Chapter 2

Proving Algorithm Correctness

In Chapter 1, we specified several problems and presented various algorithms for solving these problems. For each algorithm, we argued somewhat informally that it met its specification. In this chapter, we introduce a mathematical foundation for more rigorous proofs of algorithm correctness. Such proofs not only give us more confidence in the correctness of our algorithms, but also help us to find subtle errors. Furthermore, some of the tools we introduce here are also used in the context of analyzing algorithm performance.

2.1 The Basics

First consider the algorithm SimpleSelect, shown in Figure 1.2 on page 6. This algorithm is simple enough that ordinarily we would not bother to give a formal proof of its correctness; however, such a proof serves to illustrate the basic approach. Recall that in order for an algorithm to meet its specification, it must be the case that whenever the precondition is satisfied initially, the postcondition is also satisfied when the algorithm finishes.

Theorem 2.1 SimpleSelect meets the specification given in Figure 1.1.

Proof: First, we assume that the precondition for SimpleSelect($A[1..n]$) is satisfied initially; i.e., we assume that initially $A[1..n]$ is an array of numbers, $1 \leq k \leq n$, and both k and n are natural numbers. This assumption immediately implies the precondition for Sort($A[1..n]$), namely, that

$A[1..n]$ is an array of numbers, and that n is a natural number. In order to talk about both the initial and final values of A without confusion, let us represent the final value of A by A' and use A only to denote its initial value. Because Sort permutes A (we know this from its postcondition), A' contains the same collection of values as does A.

Suppose $A'[i] < A'[k]$ for some i, $1 \leq i \leq n$. Then $i < k$, for if $k < i$, $A'[k] > A'[i]$ violates the postcondition of Sort. Hence, there are fewer than k elements of A', and hence of A, with value less than $A'[k]$.

Now suppose $1 \leq i \leq k$. From the postcondition of Sort, $A'[i] \leq A'[k]$. Hence, there are at least k elements of A with value less than or equal to $A'[k]$. The returned value $A'[k]$ therefore satisfies the postcondition of Select. □

Because of what it means for an algorithm to meet its specification, any proof of correctness will begin by assuming the precondition. The goal of the proof is then to prove that the postcondition is satisfied when the algorithm finishes. In order to reach this goal, we reason about the effect of each statement in turn. If a statement is a call to another algorithm, we use that algorithm's specification. Specifically, we must first make sure its precondition is satisfied (for otherwise, we cannot know what the algorithm does), then we conclude that its postcondition is satisfied when it finishes. We can then use this postcondition to continue our reasoning.

If the precondition is defined to be true, *we don't need to assume it, because we know that* true *is true.*

The proof of Theorem 2.1 illustrates a common difficulty with correctness proofs. In algorithms, variables typically change their values as the algorithm progresses. However, in proofs, a variable must maintain a single value in order to maintain consistent reasoning. In order to avoid confusion, we introduce different names for the value of the same variable at different points in the algorithm. It is customary to use the variable name from the algorithm to denote the initial value, and to add a "prime" ($'$) to denote its final value. If intermediate value must be considered, "double-primes" ($''$), superscripts, or subscripts can be used.

Although a variable should not change its value over the course of a proof, we do sometimes use the same variable with different values when proving different facts. For example, the variable i in the proof of Theorem 2.1 can have different values in the second and third paragraphs, respectively. It might be helpful to think of such usage as multiple local variables, each having the same name. Within each scope, the value of the variable does not change.

The proof of Theorem 2.1 is straightforward because the algorithm is straight-line — the flow of control proceeds sequentially from the first statement in the algorithm to the last. The addition of **if** statements complicates matters only slightly — we must simply apply case analysis. However, the addition of recursion and/or loops requires some additional machinery, which we will develop over the remainder of this chapter.

2.2 Handling Recursion

Let us now consider InsertSort from Figure 1.3 on page 7. To handle the **if** statement, we can consider two cases, depending on whether $n > 1$. In case $n > 1$, however, there is a recursive call to InsertSort. As is suggested in Section 1.2, we might simply use the specification of Sort just like we would for a call to any other algorithm. However, this type of reasoning is logically suspect — we are assuming the correctness of InsertSort in order to prove the correctness of InsertSort.

In order to break this circularity, we use the *principle of mathematical induction*. We will be using this principle throughout this text, so we will now take the time to present and prove it. The version we present is a technique for proving properties of the natural numbers. The property that we wish to prove in this case is that InsertSort($A[1..n]$) satisfies its specification for every natural number n. Let us now formally define what we mean by a property of the natural numbers, so that our statement of the induction principle will be clear.

Definition 2.2 A *property of the natural numbers* is a mapping

$$P : \mathbb{N} \rightarrow \{\mathsf{true}, \mathsf{false}\};$$

i.e., for a natural number n, $P(n)$ is either true or false.

Theorem 2.3 (Mathematical Induction Principle) Let $P(n)$ be a property of the natural numbers. Suppose that every natural number n satisfies the following Induction Property:

- whenever $P(i)$ is true for every natural number $i < n$, $P(n)$ is also true.

Then $P(n)$ is true for every natural number n.

Before we prove this theorem, let us consider how we can use it. Suppose we wish to prove a property $P(n)$ for every natural number n. Theorem 2.3 tells us that it is sufficient to prove the Induction Property. We can break this proof into two parts:

1. The *induction hypothesis*. We assume that for some arbitrary $n \in \mathbb{N}$, $P(i)$ is true for every natural number $i < n$.
2. The *induction step*. Using the induction hypothesis, we prove that $P(n)$ is true.

Some readers may be familiar with another version of induction consisting of three steps:

1. The *base case*. $P(0)$ is shown to be true.
2. The *induction hypothesis*. $P(n)$ is assumed to be true for some arbitrary natural number n.
3. The *induction step*. Using the induction hypothesis, $P(n+1)$ is proved.

Though this technique is also valid, the version given by Theorem 2.3 is more appropriate for the study of algorithms. To see why, consider how we are using the top-down approach. We associate with each input a natural number that in some way describes the size of the input. We then recursively apply the algorithm to one or more inputs of strictly smaller size. Theorem 2.3 tells us that in order to prove that this algorithm is correct for inputs of all sizes, we may assume that for arbitrary n, the algorithm is correct for all inputs of size less than n. Thus, we may reason about a recursive algorithm in the same way we reason about an algorithm that calls other algorithms, provided the size of the parameters is smaller for the recursive calls.

Now let us turn to the proof of Theorem 2.3.

Proof of Theorem 2.3: Suppose every natural number n satisfies the Induction Property given in the statement of the theorem. In order to derive a contradiction, assume that for some $n \in \mathbb{N}$, $P(n) =$ false. Specifically, let n be the smallest such value. Then for every $i < n$, $P(i) =$ true. By the Induction Property, $P(n) =$ true — a contradiction. We conclude that our assumption was invalid, so that $P(n) =$ true for every natural number n. □

Before we illustrate the use of this principle by proving the correctness of InsertSort, let us briefly discuss an element of style regarding induction proofs. One of the distinguishing features of this particular induction principle is the absence of a base case. However, in most proofs, there are special cases in which the induction hypothesis is not used. These are typically the smallest cases, where the induction hypothesis gives us little or no information. It would be a bit of a misnomer to use the term, "induction step", for such cases. It is stylistically better to separate these cases into one or more base cases. Hence, even though a base case is not required, we usually include one (or more).

