Part I

Fundamentals

Chapter 1

Introduction

A large software system consists of many components, each designed to perform a specific task. The overall quality of a given system is determined largely by how well these components function individually and interact with each other and the external environment. The focus of this book is on individual components — algorithms and data structures — that can be used in a wide variety of software systems. In this chapter, we will present an overview of what these components are and how we will study them throughout the remainder of the book. In the course of this overview, we will illustrate a *top-down approach* for thinking about both algorithms and data structures. This approach will provide an essential framework for both designing and understanding algorithms and data structures.

1.1 Specifications

Before we can design or analyze any software component — including an algorithm or a data structure — we must first know what it is supposed to accomplish. A formal statement of what a software component is meant to accomplish is called a *specification*. Here, we will discuss specifically the specification of an algorithm. The specification of a data structure is similar, but a bit more involved. For this reason, we will wait until Chapter 4 to discuss the specification of data structures in detail.

Suppose, for example, that we wish to find the kth smallest element of an array of n numbers. Thus, if $k = 1$, we are looking for the smallest element in the array, or if $k = n$, we are looking for the largest. We will refer to this problem as the *selection problem*. Even for such a seemingly simple problem, there is a potential for ambiguity if we are not careful to state the

problem precisely. For example, what do we expect from the algorithm if k is larger than n? Do we allow the algorithm to change the array? Do all of the elements in the array need to be different? If not, what exactly do we mean by the kth smallest?

Specifically, we need to state the following:

- a *precondition*, which is a statement of the assumptions we make about the input to the algorithm and the environment in which it will be executed; and
- a *postcondition*, which is a statement of the required result of executing the algorithm, assuming the precondition is satisfied.

The precondition and postcondition, together with a *function header* giving a name and parameter list, constitute the *specification* of the algorithm. We say that the algorithm *meets its specification* if the postcondition is satisfied whenever the precondition is satisfied. Note that we guarantee nothing about the algorithm if its precondition is not satisfied.

The selection problem will have two parameters, an array A and a positive integer k. The precondition should specify what we are assuming about these parameters. For this problem, we will assume that the first element of A is indexed by 1 and the last element is indexed by some natural number (i.e., nonnegative integer) n; hence, we describe the array more precisely using the notation $A[1..n]$. We will assume that each element of $A[1..n]$ is a number; hence, our precondition must include this requirement. Furthermore, we will not require the algorithm to verify that k is within a proper range; hence, the precondition must require that $1 \leq k \leq n$, where $k \in \mathbb{N}$ ($\mathbb{N}$ is the mathematical notation for the set of natural numbers).

We adopt the convention that if the last index is smaller than the first (e.g., $A[1..0]$), then the array has no elements.

Now let us consider how we might specify a postcondition. We need to come up with a precise definition of the kth smallest element of $A[1..n]$. Consider first the simpler case in which all elements of $A[1..n]$ are distinct. In this case, we would need to return the element $A[i]$ such that exactly k elements of $A[1..n]$ are less than or equal to $A[i]$. However, this definition of the kth smallest element might be meaningless when $A[1..n]$ contains duplicate entries. Suppose, for example, that $A[1..2]$ contains two 0s and that $k = 1$. There is then no element $A[i]$ such that exactly 1 element of $A[1..2]$ is less than or equal to $A[i]$.

To better understand the case in which elements of A might be duplicated, let us consider a specific example. Let $A[1..8] = \langle 1, 5, 6, 9, 9, 9, 9, 10 \rangle$, and let $k = 5$. An argument could be made that 10 is the kth smallest because there are exactly k *distinct* values less than or equal to 10. However,

Figure 1.1 Specification for the selection problem

Precondition: $A[1..n]$ is an array of NUMBERs, $1 \leq k \leq n$, k and n are NATs.
Postcondition: Returns the value x in $A[1..n]$ such that fewer than k elements of $A[1..n]$ are strictly less than x, and at least k elements of $A[1..n]$ are less than or equal to x. The elements of $A[1..n]$ may be permuted.

SELECT$(A[1..n], k)$

---

if we were to adopt this definition, there would be no kth smallest element for $k = 6$. Because A is sorted, it would be better to conclude that the kth smallest is $A[k] = 9$. Note that all elements strictly less than 9 are in $A[1..4]$; i.e., there are strictly fewer than k elements less than the kth smallest. Furthermore, rearranging A would not change this fact. Likewise, observe that all the elements in $A[1..5]$ are less than or equal to 9; i.e., there are at least k elements less than or equal to the kth smallest. Again, rearranging A would not change this fact.

The above example suggests that the proper definition of the kth smallest element of an array $A[1..n]$ is the value x such that

- there are fewer than k elements $A[i] < x$; and
- there are at least k elements $A[i] \leq x$.

It is possible to show, though we will not do so here, that for any array $A[1..n]$ and any positive integer $k \leq n$, there is exactly one value x satisfying both of the above conditions. We will therefore adopt this definition of the kth smallest element.

The complete specification for the selection problem is shown in Figure 1.1. To express the data types of n and the elements of A, we use NAT to denote the natural number type and NUMBER to denote the number type. Note that NAT is a *subtype* of NUMBER — every NAT is also a NUMBER. In order to place fewer constraints on the algorithm, we have included in the postcondition a statement that the elements of $A[1..n]$ may be permuted (i.e., rearranged). In order for a specification to be precise, the postcondition must state when side-effects such as this may occur. In order to keep specifications from becoming overly wordy, we will adopt the convention that no values may be changed unless the postcondition explicitly allows the change.

