$h(n)$	$f(n)$
$O(n^r)$, $r < 0$	$O(1)$
$\Theta((\log n)^i)$, $i \geq 0$	$\Theta(((\log n)^{i+1})/(i+1))$
$\Omega(n^r)$, $r > 0$	$\Theta(h(n))$

TABLE 1.4 $f(n)$ values for various $h(n)$ values

where a and b are known constants. The merge sort recurrence, Equation 1.6, is in this form. Although the recurrence for binary search, Equation 1.5, isn't exactly in this form, the $n \leq 1$ may be changed to $n = 1$ by eliminating the case $n = 0$. To solve Equation 1.7, we assume that $t(1)$ is known and that n is a power of b (i.e., $n = b^k$). Using the substitution method, we can show that

$$t(n) = n^{\log_b a}[t(1) + f(n)] \tag{1.8}$$

where $f(n) = \sum_{j=1}^{k} h(b^j)$ and $h(n) = g(n)/n^{\log_b a}$.

Table 1.4 tabulates the asymptotic value of $f(n)$ for various values of $h(n)$. This table allows us to easily obtain the asymptotic value of $t(n)$ for many of the recurrences we encounter when analyzing divide-and-conquer algorithms.

Let us solve the binary search and merge sort recurrences using this table. Comparing Equation 1.5 with $n \leq 1$ replaced by $n = 1$ with Equation 1.7, we see that $a = 1$, $b = 2$, and $g(n) = c$. Therefore, $\log_b(a) = 0$, and $h(n) = g(n)/n^{\log_b a} = c = c(\log n)^0 = \Theta((\log n)^0)$. From Table 1.4, we obtain $f(n) = \Theta(\log n)$. Therefore, $t(n) = n^{\log_b a}(c + \Theta(\log n)) = \Theta(\log n)$.

For the merge sort recurrence, Equation 1.6, we obtain $a = 2$, $b = 2$, and $g(n) = cn$. So $\log_b a = 1$ and $h(n) = g(n)/n = c = \Theta((\log n)^0)$. Hence $f(n) = \Theta(\log n)$ and $t(n) = n(t(1) + \Theta(\log n)) = \Theta(n \log n)$.

1.7 Amortized Complexity

1.7.1 What is Amortized Complexity?

The complexity of an algorithm or of an operation such as an insert, search, or delete, as defined in Section 1.1, is the *actual complexity* of the algorithm or operation. The actual complexity of an operation is determined by the step count for that operation, and the actual complexity of a sequence of operations is determined by the step count for that sequence. The actual complexity of a sequence of operations may be determined by adding together the step counts for the individual operations in the sequence. Typically, determining the step count for each operation in the sequence is quite difficult, and instead, we obtain an upper bound on the step count for the sequence by adding together the worst-case step count for each operation.

When determining the complexity of a sequence of operations, we can, at times, obtain tighter bounds using *amortized complexity* rather than worst-case complexity. Unlike the actual and worst-case complexities of an operation which are closely related to the step count for that operation, the amortized complexity of an operation is an accounting artifact that often bears no direct relationship to the actual complexity of that operation. The amortized complexity of an operation could be anything. *The only requirement is that the* *sum of the amortized complexities of all operations in the sequence be greater than or equal to the sum of the actual complexities.* That is

$$\sum_{1 \le i \le n} \mathit{amortized}(i) \ge \sum_{1 \le i \le n} \mathit{actual}(i) \tag{1.9}$$

where $\mathit{amortized}(i)$ and $\mathit{actual}(i)$, respectively, denote the amortized and actual complexities of the ith operation in a sequence of n operations. Because of this requirement on the sum of the amortized complexities of the operations in any sequence of operations, we may use the sum of the amortized complexities as an upper bound on the complexity of any sequence of operations.

You may view the amortized cost of an operation as being the amount you charge the operation rather than the amount the operation costs. You can charge an operation any amount you wish so long as the amount charged to all operations in the sequence is at least equal to the actual cost of the operation sequence.

