divide an unsorted sequence into three equal-sized sequences, recursively sort each one, and then do a three-way merge of three sorted sequences to produce a sorted version of the original sequence. In the recursion tree R for this recurrence, each internal node v has three children and has a size and an overhead associated with it, which corresponds to the time needed to merge the subproblem solutions produced by v's children. We illustrate the tree R in Figure 5.4. Note that the overheads of the nodes of each level sum to bn. Thus, observing that the depth of R is $\log_3 n$, we have that $T(n)$ is $O(n \log n)$.

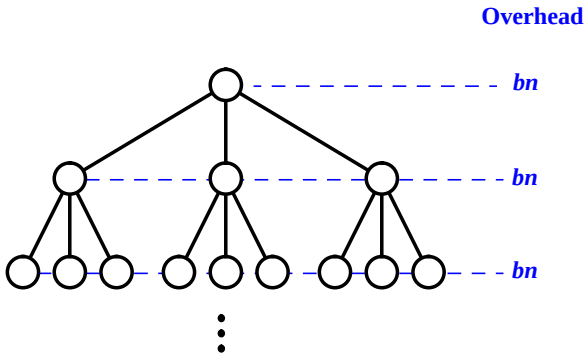


Figure 5.4: The recursion tree R used in Example 5.3, where we show the cumulative overhead of each level.

The Guess-and-Test Method

Another method for solving recurrence equations is the ***guess-and-test*** technique. This technique involves first making an educated guess as to what a closed-form solution of the recurrence equation might look like and then justifying that guess, usually by induction. For example, we can use the guess-and-test method as a kind of "binary search" for finding good upper bounds on a given recurrence equation. If the justification of our current guess fails, then it is possible that we need to use a faster-growing function, and if our current guess is justified "too easily," then it is possible that we need to use a slower-growing function. However, using this technique requires our being careful, in each mathematical step we take, in trying to justify that a certain hypothesis holds with respect to our current "guess." We explore an application of the guess-and-test method in the examples that follow.

Example 5.4: *Consider the following recurrence equation (assuming the base case $T(n) = b$ for $n < 2$):*

$$T(n) = 2T(n/2) + bn\log n.$$

This looks very similar to the recurrence equation for the merge-sort routine, so we might make the following as our first guess:

$$\text{First guess: } T(n) \leq cn\log n,$$

for some constant $c > 0$. We can certainly choose c large enough to make this true for the base case, so consider the case when $n \geq 2$. If we assume our first guess is an inductive hypothesis that is true for input sizes smaller than n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n. \end{aligned}$$

But there is no way that we can make this last line less than or equal to $cn\log n$ for $n \geq 2$. Thus, this first guess was not sufficient. Let us therefore try

$$\text{Better guess: } T(n) \leq cn\log^2 n,$$

for some constant $c > 0$. We can again choose c large enough to make this true for the base case, so consider the case when $n \geq 2$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller then n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log^2 n - 2\log n + 1) + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n, \end{aligned}$$

provided $c \geq b$. Thus, we have shown that $T(n)$ is indeed $O(n\log^2 n)$ in this case.

We must take care in using this method. Just because one inductive hypothesis for $T(n)$ does not work, that does not necessarily imply that another one proportional to this one will not work.

Example 5.5: *Consider the following recurrence equation (assuming the base case $T(n) = b$ for $n < 2$):*

$$T(n) = 2T(n/2) + \log n.$$

This recurrence is the running time for the bottom-up heap construction discussed in Section 2.4.4, which we have shown is $O(n)$. Nevertheless, if we try to prove this fact with the most straightforward inductive hypothesis, we will run into some difficulties. In particular, consider the following:

$$\text{First guess: } T(n) \leq cn,$$

for some constant $c > 0$. We can choose c large enough to make this true for the base case, certainly, so consider the case when $n \geq 2$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2(c(n/2)) + \log n \\ &= cn + \log n. \end{aligned}$$