1.2 Algorithms

Once we have a specification, we need to produce an algorithm to implement that specification. This algorithm is a precise statement of the computational steps taken to produce the results required by the specification. An algorithm differs from a program in that it is usually not specified in a programming language. In this book, we describe algorithms using a notation that is precise enough to be implemented as a programming language, but which is designed to be read by humans.

A straightforward approach to solving the selection problem is as follows:

1. Sort the array in nondecreasing order.
2. Return the kth element.

If we already know how to sort, then we have solved our problem; otherwise, we must come up with a sorting algorithm. By using sorting to solve the selection problem, we say that we have *reduced* the selection problem to the sorting problem.

Solving a problem by reducing it to one or more simpler problems is the essence of the top-down approach to designing algorithms. One advantage to this approach is that it allows us to abstract away certain details so that we can focus on the main steps of the algorithm. In this case, we have a selection algorithm, but our algorithm requires a sorting algorithm before it can be fully implemented. This abstraction facilitates understanding of the algorithm at a high level. Specifically, if we know what is accomplished by sorting — but not necessarily how it is accomplished — then because the selection algorithm consists of very little else, we can readily understand what it does.

When we reduce the selection problem to the sorting problem, we need a specification for sorting as well. For this problem, the precondition will be that $A[1..n]$ is an array of NUMBERs, where $n \in \mathbb{N}$. Our postcondition will be that $A[1..n]$ is a permutation of its initial values such that for $1 \leq i < j \leq n$, $A[i] \leq A[j]$ — i.e., that $A[1..n]$ contains its initial values in nondecreasing order. Our selection algorithm is given in Figure 1.2. Note that SORT is only specified — its algorithm is not provided.

Let us now refine the SIMPLESELECT algorithm of Figure 1.2 by designing a sorting algorithm. We will reduce the sorting problem to the problem of inserting an element into a sorted array. In order to complete the reduction, we need to have a sorted array in which to insert. We have thus returned to our original problem. We can break this circularity, however,

Figure 1.2 An algorithm implementing the specification of Figure 1.1

SIMPLESELECT($A[1..n], k$)
  SORT($A[1..n]$)
  return $A[k]$

Precondition: $A[1..n]$ is an array of NUMBERS, n is a NAT.
Postcondition: $A[1..n]$ is a permutation of its initial values such that for $1 \leq i < j \leq n$, $A[i] \leq A[j]$.

SORT($A[1..n]$)

---

by using the top-down approach in a different way. Specifically, we reduce larger instances of sorting to smaller instances. In this application of the top-down approach, the simpler problem is actually a smaller instance of the same problem.

The algorithm is given in Figure 1.3. Though this application of the top-down approach may at first seem harder to understand, we can think about it in the same way as we did for SIMPLESELECT. If $n \leq 1$, the postcondition for SORT($A[1..n]$) (given in Figure 1.2) is clearly met. For $n > 1$, we use the specification of SORT to understand that INSERTSORT($[1..n-1]$) sorts $A[1..n-1]$. Thus, the precondition for INSERT($A[1..n]$) is satisfied. We then use the specification of INSERT($A[1..n]$) to understand that when the algorithm completes, $A[1..n]$ is sorted.

The thought process outlined above might seem mysterious because it doesn't follow the sequence of steps in an execution of the algorithm. However, it is a much more powerful way to think about algorithms.

To complete the implementations of SIMPLESELECT and INSERTSORT, we need an algorithm for INSERT. We can again use the the top-down approach to reduce an instance of INSERT to a smaller instance of the same problem. According to the precondition (see Figure 1.3), $A[1..n-1]$ must be sorted in nondecreasing order; hence, if either $n = 1$ or $A[n] \geq A[n-1]$, the postcondition is already satisfied. Otherwise, $A[n-1]$ must be the largest element in $A[1..n]$. Therefore, if we swap $A[n]$ with $A[n-1]$, $A[1..n-2]$ is sorted in nondecreasing order and $A[n]$ is the largest element in $A[1..n]$. If we then solve the smaller instance $A[1..n-1]$, we will satisfy the postcondition.

The algorithm is shown in Figure 1.4. In this book, we will assume

Figure 1.3 An algorithm implementing the specification of SORT, given in Figure 1.2

```
InsertSort(A[1..n])
  if n > 1
    InsertSort(A[1..n − 1])
    Insert(A[1..n])
```

Precondition: $A[1..n]$ is an array of NUMBERs such that n is a NAT, and for $1 \leq i < j \leq n-1$, $A[i] \leq A[j]$.
Postcondition: $A[1..n]$ is a permutation of its initial values such that for $1 \leq i < j \leq n$, $A[i] \leq A[j]$.