Now we will illustrate the principle of induction by proving the correctness of InsertSort.

Theorem 2.4 InsertSort, given in Figure 1.3, satisfies the specification of Sort, given in Figure 1.2.

Proof: By induction on n.

Base: $n \leq 1$. In this case the algorithm does nothing, but its postcondition is vacuously satisfied (i.e., there are no i, j such that $1 \leq i < j \leq n$).

Induction Hypothesis: Assume that for some $n > 1$, for every $k < n$, InsertSort($A[1..k]$) satisfies its specification.

Induction Step: We first assume that initially, the precondition for InsertSort($A[1..n]$) is satisfied. Then the precondition for InsertSort($A[1..n-1]$) is also initially satisfied. By the Induction Hypothesis, we conclude that InsertSort($A[1..n-1]$) satisfies its specification; hence, its postcondition holds when it finishes. Let A'' denote the value of A after InsertSort($A[1..n-1]$) finishes. Then $A''[1..n-1]$ is a permutation of $A[1..n-1]$ in nondecreasing order, and $A''[n] = A[n]$. Thus, A'' satisfies the precondition of Insert. Let A' denote the value of A after Insert($A[1..n]$) is called. By the postcondition of Insert, $A'[1..n]$ is a permutation of $A[1..n]$ in nondecreasing order. InsertSort therefore satisfies its specification. $\square$

Because the algorithm contains an **if** statement, the proof requires a case analysis. The two cases are handled in the Base and the Induction Step, respectively. Note that the way we prove the Induction Step is very similar to how we proved Theorem 2.1. The only difference is that we have the Induction Hypothesis available to allow us to reason about the recursive call.

2.3 Handling Iteration

Let us now consider MaxSumBU, shown in Figure 1.14 on page 18. This algorithm contains a **for** loop. As we did with recursion, we would like to be able to apply straight-line reasoning techniques. In Chapter 1, we used invariants in order to focus on a single loop iteration. Because we already know how to handle code without loops, focusing on a single iteration allows us to apply techniques we have already developed. Loop invariants therefore give us the power to reason formally about loops.

Suppose we wish to show that property P holds upon completion of a given loop. Suppose further that we can show each of the following:

1. **Initialization:** The invariant holds prior to the first loop iteration.
2. **Maintenance:** If the invariant holds at the beginning of an arbitrary loop iteration, then it must also hold at the end of that iteration.
3. **Termination:** The loop always terminates.
4. **Correctness:** Whenever the loop invariant and the loop exit condition both hold, then P must hold.

It is not hard to show by induction on the number of loop iterations that if both the initialization and maintenance steps hold, then the invariant holds at the end of each iteration. If the loop always terminates, then the invariant and the loop exit condition will both hold when this happens. The correctness step then guarantees that property P will hold after the loop completes. The above four steps are therefore sufficient to prove that P holds upon completion of the loop.

We now illustrate this proof technique by showing correctness of MaxSumBU. In our informal justification of this algorithm, we used equations (1.1) and (1.2); however, because we did not prove these equations, our proof of correctness should not use them.

Theorem 2.5 MaxSumBU satisfies its specification.

Proof: Suppose the precondition holds initially. We will show that when the loop finishes, m contains the maximum subsequence sum of $A[0..n-1]$, so that the postcondition is satisfied. Note that the loop invariant states that m is the maximum subsequence sum of $A[0..i-1]$.

Initialization: Before the loop iterates the first time, i has a value of 0. The maximum subsequence sum of $A[0..-1]$ is defined to be 0. m is initially assigned this value. Likewise, the maximum suffix sum of $A[0..-1]$ is defined to be 0, and *msuf* is initially assigned this value. Therefore, the invariant initially holds.

Maintenance: Suppose the invariant holds at the beginning of some iteration. We first observe that because the iteration occurs, $0 \leq i \leq n-1$; hence, $A[i]$ is a valid array location. Let msuf', m' and i' denote the values of msuf, m and i, respectively, at the end of this iteration. Then

- $\mathit{msuf}' = \max(0, \mathit{msuf} + A[i])$;
- $m' = \max(m, \mathit{msuf}')$; and
- $i' = i + 1$.

For $0 \leq j \leq n$, let s_j denote the maximum subsequence sum of $A[0..j-1]$, and let t_j denote the maximum suffix sum of $A[0..j-1]$. From the invariant, $\mathit{msuf} = t_i$, and $m = s_i$. We need to show that $\mathit{msuf}' = t_{i'}$ and $m' = s_{i'}$. *Corrected 1/27/12.*

Using the definition of a maximum suffix sum, we have

$$\begin{aligned}
\mathit{msuf}' &= \max(0, \mathit{msuf} + A[i]) \\
&= \max(0, t_i + A[i]) \\
&= \max\left(0, \max\left\{\sum_{k=l}^{i-1} A[k] \mid 0 \leq l \leq i\right\} + A[i]\right) \\
&= \max\left\{0, A[i] + \sum_{k=l}^{i-1} A[k] \mid 0 \leq l \leq i\right\} \\
&= \max\left\{0, \sum_{k=l}^{i} A[k] \mid 0 \leq l \leq i\right\} \\
&= \max\left\{\sum_{k=l}^{i'-1} A[k] \mid 0 \leq l \leq i'\right\} \\
&= t_{i'}.
\end{aligned}$$

Likewise, using the definitions of a maximum subsequence sum and a maximum suffix sum, we have

$$\begin{aligned} m' &= \max(m, msuf') \\ &= \max(s_i, t_{i'}) \\ &= \max\left(\max\left\{\sum_{k=l}^{h-1} A[k] \mid 0 \leq l \leq h \leq i\right\}, \max\left\{\sum_{k=l}^{i'-1} A[k] \mid 0 \leq l \leq i'\right\}\right) \\ &= \max\left\{\sum_{k=l}^{h-1} A[k] \mid 0 \leq l \leq h \leq i'\right\} \\ &= s_{i'}. \end{aligned}$$

Therefore, the invariant holds at the end of the iteration.

Termination: Because the loop is a **for** loop, it clearly terminates.

*In this textbook, a **for** loop always contains a single index variable, which either is incremented by a fixed positive amount each iteration until it exceeds a fixed value or is decremented by a fixed positive amount each iteration until it is less than a fixed value. The index cannot be changed otherwise. Such loops will always terminate. Correctness case for Theorem 2.5 corrected 1/27/12.*

Correctness: The loop exits when $i = n$. Thus, from the invariant, m is the maximum subsequence sum of $A[0..n-1]$ when the loop terminates. □

As can be seen from the above proof, initialization and maintenance can be shown using techniques we have already developed. Furthermore, the correctness step is simply logical inference. In the case of Theorem 2.5, termination is trivial, because **for** loops always terminate. Note, however, that in order for such a proof to be completed, it is essential that a proper loop invariant be chosen. Specifically, the invariant must be chosen so that:

- it is true every time the loop condition is tested;
- it is possible to prove that if it is true at the beginning of an arbitrary iteration, it must also be true at the end of that iteration; and
- when coupled with the loop exit condition, it is strong enough to prove the desired correctness property.