But there is no way that we can make this last line less than or equal to cn for $n \geq 2$. Thus, this first guess was not sufficient, even though $T(n)$ is indeed $O(n)$. Still, we can show this fact is true by using

$$\text{Better guess: } T(n) \leq c(n - \log n),$$

for some constant $c > 0$. We can again choose c large enough to make this true for the base case; in fact, we can show that it is true any time $n < 8$. So consider the case when $n \geq 8$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2c((n/2) - \log(n/2)) + \log n \\ &= cn - 2c\log n + 2c + \log n \\ &= c(n - \log n) - c\log n + 2c + \log n \\ &\leq c(n - \log n), \end{aligned}$$

provided $c \geq 3$ and $n \geq 8$. Thus, we have shown that $T(n)$ is indeed $O(n)$ in this case.

The guess-and-test method can be used to establish either an upper or lower bound for the asymptotic complexity of a recurrence equation. Even so, as the above example demonstrates, it requires that we have developed some skill with mathematical induction.

The Master Method

Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively. There is, nevertheless, one method for solving divide-and-conquer recurrence equations that is quite general and does not require explicit use of induction to apply correctly. It is the ***master method***. The master method is a "cook-book" method for determining the asymptotic characterization of a wide variety of recurrence equations. Namely, it is used for recurrence equations of the form

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d, \end{cases}$$

where $d \geq 1$ is an integer constant, $a > 0$, $c > 0$, and $b > 1$ are real constants, and $f(n)$ is a function that is positive for $n \geq d$. Such a recurrence equation would arise in the analysis of a divide-and-conquer algorithm that divides a given problem into a subproblems of size at most n/b each, solves each subproblem recursively, and then "merges" the subproblem solutions into a solution to the entire problem. The function $f(n)$, in this equation, denotes the total additional time needed to divide the problem into subproblems and merge the subproblem solutions into a solution to the entire problem. Each of the recurrence equations given above uses this form, as do each of the recurrence equations used to analyze divide-and-conquer algorithms given earlier in this book. Thus, it is indeed a general form for divide-and-conquer recurrence equations.

The master method for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing $f(n)$ to the special function $n^{\log_b a}$ (we will show later why this special function is so important).

Theorem 5.6 [The Master Theorem]: *Let $f(n)$ and $T(n)$ be defined as above.*

1. *If there is a small constant $\varepsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.*
2. *If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If there are small constants $\varepsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.*

Case 1 characterizes the situation where $f(n)$ is polynomially smaller than the special function, $n^{\log_b a}$. Case 2 characterizes the situation when $f(n)$ is asymptotically close to the special function, and Case 3 characterizes the situation when $f(n)$ is polynomially larger than the special function.

We illustrate the usage of the master method with a few examples (with each taking the assumption that $T(n) = c$ for $n < d$, for constants $c \geq 1$ and $d \geq 1$).

Example 5.7: *Consider the recurrence*

$$T(n) = 4T(n/2) + n.$$

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. Thus, we are in Case 1, for $f(n)$ is $O(n^{2-\varepsilon})$ for $\varepsilon = 1$. This means that $T(n)$ is $\Theta(n^2)$ by the master method.

Example 5.8: *Consider the recurrence*

$$T(n) = 2T(n/2) + n\log n,$$

which is one of the recurrences given above. In this case, $n^{\log_b a} = n^{\log_2 2} = n$. Thus, we are in Case 2, with $k = 1$, for $f(n)$ is $\Theta(n\log n)$. This means that $T(n)$ is $\Theta(n\log^2 n)$ by the master method.

Example 5.9: *Consider the recurrence*

$$T(n) = T(n/3) + n,$$

which is the recurrence for a geometrically decreasing summation that starts with n. In this case, $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for $f(n)$ is $\Omega(n^{0+\varepsilon})$, for $\varepsilon = 1$, and $af(n/b) = n/3 = (1/3)f(n)$. This means that $T(n)$ is $\Theta(n)$ by the master method.

Example 5.10: *Consider the recurrence*

$$T(n) = 9T(n/3) + n^{2.5}.$$

In this case, $n^{\log_b a} = n^{\log_3 9} = n^2$. Thus, we are in Case 3, since $f(n)$ is $\Omega(n^{2+\varepsilon})$ (for $\varepsilon = 1/2$) and $af(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2} f(n)$. This means that $T(n)$ is $\Theta(n^{2.5})$ by the master method.