INSERT$(A[1..n])$

Figure 1.4 An algorithm implementing the specification of INSERT, given in Figure 1.3

```
RecursiveInsert(A[1..n])
  if n > 1 and A[n] < A[n − 1]
    A[n] ↔ A[n − 1]
    RecursiveInsert(A[1..n − 1])
```

that a logical **and** or **or** is evaluated by first evaluating its first operand, then if necessary, evaluating its second operand. Thus, in evaluating the **if** statement in RECURSIVEINSERT, if $n \leq 1$, the second operand will not be evaluated, as the value of the expression must be false. We use the notation $x \leftrightarrow y$ to swap the values of variables x and y.

1.3 Proving Algorithm Correctness

Once we have an algorithm, we would like some assurance that it meets its specification. We have argued somewhat informally that the three algorithms, SIMPLESELECT, INSERTSORT, and RECURSIVEINSERT, meet their

respective specifications; however, these arguments may not convince everyone. For example, it might not be clear that it is valid to assume that the recursive call to InsertSort in Figure 1.3 meets its specification. After all, the whole point of that discussion was to argue that InsertSort meets its specification. That argument might therefore seem circular.

In the next chapter, we will show how to prove formally that an algorithm meets its specification. These proofs will rest on solid mathematical foundations, so that a careful application of the techniques should give us confidence that a given algorithm is, indeed, correct. In particular, we will formally justify the reasoning used above — namely, that we can assume that a recursive call meets its specification, provided its input is of a smaller size than that of the original call, where the size is some natural number.

The ability to prove algorithm correctness is quite possibly the most underrated skill in the entire discipline of computing. First, knowing how to prove algorithm correctness also helps us in the design of algorithms. Specifically, once we understand the mechanics of correctness proofs, we can design the algorithm with a proof of correctness in mind. This approach makes designing correct algorithms much easier. Second, the exercise of working through a correctness proof — or even sketching such a proof — often uncovers subtle errors that would be difficult to find with testing alone. Third, this ability brings with it a capacity to understand specific algorithms on a much deeper level. Thus, the ability to prove algorithm correctness is a powerful tool for designing and understanding algorithms. Because these activities are closely related to programming, this ability greatly enhances programming abilities as well.

The proof techniques we will introduce fit nicely with the top-down approach to algorithm design. As a result, the top-down approach itself becomes an even more powerful tool for designing and understanding algorithms.

1.4 Algorithm Analysis

Once we have a correct algorithm, we need to be able to evaluate how well it does its job. This evaluation essentially boils down to an analysis of the resource usage of the algorithm, where the resources might be time or memory, for example. In Chapter 3, we will present tools for mathematically analyzing such resource usage. For now, let us discuss this resource usage less formally.

Consider the InsertSort algorithm given in Figures 1.3 and 1.4, for ex-

ample. If we were to implement this algorithm in a programming language and run it on a variety of examples, we would discover that for sufficiently large inputs — typically a few thousand elements — the program will terminate due to a stack overflow. This problem is not, strictly speaking, an error in the algorithm, but instead is a reflection of the fact that it uses the machine's runtime stack inefficiently. Using the analysis tools of Chapter 3, it is possible to show that this algorithm's stack usage is linear in the size of the array. Because the runtime stack is usually much smaller than the total memory available, we usually would like the stack usage to be bounded by a slow-growing function of the size of the input — a logarithmic function, for example. Thus, if we were to do the analysis prior to implementing the algorithm, we could uncover the inefficiency earlier in the process.

Having uncovered an inefficiency, we would like to eliminate it. The runtime stack is used when performing function calls. Specifically, each time a function call is made, information is placed onto the stack. When the function returns, this information is removed. As a result, when an algorithm uses the runtime stack inefficiently, the culprit is almost always recursion. This in not to say that recursion always uses the stack inefficiently — indeed, we will see many efficient recursive algorithms in this book. The analysis tools of Chapter 3 will help us to determine when recursion is or is not used efficiently. For now, however, we will focus on removing the recursion.

It turns out that while recursion is the most obvious way to implement an algorithm designed using the top-down approach, it is by no means the only way. One alternative is to implement the top-down solution in a bottom-up way. The top-down solution for InsertSort is to reduce a large instance to a smaller instance and an instance of Insert by first sorting the smaller instance, then applying Insert. We can apply this same idea in a bottom-up fashion by observing that the first element of any nonempty array is necessarily sorted, then extending the sorted portion of the array by applying Insert. In other words, we repeatedly apply Insert to $A[1..2], A[1..3], \ldots, A[1..n]$. This can be done using a loop.

The bottom-up implementation works well for InsertSort because the recursive call is essentially the first step of the computation. However, the recursive call in RecursiveInsert is necessarily the last step. Fortunately, there is a fairly straightforward way of removing the recursion from this type of algorithm, as well. When a reduction has the form that the solution to the simpler problem solves the original problem, we call this reduction a *transformation*. When we transform a large instance to a smaller instance of the same problem, the natural recursive algorithm is *tail recursive*, meaning

Figure 1.5 Translation of a typical tail recursive algorithm to an iterative algorithm

```
TailRecursiveAlgorithm(p_1, ..., p_k)
   if ⟨BooleanCondition⟩
      ⟨BaseCase⟩
   else
      ⟨RecursiveCaseComputation⟩
      TailRecursiveAlgorithm(q_1, ..., q_k)

IterativeAlgorithm(p_1, ..., p_k)
   while not ⟨BooleanCondition⟩
      ⟨RecursiveCaseComputation⟩
      p_1 ← q_1; ...; p_k ← q_k
   ⟨BaseCase⟩
```

that the last step is a recursive call.

