Special Topic 6.5

Loop Invariants

Consider the task of computing a^n, where a is a floating-point number and n is a positive integer. Of course, you can multiply a · a · . . . · a, n times, but if n is large, you'll end up doing a lot of multiplication. The following loop computes a^n in far fewer steps:

```
double a = . . .;
int n = . . .;
double r = 1;
double b = a;
int i = n;
while (i > 0)
{
   if (i % 2 == 0) // n is even
   {
      b = b * b;
      i = i / 2;
   }
   else
   {
      r = r * b;
      i--;
   }
}
// Now r equals a to the nth power
```

Consider the case `n = 100`. The method performs the steps shown in the table below.

Computing a^{100}		
b	i	r
a	100	1
a^2	50	
a^4	25	
	24	a^4
a^8	12	
a^{16}	6	
a^{32}	3	
	2	a^{36}
a^{64}	1	
	0	a^{100}

Amazingly enough, the algorithm yields exactly a^{100}. Do you understand why? Are you convinced it will work for all values of `n`? Here is a clever argument to show that the method always computes the correct result. It demonstrates that whenever the program reaches the top of the `while` loop, it is true that

$$r \cdot b^i = a^n \tag{I}$$

Certainly, it is true the first time around, because `b = a` and `i = n`. Suppose that (I) holds at the beginning of the loop. Label the values of `r`, `b`, and `i` as "old" when entering the loop, and as "new" when exiting the loop. Assume that upon entry

$$r_{old} \cdot b_{old}{}^{i_{old}} = a^n$$

In the loop you must distinguish two cases: i_{old} even and i_{old} odd. If i_{old} is even, the loop performs the following transformations:

$$\begin{aligned} r_{new} &= r_{old} \\ b_{new} &= b_{old}^2 \\ i_{new} &= i_{old}/2 \end{aligned}$$

Therefore,

$$\begin{aligned} r_{new} \cdot b_{new}{}^{i_{new}} &= r_{old} \cdot \left(b_{old}\right)^{2 \cdot i_{old}/2} \\ &= r_{old} \cdot b_{old}{}^{i_{old}} \\ &= a^n \end{aligned}$$

On the other hand, if i_{old} is odd, then

$$\begin{aligned} r_{new} &= r_{old} \cdot b_{old} \\ b_{new} &= b_{old} \\ i_{new} &= i_{old} - 1 \end{aligned}$$

Therefore,

$$\begin{aligned} r_{new} \cdot b_{new}^{\,i_{new}} &= r_{old} \cdot b_{old} \cdot b_{old}^{\,i_{old}-1} \\ &= r_{old} \cdot b_{old}^{\,i_{old}} \\ &= a^n \end{aligned}$$

In either case, the new values for `r`, `b`, and `i` fulfill the *loop invariant* (I). So what? When the loop finally exits, (I) holds again:

$$r \cdot b^i = a^n$$

Furthermore, we know that `i = 0`, because the loop is terminating. But because `i = 0`, $r \cdot b^i = r \cdot b^0 = r$. Hence $r = a^n$, and the method really does compute the `n`th power of `a`.

This technique is quite useful, because it can explain an algorithm that is not at all obvious. The condition (I) is called a **loop invariant** because it is true when the loop is entered, at the top of each pass, and when the loop is exited. If a loop invariant is chosen skillfully, you may be able to deduce correctness of a computation. See *Programming Pearls* (Jon Bentley, Addison-Wesley 1986, Chapter 4) for another nice example.