Thus, if we choose an invariant that is too strong, it may not be true each time the loop condition is tested. On the other hand, if we choose an invariant that is too weak, we may not be able to prove the correctness property. Furthermore, even if the invariant is true on each iteration and is strong enough to prove the correctness property, it may still be impossible to prove the maintenance step. We will discuss this issue in more detail shortly.

Figure 2.1 A loop whose termination for all n is unknown

```
while $n > 1$
    if $n \bmod 2 = 0$
        $n \leftarrow n/2$
    else
        $n \leftarrow 3n + 1$
```

For **while** loops, the proof of termination is usually nontrivial and in some cases quite difficult. An example that is not too difficult is IterativeInsert in Figure 1.6 (page 11). To prove termination of this loop, we need to show that each iteration makes progress toward satisfying the loop exit condition. The exit condition for this loop is that $j \leq 1$ or $A[j] \geq A[j-1]$. Usually, the way in which a loop will make progress toward meeting such a condition is that each iteration will decrease the difference between the two sides of an inequality. In this case, j is decreased by each iteration, and therefore becomes closer to 1. (The other inequality in the exit condition is not needed to prove termination — if it becomes true, the loop just terminates that much sooner.) We can therefore prove the following theorem.

Theorem 2.6 The **while** loop in IterativeInsert always terminates.

Proof: We first observe that each iteration of the **while** loop decreases j by 1. Thus, if the loop continues to iterate, eventually $j \leq 1$, and the loop then terminates. □

Proving termination of a **while** loop can be much more difficult than the proof of the above theorem. For example, consider the **while** loop shown in Figure 2.1. The mod operation, when applied to positive integers, gives the remainder obtained when an integer division is performed; thus, the **if** statement tests whether n is even. Though many people have studied this computation over a number of years, as of this writing, it is unknown whether this loop terminates for all initial integer values of n. This question is known as the *Collatz problem*.

On the other hand, when algorithms are designed using the top-down approach, proving termination of any resulting **while** loops becomes much easier. Even if an examination of the **while** loop condition does not help us to find a proof, we should be able to derive a proof from the reduction we used to solve the problem. Specifically, a loop results from the reduction of larger instances of a problem to smaller instances of the same problem, where the size of the instance is a natural number. We should therefore be able to prove that the expression denoting the size of the instance is a natural number that is decreased by every iteration. Termination will then follow.

For example, consider again the algorithm IterativeInsert. In the design of this algorithm (see Section 1.4), we reduced larger instances to smaller instances, where the size of an instance was n, the number of array elements. In removing the tail recursion from the algorithm, we replaced n by j. j should therefore decrease as the size decreases. We therefore base our correctness proof on this fact.

Let us now consider an algorithm with nested loops, such as Insertion-Sort, shown in Figure 1.7 on page 11. When loops are nested, we apply the same technique to each loop as we encounter it. Specifically, in order to prove maintenance for the outer loop, we need to prove that the inner loop satisfies some correctness property, which should in turn be sufficient to complete the proof of maintenance for the outer loop. Thus, nested within the maintenance step of the outer loop is a complete proof (i.e., initialization, maintenance, termination and correctness) for the inner loop.

When we prove initialization for the inner loop, we are not simply reasoning about the code leading to the first execution of that loop. Rather, we are reasoning about the code that initializes the loop on any iteration of the outer loop. For this reason, we cannot consider the initialization code for the outer loop when proving the initialization step for the inner loop. Instead, because the proof for the inner loop is actually a part of the maintenance proof for the outer loop, we can use any facts available for use in the proof of maintenance for the outer loop. Specifically, we can use the assumption that the invariant holds at the beginning of the outer loop iteration, and we can reason about any code executed prior to the inner loop during this iteration. We must then show that the invariant of the inner loop is satisfied upon executing this code.

We will now illustrate this technique by giving a complete proof that InsertionSort meets its specification.

Theorem 2.7 InsertionSort meets its specification.

Proof: We must show that when the **for** loop finishes, $A[1..n]$ is a permutation of its original values in nondecreasing order.

Initialization: (Outer loop) When the loop begins, $i = 1$ and the contents of $A[1..n]$ have not been changed. Because $A[1..i-1]$ is an empty array, it is in nondecreasing order.

Maintenance: (Outer loop) Suppose the invariant holds at the beginning of some iteration. Let $A'[1..n]$ denote the contents of A at the end of the iteration, and let i' denote the value of i at the end of the iteration. Then $i' = i + 1$. We must show that the **while** loop satisfies the correctness property that $A'[1..n]$ is a permutation of the original values of $A[1..n]$, and that $A'[1..i'-1] = A'[1..i]$ is in nondecreasing order.

Initialization: (Inner loop) Because $A[1..n]$ has not been changed since the beginning of the current iteration of the outer loop, from the outer loop invariant, $A[1..n]$ is a permutation of its original values. From the outer loop invariant, $A[1..i-1]$ is in nondecreasing order; hence, because $j = i$, we have for $1 \leq k < k' \leq i$, where $k' \neq j$, $A[k] \leq A[k']$.

Maintenance: (Inner loop) Suppose the invariant holds at the beginning of some iteration. Let $A'[1..n]$ denote the contents of $A[1..n]$ following the iteration, and let j' denote the value of j following the iteration. Hence,

i. $A'[j] = A[j-1]$;

ii. $A'[j-1] = A[j]$;

iii. $A'[k] = A[k]$ for $1 \leq k \leq n$, $k \neq j$, and $k \neq j-1$; and

iv. $j' = j - 1$.

Thus, $A'[1..n]$ is a permutation of $A[1..n]$. From the invariant, $A'[1..n]$ is therefore a permutation of the original values of $A[1..n]$. Suppose $1 \leq k < k' \leq i$, where $k' \neq j' = j - 1$. We must show that $A'[k] \leq A'[k']$. We consider three cases.

Case 1: $k' < j - 1$. Then $A'[k] = A[k]$ and $A'[k'] = A[k']$. From the invariant, $A[k] \leq A[k']$; hence, $A'[k] \leq A'[k']$.

Case 2: $k' = j$. Then $A'[k'] = A[j-1]$. If $k = j-1$, then $A'[k] = A[j]$, and from the **while** loop condition, $A[j] < A[j-1]$. Otherwise, $k < j-1$ and

$A'[k] = A[k]$; hence, from the invariant, $A[k] \leq A[j-1]$. In either case, we conclude that $A'[k] \leq A'[k']$.

Case 3: $k' > j$. Then $A'[k'] = A[k']$, and $A'[k] = A[l]$, where l is either k, j, or $j-1$. In each of these cases, $l < k'$; hence, from the invariant, $A[l] \leq A[k']$. Thus, $A'[k] \leq A'[k']$.

Termination: (Inner loop) Each iteration decreases the value of j by 1; hence, if the loop keeps iterating, j must eventually be no greater than 1. At this point, the loop will terminate.

Correctness: (Inner loop) Let $A'[1..n]$ denote the contents of $A[1..n]$ when the **while** loop terminates, and let i and j denote their values at this point. From the invariant, $A'[1..n]$ is a permutation of its original values. We must show that $A'[1..i]$ is in nondecreasing order. Let $1 \leq k < k' \leq i$. We consider two cases.

