Chapter 2

Arrays, Iteration, Invariants

Data is ultimately *stored* in computers as patterns of bits, though these days most programming languages deal with higher level objects, such as characters, integers, and floating point numbers. Generally, we need to build algorithms that manipulate collections of such objects, so we need procedures for storing and sequentially processing them.

2.1 Arrays

In computer science, the obvious way to store an ordered collection of items is as an *array*. Array items are typically stored in a sequence of computer memory locations, but to discuss them, we need a convenient way to write them down on paper. We can just write the items in order, separated by commas and enclosed by square brackets. Thus,

$$[1, 4, 17, 3, 90, 79, 4, 6, 81]$$

is an example of an array of integers. If we call this array a, we can write it as:

$$a = [1, 4, 17, 3, 90, 79, 4, 6, 81]$$

This array a has 9 items, and hence we say that its *size* is 9. In everyday life, we usually start counting from 1. When we work with arrays in computer science, however, we more often (though not always) start from 0. Thus, for our array a, its positions are $0, 1, 2, \ldots, 7, 8$. The element in the 8^{th} position is 81, and we use the notation $a[8]$ to denote this element. More generally, for any integer i denoting a position, we write $a[i]$ to denote the element in the i^{th} position. This position i is called an *index* (and the plural is *indices*). Then, in the above example, $a[0] = 1$, $a[1] = 4$, $a[2] = 17$, and so on.

It is worth noting at this point that the symbol = is quite *overloaded*. In mathematics, it stands for equality. In most modern programming languages, = denotes assignment, while equality is expressed by ==. We will typically use = in its mathematical meaning, unless it is written as part of code or pseudocode.

We say that the individual items $a[i]$ in the array a are *accessed* using their index i, and one can move sequentially through the array by incrementing or decrementing that index, or jump straight to a particular item given its index value. Algorithms that process data stored as arrays will typically need to visit systematically all the items in the array, and apply appropriate operations on them.

2.2 Loops and Iteration

The standard approach in most programming languages for repeating a process a certain number of times, such as moving sequentially through an array to perform the same operations on each item, involves a *loop*. In *pseudocode*, this would typically take the general form

```
For i = 1,...,N,
   do something
```

and in programming languages like *C* and *Java* this would be written as the *for-loop*

```
for( i = 0 ; i < N ; i++ ) {
   // do something
}
```

in which a *counter* i keep tracks of doing "the something" N times. For example, we could compute the sum of all 20 items in an array a using

```
for( i = 0, sum = 0 ; i < 20 ; i++ ) {
   sum += a[i];
}
```

We say that there is *iteration* over the index i. The general *for-loop* structure is

```
for( INITIALIZATION ; CONDITION ; UPDATE ) {
   REPEATED PROCESS
}
```

in which any of the four parts are optional. One way to write this out explicitly is

```
INITIALIZATION
if ( not CONDITION ) go to LOOP FINISHED
LOOP START
   REPEATED PROCESS
   UPDATE
   if ( CONDITION ) go to LOOP START
LOOP FINISHED
```

In these notes, we will regularly make use of this basic loop structure when operating on data stored in arrays, but it is important to remember that different programming languages use different syntax, and there are numerous variations that check the condition to terminate the repetition at different points.

2.3 Invariants

An *invariant*, as the name suggests, is a condition that does not change during execution of a given program or algorithm. It may be a simple inequality, such as "$i < 20$", or something more abstract, such as "the items in the array are sorted". Invariants are important for data structures and algorithms because they enable *correctness proofs* and *verification*.

In particular, a *loop-invariant* is a condition that is true at the beginning and end of every iteration of the given loop. Consider the standard simple example of a procedure that finds the minimum of n numbers stored in an array a:

```
minimum(int n, float a[n]) {
   float min = a[0];
   // min equals the minimum item in a[0],...,a[0]
   for(int i = 1 ; i != n ; i++) {
      // min equals the minimum item in a[0],...,a[i-1]
      if (a[i] < min) min = a[i];
   }
   // min equals the minimum item in a[0],...,a[i-1], and i==n
   return min;
}
```

At the beginning of each iteration, and end of any iterations before, the invariant "min equals the minimum item in $a[0], ..., a[i-1]$" is true – it starts off true, and the repeated process and update clearly maintain its truth. Hence, when the loop terminates with "$i == n$", we know that "min equals the minimum item in $a[0], ..., a[n-1]$" and hence we can be sure that min can be returned as the required minimum value. This is a kind of *proof by induction*: the invariant is true at the start of the loop, and is preserved by each iteration of the loop, therefore it must be true at the end of the loop.

As we noted earlier, formal proofs of correctness are beyond the scope of these notes, but identifying suitable loop invariants and their implications for algorithm correctness as we go along will certainly be a useful exercise. We will also see how invariants (sometimes called *inductive assertions*) can be used to formulate similar correctness proofs concerning properties of data structures that are defined inductively.