

4. Divide and Conquer

In this chapter, we explore the divide-and-conquer strategy for algorithm design. The divide-and-conquer strategy has its roots from the colonial-wars tactics employed by army generals to defeat their opposing armies. To apply the strategy, we *divide* an instance of a problem into two (or more) smaller-sized instances whose solutions will be *combined* to obtain the solution to the original problem. The smaller instances should be instances of the original problem, and they are solved by repeated application of divide-and-combine, unless the instance is small enough where a direct solution is possible.

A divide-and-conquer algorithm is readily expressible into a recursive procedure, as shown by Listing 4.1. The general case consists of three steps, which are executed in sequence: *divide*, *conquer*, and *combine*.

```
Solution-Form Solve(P,n)
{ if n is small enough then solve P directly and return the solution;
  else
  { // 1. Divide Step
    Divide P into two subproblems P1 and P2 of size m1 and m2 each, where m1 ≈ m2 ≈ n/2
    // 2. Conquer Step
    S1 ← Solve(P1,m1);
    S2 ← Solve(P2,m1);
    // 3. Combine step
    S ← Combine(S1,S2);
    return S;
  }
}
```

Listing 4.1 A generic structure of a divide-and-conquer algorithm.

4.1 Solving Recurrence Equations

The use of recurrence equations is fundamental to the analysis of recursive algorithms. A recurrence equation is a function that expresses the running time for an input of size n in terms of some expression on n and the same function of inputs of smaller sizes. We let $T(n)$ denote the (best, worst, or average) case running time, or the number of barometer operations, on inputs of size n . If n is sufficiently small (i.e., $n \leq n_0$ for some positive constant n_0), then the problem is trivial, and no further division is necessary. In this case, the solution runs in constant time: $T(n) = c$. This is the point at which the recursion “bottoms out”. If $n > n_0$, we divide the problem into a subproblems, each of size n/b , where $a \geq 1$ and $b > 1$. Suppose our algorithm takes time $D(n)$ to divide the original problem instance into subinstances, and time $C(n)$ to combine the solutions to these instances into a solution to the original problem instance. Thus, we obtain the following recurrence equation for $T(n)$:

$$T(n) = \begin{cases} c & n \leq n_0 \\ aT(n/b) + D(n) + C(n) & n > n_0 \end{cases}$$

Here are some examples (in the examples, $T(n)$ denotes the running time on an input of size n):

Mergesort: To sort an array of size n , we sort the left half, sort the right half, and then merge the two halves. We can do the merge in linear time (i.e., cn for some positive constant c). So, if $T(n)$ denotes the running time on an input of size n , we end up with the recurrence $T(n) = 2T(n/2) + cn$. This can also be expressed as $T(n) = 2T(n/2) + O(n)$.

Selection Sort: In selection sort, we find the smallest element in the input sequence and swap with the leftmost element and then recursively sort the remainder (less the leftmost element) of the sequence. This leads to the recurrence $T(n) = cn + T(n-1)$.

Polynomial Multiplication: The straightforward divide-and-conquer algorithm to multiply two polynomials of degree n leads to $T(n) = 4T(n/2) + cn$. However, a clever rearrangement of the terms (see Section 4.6) improves this to $T(n) = 3T(n/2) + cn$.

4.1.1 The Substitution Method

One of the simplest techniques to find a closed (nonrecursive) formula for a given recurrence $T(n)$ is to use repeated substitution to get rid of the recursive term. This process is iterative in nature and can be performed in one of two ways: *forward substitution* or *backward substitution*.

Forward Substitution

Using forward substitution, we generate the terms of the recurrence in a forward way, starting with the term(s) given by the base recurrence(s). In the process, we try to identify a pattern that can be expressed by a closed formula.

Divide and Conquer

Example 4.1 We use forward substitution to solve the recurrence: $T(1)=1$; $T(n)=2T(n-1)+1$ for $n > 1$.

By a process of repeated substitution we get,

$$\begin{aligned}T(1) &= 1 \\T(2) &= 2 T(1) + 1 = 2(1)+1= 3 \\T(3) &= 2 T(2) + 1 = 2(3) +1 = 7 \\T(4) &= 2 T(3) + 1 = 2(7) + 1 = 15\end{aligned}$$

We note that successive terms are simply the consecutive powers of 2 less 1. Thus, we claim that, $T(n)=2^n - 1$ for all $n \geq 1$. This claim can be proven by induction.

Base Step: $T(1) = 2^{1-1} = 1$ (and it is given that $T(1)=1$).

Induction Step: Assume $T(k)=2^k - 1$ is true for $1 \leq k < n$; we show that $T(n)=2^n - 1$ as follows:

$$T(n) = 2T(n-1) + 1 = 2[2^{n-1}-1] + 1 = 2^n - 1.$$

The method of forward substitution works in limited cases because it is usually hard to recognize a pattern from the first few terms.

Backward Substitution

Using backward substitution, we generate the terms of the recurrence in a backward way, starting with the term given by the general recurrence. In every step, we apply the recurrence equation to a recursive term appearing in the RHS of the equation for $T(n)$. Following every substitution step, we collect like terms together to help identify a pattern that can be expressed by a closed formula.

Example 4.2 We use backward substitution to solve the recurrence: $T(1)=0$; $T(n)=3T(n/2)+n$ for $n > 1$.

We start with $T(n) = 3T(n/2) + n$. Next, we replace $T(n/2)$ by $3T(n/4)+n/2$ to get the following:

$$T(n) = 3 [3T(n/4) + n/2] + n = 9T(n/4) + 3n/2 + n$$

Next, we replace $T(n/4)$ by $3T(n/8) + n/4$ to get

$$T(n) = 9 [3T(n/8) + n/4] + 3 n/2 + n = 27T(n/8) + 9n/4 + 3n/2 + n$$

In general,

$$T(n) = 3^i T(n/2^i) + 3^{i-1} n/2^{i-1} + \dots + 3n/2 + n$$

To reach $T(1)$, we assume $n=2^k$ and let $i=k$. Thus, we get

$$T(n) = 3^k T(1) + 3^{k-1} n/2^{k-1} + \dots + 3n/2 + n = n [(3/2)^{k-1} + \dots + (3/2)^1 + (3/2)^0]$$

Finally, we utilize the formula for a geometric progression (see Example 3.2) to get,

$$T(n) = n [((3/2)^k - 1) / ((3/2) - 1)] = 2[3^{\log_2 n} - n] = 2 [n^{\log_2 3} - n].$$

4.1.2 The Induction Method

A useful technique to solve recurrences is to make a guess and then use induction to prove that the guess is correct. Here is an example.

Example 4.3 We find a solution to the recurrence $T(1) = 0$; $T(n) = kT(n/k) + cn$. We assume that n is a power of k . First, let us construct a few initial values of $T(n)$, starting from the given $T(1)$ and successively applying the recurrence:

$$\begin{aligned}T(1) &= 0 \\T(k) &= kT(1) + ck = ck \\T(k^2) &= kT(k) + ck^2 = 2ck^2 \\T(k^3) &= kT(k^2) + ck^3 = 3ck^3\end{aligned}$$

This leads us to guess (claim) that the solution to the recurrence is given by $T(k^m) = cmk^m$. We use induction on m to prove our guess. The proof is as follows.

Base Step: $T(k^0) = c(0)k^0 = 0$, which is consistent with $T(1) = 0$ (the boundary condition of the recurrence).

Induction Step: We assume that the claim holds for $n = k^{m-1}$ (i.e., $T(k^{m-1}) = c(m-1)k^{m-1}$) and show that it holds for $n = k^m$. Then

$$\begin{aligned}T(k^m) &= kT(k^{m-1}) + ck^m && \text{(using the recurrence definition)} \\&= k [c(m-1)k^{m-1}] + ck^m && \text{(using the induction hypothesis to substitute for } T(k^{m-1})) \\&= c(m-1+1)k^m = cmk^m\end{aligned}$$

Thus, we conclude that $T(n) = cn \log_k n$.

Example 4.4 Use induction to find the O-order of the recurrence $T(n) = 2T(n/2) + n$.

We guess that $T(n) = O(n \log n)$. Thus, we guess that $T(n) \leq cn \log n$ (for some positive constant c) is a solution to the recurrence. For the induction step, we assume $T(k) \leq ck \log k$ is true for $k < n$, and show $T(n) \leq cn \log n$. We start with the recurrence $T(n) = 2T(n/2) + n$ and then substitute for $T(n/2)$ using the induction hypothesis to get the following:

$$\begin{aligned}T(n) &= 2T(n/2) + n \\&\leq 2 [(cn/2) \log n/2] + n \\&= cn \log n/2 + n \\&= cn \log n - cn \log 2 + n \\&= cn \log n - cn + n \\&\leq cn \log n \text{ (when } c \geq 1)\end{aligned}$$

Note that we have solved the recurrence using induction without properly defining a base case. For the O-order, we are looking for an n_0 such that the inequality is valid for all $n \geq n_0$. We can pick n_0 ourselves. Note that the base case of the recurrence (not actually given here) can be *different* from the base case of the induction. Suppose the recurrence base-case is $T(1) = 1$. This is at odds with our equation because $T(1) \leq c(1 \log 1) = 0$. We know that, assuming $T(1) = 1$, $T(2) = 4$ and $T(3) = 5$, which do not contradict our equation, and that $T(n)$, for $n > 3$, does not depend directly on $T(1)$, so let our bases cases be $T(2)$ and $T(3)$ (i.e., $n_0 = 2$).

Divide and Conquer

A Common Mistake

We must be careful when using asymptotic notation. Here is an erroneous solution to the preceding example. Guess that $T(n) = O(n)$. Thus, $T(n) \leq cn$. Assuming $T(k) \leq ck$ is true for $k < n$, we show $T(n) \leq cn$, as follows:

$$\begin{aligned} T(n) &= 2 T(n/2) + n \\ &= 2 [cn/2] + n \\ &= cn + n \\ &= O(n) \leftarrow \text{wrong!} \end{aligned}$$

Why? Because we have not proven the *exact* form of the induction hypothesis (i.e., $T(n) \leq cn$). The constants do not matter in the end, but we can not drop them (or change them) during the proof.

4.1.3 The Characteristic Equation Method

Certain classes of recurrence have canned formulas for their solutions. In this section, we consider two such classes, which occur quite frequently.

Homogeneous Linear Recurrences

A *homogeneous linear recurrence* (with constant coefficients) is of the following form:

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots + a_k T(n-k),$$

with k initial conditions (i.e., values for $T(0), \dots, T(k-1)$).

The characteristic equation for this recurrence is as follows:

$$x^k - a_1 x^{k-1} - a_2 x^{k-2} - \dots - a_k = 0.$$

Case 1: Distinct Roots

If r_1, r_2, \dots, r_k are distinct roots of the characteristic equation, the recurrence has a solution of the following form:

$$T(n) = c_1 r_1^n + c_2 r_2^n + \dots + c_k r_k^n$$

for some choice of constants c_1, c_2, \dots, c_k . These constants can be determined from the k initial conditions.

Example 4.5 Solve the recurrence $T(n) = 7T(n-1) - 6T(n-2)$; $T(0) = 2, T(1) = 7$.

Characteristic equation: $x^2 - 7x + 6 = (x-6)(x-1) = 0$; the roots are $r_1 = 6$ and $r_2 = 1$.

General form of the solution: $T(n) = c_1 6^n + c_2 (1)^n = c_1 6^n + c_2$

Constraints for the constants:

$$\begin{aligned} T(0) = 2 &= c_1 + c_2 \\ T(1) = 7 &= 6c_1 + c_2 \end{aligned}$$

Solution for the constants: $c_1 = 1$ and $c_2 = 1$.

Solution for the recurrence: $T(n) = 6^n + 1$.

Case 2: Repeated Roots

Each root generates one term and the solution is a linear combination (i.e., a sum) of these terms. Whereas a nonrepeated root r generates the term r^n , a root r with multiplicity k generates the terms (along with their respective constants): $r^n, nr^n, n^2r^n, \dots, n^{k-1}r^n$.

Example 4.6 Solve the recurrence $T(n) = 3T(n-1) - 4T(n-3)$; $T(0) = -4$, $T(1) = 2$, $T(2) = 6$.

Characteristic equation: $x^3 - 3x^2 + 4 = (x+1)(x-2)^2 = 0$; the roots are $r_1 = -1$ and $r_2 = 2$ of multiplicity = 2.

General form of the solution: $T(n) = c_1(-1)^n + c_22^n + c_3n2^n$.

Constraints for the constants:

$$T(0) = -4 = c_1 + c_2$$

$$T(1) = 2 = -c_1 + 2c_2 + 2c_3$$

$$T(2) = 6 = c_1 + 4c_2 + 8c_3$$

Solution for the constants: $c_1 = -2$, $c_2 = -2$ and $c_3 = 2$.

Solution for the recurrence: $T(n) = -2(-1)^n - 2(2^n) + 2n(2^n) = -2(-1)^n + 2(n-1)2^n$.

The next example illustrates the *change of variable* technique.

Example 4.7 Solve the recurrence $T(n) = 3T(n/2)$; $T(1) = 1$. Assume $n = 2^k$.

The recurrence can be written as $T(2^k) = 3T(2^{k-1})$. This is not of the form that directly fits a linear recurrence. However, by change of variable, we let $t_k = T(2^k)$ and then the previous recurrence can be written as $t_k = 3t_{k-1}$.

Characteristic equation: $x-3 = 0$.

General form of the solution: $t_k = c_13^k$.

Constraints for the constants: $T(1)$ corresponds to t_0 ; thus, $t_0 = 1 = c_1$.

Solution for the recurrence: $t_k = 3^k \Rightarrow T(n) = 3^{\log_2 n} = n^{\log_2 3}$.

Divide and Conquer

Nonhomogeneous Linear Recurrences

A *nonhomogeneous linear recurrence* (with constant coefficients) is of the form:

$$T(n) = a_1T(n-1) + a_2T(n-2) + \dots + a_kT(n-k) + F(n).$$

There is no known general method for solving nonhomogeneous linear recurrences except in few special cases. We will consider the case of $F(n) = p(n)C^n$, where C is a constant and $p(n)$ is a polynomial in terms of n . The (general) solution is a sum of the *homogeneous solution* T_h and a *particular solution* T_p . This general solution is obtained using the following steps:

1. The homogeneous solution T_h is obtained using the approach outlined previously.
2. The particular solution T_p is a solution to the full recurrence that need not be consistent with the boundary conditions. For $F(n) = (b_in^i + b_{i-1}n^{i-1} + \dots + b_0)C^n$, there are two cases to consider, depending on whether C is a root of the characteristic equation of the homogeneous recurrence:

Case 1: C is not a root:

The particular solution $T_p(n)$ is of the form $T_p(n) = (c_in^i + c_{i-1}n^{i-1} + \dots + c_0)C^n$.

Case 2: C is a root with multiplicity of m :

The particular solution $T_p(n)$ is of the form $T_p(n) = n^m (c_in^i + c_{i-1}n^{i-1} + \dots + c_0)C^n$.

In either case, we solve for the unknown constants by substituting the preceding solution-form into the original recurrence.

3. Form the general solution, which is the sum of the homogeneous solution and the particular solution (i.e., $T(n) = T_h(n) + T_p(n)$).
4. Substitute the boundary conditions into the general solution and solve the resulting equations.

Example 4.8 Find the general solution to the recurrence $T(n) = 3T(n-1) + 2n$. What is the solution with $T(1)=1$?

We need to find the homogeneous solution T_h and the particular solution T_p . For the homogeneous solution, the associated characteristic equation is $x-3=0$. Thus, $T_h(n)=c_13^n$. Next, we find the particular solution. Here, the nonhomogeneous part $F(n)=2n(1)^n$, where 1 is not a root of the characteristic equation. Thus, the particular solution is given by $T_p(n)=an+b$ for some constants a and b , which we must determine based on the given recurrence. We substitute $T_p(n) = an+b$ in $T_p(n) = 3T_p(n-1)+2n$ to get

$$an+b = 3(a(n-1)+b) + 2n.$$

Rearranging so that the expression appears as a polynomial in terms of n with all terms appearing in one side,

$$(-2-2a)n + (-2b+3a) = 0.$$

From this we conclude that $-2-2a=0$ and $-2b+3a=0$. Thus, $a = -1$ and $b = -3/2$. Consequently, $T_p(n) = -n-3/2$. Finally, we obtain the general solution $T(n) = T_p(n)+T_h(n) = c_13^n -n-3/2$.

To find the specific solution with $T(1)=1$, we use the general solution to get $1=T(1)=c_1 3^{-1}-3/2$. This gives $c_1=7/6$. Consequently, $T(n) = T_h(n)+T_p(n) = (7/6)3^n -n -3/2$.

Example 4.9 Find the solution to the recurrence $T(n) = 2T(n-1)+ n^2 2^n$ for $n > 1$ with $T(1) = 0$.

For the homogeneous solution, the associated characteristic equation is $x-2 = 0$. Thus, $T_h(n) = c_1 2^n$. For the particular solution, the nonhomogeneous part $F(n)=n^2(2)^n$. Since 2 is a root of the characteristic equation of multiplicity=1, the particular solution is given by $T_p(n)=n(an^2+bn+c)2^n$ for some constants a , b and c , which we must determine based on the given recurrence. We substitute $T_p(n)=n(an^2+bn+c)2^n$ in $T_p(n)= 2T_p(n-1)+ n^2 2^n$ to get the following:

$$n(an^2+bn+c)2^n = 2[(n-1)(a(n-1)^2+b(n-1)+c) 2^{n-1}] + n^2 2^n.$$

We divide both sides by 2^n to get

$$n(an^2+bn+c) = (n-1) (a(n-1)^2+b(n-1)+c) + n^2$$

This can be rewritten as:

$$an^3+bn^2+cn = a(n-1)^3 +b(n-1)^2 +c(n-1)+ n^2 \Rightarrow (-3a+1) n^2 + (3a- 2b) n+(b-a-c) = 0.$$

From this we conclude that $(-3a+1)=0$, $(3a-2b)=0$, and $(b-a-c)=0$. Thus, $a=1/3$, $b=1/2$, and $c=1/6$. Consequently, $T_p(n) = n(an^2+bn+c)2^n = n((1/3)n^2+(1/2)n+(1/6))2^n$. Thus, the general solution is given by $T(n) = T_h(n)+T_p(n) = c_1 2^n+n((1/3)n^2+(1/2)n+(1/6))2^n$. Now we can solve for c_1 using the boundary condition $0=T(1)=c_1 2^1+ (1/3+1/2+1/6)(2)=2c_1+2 \Rightarrow c_1 = -1$. Consequently, the solution to the original recurrence is $T(n) = -2^n+ n((1/3)n^2+(1/2)n+(1/6))2^n$. Let us verify a few terms. By the recurrence, $T(2) = 2T(1)+(2^2)(2^2) = 16$ and $T(3) = 2T(2)+(3^2)(2^3) = 104$. From the claimed solution to the recurrence, $T(3) = -2^3 + 3((1/3)3^2+(1/2)3+(1/6))2^3 = -8 + 3(3+3/2+1/6) (8) = -8 + (72+ 36+4) = 104$.