Figure 1.5 illustrates how a typical tail recursive algorithm can be converted to an iterative algorithm. The loop iterates as long as the condition indicating the base case (i.e., the end of the recursion) is false. Each iteration performs all the computation in the recursive case except the recursive call. To simulate the recursive call, the formal parameters $p_1, \ldots, p_k$ are given the values of their corresponding actual parameters, $q_1, \ldots, q_k$, and the loop continues (a statement of the form "$x \leftarrow y$" assigns the value of y to the variable x). Once the condition indicating the base case is true, the loop terminates, the base case is executed, and the algorithm terminates.

Figure 1.6 shows the result of eliminating the tail recursion from RecursiveInsert. Because RecursiveInsert is structured in a slightly different way from what is shown in Figure 1.5, we could not apply this translation verbatim. The tail recursion occurs, not in the **else** part, but in the **if** part, of the **if** statement in RecursiveInsert. For this reason, we did not negate the condition when forming the **while** loop. The base case is then the empty **else** part of the **if** statement. Because there is no code in the base case, there is nothing to place after the loop. In order to avoid changing the value of n, we have copied its value to j, and used j in place of n throughout the algorithm. Furthermore, because the meaning of the state-

Figure 1.6 Iterative algorithm implementing the specification of INSERT, given in Figure 1.3

```
ITERATIVEINSERT(A[1..n])
    j ← n
    // Invariant: A[1..n] is a permutation of its original values such that
    // for 1 ≤ k < k′ ≤ n, if k′ ≠ j, then A[k] ≤ A[k′].
    while j > 1 and A[j] < A[j − 1]
        A[j] ↔ A[j − 1]; j ← j − 1
```

Figure 1.7 Insertion sort implementation of SORT, specified in Figure 1.1

```
INSERTIONSORT(A[1..n])
    // Invariant: A[1..n] is a permutation of its original values
    // such that A[1..i − 1] is in nondecreasing order.
    for i ← 1 to n
        j ← i
        // Invariant: A[1..n] is a permutation of its original values
        // such that for 1 ≤ k < k′ ≤ i, if k′ ≠ j, then A[k] ≤ A[k′].
        while j > 1 and A[j] < A[j − 1]
            A[j] ↔ A[j − 1]; j ← j − 1
```

ment "$A[1..j] \leftarrow A[1..j-1]$" is not clear, we instead simulated the recursion by the statement "$j \leftarrow j - 1$".

Prior to the loop in ITERATIVEINSERT, we have given a *loop invariant*, which is a statement that must be true at the beginning and end of each iteration of the loop. Loop invariants will be an essential part of the correctness proof techniques that we will introduce in the next chapter. For now, however, we will use them to help us to keep track of what the loop is doing.

We consider each test of a loop exit condition to mark the beginning/end of an iteration.

The resulting sorting algorithm, commonly known as insertion sort, is shown in Figure 1.7. Besides removing the two recursive calls, we have also combined the two functions into one. Also, we have started the **for** loop at 1, rather than at 2, as our earlier discussion suggested. As far as correctness

goes, there is no difference which starting point we use, as the inner loop will not iterate when $i = 1$; however, it turns out that the correctness proof, which we will present in the next chapter, is simpler if we begin the loop at 1. Furthermore, the impact on performance is minimal.

While analysis techniques can be applied to analyze stack usage, a far more common application of these techniques is to analyze running time. For example, we can use these techniques to show that while InsertionSort is not, in general, a very efficient sorting algorithm, there are important cases in which it is a very good choice. In Section 1.6, we will present a case study that demonstrates the practical utility of running time analysis.

1.5 Data Structures

One of the major factors influencing the performance of an algorithm is the way the data items it manipulates are organized. For this reason, it is essential that we include data structures in our study of algorithms. The themes that we have discussed up to this point all apply to data structures, but in a somewhat different way.

For example, a specification of a data structure must certainly include preconditions and postconditions for the operations on the structure. In addition, the specification must in some way describe the information represented by the structure, and how operations on the structure affect that information. As a result, proofs of correctness must take into account this additional detail.

While the analysis techniques of Chapter 3 apply directly to operations on data structures, it will be necessary to introduce a new technique that analyzes sequences of operations on a data structure. Furthermore, even the language we are using to express algorithms will need to be enriched in order to allow design and analysis of data structures. We will present all of these extensions in Chapter 4.

1.6 A Case Study: Maximum Subsequence Sum

We conclude this chapter by presenting a case study that illustrates the practical impact of algorithm efficiency. The inefficiencies that we will address all can be discovered using the analysis techniques of Chapter 3. This case study will also demonstrate the usefulness of the top-down approach in designing significantly more efficient algorithms.

Figure 1.8 The subsequence with maximum sum may begin and end anywhere in the array, but must be contiguous

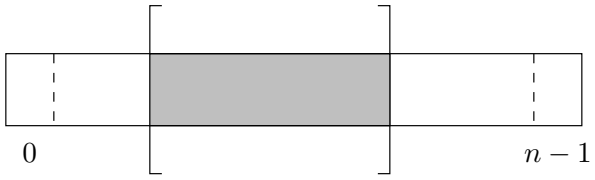


Figure 1.9 Specification for the maximum subsequence sum problem

Precondition: $A[0..n-1]$ is an array of NUMBERs, n is a NAT.
Postcondition: Returns the maximum subsequence sum of A.