Case 1: $k' = j$. Then $j > 1$. From the loop exit condition, it follows that $A'[j-1] \leq A'[j] = A'[k']$. From the invariant, if $k \neq j-1$, then $A'[k] \leq A'[j-1]$; hence, regardless of whether $k = j-1$, $A'[k] \leq A'[k']$.

Case 2: $k' \neq j$. Then from the invariant, $A'[k] \leq A'[k']$.

This completes the proof for the inner loop, and hence the proof of maintenance for the outer loop.

Termination: (Outer loop) Because the loop is a **for** loop, it must terminate.

Correctness: (Outer loop) Let $A'[1..n]$ denote its final contents. From the invariant, $A'[1..n]$ is a permutation of its original values. From the loop exit condition ($i = n+1$) and the invariant, $A'[1..n]$ is in nondecreasing order. Therefore, the postcondition is satisfied. □

Now that we have shown that InsertionSort is correct, let us consider how we might have found the invariant for the inner loop. The inner loop implements a transformation of larger instances of the insertion problem, specified in Figure 1.3 on page 7, to smaller instances of the same problem. The loop invariant should therefore be related to the precondition for Insert.

The current instance of the insertion problem is represented by $A[1..j]$. Therefore, a first choice for an invariant might be that $A[1..j]$ is a permutation of its original values, and that $A[1..j-1]$ is sorted. However, this invariant is not strong enough to prove the correctness property. To see why, observe that the loop exit condition allows the loop to terminate when $j = 1$. In this case, $A[1..j]$ has only one element, $A[1..j-1]$ is empty, and the invariant tells us almost nothing.

Clearly, we need to include in our invariant that $A[1..n]$ is a permutation of its initial values. Furthermore, we need more information about what has already been sorted. Looking at the invariant for the outer loop, we might try saying that both $A[1..j-1]$ and $A[j..i]$ are in nondecreasing order. By coupling this invariant with the loop exit condition (i.e, either $j = 1$ or $A[j-1] \leq A[j]$), we can then show that $A[1..i]$ is sorted. Furthermore, it is possible to show that this invariant is true every time the loop condition is tested. However, it still is not sufficient to prove the maintenance step for this loop. To see why, observe that it tells us nothing about how $A[j-1]$ compares with $A[j+1]$. Thus, when $A[j-1]$ is swapped with $A[j]$, we cannot show that $A[j] \leq A[j+1]$.

We need to express in our invariant that when we choose two indices $k < k'$, where $k' \neq j$, we must have $A[k] \leq A[k']$. The invariant in Figure 1.7 states precisely this fact. Arriving at this invariant, however, required some degree of effort.

We mentioned in Section 1.4 that starting the **for** loop with $i = 1$, rather than $i = 2$, simplifies the correctness proof without affecting the correctness. We can now explain what we meant. Note that if we were to begin the **for** loop with $i = 2$, its invariant would no longer be established initially if $n = 0$. Specifically, $A[1..i-1] = A[1..1]$, and if $n = 0$, $A[1]$ is not a valid array location. A more complicated invariant — and consequently a more complicated proof — would therefore be required to handle this special case. By instead beginning the loop at 1, we have sacrificed a very small amount of run-time overhead for the purpose of simplifying the invariant.

2.4 Combining Recursion and Iteration

In this section, we will present an alternative approach to solving the selection problem, specified in Figure 1.1 on page 4. This approach will ultimately result in a recursive algorithm that also contains a loop. We will then show how to combine the techniques presented in the last two sections in order to prove such an algorithm to be correct.

Figure 2.2 Specification of the median problem

Precondition: $A[1..n]$ is an array of numbers, and n is a positive integer.
Postcondition: Returns the median of $A[1..n]$; i.e., returns x such that fewer than $\lceil n/2 \rceil$ elements are less than x and at least $\lceil n/2 \rceil$ elements are less than or equal to x.

```
MEDIAN(A[1..n])
```

We will reduce the selection problem to the following three problems:

- the *Dutch national flag problem*, defined below;
- the problem of finding the median of a nonempty array of numbers (see Figure 2.2 for a specification of this problem); and
- a smaller instance of the selection problem.

Figure 2.2 uses the notation $\lceil x \rceil$, pronounced the ceiling *of x, to denote the smallest integer no smaller than x. Thus, $\lceil 3/2 \rceil = 2$, and $\lceil -3/2 \rceil = -1$.*

Somewhat informally, the input to the Dutch national flag problem is an array of items, each of which is colored either red, white, or blue. The goal is to arrange the items so that all of the red items precede all of the white items, which in turn precede all of the blue items. This order is the order of the colors appearing on the Dutch national flag, from top to bottom. We will modify the problem slightly by assuming that all items are numbers, and that a number is red if it is strictly less than some given value p, white if it is equal to p, or blue if it is strictly greater than p.

The formal specification of this problem is given in Figure 2.3. Note that we use the type INT to represent an integer. Notice also that because it may be important to know the number of items of each color, these values are returned in a 3-element array.

We can then find the kth smallest element in a nonempty array as follows:

1. Let p be the median element of the array.
2. Solve the resulting Dutch national flag problem.
3. If there are at least k red elements, return the kth smallest red element.
4. Otherwise, if there are at least k red and white elements combined, return p.
5. Otherwise, return the $(k - j)$th smallest blue element, where j is the number of red and white elements combined.

Figure 2.3 Specification of DutchFlag

Precondition: $A[lo..hi]$ is an array of Numbers, lo and hi are Ints such that $hi \geq lo - 1$, and p is a Number.
Postcondition: $A[lo..hi]$ is a permutation of its original values such that, all items less than p precede all items equal to p, which in turn precede all items greater than p. Returns an array $N[1..3]$ in which $N[1]$ is the number of items less than p, $N[2]$ is the number of items equal to p, and $N[3]$ is the number of items greater than p in $A[lo..hi]$.

```
DutchFlag($A[lo..hi], p$)
```

Note that after we have solved the Dutch national flag problem, all elements less than p appear first in the array, followed by all elements equal to p, followed by all elements greater than p. Furthermore, because steps 3 and 5 apply to portions of the array that do not contain p, these steps solve strictly smaller problem instances.

In what follows, we will develop a solution to the Dutch national flag problem. We will then combine that solution with the above reduction to obtain a solution to the selection problem (we will simply use the specification for Median). We will then prove that the resulting algorithm is correct.

In order to conserve resources, we will constrain our solution to the Dutch national flag problem to rearrange items by swapping them. We will reduce a large instance of the problem to a smaller instance. We begin by examining the last item. If it is blue, then we can simply ignore it and solve what is left. If it is red, we can swap it with the first item and again ignore it and solve what is left. If it is white, we need to find out where it belongs; hence, we temporarily ignore it and solve the remaining problem. We then swap it with the first blue item, or if there are no blue items, we can leave it where it is. This algorithm is shown in Figure 2.4.

If we were to implement this solution, or to analyze it using the techniques of Chapter 3, we would soon discover that its stack usage is too high. Furthermore, none of the recursive calls occur at either the beginning or the end of the computation; hence, the recursion is not tail recursion, and we cannot implement it bottom-up.