Divide and Conquer

4.1.4 Recursion Trees and the Master Theorem

The final method we examine, which is especially useful for divide-and-conquer recurrences, makes use of the recursion tree. We will use this method to produce a “master formula” that can be applied to many recurrences of the form given in the associated *Master Theorem*.

Theorem 4.1 (Master Theorem) *The solution to the recurrence $T(n) = aT(n/b) + \Theta(n^k)$; $T(1) = \Theta(1)$, where a , b , and k are all constants, is given by:*

$$\begin{aligned} T(n) &= \Theta(n^k) && \text{if } a < b^k \\ T(n) &= \Theta(n^k \log n) && \text{if } a = b^k \\ T(n) &= \Theta(n^{\log_b a}) && \text{if } a > b^k \end{aligned}$$

Note: (Analogous results hold for the O and Ω notations). The theorem just given is a simplified version. The general version does not restrict the nonrecursive term to be a polynomial in terms of n . In the preceding recurrence, we use n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$; otherwise, the recurrence is not well defined if n/b is not an integer. Replacing each of the terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ does not affect the asymptotic behavior of the recurrence.

The given recurrence represents the running time of an algorithm that divides the problem into a subproblems of size n/b each, solving each subproblem recursively. The term $\Theta(n^k)$ represents the time used by the divide (prerecursion step) and combine (postrecursion step). A *recursion tree* is just a tree that represents this process, where each node represents the divide-and-combine work and then has one child for each recursive call. The leaves of the tree are the base cases of the recursion. Figure 4.1 shows such a tree (Note: Without loss of generality, we can assume that the nonrecursive term corresponding to $\Theta(n^k)$ is given as cn^k for some positive constant c).

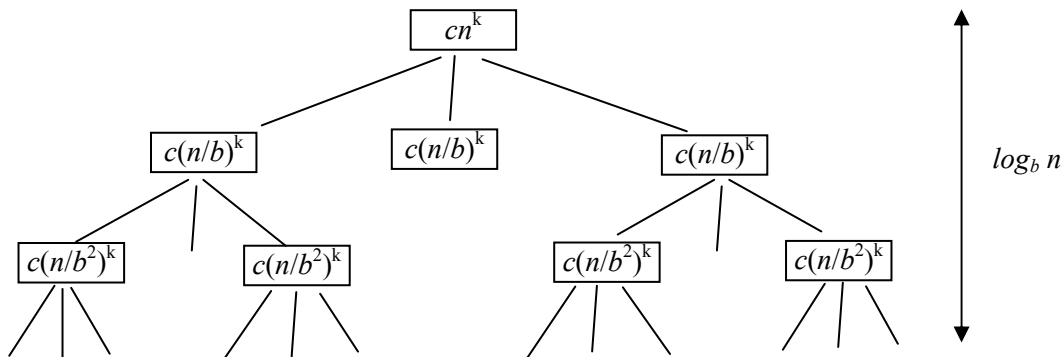


Figure 4.1 The recursion tree corresponding to a divide-and-conquer recurrence.

To compute the result of the recurrence, we simply need to add up all the values in the tree. We can do this by adding them up level-by-level. The top level has value cn^k , the next level sums to $ca(n/b)^k$, the next level sums to $ca^2(n/b^2)^k$, and so on. The depth of the tree (the number of levels not including the root) is $\log_b n$. Therefore, we get $T(n)$ as the summation given by Equation 4.1.

$$T(n) = cn^k [1 + a/b^k + (a/b^k)^2 + (a/b^k)^3 + \dots + (a/b^k)^{\log_b n}] \quad 4.1$$

To ease the manipulation of this, let us define $r = a/b^k$. Notice that r is a *constant* because a , b , and k are all constants. Using our definition of r , the summation simplifies to the following:

$$T(n) = cn^k [1 + r + r^2 + r^3 + \dots + r^{\log_b n}] \quad 4.2$$

Based on the value of r in Equation 4.2, we can consider the following three cases.

Case 1: $r < 1$. In this case, the sum is a convergent series. Even if we imagine the series going to infinity, we still get that the sum $1+r+r^2+\dots = 1/(1-r)$. So, we can upper-bound formula in Equation 4.2 by $cn^k/(1-r)$, and lower bound it by just the first term cn^k . Since r and c are constants, this solves to $\Theta(n^k)$.

$$T(n) = \Theta(n^k) \quad 4.3$$

Case 2: $r = 1$. In this case, all terms in the summation of Equation 4.2 are equal to 1, so the result is given by Equation 4.4.

$$T(n) = cn^k(\log_b n + 1) = \Theta(n^k \log n) \quad 4.4$$

Case 3: $r > 1$. In this case, the last term of the summation dominates. Thus, we get,

$$T(n) = cn^k r^{\log_b n} [(1/r)^{\log_b n} + \dots + 1/r + 1] \quad 4.5$$

Since $1/r < 1$, we can now use the same reasoning as in Case 1. The summation is at most $1/(1-1/r)$, which is a constant. Therefore, we get,

$$T(n) = \Theta(n^k (a/b^k)^{\log_b n}) \quad 4.6$$

We simplify this formula by noticing that $b^{k \log_b n} = n^k$, so we get

$$T(n) = \Theta(a^{\log_b n}) \quad 4.7$$

Finally, Equation 4.8 can be gotten from Equation 4.7, if we swap a with n — To see this, take \log_b of both equations.

$$T(n) = \Theta(n^{\log_b a}) \quad 4.8$$

The preceding three cases for the recursion tree can be likened to stacks of bricks as shown in Figure 4.2. We can view each node in the recursion tree as a brick of height 1 and width equal to its associated value. The value of the recurrence is the area of the stack. In the first case, the area is dominated by the top brick; in the second case, all levels provide an equal contribution, and in the last case, the area is dominated by the bottom level.

Divide and Conquer

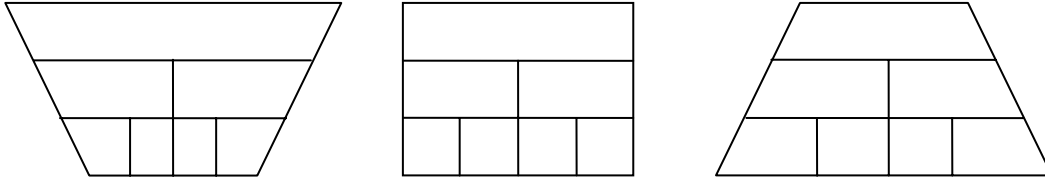


Figure 4.2 The three cases for the recursion tree for the recurrence given in the Master Theorem.

4.1.5 Average-Case Analysis of Quicksort

For the analysis of the average behavior of an algorithm, it is appropriate to assume some probability distribution on the input. In the case of Quicksort, we have seen that if the input is already sorted, and we always chose the leftmost element as the pivot then Quicksort will run in $O(n^2)$. In practice, such case rarely happens. On the other hand, if we assume that each of the $n!$ permutations of the n elements (assuming the elements are all distinct) then this ensure that each element in the input is equally likely to be the leftmost element and thus chosen as the pivot (i.e., for an input $A[1..n]$, $Probability(pivot=A[i]) = 1/n$). Then in such a case, formal analysis shows that Quicksort will run in $O(n \log n)$ on average. (Note: One way to ensure $Probability(pivot=A[i]) = 1/n$ is to choose the pivot at random from the n elements; this is known as *randomized Quicksort*).

Let $C(n)$ denote the number of comparisons performed by the algorithm on the average on an input $A[1..n]$. A Quicksort-call on n elements will call Partition, which does $n-1$ comparisons and then, assuming that Partition() returns a pivot location p ($1 \leq p \leq n$), we execute two Quicksort calls, on $p-1$ and $n-p$ elements. Thus, the number of comparisons performed by Quicksort is given as,

$$C(n) = n-1 + C(p-1) + C(n-p).$$

From the assumption that $Probability(pivot=A[i]) = 1/n$, then it is equally likely that p can be any of the values $1, 2, \dots, n$; thus, the *expected* number of comparisons is given as,

$$C(n) = (n-1) + \frac{1}{n} \sum_{p=1}^n (C(p-1) + C(n-p)) \quad 4.9$$

We note that $\sum_{p=1}^n C(n-p) = C(n-1) + C(n-2) + \dots + C(0) = \sum_{p=1}^n C(p-1)$. Thus, the preceding equation can be rewritten as,

$$C(n) = (n-1) + \frac{2}{n} \sum_{p=1}^n C(p-1) \quad 4.10$$

This type of recurrence is known as a *full-history recurrence*. It seems to be difficult to solve but we can utilize a trick that relates $C(n)$ and $C(n-1)$. First, multiply both sides of Equation 4.10 by n to get,

$$nC(n) = n(n-1) + 2 \sum_{p=1}^n C(p-1). \quad 4.11$$

We can replace n by $(n-1)$ throughout to get,

$$(n-1)C(n-1) = (n-1)(n-2) + 2 \sum_{p=1}^{n-1} C(p-1). \quad 4.12$$

Subtracting Equation 4.12 from Equation 4.11, and rearranging terms yields

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2(n-1)}{n(n+1)}. \quad 4.13$$

Using a new variable $D(n) = \frac{C(n)}{n+1}$, we can rewrite the last recurrence as:

$$D(n) = D(n-1) + \frac{2(n-1)}{n(n+1)}, \quad D(1) = 0. \quad 4.14$$

Clearly, the solution of the preceding equation is

$$D(n) = 2 \sum_{i=1}^n \frac{i-1}{i(i+1)}.$$

We simplify the preceding expression as follows.

$$\begin{aligned} 2 \sum_{i=1}^n \frac{i-1}{i(i+1)} &= 2 \sum_{i=1}^n \frac{2}{(i+1)} - 2 \sum_{i=1}^n \frac{1}{i} \\ &= 4 \sum_{i=2}^{n+1} \frac{1}{i} - 2 \sum_{i=1}^n \frac{1}{i} \\ &= 2 \sum_{i=1}^n \frac{1}{i} - \frac{4n}{n+1}. \end{aligned}$$

Since $H_n = \sum_{i=1}^n \frac{1}{i} = \Theta(n \log n)$ (See Example 1.15), it follows that the preceding expression simplifies to: $\Theta(n \log n) - \Theta(n) = \Theta(n \log n)$.

Divide and Conquer

4.2 Constructing a Tournament Schedule - revisited

Let us consider once more the construction of a tournament schedule for n players. The problem is described in Section 3.4. This time, we use divide-and-conquer. The following solution is taken from [Aho83]. We consider the design of a round robin tournament schedule for $n = 2^k$ players for an integer $k > 1$. The divide-and-conquer approach constructs a schedule for one-half of the players. This schedule is designed by a recursive application of the algorithm by finding a schedule for one half of these players and so on. When we get down to two players, we have the base case and we simply pair them up.

Suppose there are eight players. The schedule for players 1 through 4 fills the upper left corner (4 rows by 3 columns) of the schedule being constructed. The lower left corner (4 rows by 3 columns) of the schedule must match the high numbered players (5 through 8) against one another. This sub-schedule is obtained by adding 4 to each entry in the upper left.

Now we have a partial solution to the problem. All that remains is to have lower-numbered players play high-numbered players; or equivalently, fill the top-right and bottom-right sections of the schedule. For the top-right section, this is easily accomplished by having players 1 through 4 play 5 through 8, respectively, on day 4 and cyclically permuting 5 through 8 on subsequent days. Similarly, for the bottom-right section, we have players 5 through 8 play 1 through 4, respectively, on day 4 and cyclically permuting 1 through 4 on subsequent days. The process is illustrated in Figure 4.3. This process can be generalized to construct a schedule for 2^k players for any k .

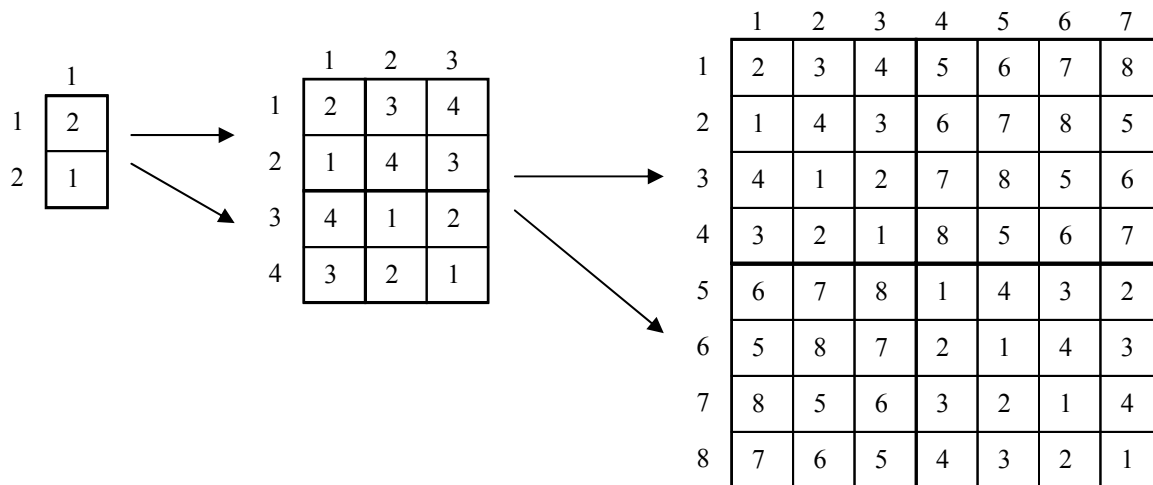


Figure 4.3 Using divide-and-conquer to construct a tournament schedule for 8 players.

Exercise 4.1 Write, with proper explanation, recurrence equations for the running time $T(n)$ of the preceding algorithm.

4.3 The MinMax Problem

We consider a simple problem that, nonetheless, illustrates the utility of divide-and-conquer in algorithm design.

The MinMax Problem. Given a sequence of integers, find the minimum and the maximum values.

The straightforward solution to this problem is to scan the elements searching for the minimum, which would require $n-1$ comparisons, then scan the elements one more time to search for the maximum. Thus in total, there will be $2n-2$ comparisons. Can we do better? *Yes*, divide-and-conquer does it using $(3n/2)-2$ comparisons (i.e., a savings by $n/2$ comparisons). *How?* Answer: read on.

```
integer-pair MinMax(int [] A, int lo, int hi)
{  if (lo==hi) return (A[lo],A[lo])  // one-element case
  else if (lo = hi-1)  // two-element case
  {  if A[lo] <= A[hi] return (A[lo],A[hi]);
    else return (A[hi],A[lo]);
  }
  else // general case
  {  mid = (lo+hi)/2;
    (x1,y1) = MinMax(A, lo, mid);
    (x2,y2) = MinMax(A, mid+1, hi);
    // set x1 as the smaller of x1 and x2
    if (x2 < x1) x1 = x2;
    // set y1 as the larger of y1 and y2
    if (y2 > y1) y1 = y2;
    return (x1,y1);
  }
}
```

Listing 4.2 A divide-and-conquer algorithm for the MinMax problem.

Listing 4.2 gives a divide-and-conquer algorithm for this problem. For a sequence of one element (i.e., $lo=hi$), the element itself is both the minimum and the maximum. For a sequence of two elements (i.e., $lo=hi-1$), one comparison suffices to know the minimum and the maximum. For a sequence having more than two elements we divide the sequence into two equal (or nearly equal) parts, find the minimum and the maximum for each part and use these to compute the overall minimum and maximum.

Let $C(n)$ denote the count of comparisons performed by the algorithm on n elements. Based on the algorithm, we can write the following recurrence equations for $C(n)$.

Base cases: $C(1) = 0$; $C(2) = 1$.

General case: $C(n) = C(n/2) + C(n- n/2) + 2$.

Let us solve this recurrence assuming $n=2^k$ for some nonnegative integer k . First, note that the general-case recurrence can be rewritten as $C(n)=2C(n/2)+2$. Thus, using backward substitution, we get the following:

$$\begin{aligned} C(n) &= 2C(n/2) + 2 \\ &= 2 [2C(n/4) + 2] + 2 = 4 C(n/4) + 4 + 2 \\ &= 4 [2C(n/8) + 2] + 4 + 2 = 8 C(n/8) + 8 + 4 + 2, \end{aligned}$$

and in general,

Divide and Conquer

$$C(n) = 2^i C(n/2^i) + 2^i + \dots + 4 + 2 \quad 4.15$$

To reach $C(2)$, we let $i = k-1$. Thus,

$$C(n) = 2^{k-1} C(2) + 2^{k-1} + \dots + 4 + 2 = 2^{k-1} C(2) + [2^k - 2] = 2^{k-1} + 2^k - 2 = 3n/2 - 2 \quad 4.16$$

Thus, we conclude that the divide-and-conquer algorithm does $3n/2 - 2$ comparisons. This is a *significant* reduction by $n/2$ comparisons over the straightforward algorithm. Let us investigate the reason behind this reduction. It turns out that the reduction in the number of comparisons is due to two facts (note that (a) and (b) \Rightarrow a reduction of $n/2$ comparisons):

- (a) *Explicit handling of a two-element case using one comparison.* Observe that two comparisons are used if a two-element case is handled as a general case.

AND

- (b) *The two-element case occurs $n/2$ times.*

In other words, there will not be any reduction in the number of comparisons if (a) is not satisfied. To see this, note that if the algorithm does not handle a two-element case explicitly as a base case, then Equation 4.16 becomes,

$$C(n) = 2^k C(1) + 2^k + \dots + 4 + 2 = 2^k C(1) + [2^{k+1} - 2] = 2^{k+1} - 2 = 2n - 2 \quad 4.17$$

The justification for (b) becomes apparent when we consider the tree of recursive calls — See Figure 4.4 for $n=16$. In general (even if n is not a power of 2), we can see that the leaves of the recursive tree correspond to the partitioning of the n elements into 2-element (disjoint) sets. Thus, there will be $n/2$ such cases.

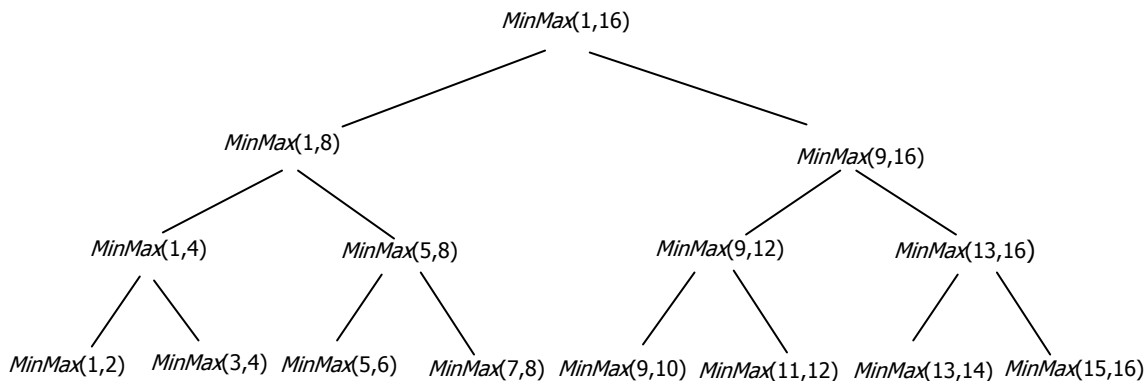


Figure 4.4 The tree of recursive calls for computing $MinMax(A,1,16)$.