Relative to the actual and amortized costs of each operation in a sequence of n operations, we define a *potential function* $P(i)$ as below

$$P(i) = \mathit{amortized}(i) - \mathit{actual}(i) + P(i-1) \tag{1.10}$$

That is, the ith operation causes the potential function to change by the difference between the amortized and actual costs of that operation. If we sum Equation 1.10 for $1 \le i \le n$, we get

$$\sum_{1 \le i \le n} P(i) = \sum_{1 \le i \le n} (\mathit{amortized}(i) - \mathit{actual}(i) + P(i-1))$$

or

$$\sum_{1 \le i \le n} (P(i) - P(i-1)) = \sum_{1 \le i \le n} (\mathit{amortized}(i) - \mathit{actual}(i))$$

or

$$P(n) - P(0) = \sum_{1 \le i \le n} (\mathit{amortized}(i) - \mathit{actual}(i))$$

From Equation 1.9, it follows that

$$P(n) - P(0) \ge 0 \tag{1.11}$$

When $P(0) = 0$, the potential $P(i)$ is the amount by which the first i operations have been overcharged (i.e., they have been charged more than their actual cost).

Generally, when we analyze the complexity of a sequence of n operations, n can be any nonnegative integer. Therefore, Equation 1.11 must hold for all nonnegative integers.

The preceding discussion leads us to the following three methods to arrive at amortized costs for operations:

1. **Aggregate Method**
   In the aggregate method, we determine an upper bound for the sum of the actual costs of the n operations. The amortized cost of each operation is set equal to this upper bound divided by n. You may verify that this assignment of amortized costs satisfies Equation 1.9 and is, therefore, valid.

2. **Accounting Method**
   In this method, we assign amortized costs to the operations (probably by guessing what assignment will work), compute the $P(i)$s using Equation 1.10, and show that $P(n) - P(0) \geq 0$.
3. **Potential Method**
   Here, we start with a potential function (probably obtained using good guess work) that satisfies Equation 1.11 and compute the amortized complexities using Equation 1.10.

1.7.2 Maintenance Contract

Problem Definition

In January, you buy a new car from a dealer who offers you the following maintenance contract: \$50 each month other than March, June, September and December (this covers an oil change and general inspection), \$100 every March, June, and September (this covers an oil change, a minor tune-up, and a general inspection), and \$200 every December (this covers an oil change, a major tune-up, and a general inspection). We are to obtain an upper bound on the cost of this maintenance contract as a function of the number of months.

Worst-Case Method

We can bound the contract cost for the first n months by taking the product of n and the maximum cost incurred in any month (i.e., \$200). This would be analogous to the traditional way to estimate the complexity–take the product of the number of operations and the worst-case complexity of an operation. Using this approach, we get \$200n as an upper bound on the contract cost. The upper bound is correct because the actual cost for n months does not exceed \$200n.

Aggregate Method

To use the aggregate method for amortized complexity, we first determine an upper bound on the sum of the costs for the first n months. As tight a bound as is possible is desired. The sum of the actual monthly costs of the contract for the first n months is

$$\begin{aligned}
200 * \lfloor n/12 \rfloor &+ 100 * (\lfloor n/3 \rfloor - \lfloor n/12 \rfloor) + 50 * (n - \lfloor n/3 \rfloor) \\
&= 100 * \lfloor n/12 \rfloor + 50 * \lfloor n/3 \rfloor + 50 * n \\
&\leq 100 * n/12 + 50 * n/3 + 50 * n \\
&= 50n(1/6 + 1/3 + 1) \\
&= 50n(3/2) \\
&= 75n
\end{aligned}$$

The amortized cost for each month is set to \$75. Table 1.5 shows the actual costs, the amortized costs, and the potential function value (assuming $P(0) = 0$) for the first 16 months of the contract.

