Loop Invariants

Definition

A *loop invariant* for a given loop is a Boolean statement that is true before the first iteration of the loop and is not changed to false during any iteration of the loop. (We call that the inductive condition.) Consequently, the loop invariant is true after the loop terminates. Do not confuse the loop invariant with the loop condition, which is also a Boolean statement.

```
 1 int i = n; // input condition: n >= 0
 2 int j = 0;
 3 // loop invariant holds here
 4 while(i > 0)
 5  {
 6   // if the loop invariant holds here,
 7   i = i-1;
 8   j = j+1;
 9   // then the loop invariant holds here.
10  }
11   // thus the loop invariant holds here.
12   cout << j;
```

The goal of this code is to output n. The loop invariant is the statement "$i \geq 0$ and $i + j = n$" The invariant is clearly true at line 3. We need to prove that, if the invariant holds at line 6, it holds at line 9. It then must hold at line 11.

Consider one iteration. let i and i' be the value of i at the beginning and end of one iteration, that is, at lines 6 and 9, respectively. Similarly, let j and j' be the values of j at lines 6 and 9.

At line 3, the invariant holds, because $i \geq 0$ by the input condition, and $i + j = i = n$. We need to prove the inductive condition, that is, that if the invariant holds at line 6 during an iteration it holds at line 9 during the same iteration. Suppose the invariant holds at line 6 during one iteration. Hence $i \geq 0$ and $i + j = n$. We have $i' = i - 1$ and $j' = j + 1$. By the loop condition, $i > 0$, hence $i \geq 1$ since i is an integer, hence $i' = i - 1 \geq 1 - 1 = 0$. Since $j' = j + 1$ we have $i' + j' = (i - 1) + (j + 1) = i + j = n$. Thus the invariant holds at line 9, since $i' \geq 0$ and $i' + j' = n$.

We need to prove the code correct. At line 11, since the loop condition is false, we have $i \leq 0$. By the loop invariant, $i \geq 0$, hence $i = 0$. By the loop invariant, $i + j = n$. Thus $j = n - i = n$ at line 11, hence n is the output at line 12.

Squaring

We now look at another example. The function *square* below returns the square of its parameter.

```
 1 int square(int n) // input condition: n >= 0
 2  {
 3   int i = 0;
 4   int s = 0;
 5   // invariant holds here
 6   while(i < n)
 7    {
 8     // if invariant holds here,
 9     s=s+i+i+1;
10     i++;
11     // then it holds here.
12    }
13   // invariant holds here.
14   return s;
15  }
```

The loop invariant is "$i \leq n$ and $i^2 = s$."

The loop invariant holds at line 5, since $i = 0 \leq n$, and $0^2 = 0$. Now we prove the inductive condition. Let i, i' and s, s' be the values of the variables at lines 8 and 11, respectively, for an iteration. Then $i' = i + 1$ and $s' = s + 2i + 1$. Assume that the invariant holds at line 8. At line 11 $i < n$ by the loop condition; since i is an integer, $i <= n - 1$, hence $i' = i + 1 \leq n$; and $(i')^2 = (i + 1)^2 = i^2 + 2i + 1 = s + 2i + 1 = s'$. Thus, the invariant holds at line 11.

Finally, we prove correctness. At line 11, the invariant holds, and $i \geq n$ since the loop condition is false. By the loop invariant, $i \leq n$, hence $i = n$, and $s = i^2$. The output is $s = i^2 = n^2$, and thus the function is correct.

Multiplication

The purpose of the code below is to compute a product. This code works even though it does not use the multiplication operator, although we double or halve numbers.

```
int product(int a, int n) // input condition: n >= 0
 {
  int s = 0;
  int b = a;
  int m = n;
  // Loop Invariant: mb + s = na
  while(m > 0)
   {
    if(m%2) // that means m is odd
     s = s+b;
```

```
    b = b+b;
    m = m/2;
   }
  cout << s << endl;
 }
```

We now prove that $na = mb + s$ is a loop invariant. We first note that it holds before the first iteration of the loop, since $mb + s = na + 0 = na$. We now prove the inductive step, namely that if the invariant holds at the beginning of an iteration it holds at the end of that iteration. Assume $mb + s = na$ at the beginning of an iteration. Let m', b', s' be the value of the variables at the end of the iteration. We thus need to prove that $m'b' + s' = na$.

Case I: m is even. Then $m' = m/2$, $b' = 2b$, and $s' = s$.

$$\begin{aligned} m'b' + s' &= (m/2)(2b) + s \\ &= mb + s = na \end{aligned}$$

Case II: m is odd. Then $m' - (m-1)/2$, $b' = 2b$, and $s' = s + b$.

$$\begin{aligned} m'b' + s' &= ((m-1)/2)(2b) + s + b \\ &= (m-1)b + s + b \\ &= mb + s = na \end{aligned}$$

Finally, we prove correctness, that is, the output is na. Since the loop invariant holds at the end of every iteration, it holds after the last iteration. Thus $na = mb + s$ after the last iteration. But $m = 0$, which means that $na = s$

Powers

The code below computes x^n, where n is an integer and x is a real number. We assume that $n \geq 0$.

```
float power(float x, int n) // input condition: n >= 0
 {
  float z = 1.0;
  float y = x;
  int m = n;
  while(m > 0)
   {
    if(m%2) // that means m is odd
     z = z*y;
    y = y*y;
    m = m/2;
   }
  cout << z << endl;
 }
```

Verify that $y^m * z = x^n$ is a loop invariant, and that the code outputs x^n.