Exercise 4.2 Another algorithm that satisfies the conditions (a) and (b) stated previously is obtained using induction by handling the n elements as a two-element case (using three comparisons) and $n-2$ elements. Express this algorithm in pseudocode.

4.4 Finding the Majority Element

In an election involving n voters and k candidates, each voter casts his vote to one of the k candidates. Thus, the outcome from voting can be represented as an n -element sequence where an element is an integer in $[1, k]$. There are then various criteria to determine the winner. One criterion could be to declare the candidate who scores *more than 50%* of the votes as the winner.

Such element is known as the *majority* element. Since the majority element is not always assured, an alternative criterion is that the winner be the candidate that scores most votes. Such element is known as the *mode* element. Still another possibility is to have a rerun election limited to the candidates who score above certain threshold.

Next, we discuss algorithms for finding the majority element. Note that the majority, if it exists, is unique. This is because we cannot have two distinct elements each of which appears more than 50%.

The Majority-Element Problem. Given a sequence of n elements where each element is an integer in $[1, k]$, return the majority element (an element that appears more than $n/2$ times) or zero if no majority element is found.

A sequence of size 2 has a majority only if both elements are equal. For an input of size $n=8$ (and $n=9$), the majority element must appear at least 5 times. For example, for the sequence 1,2,1,3,1,1,4,1 the majority element is 1.

Next, we consider several algorithms for finding the majority element.

A Distribution-Based Algorithm

There is a simple and fast algorithm for determining the majority if k is small. Simply, use an array $Count[1..k]$ where $Count[i]$ is the number of occurrences of element i . This array can be computed by scanning the input sequence once. Then the $Count$ array is scanned to determine whether there is any entry having a value $> n/2$. Such algorithm has $O(n+k)$ (i.e., $O(n)$ since k is much smaller than n) running time and $O(k)$ space.

Exercise 4.3 Give program code for the preceding algorithm.

Exercise 4.4 The preceding algorithm is very inefficient if k is very large in comparison with n . Explain how hashing might be useful to efficiently implement the algorithm in this case. Give program code for the modified algorithm and state its running time and space complexity. Note: If k is much larger than n , many of the values in $[1, k]$ do not appear as elements.

A Comparison-Based Algorithm

A simple comparison-based algorithm for finding the majority is as follows: Count the occurrences of the elements, one at a time, return when finding an element whose count is more than $n/2$. The algorithm is shown in Listing 4.3.

Note that, as an optimization measure, there is no point of searching for the i -th element among the elements that appear in positions $< i$. *Can you see why?* Another optimization measure that can be incorporated is to end the current iteration of the outer loop if $(count + \text{count of remaining elements to be checked}) \leq n/2$.

The dominant operation in this algorithm is the comparison “ $(A[j] == item)$ ”. In the worst case (i.e., when there is no majority), it is executed $(n-1)+(n-2)+\dots+1 = n(n-1)/2 = O(n^2)$. Thus, this algorithm has $O(n^2)$ running time in the worst case. The algorithm uses $O(1)$ space.

Divide and Conquer

```
Input: A positive integer array A[1..n]
Output: Return the majority element or zero if no majority is found

int Majority(int[] A, int n)
{ for(int i=1; i<= n; i++)
  { int count = 1;
    int item = A[i];
    for(int j=i+1; j <= n; j++)
      { if (A[j] == item) count++; }

    if (count > n/2) return item;
  }
  return 0;
}
```

Listing 4.3 An $O(n^2)$ -algorithm for finding the majority element.

A Divide-and-Conquer Algorithm

Let us use divide-and-conquer approach to develop an algorithm for the majority problem. First, we note the following fact. If the n elements are divided into equal (or nearly equal) halves then the majority among the n elements must be a majority in the first half or a majority in the second half. This claim can be proven using a proof by contradiction, as follows.

If n elements are divided into two halves, then these will be of sizes $n/2$ and $n-n/2$. Suppose the majority element x (which, by definition, appears more than $n/2$ times in the original sequence) is neither a majority in the first half, nor a majority in the second half. This implies that x appears $\leq (n/2)/2$ in the first half and $\leq (n-n/2)/2$ in the second half. This implies that the total count of occurrences of x in the original sequence is $\leq (n/2)/2 + (n-n/2)/2 \leq n/2$, but then x cannot be a majority (i.e., we have reached a contradiction).

Note that the preceding claim is a *one-way* implication. In other words, it is possible to have a majority in one of the halves yet that element is not the overall majority. For example, given the two halves: 1,1,1,2 and 3,4, 5,6; we see that, while 1 is a majority in the first half, the two halves put together have no majority. This means that the majority element in one of the halves is a *candidate* majority for the whole sequence (Note: It is possible to have *two* candidates). Thus, it suffices to check the count of occurrences in the whole sequence for at most two candidates. The algorithm is given in Listing 4.4.

```

Input: A positive integer array A[lo..hi]
Output: Return the majority element or zero if no majority is found

int Majority(int[] A, int lo, int hi)
{ if (lo > hi) return 0; // empty sequence case
  else if (lo == hi) return A[lo]; // one-element sequence case
  else // general case
    { int mid = (lo+hi)/2; // integer division
      int x = Majority(A,lo,mid);
      int y = Majority(A,mid+1,hi);
      if (x==y) return x; // x and y are both zero or both majority
      if (x > 0) // x is a majority in 1st half
        if ( Count(A,lo, hi,x) > (hi-lo+1)/2 ) return x;
      if (y > 0) // y is a majority in 2nd half
        if ( Count(A,lo, hi,y) > (hi-lo+1)/2 ) return y;
      return 0;
    }
}

```

Listing 4.4 A divide-and-conquer algorithm for finding the majority element.

To analyze the preceding algorithm, let us write recurrence equations for $C(n)$, the (worst case) number of element comparisons executed by the algorithm. To simplify things, we will only count comparisons executed by the *Count()* method. Through inspection, we deduce the following equations:

$$C(n) = C(n/2) + C(n-n/2) + 2n$$

$$C(0) = C(1) = 0$$

If we are merely interested in determining the order of running time (instead of the exact number of element comparisons), we can approximate the general recurrence as $C(n)=2C(n/2)+2n$. This can be solved using backward substitution as follows:

$$\begin{aligned}
 C(n) &= 2C(n/2) + 2n \\
 &= 2 [2C(n/4) + 2(n/2)] + 2n = 4C(n/4) + 2n + 2n \\
 &= 4 [2C(n/8) + 2(n/4)] + 2n + 2n = 8C(n/8) + 2n + 2n + 2n,
 \end{aligned}$$

and, in general, $C(n) = 2^i C(n/2^i) + i(2n)$. Assuming $n=2^k$, we get $C(n) = 2^i C(n/2^i) + i(2n)$. To reach $C(1)$, we let $i=k$. Thus, $C(n) = 2^k C(n/2^k) + k(2n) = k(2n) = 2n \log n = \Theta(n \log n)$.

Note: The recurrence $C(n)=2C(n/2)+O(n)$ is a familiar recurrence for many divide-and-conquer algorithms where the time for divide-and-combine is linear in the input size (Mergesort is one such example). The solution to such recurrence is $C(n) = O(n \log n)$.

There is a rather interesting and very efficient (i.e., $O(n)$) algorithm for finding the majority element. It is based on induction (i.e., problem size reduction) by *elimination of noncandidate*. The algorithm is discussed in the solution for Problem 5 (See end-of-chapter solved exercises).

Exercise 4.5 The preceding divide-and-conquer majority algorithm can be made more efficient by having it memorize and return, in addition to the majority element, the count of occurrences. Rewrite the algorithm to take this into account. Next, write the recurrence equation for $C(n)$ (as defined above) for the modified algorithm and indicate the worst-case order of running time in this case.

Divide and Conquer

Exercise 4.6 Convert the preceding divide-and-conquer majority algorithm into an iterative algorithm. Hint: Use recursion unfolding but do it level-by-level based on the tree of recursive calls.

Exercise 4.7 Assuming that the input sequence contains a majority element, carry out best-case analysis on element comparisons for the divide-and-conquer majority algorithm given previously. How many times does the *Count()* method get executed in the best case?

4.5 The Skyline Problem

We consider a problem related to the drawing of geometric figures. The problem is concerned with the removal of hidden lines — lines obscured by other parts of a drawing.

The Skyline Problem. Given the exact locations and shapes of n rectangular buildings in a 2-dimensional city, give an algorithm that computes the skyline (in 2 dimensions) of these buildings, eliminating hidden lines.

As an example of an input is given in Figure 4.5(a); the corresponding output is given in Figure 4.5(b).

We assume that the bottom of all buildings lie on a fixed horizontal line. A building B_i is represented by the triple (L_i, H_i, R_i) where L_i and R_i denote the left and right x -coordinates of the building respectively, and H_i denotes the building's height. The input is a list of triples; one per building. The output is the skyline specified as a list of x -coordinates and heights connecting them arranged in order by x -coordinates. For the example shown in Figure 4.5, the input and output are:

Input: $(1, \underline{11}, 5), (2, \underline{6}, 7), (3, \underline{13}, 9), (12, \underline{7}, 16), (14, \underline{3}, 25), (19, \underline{18}, 22)$

Output: $(1, \underline{11}, 3, \underline{13}, 9, 12, \underline{7}, 16, \underline{3}, 19, \underline{18}, 22, \underline{3}, 25, 0)$

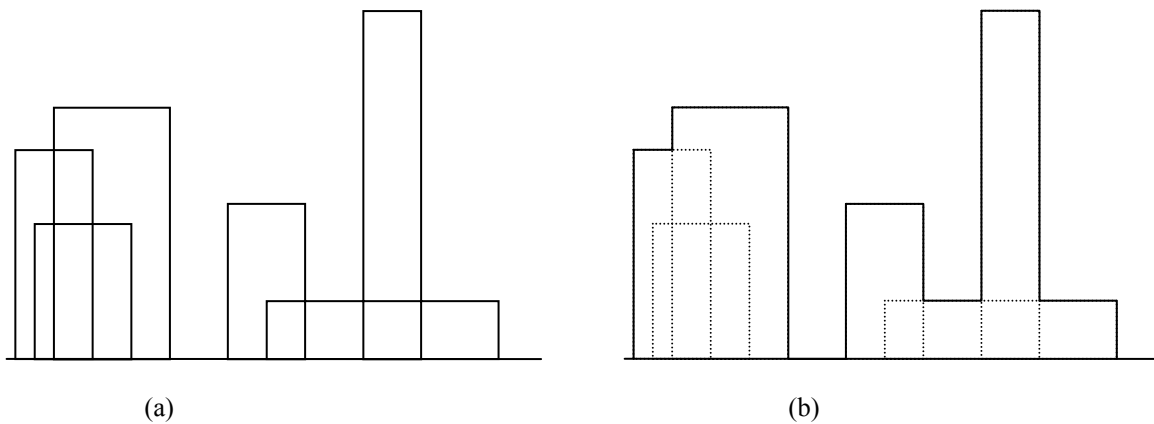


Figure 4.5 The skyline problem (a) input (b) output (the skyline).

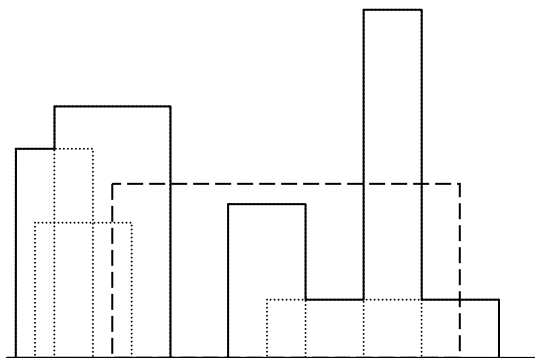


Figure 4.6 Addition of a building B_n (dashed line) to the skyline of Figure 4.5(b).

Divide and Conquer

The straightforward algorithm for this problem uses induction on n (number of buildings). The base step is for $n=1$ and the skyline can be obtained directly from B_1 . As an induction step, we assume that we know S_{n-1} (the skyline for $n-1$ buildings), and then must show how to add the n -th building B_n to the skyline. The process of adding a building to the skyline can be examined by looking at Figure 4.6, where we add the building $B_n=(L_n, H_n, R_n)=(6, 7, 24)$ to the skyline $(1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 25, 0)$.

We scan the skyline from left to right stopping at the first x -coordinate x_1 that immediately precedes L_n (in this case $x_1=3$) and then we extract the part of the skyline overlapping with B_n as a set of strips $(x_1, h_1), (x_2, h_2), \dots, (x_m, h_m)$ such that $x_m < R_n$ and $x_{m+1} \geq R_n$ (or x_m is last). In this set of strips, a strip will have its h_i replaced by H_n if $H_n > h_i$ (because the strip is now covered by B_n). If there is no x_{m+1} then we add an extra strip (replacing the last 0 in the old skyline) $(x_m, H_n, R_n, 0)$. Also, we check whether two adjacent strips have the same height; if so, they are merged together into one strip. This process can be viewed as a *merging* of B_n and S_{n-1} .

In the worst case, this algorithm would need to examine all the $n-1$ triples when adding B_n (this is certainly the case if B_n is so wide that it encompasses all other triples). Likewise, adding B_{n-1} would need to examine $n-2$ triples, and so on. This implies that the algorithm is $O(n)+O(n-1)+\dots+O(1)=O(n^2)$.

A Divide-and-Conquer Algorithm

Is merging two skylines substantially different from merging a building with a skyline? The answer is, of course, No. This suggests that we use divide-and-conquer. Divide the input of n buildings into two equal (or nearly equal) sets. Compute (recursively) the skyline for each set then merge the two skylines.

The algorithm corresponds to the *FindSkyLine()* method given in Listing 4.5. This has a structure similar to Mergesort. Unlike Mergesort, the input and output for *FindSkyLine* are of different data-types. In Mergesort, the input and output are arrays of some base type. However, that does not matter much. We simply need to merge two skylines (and not two sets of buildings). For instance, given two skylines $A=(a_1, ha_1, a_2, ha_2, \dots, a_n, 0)$ and $B=(b_1, hb_1, b_2, hb_2, \dots, b_m, 0)$, we merge these lists as the new list: $(c_1, hc_1, c_2, hc_2, \dots, c_{n+m}, 0)$. Clearly, we merge the list of a s and b s just like in the standard Merge algorithm. But, in addition to that, we have to decide on the correct height in between these boundary values. We use two variables *CurH1* and *CurH2* (note that these are the heights prior to encountering the heads of the lists) to store the current height of the first and the second skyline, respectively. When comparing the head entries (*CurH1*, *CurH2*) of the two skylines, we introduce a new *strip* (and append to the output skyline) whose x -coordinate is the minimum of the entries' x -coordinates and whose height is the maximum of *CurH1* and *CurH2*.

For our purpose — see Listing 4.6 — a skyline is a list of integer pairs. For legibility, we define a *strip* structure to represent a pair (an x -coordinate component lx and a height component h). We define a *Skyline* class that maintains a list of *strips*. For simplicity, the list is built using a *statically allocated* array. The following is a typical code to prepare the proper input and then invoke *FindSkyline()*.

```
int n = 6;
Bldg[] B = new Bldg[n];
B[0] = new Bldg(1,11,5); B[1] = new Bldg(2,6,7); B[2] = new Bldg(3,13,9);
B[3] = new Bldg(12,7,16); B[4] = new Bldg(14,3,25); B[5] = new Bldg(19,18,22);

Skyline sk = Skyline.FindSkyline(B,0,n-1);
Console.WriteLine("The skyline: " + sk.ToString());
```

The algorithm given in Listing 4.5 produces “noncompact” output. For example, for the preceding input, we get the following output: $(1, 11, 2, 11, 3, 13, 5, 13, 7, 13, 9, 0, 12, 7, 14, 7, 16, 3, 19, 18, 22, 3, 25, 0)$. While merging two skylines or after we are done (say, within *ToString()* method of the *Skyline* class), we can massage the skyline to

eliminate redundant strips, such as 1, 11, 2, 11, whenever we see two adjacent strips having the same height. Similarly, we eliminate strips that happen to have the same x -coordinate.

Let $T(n)$ denote the running time of this algorithm for n buildings. Since merging two skylines of size $n/2$ takes $O(n)$, we find that $T(n)$ satisfies the recurrence $T(n)=2T(n/2)+O(n)$. This is just like Mergesort. Thus, we conclude that the divide-and-conquer algorithm for the skyline problem is $O(n \log n)$.

```
static Skyline FindSkyline(Bldg[] B, int lo, int hi)
{ if (lo == hi)
  { Skyline sk = new Skyline(2);
    sk.Append(new strip(B[lo].lx, B[lo].h) );
    sk.Append(new strip(B[lo].rx, 0));
    return sk;
  }
  int mid = (lo+hi)/2;
  Skyline sk1 = FindSkyline(B, lo, mid);
  Skyline sk2 = FindSkyline(B, mid+1, hi);
  return MergeSkyline(sk1, sk2);
}

static Skyline MergeSkyline(Skyline SK1, Skyline SK2)
{ Skyline SK = new Skyline(SK1.Count + SK2.Count); // Allocate array space
  int CurH1 = 0; int CurH2 = 0;
  while ((SK1.Count > 0) && (SK2.Count > 0))
  if (SK1.Head().lx < SK2.Head().lx)
  { int CurX = SK1.Head().lx;
    CurH1 = SK1.Head().h;
    int MaxH = CurH1;
    if (CurH2 > MaxH) MaxH = CurH2;
    SK.Append(new strip(CurX, MaxH));
    SK1.RemoveHead();
  }
  else
  { int CurX = SK2.Head().lx;
    CurH2 = SK2.Head().h;
    int MaxH = CurH1;
    if (CurH2 > MaxH) MaxH = CurH2;
    SK.Append(new strip(CurX, MaxH));
    SK2.RemoveHead();
  }

  while (SK1.Count > 0) // Append SK1 to Skyline
  { strip str = SK1.RemoveHead(); SK.Append(str); }
  while (SK2.Count > 0) // Append SK2 to Skyline
  { strip str = SK2.RemoveHead(); SK.Append(str); }

  return SK;
}
```

Listing 4.5 A divide-and-conquer algorithm for the skyline problem.

Divide and Conquer

```
struct Bldg
{ internal int lx, rx, h;
  public Bldg(int x1, int h1, int x2) { lx = x1; h = h1; rx = x2; }
}

class Skyline
{ struct strip
  { internal int lx, h;
    internal strip(int x1, int h1) { lx = x1; h = h1; }
  }

  strip[] strips;
  public int Count;
  int StartLoc;

  public Skyline(int n)
  { Count = 0; StartLoc = 0; strips = new strip[n]; }

  public void Append(strip str)
  { strips[StartLoc+Count] = str; Count++; }

  public strip Head() { return strips[StartLoc]; }

  public strip RemoveHead()
  { strip str = strips[StartLoc];
    Count--; StartLoc++;
    return str;
  }

  public override string ToString()
  { string str = "";
    for(int i = StartLoc; i < StartLoc+Count; i++)
    { if (i > StartLoc) str = str + ",";
      str = str + strips[i].lx + "," + strips[i].h;
    }
    return "(" + str + ")";
  }
}
```

Listing 4.6 A *Skyline* class used by the skyline algorithm of Listing 4.5 .

4.6 Polynomial Multiplication

We consider the problem of multiplying polynomials.

