Chapter 1

Basic Concepts in Algorithmic Analysis

1.1 Introduction

The most general intuitive idea of an *algorithm* * is a procedure that consists of a finite set of instructions which, given an *input* from some set of *possible inputs*, enables us to obtain an *output* if such an output exists or else obtain nothing at all if there is no output for that particular input through a systematic execution of the instructions. The set of possible inputs consists of all inputs to which the algorithm gives an output. If there is an output for a particular input, then we say that the algorithm can be *applied* to this input and *process* it to give the corresponding output. We require that an algorithm halts on every input, which implies that each instruction requires a finite amount of time, and each input has a finite length. We also require that the output of a legal input to be unique, that is, the algorithm is deterministic in the sense that the same set of instructions are executed when the algorithm is initiated on a particular input more than once. In Chapter 14, we will relax this condition when we study randomized algorithms.

The design and analysis of algorithms are of fundamental importance in the field of computer science. As Donald E. Knuth stated "Computer science is the study of algorithms". This should not be surprising, as every area in computer science depends heavily on the design of efficient algo-

*According to the *American Heritage Dictionary*, the word "algorithm" is derived from the name of Muhammad ibn Mūsā al-Khwārizmī (780?-850?), a Muslim mathematician whose works introduced Arabic numerals and algebraic concepts to Western mathematics. The word "algebra" stems from the title of his book Kitab al jabr wa'l-muqābala.

rithms. As simple examples, compilers and operating systems are nothing but direct implementations of special purpose algorithms.

The objective of this chapter is twofold. First, it introduces some simple algorithms, particularly related to searching and sorting. Second, it covers the basic concepts used in the design and analysis of algorithms. We will cover in depth the notion of "running time" of an algorithm, as it is of fundamental importance to the design of efficient algorithms. After all, time is the most precious measure of an algorithm's efficiency. We will also discuss the other important resource measure, namely the space required by an algorithm.

Although simple, the algorithms presented will serve as the basis for many of the examples in illustrating several algorithmic concepts. It is instructive to start with simple and useful algorithms that are used as building blocks in more complex algorithms.

1.2 Historical Background

The question of whether a problem can be solved using an *effective procedure*, which is equivalent to the contemporary notion of the algorithm, received a lot of attention in the first part of this century, and especially in the 1930's. The focus in that period was on classifying problems as being solvable using an effective procedure or not. For this purpose, the need arose for a model of computation by the help of which a problem can be classified as solvable if it is possible to construct an algorithm to solve that problem using that model. Some of these models are the recursive functions of Gödel, λ-calculus of Church, Post machines of Post and the Turing machines of Turing. The RAM model of computation was introduced as a theoretical counterpart of existing computing machines. By *Church Thesis*, all these models have the same power, in the sense that if a problem is solvable on one of them, then it is solvable on all others.

Surprisingly, it turns out that "almost all" problems are unsolvable. This can be justified easily as follows. Since each algorithm can be thought of as a function whose domain is the set of nonnegative integers and whose range is the set of real numbers, the set of functions to be computed is uncountable. Since any algorithm, or more specifically a program, can be encoded as a binary string, which corresponds to a unique positive integer, the number of functions that can be computed is countable. So, infor-

mally, the number of solvable problems is equinumerous with the set of integers (which is countable), while the number of unsolvable problems is equinumerous with the set of real numbers (which is uncountable). As a simple example, no algorithm can be constructed to decide whether seven consecutive 1's occur in the decimal expansion of π. This follows from the definition of an algorithm, which stipulates that the amount of time an algorithm is allowed to run must be finite. Another example is the problem of deciding whether a given equation involving a polynomial with n variables $x_1, x_2, \ldots, x_n$ has integer solutions. This problem is unsolvable, no matter how powerful the computing machine used is. That field which is concerned with decidability and solvability of problems is referred to as *computability theory* or *theory of computation*, although some computer scientists advocate the inclusion of the current field of algorithms as part of this discipline.

After the advent of digital computers, the need arose for investigating those solvable problems. In the beginning, one was content with a simple program that solves a particular problem without worrying about the amount of resources, in particular time, that this program requires. Then the need for efficient programs that use as few resources as possible evolved as a result of the limited resources available and the need to develop complex algorithms. This led to the evolution of a new area in computing, namely *computational complexity*. In this area, a problem that is classified as solvable is studied in terms of its efficiency, that is, the time and space needed to solve that problem. Later on, other resources were introduced, e.g. communication cost and the number of processors if the problem is analyzed using a parallel model of computation.

Unfortunately, some of the conclusions of this study turned out to be negative: There are many *natural* problems that are *practically* unsolvable due to the need for huge amount of resources, and in particular time. On the other hand, efficient algorithms have been devised to solve many problems. Not only that; it was also proven that those algorithms are optimal in the sense that if any new algorithm to solve the same problem is discovered, then the gain in terms of efficiency is virtually minimal. For example, the problem of sorting a set of elements has been studied extensively; and as a result, several efficient algorithms to solve this problem have been devised, and it was proven that these algorithms are optimal in the sense that no substantially better algorithm can ever be devised in the future.

1.3 Binary Search

Henceforth, in the context of searching and sorting problems, we will assume that the elements are drawn from a linearly ordered set, for example the set of integers. This will also be the case for similar problems, like finding the median, the kth smallest element, and so forth. Let $A[1..n]$ be a sequence of n elements. Consider the problem of determining whether a given element x is in A. This problem can be rephrased as follows. Find an index $j, 1 \leq j \leq n$, such that $x = A[j]$ if x is in A, and $j = 0$ otherwise. A straightforward approach is to scan the entries in A and compare each entry with x. If after j comparisons, $1 \leq j \leq n$, the search is *successful*, i.e., $x = A[j]$, j is returned; otherwise a value of 0 is returned indicating an *unsuccessful* search. This method is referred to as *sequential search*. It is also called *linear search*, as the maximum number of element comparisons grows linearly with the size of the sequence. The algorithm is shown as Algorithm LINEARSEARCH.

Algorithm 1.1 LINEARSEARCH
Input: An array $A[1..n]$ of n elements and an element x.
Output: j if $x = A[j], 1 \leq j \leq n$, and 0 otherwise.

```
1. j ← 1
2. while (j < n) and (x ≠ A[j])
3.       j ← j + 1
4. end while
5. if x = A[j] then return j else return 0
```

Intuitively, scanning all entries of A is inevitable if no more information about the ordering of the elements in A is given. If we are also given that the elements in A are sorted, say in nondecreasing order, then there is a much more efficient algorithm. The following example illustrates this efficient search method.

Example 1.1 Consider searching the array

$$A[1..14] = \boxed{1 \mid 4 \mid 5 \mid 7 \mid 8 \mid 9 \mid 10 \mid 12 \mid 15 \mid 22 \mid 23 \mid 27 \mid 32 \mid 35}.$$

In this instance, we want to search for element $x = 22$. First, we compare x with the middle element $A[\lfloor (1 + 14)/2 \rfloor] = A[7] = 10$. Since $22 > A[7]$, and since it is

known that $A[i] \leq A[i+1], 1 \leq i < 14$, x cannot be in $A[1..7]$, and therefore this portion of the array can be discarded. So, we are left with the subarray

$$A[8..14] = \boxed{12|15|22|23|27|32|35}.$$

Next, we compare x with the middle of the remaining elements $A[\lfloor (8+14)/2 \rfloor] = A[11] = 23$. Since $22 < A[11]$, and since $A[i] \leq A[i+1], 11 \leq i < 14$, x cannot be in $A[11..14]$, and therefore this portion of the array can also be discarded. Thus, the remaining portion of the array to be searched is now reduced to

$$A[8..10] = \boxed{12|15|22}.$$

Repeating this procedure, we discard $A[8..9]$, which leaves only one entry in the array to be searched, that is $A[10] = 22$. Finally, we find that $x = A[10]$, and the search is successfully completed.

In general, let $A[low..high]$ be a nonempty array of elements sorted in nondecreasing order. Let $A[mid]$ be the middle element, and suppose that $x > A[mid]$. We observe that if x is in A, then it must be one of the elements $A[mid+1], A[mid+2], \ldots, A[high]$. It follows that we only need to search for x in $A[mid+1..high]$. In other words, the entries in $A[low..mid]$ are discarded in subsequent comparisons since, by assumption, A is sorted in nondecreasing order, which implies that x cannot be in this half of the array. Similarly, if $x < A[mid]$, then we only need to search for x in $A[low..mid-1]$. This results in an efficient strategy which, because of its repetitive halving, is referred to as *binary search*. Algorithm BINARYSEARCH gives a more formal description of this method.

Algorithm 1.2 BINARYSEARCH
Input: An array $A[1..n]$ of n elements sorted in nondecreasing order and an element x.
Output: j if $x = A[j], 1 \leq j \leq n$, and 0 otherwise.

```
1. low ← 1;    high ← n;    j ← 0
2. while (low ≤ high) and (j = 0)
3.       mid ← ⌊(low + high)/2⌋
4.       if x = A[mid] then j ← mid
5.       else if x < A[mid] then high ← mid − 1
6.       else low ← mid + 1
7. end while
8. return j
```

1.3.1 *Analysis of the binary search algorithm*

Henceforth, we will assume that each three-way comparison (if-then-else) counts as one comparison. Obviously, the minimum number of comparisons is 1, and it is achievable when the element being searched for, x, is in the middle position of the array. To find the maximum number of comparisons, let us first consider applying binary search on the array $\boxed{2}\boxed{3}\boxed{5}\boxed{8}$. If we search for 2 or 5, we need two comparisons, whereas searching for 8 costs three comparisons. Now, in the case of unsuccessful search, it is easy to see that searching for elements such as 1, 4, 7 or 9 takes 2, 2, 3 and 3 comparisons, respectively. It is not hard to see that, in general, the algorithm always performs the maximum number of comparisons whenever x is greater than or equal to the maximum element in the array. In this example, searching for any element greater than or equal to 8 costs three comparisons. Thus, to find the maximum number of comparisons, we may assume without loss of generality that x is greater than or equal to $A[n]$.

Example 1.2 Suppose that we want to search for $x = 35$ or $x = 100$ in

$$A[1..14] = \boxed{1}\boxed{4}\boxed{5}\boxed{7}\boxed{8}\boxed{9}\boxed{10}\boxed{12}\boxed{15}\boxed{22}\boxed{23}\boxed{27}\boxed{32}\boxed{35}.$$

In each iteration of the algorithm, the bottom half of the array is discarded until there is only one element:

$$\boxed{12}\boxed{15}\boxed{22}\boxed{23}\boxed{27}\boxed{32}\boxed{35} \rightarrow \boxed{27}\boxed{32}\boxed{35} \rightarrow \boxed{35}.$$

Therefore, to compute the *maximum* number of element comparisons performed by Algorithm BINARYSEARCH, we may assume that x is greater than or equal to all elements in the array to be searched. To compute the number of remaining elements in $A[1..n]$ in the second iteration, there are two cases to consider according to whether n is even or odd. If n is even, then the number of entries in $A[mid + 1..n]$ is $n/2$; otherwise it is $(n-1)/2$. Thus, in both cases, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2 \rfloor$.

Similarly, the number of remaining elements to be searched in the third iteration is $\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/4 \rfloor$ (see Eq. 2.3 on page 71).