MAXSUM($A[0..n-1]$)

Let us consider the problem of computing the maximum sum of any contiguous sequence in an array of numbers (see Figure 1.8). The numbers in the array may be positive, negative, or zero, and we consider the empty sequence to have a sum of 0. More formally, given an array $A[0..n-1]$ of numbers, where $n \in \mathbb{N}$, the *maximum subsequence sum* of A is defined to be

$$\max\left\{\sum_{k=i}^{j-1} A[k] \mid 0 \le i \le j \le n\right\}.$$

Note that when $i = j$, the sum has a beginning index of i and an ending index of $i-1$. By convention, we always write summations so that the index (k in this case) *increases* from its initial value (i) to its final value ($j-1$). As a result of this convention, whenever the final value is less than the initial value, the summation contains no elements. Again by convention, such an empty summation is defined to have a value of 0. Thus, in the above definition, we are including the empty sequence and assuming its sum is 0. The specification for this problem is given in Figure 1.9. Note that according to this specification, the values in $A[0..n-1]$ may not be modified.

Similar conventions hold for products, except that an empty product is assumed to have a value of 1.

Example 1.1 Suppose $A[0..5] = \langle -1, 3, -2, 7, -9, 7\rangle$. Then the subsequence $A[1..3] = \langle 3, -2, 7\rangle$ has a sum of 8. By exhaustively checking all other con-

Figure 1.10 A simple algorithm implementing the specification given in Figure 1.9

```
MaxSumIter(A[0..n − 1])
  m ← 0
  for i ← 0 to n
    for j ← i to n
      sum ← 0
      for k ← i to j − 1
        sum ← sum + A[k]
      m ← Max(m, sum)
  return m
```

tiguous subsequences, we can verify that this is, in fact, the maximum. For example, the subsequence $A[1..5]$ has a sum of 6.

Example 1.2 Suppose $A[0..3] = \langle -3, -4, -1, -5 \rangle$. Then all nonempty subsequences have negative sums. However, any empty subsequence (e.g., $A[0..-1]$) by definition has a sum of 0. The maximum subsequence sum of this array is therefore 0.

We can easily obtain an algorithm for this problem by translating the definition of a maximum subsequence sum directly into an iterative solution. The result is shown in Figure 1.10. By applying the analysis techniques of Chapter 3, it can be shown that the running time of this algorithm is proportional to n^3, where n is the size of the array.

In order to illustrate the practical ramifications of this analysis, we implemented this algorithm in the Java™ programming language and ran it on a personal computer using randomly generated data sets of size 2^k for various values of k. On a data set of size $2^{10} = 1024$, MaxSumIter required less than half a second, which seems reasonably fast. However, as the size of the array increased, the running time degraded quickly:

- $2^{11} = 2048$ elements: 2.6 seconds
- $2^{12} = 4096$ elements: 20 seconds
- $2^{13} = 8192$ elements: 2 minutes, 40 seconds

Figure 1.11 An algorithm for maximum subsequence sum (specified in Figure 1.9), optimized by removing an unnecessary loop from MaxSumIter (Figure 1.10)

```
MaxSumOpt(A[0..n − 1])
    m ← 0
    for i ← 0 to n − 1
        sum ← 0
        for k ← i to n − 1
            sum ← sum + A[k]
            m ← Max(m, sum)
    return m
```

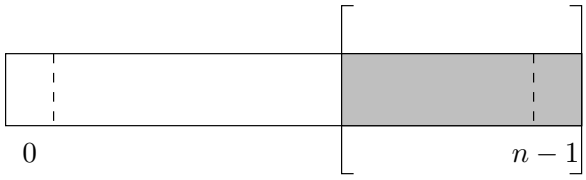
- $2^{14} = 16{,}384$ elements: over 21 minutes

Notice that as the size of the array doubles, the running time increases by roughly a factor of 8. This is not surprising if we realize that the running time should be cn^3 for some c, and $c(2n)^3 = 8cn^3$. We can therefore estimate that for a data set of size $2^{17} = 131{,}072$, the running time should be $8^3 = 512$ times as long as for 2^{14}. This running time is over a week!

If we want to solve this problem on large data sets, we clearly would like to improve the running time. A careful examination of Figure 1.10 reveals some redundant computation. Specifically, the inner loop computes sums of successively longer subsequences. Much work can be saved if we compute sums of successive sequences by simply adding the next element to the preceding sum. Furthermore, a small optimization can be made by running the outer loop to $n-1$, as the inner loop would not execute on this iteration. The result of this optimization is shown in Figure 1.11.

It turns out that this algorithm has a running time proportional to n^2, which is an improvement over n^3. To show how significant this improvement is, we again coded this algorithm and timed it for arrays of various sizes. The difference was dramatic — for an input of size 2^{17}, which we estimated would require a week for MaxSumIter to process, the running time of MaxSumOpt was only 33 seconds. However, because $c(2n)^2 = 4cn^2$, we would expect the running time to increase by a factor of 4 each time the array size is doubled. This behavior proved to be true, as a data set of size $2^{20} = 1{,}048{,}576$ required almost 40 minutes. Extrapolating this behavior,

Figure 1.12 The suffix with maximum sum may begin anywhere in the array, but must end at the end of the array



we would expect a data set of size $2^{24} = 16{,}777{,}216$ to require over a week. (MaxSumIter would require over 43,000 years to process a data set of this size.)