The Polynomial-Multiplication Problem. Given two polynomials of degree n , $A(x)=a_0+a_1x+\dots+a_nx^n$ and $B(x)=b_0+b_1x+\dots+b_nx^n$; compute the product $A(x)B(x)$.

Assume that the coefficients a_i s and b_i s are stored in arrays $A[0..n]$ and $B[0..n]$. The cost of a matrix-multiplication algorithm is the number of scalar multiplications and additions performed.

Convolutions

Let $A(x) = \sum_{i=0}^n a_i x^i$ and $B(x) = \sum_{i=0}^m b_i x^i$.

Then $A(x) \times B(x) = C(x) = \sum_{k=0}^{n+m} c_k x^k$ where $c_k = \sum_{i=0}^k a_i b_{k-i}$ for $0 \leq k \leq n+m$.

The vector $(c_0, c_1, \dots, c_{n+m})$ is known as the *convolution* of the vectors (a_0, a_1, \dots, a_n) and (b_0, b_1, \dots, b_m) . Calculating convolutions (and, thus, polynomial multiplication) is a major problem in digital signal processing. Convolutions appear in some unexpected places. For example, every row in *Pascal's triangle* (in this triangle, the n -th row consists of the binomial coefficients $\binom{n}{i}$ for $i=0$ to n) can be obtained from the previous row by convolution with the vector $[1, 1]$; equivalently, if the polynomial $p(x)$ represents a row, the next row is given by $(1+x)^* p(x)$.

Example 4.10 Given, $A(x) = 1 + 2x + 3x^2$ and $B(x) = 4 + 5x + 6x^2$, then

$$A(x)B(x) = (1 \times 4) + (1 \times 5 + 2 \times 4)x + (1 \times 6 + 2 \times 5 + 3 \times 4)x^2 + (2 \times 6 + 3 \times 5)x^3 + (3 \times 6)x^4.$$

For the polynomial-multiplication problem, it is generally assumed that the two input polynomials are of the same degree n . If the input polynomials are of different degrees, then we simply view the smaller-degree polynomial as having zero coefficients for its high-order terms.

A Direct (Brute-Force) Approach

Let $A(x) = \sum_{i=0}^n a_i x^i$ and $B(x) = \sum_{i=0}^n b_i x^i$.

Then $A(x) \times B(x) = C(x) = \sum_{k=0}^{2n} c_k x^k$ where $c_k = \sum_{i=0}^k a_i b_{k-i}$ for $0 \leq k \leq 2n$.

The direct approach is to compute all c_k using the preceding formula. The total number of scalar multiplications and additions needed are $\Theta(n^2)$ and $\Theta(n^2)$, respectively. Hence, the complexity is $\Theta(n^2)$.

Can we do better? Let us try a divide-and-conquer approach.

A Divide-and-Conquer Approach

Let $m = \lfloor n/2 \rfloor$ and define $A_0(x)$ and $A_1(x)$ as follows:

$$\begin{aligned} A_0(x) &= a_0 + a_1x + \dots + a_{m-1}x^{m-1} \\ A_1(x) &= a_m + a_{m+1}x + \dots + a_nx^{n-m} \end{aligned}$$

Divide and Conquer

Clearly, $A(x) = A_0(x) + x^m A_1(x)$. Similarly, we define $B_0(x)$ and $B_1(x)$ such that $B(x) = B_0(x) + x^m B_1(x)$.

Now, $A(x)B(x) = A_0(x)B_0(x) + x^m [A_0(x)B_1(x) + A_1(x)B_0(x)] + x^{2m} A_1(x)B_1(x)$. This latter expression requires four polynomial-multiplication operations where the operands involved are polynomials of degree $n/2$. In other words, the original problem of size n is now divided into 4 subproblems of size $n/2$.

Example 4.11 Given $A(x) = 2 + 5x + 3x^2 + x^3 - x^4$ and $B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$, we get the following:

$$\begin{aligned} A_0(x) &= 2 + 5x; & A_1(x) &= 3 + x - x^2 \\ B_0(x) &= 1 + 2x; & B_1(x) &= 2 + 3x + 6x^2 \end{aligned}$$

$$\begin{aligned} A_0(x)B_0(x) &= 2 + 9x + 10x^2 \\ A_0(x)B_1(x) &= 4 + 16x + 27x^2 + 30x^3 \\ A_1(x)B_0(x) &= 3 + 7x + x^2 - 2x^3 \\ A_1(x)B_1(x) &= 6 + 11x + 19x^2 + 3x^3 - 6x^4 \end{aligned}$$

$$\begin{aligned} \text{Thus, } A(x)B(x) &= (2+9x+10x^2) + x^2 [(4+16x+27x^2 + 30x^3) + (3+7x+x^2 - 2x^3)] + x^4 (6+11x+19x^2+3x^3 - 6x^4) \\ &= 2 + 9x + 17x^2 + 23x^3 + 34x^4 + 39x^5 + 19x^6 + 3x^7 - 6x^8. \end{aligned}$$

The conquer step solves four subproblems of size $n/2$ each. The combine step adds four polynomials of degree $n/2$ each. This is $\Theta(n)$. Thus, $T(n) = 4T(n/2) + \Theta(n)$. The solution for this recurrence (i.e., using the master theorem) is $T(n) = \Theta(n^2)$. This is no better than the direct approach.

Question: Given four numbers A_0, A_1, B_0, B_1 , how many multiplications are needed to compute the three values $A_0B_0, A_0B_1 + A_1B_0$, and A_1B_1 ?

Obviously, this can be done using four multiplications, but there is a way of doing it using only three multiplications. Define Y, U and Z as follows:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0 B_0 \\ Z &= A_1 B_1 \end{aligned}$$

U and Z are what we originally wanted and $A_0 B_1 + A_1 B_0 = Y - U - Z$.

Improving the Divide-and-Conquer Algorithm

Define $Y(x), U(x)$ and $Z(x)$ such that:

$$\begin{aligned} Y(x) &= (A_0(x) + A_1(x)) \times (B_0(x) + B_1(x)) \\ U(x) &= A_0(x)B_0(x) \\ Z(x) &= A_1(x)B_1(x) \end{aligned}$$

Then, $Y(x) - U(x) - Z(x)$ gives $A_0(x)B_1(x) + A_1(x)B_0(x)$.

Hence, $A(x)B(x)$ is given by $U(x) + x^m [Y(x) - U(x) - Z(x)] + x^{2m}Z(x)$. This way, we need to call the multiply procedure three times: first to compute Y , second to compute U , and a third time to compute Z .

Running-Time Analysis of the Modified Algorithm

The conquer step solves three subproblems of size $n/2$ each. The combine step adds six polynomials of degree $n/2$ each. This is $\Theta(n)$. Thus, $T(n)=3T(n/2)+\Theta(n)$. The solution to this recurrence (i.e., using the Master Theorem) is $T(n)=\Theta(n^{\log_2 3})=\Theta(n^{1.58})$.

The previous discussion shows that a straight-forward divide-and-conquer approach may not give the best solution. Our original divide-and-conquer algorithm was just as bad as brute force. However, through clever rearrangement of terms, we were able to get an efficient algorithm. This same algorithm can be adapted for multiplying two large integers; we simply think of the digits of an integer as the coefficients of a polynomial. For example, the decimal number $456 = 4*10^2 + 5*10 + 6$ can be thought as corresponding to the polynomial $p(x) = 4x^2 + 5x + 6$.

Cooley [Coo65] devised an $O(n \log n)$ -algorithm for multiplying two polynomials of degree n . Cooley's algorithm relies on using the *fast Fourier transform (FFT)*. In this case, a polynomial is represented by its values at specially chosen points, and the polynomial-multiplication problem is reduced into an FFT problem. The FFT algorithm itself is a divide-and-conquer algorithm and is considered one of the most important discoveries in the field of algorithms in recent decades.

Divide and Conquer

4.7 Matrix Multiplication

Matrix multiplication is an important problem in linear algebra; it is used for solving linear systems and matrix inversion. It is also needed for computing transitive closure. Matrix multiplication arises in computer graphics applications, such as coordinate transformations via scaling, rotation, and translation.

The Matrix Multiplication Problem. Given two matrices A of size $m \times n$ and B of size $n \times r$, the product matrix $C = A \times B$ is defined such that $C[i, j] = \sum_{k=1}^n A[i, k] * B[k, j]$ for $1 \leq i \leq m$ and $1 \leq j \leq r$.

This definition leads to the following standard algorithm.

```
// Compute C = AxB, where A is mxn matrix, B is nxr matrix, C is mxr matrix
for i=1 to m
  for j = 1 to r
    { C[i, j] = 0;
      for k = 1 to n
        C[i, j] = C[i, j] + A[i, k]*B[k, j];
    }
```

Complexity of the Standard Algorithm

The standard algorithm computes a total of mr entries for the C matrix where the computation of each entry uses $\Theta(n)$ scalar additions and $\Theta(n)$ scalar multiplications. Thus, the algorithm runs in $\Theta(mnr)$ time. For an input consisting of $n \times n$ square matrices, the algorithm does n^3 multiplications and $n^2(n-1)$ additions. Hence, the complexity of the algorithm is $\Theta(n^3)$.

Strassen's algorithm and Winograd's algorithm are two matrix multiplication algorithms that are asymptotically faster than the standard algorithm. These are based on clever divide-and-conquer recurrences. However, they are difficult to program and require very large matrices to beat the standard algorithm. In particular, some empirical results show that Strassen's algorithm is unlikely to beat the standard algorithm for $n \leq 100$.

4.7.1 Strassen's Matrix Multiplication

Strassen's algorithm is a divide-and-conquer algorithm. For clarity, we will assume that the input matrices are both $n \times n$ and that n is a power of 2. If n is not a power of 2, matrices can be padded with rows and columns of zeros. We decompose each matrix in four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}, C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Strassen's algorithm computes seven new matrices, M_1 through M_7 .

$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$

Then the C matrix is given by:

$$\begin{aligned} C_{00} &= M_1 + M_4 - M_5 + M_7 \\ C_{01} &= M_3 + M_5 \\ C_{10} &= M_2 + M_4 \\ C_{11} &= M_1 + M_3 - M_2 + M_6 \end{aligned}$$

It is not difficult to verify that the submatrices of C are calculated correctly if we use Strassen's formulas. For example, C_{00} is $A_{00} * B_{00} + A_{01} * B_{10}$ is equal to $M_1 + M_4 - M_5 + M_7$. Note that the expressions for the matrices M_1 through M_7 involve matrix multiplication, which is computed (recursively) using Strassen's algorithm. These matrices are computed directly as addition and multiplication of numbers only if they are of size 1×1 .

Complexity Analysis of Strassen's Algorithm

Let $M(n)$ denote the number of multiplications made by Strassen's algorithm for multiplying two $n \times n$ matrices (where n is a power of 2), then $M(n)$ is given by the following recurrence:

$$M(n) = 7M(n/2) \text{ for } n > 1, \quad M(1) = 1.$$

Since the savings in the number of multiplications is achieved at the expense of making extra additions, let us consider the number of additions $A(n)$ made by Strassen's algorithm for multiplying two $n \times n$ matrices, which is given by the recurrence:

$$A(n) = 7A(n/2) + 18(n/2)^2 \text{ for } n > 1, \quad A(1) = 0.$$

For the solution's order, we can use the Master Theorem where we find that both $M(n)$ and $A(n)$ are $O(n^{\log_2 7})$. Thus, the complexity of Strassen's algorithm is $O(n^{\log_2 7}) = O(n^{2.81})$.

Note that for multiplying two $n \times n$ matrices where $n=2$, Strassen's algorithm does 7 multiplications and 18 additions whereas the standard algorithm does 8 multiplications and 4 additions. This means that when using Strassen's algorithm, we have traded 1 multiplication for 14 additions, which does not appear to be any savings. However, in the long run (i.e., for $n > 100$), the saving in multiplications will outnumber the count of extra additions. To avoid the cost of the extra additions that would undo any savings in multiplications, a proper implementation of Strassen's algorithm should call the standard algorithm whenever n falls below a certain threshold (for example, $n < 80$).

4.7.2 Winograd's Matrix Multiplication

It is obvious that an element in the product (output) matrix is the *dot product* of a row and a column from the input matrices. Winograd observed that the dot product can be factored in a way that allows us to preprocess some of the work. For example, consider the dot product of the vectors $V=(v_1, v_2, v_3, v_4)$ and $W=(w_1, w_2, w_3, w_4)$. It is given by the following:

$$V \bullet W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4$$

It is also given by the following:

$$V \bullet W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4.$$

Divide and Conquer

At first look, it appears that the second equation does more work than the first, but what might not be obvious is the fact that the second equation allows us to preprocess some of the work because the last few terms involve either V alone or W alone.

Let A be an $m \times n$ matrix and B be an $n \times r$ matrix. Let $C = A \times B$. Then, assuming n is even, to calculate C , first the rows of A and columns of B are processed as follows:

$$\begin{aligned} Row_i &= a_{i1} * a_{i2} + a_{i3} * a_{i4} + a_{i5} * a_{i6} + \dots + a_{i,n-1} * a_{i,n} \\ Col_i &= b_{1i} * b_{2i} + b_{3i} * b_{4i} + b_{5i} * b_{6i} + \dots + b_{n-1,i} * b_{n,i} \end{aligned}$$

Then, the C matrix is obtained as follows:

$$c_{ij} = \sum_{k=1}^{n/2} [(a_{i,2k-1} + b_{2k,j}) * (a_{i,2k} + b_{2k-1,j})] - Row_i - Col_j \quad \text{for } 1 \leq i, j \leq n$$

Based on the preceding formulation, we can express Winograd's algorithm by the program code given in Listing 4.7.

Analysis of Winograd's Algorithm

Table 4.1 gives the count of scalar additions and multiplication executed by Winograd's algorithm, assuming n (the shared dimension for the input matrices) is even. Table 4.2 contrasts these numbers, for n being a power of 2, with the standard algorithm and Strassen's algorithm.

	Additions	Multiplications
Preprocessing of A	$m(n/2-1)$	$m(n/2)$
Preprocessing of B	$r(n/2-1)$	$r(n/2)$
Compute entries of C	$mr(n+n/2+1)$	$mr(n/2)$
Total	$[m(n-2) + r(n-2) + nr(3n+2)]/2$	$(mnr + mn+nr)/2$

Table 4.1 The counts of additions and multiplications for Winograd's algorithm for even n (shared dimension).

```

Input: A is m×n matrix and B is n×r matrix
Output: The matrix C = A×B; C is m×r

void MatrixMultiply(int[,] A, int[,] B, ref int[,] C)
{ C = new int[m+1,r+1];
  nby2 = n/2;
  // Compute row factors
  for i = 1 to m // i ranges over 1st dimension of A
  { row[i] = 0;
    for j = 1 to nby2
      row[i] = row[i] + A[i,2*j-1]*A[i,2*j];
    }

  // Compute column factors
  for i = 1 to r // i ranges over 2nd dimension of B
  { col[i] = 0;
    for j = 1 to nby2
      col[i] = col[i] + B[2*j-1,i]*B[2*j,i];
    }

  // Compute matrix C
  for i = 1 to m
    for j = 1 to r
      { C[i,j] = -row[i]-col[j];
        for k = 1 to nby2
          C[i,j] = C[i,j] + (A[i,2*k-1]+B[2*k,j])*(A[i,2*k]+B[2*k-1,j]);
        }

  // Add terms for odd dimension
  if (2*nby2 != n)
    for i = 1 to m
      for j = 1 to r
        for k = 1 to nby2
          C[i,j] = C[i,j] + A[i,n]*B[n,j];
    }
}

```

Listing 4.7 Winograd's matrix multiplication algorithm.

Divide and Conquer

	Additions	Multiplications
Standard algorithm	$n^3 - n^2$	n^3
Strassen's algorithm	$6n^{2.81} - 6n^2$	$n^{2.81}$
Winograd's algorithm	$(3n^3 + 4n^2 - 4n)/2$	$(n^3 + 2n^2)/2$

Table 4.2 The counts of additions and multiplications for various matrix multiplication algorithms; the input matrices are of size $n \times n$.

Concluding Remarks

The best complexity currently known on matrix multiplication is $O(n^{2.376})$ for Coppersmith–Winograd algorithm [Cop90]. However, the algorithm is of little practical significance because of the very large constant coefficient hidden by the Big O Notation.

Matrix multiplication has a particularly interesting interpretation in counting the number of paths between two vertices in a graph. Let A be the adjacency matrix of a graph G , meaning $A[i,j]=1$ if there is an edge between i and j ; otherwise, $A[i,j]=0$. Now consider the square of this matrix, $A^2 = A \times A$. If $A^2[i,j] \geq 1$, this means that there exists a value k such that $A[i,k]=A[k,j]=1$, so i to k to j is a path of length 2 in G . More generally, $A^n[i,j]$ counts the number of paths of length exactly n (edges) from i to j . This count includes nonsimple paths, where vertices are repeated, such as i to k to i .

4. Solved Exercises

1. Use backward substitution to solve the following recurrence: $T(1) = O(1)$; $T(n) = T(n/2) + \log n$.

Solution:

$$\begin{aligned}
 T(n) &= T(n/2) + \log n \\
 &= [T(n/4) + \log n/2] + \log n \\
 &= [T(n/8) + \log n/4] + \log n/2 + \log n \\
 &= T(n/2^i) + \log n/2^{i-1} + \dots + \log n/2 + \log n \\
 &= T(n/2^i) + \sum_{k=0}^{i-1} \log \left(\frac{n}{2^k} \right) \\
 &= T(n/2^i) + \sum_{k=0}^{i-1} (\log n - \log (2^k)) \\
 &= T(n/2^i) + \sum_{k=0}^{i-1} \log n - \sum_{k=0}^{i-1} \log (2^k) \\
 &= T(n/2^i) + i \log n - \sum_{k=0}^{i-1} k \\
 &= T(n/2^i) + i \log n - \frac{(i-1)i}{2}
 \end{aligned}$$

The recurrence will reach the base case after $\log n$ iterations. Assign $\log n$ to i :

$$T(n) = T\left(\frac{n}{2^{\log n}}\right) + \log n \cdot \log n - \frac{(\log n - 1)\log n}{2} = T(1) + O(\log^2 n) = O(\log^2 n).$$

2. Solve the following recurrence: $T(1) = 1$; $T(n) = T(n-1) + 1/n$.

Solution:

$$T(n) = 1/n + 1/(n-1) + 1/(n-2) + \dots + 1 \leq \int_1^n \frac{1}{x} dx = \ln x \Big|_1^n = \ln n$$

3. Given the recurrence $T(n) = 4T(n/2) + n^k$, what is the largest value of exponent k such that $T(n)$ is $O(n^3)$? Assume that $k \geq 0$.

Solution: Recall the Master Theorem. The solution of $T(n) = aT(n/b) + n^k$ is given as follows:

$$\begin{aligned}
 T(n) &= O(n^k) && \text{if } a < b^k \\
 T(n) &= O(n^k \log n) && \text{if } a = b^k \\
 T(n) &= O(n^p), p = \log_b a && \text{if } a > b^k
 \end{aligned}$$

Here, we have $T(n) = 4T(n/2) + n^k$. Here $p = \log_2 4 = 2$.

Divide and Conquer

If $k < 2$ then $a > b^k \Rightarrow T(n) = O(n^2)$, hence $T(n)$ is $O(n^3)$.