In general, in the jth pass through the while loop, the number of remaining elements is $\lfloor n/2^{j-1} \rfloor$. The iteration is continued until either x is found or the size of the subsequence being searched reaches 1, whichever

occurs first. As a result, the maximum number of iterations needed to search for x is that value of j satisfying the condition

$$\lfloor n/2^{j-1} \rfloor = 1.$$

By the definition of the floor function, this happens exactly when

$$1 \leq n/2^{j-1} < 2,$$

or

$$2^{j-1} \leq n < 2^j,$$

or

$$j - 1 \leq \log n < j.^{\dagger}$$

Since j is integer, we conclude that

$$j = \lfloor \log n \rfloor + 1.$$

Alternatively, the performance of the binary search algorithm can be described in terms of a *decision tree*, which is a binary tree that exhibits the behavior of the algorithm. Figure 1.1 shows the decision tree corresponding to the array given in Examples 1.1. The darkened nodes are those compared against x in Examples 1.1.

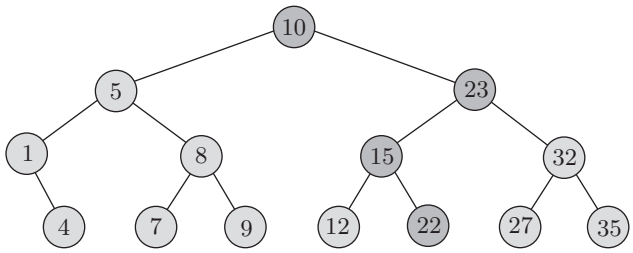

Fig. 1.1 A decision tree that shows the behavior of binary search.

Note that the decision tree is a function of the *number* of the elements in the array only. Figure 1.2 shows two decision trees corresponding to two arrays of sizes 10 and 14, respectively. As implied by the two figures, the

[†]Unless otherwise stated, all logarithms in this book are to the base 2. The natural logarithm of x will be denoted by $\ln x$.

maximum number of comparisons in both trees is 4. In general, the maximum number of comparisons is one plus the height of the corresponding decision tree (see Sec. 3.5 for the definition of height). Since the height of such a tree is $\lfloor \log n \rfloor$, we conclude that the maximum number of comparisons is $\lfloor \log n \rfloor + 1$. We have in effect given two proofs of the following theorem:

Theorem 1.1 The number of comparisons performed by Algorithm BINARYSEARCH on a sorted array of size n is at most $\lfloor \log n \rfloor + 1$.

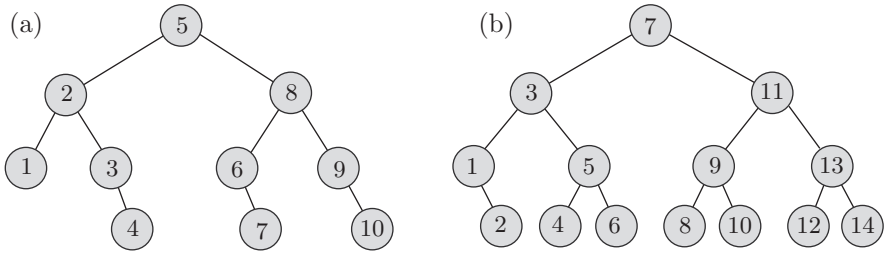

Fig. 1.2 Two decision trees corresponding to two arrays of sizes 10 and 14.

1.4 Merging two Sorted Lists

Suppose we have an array $A[1..m]$ and three indices p, q and r, with $1 \leq p \leq q < r \leq m$, such that both the subarrays $A[p..q]$ and $A[q+1..r]$ are individually sorted in nondecreasing order. We want to rearrange the elements in A so that the elements in the subarray $A[p..r]$ are sorted in nondecreasing order. This process is referred to as *merging* $A[p..q]$ with $A[q+1..r]$. An algorithm to merge these two subarrays works as follows. We maintain two pointers s and t that initially point to $A[p]$ and $A[q+1]$, respectively. We prepare an empty array $B[p..r]$ which will be used as a temporary storage. Each time, we compare the elements $A[s]$ and $A[t]$ and append the smaller of the two to the auxiliary array B; if they are equal we will choose to append $A[s]$. Next, we update the pointers: If $A[s] \leq A[t]$, then we increment s, otherwise we increment t. This process ends when $s = q+1$ or $t = r+1$. In the first case, we append the remaining elements $A[t..r]$ to B, and in the second case, we append $A[s..q]$ to B. Finally, the

array $B[p..r]$ is copied back to $A[p..r]$. This procedure is given in Algorithm MERGE.

Algorithm 1.3 MERGE
Input: An array $A[1..m]$ of elements and three indices p, q and r, with $1 \le p \le q < r \le m$, such that both the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted individually in nondecreasing order.
Output: $A[p..r]$ contains the result of merging the two subarrays $A[p..q]$ and $A[q+1..r]$.

```
1.  comment: B[p..r] is an auxiliary array.
2.  s ← p;   t ← q + 1;   k ← p
3.  while s ≤ q and t ≤ r
4.      if A[s] ≤ A[t] then
5.          B[k] ← A[s]
6.          s ← s + 1
7.      else
8.          B[k] ← A[t]
9.          t ← t + 1
10.     end if
11.     k ← k + 1
12. end while
13. if s = q + 1 then B[k..r] ← A[t..r]
14. else B[k..r] ← A[s..q]
15. end if
16. A[p..r] ← B[p..r]
```

Let n denote the size of the array $A[p..r]$ in the input to Algorithm MERGE, i.e., $n = r - p + 1$. We want to find the number of comparisons that are needed to rearrange the entries of $A[p..r]$. It should be emphasized that from now on when we talk about the number of comparisons performed by an algorithm, we mean *element comparisons*, i.e., the comparisons involving objects in the input data. Thus, all other comparisons, e.g. those needed for the implementation of the **while** loop, will be excluded.

Let the two subarrays be of sizes n_1 and n_2, where $n_1 + n_2 = n$. The least number of comparisons happens if each entry in the smaller subarray is less than all entries in the larger subarray. For example, to merge the two subarrays

| 2 | 3 | 6 | and | 7 | 11 | 13 | 45 | 57 |,

the algorithm performs only three comparisons. On the other hand, the

number of comparisons may be as high as $n-1$. For example, to merge the two subarrrays

| 2 | 3 | 66 | and | 7 | 11 | 13 | 45 | 57 |,

seven comparisons are needed. It follows that the number of comparisons done by Algorithm MERGE is at least n_1 and at most $n-1$.

Observation 1.1 The number of element comparisons performed by Algorithm MERGE to merge two nonempty arrays of sizes n_1 and n_2, respectively, where $n_1 \leq n_2$, into one sorted array of size $n = n_1 + n_2$ is between n_1 and $n-1$. In particular, if the two array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons needed is between $\lfloor n/2 \rfloor$ and $n-1$.

How about the number of *element assignments* (again here we mean assignments involving input data)? At first glance, one may start by looking at the **while** loop, the **if** statements, etc. in order to find out how the algorithm works and then compute the number of element assignments. However, it is easy to see that each entry of array B is assigned exactly once. Similarly, each entry of array A is assigned exactly once, when copying B back into A. As a result, we have the following observation:

Observation 1.2 The number of element assignments performed by Algorithm MERGE to merge two arrays into one sorted array of size n is exactly $2n$.

1.5 Selection Sort

Let $A[1..n]$ be an array of n elements. A simple and straightforward algorithm to sort the entries in A works as follows. First, we find the minimum element and store it in $A[1]$. Next, we find the minimum of the remaining $n-1$ elements and store it in $A[2]$. We continue this way until the second largest element is stored in $A[n-1]$. This method is described in Algorithm SELECTIONSORT.

It is easy to see that the number of element comparisons performed by the algorithm is exactly

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

Algorithm 1.4 SELECTIONSORT
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for i ← 1 to n − 1
2.     k ← i
3.     for j ← i + 1 to n      {Find the ith smallest element.}
4.         if A[j] < A[k] then k ← j
5.     end for
6.     if k ≠ i then interchange A[i] and A[k]
7. end for
```

It is also easy to see that the number of element interchanges is between 0 and $n-1$. Since each interchange requires three element assignments, the number of element assignments is between 0 and $3(n-1)$.

Observation 1.3 The number of element comparisons performed by Algorithm SELECTIONSORT is $n(n-1)/2$. The number of element assignments is between 0 and $3(n-1)$.

1.6 Insertion Sort

As stated in Observation 1.3 above, the number of comparisons performed by Algorithm SELECTIONSORT is *exactly* $n(n-1)/2$ regardless of how the elements of the input array are ordered. Another sorting method in which the number of comparisons depends on the order of the input elements is the so-called INSERTIONSORT. This algorithm, which is shown below, works as follows. We begin with the subarray of size 1, $A[1]$, which is already sorted. Next, $A[2]$ is inserted before or after $A[1]$ depending on whether it is smaller than $A[1]$ or not. Continuing this way, in the ith iteration, $A[i]$ is inserted in its proper position in the sorted subarray $A[1..i-1]$. This is done by scanning the elements from index $i-1$ down to 1, each time comparing $A[i]$ with the element at the current position. In each iteration of the scan, an element is shifted one position up to a higher index. This process of scanning, performing the comparison and shifting continues until an element less than or equal to $A[i]$ is found, or when all the sorted sequence so far is exhausted. At this point, $A[i]$ is inserted in its proper position, and the process of inserting element $A[i]$ in its proper

place is complete.

Algorithm 1.5 INSERTIONSORT
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for i ← 2 to n
2.     x ← A[i]
3.     j ← i − 1
4.     while (j > 0) and (A[j] > x)
5.         A[j + 1] ← A[j]
6.         j ← j − 1
7.     end while
8.     A[j + 1] ← x
9. end for
```

Unlike Algorithm SELECTIONSORT, the number of element comparisons done by Algorithm INSERTIONSORT depends on the order of the input elements. It is easy to see that the number of element comparisons is minimum when the array is already sorted in nondecreasing order. In this case, the number of element comparisons is exactly $n-1$, as each element $A[i]$, $2 \leq i \leq n$, is compared with $A[i-1]$ only. On the other hand, the maximum number of element comparisons occurs if the array is already sorted in decreasing order and all elements are distinct. In this case, the number of element comparisons is

$$\sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2},$$

as each element $A[i]$, $2 \leq i \leq n$, is compared with each entry in the subarray $A[1..i-1]$. This number coincides with that of Algorithm SELECTIONSORT.

Observation 1.4 The number of element comparisons performed by Algorithm INSERTIONSORT is between $n-1$ and $n(n-1)/2$. The number of element assignments is equal to the number of element comparisons plus $n-1$.

Notice the correlation of element comparisons and assignments in Algorithm INSERTIONSORT. This is in contrast to the independence of the number of element comparisons in Algorithm SELECTIONSORT related to data arrangement.

1.7 Bottom-up Merge Sorting

The two sorting methods discussed thus far are both inefficient in the sense that the number of operations required to sort n elements is proportional to n^2. In this section, we describe an efficient sorting algorithm that performs much fewer element comparisons. Suppose we have the following array of eight numbers that we wish to sort:

9	4	5	2	1	7	4	6

.

Consider the following method for sorting these numbers (see Fig. 1.3).

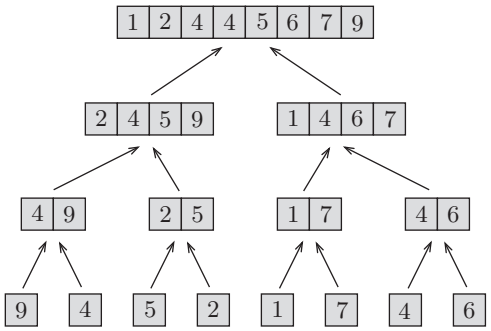

Fig. 1.3 Example of bottom-up merge sorting.

First, we divide the input elements into four pairs and merge each pair into one 2-element sorted sequence. Next, we merge each two consecutive 2-element sequences into one sorted sequence of size four. Finally, we merge the two resulting sequences into the final sorted sequence as shown in the figure.