We can, however, use a technique called *generalization* that will allow us to solve the problem using a transformation. We first observe that the

Figure 2.4 A top-down implementation DutchFlag, specified in Figure 2.3

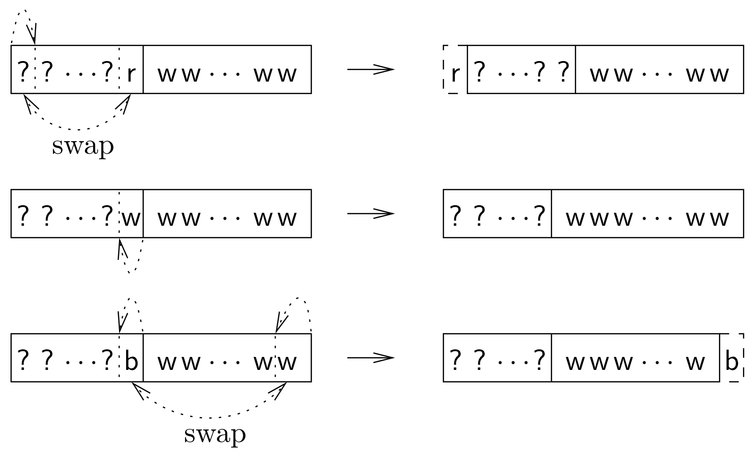
```
DutchFlagTD($A[lo..hi], p$)
    if $hi < lo$
        $N \leftarrow$ new Array[1..3]; $N[1] \leftarrow 0$; $N[2] \leftarrow 0$; $N[3] \leftarrow 0$
    else if $A[hi] < p$
        $A[lo] \leftrightarrow A[hi]$; $N \leftarrow$ DutchFlagTD($A[lo + 1..hi], p$)
        $N[1] \leftarrow N[1] + 1$
    else if $A[hi] = p$
        $N \leftarrow$ DutchFlagTD($A[lo..hi - 1], p$)
        $A[hi] \leftrightarrow A[lo + N[1] + N[2]]$; $N[2] \leftarrow N[2] + 1$
    else
        $N \leftarrow$ DutchFlagTD($A[lo..hi - 1], p$); $N[3] \leftarrow N[3] + 1$
    return $N$
```

only reason we must wait until after the recursive calls to increment the appropriate element of N is that the recursive call is responsible for constructing and initializing N. If instead, we could provide initial values for $N[1..3]$ to the recursive calls, we could then incorporate the color of the last element into these initial values. We therefore generalize the problem by requiring as input initial values for the number of items of each color. The returned array will then contain values representing the number of items of the corresponding color, plus the corresponding initial value from the input. By using 0 for all three initial values, we obtain the number of each color in the entire array; hence, we have defined a more general problem.

We can use this generalization to make two of the calls tail recursion. In order to be able to handle a white item, though, we need to modify our generalization slightly. Specifically, we need to know in advance where to put a white item. In order to be able to do this, let us specify that if w is given as the initial value for the number of white items, then the last w items in the array are white. Note that this variation is still a generalization of the original problem, because if $w = 0$, no additional constraints are placed on the input array.

Suppose we have an instance of this more general problem. If the initial value for the number of white items is equal to the number of elements in the array, then we can copy the initial values into $N[1..3]$ and return. Otherwise,

Figure 2.5 The transformation for Dutch national flag.



we examine the item preceding the first known white item (see Figure 2.5). If it is red, we swap it with the first item and solve the smaller problem obtained by ignoring the first item. If it is white, we solve the problem that results from incrementing the initial number of white items. If it is blue, we swap it with the last element, and solve the smaller problem obtained by ignoring the last item. A recursive implementation of this strategy is shown in Figure 2.6.

The way we handle the case in which an item is white is suspicious in that the reduced instance is an array with the same number of elements. However, note that in each case, the number of elements of unknown color is decreased by the reduction. Thus, if we choose our definition of "size" to be the number of elements of unknown color, then our reduction does decrease the size of the problem in each case. Recall that our notion of size is any natural number which decreases in all "smaller" instances. Our reduction is therefore valid.

Figure 2.7 shows the result of eliminating the tail recursion from DutchFlagTailRec, incorporating it into the selection algorithm described earlier in this section, and making some minor modifications. First, *lo* and *hi* have been replaced by 1 and n, respectively. Second, the array N has been removed, and r, w, b are used directly instead. Finally, referring to Figure 2.6, note that when a recursive call is made, *lo* is incremented exactly when r is incremented, and *hi* is decremented exactly when b is incremented. Be-

Figure 2.6 Tail recursive solution to a generalization of the Dutch national flag problem

Precondition: $A[lo..hi]$ is an array of NUMBERs whose last w items are equal to p, lo and hi are INTs such that $w \leq hi - lo + 1$, and r, w, and b are NATs.
Postcondition: $A[lo..hi]$ is a permutation of its original values such that all items less than p precede all items equal to p, which in turn precede all items greater than p. Returns an array $N[1..3]$ in which $N[1]$ is r plus the number of items less than p, $N[2]$ is the number of items equal to p, and $N[3]$ is b plus the number of items greater than p in $A[lo..hi]$.

```
DutchFlagTailRec($A[lo..hi], p, r, w, b$)
    if $w \geq hi - lo + 1$
        $N \leftarrow$ new Array[1..3]; $N[1] \leftarrow r$; $N[2] \leftarrow w$; $N[3] \leftarrow b$
        return $N$
    else
        $j \leftarrow hi - w$
        if $A[j] < p$
            $A[j] \leftrightarrow A[lo]$
            return DutchFlagTailRec($A[lo + 1..hi], p, r + 1, w, b$)
        else if $A[j] = p$
            return DutchFlagTailRec($A[lo..hi], p, r, w + 1, b$)
        else
            $A[j] \leftrightarrow A[hi]$
            return DutchFlagTailRec($A[lo..hi - 1], p, r, w, b + 1$)
```

cause we are replacing lo with 1, which cannot be changed, and hi with n, which we would rather not change, we instead use the expressions $r + 1$ and $n - b$, respectively. Thus, for example, instead of having a **while** loop condition of $w < hi - lo + 1$, we replace lo with $r + 1$ and hi with $n - b$, rearrange terms, and obtain $r + w + b < n$.

As we have already observed, the invariant for a loop implementing a transformation is closely related to the precondition for the problem. Thus, in order to obtain the loop invariant, we take the precondition for DutchFlagTailRec, remove "$A[lo..hi]$ is an array of NUMBERS", as this is understood, and replace lo with $r + 1$ and hi with $n - b$. This gives us most of

Figure 2.7 An algorithm for solving the selection problem, specified in Figure 1.1, using the median

```
SelectByMedian($A[1..n], k$)
    $p \leftarrow$ Median($A[1..n]$); $r \leftarrow 0$; $w \leftarrow 0$; $b \leftarrow 0$
    // **Invariant:** $r, w, b \in \mathbb{N}$, $r + w + b \leq n$, and $A[i] < p$ for $1 \leq i \leq r$,
    // $A[i] = p$ for $n - b - w < i \leq n - b$, and $A[i] > p$ for $n - b < i \leq n$.
    while $r + w + b < n$
        $j \leftarrow n - b - w$
        if $A[j] < p$
            $r \leftarrow r + 1$; $A[j] \leftrightarrow A[r]$
        else if $A[j] = p$
            $w \leftarrow w + 1$
        else
            $A[j] \leftrightarrow A[n - b]$; $b = b + 1$
    if $r \geq k$
        return SelectByMedian($A[1..r], k$)
    else if $r + w \geq k$
        return $p$
    else
        return SelectByMedian($A[1 + r + w..n], k - (r + w)$)
```

the invariant. However, we must also take into account that the iterations do not actually change the size of the problem instance; hence, the invariant must also include a characterization of what has been done outside of $A[r + 1..n - b]$. The portion to the left is where red items have been placed, and the portion to the right is where blue items have been placed. We need to include these constraints in our invariant.