While MaxSumOpt gives a dramatic speedup over MaxSumIter, we would like further improvement if we wish to solve very large problem instances. Note that neither MaxSumIter nor MaxSumOpt was designed using the top-down approach. Let us therefore consider how we might solve the problem in a top-down way. For ease of presentation let us refer to the maximum subsequence sum of $A[0..n-1]$ as s_n. Suppose we can obtain s_{n-1} (i.e., the maximum subsequence sum of $A[0..n-2]$) for $n > 0$. Then in order to compute the overall maximum subsequence sum we need the maximum of s_{n-1} and all of the sums of subsequences $A[i..n-1]$ for $0 \leq i \leq n$. Thus, we need to solve another problem, that of finding the *maximum suffix sum* (see Figure 1.12), which we define to be

$$\max\left\{\sum_{k=i}^{n-1} A[k] \mid 0 \leq i \leq n\right\}.$$

In other words, the maximum suffix sum is the maximum sum that we can obtain by starting at any index i, where $0 \leq i \leq n$, and adding together all elements from index i up to index $n-1$. (Note that by taking $i = n$, we include the empty sequence in this maximum.) We then have a top-down solution for computing the maximum subsequence sum:

$$s_n = \begin{cases} 0 & \text{if } n = 0 \\ \max(s_{n-1}, t_n) & \text{if } n > 0, \end{cases} \tag{1.1}$$

where t_n is the maximum suffix sum of $A[0..n-1]$.

Figure 1.13 A top-down solution for maximum subsequence sum, specified in Figure 1.9

```
MaxSumTD(A[0..n − 1])
  if n = 0
    return 0
  else
    return Max(MaxSumTD(A[0..n−2]), MaxSuffixTD(A[0..n−1]))
```

Precondition: $A[0..n-1]$ is an array of Numbers, n is a Nat.
Postcondition: Returns the maximum suffix sum of A.

```
MaxSuffixTD(A[0..n − 1])
  if n = 0
    return 0
  else
    return Max(0, A[n − 1] + MaxSuffixTD(A[0..n − 2]))
```

Let us consider how to compute the maximum suffix sum t_n using the top-down approach. Observe that every suffix of $A[0..n-1]$ — except the empty suffix — ends with $A[n-1]$. If we remove $A[n-1]$ from all of these suffixes, we obtain all of the suffixes of $A[0..n-2]$. Thus, $t_{n-1}+A[n-1] = t_n$ unless $t_n = 0$. We therefore have

$$t_n = \max(0, A[n-1] + t_{n-1}). \tag{1.2}$$

Using (1.1) and (1.2), we obtain the recursive solution given in Figure 1.13. Note that we have combined the algorithm for MaxSuffixTD with its specification.

Unfortunately, an analysis of this algorithm shows that it also has a running time proportional to n^2. What is worse, however, is that an analysis of its stack usage reveals that it is linear in n. Indeed, the program implementing this algorithm threw a StackOverflowError on an input of size 2^{12}.

While these results are disappointing, we at least have some techniques for improving the stack usage. Note that in both MaxSumTD and MaxSuffixTD, the recursive calls don't depend on any of the rest of the computation; hence, we should be able to implement both algorithms in a bottom-

Figure 1.14 A bottom-up calculation of the maximum subsequence sum, specified in Figure 1.9

```
MaxSumBU(A[0..n − 1])
    m ← 0; msuf ← 0
    // Invariant: m is the maximum subsequence sum of A[0..i − 1],
    // msuf is the maximum suffix sum for A[0..i − 1]
    for i ← 0 to n − 1
        msuf ← Max(0, msuf + A[i])
        m ← Max(m, msuf)
    return m
```

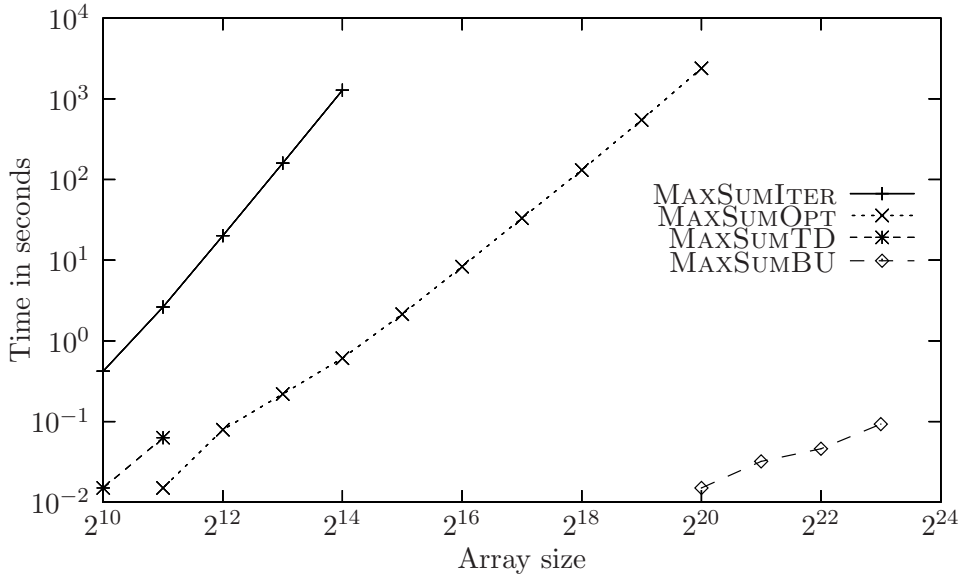
up fashion in order to remove the recursion. Furthermore, we can simplify the resulting code with the realization that once t_i is computed (using (1.2)), we can immediately compute s_i using (1.1). Thus, we can compute both values within a single loop. The result is shown in Figure 1.14.