In general, let A be an array of n elements that is to be sorted. We first merge $\lfloor n/2 \rfloor$ consecutive pairs of elements to yield $\lfloor n/2 \rfloor$ sorted sequences of size 2. If there is one remaining element, then it is passed to the next iteration. Next, we merge $\lfloor n/4 \rfloor$ pairs of consecutive 2-element sequences to yield $\lfloor n/4 \rfloor$ sorted sequences of size 4. If there are one or two remaining elements, then they are passed to the next iteration. If there are three elements left, then two (sorted) elements are merged with one element to form a 3-element sorted sequence. Continuing this way, in the jth iteration, we merge $\lfloor n/2^j \rfloor$ pairs of sorted sequences of size 2^{j-1} to yield $\lfloor n/2^j \rfloor$ sorted sequences of size 2^j. If there are k remaining elements, where $1 \leq k \leq 2^{j-1}$,

then they are passed to the next iteration. If there are k remaining elements, where $2^{j-1} < k < 2^j$, then these are merged to form a sorted sequence of size k.

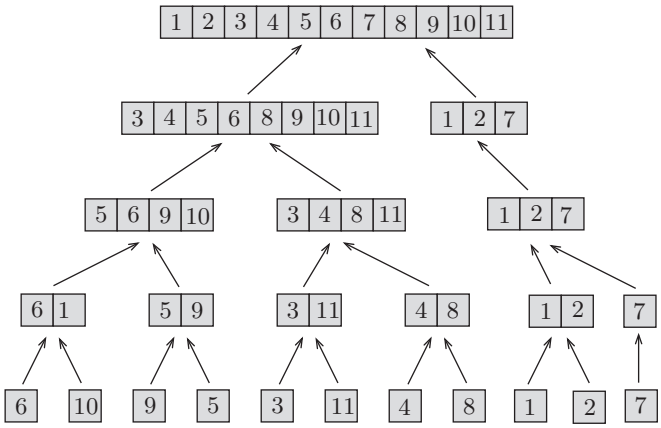

Fig. 1.4 Example of bottom-up merge sorting when n is not a power of 2.

Algorithm BOTTOMUPSORT implements this idea. The algorithm maintains the variable s which is the size of sequences to be merged. Initially, s is set to 1, and is doubled in each iteration of the outer **while** loop. $i+1$, $i+s$ and $i+t$ define the boundaries of the two sequences to be merged. Step 8 is needed in the case when n is not a multiple of t. In this case, if the number of remaining elements, which is $n-i$, is greater than s, then one more merge is applied on a sequence of size s and the remaining elements.

Example 1.3 Figure 1.4 shows an example of the working of the algorithm when n is not a power of 2. The behavior of the algorithm can be described as follows.

(1) In the first iteration, $s = 1$ and $t = 2$. Five pairs of 1-element sequences are merged to produce five 2-element sorted sequences. After the end of the inner **while** loop, $i + s = 10 + 1 \not< n = 11$, and hence no more merging takes place.

(2) In the second iteration, $s = 2$ and $t = 4$. Two pairs of 2-element sequences are merged to produce two 4-element sorted sequences. After the end of the inner **while** loop, $i + s = 8 + 2 < n = 11$, and hence one sequence of size $s = 2$ is merged with the one remaining element to produce a 3-element sorted sequence.

(3) In the third iteration, $s = 4$ and $t = 8$. One pair of 4-element sequences are merged to produce one 8-element sorted sequence. After the end of the inner