Note that in Figure 2.7, the last line of SelectByMedian contains a recursive call in which the first parameter is $A[1 + r + w..n]$. However, the specification given in Figure 1.1 (page 4) states that the first parameter must be of the form $A[1..n]$. To accommodate such a mismatch, we adopt a convention that allows for automatic re-indexing of arrays when the specification requires a parameter to be an array whose beginning index is a fixed value. Specifically, we think of the sub-array $A[1 + r + w..n]$ as an array $B[1..n - (r + w)]$. B is then renamed to A when it is used as the actual parameter in the recursive call.

Let us now prove the correctness of SelectByMedian. Because SelectByMedian contains a loop, we must prove this loop's correctness using the techniques of Section 2.3. Specifically, we need the following lemma, whose proof we leave as an exercise.

Lemma 2.8 If the precondition for SelectByMedian is satisfied, then its **while** loop always terminates with $A[1..n]$ being a permutation of its original elements such that

- $A[i] < p$ for $1 \leq i \leq r$;
- $A[i] = p$ for $r < i \leq r + w$; and
- $A[i] > p$ for $r + w < i \leq n$.

Furthermore, when the loop terminates, r, w, and b are natural numbers such that $r + w + b = n$.

We can then prove the correctness of SelectByMedian using induction.

Theorem 2.9 SelectByMedian meets the specification of Select given in Figure 1.1.

Proof: By induction on n.

Induction Hypothesis: Assume that for some $n \geq 1$, whenever $1 \leq m < n$, SelectByMedian$(A[1..m], k)$ meets its specification, where $A[1..m]$ denotes (by re-indexing if necessary) any array with m elements.

Induction Step: Suppose the precondition is satisfied. By Lemma 2.8, the **while** loop will terminate with $A[1..n]$ being a permutation of its original elements such that $A[1..r]$ are less than p, $A[r+1..r+w]$ are equal to p, and $A[r + w + 1..n]$ are greater than p. We consider three cases.

Case 1: $k \leq r$. In this case, there are at least k elements less than p, so the kth smallest is less than p. Because $A[1..r]$ are all the elements smaller than p, the kth smallest of $A[1..n]$ is the kth smallest of $A[1..r]$. Because p is an element of $A[1..n]$ that is not in $A[1..r]$, $r < n$. Furthermore, because $k \leq r$, the precondition of Select is satisfied by the recursive call SelectByMedian$(A[1..r], k)$. By the Induction Hypothesis, this recursive call returns the kth smallest element of $A[1..r]$, which is the kth smallest of $A[1..n]$.

Case 2: $r < k \leq r + w$. In this case, there are fewer than k elements less than p and at least k elements less than or equal to p. p is therefore the kth smallest element.

Case 3: $r + w < k$. In this case, there are fewer than k elements less than or equal to p. The kth smallest must therefore be greater than p. It must therefore be in $A[r + w + 1..n]$. Because every element in $A[1..r + w]$ is less than the kth smallest, the kth smallest must be the $(k-(r+w))$th smallest element in $A[r + w + 1..n]$. Because p is an element of $A[1..n]$ that is not in $A[r+w+1..n]$, $r+w+1 > 1$, so that the number of elements in $A[r+w+1..n]$ is less than n. Let us refer to $A[r + w + 1..n]$ as $B[1..n - (r + w)]$. Then because $r + w < k$, $1 \leq k - (r + w)$, and because $k \leq n$, $k - (r + w) \leq n - (r + w)$. Therefore, the precondition for Select is satisfied by the recursive call SelectByMedian$(B[1..n - (r + w)], k - (r + w))$. By the Induction Hypothesis, this recursive call returns the $(k-(r+w))$th smallest element of $B[1..n - (r + w)] = A[r + w + 1..n]$. This element is the kth smallest of $A[1..n]$. $\square$

In some cases, a recursive call might occur inside a loop. For such cases, we would need to use the induction hypothesis when reasoning about the loop. As a result, it would be impossible to separate the proof into a lemma dealing with the loop and a theorem whose proof uses induction and the lemma. We would instead need to prove initialization, maintenance, termination, and correctness of the loop within the induction step of the induction proof.

2.5 Mutual Recursion

The techniques we have presented up to this point are sufficient for proving the correctness of most algorithms. However, one situation can occur which reveals a flaw in these techniques. Specifically, it is possible to prove correctness using these techniques, when in fact the algorithm is incorrect. The situation leading to this inconsistency is known as *mutual recursion.* In a simple case, we have one algorithm, A, which calls another algorithm, B, which in turn calls A. More generally, we may have a sequence of algorithms, $A_1, \ldots, A_n$, where A_i calls A_{i+1} for $1 \leq i < n$, and A_n calls A_1.

Figure 2.8 An implementation of Median, specified in Figure 2.7, using Select, as specified in Figure 1.1

```
MedianBySelect($A[1..n]$)
    return Select($A[1..n]$, $\lceil n/2 \rceil$)
```

For example, suppose we were to implement Median, as specified in Figure 2.7, by reducing it to the selection problem. This reduction is straightforward, as it is easily seen that the median is just the $\lceil n/2 \rceil$nd smallest element. We therefore have the algorithm shown in Figure 2.8. Its proof of correctness is trivial.

We therefore have the algorithm SelectByMedian, which correctly implements Select if we use a correct implementation of Median. We also have the algorithm MedianBySelect, which correctly implements Median if we use a correct implementation of Select. The problem arises when we use both of these implementations together. The proof of correctness for the resulting implementation contains circular reasoning. In fact, the implementation is not correct, as can be seen if we replace the call to Median in Figure 2.7 with the call, SelectByMedian($A[1..n]$, $\lceil n/2 \rceil$). We now have a recursive call whose argument is no smaller than that of the original call. As a result, we have infinite recursion.

Though we will not prove it here, it turns out that nontermination is the only way in which combining correct algorithms in a mutually recursive fashion can result in an incorrect implementation. Thus, if we can prove that the implementation terminates, we can conclude that it is correct. In Chapter 3, we will present techniques for showing not just termination, but the time it takes for an algorithm to terminate. These techniques will be general enough to apply to mutually recursive algorithms.

In Chapter 15, we will present algorithms that use mutual recursion. As we will see there, mutual recursion is sometimes useful for breaking a complicated algorithm into manageable pieces. Apart from that chapter, we will not dwell on mutual recursion. We should always be careful, however, when we combine algorithms, that we do not inadvertently introduce mutual recursion without proving termination of the implementation.