Notice that some months are charged more than their actual costs and others are charged less than their actual cost. The cumulative difference between what the operations are charged and their actual costs is given by the potential function. The potential function satisfies Equation 1.11 for all values of n. When we use the amortized cost of \$75 per month, we get \$75n as an upper bound on the contract cost for n months. This bound is tighter than the bound of \$200n obtained using the worst-case monthly cost.

month	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
actual cost	50	50	100	50	50	100	50	50	100	50	50	200	50	50	100	50
amortized cost	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75
P()	25	50	25	50	75	50	75	100	75	100	125	0	25	50	25	50

TABLE 1.5 Maintenance contract

Accounting Method

When we use the accounting method, we must first assign an amortized cost for each month and then show that this assignment satisfies Equation 1.11. We have the option to assign a different amortized cost to each month. In our maintenance contract example, we know the actual cost by month and could use this actual cost as the amortized cost. It is, however, easier to work with an equal cost assignment for each month. Later, we shall see examples of operation sequences that consist of two or more types of operations (for example, when dealing with lists of elements, the operation sequence may be made up of search, insert, and remove operations). When dealing with such sequences we often assign a different amortized cost to operations of different types (however, operations of the same type have the same amortized cost).

To get the best upper bound on the sum of the actual costs, we must set the amortized monthly cost to be the smallest number for which Equation 1.11 is satisfied for all n. From the above table, we see that using any cost less than \$75 will result in $P(n) - P(0) < 0$ for some values of n. Therefore, the smallest assignable amortized cost consistent with Equation 1.11 is \$75.

Generally, when the accounting method is used, we have not computed the aggregate cost. Therefore, we would not know that \$75 is the least assignable amortized cost. So we start by assigning an amortized cost (obtained by making an educated guess) to each of the different operation types and then proceed to show that this assignment of amortized costs satisfies Equation 1.11. Once we have shown this, we can obtain an upper bound on the cost of any operation sequence by computing

$$\sum_{1 \le i \le k} f(i) * \mathit{amortized}(i)$$

where k is the number of different operation types and $f(i)$ is the frequency of operation type i (i.e., the number of times operations of this type occur in the operation sequence).

For our maintenance contract example, we might try an amortized cost of \$70. When we use this amortized cost, we discover that Equation 1.11 is not satisfied for $n = 12$ (for example) and so \$70 is an invalid amortized cost assignment. We might next try \$80. By constructing a table such as the one above, we will observe that Equation 1.11 is satisfied for all months in the first 12 month cycle, and then conclude that the equation is satisfied for all n. Now, we can use \$$80n$ as an upper bound on the contract cost for n months.

Potential Method

We first define a potential function for the analysis. The only guideline you have in defining this function is that the potential function represents the cumulative difference between the amortized and actual costs. So, if you have an amortized cost in mind, you may be able to use this knowledge to develop a potential function that satisfies Equation 1.11, and then use the potential function and the actual operation costs (or an upper bound on these actual costs) to verify the amortized costs.

If we are extremely experienced, we might start with the potential function

$$t(n) = \begin{cases} 0 & n \bmod 12 = 0 \\ 25 & n \bmod 12 = 1 \text{ or } 3 \\ 50 & n \bmod 12 = 2,\ 4,\ \text{or } 6 \\ 75 & n \bmod 12 = 5,\ 7,\ \text{or } 9 \\ 100 & n \bmod 12 = 8 \text{ or } 10 \\ 125 & n \bmod 12 = 11 \end{cases}$$

Without the aid of the table (Table 1.5) constructed for the aggregate method, it would take quite some ingenuity to come up with this potential function. Having formulated a potential function and verified that this potential function satisfies Equation 1.11 for all n, we proceed to use Equation 1.10 to determine the amortized costs.