Algorithm 1.6 BOTTOMUPSORT
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. t ← 1
2. while t < n
3.     s ← t;    t ← 2s;    i ← 0
4.     while i + t ≤ n
5.         MERGE(A, i + 1, i + s, i + t)
6.         i ← i + t
7.     end while
8.     if i + s < n then MERGE(A, i + 1, i + s, n)
9. end while
```

while loop, $i + s = 8 + 4 \not< n = 11$ and hence no more merging takes place.

(4) In the fourth iteration, $s = 8$ and $t = 16$. Since $i + t = 0 + 16 \not\leq n = 11$, the inner **while** loop is not executed. Since $i + s = 0 + 8 < n = 11$, the condition of the **if** statement is satisfied, and hence one merge of 8-element and 3-element sorted sequences takes place to produce a sorted sequence of size 11.

(5) Since now $t = 16 > n$, the condition of the outer **while** loop is not satisfied, and consequently the algorithm terminates.

1.7.1 *Analysis of bottom-up merge sorting*

Now, we compute the number of element comparisons performed by the algorithm for the special case when n is a power of 2. In this case, the outer **while** loop is executed $k = \log n$ times, once for each level in the sorting tree except the topmost level (see Fig. 1.3). Observe that since n is a power of 2, $i = n$ after the execution of the inner **while** loop, and hence Algorithm MERGE will never be invoked in Step 8. In the first iteration, there are $n/2$ comparisons. In the second iteration, $n/2$ sorted sequences of two elements each are merged in pairs. The number of comparisons needed to merge each pair is either 2 or 3. In the third iteration, $n/4$ sorted sequences of four elements each are merged in pairs. The number of comparisons needed to merge each pair is between 4 and 7. In general, in the jth iteration of the while loop, there are $n/2^j$ merge operations on two subarrays of size 2^{j-1} and it follows, by Observation 1.1, that the number of comparisons needed in the jth iteration is between $(n/2^j)2^{j-1}$ and $(n/2^j)(2^j - 1)$. Thus, if we

let $k = \log n$, then the number of element comparisons is at least

$$\sum_{j=1}^{k} \left(\frac{n}{2^j}\right) 2^{j-1} = \sum_{j=1}^{k} \frac{n}{2} = \frac{kn}{2} = \frac{n \log n}{2},$$

and is at most

$$\begin{aligned}
\sum_{j=1}^{k} \frac{n}{2^j} \left(2^j - 1\right) &= \sum_{j=1}^{k} \left(n - \frac{n}{2^j}\right) \\
&= kn - n \sum_{j=1}^{k} \frac{1}{2^j} \\
&= kn - n \left(1 - \frac{1}{2^k}\right) \qquad \text{(Eq. 2.11, page 78)} \\
&= kn - n \left(1 - \frac{1}{n}\right) \\
&= n \log n - n + 1.
\end{aligned}$$

As to the number of element assignments, there are, by Observation 1.2 applied to each merge operation, $2n$ element assignments in each iteration of the outer **while** loop for a total of $2n \log n$. As a result, we have the following observation:

Observation 1.5 The total number of element comparisons performed by Algorithm BOTTOMUPSORT to sort an array of n elements, where n is a power of 2, is between $(n \log n)/2$ and $n \log n - n + 1$. The total number of element assignments done by the algorithm is exactly $2n \log n$.

1.8 Time Complexity

In this section we study an essential component of algorithmic analysis, namely determining the running time of an algorithm. This theme belongs to an important area in the theory of computation, namely computational complexity, which evolved when the need for efficient algorithms arose in the 1960's and flourished in the 1970's and 1980's. The main objects of study in the field of computational complexity include the time and space needed by an algorithm in order to deliver its output when presented with

legal input. We start this section with an example whose sole purpose is to reveal the importance of analyzing the running time of an algorithm.

Example 1.4 We have shown before that the maximum number of element comparisons performed by Algorithm BOTTOMUPSORT when n is a power of 2 is $n \log n - n + 1$, and the number of element comparisons performed by Algorithm SELECTIONSORT is $n(n-1)/2$. The elements may be integers, real numbers, strings of characters, etc. For concreteness, let us assume that each element comparison takes 10^{-6} seconds on some computing machine. Suppose we want to sort a small number of elements, say 128. Then the time taken for comparing elements using Algorithm BOTTOMUPSORT is at most $10^{-6}(128 \times 7 - 128 + 1) = 0.0008$ seconds. Using Algorithm SELECTIONSORT, the time becomes $10^{-6}(128 \times 127)/2 = 0.008$ seconds. In other words, Algorithm BOTTOMUPSORT uses one tenth of the time taken for comparison using Algorithm SELECTIONSORT. This, of course, is not noticeable, especially to a novice programmer whose main concern is to come up with a program that does the job. However, if we consider a larger number, say $n = 2^{20} = 1,048,576$ which is typical of many real world problems, we find the following: The time taken for comparing elements using Algorithm BOTTOMUPSORT is at most $10^{-6}(2^{20} \times 20 - 2^{20} + 1) = 20$ seconds, whereas, using Algorithm SELECTIONSORT, the time becomes $10^{-6}(2^{20} \times (2^{20} - 1))/2 = 6.4$ days!

The calculations in the above example reveal the fact that time is undoubtedly an extremely precious resource to be investigated in the analysis of algorithms.

1.8.1 *Order of growth*

Obviously, it is meaningless to say that an algorithm A, when presented with input x, runs in time y seconds. This is because the actual time is not only a function of the algorithm used: it is a function of numerous factors, e.g. how and on what machine the algorithm is implemented and in what language or even what compiler or programmer's skills, to mention a few. Therefore, we should be content with only an approximation of the exact time. But, first of all, when assessing an algorithm's efficiency, do we have to deal with exact or even approximate times? It turns out that we really do not need even approximate times. This is supported by many factors, some of which are the following. First, when analyzing the running time of an algorithm, we usually compare its behavior with another algo-

rithm that solves the same problem, or even a different problem. Thus, our estimates of times are *relative* as opposed to *absolute*. Second, it is desirable for an algorithm to be not only machine independent, but also capable of being expressed in any language, including human languages. Moreover, it should be technology independent, that is, we want our measure of the running time of an algorithm to survive technological advances. Third, our main concern is not in small input sizes; we are mostly concerned with the behavior of the algorithm under investigation on large input instances.

In fact, counting the number of operations in some "reasonable" implementation of an algorithm is more than what is needed. As a consequence of the third factor above, we can go a giant step further: A precise count of the number of all operations is very cumbersome, if not impossible, and since we are interested in the running time for large input sizes, we may talk about the *rate of growth* or the *order of growth* of the running time. For instance, if we can come up with some constant $c > 0$ such that the running time of an algorithm A when presented with an input of size n is at most cn^2, c becomes inconsequential as n gets bigger and bigger. Furthermore, specifying this constant does not bring about extra insight when comparing this function with another one of different order, say dn^3 for an algorithm B that solves the same problem. To see this, note that the ratio between the two functions is dn/c and, consequently, the ratio d/c has virtually no effect as n becomes very large. The same reasoning applies to lower order terms as in the function $f(n) = n^2 \log n + 10n^2 + n$. Here, we observe that the larger the value of n the lesser the significance of the contribution of the lower order terms $10n^2$ and n. Therefore, we may say about the running times of algorithms A and B above to be "*of order*" or "*in the order of*" n^2 and n^3, respectively. Similarly, we say that the function $f(n)$ above is of order $n^2 \log n$.

Once we dispose of lower order terms and leading constants from a function that expresses the running time of an algorithm, we say that we are measuring the *asymptotic running time* of the algorithm. Equivalently, in the analysis of algorithms terminology, we may refer to this asymptotic time using the more technical term "time complexity".

Now, suppose that we have two algorithms A_1 and A_2 of running times in the order of $n \log n$. Which one should we consider to be preferable to the other? Technically, since they have the same time complexity, we say that they have the same running time *within a multiplicative constant*, that is, the ratio between the two running times is constant. In some

cases, the constant may be important and more detailed analysis of the algorithm or conducting some experiments on the behavior of the algorithm may be helpful. Also, in this case, it may be necessary to investigate other factors, e.g. space requirements and input distribution. The latter is helpful in analyzing the behavior of an algorithm on the average.

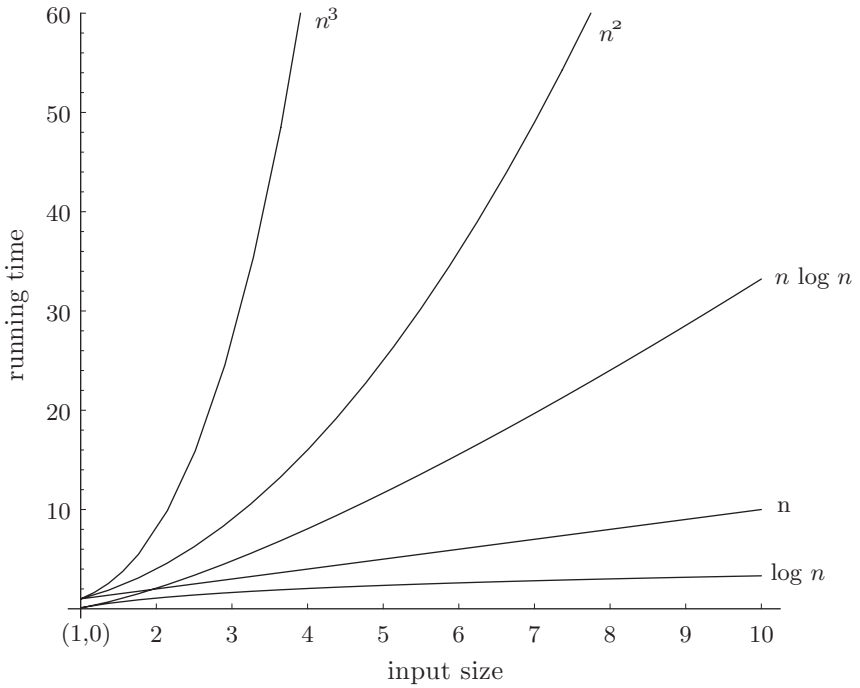

Fig. 1.5 Growth of some typical functions that represent running times.

Figure 1.5 shows some functions that are widely used to represent the running times of algorithms. Higher order functions and exponential and hyperexponential functions are not shown in the figure. Exponential and hyperexponential functions grow much faster than the ones shown in the figure, even for moderate values of n. Functions of the form $\log^k n, cn, cn^2, cn^3$ are called, respectively, *logarithmic*, *linear*, *quadratic* and *cubic*. Functions of the form n^c or $n^c \log^k n, 0 < c < 1$, are called *sublinear*. Functions that lie between linear and quadratic, e.g. $n \log n$ and $n^{1.5}$, are called *subquadratic*. Table 1.1 shows approximate running times of algorithms with

time complexities $\log n, n, n \log n, n^2, n^3$ and 2^n, for $n = 2^3, 2^4, \ldots, 2^{20} \approx$ one million, assuming that each operation takes one nanosecond. Note the explosive running time (measured in centuries) when it is of the order 2^n.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Table 1.1 Running times for different sizes of input. "nsec" stands for nanoseconds, "μ" is one microsecond and "cent" stands for centuries.

Definition 1.1 We denote by an "elementary operation" any computational step whose cost is always upperbounded by a constant amount of time regardless of the input data or the algorithm used.

Let us take, for instance, the operation of adding two integers. For the running time of this operation to be constant, we stipulate that the size of its operands be fixed no matter what algorithm is used. Furthermore, as we are now dealing with the asymptotic running time, we can freely choose any positive integer k to be the "word length" of our "model of computation". Incidentally, this is but one instance in which the beauty of asymptotic notation shows off; the word length can be any *fixed* positive integer. If we want to add arbitrarily large numbers, an algorithm whose running time is proportional to its input size can easily be written in terms of the elementary operation of addition. Likewise, we can choose from a

large pool of operations and apply the fixed-size condition to obtain as many number of elementary operations as we wish. The following operations on *fixed-size* operands are examples of elementary operation.

- Arithmetic operations: addition, subtraction, multiplication and division.
- Comparisons and logical operations.
- Assignments, including assignments of pointers when, say, traversing a list or a tree.

In order to formalize the notions of *order of growth* and *time complexity*, special mathematical notations have been widely used. These notations make it convenient to compare and analyze running times with minimal use of mathematics and cumbersome calculations.

1.8.2 *The O-notation*

We have seen before (Observation 1.4) that the number of elementary operations performed by Algorithm INSERTIONSORT is *at most* cn^2, where c is some appropriately chosen positive constant. In this case, we say that the running time of Algorithm INSERTIONSORT is $O(n^2)$ (read "Oh of n^2" or "big-Oh of n^2"). This can be interpreted as follows. Whenever the number of elements to be sorted is equal to or exceeds some threshold n_0, the running time is *at most* cn^2 for some constant $c > 0$. It should be emphasized, however, that this does not mean that the running time is *always* as large as cn^2, even for large input sizes. Thus, the O-notation provides an *upper bound* on the running time; it *may* not be indicative of the *actual* running time of an algorithm. For example, for any value of n, the running time of Algorithm INSERTIONSORT is $O(n)$ if the input is already sorted in nondecreasing order.

In general, we say that the running time of an algorithm is $O(g(n))$, if whenever the input size is equal to or exceeds some threshold n_0, its running time can be bounded *above* by some positive constant c times $g(n)$. The formal definition of this notation is as follows.[‡]

Definition 1.2 Let $f(n)$ and $g(n)$ be two functions from the set of natural

[‡]The more formal definition of this and subsequent notations is in terms of sets. We prefer not to use their exact formal definitions, as it only complicates things unnecessarily.

numbers to the set of nonnegative real numbers. $f(n)$ is said to be $O(g(n))$ if there exists a natural number n_0 and a constant $c > 0$ such that

$$\forall\ n \geq n_0,\ f(n) \leq cg(n).$$

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \text{ implies } f(n) = O(g(n)).$$

Informally, this definition says that f grows no faster than some constant times g. The O-notation can also be used in equations as a simplification tool. For instance, instead of writing

$$f(n) = 5n^3 + 7n^2 - 2n + 13,$$

we may write

$$f(n) = 5n^3 + O(n^2).$$

This is helpful if we are not interested in the *details* of the lower order terms.

1.8.3 *The Ω-notation*

While the O-notation gives an upper bound, the Ω-notation, on the other hand, provides a *lower bound* within a constant factor of the running time. We have seen before (Observation 1.4) that the number of elementary operations performed by Algorithm INSERTIONSORT is *at least* cn, where c is some appropriately chosen positive constant. In this case, we say that the running time of Algorithm INSERTIONSORT is $\Omega(n)$ (read "omega of n", or "big-omega of n"). This can be interpreted as follows. Whenever the number of elements to be sorted is equal to or exceeds some threshold n_0, the running time is *at least* cn for some constant $c > 0$. As in the O-notation, this does not mean that the running time is *always* as small as cn. Thus, the Ω-notation provides a lower bound on the running time; it *may* not be indicative of the *actual* running time of an algorithm. For example, for any value of n, the running time of Algorithm INSERTIONSORT is $\Omega(n^2)$ if the input consists of distinct elements that are sorted in decreasing order.

In general, we say that an algorithm is $\Omega(g(n))$, if whenever the input size is equal to or exceeds some threshold n_0, its running time can be bounded *below* by some positive constant c times $g(n)$.

This notation is widely used to express lower bounds on *problems* as well. In other words, it is commonly used to state a lower bound for *any* algorithm that solves a specific problem. For example, we say that the problem of matrix multiplication is $\Omega(n^2)$. This is shorthand for saying "any algorithm for multiplying two $n \times n$ matrices is $\Omega(n^2)$". Likewise, we say that the problem of sorting by comparisons is $\Omega(n \log n)$, to mean that no comparison-based sorting algorithm with time complexity that is asymptotically less than $n \log n$ can ever be devised. Chapter 12 is devoted entirely to the study of lower bounds of problems. The formal definition of this notation is symmetrical to that of the O-notation.

Definition 1.3 Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Omega(g(n))$ if there exists a natural number n_0 and a constant $c > 0$ such that

$$\forall\ n \geq n_0,\ f(n) \geq cg(n).$$

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \text{ implies } f(n) = \Omega(g(n)).$$

Informally, this definition says that f grows at least as fast as some constant times g. It is clear from the definition that

$$f(n) \text{ is } \Omega(g(n)) \text{ if and only if } g(n) \text{ is } O(f(n)).$$

1.8.4 *The Θ-notation*

We have seen before that the number of elementary operations performed by Algorithm SELECTIONSORT is *always proportional to* n^2 (Observation 1.3). Since each elementary operation takes a constant amount of time, we say that the running time of Algorithm SELECTIONSORT is $\Theta(n^2)$ (read "theta of n^2"). This can be interpreted as follows. There exist two constants c_1 and c_2 *associated with the algorithm* with the property that on any input of size

$n \geq n_0$, the running time is between $c_1 n^2$ and $c_2 n^2$. These two constants encapsulate many factors pertaining to the details of the implementation of the algorithm and the machine and technology used. As stated earlier, the details of the implementation include numerous factors such as the programming language used and the programmer's skill.

It can also be shown that the running time of Algorithm BOTTOMUPSORT is $\Theta(n \log n)$ for any positive integer n.

In general, we say that the running time of an algorithm is *of order* $\Theta(g(n))$ if whenever the input size is equal to or exceeds some threshold n_0, its running time can be bounded *below* by $c_1 g(n)$ and *above* by $c_2 g(n)$, where $0 < c_1 \leq c_2$. Thus, this notation is used to express the *exact order* of an algorithm, which implies an *exact bound* on its running time. The formal definition of this notation is as follows.

Definition 1.4 Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Theta(g(n))$ if there exists a natural number n_0 and two positive constants c_1 and c_2 such that

$$\forall\ n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ implies } f(n) = \Theta(g(n)),$$

where c is a *constant strictly greater than 0*.

An important consequence of the above definition is that

$$f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

Unlike the previous two notations, the Θ-notation gives an exact picture of the rate of growth of the running time of an algorithm. Thus, the running time of some algorithms as INSERTIONSORT cannot be expressed using this notation, as the running time ranges from linear to quadratic. On the other hand, the running time of some algorithms like Algorithm SELECTIONSORT and Algorithm BOTTOMUPSORT can be described precisely using this notation.

It may be helpful to think of O as *similar to* $\leq$, Ω as *similar to* $\geq$ and Θ as *similar to* $=$. We emphasized the phrase "similar to" since one

should be cautious not to confuse the exact relations with the asymptotic notations. For example $100n = O(n)$ although $100n \geq n$, $n = \Omega(100n)$ although $n \leq 100n$ and $n = \Theta(100n)$ although $n \neq 100n$.

1.8.5 *Examples*

The above O, Ω and Θ notations are not only used to describe the time complexity of an algorithm; they are so general that they can be applied to characterize the asymptotic behavior of any other resource measure, say the amount of space used by an algorithm. Theoretically, they may be used in conjunction with any abstract function. For this reason, we will not attach any measures or meanings with the functions in the examples that follow. We will assume in these examples that $f(n)$ is a function from the set of natural numbers to the set of nonnegative real numbers.

Example 1.5 Let $f(n) = 10n^2 + 20n$. Then, $f(n) = O(n^2)$ since for all $n \geq 1, f(n) \leq 30n^2$. $f(n) = \Omega(n^2)$ since for all $n \geq 1, f(n) \geq n^2$. Also, $f(n) = \Theta(n^2)$ since for all $n \geq 1, n^2 \leq f(n) \leq 30n^2$. We can also establish these three relations using the limits as mentioned above. Since $\lim_{n\to\infty}(10n^2 + 20n)/n^2 = 10$, we see that $f(n) = O(n^2), f(n) = \Omega(n^2)$ and $f(n) = \Theta(n^2)$.

Example 1.6 In general, let $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$. Then, $f(n) = \Theta(n^k)$. Recall that this implies that $f(n) = O(n^k)$ and $f(n) = \Omega(n^k)$.

Example 1.7 Since

$$\lim_{n\to\infty} \frac{\log n^2}{n} = \lim_{n\to\infty} \frac{2\log n}{n} = \lim_{n\to\infty} \frac{2}{\ln 2}\frac{\ln n}{n} = \frac{2}{\ln 2}\lim_{n\to\infty}\frac{1}{n} = 0$$

(differentiate both numerator and denominator), we see that $f(n)$ is $O(n)$, but not $\Omega(n)$. It follows that $f(n)$ is not $\Theta(n)$.

Example 1.8 Since $\log n^2 = 2\log n$, we immediately see that $\log n^2 = \Theta(\log n)$. In general, for any *fixed constant* k, $\log n^k = \Theta(\log n)$.

Example 1.9 Any constant function is $O(1), \Omega(1)$ and $\Theta(1)$.

Example 1.10 It is easy to see that 2^n is $\Theta(2^{n+1})$. This is an example of many functions that satisfy $f(n) = \Theta(f(n+1))$.

Example 1.11 In this example, we give a monotonic increasing function $f(n)$ such that $f(n)$ is not $\Omega(f(n+1))$ and hence not $\Theta(f(n+1))$. Since $(n+1)! =$

$(n+1)n! > n!$, we have that $n! = O((n+1)!)$. Since

$$\lim_{n\to\infty} \frac{n!}{(n+1)!} = \lim_{n\to\infty} \frac{1}{n+1} = 0,$$

we conclude that $n!$ is not $\Omega((n+1)!)$. It follows that $n!$ is not $\Theta((n+1)!)$

Example 1.12 Consider the series $\sum_{j=1}^{n} \log j$. Clearly,

$$\sum_{j=1}^{n} \log j \leq \sum_{j=1}^{n} \log n.$$

That is,

$$\sum_{j=1}^{n} \log j = O(n \log n).$$

Also,

$$\sum_{j=1}^{n} \log j \geq \sum_{j=1}^{\lfloor n/2 \rfloor} \log\left(\frac{n}{2}\right) = \lfloor n/2 \rfloor \log\left(\frac{n}{2}\right) = \lfloor n/2 \rfloor \log n - \lfloor n/2 \rfloor.$$

Thus,

$$\sum_{j=1}^{n} \log j = \Omega(n \log n).$$

It follows that

$$\sum_{j=1}^{n} \log j = \Theta(n \log n).$$

Example 1.13 We want to find an exact bound for the function $f(n) = \log n!$. First, note that $\log n! = \sum_{j=1}^{n} \log j$. We have shown in Example 1.12 that $\sum_{j=1}^{n} \log j = \Theta(n \log n)$. It follows that $\log n! = \Theta(n \log n)$.

Example 1.14 Since $\log n! = \Theta(n \log n)$ and $\log 2^n = n$, we deduce that $2^n = O(n!)$ but $n!$ is not $O(2^n)$. Similarly, since $\log 2^{n^2} = n^2 > n \log n$, and $\log n! = \Theta(n \log n)$ (Example 1.13), it follows that $n! = O(2^{n^2})$, but 2^{n^2} is not $O(n!)$.

Example 1.15 It is easy to see that

$$\sum_{j=1}^{n} \frac{n}{j} \leq \sum_{j=1}^{n} \frac{n}{1} = O(n^2).$$

However, this upper bound is not useful since it is not tight. We will show in Example 2.16 that

$$\frac{\log(n+1)}{\log e} \leq \sum_{j=1}^{n} \frac{1}{j} \leq \frac{\log n}{\log e} + 1.$$

That is

$$\sum_{j=1}^{n} \frac{1}{j} = O(\log n) \text{ and } \sum_{j=1}^{n} \frac{1}{j} = \Omega(\log n).$$

It follows that

$$\sum_{j=1}^{n} \frac{n}{j} = n \sum_{j=1}^{n} \frac{1}{j} = \Theta(n \log n).$$

Example 1.16 Consider the brute-force algorithm for primality test given in Algorithm BRUTE-FORCE PRIMALITYTEST.

Algorithm 1.7 BRUTE-FORCE PRIMALITYTEST
Input: A positive integer $n \geq 2$.
Output: *true* if n is prime and *false* otherwise.

