```
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3      sum = sum + A[i]
4  return sum
```

2.1-3
Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

2.1-4
Consider the ***searching problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1:n]$ and a value x.

Output: An index i such that x equals $A[i]$ or the special value NIL if x does not appear in A.

Write pseudocode for ***linear search***, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-5
Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. You might consider resources such as memory, communication bandwidth, or energy consumption. Most often, however, you'll want to measure computational time. If you analyze several candidate algorithms for a problem,

you can identify the most efficient one. There might be more than just one viable candidate, but you can often rule out several inferior algorithms in the process.

Before you can analyze an algorithm, you need a model of the technology that it runs on, including the resources of that technology and a way to express their costs. Most of this book assumes a generic one-processor, ***random-access machine (RAM)*** model of computation as the implementation technology, with the understanding that algorithms are implemented as computer programs. In the RAM model, instructions execute one after another, with no concurrent operations. The RAM model assumes that each instruction takes the same amount of time as any other instruction and that each data access—using the value of a variable or storing into a variable—takes the same amount of time as any other data access. In other words, in the RAM model each instruction or data access takes a constant amount of time—even indexing into an array.[9]

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then you could sort in just one step. Such a RAM would be unrealistic, since such instructions do not appear in real computers. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

The data types in the RAM model are integer, floating point (for storing real-number approximations), and character. Real computers do not usually have a separate data type for the boolean values TRUE and FALSE. Instead, they often test whether an integer value is 0 (FALSE) or nonzero (TRUE), as in C. Although we typically do not concern ourselves with precision for floating-point values in this book (many numbers cannot be represented exactly in floating point), precision is crucial for most applications. We also assume that each word of data has a limit on the number of bits. For example, when working with inputs of size n, we typically

---

[9] We assume that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations. For example, if array $A[1:n]$ starts at memory address 1000 and each element occupies four bytes, then element $A[i]$ is at address $1000 + 4(i - 1)$. In general, computing the address in memory of a particular array element requires at most one subtraction (no subtraction for a 0-origin array), one multiplication (often implemented as a shift operation if the element size is an exact power of 2), and one addition. Furthermore, for code that iterates through the elements of an array in order, an optimizing compiler can generate the address of each element using just one addition, by adding the element size to the address of the preceding element.

assume that integers are represented by $c \log_2 n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant-time instruction? In the general case, no: to compute x^n when x and n are general integers typically takes time logarithmic in n (see equation (31.34) on page 934), and you must worry about whether the result fits into a computer word. If n is an exact power of 2, however, exponentiation can usually be viewed as a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by n positions to the left. In most computers, shifting the bits of an integer by 1 position to the left is equivalent to multiplying by 2, so that shifting the bits by n positions to the left is equivalent to multiplying by 2^n. Therefore, such computers can compute 2^n in 1 constant-time instruction by shifting the integer 1 by n positions to the left, as long as n is no more than the number of bits in a computer word. We'll try to avoid such gray areas in the RAM model and treat computing 2^n and multiplying by 2^n as constant-time operations when the result is small enough to fit in a computer word.

The RAM model does not account for the memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. Several other computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. Section 11.5 and a handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book do not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because an algorithm might behave differently for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Analysis of insertion sort

How long does the INSERTION-SORT procedure take? One way to tell would be for you to run it on your computer and time how long it takes to run. Of course, you'd