Example 5.11: *Finally, consider the recurrence*

$$T(n) = 2T(n^{1/2}) + \log n.$$

Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable $k = \log n$, which lets us write

$$T(n) = T(2^k) = 2T(2^{k/2}) + k.$$

Substituting into this the equation $S(k) = T(2^k)$, we get that

$$S(k) = 2S(k/2) + k.$$

Now, this recurrence equation allows us to use master method, which specifies that $S(k)$ is $O(k\log k)$. Substituting back for $T(n)$ implies $T(n)$ is $O(\log n \log\log n)$.

Rather than rigorously prove Theorem 5.6, we instead discuss the justification behind the master method at a high level.

If we apply the iterative substitution method to the general divide-and-conquer recurrence equation, we get

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ &= a(aT(n/b^2) + f(n/b)) + f(n) = a^2T(n/b^2) + af(n/b) + f(n) \\ &= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\ &\vdots \\ &= a^{\log_b n}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) \\ &= n^{\log_b a}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i), \end{aligned}$$

where the last substitution is based on the identity $a^{\log_b n} = n^{\log_b a}$. Indeed, this equation is where the special function comes from. Given this closed-form characterization of $T(n)$, we can intuitively see how each of the three cases is derived. Case 1 comes from the situation when $f(n)$ is small and the first term above dominates. Case 2 denotes the situation when each of the terms in the above summation is proportional to the others, so the characterization of $T(n)$ is $f(n)$ times a logarithmic factor. Finally, Case 3 denotes the situation when the first term is smaller than the second and the summation above is a sum of geometrically decreasing terms that start with $f(n)$; hence, $T(n)$ is itself proportional to $f(n)$.

The proof of Theorem 5.6 formalizes this intuition, but instead of giving the details of this proof, we present two applications of the master method below.

5.2.2 Integer Multiplication

We consider, in this subsection, the problem of multiplying ***big integers***, that is, integers represented by a large number of bits that cannot be handled directly by the arithmetic unit of a single processor. Multiplying big integers has applications to data security, where big integers are used in encryption schemes.

Given two big integers I and J represented with n bits each, we can easily compute $I+J$ and $I-J$ in $O(n)$ time. Efficiently computing the product $I \cdot J$ using the common grade-school algorithm requires, however, $O(n^2)$ time. In the rest of this section, we show that by using the divide-and-conquer technique, we can design a subquadratic-time algorithm for multiplying two n-bit integers.

Let us assume that n is a power of two (if this is not the case, we can pad I and J with 0's). We can therefore divide the bit representations of I and J in half, with one half representing the ***higher-order*** bits and the other representing the ***lower-order*** bits. In particular, if we split I into I_h and I_l and J into J_h and J_l, then

$$\begin{aligned} I &= I_h 2^{n/2} + I_l, \\ J &= J_h 2^{n/2} + J_l. \end{aligned}$$

Also, observe that multiplying a binary number I by a power of two, 2^k, is trivial—it simply involves shifting left (that is, in the higher-order direction) the number I by k bit positions. Thus, provided a left-shift operation takes constant time, multiplying an integer by 2^k takes $O(k)$ time.

Let us focus on the problem of computing the product $I \cdot J$. Given the expansion of I and J above, we can rewrite $I \cdot J$ as

$$I \cdot J = (I_h 2^{n/2} + I_l) \cdot (J_h 2^{n/2} + J_l) = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l.$$

Thus, we can compute $I \cdot J$ by applying a divide-and-conquer algorithm that divides the bit representations of I and J in half, recursively computes the product four products of $n/2$ bits each (as described above), and then merges the solutions to these subproducts in $O(n)$ time using addition and multiplication by powers of two. We can terminate the recursion when we get down to the multiplication of two 1-bit numbers, which is trivial. This divide-and-conquer algorithm has a running time that can be characterized by the following recurrence (for $n \geq 2$):

$$T(n) = 4T(n/2) + cn,$$

for some constant $c > 0$. We can then apply the master theorem to note that the special function $n^{\log_b a} = n^{\log_2 4} = n^2$ in this case; hence, we are in Case 1 and $T(n)$ is $\Theta(n^2)$. Unfortunately, this is no better than the grade-school algorithm.