From Equation 1.10, we obtain $amortized(i) = actual(i) + P(i) - P(i-1)$. Therefore,

$$\begin{aligned} amortized(1) &= actual(1) + P(1) - P(0) = 50 + 25 - 0 = 75 \\ amortized(2) &= actual(2) + P(2) - P(1) = 50 + 50 - 25 = 75 \\ amortized(3) &= actual(3) + P(3) - P(2) = 100 + 25 - 50 = 75 \end{aligned}$$

and so on. Therefore, the amortized cost for each month is \$75. So, the actual cost for n months is at most \$$75n$.

1.7.3 The McWidget Company

Problem Definition

The famous McWidget company manufactures widgets. At its headquarters, the company has a large display that shows how many widgets have been manufactured so far. Each time a widget is manufactured, a maintenance person updates this display. The cost for this update is \$$c + dm$, where c is a fixed trip charge, d is a charge per display digit that is to be changed, and m is the number of digits that are to be changed. For example, when the display is changed from 1399 to 1400, the cost to the company is \$$c + 3d$ because 3 digits must be changed. The McWidget company wishes to amortize the cost of maintaining the display over the widgets that are manufactured, charging the same amount to each widget. More precisely, we are looking for an amount \$$e = amortized(i)$ that should levied against each widget so that the sum of these charges equals or exceeds the actual cost of maintaining/updating the display (\$$e * n \geq$ actual total cost incurred for first n widgets for all $n \geq 1$). To keep the overall selling price of a widget low, we wish to find as small an e as possible. Clearly, $e > c + d$ because each time a widget is made, at least one digit (the least significant one) has to be changed.

Worst-Case Method

This method does not work well in this application because there is no finite worst-case cost for a single display update. As more and more widgets are manufactured, the number of digits that need to be changed increases. For example, when the 1000th widget is made, 4 digits are to be changed incurring a cost of $c + 4d$, and when the 1,000,000th widget is made, 7 digits are to be changed incurring a cost of $c + 7d$. If we use the worst-case method, the amortized cost to each widget becomes infinity.

widget	1	2	3	4	5	6	7	8	9	10	11	12	13	14
actual cost	1	1	1	1	1	1	1	1	1	2	1	1	1	1
amortized cost—	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12
P()	0.12	0.24	0.36	0.48	0.60	0.72	0.84	0.96	1.08	0.20	0.32	0.44	0.56	0.68

widget	15	16	17	18	19	20	21	22	23	24	25	26	27	28
actual cost	1	1	1	1	1	2	1	1	1	1	1	1	1	1
amortized cost—	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12
P()	0.80	0.92	1.04	1.16	1.28	0.40	0.52	0.64	0.76	0.88	1.00	1.12	1.24	1.36

TABLE 1.6 Data for widgets

Aggregate Method

Let n be the number of widgets made so far. As noted earlier, the least significant digit of the display has been changed n times. The digit in the ten's place changes once for every ten widgets made, that in the hundred's place changes once for every hundred widgets made, that in the thousand's place changes once for every thousand widgets made, and so on. Therefore, the aggregate number of digits that have changed is bounded by

$$n(1 + 1/10 + 1/100 + 1/1000 + ...) = (1.11111...)n$$

So, the amortized cost of updating the display is \$$c + d(1.11111...)n/n < c + 1.12d$. If the McWidget company adds \$$c+1.12d$ to the selling price of each widget, it will collect enough money to pay for the cost of maintaining the display. Each widget is charged the cost of changing 1.12 digits regardless of the number of digits that are actually changed. Table 1.6 shows the actual cost, as measured by the number of digits that change, of maintaining the display, the amortized cost (i.e., 1.12 digits per widget), and the potential function. The potential function gives the difference between the sum of the amortized costs and the sum of the actual costs. Notice how the potential function builds up so that when it comes time to pay for changing two digits, the previous potential function value plus the current amortized cost exceeds 2. From our derivation of the amortized cost, it follows that the potential function is always nonnegative.