If $k = 2$ then $a = b^k \Rightarrow T(n) = O(n^2 \log n)$, hence $T(n)$ is $O(n^3)$.

If $k = 3$ then $a < b^k \Rightarrow T(n) = O(n^3)$, hence $T(n)$ is $O(n^3)$.

If $k > 3$ then $a < b^k \Rightarrow T(n) = O(n^k)$, for $k > 3$. Hence $T(n)$ is not $O(n^3)$.

Hence, we conclude that this holds for $k \leq 3$.

4. Use the recursion tree to find an upper bound on the solution for the following recurrence (assume that n is a power of 7):

$$T(n) = \begin{cases} 5T\left(\frac{n}{7}\right) + \log_7 n & n > 1 \\ 1 & n = 1 \end{cases}$$

Solution: Each node in the recursion tree gives rise to five children whereas the height of the recursion tree is $\log_7 n$. Thus, $T(n)$ corresponds to the following:

$$\sum_{i=0}^{\log_7 n} \left(5^i \log_7 \left(\frac{n}{7^i} \right) \right) = \sum_{i=0}^{\log_7 n} \left(5^i (\log_7 n - \log_7 7^i) \right) = \sum_{i=0}^{\log_7 n} \left(5^i (\log_7 n - i) \right) \leq \sum_{i=0}^{\log_7 n} \left(5^i (\log_7 n) \right) = \log_7 n \sum_{i=0}^{\log_7 n} (5^i)$$

The solution to the last summation is:

$$\sum_{i=0}^{\log_7 n} (5^i) = \frac{5^{1+\log_7 n} - 1}{4} = \frac{5(5^{\log_7 n}) - 1}{4} \leq \frac{5(5^{\log_5 n}) - 1}{4} = \frac{5(n) - 1}{4} = O(n)$$

$T(n)$ is, therefore, $O(n \log n)$. This turns out to be a rather loose bound because of the dropping of “ $-i$ ” in the above derivation. If we note that $T(n) \leq 5T(n/7) + n$ then, using the Master Theorem, Case 2 applies (because, for $a=5$, $b=7$, $k=1$, we have $a < b^k$) and we have $T(n) = O(n)$. Even then, this is not the tightest possible bound. The Master Theorem can be restated for the case where the nonrecursive term (in the recurrence) $f(n)$ is not a polynomial in terms of n , by essentially comparing the asymptotic order of $f(n)$ to $n^{\log_b a}$. We test the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{\log_7 n}{n^{\log_7 5}}$, and find that there is dominance in the denominator. Case 3 applies, so the solution is $T(n) = O(n^{\log_7 5})$.

5. Finding the majority element by *elimination of a noncandidate*. The majority element exhibits the following property:

The majority element is unaffected (i.e., remains the majority in the modified sequence) when we remove one of its occurrences and remove one other element.

Now consider two different elements a and b . If neither of them is the majority, we can safely remove both a and b and the majority will be unaffected. Otherwise, if either a or b is the majority, then, by the preceding property, we can remove both a and b and the majority will be unaffected. This suggests the following approach to *finding a majority candidate (FindMC)* in a sequence $A[1..n]$. (Note: After finding a majority candidate, we count its total occurrences in the original input to determine whether it is a majority.)

The elements are scanned from first to last. We use two variables, C (candidate element) and M (multiplicity). When we consider $A[i]$, C is the only candidate majority for the sequence $A[1..i-1]$ and M is the number of times C occurred in $A[1..i-1]$ less the times C was eliminated. If $A[i] \neq C$ then we can remove $A[i]$ and one copy of C by skipping over $A[i]$ and decrementing M ; otherwise, if $A[i] = C$, we skip over $A[i]$ and increment M . In this process we cannot let M be 0; therefore, C is reset to a new candidate every time M becomes 0. When all elements are scanned, we check M . If $M = 0$, this implies that there is no majority candidate; otherwise, C is a majority candidate. The following listing shows the algorithm.

```

Input: A positive integer array A[1..n]
Output: Return the majority element or zero if no majority is found

int Majority(int[] A, int lo, int hi)
{ int mc = FindMC(A,lo,hi);
  if (mc > 0)
    if (Count(A,lo, hi,mc) > (hi-lo+1)/2) return mc;
  return 0;
}

int FindMC(int[] A, int lo, int hi)
{ // return a majority candidate (C) by noncandidate elimination
  // uses two variables C: candidate element; M: multiplicity of the candidate
  int C = A[lo]; int M = 1;
  for(int i=lo+1; i <= hi; i++)
  { if (M == 0) // reset to using a new candidate
    { C = A[i]; M = 1; }
    else
    { if (A[i]== C) M++;
      else M--; // remove A[i] and remove one copy of C
    }
  }
  if (M > 0) return C;
  else return 0;
}

```

4. Exercises

1. Using the characteristic equation method, find the solutions for the following three recurrences using Θ -notation:
 - a. $T(n) = T(n-1) - 6T(n-2)$.
 - b. $T(n) = 3T(n-1) - T(n-2) - 3T(n-3)$.
 - c. $T(n) = 2T(n-1) - T(n-2)$.

2. Use the Master Theorem to find the complexity of the following functions:
 - a. $T(n) = 2T(n/4) + 7n - 15$.
 - b. $T(n) = 9T(n/3) + 3n^2 + 5n + 16$.
 - c. $T(n) = 8T(n/2) + 15$.

3. In the following problem, assume that n is a power of 3.
 - a. $T(n) = 2T(n/4) + 7n - 15$.
 - b. $T(n) = 9T(n/3) + 3n^2 + 5n + 16$.
 - c. $T(n) = 8T(n/2) + 15$.

3. In the following problem, assume that n is a power of 3.

$$T(n) = \begin{cases} 3T\left(\frac{n}{3}\right) + \Theta(n^2) & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

- a. Use the Master Theorem to find asymptotic upper and lower bounds for $T(n)$.
 - b. Use a recursion tree to find asymptotic upper and lower bounds for $T(n)$.
 - c. Use induction to verify your upper and lower bounds.
4. Solve the recurrence $T(n) = 2T(\sqrt{n}) + \log_2 n$. Hint: Consider using *change of variable* twice.
5. In the following problem, you may assume that n is a power of 3. Suppose there are three alternatives for dividing a problem of size n into smaller-size subproblems: If you solve three subproblems of size $n/2$, then the cost for combining the solutions of the subproblems to obtain a solution for the original problem is $\Theta(n^2 \sqrt{n})$; if you solve four subproblems of size $n/2$, then the cost for combining the solutions is $\Theta(n^2)$; if you solve five subproblems of size $n/2$, then the cost for combining the solutions is $\Theta(n \log n)$. Which alternative do you prefer and why?
6. Given the recurrence: $T(n) = T(n/2) + 3T(n/3) + 4T(n/4) + 7n^2$, complete the following:
 - a. Show that $T(n) = O(n^3)$.
 - b. Show that $T(n) = \Omega(n^{3/2})$.
7. Use the Master Theorem to solve the recurrence: $T(n) = 9T(n/3) + (n+2)(n-2)$. Assume that $T(1) = 1$, and that n is a power of 3.
8. Show how the Makeheap algorithm given in Section 2.2 can be derived using a divide-and-conquer approach. In this context, write a top-down recursive version of the algorithm and the recurrence equations for its running time.
9. Show that if there are 26 coins with one counterfeit coin (either heavier or lighter than a genuine coin), the counterfeit coin can be found in three weighings. Generalize this to find an expression for the number of weighings needed to find the counterfeit coin among n coins. Hint: Consider dividing the pile into three parts of about $n/3$ coins each.

10. The straightforward algorithm of scanning an array of n elements twice to return the largest element and the second-largest element does $(n-1)+(n-2)$ comparisons. Design an algorithm that does about $n+\log n$ comparisons.
11. Show that for a sorted input sequence $A[1..n]$, the majority element can be found using at most $n/2+2$ comparisons. Hint: For a sorted input, which position is guaranteed to contain the pivot element?
12. Given a sorted array of distinct integers $A[1..n]$, an index i is called an *anchor* if $A[i]=i$. Design a divide-and-conquer algorithm for finding an anchor in $A[1..n]$ if one exists. Your algorithm should run in $O(\log n)$ time.
13. Consider the divide-and-conquer majority algorithm given in Section 4.4. Given an input of n *distinct* elements, show that the algorithm does $2n$ comparisons (counting only the comparisons that are executed by the *Count()* method). Does this result depend on n being a power of 2?
14. Consider the divide-and-conquer algorithm for polynomial multiplication given in this chapter. Given $A(x) = 1 + 2x + 4x^2 + x^3 - 2x^4$ and $B(x) = 3 - x + 2x^2 + 3x^3 + 6x^4$, find the polynomials $A_0(x)$, $A_1(x)$, $B_0(x)$, and $B_1(x)$. Also, find the expression for $A(x)B(x)$ in terms of these polynomials as computed by the algorithm.
15. Consider the following ThreeSort() algorithm:

```

ThreeSort(A[i..j])
{
  n = j-i+1; // number of elements
  if (n==1) return;
  if (n==2) and (A[i] > A[j]) then swap A[i] with A[j]
  else if (n > 2)
  {
    third = round(n/3);
    ThreeSort(A[i..j-third]); // sort first 2/3rds
    ThreeSort(A[i+third..j]); // sort last 2/3rds
    ThreeSort(A[i..j-third]); // sort first 2/3rds
  }
}

```

Let $C(n)$ be the worst case number of element comparisons performed by the preceding algorithm.

- a. Find the recurrence equations for $C(n)$ including base equations.
 - b. Use master theorem to find a Θ -expression for $C(n)$.
16. A *Latin square* is an $n \times n$ grid where each row and column contains the numbers 1 to n . Design a divide-and-conquer algorithm to construct a Latin square of size n (assume n is a power of 2).
 17. Suppose you are given an unsorted array A of integers in the range 0 to n except for one integer, denoted as the *missing number*. Assume $n=2^k-1$. Design an $O(n)$ divide-and-conquer algorithm to find the missing number.
 18. Let A be an integer array consisting of two sections, one with numbers increasing followed by a section with numbers decreasing. Design an $O(\log n)$ algorithm to find the index of the maximum number. Hint: Divide the array into three equal size sections and devise a way to safely throw away one of them.
 19. You are given a sequence of numbers $A = a_1, a_2, \dots, a_n$. An exchanged pair in A is a pair (a_i, a_j) such that $i < j$ and $a_i > a_j$. Note that an element a_i can be part part of m exchanged pairs, where m is $\leq n-1$, and that the maximal possible number of exchanged pairs in A is $n(n-1)/2$, which is achieved if the array is sorted in descending order. Develop a divide-and-conquer algorithm that counts the number of exchanged pairs in A in $O(n \log n)$ time. Argue why your algorithm is correct, and why your algorithm takes $O(n \log n)$ time.

Divide and Conquer

20. A number is *simple* if it consists of repeated decimal digits. For example, 3333 and 7777 are simple numbers. Devise an algorithm to multiply two n -digit simple numbers in $O(n)$ time, where we count a one-digit addition or multiplication as a basic operation. Hint: Use divide-and-conquer. To justify the running time, give a recurrence (and its solution) for your algorithm. You may assume that n is a power of 2.
21. Show how Strassen's algorithm computes the matrix product of the following matrices.
- a.
- $$A = \begin{pmatrix} 3 & 1 \\ 4 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 1 \\ 5 & -2 \end{pmatrix}$$
- b.
- $$A = \begin{pmatrix} 3 & 1 & 1 \\ 2 & 2 & 1 \\ 1 & 2 & -1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 1 & 1 \\ 5 & -2 & 1 \\ 2 & 2 & -1 \end{pmatrix}$$
22. Assume you are given the procedure $Strassen(A, B, n)$ which implements Strassen's algorithm. Recall that the procedure computes the product of two square matrices A and B of size $n \times n$.
- a. By calling $Strassen(A, B, n)$, show how to multiply an $n \times kn$ matrix by a $kn \times n$ matrix for integer $k > 1$.
- b. By calling $Strassen(A, B, n)$, show how to multiply a $kn \times n$ matrix by an $n \times kn$ matrix for integer $k > 1$.

Briefly describe your algorithm and analyze its time complexity as a function of n and k .

5. Dynamic Programming

As an algorithm-design technique, dynamic programming centers around expressing $S(n)$, the solution to a problem of size n , in terms of solutions to subproblems $S(k)$ for $k < n$. Often there needs to be further parameters for $S(n)$ such as $S(n,r)$. Thus $S(n,r) = f(S(n',r'), S(n'',r''), \dots)$. However, a direct recursive approach to solving such a problem based on the recursive formulation would result in encountering certain instances of subproblems more than once, which often leads to an exponential-time algorithm. To avoid this, it is a characteristic of dynamic programming that the recursive formulation is subsequently transformed into a bottom-up iterative implementation that does *store* and subsequent *lookup* of solutions to subproblems. We illustrate the technique through several dynamic-programming algorithms for different problems.

The development of dynamic programming is credited to Richard Bellman (1920-1984) who gave the technique its name [Bel57]. However, in the 1950s computer programming was in its infancy and the phrase *dynamic programming* has little to do with computer programming as we know it today. According to Bellman's accounts, he used the word *programming* as a synonym for planning and *dynamic* as a synonym for time-varying.

5.1 Computing the Binomial Coefficients

The number of subsets of size r chosen from of a set of size n is denoted by $C(n,r) \equiv \binom{n}{r}$ — read as “the combination of n elements chosen r at a time” or, more simply, “ n chose r ” — where n and r are nonnegative integers and $0 \leq r \leq n$. For example, $C(4,2)$ is the number of subsets of size 2 chosen from a 4-element set. In this case, $C(4,2) = 4 \cdot 3 / 2! = 6$, which is equivalent to counting the subsets of size 2 chosen from the set $\{a,b,c,d\}$ — there are 6 subsets, namely $\{a,b\}$, $\{a,c\}$, $\{a,d\}$, $\{b,c\}$, $\{b,d\}$, and $\{c,d\}$.

The $C(n,r)$ s are known as the binomial coefficients because they appear as the factors in the expanded form of the binomial $(x+y)^n$; namely,

$$(x+y)^n = C(n,0) x^n y^0 + C(n,1) x^{n-1} y^1 + \dots + C(n,i) x^{n-i} y^i + \dots + C(n,n) x^0 y^n.$$

Given n and r , $C(n,r)$ can be evaluated using Equation 5.1. However, an alternative formula that does not involve multiplication or division is given by Equation 5.2. This formula is an example of a combinatorial identity and is known as *Pascal's Identity*.

$$C(n,r) = n! / ((n-r)! r!) = (n (n-1) \dots (n-r+1)) / r! \tag{5.1}$$

$$C(n,r) = C(n-1,r-1) + C(n-1,r) \tag{5.2}$$

The justification for Pascal's Identity is rather simple. Consider the n -th element and its presence in the subsets of size r ; the subsets of size r either include or exclude the n -th element. If the n -th element is chosen, we have to choose the remaining $r-1$ elements from the first $n-1$ elements, which can be done in $\binom{n-1}{r-1}$ ways. On the

Dynamic Programming

other hand, if the n -th element is not chosen, we have to choose r elements from the first $n-1$ elements, which can be done in $\binom{n-1}{r}$ ways.

The recursive formula in Equation 5.25.1 gives the basis for a recursive algorithm, but we need some base cases. Through the reduction process, the second parameter might reach zero and, in this case, the number of subsets of size zero is 1 (i.e., there is only one subset; namely, the empty set). Thus, Equation 5.3 is an appropriate base case. Furthermore, we do not like to deal with cases where the second parameter exceeds the first parameter — this happens because the second term in the RHS of Equation 5.2 allows the first parameter to decrease while the second parameter remains unchanged. Hence, we use Equation 5.4 as one more base case.

$$C(n,0) = 1 \quad 5.3$$

$$C(n,n) = 1 \quad 5.4$$

Equations 5.2, 5.3, and 5.4 readily translate into the following recursive (and novice) algorithm:

```
int C(int n, int r)
{ if (r==0) return 1;
  else if (n==r) return 1;
  else return C(n-1,r-1)+C(n-1,r);
}
```

However, there is a major source of inefficiency in this recursive algorithm. It is not difficult to see that certain subproblem instances (a pair of (n,r) values for the input parameters defines an instance) are being solved more than once.

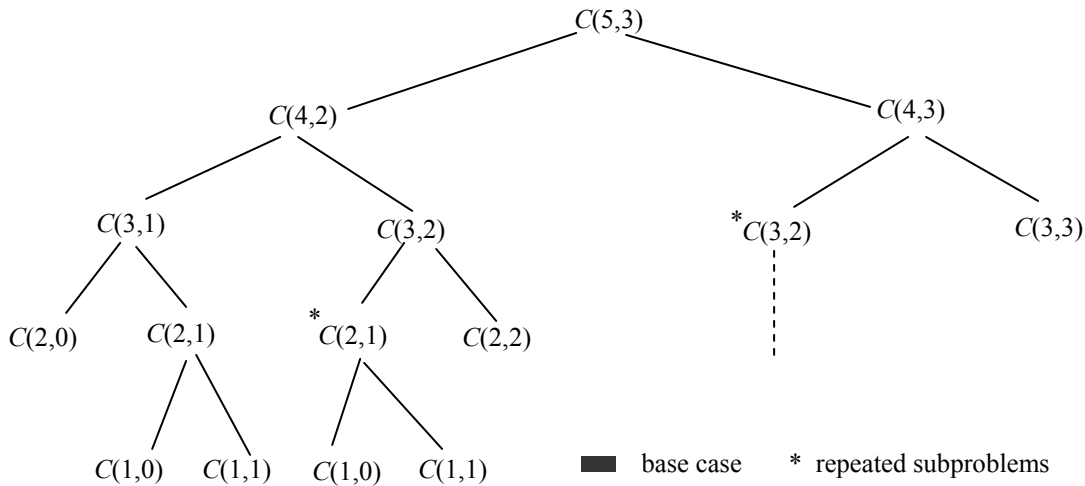


Figure 5.1 The tree of recursive calls for computing $C(5,3)$ using a novice recursive algorithm.

As illustrated in Figure 5.1, we see that the subproblems $C(3,2)$ and $C(2,1)$ are encountered more than once (repeated occurrences are marked with *). Therefore, this algorithm will end up solving a large number (much more than necessary) of subproblems. We already know that there are $(n+1) \times (r+1)$ different problems because the domain for the first parameter is $[0,n]$ and for the second parameter is $[0,r]$. Wouldn't it be more efficient to evaluate $C(n,r)$ bottom-up (from the smallest-size subproblem to the largest-size subproblem) and remember previous solutions? We can use an $(n+1) \times (r+1)$ table (matrix) where the (i,j) -entry stores the solution for $C(i,j)$.

As illustrated by Figure 5.2, the solution becomes simply filling a matrix one row at a time, where each row corresponds to a value of the first parameter. The corresponding iterative algorithm is given in Listing 5.1. This algorithm computes at most $(n+1) \times (r+1)$ matrix entries requiring a constant time per entry. Therefore, the algorithm has $\Theta(nr)$ running time.