```
1. s ← ⌊√n⌋
2. for j ← 2 to s
3.     if j divides n then return false
4. end for
5. return true
```

We will assume here that $\sqrt{n}$ can be computed in $O(1)$ time. Clearly, the algorithm is $O(\sqrt{n})$, since the number of iterations is exactly $\lfloor\sqrt{n}\rfloor - 1$ when the input is prime. Besides, the number of primes is infinite, which means that the algorithm performs exactly $\lfloor\sqrt{n}\rfloor - 1$ iterations for an infinite number of values of n. It is also easy to see that for infinitely many values of n, the algorithm performs only $O(1)$ iterations (e.g. when n is even), and hence the algorithm is $\Omega(1)$. Since the algorithm may take $\Omega(\sqrt{n})$ time on some inputs and $O(1)$ time on some other inputs infinitely often, it is neither $\Theta(\sqrt{n})$ nor $\Theta(1)$. It follows that the algorithm is not $\Theta(f(n))$ for any function f.

1.8.6 *Complexity Classes and the o-notation*

Let R be the relation on the set of complexity functions defined by $f\ R\ g$ if and only if $f(n) = \Theta(g(n))$. It is easy to see that R is reflexive, symmetric

and transitive, i.e., an equivalence relation (see Sec. 2.1.2.1). The equivalence classes induced by this relation are called *complexity classes*. The complexity class to which a complexity function $g(n)$ belongs includes all functions $f(n)$ of order $\Theta(g(n))$. For example, all polynomials of degree 2 belong to the same complexity class n^2. To show that two functions belong to different classes, it is useful to use the o-notation(read "little oh") defined as follows.

Definition 1.5 Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$ if for *every* constant $c > 0$ there exists a positive integer n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$. Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \text{ implies } f(n) = o(g(n)).$$

Informally, this definition says that $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity. It follows from the definition that

$$f(n) = o(g(n)) \text{ if and only if } f(n) = O(g(n)), \text{ but } g(n) \neq O(f(n)).$$

For example, $n \log n$ is $o(n^2)$ is equivalent to saying that $n \log n$ is $O(n^2)$ but n^2 is *not* $O(n \log n)$.

We also write $f(n) \prec g(n)$ to denote that $f(n)$ is $o(g(n))$. Using this notation, we can concisely express the following hierarchy of complexity classes.

$$1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{3/4} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}.$$

1.9 Space Complexity

We define the space used by an algorithm to be the number of memory cells (or words) needed to carry out the computational steps required to solve an instance of the problem *excluding the space allocated to hold the input*. In other words, it is only the *work space* required by the algorithm. The reason for not including the input size is basically to distinguish between

algorithms that use "less than" linear space throughout their computation. All definitions of order of growth and asymptotic bounds pertaining to time complexity carry over to *space complexity*. It is clear that the work space cannot exceed the running time of an algorithm, as writing into each memory cell requires at least a constant amount of time. Thus, if we let $T(n)$ and $S(n)$ denote, respectively, the time and space complexities of an algorithm, then $S(n) = O(T(n))$.

To appreciate the importance of space complexity, suppose we want to sort $n = 2^{20} = 1,048,576$ elements. If we use Algorithm SELECTIONSORT, then we need no extra storage. On the other hand, if we use Algorithm BOTTOMUPSORT, then we need $n = 1,048,576$ extra memory cells as a temporary storage for the input elements (see Example 1.19 below).

In the following examples, we will look at some of the algorithms we have discussed so far and analyze their space requirements.

Example 1.17 In Algorithm LINEARSEARCH, only one memory cell is used to hold the result of the search. If we add local variables, e.g. for looping, we conclude that the amount of space needed is $\Theta(1)$. This is also the case in algorithms BINARYSEARCH, SELECTIONSORT and INSERTIONSORT.

Example 1.18 In Algorithm MERGE for merging two sorted arrays, we need an auxiliary amount of storage whose size is exactly that of the input, namely n (Recall that n is the size of the array $A[p..r]$). Consequently, its space complexity is $\Theta(n)$.

Example 1.19 When attempting to compute an estimate of the space required by Algorithm BOTTOMUPSORT, one may find it to be complex at first. Nevertheless, it is not difficult to see that the space needed is no more than n, the size of the input array. This is because we can set aside an array of size n, say $B[1..n]$, to be used by Algorithm MERGE as an auxiliary storage for carrying out the merging process. It follows that the space complexity of Algorithm BOTTOMUPSORT is $\Theta(n)$.

Example 1.20 In this example, we will "devise" an algorithm that uses $\Theta(\log n)$ space. Let us modify the Algorithm BINARYSEARCH as follows. After the search terminates, output a *sorted* list of all those entries of array A that have been compared against x. This means that after we test x against $A[mid]$ in each iteration, we must save $A[mid]$ using an auxiliary array, say B, which can be sorted later. As the number of comparisons is at most $\lfloor \log n \rfloor + 1$, it is easy to see that the size of B should be at most this amount, i.e., $O(\log n)$.

Example 1.21 An algorithm that *outputs* all permutations of a given n characters needs only $\Theta(n)$ space. If we want to keep these permutations so that they can be used in subsequent calculations, then we need at least $n \times n! = \Theta((n+1)!)$ space.

Naturally, in many problems there is a time-space tradeoff: The more space we allocate for the algorithm the faster it runs, and vice versa. This, of course, is within limits: in most of the algorithms that we have discussed so far, increasing the amount of space does not result in a noticeable speed-up in the algorithm running time. However, it is almost always the case that decreasing the amount of work space required by an algorithm results in a degradation in the algorithm's speed.

1.10 Optimal Algorithms

In Sec. 12.3.2, we will show that the running time of any algorithm that sorts an array with n entries *using element comparisons* must be $\Omega(n \log n)$ in the worst case (see Sec. 1.12). This means that we cannot hope for an algorithm that runs in time that is asymptotically less than $n \log n$ in the worst case. For this reason, it is commonplace to call any algorithm that sorts using element comparisons in time $O(n \log n)$ *an optimal algorithm for the problem of comparison-based sorting*. By this definition, it follows that Algorithm BOTTOMUPSORT is optimal. In this case, we also say that it is optimal *within a multiplicative constant* to indicate the possibility of the existence of another sorting algorithm whose running time is a *constant* fraction of that of BOTTOMUPSORT. In general, if we can prove that any algorithm to solve problem Π must be $\Omega(f(n))$, then we call any algorithm to solve problem Π in time $O(f(n))$ *an optimal algorithm* for problem Π.

Incidentally, this definition, which is widely used in the literature, does not take into account the space complexity. The reason is twofold. First, as we indicated before, time is considered to be more precious than space so long as the space used is within reasonable limits. Second, most of the existing *optimal* algorithms compare to each other in terms of space complexity in the order of $O(n)$. For example, Algorithm BOTTOMUPSORT, which needs $\Theta(n)$ of space as auxiliary storage, is called optimal, although there are other algorithms that sort in $O(n \log n)$ time and $O(1)$ space. For example, Algorithm HEAPSORT, which will be introduced in Sec. 4.2.3, runs in time $O(n \log n)$ using only $O(1)$ amount of space.

1.11 How to Estimate the Running Time of an Algorithm

As we discussed before, a bound on the running time of an algorithm, be it upper, lower or exact, can be estimated to within a constant factor if we restrict the operations used by the algorithm to those we referred to as *elementary operations*. Now it remains to show how to analyze the algorithm in order to obtain the desired bound. Of course, we can get a precise bound by summing up all elementary operations. This is undoubtedly ruled out, as it is cumbersome and quite often impossible. There is, in general, no mechanical procedure by the help of which one can obtain a "reasonable" bound on the running time or space usage of the algorithm at hand. Moreover, this task is mostly left to intuition and, in many cases, to ingenuity too. However, in many algorithms, there are some agreed upon techniques that give a tight bound with straightforward analysis. In the following, we discuss some of these techniques using simple examples.

1.11.1 *Counting the number of iterations*

It is quite often the case that the running time is proportional to the number of passes through **while** loops and similar constructs. Thus, it follows that counting the number of iterations is a good indicative of the running time of an algorithm. This is the case with many algorithms including those for searching, sorting, matrix multiplication and so forth.

Example 1.22 Consider Algorithm COUNT1, which consists of two nested loops and a variable *count* which counts the number of iterations performed by the algorithm on input $n = 2^k$, for some positive integer k.

Algorithm 1.8 COUNT1
Input: $n = 2^k$, for some positive integer k.
Output: *count* = number of times Step 4 is executed.