Accounting Method

We begin by assigning an amortized cost to the individual operations, and then we show that these assigned costs satisfy Equation 1.11. Having already done an amortized analysis using the aggregate method, we see that Equation 1.11 is satisfied when we assign an amortized cost of \$$c + 1.12d$ to each display change. Typically, however, the use of the accounting method is not preceded by an application of the aggregate method and we start by guessing an amortized cost and then showing that this guess satisfies Equation 1.11.

Suppose we assign a guessed amortized cost of \$$c + 2d$ for each display change.

$$\begin{aligned}
P(n) - P(0) &= \sum_{1 \le i \le n} (amortized(i) - actual(i)) \\
&= (c + 2d)n - \sum_{1 \le i \le n} actual(i) \\
&= (c + 2d)n - (c + (1 + 1/10 + 1/100 + ...)d)n \\
&\ge (c + 2d)n - (c + 1.12d)n \\
&\ge 0
\end{aligned}$$

This analysis also shows us that we can reduce the amortized cost of a widget to \$$c+1.12d$.

An alternative proof method that is useful in some analyses involves distributing the excess charge $P(i) - P(0)$ over various accounting entities, and using these stored excess charges (called *credits*) to establish $P(i+1) - P(0) \geq 0$. For our McWidget example, we use the display digits as the accounting entities. Initially, each digit is 0 and each digit has a credit of 0 dollars. Suppose we have guessed an amortized cost of $\$c + (1.111...)d$. When the first widget is manufactured, $\$c + d$ of the amortized cost is used to pay for the update of the display and the remaining $\$(0.111...)d$ of the amortized cost is retained as a credit by the least significant digit of the display. Similarly, when the second through ninth widgets are manufactured, $\$c + d$ of the amortized cost is used to pay for the update of the display and the remaining $\$(0.111...)d$ of the amortized cost is retained as a credit by the least significant digit of the display. Following the manufacture of the ninth widget, the least significant digit of the display has a credit of $\$(0.999...)d$ and the remaining digits have no credit. When the tenth widget is manufactured, $\$c + d$ of the amortized cost are used to pay for the trip charge and the cost of changing the least significant digit. The least significant digit now has a credit of $\$(1.111...)d$. Of this credit, $\$d$ are used to pay for the change of the next least significant digit (i.e., the digit in the ten's place), and the remaining $\$(0.111...)d$ are transferred to the ten's digit as a credit. Continuing in this way, we see that when the display shows 99, the credit on the ten's digit is $\$(0.999...)d$ and that on the one's digit (i.e., the least significant digit) is also $\$(0.999...)d$. When the 100th widget is manufactured, $\$c + d$ of the amortized cost are used to pay for the trip charge and the cost of changing the least significant digit, and the credit on the least significant digit becomes $\$(1.111...)d$. Of this credit, $\$d$ are used to pay for the change of the ten's digit from 9 to 0, the remaining $\$(0.111...)d$ credit on the one's digit is transferred to the ten's digit. The credit on the ten's digit now becomes $\$(1.111...)d$. Of this credit, $\$d$ are used to pay for the change of the hundred's digit from 0 to 1, the remaining $\$(0.111...)d$ credit on the ten's digit is transferred to the hundred's digit.

The above accounting scheme ensures that the credit on each digit of the display always equals $\$(0.111...)dv$, where v is the value of the digit (e.g., when the display is 206 the credit on the one's digit is $\$(0.666...)d$, the credit on the ten's digit is \$0, and that on the hundred's digit is $\$(0.222...)d$.

From the preceding discussion, it follows that $P(n) - P(0)$ equals the sum of the digit credits and this sum is always nonnegative. Therefore, Equation 1.11 holds for all n.

Potential Method

We first postulate a potential function that satisfies Equation 1.11, and then use this function to obtain the amortized costs. From the alternative proof used above for the accounting method, we can see that we should use the potential function $P(n) = (0.111...)d\sum_i v_i$, where v_i is the value of the ith digit of the display. For example, when the display shows 206 (at this time $n = 206$), the potential function value is $(0.888...)d$. This potential function satisfies Equation 1.11.