Spacewise, we observe that this algorithm has $\Theta(nr)$ space complexity. However, the space complexity can be reduced by noting that in computing the i -th row, we only need the $(i-1)$ -th row and no other rows. This suggests that the algorithm be modified as given in Listing 5.2, where we use a matrix with 2 rows \times $(r+1)$ columns leading to $\Theta(r)$ space complexity.

$i \backslash j$	0	1	2	3	4	...	r
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
⋮							
n							



 $C(n,r)$ is $C[n,r]$

Figure 5.2 The matrix $C[0..n, 0..r]$ corresponding to bottom-up evaluation of $C(n,r)$.

```

// returns the binomial coefficient C(n,r)
int Comb(int n, int r)
{ int[,] C = new int[n+1,r+1]; // C matrix is (n+1) rows x (r+1) columns
  C[0,0] = 1;
  for(int i=1; i <= n; i++)
    for(int j=0; j <= r; j++)
      if (j==0) C[i,j] = 1;
      else if (j==i) C[i,j] = 1;
      else if (j < i) C[i,j] = C[i-1,j-1] + C[i-1,j];
  return C[n,r];
}

```

Listing 5.1 An iterative algorithm to compute $C(n,r)$ using $\Theta(nr)$ time and $\Theta(nr)$ space.

Dynamic Programming

```
// returns the binomial coefficient C(n,r)
int Comb(int n, int r)
{ int[,] C = new int[2,r+1]; // C matrix is 2 rows x r+1 columns
  int cur_row, prev_row;
  C[0,0] = 1;
  for(int i=1; i <= n; i++)
  { cur_row = i % 2; prev_row = (i-1) % 2;
    for(int j=0; j <= r; j++)
      if (j==0) C[cur_row,j] = 1;
      else if (j==i) C[cur_row,j] = 1;
      else if (j < i) C[cur_row,j] = C[prev_row,j-1] + C[prev_row,j];
    }
  return C[n % 2,r];
}
```

Listing 5.2 An efficient algorithm to evaluate $C(n,r)$ using $\Theta(nr)$ time and $\Theta(r)$ space.

5.2 The 0/1-Knapsack Problem

The *Knapsack problem* is modeling the situation where a burglar enters a house and he finds a collection of items of various values and sizes (or weights). The thief likes to pack his sack with as many items as the sack capacity permits while maximizing the value of the items collected. More formally, given a sack with capacity C and n items with values v_1, v_2, \dots, v_n and, respectively, sizes s_1, s_2, \dots, s_n . Let S denote a subset of $\{1, 2, \dots, n\}$. The 0/1-knapsack problem is to find S , where,

$$\text{Maximize } f(S) = \sum_{i \in S} v_i$$

over all S

Subject to:

$$\sum_{i \in S} s_i \leq C$$

The knapsack problem is an example of an *optimization* problem, where the goal is to *maximize* (or *minimize*) some expression subject to some constraints. The expression to be maximized (or minimized) is known as the *objective function*. There are several versions (variants) of the knapsack problem. The problem stated here is known as the *0/1-knapsack* problem. This version is characterized by having a single instance of each item and the decision regarding each item is either to add or not to add to the sack.

A brute-force approach to solve this problem will enumerate all possible subsets S , one at a time, and for each S , check the constraint and compare $f(S)$ with the maximum-value found thus far. This will have $\Omega(2^n)$ running time because it has to generate and examine 2^n subsets. Thus, we conclude that using such an algorithm is only feasible if n is small, for example, $n \leq 30$. Let us try dynamic programming.

In dynamic programming, the solution to a problem is viewed as a series (sequence) of decisions. We note that the decision regarding the i -th item is whether to include it in the sack; this is partly dependent on the sack capacity. Thus, we define $V(i, j)$ as the *maximum value* obtained by considering items $\{1, 2, \dots, i\}$ and a sack of capacity j . This implies that the solution to the original problem is given by $V(n, C)$.

Next, for size reduction, we have to relate $V(i, j)$ to $V(i-1, j')$. *How?* Well, $V(i, j)$ considers one more item (the i -th item) in relation to $V(i-1, j')$. If the i -th item is not included in the solution for $V(i, j)$, $V(i, j)$ is simply $V(i-1, j)$; otherwise, if the i -th item is included, we get its value v_i plus $V(i-1, j-s_i)$ — the maximum value of filling the remaining sack capacity $(j-s_i)$ with items $\{1, 2, \dots, i-1\}$. Note that we can include the i -th item only if the sack capacity permits, i.e., if $j \geq s_i$. Because our goal is to maximize the value of items picked, the choice whether to include the i -th item is resolved in favor of the choice that leads to a higher value, which leads to Equation 5.5. Finally, as base cases, we need Equation 5.6, where $V(i, j) = 0$ if we are to consider none of the items ($i=0$) or the sack has zero capacity ($j=0$).

$$V(i, j) = \text{maximum of } \{ V(i-1, j), \text{ (if } j \geq s_i) \ V(i-1, j-s_i) + v_i \} \tag{5.5}$$

$$V(i, j) = 0 \text{ if } i = 0 \text{ or } j = 0 \tag{5.6}$$

Implementation and Complexity Analysis

The preceding recursive formulation should not be implemented using recursion because it will lead to the same problem we saw earlier with evaluating the combination formula; namely, many subproblem instances will be

Dynamic Programming

encountered repeatedly. Therefore, we resort to a bottom-up iterative implementation. In this case, the solution is simply to fill a matrix $V[0..n,0..C]$, where the entry $V[i,j]$ corresponds to $V(i,j)$. The matrix needs to be filled row-wise (because a row corresponds to a level in the tree of recursive calls) starting from row 0 (the base case). The iterative algorithm is given as the *Iterative_KS()* method shown in Listing 5.3.

It is clear that the order of running time is $\Theta(nC)$. Clearly, this algorithm is much faster than the brute-force algorithm because nC is a much smaller number than 2^n for large n . But is this algorithm fast enough for large input? Well, for large n (for example, $n=100,000$) and relatively small C (for example, $C=100$), this algorithm will compute a matrix of 10 million entries, which is expected to finish in a few seconds (filling an entry corresponds to about 10 instructions; these days, a typical processor can execute 100 million instructions per second). What if C is large, for example, $C=100,000$? In this case, things get ugly. For one thing, the matrix will be huge, having a total of 10 billion entries. A computer with only 1 Gigabyte RAM will do lots of swapping between physical memory and the swap area on disk and the algorithm will run painfully slow.

The other interesting observation is that every time we double C (i.e., an increase of input-size by 1 bit), the running time will double. Thus, we observe that in the expression for the running time nC , the factor C is more or less the culprit behind the algorithm's inefficiency. (Note: Doubling the number of items n requires an increase in input size by more than 1 bit because we have to provide size and value information for the additional items). Since C is encoded using $\log C$ bits, the algorithm running time is $\Theta(nC)=\Theta(n2^{\log C})$. Thus, the running time is *exponential* in the size of the input expressed in bits. Such an algorithm is said to have *pseudopolynomial running time*.

```
Input: Number of items n; sack capacity C; item values v[1..n]; item sizes s[1..n]
Output: return maximum sack value and the solution matrix V[0..n,0..C]

int Iterative_KS(int n, int C, int[] v, int[] s, ref int[,] V)
{ V = new int[0..n,0..C];
  for(int j=0; j<= C; j++) V[0,j] = 0; // Base step

  for(int i=1; i<= n; i++)
    for(int j=0; j <= C; j++)
      { V[i,j] = V[i-1,j];
        if (j >= s[i])
          { int t = V[i-1,j-s[i]]+v[i];
            if (t > V[i,j]) V[i,j] = t;
          }
      }
  return V[n,C];
}
```

Listing 5.3 An iterative algorithm for the 0/1-knapsack problem.

Example 5.1 Figure 5.3 shows the matrix representing the solution to the 0/1-knapsack problem for sack capacity=16 and four items with sizes: 3, 5, 7, 8 and values: 4, 6, 7, 9.

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4
2	0	0	0	4	4	6	6	6	10	10	10	10	10	10	10	10	10
3	0	0	0	4	4	6	6	7	10	10	11	11	13	13	13	17	17
4	0	0	0	4	4	6	6	7	10	10	11	13	13	15	15	17	19

Figure 5.3 The matrix $V[0..n, 0..C]$ computed by the dynamic-programming algorithm for the 0/1-knapsack problem.

5.3 Recovering Solution Components

The first and crucial step to develop a dynamic-programming algorithm is to come up with a proper notation that represents the value of the (optimal) solution in terms of the problem's data. For the 0/1-knapsack problem, this is $V(i,j)$ and its associated meaning. As can be seen in the formulation for the 0/1-knapsack problem, it suffices that the formulation provides a recurrence equation for the optimal solution as a sum of item values and not the items themselves. However, we are also interested in finding the items that achieve the optimal solution. As it turns out, the choices regarding the inclusion of any item are already implied in the way various $V(i,j)$ entries are computed. A simple process of reasoning backward starting from $V(n,C)$ can be used to recover the items that represent the optimal solution. We can compare $V(n,C)$ to $V(n-1,C)$. If $V(n,C) > V(n-1,C)$, this definitely means that the n -th item is part of the optimal solution, and then we can recover the rest of the items by examining $V(n-1,C-s_n)$. On the other hand, if $V(n,C)=V(n-1,C)$, we conclude that the n -th item is not part of the optimal solution. Note that in the latter case, it is also possible that $V(n,C)=V(n-1,C-s_n)+v_n$, which means that the n -th item is part of the optimal solution. This is quite plausible because the set of items corresponding to an optimal filling of the sack might not be unique. If we are merely interested in recovering one solution set, then we can take " $V(n,C)=V(n-1,C)$ " to mean that the n -th item is not part of the solution. Listing 5.4 shows a program method `KS_Items()` that returns (only one set, and not all possible sets) the items representing the optimal solution for the knapsack problem.

In summary, we can state the following principle regarding dynamic-programming algorithms.

DP formulation principle: The recurrence formulation for a dynamic-programming algorithm is only for the optimal value (of the objective function) and not the items that represent the optimal value. The items that represent the (optimal) solution are recoverable by working backward from the solution to the recurrence.

```

Input: The solution matrix V; item sizes s[1..n]; sack capacity C
Output: The items (as space separated string) that represent the optimal solution

string KS_Items(int n, int C, int[] s, int[,] V)
{
    string outstr="";
    int j = C;
    for(int i=n; i > 0; i--)
        if ( V[i,j] > V[i-1,j] )
            { outstr= i + " " + outstr; j = j-s[i]; }
    return outstr;
}

```

Listing 5.4 An algorithm for recovering the items of the optimal solution for the 0/1-knapsack problem.

Exercise 5.1 Modify the algorithm in Listing 5.4 to return not just one set of items, but all sets that correspond to the optimal solution. Hint: At a junction point (where the include/exclude choices of an item are both possible) gives rise to two solutions (paths), push the partial solution collected thus far into the stack and follow the other path to completion. Then process things off the stack.

5.4 The Subset-Sum Problem

The following problem is a simplified version of the 0/1-knapsack problem and is known as the *subset-sum problem*.

The Subset-Sum Problem. Given a positive integer K and a set $A = \{a_1, a_2, \dots, a_n\}$ of n positive integers, determine whether there exists a subset of A whose sum of elements = K .

A brute-force approach to solve this problem by enumerating all subsets of A will have $\Omega(2^n)$ running time. Such an algorithm, because of its exponential time, is practical only for small values of n (i.e., $n \leq 30$).

Reducing the Subset-Sum Problem to the 0/1-Knapsack Problem

The subset-sum problem can be viewed as a special case of the 0/1-knapsack problem. Construct an instance of the 0/1-knapsack problem with $C=K$, and n items with the a_i s being both item values and item sizes. Now, the problem becomes: Maximize the sum of a set a_i s subject to their sum $\leq C$. Clearly if there is a solution with the sum of item sizes = C , it will be preferred (because it has a higher value for the objective function) over any solution whose sum of item sizes $< C$. After the knapsack problem is solved, we check if the sum of item sizes of the optimal solution = C . If so, then the answer is true for the original subset-sum problem; otherwise, the answer is false.

A Specialized Dynamic-Programming Algorithm for the Subset-Sum Problem

The subset-sum problem has less data than the 0/1-knapsack problem; therefore, we expect that it has a simpler dynamic-programming formulation. As in the formulation for the knapsack problem, we view the solution as a sequence of decisions, one decision per element. The decision regarding an element a_i is whether the element is part of the solution. Thus, let us define $SubsetSum(i,m)$ is *true* [*false*] if there is [is not] a subset of the elements $\{a_1, a_2, \dots, a_i\}$ whose sum= m . Clearly, the solution to the original problem is given by $SubsetSum(n,K)$. Next, we express $SubsetSum(i,m)$ in terms of $SubsetSum(i-1,m')$. A true solution for $SubsetSum(i,m)$ either includes or does not include the i -th element. For the latter case, $SubsetSum(i-1,m)$ must be true. For the former case, which is preconditioned on $m \geq a_i$, $SubsetSum(i-1,m-a_i)$ must be true. This leads us to conclude, as stated by Equation 5.7, that $SubsetSum(i,m) = SubsetSum(i-1,m)$ logical-OR $SubsetSum(i-1,m-a_i)$. Equation 5.8 specifies the appropriate base case.

$$SubsetSum(i,m) = SubsetSum(i-1,m) \text{ or } (\text{if } m \geq a_i) \text{ } SubsetSum(i-1,m-a_i) \quad 5.7$$

$$SubsetSum(0,m) = \text{true} \text{ if } m = 0; \text{false, otherwise} \quad 5.8$$

The bottom-up implementation of this algorithm is given in Listing 5.5. Note that, as in the solution for the 0/1-knapsack problem, we are using a matrix $V[0..n,0..K]$ to store solutions of subproblems — i.e., $V[i,j]$ corresponds to $SubsetSum(i,j)$. Listing 5.6 gives the algorithm for recovering the elements representing the solution by working backward from $V[n,K]$.

Dynamic Programming

Input: A set of positive integers $A[1..n]$ and a positive integer K
 Output: returns true (and V matrix) iff there is a subset whose sum of elements = K

```
bool SubsetSum(int[] A, int n, int K, ref int[,] V)
{
    V = new int[n+1,K+1];
    V[0,0] = true;
    for(int j=1; j <= K; j++)
        V[0,j] = false;

    for(int i=1; i <= n; i++)
        for(int j=0; j <= K; j++)
            { V[i,j] = V[i-1,j];
              if (! V[i,j])
                  if (j >= A[i]) V[i,j] = V[i-1,j-A[i]];
            }
    return V[n,K];
}
```

Listing 5.5 A dynamic-programming algorithm for the subset-sum problem.

Input: A set of positive integers $A[1..n]$ and a positive integer K ;
 Boolean solution matrix V
 Output: The elements (as space-separated string) that sum to K

```
string SetFromV(int[] A, int n, int K, bool[,] V)
{
    string outstr="";
    int j = K;
    for(int i=n; i > 0; i--)
        if ( V[i,j] && ! V[i-1,j] )
            { outstr= outstr+ " " + A[i]; j = j-A[i]; }
    return outstr;
}
```

Listing 5.6 An algorithm for recovering the elements of a solution to the subset-sum problem.

Example 5.2 Figure 5.4 shows the matrix representing the solution to the subset-sum problem for $K=14$ and five items: 3, 7, 10, 4, 11.

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	T	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	T	F	F	F	T	F	F	T	F	F	F	F
3	T	F	F	T	F	F	F	T	F	F	T	F	F	T	F
4	T	F	F	T	T	F	F	T	F	F	T	T	F	T	T
5	T	F	F	T	T	F	F	T	F	F	T	T	F	T	T

Figure 5.4 The matrix $V[0..n, 0..K]$ computed by the dynamic-programming algorithm for the subset-sum problem.

5.5 Memoization

Recall that the problem with a recursive implementation of a dynamic-programming algorithm is that it will encounter repeated subproblem instances. Solving subproblems in the order of smallest size to largest size is one way around this problem. This corresponds to the iterative bottom-up approach that we have already discussed. Yet there is another way. Why not implement the recursive formulation as is (i.e., as a recursive program method) but augment it with a *lookup* capability. Thus, if a subproblem instance has already been solved, then lookup its solution; otherwise, solve it and store its solution. This approach is known as *memoization* (a short form for *memorization*). This technique is illustrated in Listing 5.7, where we modify the dynamic-programming algorithm for the subset-sum problem given in the previous section. Note first that we have changed $V[i,j]$ from being *true/false* to having one of three values: $-1=$ *unsolved*, $0=$ *solution is false*, $1=$ *solution is true*.

To assess the savings realized from memoization, the program in Listing 5.7 is augmented with the variable *sc* (subproblem count) to count the number of subproblems solved — i.e., the number of times the method *SubsetSum()* is called. The program is used to determine whether there are elements that sum to $K=1000$ in a set of 20 randomly-generated numbers between 10 and 1000. For such input, the bottom-up iterative algorithm solves 20,000 subproblems (i.e., the size of the solution matrix not counting the 0-row and 0-column). For the memoized algorithm, we find that the subproblem count is less than 1000 most of the time. Note that the subproblem count does not account for the initialization of the solution matrix. Of course, initializing all matrix entries might undo any savings realized from memoization. A reasonable approach is to run the recursive algorithm twice; first to determine and initialize all the needed entries and then run the normal memorized version. Another effective approach, which is applicable to implementing memoization in general, is to do away with matrix allocation altogether and, instead, use hashing to store the $i-j$ entries that are encountered during recursion. We leave the implementation of such an approach as an exercise, which is stated next.

Exercise 5.2 Modify the program in Listing 5.7 to include *SubsetSumHash()* method, which is essentially the same as *SubsetSum()*, but uses hashing to store and lookup encountered $i-j$ entries.

Dynamic Programming

```
class SubsetSumMemoization
{
    static int sc=0; // sc for subproblem count
    static int[] A; // Input: a set of numbers
    static int[,] V; // Solution matrix is global

    static void Main(string[] args)
    {
        Random rand = new Random();
        int n = 20;
        // Generate n random values between 10 and 1000
        A = new int[n+1];
        for(int i=1; i<=n; i++) A[i] = rand.Next(990)+10;

        int K=1000;
        V = new int[n+1,K+1];
        // Mark all subproblems as being unsolved
        for(int i=0; i<= n; i++)
            for(int j=0; j <= K; j++) V[i,j]=-1;

        string prdata = "";
        for(int i=1; i<=n; i++) prdata = prdata+ A[i] + " ";

        Console.WriteLine("SubsetSum Problem for {" + prdata+ "} and K=" + K);
        if (SubsetSum(n,K)==1)
        {
            Console.WriteLine(SetFromV(A, n, K, V));
            Console.WriteLine("Subproblem Count:" + sc);
        }
        else Console.WriteLine("No solution" );
    }

    static int SubsetSum(int i, int j) // Assume V is global
    {
        // Handle base cases
        if (i==0)
            if (j ==0) return 1;
            else return 0;

        // Check if problem is solved before
        if (V[i,j] != -1) return V[i,j];

        sc++; // Increment subproblem count

        // Note: To compute V[i,j] we cannot reference V because we cannot be sure
        // that the needed entry has been computed
        V[i,j] = SubsetSum(i-1,j);
        if ((j >= A[i]) && (V[i,j] != 1))
            V[i,j] = SubsetSum(i-1,j-A[i]);
        return V[i,j];
    }

    static string SetFromV(int[] A, int n, int K, int[,] V)
    {
        string items="";
        int j = K;
        for(int i=n; i> 0; i--)
            if ((V[i,j]==1) && (V[i-1,j]==0))
                { items = A[i] +" "+ items; j= j-A[i]; }
        return "{ " + items + " }";
    }
}
```

Listing 5.7 A dynamic-programming algorithm for the subset-sum problem implemented using memoization.