The master method gives us some insight into how we might improve this algorithm. If we can reduce the number of recursive calls, then we will reduce the complexity of the special function used in the master theorem, which is currently the dominating factor in our running time. Fortunately, if we are a little more clever in how we define subproblems to solve recursively, we can in fact reduce the number of recursive calls by one. In particular, consider the product

$$(I_h - I_l) \cdot (J_l - J_h) = I_h J_l - I_l J_l - I_h J_h + I_l J_h.$$

This is admittedly a strange product to consider, but it has an interesting property. When expanded out, it contains two of the products we want to compute (namely, $I_h J_l$ and $I_l J_h$) and two products that can be computed recursively (namely, $I_h J_h$ and $I_l J_l$). Thus, we can compute $I \cdot J$ as follows:

$$I \cdot J = I_h J_h 2^n + [(I_h - I_l) \cdot (J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l.$$

This computation requires the recursive computation of three products of $n/2$ bits each, plus $O(n)$ additional work. Thus, it results in a divide-and-conquer algorithm with a running time characterized by the following recurrence equation (for $n \geq 2$):

$$T(n) = 3T(n/2) + cn,$$

for some constant $c > 0$.

Theorem 5.12: *We can multiply two n-bit integers in $O(n^{1.585})$ time.*

Proof: We apply the master theorem with the special function $n^{\log_b a} = n^{\log_2 3}$; hence, we are in Case 1 and $T(n)$ is $\Theta(n^{\log_2 3})$, which is itself $O(n^{1.585})$. ∎

Using divide-and-conquer, we have designed an algorithm for integer multiplication that is asymptotically faster than the straightforward quadratic-time method. We can actually do even better than this, achieving a running time that is "almost" $O(n \log n)$, by using a more complex divide-and-conquer algorithm called the ***fast Fourier transform***, which we discuss in Section 10.4.

5.2.3 Matrix Multiplication

Suppose we are given two $n \times n$ matrices X and Y, and we wish to compute their product $Z = XY$, which is defined so that

$$Z[i,j] = \sum_{k=0}^{e-1} X[i,k] \cdot Y[k,j],$$

which is an equation that immediately gives rise to a simple $O(n^3)$ time algorithm.

Another way of viewing this product is in terms of submatrices. That is, let us assume that n is a power of two and let us partition X, Y, and Z each into four $(n/2) \times (n/2)$ matrices, so that we can rewrite $Z = XY$ as

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Thus,

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH. \end{aligned}$$

We can use this set of equations in a divide-and-conquer algorithm that computes $Z = XY$ by computing I, J, K, and L from the subarrays A through G. By the above equations, we can compute I, J, K, and L from the eight recursively computed matrix products on $(n/2) \times (n/2)$ subarrays, plus four additions that can be done in $O(n^2)$ time. Thus, the above set of equations give rise to a divide-and-conquer algorithm whose running time $T(n)$ is characterized by the recurrence

$$T(n) = 8T(n/2) + bn^2,$$

for some constant $b > 0$. Unfortunately, this equation implies that $T(n)$ is $O(n^3)$ by the master theorem; hence, it is no better than the straightforward matrix multiplication algorithm.

Interestingly, there is an algorithm known as ***Strassen's Algorithm***, that organizes arithmetic involving the subarrays A through G so that we can compute I, J, K, and L using just seven recursive matrix multiplications. It is somewhat mysterious how Strassen discovered these equations, but we can easily verify that they work correctly.