2.6 Finding Errors

The process of proving correctness of an algorithm is more than just an academic exercise. A proper correctness proof should give us confidence that a given algorithm is, in fact, correct. It therefore stands to reason that if a given algorithm is incorrect, the proof of correctness should fail at some point. The process of proving correctness can therefore help us to find errors in algorithms.

Suppose, for example, that in MaxSumBU (see Figure 1.14 on page 18), we had miscalculated *msuf* using the statement

$$\mathit{msuf} \leftarrow \mathit{msuf} + A[i].$$

We could have made such an error by forgetting that there is a suffix of $A[0..i]$ — the empty suffix — that does not end with $A[i]$. We would expect that a proof of correctness for such an erroneous algorithm should break down at some point.

This error certainly wouldn't affect the Initialization part of the proof, as the initialization code is unchanged. Likewise, the Correctness part doesn't depend directly on code within the loop, but only on the invariant and the loop exit condition. Because the Termination part is trivial, we are left with the Maintenance part. Because we have changed the calculation of *msuf*, we would expect that we would be unable to prove that the second part of the invariant is maintained (i.e., that *msuf* is the maximum suffix sum for $A[0..i-1]$).

Let us consider how this statement changes the Maintenance part of the proof of Theorem 2.5. The first change is that now $\mathit{msuf}' = \mathit{msuf} + A[i]$.

This affects the derivation from *msuf*$'$ in the following way:

$$\begin{aligned}
\mathit{msuf}' &= \mathit{msuf} + A[i]\\
&= t_i + A[i]\\
&= \max\left\{\sum_{k=l}^{i-1} A[k] \mid 0 \le l \le i\right\} + A[i]\\
&= \max\left\{A[i] + \sum_{k=l}^{i-1} A[k] \mid 0 \le l \le i\right\}\\
&= \max\left\{\sum_{k=l}^{i} A[k] \mid 0 \le l \le i\right\}\\
&= \max\left\{\sum_{k=l}^{i'-1} A[k] \mid 0 \le l \le i'-1\right\}.
\end{aligned}$$

However,

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

Note that the set on the right-hand side of this last equality has one more element than does the set on the right-hand side of the preceding equality. This element is generated by $l = i'$, which results in an empty sum having a value of 0. All of the remaining elements are derived from values $l \le i' - 1$, which result in nonempty sums of elements from $A[0..i]$. Thus, if $A[0..i]$ contains only negative values, $\mathit{msuf}' < t_{i'}$. It is therefore impossible to prove that these values are equal.

A failure to come up with a proof of correctness does not necessarily mean the algorithm is incorrect. It may be that we have not been clever enough to find the proof. Alternatively, it may be that an invariant has not been stated properly, as discussed in Section 2.3. Such a failure always reveals, however, that we do not yet understand the algorithm well enough to prove that it is correct.

2.7 Summary

We have introduced two main techniques for proving algorithm correctness, depending on whether the algorithm uses recursion or iteration:

- The correctness of a recursive algorithm should be shown using induction.

- The correctness of an iterative algorithm should be shown by proving initialization, maintenance, termination, and correctness for each of the loops.

Some algorithms might contain both recursion and iteration. In such cases, both techniques should be used. Because the algorithm is recursive, its correctness should be shown using induction. In order to complete the induction, the loops will need to be handled by proving initialization, maintenance, termination, and correctness. In Chapter 4, we will see how these techniques can be extended to proving the correctness of data structures.

Though correctness proofs are useful for finding errors in algorithms and for giving us confidence that algorithms are correct, they are also quite tedious. On the other hand, if an algorithm is fully specified and designed in a top-down fashion, and if proper loop invariants are provided, working out the details of a proof is usually not very hard. For this reason, we will not provide many correctness proofs in the remainder of this text, but will leave them as exercises. We will instead give top-down designs of algorithms and provide invariants for most of the loops contained in them.

2.8 Exercises

Exercise 2.1 Induction can be used to prove solutions for summations. Use induction to prove each of the following:

a. The *arithmetic series*:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{2.1}$$

b. The *geometric series*:

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} \tag{2.2}$$

for any real $x \neq 1$.

Exercise 2.2 Let

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ \displaystyle\sum_{i=0}^{n-1} f(i) & \text{if } n > 0. \end{cases}$$

Use induction to prove that for all $n > 0$, $f(n) = 2^{n-1}$.

* **Exercise 2.3** The *Fibonacci sequence* is defined as follows:

$$F_n = \begin{cases} n & \text{if } 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases} \tag{2.3}$$

Use induction to prove each of the following properties of the Fibonacci sequence:

a. For every $n > 0$,

$$F_{n-1}F_n + F_nF_{n+1} = F_{2n} \tag{2.4}$$

and

$$F_n^2 + F_{n+1}^2 = F_{2n+1}. \tag{2.5}$$

[**Hint:** Prove both equalities together in a single induction argument.]

b. For every $n \in \mathbb{N}$,

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}, \tag{2.6}$$

where ϕ is the *golden ratio*:

$$\phi = \frac{1 + \sqrt{5}}{2}.$$

Exercise 2.4 Prove that RecursiveInsert, shown in Figure 1.4 on page 7, meets is specification, given in Figure 1.3 on page 7.

Exercise 2.5 Prove that MaxSuffixTD and MaxSumTD, given in Figure 1.13 (page 17), meet their specifications. For MaxSumTD, use the specification of MaxSum given in Figure 1.9 (page 13).

Exercise 2.6 Prove that DotProduct, shown in Figure 1.17 on page 21, meets its specification.

Exercise 2.7 Prove that Factorial, shown in Figure 2.9, meets its specification. $n!$ (pronounced, "n factorial") denotes the product $1 \cdot 2 \cdots n$ ($0!$ is defined to be 1).

Exercise 2.8 A minor modification of MaxSumOpt is shown in Figure 2.10 with its loop invariants. Prove that it meets the specification of MaxSum, given in Figure 1.9 (page 13).

Figure 2.9 Algorithm for Factorial

Precondition: n is a Nat.
Postcondition: Returns $n!$.

```
Factorial($n$)
    $p \leftarrow 1$
    // **Invariant:** $p = (i-1)!$
    for $i \leftarrow 1$ to $n$
        $p \leftarrow ip$
    return $p$
```

Figure 2.10 A minor modification of MaxSumOpt with loop invariants

```
MaxSumOpt2($A[0..n-1]$)
    $m \leftarrow 0$
    // **Invariant:** $m$ is the maximum of 0 and all sums of sequences $A[l..h-1]$
    // such that $0 \leq l < i$ and $l \leq h \leq n$.
    for $i \leftarrow 0$ to $n-1$
        $sum \leftarrow 0$; $p \leftarrow 0$
        // **Invariant:** $sum$ is the sum of the sequence $A[i..k-1]$, and $p$ is
        // the maximum prefix sum of $A[i..k-1]$.
        for $k \leftarrow i$ to $n-1$
            $sum \leftarrow sum + A[k]$
            $p \leftarrow \text{Max}(p, sum)$
        $m \leftarrow \text{Max}(m, p)$
    return $m$
```

Figure 2.11 A minor modification of MaxSumIter with invariants

```
MaxSumIter2($A[0..n-1]$)
    $m \leftarrow 0$
    // Invariant: $m$ is the maximum of 0 and all sums of sequences $A[l..h-1]$
    // such that $0 \leq l < i$ and $l \leq h \leq n$.
    for $i \leftarrow 0$ to $n$
        $p \leftarrow 0$
        // Invariant: $p$ is the maximum prefix sum of $A[i..j-1]$.
        for $j \leftarrow i$ to $n-1$
            $sum \leftarrow 0$
            // Invariant: $sum$ is the sum of the sequence $A[i..k-1]$.
            for $k \leftarrow i$ to $j$
                $sum \leftarrow sum + A[k]$
            $p \leftarrow$ Max($p, sum$)
        $m \leftarrow$ Max($p, m$)
    return $m$
```

Exercise 2.9 A minor modification of MaxSumIter is shown in Figure 2.11 with its loop invariants. Prove that it meets the specification of MaxSum, given in Figure 1.9 (page 13).