5.6 Longest Common Subsequence

Before we present the longest common subsequence problem, let us consider a similar but simpler problem. As in the string search problem (Section 2.4.7), we are given a short string (pattern) and long string (text), but now we are to find if the letters of the pattern appear in order (but possibly separated) in the text. If they do then the pattern is said to be a *subsequence* of the text. As an example, “ping” is a subsequence of “expert in programming”. To easily see this, we can italicize the subsequence matching with the pattern at the appropriate places in the text as: “ex*P*ert *IN* pro*GR*amming”. In general, we can do this using a finite automata (finite-state machine), as shown by Figure 5.5.

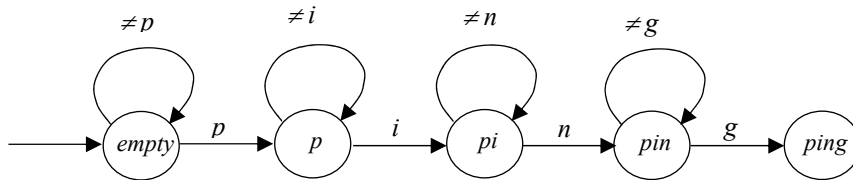


Figure 5.5 A finite automata for solving an instance of the subsequence problem.

The primary step is the reading of the next character from the input text and matching it with the current character from the pattern but we only advance along the pattern upon matching. This step is repeated until the last character of the pattern is matched (in which case the pattern does appear as a subsequence in the text) or we reach the end of the text. Thus, the solution is given by the following procedure:

```
bool SubSeq(string pat, string txt)
{ int i = 0; // i ranges over txt
  int j = 0; // j ranges over pat
  while ((i < txt.Length) && (j < pat.Length))
  { if (txt[i] == pat[j]) j++;
    i++;
  }
  return (j == pat.Length);
}
```

The Longest Common Subsequence Problem

What if the pattern does not occur in the text? In this case, we are interested in finding the longest subsequence that occurs both in the pattern and in the text. This is the *longest common subsequence problem*. Because the pattern and the text have symmetric roles, we will not distinguish between them and simply call them sequences A and B . Note that the previous automata-based method does not solve the problem; it only gives the *longest prefix* of A (the pattern) that is a subsequence of B , but the longest common subsequence of A and B is not always a prefix of A .

There are several applications where the longest common subsequence problem is encountered. A good example is in molecular biology. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four submolecules forming DNA. When biologists find a new sequence, they need to know what other sequences it is most similar to. A reasonable measure of how similar two sequences are would be the length of the longest common subsequence.

Let us now formalize a dynamic-programming algorithm for the longest common subsequence (LCS) problem. Assume that the input is given by the sequences $A = a_1, \dots, a_m$ and $B = b_1, \dots, b_n$. Let $lcs(i, j)$ be the length of the longest common subsequence of a_1, \dots, a_i and b_1, \dots, b_j . If $a_i = b_j$, then $lcs(i, j) = lcs(i-1, j-1) + 1$; this is because

Dynamic Programming

we can append $a_i (=b_j)$ to the LCS of $A[1..i-1]$ and $B[1..j-1]$. Otherwise, we have to disregard a_i or b_j ; thus, $lcs(i,j) = \text{maximum} \{ lcs(i,j-1), lcs(i-1,j) \}$. We also need to supply the appropriate base cases: $lcs(0,j) = lcs(i,0) = 0$ (i.e., the longest common subsequence is empty (of length 0) if either A or B is empty). In summary, the algorithm is given by the following:

$$lcs(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + lcs(i-1, j-1) & \text{if } a_i = b_j \\ \text{maximum} \{ lcs(i-1, j), lcs(i, j-1) \} & \text{if } a_i \neq b_j \end{cases}$$

Note that there are only $(m+1) \times (n+1)$ possible subproblems because there are only $m+1$ choices for i and $n+1$ choices for j . Hence, by treating the $lcs(i,j)$ s as matrix entries (instead of recursive calls) and computing these entries in the appropriate order, we get the following $\Theta(mn)$ -time algorithm.

```
int LCS(char[] A, int m, char[] B, int n)
{
    // Assume the input is given by char arrays: A[1..m] and B[1..n]
    int[,] L = new int[m+1,n+1]; // L matrix is (m+1) rows x (n+1) columns
    for(int i=0; i <= m; i++)
        for(int j=0; j <= n; j++)
            if((i==0) || (j==0)) L[i,j] = 0;
            else if (A[i]==B[j]) L[i,j] = 1 + L[i-1,j-1];
            else L[i,j] = max { L[i-1,j] , L[i,j-1] };
    return L[m,n];
}
```

The preceding algorithm is $\Theta(mn)$ space but it can be modified to be $\Theta(\text{minimum}(m,n))$ — see problem 4 in the end-of-chapter exercise.

Example 5.3 Figure 5.6 shows the matrix representing the solution to the longest common subsequence of the strings “singer” and “programmer”. Clearly, the longest common subsequence is “ger”, which is of length 3.

$i \backslash j$	0	1-p	2-r	3-o	4-g	5-r	6-a	7-m	8-m	9-e	10-r
0	0	0	0	0	0	0	0	0	0	0	0
1-s	0	0	0	0	0	0	0	0	0	0	0
2-i	0	0	0	0	0	0	0	0	0	0	0
3-n	0	0	0	0	0	0	0	0	0	0	0
4-g	0	0	0	0	1	1	1	1	1	1	1
5-e	0	0	0	0	1	1	1	1	1	2	2
6-r	0	0	0	0	1	2	2	2	2	2	3

Figure 5.6 The matrix $L[0..m, 0..n]$ derived from the DP recurrence for the longest common subsequence problem.

Exercise 5.3 Rewrite the recurrence for the LCS problem given above to avoid using (for any of its parameters) the empty sequence as a base case and use 1-element sequence instead.

Exercise 5.4 Modify the program code given previously into a CSharp program method $lcstr(A,B)$ that returns a longest common substring of the input strings A, B . Hint: First, compute the matrix L and then trace the path backward from $L[m,n]$.

5.7 Weighted-Interval Scheduling

Suppose we have a resource (such as a conference hall) that is to be shared. There are n requests for room reservations; each request has a specified start time s_i and finish time f_i . Also, each request has a weight (value) v_i . Two requests are *compatible* if their intervals do not overlap. A *feasible schedule* is a set consisting of compatible requests. In relation to the example of Figure 5.7, $\{1,3,6\}$ is an example of a feasible schedule with value = $v_1+v_3+v_6 = 2+4+3=9$. The problem has several versions (variants), depending on the objective, as follows:

1. Version 1: Find a schedule that maximizes the value of scheduled requests.
2. Version 2: Find a schedule that maximizes the number of scheduled requests.
3. Version 3: Find a schedule that maximizes resource utilization (i.e., minimizes the resource dle-time).

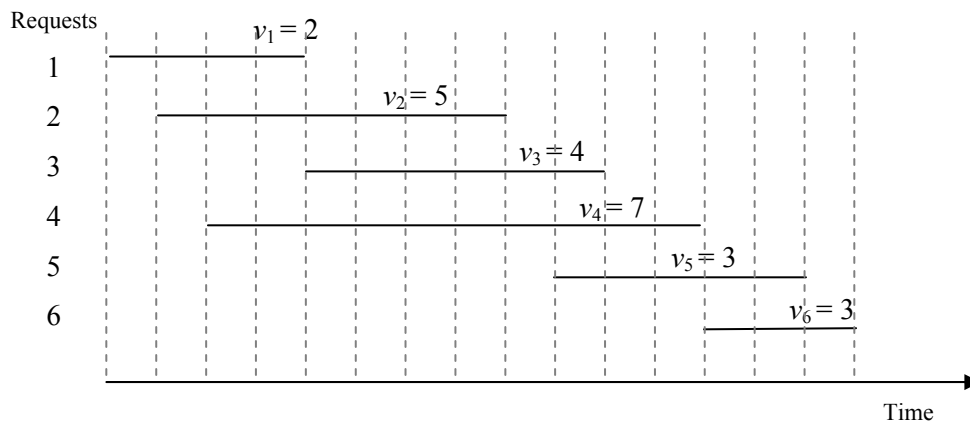


Figure 5.7 An instance of the weighted-interval scheduling problem; requests have been ordered by their finish time (i.e., $f_1 \leq f_2 \leq \dots \leq f_n$).

Version 1 is the weighted-interval scheduling problem. Versions 2 and 3 can be viewed as special cases of version 1. Version 2 is obtained by setting $v_i = 1$ for all requests. For version 3, we set $v_i = f_i - s_i$ — that is, v_i is set to the duration of the i -th request. We proceed to develop a dynamic-programming algorithm for the general version. Note: We will discuss Version 2 and other similar scheduling problems in Section 6.4.

Let $V(i,t)$ be the maximum value obtained by considering requests 1, 2... i and having a schedule that finishes in time t . The solution to the original problem is $V(n, \text{maximum}\{f_1, f_2, \dots, f_n\})$. This suggests that the n -th request that should be considered is the one with the maximum finish time. Again, if the n -th request is not included, then the maximum value is obtained by having t set as large as possible (i.e., maximum of $\{f_1, f_2, \dots, f_{n-1}\}$). Therefore, for the following formulation, we will assume that requests have been ordered (in nondecreasing) order by their finish time. Thus, the solution to our problem is $V(n, f_n)$. Now, $V(i,t)$ involves the decision of whether to include the i -th request in the schedule combined with $V(i-1, t')$. Note that if the i -th job is included, then t' (the time by which all earlier jobs must finish) is s_i (or less, of course). Thus,

$$V(i,t) = \text{maximum} \{ V(i-1, t), v_i + V(i-1, s_i) \text{ if } t \geq f_i \}; \text{ Base case, } V(0, t) = 0.$$

Example 5.4 $V(i,t)$ can be computed in a bottom-up fashion, as illustrated in Figure 5.8.

$i \backslash t$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2
2	0	0	0	0	2	2	2	2	5	5	5	5	5	5	5	5
3	0	0	0	0	2	2	2	2	5	5	6	6	6	6	6	6
4	0	0	0	0	2	2	2	2	5	5	6	6	7	7	7	7
5	0	0	0	0	2	2	2	2	5	5	6	6	7	7	8	8
6	0	0	0	0	2	2	2	2	5	5	6	6	7	7	8	10

Figure 5.8 A solution to the weighted-interval scheduling problem given in Figure 5.7.

5.8 Principle of Optimality

In the dynamic-programming formulation for optimization problems we have seen thus far, we have utilized a recursive formulation that expresses the optimal solution to a problem in terms of the *optimal* solution to the same smaller-size problem (subproblem). This is an important characteristic of dynamic programming and is known as the *principle of optimality*. In situations where the optimal solution is dependent on *nonoptimal* solutions to subproblems, and we formulate the algorithm accordingly, then there would not be any time savings. In such cases, we might as well generate and examine all solutions from the start. Therefore, we use dynamic programming only in situations where the optimal solution to a problem is expressible in terms of the optimal solution to the subproblems. The principle of optimality is normally stated as follows:

Principle of Optimality (POP): In the sequence of decisions that represent the optimal solution, any subsequence is optimal.

Consider the *shortest-path problem* in a weighted graph. The problem calls for finding the path of shortest length from some specified vertex a to another specified vertex b , where the length of a path is defined to be the sum of the weights of its edges. As illustrated by Figure 5.9, assume that the shortest path from vertex a to vertex b is as shown by the solid curved lines. Now consider the portion P of this path from vertex u to vertex v . Is P the shortest path from u to v , or is there the possibility of a shorter path (the dotted line)? We can immediately rule out the latter possibility because if the dotted path exists then we can have it replace P , and thus obtain a shorter path from a to b , contradicting the fact that the solid lines that go from a to b represent the shortest path.

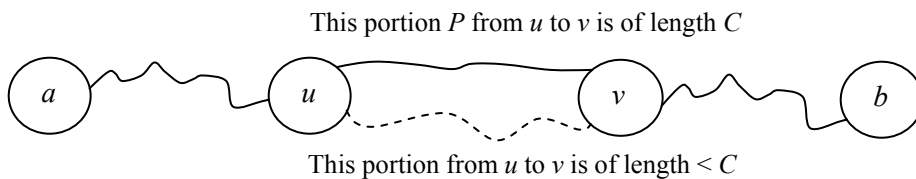


Figure 5.9 A graph illustrating that the shortest-path problem satisfies the principle of optimality.

Not all problems satisfy the principle of optimality. One such problem is the *longest simple path (LSP)* problem, which is to find such a path from one vertex to another in a weighted graph. A path is *simple* if it has no repeated vertices. A simple path avoids cycles whose presence leads to paths with infinite cost because a cycle can be traversed an arbitrarily large number of times. Now, we ask, “Does the longest simple path problem satisfy the principle of optimality?” The answer is *No*. This is illustrated by Figure 5.10. In this case, the longest simple path from a to d is a, b, c, d ; yet, the portion of this path from a to b — that is, the edge (a,b) — is not the longest simple path from a to b because the path a, d, b is a *longer* simple path from a to b than the edge (a,b) . This shows that the LSP problem does not satisfy the principle of optimality. Let us see what happens when we replace a part of an optimal solution with the optimal solution for that part. If we try to replace the edge (a,b) on the path a, b, c, d by the path a, d, b then we get the path a, d, b, c, d . The resulting path is indeed longer than the path a, b, c, d but it is not simple because vertex d appears twice. This shows that the argument we have used to argue POP for the shortest path does not hold for the longest simple path.

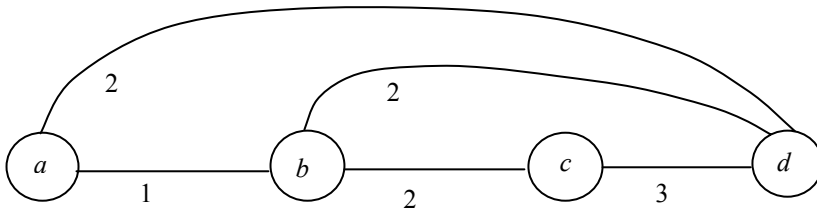


Figure 5.10 A graph illustrating that the longest simple path problem violates the principle of optimality.

In general, the principle of optimality might not hold in situations where the optimal decision at some state is not only dependent on the state but also on how that state is reached. As an example, suppose that car drivers pay tariffs for using long-distance roads that go between cities but the tariff charged depends on other roads that the driver passed through on his journey during the day. To use dynamic programming in such situations, we have to redefine the problem's state, somehow, to include the past history.

The principle of optimality is also referred to as *the optimal substructure property*.

The next two examples are related to the *coin-change* problem, where the objective is to determine the *minimum* number of coins needed to change an amount n using (an infinite supply of) coins of denomination $d_1, d_2 \dots d_m$.

Example 5.5 The optimal substructure property holds for the coin-change problem. Let S be a minimum-size set of coins for an amount n . If we remove one coin of denomination d from S , then S' , the set S with this coin removed, is a minimum-size set of coins for the amount $n-d$. For if there is a set S'' of a smaller size than S' for the amount $n-d$ then $S'' \cup \{d\}$ is a smaller-size set than S for the amount n , which is a contradiction.

The optimal substructure property *does not* say that if S_1 and S_2 are optimal solutions to subproblems, then combining S_1 and S_2 gives an optimal solution to the original problem. This is the *converse* of the optimal substructure property.

Example 5.6 Suppose that the available denominations for the coin-change problem are 10, 6, and 1. One coin of denomination 1 and one coin of denomination 6 give an optimal solution for the amount 7. Five coins of denomination 1 give an optimal solution for the amount 5. Combining these solutions, which results in six coins of denomination 1 and one coin of denomination 6, does not yield an optimal solution for the amount 12. Clearly, the optimal solution for 12 is to use two coins of denomination 6. Thus, for the coin-change problem, if S_1 and S_2 are optimal solutions, combining S_1 and S_2 does not necessarily give an optimal solution to the original problem. Therefore, the converse of the optimal substructure property does not hold for the coin-change problem.

5.9 Shortest Paths

In a shortest-paths problem, we are given a weighted directed graph $G=(V,E)$, with weight function $w: E \rightarrow \mathbb{R}$, mapping edges to real-valued weights. The length (weight, cost) of a path $p=v_0, v_1, \dots, v_k$ is the sum of weights of

its constituent edges, that is:
$$\text{length}(p) = \sum_{i=1}^k w(v_{i-1}, v_i).$$

The shortest-paths problem calls for finding the paths of minimum length. Edge weights can be interpreted as metrics other than distances. They are often used to represent time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that we want to minimize.

Variants of the Shortest-Paths Problem

There are several variants (versions) of the shortest path problem that vary in input and difficulty. We describe the basic versions.

1. *Single-source shortest-paths problem*: Given a graph $G=(V,E)$, we are to find the shortest paths from a given source (start) vertex $s \in V$ to every vertex $v \in V$. The problem does not ask for a single path that starts at s and includes all vertices; rather, it asks for finding $|V|$ shortest paths, one path to each vertex.
2. *Single-destination shortest-paths problem*: The destination vertex t is fixed, and we are to find the shortest paths to t from every vertex v . By reversing the direction of each edge in the input graph, we can reduce this problem to a single-source shortest-paths problem.
3. *Single-pair shortest-path problem*: Given two fixed vertices u and v , find a shortest path from u to v . If we solve the single-source shortest-paths problem with source vertex u , we solve this problem also. Moreover, no algorithm for this problem is known that runs asymptotically faster than the best single-source algorithms in the worst case.
4. *All-pairs shortest-paths problem*: Find a shortest path from u to v for every pair of vertices u and v . This problem can be solved by running a single-source algorithm once from each vertex; but it can be solved faster, and its structure is of interest in its own right.

The Shortest-Paths Problem is Easy

A brute-force approach to solving the single-source shortest-paths problem by enumerating all simple paths that start with the source vertex is $\Omega(n!)$ because there are in the order of $n!$ simple paths in a graph with n vertices. Such an algorithm is exponential in n and would take a long time to be of any value in practice.

From an algorithmic perspective, a problem is classified as either *easy* or *difficult (intractable)*. Any problem that is solvable using a polynomial-time algorithm is classified as an *easy problem*. On the other hand, any problem for which there is no known polynomial-time algorithm is normally classified as a *difficult problem* — the use of the word *difficult* here does not mean the problem is hard to solve; rather, it means that finding a solution takes a long time, even for moderate-size input. It turns out that the problem of finding the shortest paths is easy. On the other hand, the problem of finding the longest paths on general graphs is difficult. However, for acyclic graphs, the longest paths problem is easy — see Section 5.9.3.

Two Algorithms with a Common Idea

As we have noted earlier, the shortest path problem satisfies the principle of optimality, i.e., the shortest path is composed of shortest paths. Therefore, we can choose an appropriate parameter that signifies the path-size and then compute the shortest paths in order by path-size from the smallest-size paths to the largest-size paths.