```
1. count ← 0
2. while n ≥ 1
3.     for j ← 1 to n
4.         count ← count + 1
5.     end for
6.     n ← n/2
7. end while
8. return count
```

The **while** loop is executed $k+1$ times, where $k = \log n$. The **for** loop is executed n times, and then $n/2, n/4, \ldots, 1$. Therefore, applying Formula 2.11 (page 78) on this geometric series, the number of times Step 4 is executed is

$$\sum_{j=0}^{k} \frac{n}{2^j} = n \sum_{j=0}^{k} \frac{1}{2^j} = n(2 - \frac{1}{2^k}) = 2n - 1 = \Theta(n).$$

Since the running time is proportional to *count*, we conclude that it is $\Theta(n)$.

Example 1.23 Consider Algorithm COUNT2, which consists of two nested loops and a variable *count* which counts the number of iterations performed by the algorithm on input n, which is a positive integer.

Algorithm 1.9 COUNT2
Input: A positive integer n.
Output: *count* = number of times Step 5 is executed.

```
1. count ← 0
2. for i ← 1 to n
3.     m ← ⌊n/i⌋
4.     for j ← 1 to m
5.         count ← count + 1
6.     end for
7. end for
8. return count
```

The inner **for** loop is executed repeatedly for the following values of n:

$$n, \lfloor n/2 \rfloor, \lfloor n/3 \rfloor, \ldots, \lfloor n/n \rfloor.$$

Thus, the total number of times Step 5 is executed is

$$\sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor.$$

Since

$$\sum_{i=1}^{n} \left(\frac{n}{i} - 1 \right) \leq \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \leq \sum_{i=1}^{n} \frac{n}{i},$$

we conclude that Step 5 is executed $\Theta(n \log n)$ times (see Examples 1.15 and 2.16). As the running time is proportional to *count*, we conclude that it is $\Theta(n \log n)$.

Example 1.24 Consider Algorithm COUNT3, which consists of two nested loops and a variable *count* which counts the number of iterations performed by

the **while** loop on input n that is of the form 2^{2^k}, for some positive integer k. For each value of i, the **while** loop will be executed when

$$j = 2, 2^2, 2^4, \ldots, 2^{2^k}.$$

That is, it will be executed when

$$j = 2^{2^0}, 2^{2^1}, 2^{2^2}, \ldots, 2^{2^k}.$$

Thus, the number of iterations performed by the **while** loop is $k+1 = \log\log n+1$ for each iteration of the **for** loop. It follows that the total number of iterations performed by the **while** loop, which is the output of the algorithm, is exactly $n(\log\log n + 1) = \Theta(n \log\log n)$. We conclude that the running time of the algorithm is $\Theta(n \log\log n)$.

Algorithm 1.10 COUNT3
Input: $n = 2^{2^k}$, for some positive integer k.
Output: Number of times Step 6 is executed.

```
1. count ← 0
2. for i ← 1 to n
3.     j ← 2
4.     while j ≤ n
5.         j ← j^2
6.         count ← count + 1
7.     end while
8. end for
9. return count
```

Example 1.25 Let n be a perfect square, i.e., an integer whose square root is integer. Algorithm PSUM computes for each perfect square j between 1 and n the sum $\sum_{i=1}^{j} i$. (Obviously, this sum can be computed more efficiently).

We will assume that $\sqrt{n}$ can be computed in $O(1)$ time. We compute the running time of the algorithm as follows. The outer and inner **for** loops are executed $k = \sqrt{n}$ and j^2 times, respectively. Hence, the number of iterations performed by the inner **for** loop is

$$\sum_{j=1}^{k} \sum_{i=1}^{j^2} 1 = \sum_{j=1}^{k} j^2 = \frac{k(k+1)(2k+1)}{6} = \Theta(k^3) = \Theta(n^{1.5}).$$

It follows that the running time of the algorithm is $\Theta(n^{1.5})$.

Algorithm 1.11 PSUM
Input: $n = k^2$ for some integer k.
Output: $\sum_{i=1}^{j} i$ for each perfect square j between 1 and n.

```
1. k ← √n
2. for j ← 1 to k
3.     sum[j] ← 0
4.     for i ← 1 to j^2
5.         sum[j] ← sum[j] + i
6.     end for
7. end for
8. return sum[1..k]
```

1.11.2 *Counting the frequency of basic operations*

In some algorithms, it is cumbersome, or even impossible, to make use of the previous method in order to come up with a *tight* estimate of its running time. Unfortunately, at this point we have not covered good examples of such algorithms. Good examples that will be covered in subsequent chapters include the single-source shortest path problem, Prim's algorithm for finding minimum spanning trees, depth-first search, computing convex hulls and others. However, Algorithm MERGE will serve as a reasonable candidate. Recall that the function of Algorithm MERGE is to merge two sorted arrays into one sorted array. In this algorithm, if we try to apply the previous method, the analysis becomes lengthy and awkward. Now, consider the following argument which we have alluded to in Sec. 1.4. Just prior to Step 15 of the algorithm is executed, array B holds the final sorted list. Thus, for each element $x \in A$, the algorithm executes one element assignment operation that moves x from A to B. Similarly, in Step 15, the algorithm executes n element assignment operations in order to copy B back into A. This implies that the algorithm executes *exactly* $2n$ element assignments (Observation 1.2). On the other hand, there is no other operation that is executed more than $2n$ times. For example, *at most* one element comparison is needed to move each element from A to B (Observation 1.1).

In general, when analyzing the running time of an algorithm, we may be able to single out one elementary operation with the property that its frequency is at least as large as any other operation. Let us call such an operation a *basic operation*. We can relax this definition to include any operation whose frequency is *proportional* to the running time.

Definition 1.6 An elementary operation in an algorithm is called a *basic operation* if it is of highest frequency to within a constant factor among all other elementary operations.

Hence, according to this definition, the operation of element assignment is a basic operation in Algorithm MERGE and thus is indicative of its running time. By observation 1.2, the number of element assignments needed to merge two arrays into one array of size n is exactly $2n$. Consequently, its running time is $\Theta(n)$. Note that the operation of element comparison is in general *not* a basic operation in Algorithm MERGE, as there may be only one element comparison throughout the execution of the algorithm. If, however, the algorithm is to merge two arrays of approximately the same size (e.g. $\lfloor (n/2) \rfloor$ and $\lceil (n/2) \rceil$), then we may safely say that it is basic *for that special instance*. This happens if, for example, the algorithm is invoked by Algorithm BOTTOMUPSORT in which case the two subarrays to be sorted are of sizes $\lfloor (n/2) \rfloor$ and $\lceil (n/2) \rceil$.

In general, this method consists of identifying one basic operation and utilizing one of the asymptotic notations to find out the order of execution of this operation. This order will be the order of the running time of the algorithm. This is indeed the method of choice for a large class of problems. We list here some candidates of these basic operations:

- When analyzing searching and sorting algorithms, we may choose the element comparison operation *if it is an elementary operation*.
- In matrix multiplication algorithms, we select the operation of scalar multiplication.
- In traversing a linked list, we may select the "operation" of setting or updating a pointer.
- In graph traversals, we may choose the "action" of visiting a node, and count the number of nodes visited.

Example 1.26 Using this method, we obtain an exact bound for Algorithm BOTTOMUPSORT as follows. First, note that the basic operations in this algorithm are inherited from Algorithm MERGE, as the latter is called by the former in each iteration of the **while** loop. By the above discussion, we may safely choose the elementary operation of element comparison as the basic operation. By observation 1.5, the total number of element comparisons required by the algorithm when n is a power of 2 is between $(n \log n)/2$ and $n \log n - n + 1$. This means that the number of element comparisons when n is a power of 2 is $\Omega(n \log n)$

and $O(n \log n)$, i.e., $\Theta(n \log n)$. It can be shown that this holds even if n is not a power of 2. Since the operation of element comparison used by the algorithm is of maximum frequency to within a constant factor, we conclude that the running time of the algorithm is proportional to the number of comparisons. It follows that the algorithm runs in time $\Theta(n \log n)$.

One should be careful, however, when choosing a basic operation, as illustrated by the following example.

Example 1.27 Consider the following modification to Algorithm INSERTIONSORT. When trying to insert an element of the array in its proper position, we will not use linear search; instead, we will use a binary search *technique* similar to Algorithm BINARYSEARCH. Algorithm BINARYSEARCH can easily be modified so that it does *not* return 0 when x is not an entry of array A; instead, it returns the position of x relative to other entries of the sorted array A. For example, when Algorithm BINARYSEARCH is called with $A = \boxed{2}\boxed{3}\boxed{6}\boxed{8}\boxed{9}$ and $x = 7$, it returns 4. Incidentally, this shows that using binary search is not confined to testing for the membership of an element x in an array A; in many algorithms, it is used to find the *position* of an element x relative to other elements in a sorted list. Let Algorithm MODBINARYSEARCH be some implementation of this binary search technique. Thus, MODBINARYSEARCH$(\{2, 3, 6, 8, 9\}, 7) = 4$. The modified sorting algorithm is given in Algorithm MODINSERTIONSORT.

Algorithm 1.12 MODINSERTIONSORT
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for i ← 2 to n
2.     x ← A[i]
3.     k ← MODBINARYSEARCH (A[1..i − 1], x)
4.     for j ← i − 1 downto k
5.         A[j + 1] ← A[j]
6.     end for
7.     A[k] ← x
8. end for
```

The total number of element comparisons are those performed by Algorithm MODBINARYSEARCH. Since this algorithm is called $n - 1$ times, and since the maximum number of comparisons performed by the binary search algorithm on an array of size $i - 1$ is $\lfloor \log(i - 1) \rfloor + 1$ (Theorem 1.1), it follows that the total

number of comparisons done by Algorithm MODINSERTIONSORT is *at most*

$$\sum_{i=2}^{n}(\lfloor\log(i-1)\rfloor+1) = n-1+\sum_{i=1}^{n-1}\lfloor\log i\rfloor \leq n-1+\sum_{i=1}^{n-1}\log i = \Theta(n\log n).$$

The last equality follows from Example 1.12 and Eq. 2.18 on page 81. One may be tempted to conclude, based on the false assumption that the operation of element comparison is basic, that the overall running time is $O(n\log n)$. However, this is not the case, as the number of element assignments in Algorithm MODINSERTIONSORT is exactly that in Algorithm INSERTIONSORT when the two algorithms are run on the same input. This has been shown to be $O(n^2)$ (Observation 1.4). We conclude that this algorithm runs in time $O(n^2)$, and not $O(n\log n)$.

In some algorithms, all elementary operations are not basic. In these algorithms, it may be the case that the frequency of two or more operations combined together may turn out to be proportional to the running time of the algorithm. In this case, we express the running time as a function of the total number of times these operations are executed. For instance, if we cannot bound the number of either insertions or deletions, but can come up with a formula that bounds their total, then we may say something like: There are at most n insertions and deletions. This method is widely used in graph and network algorithms. Here we give a simple example that involves only numbers and the two operations of addition and multiplication. There are better examples that involve graphs and complex data structures.