Because this algorithm uses no recursion, its stack usage is fixed (i.e., it does not grow as n increases). Furthermore, an analysis of its running time reveals that it runs in a time linear in n. The program implementing this algorithm bears out that this is a significant improvement — this program used less than a tenth of a second on an array of size $2^{23} = 8{,}388{,}608$. Furthermore, using the fact that the running time is linear, we estimate that it would require less than 12 seconds to process an array of size $2^{30} = 1{,}073{,}741{,}824$. (Unfortunately, such an array would require more memory than was available on the PC we used.) By comparison, we estimate that on an array of this size, MaxSumOpt would require nearly 80 years, and that MaxSumIter would require over 11 billion years!

The running times of the Java™ implementations of all four algorithms are plotted on the graph in Figure 1.15. The code used to generate these results can be found on the textbook's web page.

Note that the performance improvement of MaxSumBU over either MaxSumIter or MaxSumOpt is better than what we can expect to obtain simply by using faster hardware. Specifically, to achieve the running time we measured for MaxSumBU with MaxSumOpt would require hardware over 100,000 times faster for an array of size 2^{20}. Even then, we would be able to process much larger inputs with MaxSumBU. MaxSumBU also has the advantage of being simpler than the other algorithms. Furthermore,

Figure 1.15 Experimental comparison of maximum subsequence sum algorithms



because it makes only one pass through the input, it does not require that the data be stored in an array. Rather, it can simply process the data as it reads each element. As a result, it can be used for very large data sets that might not fit into main memory.

Thus, we can see the importance of efficiency in the design of algorithms. Furthermore, we don't have to code the algorithm and test it to see how efficient it is. Instead, we can get a fairly good idea of its efficiency by analyzing it using the techniques presented in Chapter 3. Finally, understanding these analysis techniques will help us to know where to look to improve algorithm efficiency.

1.7 Summary

The study of algorithms encompasses several facets. First, before an algorithm or data structure can be considered, a specification of the requirements must be made. Having a specification, we can then design the algorithm or data structure with a proof of correctness in mind. Once we have convinced

ourselves that our solution is correct, we can then apply mathematical techniques to analyze its resource usage. Such an analysis gives us insight into how useful our solution might be, including cases in which it may or may not be useful. This analysis may also point to shortcomings upon which we might try to improve.

The top-down approach is a useful framework for designing correct, efficient algorithms. Furthermore, algorithms presented in a top-down fashion can be more easily understood. Together with the top-down approach, techniques such as bottom-up implementation and elimination of tail recursion — along with others that we will present later — give us a rich collection of tools for algorithm design. We can think of these techniques as algorithmic *design patterns*, as we use each of them in the design of a wide variety of algorithms.

In Chapters 2 and 3, we will provide the foundations for proving algorithm correctness and analyzing algorithms, respectively. In Part II, we will examine several of the most commonly-used data structures, including those that are frequently used by efficient algorithms. In Part III, we examine the most common approaches to algorithm design. In Part IV, we will study several specific algorithms to which many other problems can be reduced. Finally, in Part V, we will consider a class of problems believed to be computationally intractable and introduce some techniques for coping with them.

1.8 Exercises

Exercise 1.1 We wish to design an algorithm that takes an array $A[0..n-1]$ of numbers in nondecreasing order and a number x, and returns the location of the first occurrence of x in $A[0..n-1]$, or the location at which x could be inserted without violating the ordering if x does not occur in the array. Give a formal specification for this problem. The algorithm shown in Figure 1.16 should meet your specification.

Exercise 1.2 Give an iterative algorithm that results from removing the tail recursion from the algorithm shown in Figure 1.16. Your algorithm should meet the specification described in Exercise 1.1.

Exercise 1.3 Figure 1.17 gives a recursive algorithm for computing the dot product of two vectors, represented as arrays. Give a bottom-up implementation of this algorithm.