We begin Strassen's Algorithm by defining seven submatrix products:

$$\begin{aligned} S_1 &= A(F-H) \\ S_2 &= (A+B)H \\ S_3 &= (C+D)E \\ S_4 &= D(G-E) \\ S_5 &= (A+D)(E+H) \\ S_6 &= (B-D)(G+H) \\ S_7 &= (A-C)(E+F). \end{aligned}$$

Given these seven submatrix products, we can compute I as

$$\begin{aligned} I &= S_5+S_6+S_4-S_2 \\ &= (A+D)(E+H)+(B-D)(G+H)+D(G-E)-(A+B)H \\ &= AE+DE+AH+DH+BG-DG+BH-DH+DG-DE-AH-BH \\ &= AE+BG. \end{aligned}$$

We can compute J as

$$\begin{aligned} J &= S_1+S_2 \\ &= A(F-H)+(A+B)H \\ &= AF-AH+AH+BH \\ &= AF+BH. \end{aligned}$$

We can compute K as

$$\begin{aligned} K &= S_3+S_4 \\ &= (C+D)E+D(G-E) \\ &= CE+DE+DG-DE \\ &= CE+DG. \end{aligned}$$

Finally, we can compute L as

$$\begin{aligned} L &= S_1-S_7-S_3+S_5 \\ &= A(F-H)-(A-C)(E+F)-(C+D)E+(A+D)(E+H) \\ &= AF-AH-AE+CE-AF+CF-CE-DE+AE+DE+AH+DH \\ &= CF+DH. \end{aligned}$$

Thus, we can compute $Z = XY$ using seven recursive multiplications of matrices of size $(n/2)\times(n/2)$. Thus, we can characterize the running time $T(n)$ as

$$T(n) = 7T(n/2)+bn^2,$$

for some constant $b > 0$. Thus, by the master theorem, we have the following:

Theorem 5.13: *We can multiply two $n\times n$ matrices in $O(n^{\log 7})$ time.*

Thus, with a fair bit of additional complication, we can perform the multiplication for $n\times n$ matrices in time $O(n^{2.808})$, which is $o(n^3)$ time. As admittedly complicated as Strassen's matrix multiplication is, there are actually much more complicated matrix multiplication algorithms, with running times as low as $O(n^{2.376})$.

5.3 Dynamic Programming

In this section, we discuss the ***dynamic programming*** algorithm-design technique. This technique is similar to the divide-and-conquer technique, in that it can be applied to a wide variety of different problems. Conceptually, the dynamic programming technique is different from divide-and-conquer, however, because the divide-and-conquer technique can be easily explained in a sentence or two, and can be well illustrated with a single example. Dynamic programming takes a bit more explaining and multiple examples before it can be fully appreciated.

The extra effort needed to fully appreciate dynamic programming is well worth it, though. There are few algorithmic techniques that can take problems that seem to require exponential time and produce polynomial-time algorithms to solve them. Dynamic programming is one such technique. In addition, 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.

5.3.1 Matrix Chain-Product

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)th entry of the product as

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

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 primitive (that is, scalar) multiplications in each parenthesization, however, as is illustrated in the following example.

Example 5.14: *Let B be a 2×10-matrix, let C be a 10×50-matrix, and let D be a 50×20-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 nth ***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

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

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 $N_{i,j}$ 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 solutions 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 5.5.

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 5.5: Dynamic programming algorithm for the matrix chain-product problem.

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 5.15: *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.*

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 5.6, we illustrate the way the dynamic programming solution to the matrix chain-product problem fills in the array N.

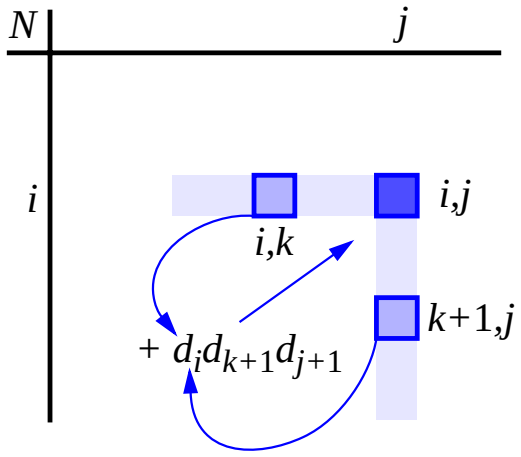


Figure 5.6: Illustration of the way the matrix chain-product dynamic-programming algorithm fills in the array N.