Let q be the number of 9s at the right end of j (i.e., when $j = 12903999$, $q = 3$). When the display changes from j to $j + 1$, the potential change is $(0.111...)d(1 - 9q)$ and the actual cost of updating the display is $\$c + (q+1)d$. From Equation 1.10, it follows that the amortized cost for the display change is

$$\text{actual cost} + \text{potential change} = c + (q+1)d + (0.111...)d(1-9q) = c + (1.111...)d$$

1.7.4 Subset Generation

Problem Definition

The subsets of a set of n elements are defined by the 2^n vectors $x[1:n]$, where each $x[i]$ is either 0 or 1. $x[i]=1$ iff the ith element of the set is a member of the subset. The subsets of a set of three elements are given by the eight vectors 000, 001, 010, 011, 100, 101, 110, and 111, for example. Starting with an array $x[1:n]$ has been initialized to zeroes (this represents the empty subset), each invocation of algorithm `nextSubset` (Figure 1.10) returns the next subset. When all subsets have been generated, this algorithm returns `null`.

```
public int [] nextSubset()
{// return next subset; return null if no next subset
   // generate next subset by adding 1 to the binary number x[1:n]
   int i = n;
   while (i > 0 && x[i] == 1)
      {x[i] = 0; i--;}

   if (i == 0) return null;
   else {x[i] = 1; return x;}
}
```

FIGURE 1.10: Subset enumerator.

We wish to determine how much time it takes to generate the first m, $1 \leq m \leq 2^n$ subsets. This is the time for the first m invocations of `nextSubset`.

Worst-Case Method

The complexity of `nextSubset` is $\Theta(c)$, where c is the number of $x[i]$s that change. Since all n of the $x[i]$s could change in a single invocation of `nextSubset`, the worst-case complexity of `nextSubset` is $\Theta(n)$. Using the worst-case method, the time required to generate the first m subsets is $O(mn)$.

Aggregate Method

The complexity of `nextSubset` equals the number of $x[i]$s that change. When `nextSubset` is invoked m times, $x[n]$ changes m times; $x[n-1]$ changes $\lfloor m/2 \rfloor$ times; $x[n-2]$ changes $\lfloor m/4 \rfloor$ times; $x[n-3]$ changes $\lfloor m/8 \rfloor$ times; and so on. Therefore, the sum of the actual costs of the first m invocations is $\sum_{0 \leq i \leq \lfloor \log_2 m \rfloor}(m/2^i) < 2m$. So, the complexity of generating the first m subsets is actually $O(m)$, a tighter bound than obtained using the worst-case method.

The amortized complexity of `nextSubset` is (sum of actual costs)$/m < 2m/m = O(1)$.

Accounting Method

We first guess the amortized complexity of `nextSubset`, and then show that this amortized complexity satisfies Equation 1.11. Suppose we guess that the amortized complexity is 2. To verify this guess, we must show that $P(m) - P(0) \geq 0$ for all m.

We shall use the alternative proof method used in the McWidget example. In this method, we distribute the excess charge $P(i) - P(0)$ over various accounting entities, and use these stored excess charges to establish $P(i+1) - P(0) \geq 0$. We use the $x[j]$s as the accounting entities. Initially, each $x[j]$ is 0 and has a credit of 0. When the first subset is generated, 1 unit of the amortized cost is used to pay for the single $x[j]$ that changes and the remaining 1 unit of the amortized cost is retained as a credit by $x[n]$, which is the $x[j]$ that has changed to 1. When the second subset is generated, the credit on $x[n]$ is used to pay for changing $x[n]$ to 0 in the while loop, 1 unit of the amortized cost is used to pay for changing $x[n-1]$ to 1, and the remaining 1 unit of the amortized cost is retained as a credit by $x[n-1]$, which is the $x[j]$ that has changed to 1. When the third subset is generated, 1 unit of the amortized cost is used to pay for changing $x[n]$ to 1, and the remaining 1 unit of the amortized cost is retained as a credit by $x[n]$, which is the $x[j]$ that has changed to 1. When the fourth subset is generated, the credit on $x[n]$ is used to pay for changing $x[n]$ to 0 in the while loop, the credit on $x[n-1]$ is used to pay for changing $x[n-1]$ to 0 in the while loop, 1 unit of the amortized cost is used to pay for changing $x[n-2]$ to 1, and the remaining 1 unit of the amortized cost is retained as a credit by $x[n-2]$, which is the $x[j]$ that has changed to 1. Continuing in this way, we see that each $x[j]$ that is 1 has a credit of 1 unit on it. This credit is used to pay the actual cost of changing this $x[j]$ from 1 to 0 in the while loop. One unit of the amortized cost of `nextSubset` is used to pay for the actual cost of changing an $x[j]$ to 1 in the else clause, and the remaining one unit of the amortized cost is retained as a credit by this $x[j]$.

