Loop invariants: Why learn about them

We now study *loop invariants*. Loop invariants will be used to help you understand loops more easily. More importantly, we want to give you a *skill* in developing loop invariants and then loops. We don't just want simply to show you algorithms, we want to teach you how to *develop* them. We teach *programming* rather than just *programs*.

Once you learn about loop invariants, we will be able to *develop* searching algorithms, sorting algorithms, and other algorithms that manipulate arrays efficiently and effectively. You will find these algorithms *memorable* —not because you memorize the code, which is almost impossible, but because *you* can then develop the algorithms from their specifications.

Let's look at two algorithms and discuss your current inability to understand or develop them.

1. Exponentiation. The Java code given below computes b^c under the assumption that $c \geq 0$. For example, if b = 3 and c = 4, the algorithm stores 81 in z.

You can see that the loop terminates since initially $y \geq 0$ and each iteration reduces y by at least 1 and keeps $y \geq 0$. But how do you conclude that z contains b^c when the loop terminates? At the moment, we have no way to explain that to you. You can hand-execute the algorithm for some initial values of b and c to see that it works, but that does not help you *know* that it is correct and how it was developed.

```
// Store b^c in z, given c >= 0
int x= b;
int y= c;
int z= 1;
while (y != 0) {
   if (y % 2 == 0)
      { x= x*x;  y= y/2; }
   else
      { z= z*x; y= y-1; }
}
```

2. Sorting an array. The algorithm below sorts array b. You can execute it on some test cases, but that doesn't help you to remember the algorithm or to be able to produce it at will. Executing an algorithm on a few cases rarely gives such understanding. We will show you a simple methodology that will let *you* develop the algorithm. Then, you can do it whenever you want —just apply the methodology and the loop writes itself.

```
// Sort array b
int k= 0;
while (k < b.length) {
      Let j be the index of the minimum value in b[k..b.length-1];
      Swap b[k] and b[j];
      k= k+1;
}
```

Engineering principle

A general principle in any engineering discipline is to split a complex task into several parts, each as independent as possible; when all tasks are done, put the parts together. We should do this with loops, too.

Look at the initialization in the first example given above (You can do the same thing with the second example). Why is x set to b, y to c, and z to 1? You can't know that until you have "understood" the whole algorithm, probably by executing it by hand. Why does the loop terminate when y is 0? Again, you can't know that until you have looked at all the code and understood it. You need to understand the whole algorithm to understand its parts. That goes against the engineering principle.

The loop invariant will help us put the engineering principle into practice for this loop, allowing us, for example, to understand the initialization without having to look at the loop itself. That's what you have to look forward to!