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

5.3.2 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 improves the efficiency of a dynamic programming algorithm that stores solutions to subproblems.

Now that we have given the general components of a dynamic programming algorithm, we next give another example of its use.

5.3.3 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:

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

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 5.1.1, and we study how knapsack problems arise in the context of Internet auctions in Exercise R-5.12.

A First Attempt at Characterizing Subproblems

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 5.7, if we use this characterization for subproblems, then an optimal solution to the global problem may actually contain a suboptimal subproblem.

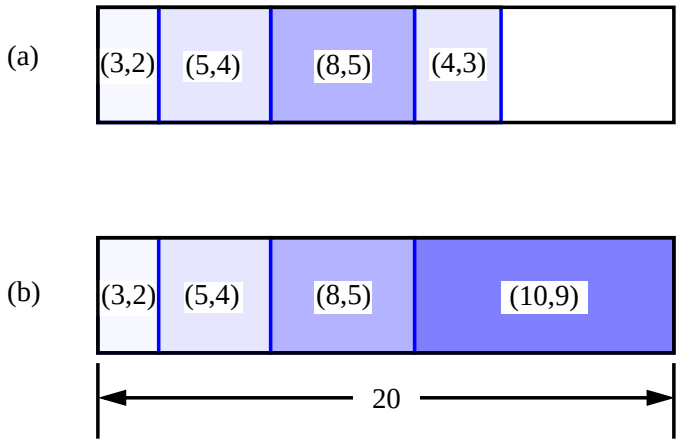


Figure 5.7: An example showing that our first approach to defining a knapsack subproblem does not work. The set S consists of five items denoted by the the (***weight***, ***benefit***) pairs $(3,2)$, $(5,4)$, $(8,5)$, $(4,3)$, and $(10,9)$. The maximum total weight is $W = 20$: (a) best solution with the first four items; (b) best solution with the first five items. We shade each item in proportion to its benefit.

A Better Subproblem Characterization

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 ***exactly*** 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 w is either the best subset of S_{k-1} that has total weight w or the best subset of S_{k-1} that has total weight $w-w_k$ plus the 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 the optimal way of summing exactly w to weight 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 5.8 (01Knapsack).

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 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 5.8: Dynamic programming algorithm for solving the 0-1 knapsack problem.

Analyzing the 0-1 Knapsack Dynamic Programming Algorithm

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 5.16: *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 5.8 (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. ■

Pseudo-Polynomial-Time Algorithms

In addition to being another useful application of the dynamic programming technique, Theorem 5.16 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. In practice, such algorithms should run much faster than any brute-force algorithm, but it is not correct to say they are true polynomial-time algorithms. In fact, there is a theory known as ***NP-completeness***, which is discussed in Chapter 13, that states that it is very unlikely that anyone will ever find a true polynomial-time algorithm for the 0-1 knapsack problem.

Elsewhere in this book, we give additional applications of the dynamic programming technique for computing reachability in a directed graph (Section 6.4.2) and for testing the similarity of two strings (Section 9.4).

5.4 Exercises

Reinforcement

R-5.1 Let $S = \{a,b,c,d,e,f,g\}$ be a collection of objects with benefit-weight values as follows: a: $(12,4)$, b: $(10,6)$, c: $(8,5)$, d: $(11,7)$, e: $(14,3)$, f: $(7,1)$, g: $(9,6)$. What is an optimal solution to the fractional knapsack problem for S assuming we have a sack that can hold objects with total weight 18? Show your work.

R-5.2 Describe how to implement the TaskSchedule method to run in $O(n \log n)$ time.

R-5.3 Suppose we are given a set of tasks specified by pairs of the start times and finish times as $T = \{(1,2),(1,3),(1,4),(2,5),(3,7),(4,9),(5,6),(6,8),(7,9)\}$. Solve the task scheduling problem for this set of tasks.