Figure 1.16 An algorithm satisfying the specification described in Exercise 1.1

```
FIND(A[0..n − 1], x)
  if n = 0 or A[n − 1] < x
    return n
  else
    return FIND(A[0..n − 2], x)
```

Figure 1.17 Algorithm for Exercise 1.3

Precondition: $A[1..n]$ and $B[1..n]$ are ARRAYs of NUMBERS, and n is a NAT.
Postcondition: Returns

$$\sum_{i=1}^{n} A[i]B[i].$$

```
DOTPRODUCT(A[1..n], B[1..n])
  if n = 0
    return 0
  else
    return DOTPRODUCT(A[1..n − 1], B[1..n − 1]) + A[n]B[n]
```

Exercise 1.4 Figure 1.18 gives a specification for COPY.

a. Show how to reduce COPY to a smaller instance of itself by giving a recursive algorithm. You may assume that A and B are distinct, non-overlapping arrays. Your algorithm should contain exactly one recursive call and no loops. *Corrected 1/27/10.*

b. Convert your algorithm from part a to an iterative algorithm by either removing tail recursion or implementing it bottom-up, whichever is more appropriate. Explain how you did your conversion.

c. The specification of COPY does not prohibit the two arrays from sharing elements, for example, COPY($A[1..n-1], A[2..n]$). Modify your

Figure 1.18 Specification of COPY

Precondition: n is a NAT.
Postcondition: For $1 \leq i \leq n$, $B[i]$ is modified to equal $A[i]$.

COPY$(A[1..n], B[1..n])$

Figure 1.19 Specification for FINDMAX

Precondition: $A[0..n-1]$ is an array of numbers, and n is a positive NAT.
Postcondition: Returns i such that $0 \leq i < n$ and for any j, $0 \leq j < n$, $A[i] \geq A[j]$.

FINDMAX$(A[0..n-1])$

algorithm from part a to handle any two arrays of the same size. Specifically, you cannot assume that the recursive call does not change $A[1..n]$. Your algorithm should contain exactly one recursive call and no loops.

Exercise 1.5 A *palindrome* is a sequence of characters that is the same when read from left to right as when read from right to left. We wish to design an algorithm that recognizes whether an array of characters is a palindrome.

a. Give a formal specification for this problem. Use CHAR to denote the data type for a character.

b. Using the top-down approach, give an algorithm to solve this problem. Your algorithm should contain a single recursive call.

c. Give an iterative version of your algorithm from part b by either implementing it bottom-up or eliminating tail recursion, whichever is appropriate.

Exercise 1.6 FINDMAX is specified in Figure 1.19.

a. Using the top-down approach, give an algorithm for FINDMAX. Note that according to the specification, your algorithm may not change

Figure 1.20 Specifications of AppendToAll and SizeOf

Precondition: $A[1..n]$ is an array of arrays and n is a Nat.
Postcondition: For $1 \leq i \leq n$, modifies $A[i]$ to contain an array whose size is 1 larger than its original size, and whose elements are the same as its original elements, with x added as its last element (i.e., x is appended to the end of each $A[i]$).

AppendToAll($A[1..n]$, x)

Precondition: A is an array.
Postcondition: Returns the number of elements in A.

SizeOf(A)

the values in A. Your algorithm should contain exactly one recursive call.

b. Give an iterative version of your algorithm from part a by either implementing it bottom-up or eliminating tail recursion, whichever is appropriate.

* c. Show how to reduce the sorting problem to FindMax (as specified in part a) and a smaller instance of sorting. Use the technique of transformation, and remove the resulting tail recursion, so that your algorithm is iterative.

*** Exercise 1.7** AppendToAll and SizeOf are specified in Figure 1.20.

a. Show how to reduce AppendToAll to Copy (specified in Figure 1.18), SizeOf, and a smaller instance of itself, by giving a recursive algorithm. Your algorithm should contain no loops and a single recursive call.

b. Convert your algorithm from part a to an iterative algorithm by either removing tail recursion or implementing it bottom-up, whichever is more appropriate. You do not need to provide algorithms for either Copy or SizeOf.

Figure 1.21 Specification for AllSubsets

Precondition: $A[1..n]$ is an Array of distinct elements, and n is a Nat.
Postcondition: Returns an Array $P[1..2^n]$ such that for $1 \leq i \leq 2^n$, $P[i]$ is an Array containing exactly one occurrence of each element of some subset of the elements in $A[1..n]$, where each subset of $A[1..n]$ is represented exactly once in $P[1..2^n]$.

AllSubsets($A[1..n]$)

*** Exercise 1.8** AllSubsets is specified in Figure 1.21.

a. Show how to reduce to AllSubsets to Copy (specified in Figure 1.18), AppendToAll (specified in Figure 1.20), SizeOf (specified in Figure 1.20), and a smaller instance of itself. Note that according to the specification, your algorithm may not change $A[1..n]$. Your algorithm should contain exactly one recursive call and no loops. (When calling Copy, your first array index does not need to be 1.)

b. Give an iterative version of your algorithm from part a by either implementing it bottom-up or eliminating tail recursion, whichever is appropriate. You do not need to provide algorithms for Copy, AppendToAll, or SizeOf.

1.9 Chapter Notes

The elements of top-down software design were introduced by Dijkstra [28] and Wirth [114] in the late 1960s and early 1970s. As software systems grew, however, these notions were found to be insufficient to cope with the sheer size of large projects. As a result, they were eventually superseded by *object-oriented design and programming*. The study of algorithms, however, does not focus on large software systems, but on small components. Consequently, a top-down approach provides an ideal framework for designing and understanding algorithms.

The maximum subsequence sum problem, as well as the algorithms MaxSumIter, MaxSumOpt, and MaxSumBU, was introduced by Bentley [12]. The sorting algorithm suggested by Exercise 1.6 is *selection sort*.

Java is a registered trademark of Sun Microsystems, Inc.