The above accounting scheme ensures that the credit on each $x[j]$ that is 1 is exactly 1, and the credit on each $x[j]$ that is 0 is 0.

From the preceding discussion, it follows that $P(m) - P(0)$ equals the number of $x[j]$s that are 1. Since this number is always nonnegative, Equation 1.11 holds for all m.

Having established that the amortized complexity of `nextSubset` is $2 = O(1)$, we conclude that the complexity of generating the first m subsets equals $m *$ amortized complexity $= O(m)$.

Potential Method

We first postulate a potential function that satisfies Equation 1.11, and then use this function to obtain the amortized costs. Let $P(j)$ be the potential just after the jth subset is generated. From the proof used above for the accounting method, we can see that we should define $P(j)$ to be equal to the number of $x[i]$s in the jth subset that are equal to 1.

By definition, the 0th subset has all $x[i]$ equal to 0. Since $P(0) = 0$ and $P(j) \geq 0$ for all j, this potential function P satisfies Equation 1.11. Consider any subset $x[1:n]$. Let q be the number of 1s at the right end of $x[]$ (i.e., $x[n]$, $x[n-1]$, $\cdots$, $x[n-q+1]$, are all 1s). Assume that there is a next subset. When the next subset is generated, the potential change is $1-q$ because q 1s are replaced by 0 in the while loop and a 0 is replaced by a 1 in the else clause. The actual cost of generating the next subset is $q+1$. From Equation 1.10, it follows that, when there is a next subset, the amortized cost for `nextSubset` is

$$\text{actual cost} \ + \ \text{potential change} = q + 1 + 1 - q = 2$$

When there is no next subset, the potential change is $-q$ and the actual cost of `nextSubset` is q. From Equation 1.10, it follows that, when there is no next subset, the amortized cost for `nextSubset` is

$$\text{actual cost} \ + \ \text{potential change} = q - q = 0$$

Therefore, we can use 2 as the amortized complexity of `nextSubset`. Consequently, the actual cost of generating the first m subsets is $O(m)$.

1.8 Practical Complexities

We have seen that the time complexity of a program is generally some function of the problem size. This function is very useful in determining how the time requirements vary as the problem size changes. For example, the run time of an algorithm whose complexity is $\Theta(n^2)$ is expected to increase by a factor of 4 when the problem size doubles and by a factor of 9 when the problem size triples.

The complexity function also may be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity $\Theta(n)$ and that algorithm Q has complexity $\Theta(n^2)$. We can assert that algorithm P is faster than algorithm Q for "sufficiently large" n. To see the validity of this assertion, observe that the actual computing time of P is bounded from above by cn for some constant c and for all n, $n \geq n_1$, while that of Q is bounded from below by dn^2 for some constant d and all n, $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, algorithm P is faster than algorithm Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

One should always be cautiously aware of the presence of the phrase *sufficiently large* in the assertion of the preceding discussion. When deciding which of the two algorithms to use, we must know whether the n we are dealing with is, in fact, sufficiently large. If algorithm P actually runs in $10^6 n$ milliseconds while algorithm Q runs in n^2 milliseconds and if we always have $n \leq 10^6$, then algorithm Q is the one to use.