Example 1.28 Suppose we are given an array $A[1..n]$ of n integers and a positive integer $k, 1 \leq k \leq n$, and asked to multiply the first k integers in A and add the rest. An algorithm to do this is sketched below. Observe here that there are *no* basic operations, since the running time is proportional to the number of times *both* additions and multiplications are performed. Thus, we conclude that there are n elementary operations: multiplications *and* additions, which implies a bound of $\Theta(n)$. Note that in this example, we could have counted the number of iterations to obtain a precise measure of the running time as well. This is because in each iteration, the algorithm takes a constant amount of time. The total number of iterations is $k+(n-k)=n$.

1.11.3 *Using recurrence relations*

In recursive algorithms, a formula bounding the running time is usually given in the form of a recurrence relation, that is, a function whose definition

```
1. prod ← 1;   sum ← 0
2. for j ← 1 to k
3.     prod ← prod × A[j]
4. end for
5. for j ← k + 1 to n
6.     sum ← sum + A[j]
7. end for
```

contains the function itself, e.g. $T(n) = 2T(n/2) + n$. Finding the solution of a recurrence relation has been studied well to the extent that the solution of a recurrence may be obtained mechanically (see Sec. 2.8 for a discussion on recurrence relations). It may be possible to derive a recurrence that bounds the number of basic operations in a nonrecursive algorithm. For example, in Algorithm BINARYSEARCH, if we let $C(n)$ be the number of comparisons performed on an instance of size n, we may express the number of comparisons done by the algorithm using the recurrence

$$C(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ C(\lfloor n/2 \rfloor) + 1 & \text{if } n \geq 2. \end{cases}$$

The solution to this recurrence reduces to a summation as follows.

$$\begin{aligned} C(n) &\leq C(\lfloor n/2 \rfloor) + 1 \\ &= C(\lfloor \lfloor n/2 \rfloor / 2 \rfloor) + 1 + 1 \\ &= C(\lfloor n/4 \rfloor) + 1 + 1 \quad \text{(Eq. 2.3, page 71)} \\ &\vdots \\ &= \lfloor \log n \rfloor + 1. \end{aligned}$$

That is, $C(n) \leq \lfloor \log n \rfloor + 1$. It follows that $C(n) = O(\log n)$. Since the operation of element comparison is a basic operation in Algorithm BINARYSEARCH, we conclude that its time complexity is $O(\log n)$.

1.12 Worst case and average case analysis

Consider the problem of adding two $n \times n$ matrices A and B of integers. Clearly, the running time expressed in the number of scalar additions of an algorithm that computes $A + B$ is the same for any two arbitrary $n \times n$ matrices A and B. That is, the running time of the algorithm is insensitive

to the input values; it is dependent only on its size measured in the number of entries. This is to be contrasted with an algorithm like INSERTIONSORT whose running time is highly dependent on the *input values* as well. By Observation 1.4, the number of element comparisons performed on an input array of size n lies between $n-1$ and $n(n-1)/2$ inclusive. This indicates that the performance of the algorithm is *not only a function of* n, but also a function of the original order of the input elements. The dependence of the running time of an algorithm on the form of input data, not only its number, is characteristic of many problems. For instance, the process of sorting is inherently dependent on the relative order of the data to be sorted. This does not mean that *all* sorting algorithms are sensitive to input data. For instance, the number of element comparisons performed by Algorithm SELECTIONSORT on an array of size n is the same regardless of the form or order of input values, as the number of comparisons done by the algorithm is a function of n only. More precisely, the time taken by a comparison-based algorithm to sort a set of n elements depends on their relative order. For instance, the number of steps required to sort the numbers 6, 3, 4, 5, 1, 7, 2 is the same as that for sorting the numbers 60, 30, 40, 50, 10, 70, 20. Obviously, it is impossible to come up with a function that describes the time complexity of an algorithm based on *both* input size and form; the latter, definitely, has to be suppressed.

Consider again Algorithm INSERTIONSORT. Let $A[1..n] = \{1, 2, \ldots, n\}$, and consider all $n!$ permutations of the elements in A. Each permutation corresponds to one possible input. The running time of the algorithm presumably differs from one permutation to another. Consider three permutations: a in which the elements in A are sorted in decreasing order, c in which the elements in A are already sorted in increasing order and b in which the elements are ordered randomly (see Fig. 1.6).

Thus, input a is a representative of the *worst case* of all inputs of size n, input c is a representative of the *best case* of all inputs of size n and input b is between the two. This gives rise to three methodologies for analyzing the running time of an algorithm: worst case analysis, average case analysis and best case analysis. The latter is not used in practice, as it does not give useful information about the behavior of an algorithm in general.

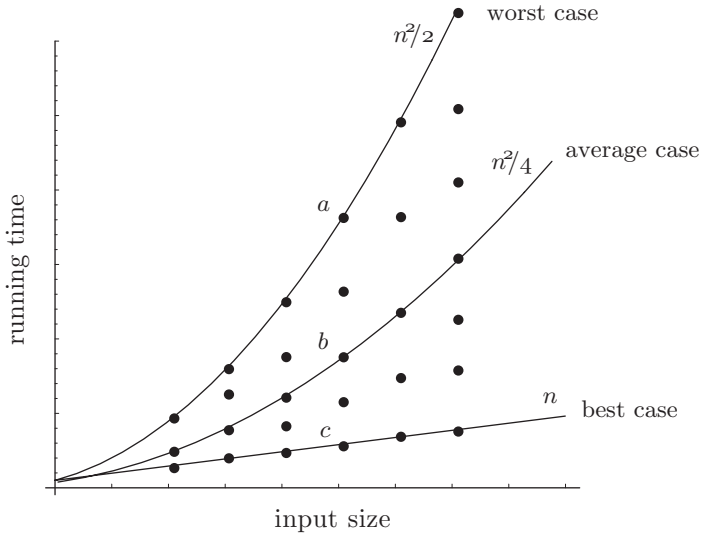

Fig. 1.6 Performance of Algorithm INSERTIONSORT: worst, average and best cases.

1.12.1 *Worst case analysis*

In worst case analysis of time complexity we select the maximum cost among all possible inputs of size n. As stated above, for any positive integer n, Algorithm INSERTIONSORT requires $\Omega(n^2)$ to process some inputs of size n (e.g. input a in Fig. 1.6). For this reason, we say that the running time of this algorithm is $\Omega(n^2)$ *in the worst case*. Since the running time of the algorithm is $O(n^2)$, we also say that the running time of the algorithm is $O(n^2)$ *in the worst case*. Consequently, we may use the stronger Θ-notation and say that the running time of the algorithm is $\Theta(n^2)$ *in the worst case*. Clearly, using the Θ-notation is preferred, as it gives the exact behavior of the algorithm in the worst case. In other words, stating that Algorithm INSERTIONSORT has a running time of $\Theta(n^2)$ in the worst case implies that it is also $\Omega(n^2)$ in the worst case, whereas stating that Algorithm INSERTIONSORT runs in $O(n^2)$ in the worst case does not. Note that for any value of n there are input instances on which the algorithm spends no more than $O(n)$ time (e.g. input c in Fig. 1.6).

It turns out that under the worst case assumption, the notions of upper and lower bounds in many algorithms coincide and, consequently, we may say that an algorithm runs in time $\Theta(f(n))$ *in the worst case*. As explained

above, this is stronger than stating that the algorithm is $O(f(n))$ *in the worst case*. As another example, we have seen before that Algorithm LINEARSEARCH is $O(n)$ and $\Omega(1)$. In the worst case, this algorithm is both $O(n)$ and $\Omega(n)$, i.e., $\Theta(n)$.

One may be tempted, however, to conclude that in the worst case the notions of upper and lower bounds *always* coincide. This in fact is not the case. Consider for example an algorithm whose running time is known to be $O(n^2)$ in the worst case. However, it has not been proven that for all values of n greater than some threshold n_0 there exists an input of size n on which the algorithm spends $\Omega(n^2)$ time. In this case, we cannot claim that the algorithm's running time is $\Theta(n^2)$ *in the worst case*, even if we know that the algorithm takes $\Theta(n^2)$ time for *infinitely many* values of n. It follows that the algorithm's running time is not $\Theta(n^2)$ *in the worst case*. This is the case in many graph and network algorithms for which only an upper bound on the number of operations can be proven, and whether this upper bound is achievable is not clear. The next example gives a concrete instance of this case.

Example 1.29 Consider for example the procedure shown below whose input is an element x and a sorted array A of n elements.

```
1. if n is odd then k ← BINARYSEARCH(A, x)
2. else k ← LINEARSEARCH(A, x)
```

This procedure searches for x in A using binary search if n is odd and linear search if n is even. Obviously, the running time of this procedure is $O(n)$, since when n is even, the running time is that of Algorithm LINEARSEARCH, which is $O(n)$. However, the procedure is *not* $\Omega(n)$ in the worst case because there does not exist a threshold n_0 such that *for all* $n \geq n_0$ there exists some input of size n that causes the algorithm to take at least cn time for some constant c. We can only ascertain that the running time is $\Omega(\log n)$ in the worst case. Note that the running time being $\Omega(n)$ for infinitely many values of n does not mean that the algorithm's running time is $\Omega(n)$ in the worst case. It follows that, in the worst case, this procedure is $O(n)$ and $\Omega(\log n)$, which implies that, in the worst case, it is not $\Theta(f(n))$ for any function $f(n)$.

1.12.2 *Average case analysis*

Another interpretation of an algorithm's time complexity is that of the average case. Here, the running time is taken to be the average time over all inputs of size n (see Fig. 1.6). In this method, it is necessary to know the probabilities of all input occurrences, i.e., it requires prior knowledge of the input distribution. However, even after relaxing some constraints including the assumption of a convenient input distribution, e.g. uniform distribution, the analysis is in many cases complex and lengthy.

Example 1.30 Consider Algorithm LINEARSEARCH. To simplify the analysis, let us assume that $A[1..n]$ contains the numbers 1 through n, which implies that all elements of A are distinct. Also, we will assume that x is in the array, i.e., $x \in \{1, 2, \ldots, n\}$. Furthermore, and most importantly indeed, we will assume that each element y in A is equally likely to be in any position in the array. In other words, the probability that $y = A[j]$ is $1/n$, for all $y \in A$. To this end, the number of comparisons performed by the algorithm on the average to find the position of x is

$$T(n) = \sum_{j=1}^{n} j \times \frac{1}{n} = \frac{1}{n} \sum_{j=1}^{n} j = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

This shows that, on the average, the algorithm performs $(n + 1)/2$ element comparisons in order to locate x. Hence, the time complexity of Algorithm LINEARSEARCH is $\Theta(n)$ on the average.