R-5.4 Characterize each of the following recurrence equations using the master method (assuming that $T(n) = c$ for $n < d$, for constants $c > 0$ and $d \geq 1$).

a. $T(n) = 2T(n/2) + \log n$
b. $T(n) = 8T(n/2) + n^2$
c. $T(n) = 16T(n/2) + (n \log n)^4$
d. $T(n) = 7T(n/3) + n$
e. $T(n) = 9T(n/3) + n^3 \log n$

R-5.5 Use the divide-and-conquer algorithm, from Section 5.2.2, to compute $10110011 \cdot 10111010$ in binary. Show your work.

R-5.6 Use Strassen's matrix multiplication algorithm to multiply the matrices

$$X = \begin{pmatrix} 3 & 2 \\ 4 & 8 \end{pmatrix} \qquad \text{and} \qquad Y = \begin{pmatrix} 1 & 5 \\ 9 & 6 \end{pmatrix}.$$

R-5.7 A complex number $a + b\mathbf{i}$, where $\mathbf{i} = \sqrt{-1}$, can be represented by the pair (a,b). Describe a method performing only three real-number multiplications to compute the pair (e,f) representing the product of $a + b\mathbf{i}$ and $c + d\mathbf{i}$.

R-5.8 Boolean matrices are matrices such that each entry is 0 or 1, and matrix multiplication is performed by using AND for $\cdot$ and OR for $+$. Suppose we are given two $n \times n$ random Boolean matrices A and B, so that the probability that any entry in either is 1, is $1/k$. Show that if k is a constant, then there is an algorithm for multiplying A and B whose expected running time is $O(n^2)$. What if k is n?

R-5.9 What is the best way to multiply a chain of matrices with dimensions that are 10×5, 5×2, 2×20, 20×12, 12×4, and 4×60? Show your work.

R-5.10 Design an efficient algorithm for the matrix chain multiplication problem that outputs a fully parenthesized expression for how to multiply the matrices in the chain using the minimum number of operations.

R-5.11 Solve Exercise R-5.1 for the 0-1 knapsack problem.

R-5.12 Sally is hosting an Internet auction to sell n widgets. She receives m bids, each of the form "I want k_i widgets for d_i dollars," for $i = 1,2,\ldots,m$. Characterize her optimization problem as a knapsack problem. Under what conditions is this a 0-1 versus fractional problem?

Creativity

C-5.1 A native Australian named Anatjari wishes to cross a desert carrying only a single water bottle. He has a map that marks all the watering holes along the way. Assuming he can walk k miles on one bottle of water, design an efficient algorithm for determining where Anatjari should refill his bottle in order to make as few stops as possible. Argue why your algorithm is correct.

C-5.2 Consider the single ***machine scheduling*** problem where we are given a set T of tasks specified by their start times and finish times, as in the task scheduling problem, except now we have only one machine and we wish to maximize the number of tasks that this single machine performs. Design a greedy algorithm for this single machine scheduling problem and show that it is correct. What is the running time of this algorithm?

C-5.3 Describe an efficient greedy algorithm for making change for a specified value using a minimum number of coins, assuming there are four denominations of coins (called quarters, dimes, nickels, and pennies), with values 25, 10, 5, and 1, respectively. Argue why your algorithm is correct.

C-5.4 Give an example set of denominations of coins so that a greedy change making algorithm will not use the minimum number of coins.

C-5.5 In the ***art gallery guarding*** problem we are given a line L that represents a long hallway in an art gallery. We are also given a set $X = \{x_0, x_1, \ldots, x_{n-1}\}$ of real numbers that specify the positions of paintings in this hallway. Suppose that a single guard can protect all the paintings within distance at most 1 of his or her position (on both sides). Design an algorithm for finding a placement of guards that uses the minimum number of guards to guard all the paintings with positions in X.

C-5.6 Design a divide-and-conquer algorithm for finding the minimum and the maximum element of n numbers using no more than $3n/2$ comparisons.

C-5.7 Given a set P of n teams in some sport, a ***round-robin tournament*** is a collection of games in which each team plays each other team exactly once. Design an efficient algorithm for constructing a round-robin tournament assuming n is a power of 2.