Exercise 2.10 Prove that DutchFlagTD, given in Figure 2.4 (page 40), meets its specification, given in Figure 2.3 (page 39).

* **Exercise 2.11** Figure 2.12 shows a slightly optimized version of InsertionSort. Prove that InsertionSort2 meets the specification given in Figure 1.2 on page 6. You will need to find appropriate invariants for each of the loops.

Exercise 2.12 Prove that DutchFlagTailRec, shown in Figure 2.6 on page 42, meets its specification.

Exercise 2.13 Prove Lemma 2.8 (page 44).

* **Exercise 2.14** Prove that Permutations, shown in Figure 2.13, meets it specification. Use the specifications of Copy, AppendToAll, and Factorial from Figures 1.18, 1.20, and 2.9, respectively.

Figure 2.13 corrected 2/10/10.

Figure 2.12 A slightly optimized version of InsertionSort

```
InsertionSort2($A[1..n]$)
    for $i \leftarrow 2$ to $n$
        $j \leftarrow i$; $t \leftarrow A[j]$
        while $j > 1$ and $A[j-1] > t$
            $A[j] \leftarrow A[j-1]$; $j \leftarrow j - 1$
        $A[j] \leftarrow t$
```

Figure 2.13 Algorithm for Permutations

Precondition: $A[1..n]$ is an array of distinct elements, and n is a Nat.
Postcondition: Returns an array $P[1..n!]$ of all of the permutations of $A[1..n]$, where each permutation is itself an array $A_i[1..n]$.

```
Permutations($A[1..n]$)
    $P \leftarrow$ new Array[1..Factorial($n$)]
    if $n = 0$
        $P[1] \leftarrow$ new Array[1..0]
    else
        $k \leftarrow 1$; $nmin1fac \leftarrow$ Factorial($n - 1$)
        // Invariant: $P[1..k-1]$ contains all of the permutations of $A[1..n]$
        // such that for $1 \leq j < k$, $P[j][n]$ is in $A[1..i-1]$.
        for $i \leftarrow 1$ to $n$
            $B \leftarrow$ new Array[1..$n-1$]
            Copy($A[1..i-1], B[1..i-1]$); Copy($A[i+1..n], B[i..n-1]$)
            $C \leftarrow$ Permutations($B[1..n-1]$)
            AppendToAll($C[1..nmin1fac], A[i]$)
            Copy($C[1..nmin1fac], P[k..k+nmin1fac-1]$)
            $k \leftarrow k + nmin1fac$
    return $P$
```

Figure 2.14 The algorithm for Exercise 2.15

Precondition: $A[lo..hi]$ is an array of NUMBERS, lo and hi are INTS, and p is a NUMBER such that either all occurrences of p in $A[lo..hi]$ precede all occurrences of other values, or all occurrences of p follow all occurrences of other values. We will refer to the first group of elements (i.e., either those equal to p or those not equal to p, whichever comes first) as yellow, and the other elements as green.
Postcondition: $A[lo..hi]$ is a permutation of its initial values such that all green elements precede all yellow elements.

```
SwapColors($A[lo..hi], p$)
    if $lo \leq hi + 1$
        $i \leftarrow lo$; $j \leftarrow hi$
        // Invariant: $lo \leq i \leq j + 1 \leq hi + 1$ and $A[lo..hi]$ is a
        // permutation of its original values such that $A[k]$ is green for
        // $lo \leq k < i$, $A[k]$ is yellow for $j < k \leq hi$, and in $A[i..j]$,
        // all yellow elements precede all green elements.
        while $i < j$ and $(A[i] = p) \neq (A[j] = p)$
            $A[i] \leftrightarrow A[j]$; $i \leftarrow i + 1$; $j \leftarrow j - 1$
```

Exercise 2.15 Prove that SWAPCOLORS, shown in Figure 2.14, meets its specification. Note that the second conjunct in the **while** condition is comparing two boolean values; thus, it is true whenever exactly one of $A[i]$ and $A[j]$ equals p.

Exercise 2.16 Figure 2.15 contains an algorithm for reducing the Dutch national flag problem to the problem solved in Figure 2.14. However, the algorithm contains several errors. Work through a proof that this algorithm meets its specification (given in Figure 2.3 on page 39), pointing out each place at which the proof fails. At each of these places, suggest a small change that could be made to correct the error. In some cases, the error might be in the invariant, not the algorithm itself.

* **Exercise 2.17** Reduce the sorting problem to the Dutch national flag problem and one or more smaller instances of itself.

Figure 2.15 Buggy algorithm for Exercise 2.16

```
DutchFlagFiveBands($A[lo..hi], p$)
    $i \leftarrow lo$; $j \leftarrow lo$; $k \leftarrow hi$; $l \leftarrow hi$
    // Invariant: $lo \le i \le j \le hi$, $lo \le k \le l \le hi$, $A[lo..i-1]$ all equal $p$,
    // $A[i..j-1]$ are all less than $p$, $A[k..l]$ are all greater than $p$, and
    // $A[l+1..hi]$ all equal $p$.
    while $j < k$
        if $A[j] < p$
            $j \leftarrow j + 1$
        else if $A[j] = p$
            $A[j] \leftrightarrow A[i]$; $i \leftarrow i + 1$; $j \leftarrow j + 1$
        else if $A[k] = p$
            $A[k] \leftrightarrow A[l]$; $k \leftarrow k - 1$; $l \leftarrow l - 1$
        else if $A[k] > p$
            $k \leftarrow k - 1$
        else
            $A[j] \leftrightarrow A[k]$; $j \leftarrow j + 1$; $k \leftarrow k - 1$
    $N \leftarrow$ new Array[1..3]
    $N[1] \leftarrow j - i$; $N[2] \leftarrow i - lo + hi - l$; $N[3] \leftarrow l - k$
    SwapColors($A[lo..j], p$)
    SwapColors($A[k..hi], p$)
    return $N[1..3]$
```

2.9 Chapter Notes

The techniques presented here for proving correctness of algorithms are based on Hoare logic [60]. More complete treatments of techniques for proving program correctness can be found in Apt and Olderog [6] or Francez [42]. Our presentation of proofs using invariants is patterned after Cormen, et al. [25].

A discussion of the Dutch national flag problem and the iterative solution used in SelectByMedian are given by Dijkstra [29]. The Collatz problem was first posted by Lothar Collatz in 1937. An up-to-date summary of its history is maintained by Eric Weisstein [111].