Example 1.31 Consider computing the average number of comparisons performed by Algorithm INSERTIONSORT. To simplify the analysis, let us assume that $A[1..n]$ contains the numbers 1 through n, which implies that all elements of A are distinct. Furthermore, we will assume that all $n!$ permutations of the numbers $1, 2, \ldots, n$ are equally likely. Now, consider inserting element $A[i]$ in its proper position in $A[1..i]$. If its proper position is $j, 1 \leq j \leq i$, then the number of comparisons performed in order to insert $A[i]$ in its proper position is $i - j$ if $j = 1$ and $i - j + 1$ if $2 \leq j \leq i$. Since the probability that its proper position in $A[1..i]$ is $1/i$, the average number of comparisons needed to insert $A[i]$ in its proper position in $A[1..i]$ is

$$\frac{i-1}{i} + \sum_{j=2}^{i} \frac{i-j+1}{i} = \frac{i-1}{i} + \sum_{j=1}^{i-1} \frac{j}{i} = 1 - \frac{1}{i} + \frac{i-1}{2} = \frac{i}{2} - \frac{1}{i} + \frac{1}{2}.$$

Thus, the average number of comparisons performed by Algorithm INSERTIONSORT is

$$\sum_{i=2}^{n}\left(\frac{i}{2}-\frac{1}{i}+\frac{1}{2}\right)=\frac{n(n+1)}{4}-\frac{1}{2}-\sum_{i=2}^{n}\frac{1}{i}+\frac{n-1}{2}=\frac{n^2}{4}+\frac{3n}{4}-\sum_{i=1}^{n}\frac{1}{i}.$$

Since

$$\ln(n+1)\le\sum_{i=1}^{n}\frac{1}{i}\le\ln n+1 \qquad \text{(Eq. 2.16, page 80)},$$

it follows that the average number of comparisons performed by Algorithm INSERTIONSORT is approximately

$$\frac{n^2}{4}+\frac{3n}{4}-\ln n=\Theta(n^2).$$

Thus, on the average, Algorithm INSERTIONSORT performs roughly half the number of operations performed in the worst case (see Fig. 1.6).

1.13 Amortized analysis

In many algorithms, we may be unable to express the time complexity in terms of the Θ-notation to obtain an exact bound on the running time. Therefore, we will be content with the O-notation, which is sometimes pessimistic. If we use the O-notation to obtain an upper bound on the running time, the algorithm may be much faster than our estimate even in the worst case.

Consider an algorithm in which an operation is executed repeatedly with the property that its running time fluctuates throughout the execution of the algorithm. If this operation takes a large amount of time *occasionally* and runs much faster most of the time, then this is an indication that *amortized analysis* should be employed, assuming that an exact bound is too hard, if not impossible.

In amortized analysis, we average out the time taken by the operation throughout the execution of the algorithm, and refer to this average as the *amortized running time* of that operation. Amortized analysis guarantees the average cost of the operation, and thus the algorithm, *in the worst case*. This is to be contrasted with the average time analysis in which the average is taken over all instances of the same size. Moreover, unlike the

average case analysis, no assumptions about the probability distribution of the input are needed.

Amortized time analysis is generally harder than worst case analysis, but this hardness pays off when we derive a lower time complexity. A good example of this analysis will be presented in Sec. 4.3 when we study the union-find algorithms, which is responsible for maintaining a data structure for disjoint sets. It will be shown that this algorithm runs in time that is almost linear using amortized time analysis as opposed to a straightforward bound of $O(n \log n)$. In this section, we present here two simple examples that convey the essence of amortization.

Example 1.32 Consider the following problem. We have a doubly linked list (see Sec. 3.2) that initially consists of one node which contains the integer 0. We have as input an array $A[1..n]$ of n positive integers that are to be processed in the following way. If the current integer x is odd, then append x to the list. If it is even, then first append x and then remove all odd elements before x in the list. A sketch of an algorithm for this problem is shown below and is illustrated in Fig. 1.7 on the input

5	7	3	4	9	8	7	3

.

```
1. for j ← 1 to n
2.     x ← A[j]
3.     append x to the list
4.     if x is even then
5.         while pred(x) is odd
6.             delete pred(x)
7.         end while
8.     end if
9. end for
```

First, 5, 7 and 3 are appended to the list. When 4 is processed, it is inserted and then 5, 7 and 3 are deleted as shown in Fig. 1.7(f). Next, as shown in Fig. 1.7(i), after 9 and 8 have been inserted, 9 is deleted. Finally, the elements 7 and 3 are inserted but not deleted, as they do not precede any input integer that is even.

Now, let us analyze the running time of this algorithm. If the input data contain no even integers, or if all the even integers are at the beginning, then no elements are deleted, and hence each iteration of the **for** loop takes constant time. On the other hand, if the input consists of $n-1$ odd integers followed by one even integer, then the number of deletions is exactly $n-1$, i.e., the number

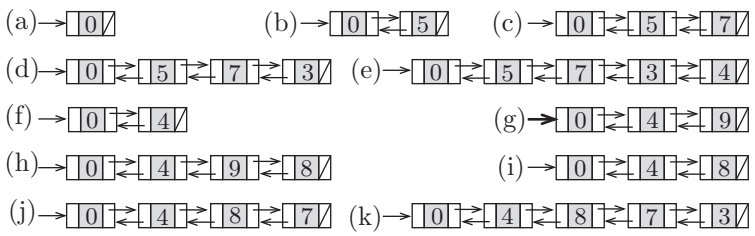

Fig. 1.7 Illustration of amortized time analysis.

of iterations of the **while** loop is $n-1$. This means that the **while** loop may cost $\Omega(n)$ time in some iterations. It follows that each iteration of the **for** loop takes $O(n)$ time, which results in an overall running time of $O(n^2)$.

Using amortization, however, we obtain a time complexity of $\Theta(n)$ as follows. The number of insertions is obviously n. As to the number of deletions, we note that no element is deleted more than once, and thus the number of deletions is between 0 and $n-1$. It follows that the total number of elementary operations of insertions and deletions altogether is between n and $2n-1$. This implies that the time complexity of the algorithm is indeed $\Theta(n)$. It should be emphasized, however, that in this case we say that the **while** loop takes *constant amortized time in the worst case*. That is, the average time taken by the **while** loop is *guaranteed* to be $O(1)$ regardless of the input.

Example 1.33 Suppose we want to allocate storage for an unknown number of elements $x_1, x_2, \ldots$ in a stream of input. One technique to handle the allocation of memory is to first allocate an array A_0 of reasonable size, say m. When this array becomes full, then upon the arrival of the $(m+1)$st element, a new array A_1 of size $2m$ is allocated and all the elements stored in A_0 are moved from A_0 to A_1. Next, the $(m+1)$st element is stored in $A_1[m+1]$. We keep doubling the size of the newly allocated array whenever it becomes full and a new element is received, until all elements have been stored.

Suppose, for simplicity, that we start with an array of size 1, that is A_0 consists of only one entry. First, upon arrival of x_1, we store x_1 in $A_0[1]$. When x_2 is received, we allocate a new array A_1 of size 2, set $A_1[1]$ to $A_0[1]$ and store x_2 in $A_1[2]$. Upon arrival of the third element x_3, we allocate a new array $A_2[1..4]$, move $A_1[1..2]$ to $A_2[1..2]$ and store x_3 in $A_2[3]$. The next element, x_4, will be stored directly in $A_2[4]$. Now since A_2 is full, when x_5 is received, we allocate a new array $A_3[1..8]$, move $A_2[1..4]$ to $A_3[1..4]$ and store x_5 in $A_3[5]$. Next, we store x_6, x_7 and x_8 in the remaining free positions of A_3. We keep doubling the size of the newly allocated array upon arrival of a new element whenever the current array becomes full, and move the contents of the current array to the newly allocated array.

We wish to count the number of element assignments. Suppose, for simplicity, that the total number of elements received, which is n, is a power of 2. Then the arrays that have been allocated are $A_0, A_1, \ldots, A_k$, where $k = \log n$. Since x_1 has been moved k times, x_2 has been moved $k-1$ times, etc., we may conclude that each element in $\{x_1, x_2, \ldots, x_n\}$ has been moved $O(k) = O(\log n)$ times. This implies that the total number of element assignments is $O(n \log n)$.

However, using amortized time analysis, we derive a much tighter bound as follows. Observe that every entry in each newly allocated array has been assigned to exactly once. Consequently, the total number of element assignments is equal to the sum of sizes of all arrays that have been allocated, which is equal to

$$\sum_{j=0}^{k} 2^j = 2^{k+1} - 1 = 2n - 1 = \Theta(n) \quad \text{(Eq. 2.9)}.$$

Thus, using amortization, it follows that the time needed to store and move each of the elements $x_1, x_2, \ldots, x_n$ is $\Theta(1)$ *amortized time*.

1.14 Input Size and Problem Instance

A measure of the performance of an algorithm is usually a function of its input: its size, order, distribution, etc. The most prominent of these, which is of interest to us here, is the input size. Using Turing machines as the model of computation, it is possible, and more convenient indeed, to measure the input to an algorithm in terms of the number of nonblank cells. This, of course, is impractical, given that we wish to investigate real world problems that can be described in terms of numbers, vertices, line segments, and other varieties of objects. For this reason, the notion of *input size* belongs to the practical part of algorithm analysis, and its interpretation has become a matter of convention. When discussing a problem, as opposed to an algorithm, we usually talk of a *problem instance*. Thus, a problem instance translates to input in the context of an algorithm that solves that problem. For example, we call an array A of n integers an instance of the problem of sorting numbers. At the same time, in the context of discussing Algorithm INSERTIONSORT, we refer to this array as an input to the algorithm.

The input size, as a quantity, is not a precise measure of the input, and its interpretation is subject to the problem for which the algorithm is, or is to be, designed. Some of the commonly used measures of input size are the following:

- In sorting and searching problems, we use the number of entries in the array or list as the input size.
- In graph algorithms, the input size usually refers to the number of vertices or edges in the graph, or both.
- In computational geometry, the size of input to an algorithm is usually expressed in terms of the number of points, vertices, edges, line segments, polygons, etc.
- In matrix operations, the input size is commonly taken to be the dimensions of the input matrices.
- In number theory algorithms and cryptography, the number of bits in the input is usually chosen to denote its length. The number of words used to represent a single number may also be chosen as well, as each word consists of a fixed number of bits.

These "heterogeneous" measures have brought about some inconsistencies when comparing the amount of time or space required by two algorithms. For example, an algorithm for adding two $n \times n$ matrices which performs n^2 additions sounds quadratic, while it is indeed linear in the input size.

The the brute-force algorithm for primality testing given in Example 1.16. Its time complexity was shown to be $O(\sqrt{n})$. Since this is a number problem, the time complexity of the algorithm is measured in terms of the number of bits in the binary representation of n. Since n can be represented using $k = \lceil \log(n+1) \rceil$ bits, the time complexity can be rewritten as $O(\sqrt{n}) = O(2^{k/2})$. Consequently, Algorithm BRUTE-FORCE PRIMALITYTEST is in fact an exponential algorithm.

Now we will compare two algorithms for computing the sum $\sum_{j=1}^{n} j$. In the first algorithm, which we will call FIRST, the input is an array $A[1..n]$ with $A[j] = j$, for each $j, 1 \leq j \leq n$. The input to the second algorithm, call it SECOND, is just the number n. These two algorithms are shown as Algorithm FIRST and Algorithm SECOND.

Obviously, both algorithms run in time $\Theta(n)$. Clearly, the time complexity of Algorithm FIRST is $\Theta(n)$. Algorithm SECOND is designed to solve a *number problem* and, as we have stated before, its input size is measured in terms of the number of bits in the binary representation of the integer n. Its input consists of $k = \lceil \log(n+1) \rceil$ bits. It follows that the time complexity of Algorithm SECOND is $\Theta(n) = \Theta(2^k)$. In other words, it is considered to be an *exponential* time algorithm. Ironically, the number of

Algorithm 1.13 FIRST
Input: A positive integer n and an array $A[1..n]$ with $A[j] = j, 1 \leq j \leq n$.
Output: $\sum_{j=1}^{n} A[j]$.

1. $sum \leftarrow 0$
2. **for** $j \leftarrow 1$ **to** n
3. $\quad sum \leftarrow sum + A[j]$
4. **end for**
5. **return** sum

Algorithm 1.14 SECOND
Input: A positive integer n.
Output: $\sum_{j=1}^{n} j$.

1. $sum \leftarrow 0$
2. **for** $j \leftarrow 1$ **to** n
3. $\quad sum \leftarrow sum + j$
4. **end for**
5. **return** sum

elementary operations performed by both algorithms is the same.