C-5.8 Let a set of intervals $S = \{[a_0, b_0], [a_1, b_1], \ldots, [a_{n-1}, b_{n-1}]\}$ of the interval $[0,1]$ be given, with $0 \leq a_i < b_i \leq 1$, for $i = 0, 1, \ldots, n-1$. Suppose further that we assign a height h_i to each interval $[a_i, b_i]$ in S. The ***upper envelope*** of S is defined to be a list of pairs $[(x_0, c_0), (x_1, c_1), (x_2, c_2), \ldots, (x_m, c_m), (x_{m+1}, 0)]$, with $x_0 = 0$ and $x_{m+1} = 1$, and ordered by x_i values, such that, for each subinterval $s = [x_i, x_{i+1}]$ the height of the highest interval in S containing s is c_i, for $i = 0, 1, \ldots, m$. Design an $O(n \log n)$-time algorithm for computing the upper envelope of S.

C-5.9 How can we modify the dynamic programming algorithm from simply computing the best benefit value for the 0-1 knapsack problem to computing the assignment that gives this benefit?

C-5.10 Suppose we are given a collection $A = \{a_1, a_2, \ldots, a_n\}$ of n positive integers that add up to N. Design an $O(nN)$-time algorithm for determining whether there is a subset $B \subset A$, such that $\sum_{a_i \in B} a_i = \sum_{a_i \in A-B} a_i$.

C-5.11 Let P be a convex polygon (Section 12.5.1). A ***triangulation*** of P is an addition of diagonals connecting the vertices of P so that each interior face is a triangle. The ***weight*** of a triangulation is the sum of the lengths of the diagonals. Assuming that we can compute lengths and add and compare them in constant time, give an efficient algorithm for computing a minimum-weight triangulation of P.

C-5.12 A ***grammar*** G is a way of generating strings of "terminal" characters from a nonterminal symbol S, by applying simple substitution rules, called ***productions***. If $B \rightarrow \beta$ is a production, then we can convert a string of the form $\alpha B\gamma$ into the string $\alpha\beta\gamma$. A grammar is in ***Chomsky normal form*** if every production is of the form "$A \rightarrow BC$" or "$A \rightarrow a$," where A, B, and C are nonterminal characters and a is a terminal character. Design an $O(n^3)$-time dynamic programming algorithm for determining if string $x = x_0x_1 \cdots x_{n-1}$ can be generated from start symbol S.

C-5.13 Suppose we are given an n-node rooted tree T, such that each node v in T is given a weight $w(v)$. An ***independent set*** of T is a subset S of the nodes of T such that no node in S is a child or parent of any other node in S. Design an efficient dynamic programming algorithm to find the maximum-weight independent set of the nodes in T, where the weight of a set of nodes is simply the sum of the weights of the nodes in that set. What is the running time of your algorithm?

Projects

P-5.1 Design and implement a big integer package supporting the four basic arithmetic operations.

P-5.2 Implement a system for efficiently solving knapsack problems. Your system should work for either fractional or 0-1 knapsack problems. Perform an experimental analysis to test the efficiency of your system.

Chapter Notes

The term "greedy algorithm" was coined by Edmonds [64] in 1971, although the concept existed before then. For more information about the greedy method and the theory that supports it, which is known as matroid theory, please see the book by Papadimitriou and Steiglitz [164].

The divide-and-conquer technique is a part of the folklore of data structure and algorithm design. The master method for solving divide-and-conquer recurrences traces its origins to a paper by Bentley, Haken, and Saxe [30]. The divide-and-conquer algorithm for multiplying two large integers in $O(n^{1.585})$ time is generally attributed to the Russians Karatsuba and Ofman [111]. The asymptotically fastest known algorithm for multiplying two n-digit numbers is an FFT-based algorithm by Schönhage and Strassen [181] that runs in $O(n \log n \log\log n)$ time.

Dynamic programming was developed in the operations research community and formalized by Bellman [26]. The matrix chain-product solution we described is due to Godbole [78]. The asymptotically fastest method is due to Hu and Shing [101, 102]. The dynamic programming algorithm for the knapsack problem is found in the book by Hu [100]. Hirchsberg [95] shows how to solve the longest common substring problem in the same time given above, but with linear space (see also [56]).