To get a feel for how the various functions grow with n, you should study Figures 1.11 and 1.12 very closely. These figures show that 2^n grows very rapidly with n. In fact, if a algorithm needs 2^n steps for execution, then when $n = 40$, the number of steps needed is approximately $1.1 * 10^{12}$. On a computer performing 1,000,000,000 steps per second, this algorithm would require about 18.3 minutes. If $n = 50$, the same algorithm would run for about 13 days on this computer. When $n = 60$, about 310.56 years will be required to execute the algorithm, and when $n = 100$, about $4 * 10^{13}$ years will be needed. We can conclude that the utility of algorithms with exponential complexity is limited to small n (typically $n \leq 40$).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

FIGURE 1.11: Value of various functions.

Algorithms that have a complexity that is a high-degree polynomial are also of limited utility. For example, if an algorithm needs n^{10} steps, then our 1,000,000,000 steps per second computer needs 10 seconds when $n = 10$; 3171 years when $n = 100$; and $3.17 * 10^{13}$ years when $n = 1000$. If the algorithm's complexity had been n^3 steps instead, then the computer would need 1 second when $n = 1000$, 110.67 minutes when $n = 10{,}000$, and 11.57 days when $n = 100{,}000$.

Figure 1.13 gives the time that a 1,000,000,000 instructions per second computer needs to execute an algorithm of complexity $f(n)$ instructions. One should note that currently only the fastest computers can execute about 1,000,000,000 instructions per second. From a

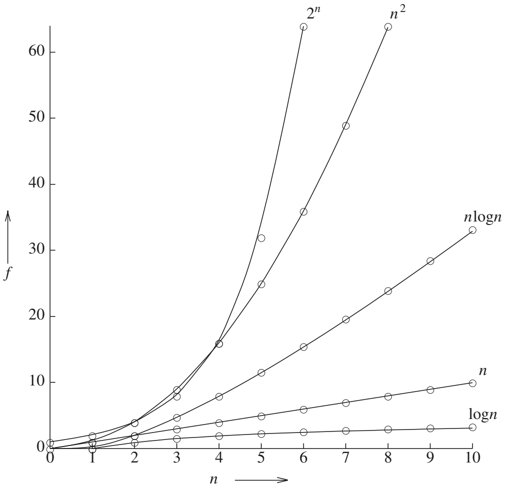

FIGURE 1.12: Plot of various functions.

practical standpoint, it is evident that for reasonably large n (say $n > 100$) only algorithms of small complexity (such as n, $n \log n$, n^2, and n^3) are feasible. Further, this is the case even if we could build a computer capable of executing 10^{12} instructions per second. In this case the computing times of Figure 1.13 would decrease by a factor of 1000. Now when $n = 100$, it would take 3.17 years to execute n^{10} instructions and $4 * 10^{10}$ years to execute 2^n instructions.

	$f(n)$						
n	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	$4 * 10^{13}$ y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	$3.17 * 10^{13}$ y	$32 * 10^{283}$ y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	$3.17 * 10^{23}$ y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	$3.17 * 10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17 * 10^7$ y	$3.17 * 10^{43}$ y	

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

FIGURE 1.13: Run times on a 1,000,000,000 instructions per second computer.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

References

[1] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, NY, 1992.

[2] J. Hennessey and D. Patterson, *Computer Organization and Design*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998, Chapter 7.

[3] E. Horowitz, S. Sahni, and S. Rajasekaran, *Fundamentals of Computer Algorithms*, W. H. Freeman and Co., New York, NY, l998.

[4] G. Rawlins, *Compared to What: An Introduction to the Analysis of Algorithms*, W. H. Freeman and Co., New York, NY, 1992.

[5] S. Sahni, *Data Structures, Algorithms, and Applications in Java*, McGraw-Hill, NY, 2000.