An obvious parameter for the path-size is the count of edges on the path. Using such a parameter means that we compute all shortest paths consisting of a single edge, then the shortest paths consisting of two edges, etc. This is the *Bellman-Ford algorithm* [Bel58, For56], which we discuss in Section 5.9.2. Alternatively, a clever (and not so obvious) parameter for the path size is the highest-numbered interior vertex on the path. Using such a parameter means that we have to compute all shortest paths where the highest-number interior vertex is 1, then all shortest paths where the highest number is 2, etc. This is *Floyd's algorithm* [Flo62].

The reachability problem is to determine the existence of a path regardless of path length. This problem can be solved by *Warshall's algorithm* [War62]. Warshall's algorithm can be seen as a specialized version of Floyd's algorithm. In the next section, we examine Floyd's and Warshall's algorithms.

5.9.1 Floyd's Shortest-Paths Algorithm

Floyd's shortest-paths algorithm solves the all-pairs shortest-paths problem. The algorithm works correctly assuming that the graph does not contain any negative-length cycle. If there is a negative-length cycle, the shortest path is not well defined, as the path can be made shorter and shorter by having it trace the cycle again and again.

For the all-pairs shortest-paths problem on a graph with n vertices, it is appropriate to represent the output as a *distance matrix* D (n rows \times n columns), where $D[i,j]$ is the length of the shortest path from vertex i to vertex j for $i,j \in [1,n]$. Let $D^k[i,j]$ denote the length of the shortest path from i to j whose interior vertices $\in \{1, 2, \dots, k\}$. Then, clearly, $D[i,j]=D^n[i,j]$. Floyd's algorithm is given by the following recurrence equations:

$$D^0[i,j] = C[i,j] \quad // \quad C[i,j] \text{ is the cost (weight) of the edge } (i,j) \quad 5.9$$

$$D^k[i,j] = \text{minimum} \{ D^{k-1}[i,j], D^{k-1}[i,k]+D^{k-1}[k,j] \} \quad 5.10$$

The justification for these equations is rather straightforward. Equation 5.9 says that the shortest path from vertex i to vertex j having no interior vertices is simply the *direct edge* from vertex i to vertex j . Equation 5.10, on the other hand, says that the shortest path from vertex i to vertex j whose interior vertices $\in \{1, 2, \dots, k\}$ may or may not include vertex k . If the path does not include vertex k then the path interior vertices $\in \{1, 2, \dots, k-1\}$, and by definition, it is of length $D^{k-1}[i,j]$. On the other hand, if the path passes through vertex k then, assuming the graph does not contain negative-length cycles, k must appear once. This case is illustrated in Figure 5.11; the path consists of two paths: a path P_1 from vertex i to vertex k with interior vertices $\in \{1, 2, \dots, k-1\}$, followed by a path P_2 from vertex k to vertex j with interior vertices $\in \{1, 2, \dots, k-1\}$. By the principle of optimality, the path P_1 must be the shortest path from vertex i to vertex k , and by definition, it is of length $D^{k-1}[i,k]$. Similarly, P_2 is of length $D^{k-1}[k,j]$.

Dynamic Programming

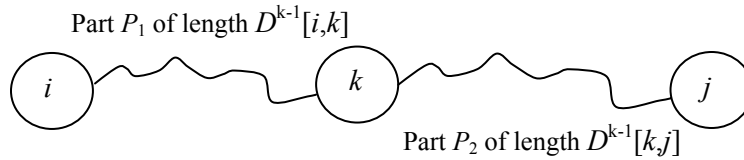


Figure 5.11 The shortest path from vertex i to vertex j whose interior vertices $\in \{1, 2, \dots, k\}$ and, at the same time, passes through vertex k .

Note that D^0 is simply the adjacency cost matrix. Then, we compute in sequence, the matrices D^1, D^2, \dots, D^n . The matrix D^k is computed from D^{k-1} using Equation 5.10, or, visually, in accordance with Figure 5.12. Thus, D^1 is computed from D^0 by examining all $[i,1]$ and $[1,j]$ entries, and executing the test: Is $(D^0[i,1]+D^0[1,j]) < D^0[i,j]$? If yes, the entry $D^1[i,j]$ is set to $D^0[i,1]+D^0[1,j]$; otherwise, $D^1[i,j]$ is simply $D^0[i,j]$.

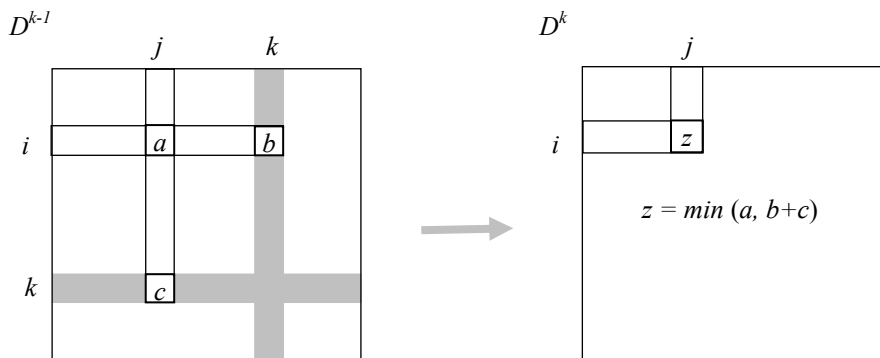


Figure 5.12 Floyd's algorithm: the computation of the (i,j) -entry in D^k .

Let us see an example of how Floyd's algorithm works. Figure 5.13 shows a directed weighted graph, the corresponding adjacency cost matrix C , and the various D matrices computed by Floyd's algorithm.

We should note that in computing the matrix D^k , the entries in the k -th row and the k -th column will remain unchanged from the previous matrix, D^{k-1} — to see this, compute $D^k[k,i]$ and $D^k[j,k]$ using the general recurrence. Because in updating other entries, we only need the entries in the k -th row and the k -th column and that these precise entries are not affected during the computation of D^k , the newly computed entries for D^k can be immediately stored back in D^{k-1} . In other words, it suffices to maintain a single copy of the D matrix, and for a given k , we execute the test: Is $(D[i,k]+D[k,j]) < D[i,j]$? If yes, the entry $D[i,j]$ is set to $D[i,k]+D[k,j]$. This leads to concise program code for Floyd's algorithm, which is given in Listing 5.8.

Recovering the Shortest Paths

To recover the shortest paths, we can utilize the *Pred* (short for *Predecessor*) matrix, where $Pred[i,j]$ is the vertex immediately preceding vertex j along the shortest path from vertex i to vertex j . If $Pred[i,j]$ is known for all pairs of vertices, then the shortest path from any vertex i to any other vertex j is simply (from vertex j backward): $j, x_1=Pred[i,j], x_2=Pred[i, x_1], \dots$ until reaching $x_k = i$.

The *Pred* matrix is computed gradually; every time we have a better estimate for the shortest path from i to j , we update the entry $Pred[i,j]$. In Listing 5.8, the modification of Floyd's algorithm to compute the *Pred* matrix is reflected by the statements shown in bold.

To get the sequence of vertices representing the shortest path from vertex i to vertex j , we invoke the function $GetPath(i,j)$. This function is recursive and is based on the fact that the *shortest path* (as a vertex sequence) *from vertex i to vertex j* is the shortest path from vertex i to $Pred[i,j]$ followed by vertex j . Note that $GetPath$ is passed the *Pred* matrix as one of its arguments. In particular, using the *Pred* matrix for the example given in Figure 5.13, we find that the shortest path from vertex 4 to vertex 1 is given as (in order from last vertex to first vertex): 1, $Pred[4,1]=2$, $Pred[4,2]=3$, $Pred[4,3]=5$, $Pred[4,5]=4$. In other words, the shortest path from 4 to 1 is: 4, 5, 3, 2, 1 with a cost (length) = $3+2+7+3 = 15$.

Dynamic Programming

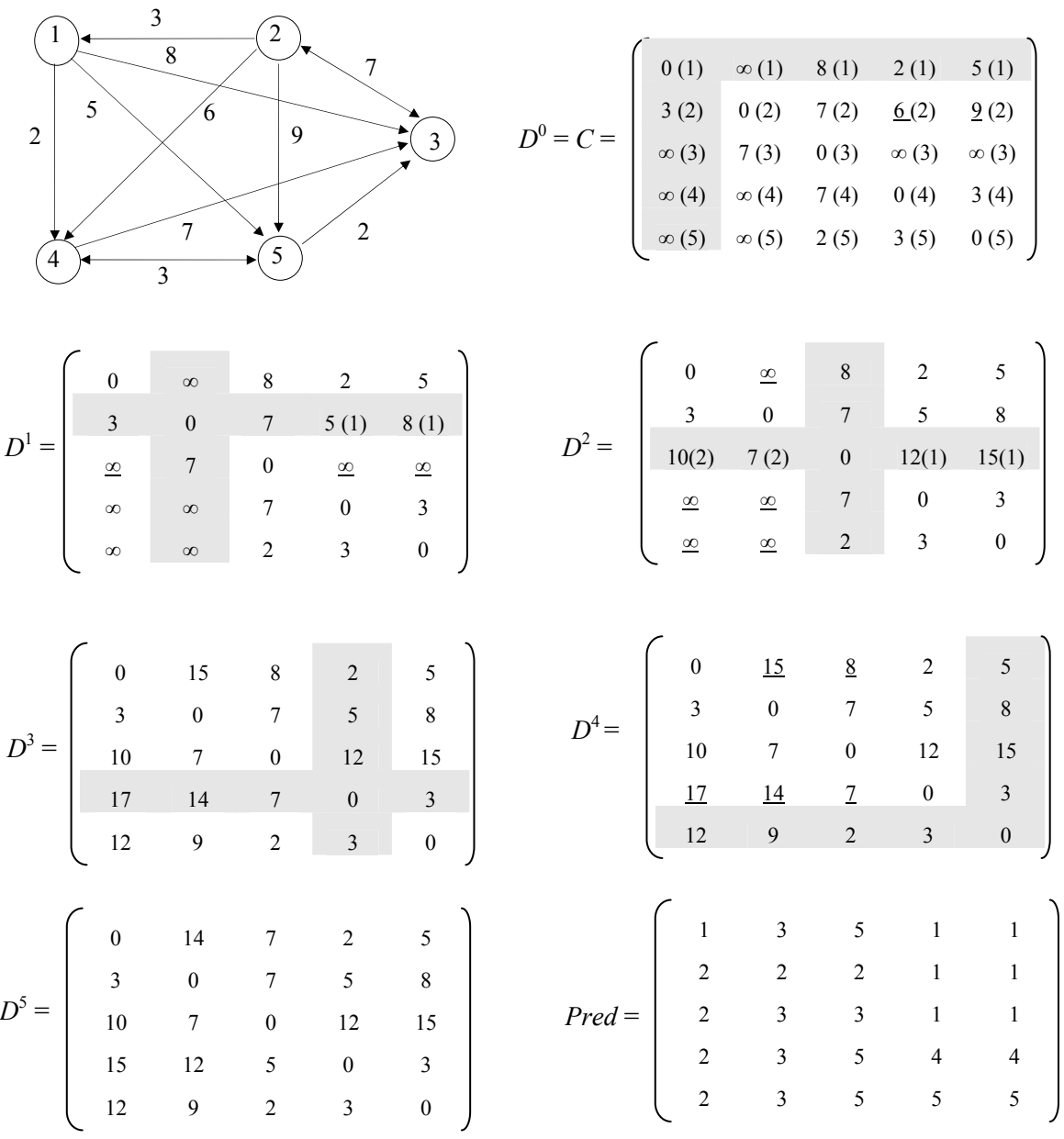


Figure 5.13 A weighted graph and the various D matrices computed by Floyd's all-pairs shortest-paths algorithm. Affected entries are shown as underlined. Entries in parentheses are for the $Pred$ matrix.

Input: Number of vertices n ; adjacency cost matrix $C[1..n,1..n]$ for a directed graph
Output: Distance matrix $D[1..n,1..n]$; Predecessor matrix $Pred[1..n,1..n]$

```
void Floyd(int n, int[,] C, ref int[,] D, ref int[,] Pred)
{ // Allocate space for matrices D and Pred
  D = new int[n+1,n+1];
  Pred = new int[n+1,n+1];
  // Initialization: Set D to  $D^0 = C$ 
  for(int i=1; i <= n; i++)
    for(int j=1; j <= n; j++)
      { D[i,j] = C[i,j];
        Pred[i,j] = i; // for recording the shortest path
      }
  // Compute in order,  $D^1, D^2, \dots D^n$ 
  for(int k=1; k <= n; k++)
    for(int i=1; i <= n; i++)
      for(int j=1; j <= n; j++)
        { int t = D[i,k] + D[k,j];
          if (t < D[i,j])
            { D[i,j] = t;
              Pred[i,j] = Pred[k,j]; // for recording the shortest path
            }
        }
}

string GetPath(int i, int j, int[,] Pred)
{ // returns, as string, a list (space separated) of vertices representing
  // the shortest path from vertex i to vertex j
  if (i == j) return i.ToString();
  else return GetPath(i, Pred[i,j], Pred) + " " + j;
}
```

Listing 5.8 Floyd's algorithm for all-pairs shortest-paths problem.

Dynamic Programming

Warshall's Transitive Closure Algorithm

In certain applications, we are given a digraph G and we are only interested in determining whether there exists a path (regardless of its length) from vertex i to vertex j for all the vertices i, j in G . Floyd's algorithm can be specialized to this problem; the resulting algorithm, which happens to predate Floyd's, is due to Warshall [War62].

Suppose our cost matrix C is just the adjacency matrix for the given digraph. That is, $C[i,j]=1$ if there is an edge from i to j , and 0 otherwise. We wish to compute the Boolean matrix A such that $A[i,j]=1$ if there is a path from i to j , and 0 otherwise. The matrix A is known as the *transitive closure* of the adjacency matrix C . The matrix A is the *reachability matrix* of its underlying graph because $A[i,j]=1$ if and only if j can be reached from i by a path in the graph.

The transitive closure can be computed using an algorithm similar to Floyd's algorithm. Let $A^k[i,j]=1$ if (and only if) there is a path from i to j whose interior vertices $\in \{1, 2, \dots, k\}$. Then, clearly, $A[i,j]=A^n[i,j]$. Warshall's algorithm uses the following recurrence equations.

$$A^0[i,j] = C[i,j] \tag{5.11}$$

$$A^k[i,j] = A^{k-1}[i,j] \text{ or } (A^{k-1}[i,k] \text{ and } A^{k-1}[k,j]) \tag{5.12}$$

The preceding equations readily translate into the program code given in Listing 5.9.

```
Input: Number of vertices n; adjacency matrix C[1..n,1..n] for a directed graph
Output: Transitive closure matrix A[1..n,1..n]
```

```
void Warshall(int n, int[,] C, ref int[,] A)
{ // Allocate space for matrix A
  A = new bool[n+1,n+1];
  // Initialization: Set A = C
  for(int i=1; i <= n; i++)
    for(int j=1; j <= n; j++)
      A[i,j] = C[i,j];

  // Compute in order, A1, A2, ... An
  for(int k=1; k <= n; k++)
    for(int i=1; i <= n; i++)
      for(int j=1; j <= n; j++)
        if (A[i,j] == false) A[i,j] = A[i,k] && A[k,j];
}
```

Listing 5.9 Warshall's algorithm for transitive closure.

Exercise 5.5 The derivation of Floyd's algorithm indicates that the algorithm works correctly only if the graph does not contain negative-length cycles. How would Floyd's algorithm be modified to discover such cycles?

5.9.2 Bellman-Ford Shortest-Paths Algorithm

For the Bellman-Ford algorithm, the path size is the number of edges on the path; thus, let $D^k[i,j]$ denote the length of the shortest path from i to j having at most k edges. Then we claim that $D[i,j]=D^{n-1}[i,j]$. The latter claim is justified as follows. In a graph with n vertices, the shortest path from some vertex to any other vertex has at most $n-1$ edges; otherwise, the path will have a cycle and a *shorter path* can be obtained by removing the cycle, assuming that any cycle, if present, has positive weight — the weight (length) of a cycle is the sum of weight of its constituent edges. Because of this assumption, Bellman-Ford algorithm works if some edges have negative weights, as long as there are no cycles with negative weight.

The Bellman-Ford algorithm is given by the following recurrence equations:

$$D^1[i,j] = C[i,j] \quad // \quad C[i,j] \text{ is the cost (weight) of the edge } (i,j) \quad 5.13$$

$$D^k[i,j] = \text{minimum} \{ D^{k-1}[i,j], \text{minimum over all edges } (i,m) D^{k-1}[i,m]+C[m,j] \} \quad 5.14$$

Based on the preceding recurrence equations, we are to compute $n-1$ matrices, D^1, D^2, \dots, D^{n-1} . For each of the last $n-2$ matrices, n^2 entries have to be computed and the computation of any entry may require examining $\Theta(n)$ edges. Thus, the algorithm is $\Theta(n^4)$.

The computation of the matrix D^i can be carried out using matrix multiplication as: $D^i = D^{i-1} \times C$, except that we modify the matrix multiplication of two matrices A ($k \times n$) and B ($n \times m$) as $R = A \times B$, as follows:

$$r_{ij} = \sum_{k=1 \text{ to } n} a_{ik} b_{kj} \Rightarrow r_{ij} = \min_{k=1 \text{ to } n} \{ a_{ik} + b_{kj} \}.$$

If we generate all the matrices D^2, D^3, \dots, D^{n-1} using the preceding approach, the resulting algorithm is clearly $\Theta(n^4)$. However, a shortest path of length at most k edges can also be viewed as the concatenation of two shortest paths, each of length at most $k/2$ edges. This implies that D^k can be computed as $D^{k/2} \times D^{k/2}$. Thus, by generating certain powers of the adjacency matrix by repeated squaring, the running time is reduced to $\Theta(n^3 \log n)$.

Bellman-Ford Algorithm for the Single-Source Shortest-Paths Problem

The previous formulation of Bellman-Ford algorithm can be easily specialized to solve the *single-source shortest-paths* problem. That is, given a weighted directed graph and a source vertex sv , we are to find the shortest paths from sv to every other vertex. In this case the D matrix can be reduced to a one-dimensional vector $D[1..n]$, where $D^k[i]$ is the length of the shortest path from sv to i using at most k edges.

$$D^1[i] = C[sv,i] \quad // \quad C[sv,i] \text{ is the cost (weight) of the edge from } sv \text{ to } i \quad 5.15$$

$$D^k[i] = \text{minimum} \{ D^{k-1}[i], \text{minimum over all edges } (i,m) D^{k-1}[m]+C[m,i] \} \quad 5.16$$

In actual implementation of the Bellman-Ford algorithm, it suffices to maintain a single copy of D . Then when it comes to computing Equation 5.16, we have two options: (a) an outer i -loop where i ranges over all the vertices and an inner loop that examines every outgoing edge from i , or (b) a *single* loop that goes over all the edges. The latter option is more efficient, especially for sparse graphs (where $|E| \approx |V|$); the algorithm's program-code is given in Listing 5.10, and a worked-out example is given by Figure 5.14. The algorithm's running-time complexity is $O(|V| \times |E|)$ — for dense graphs this is $O(|V|^3)$, and it should use the edge-list (or adjacency-lists) representation.