Quicksort

Assume that we need to sort an array A in the comparison-exchange model of computation. That is, all decisions made by the algorithm are comparison between two entries of A, and all changes of A are transpositions (swaps) of two of its entries. The basic concept of quicksort is not difficult, but writing the code is tricky. Here is the definition of quicksort:
1. Select an entry of A, and swap it into the first position of A. Call that entry *pivot*. (We ignore the question of how we choose that entry.)
2. Partition A into two subarrays, L on the left and R on the right, so that $x \leq$ pivot for all x in L and $x \geq$ pivot for all x in L.
3. Swap the first entry of A with the last entry of L. The last entry of L is now pivot.
4. Recursively sort all but the last entry of L.
5. Recursively sort R.
The tricky part is writing the code for the partition step. It is my experience that to write this code correctly, you must explicitly state the loop invariant of the partition loop, then conciously write the code to maintain the inductive property of that invariant.

There are many correct ways to write the code, as well as lots of incorrect ways. I will show one of my favorites. For simplicity, we assume that we always pick the first entry of A to be the pivot entry. We use movable indices lo and hi; during the partition loop hi decreases and lo increases. At the end of the loop, lo = hi. The loop invariant is

"A[first] = pivot, and lo $\leq$ hi, and $A[i] \leq$ pivot for all first $< i \leq$ lo, and $A[j] \geq$ pivot for all hi $<$ j $\leq$ last."

```
 1 quicksort(array A, int first, int last) // sorts A[first] ... A[last]
 2  {
 3   if(first < last) // otherwise, A is already sorted
 4    {
 3     int pivot = A[first];
 4     int lo = first;
 5     int hi = last;
 6     // loop invariant holds here
 7     while(lo < hi) // partition loop
 8      {
 9       while(A[lo+1] < pivot) lo++;
10       while(A[hi] > pivot) hi--;
11       // this is a place where it is easy to make errors
12       if(hi-lo >= 2) // the easy case
13        {
14         swap(A[lo+1],A[hi]);
15         lo++;
16         hi--;
17        }
18       else if(lo+1==hi) // this is tricky
19        lo++;
20      } // end of partition loop
20      // lo == hi and the loop invariant holds here
20     swap(A[first],A[hi]); // pivot is between the left and right subarrays
```

```
21     quicksort(first,hi-1); // recursive call to the left subarray
22     quicksort(hi+1,last); // recursive call to the left subarray
23    }
24  }
```

The code would be easier if it were known that all entries of A are distinct. In general, we cannot make that assumption, and the code must be designed in such a way that the code is not only correct, but does not automatically become quadratic when there are duplicate items. The partition loop must not only preserve the loop invariant, but must also make "progress," meaning that the subarray of unpartitioned items must shrink. In our code the value of hi $-$ lo decreases by at least 1 during each iteration, ensuring that the time complexity of the loop is $O(n)$.

The partition loop contains subloops at lines 9 and 10. Each of those must preserve the invariant. We can increment lo $A[\text{lo}+1] \leq$ pivot and decrement hi as long as $A[\text{hi}] \geq$ pivot without violating the loop invariant. However doing this would cause the algorithm to take quadratic time if all items have the same value; lo would increase to last and hi would never decrease, yielding an extremely uneven partition, and the recursion would have depth $\Theta(n)$, where n is the length of the array.

Our solution is to notice that an entry of value pivot can go to either the left or the right, and we want those two choices to happen approximately equally often. If both subloops have terminated, then $A[\text{lo}+1] \geq$ pivot and $A[\text{hi}] \leq$ pivot. They could both be equal to pivot. In lines 12–17, one goes to the left and the other to the right.

The tricky situation is that lo+1 = hi. In this case, A[lo+1] = A[hi] = pivot. We might have an infinite loop if we are not careful. We finish the partition loop by incrementing lo; we could instead, decrement hi.

At this point in the code, the loop invariant holds and the loop condition is false, implying that lo = hi. After transposing $A[\text{first}]$ and $A[\text{hi}]$, the array consists of two subarrays with one item in the middle, the pivot. The left subarray and right subarrays are then recursively sorted.

Floyd-Warshall Algorithm for the All-Pairs Minpath Problem

Consider the code of the Floyd-Warshall algorithm. Let the vertices of a weighted directed graph G be the integers $1, \ldots n$, and let $W_[i,k]$ be the weight of the arc from i to k, if such an arc exists. If not we let $W[i,k] = \infty$. In our code, $V[i,k]$ is the least weight of any path found **so far** from i to k.

```
1 For all i,k let V[i,k] = W[i,k] and back[i,k] = i.
2 For all i let V[i,i] = 0.
3 j = 0;
4 // loop invariant holds here
5 while(j < n)
6  {
7   // if the loop invariant holds here
8   j++;
9   for all i and k // really two nested loops
10    {
11     temp = V[i,j]+V[j,k];
12     if(temp < V[i,k])
13      {
```

```
14        V[i,k] = temp;
15        back[i,k] = back[j,k];
16       }
17     }
18   // then the loop invariant holds here
19  }
20  // the loop invariant holds here and j = n, hence V[i,k] is the shortest path from i to k.
```

The key to understand correctness of this code, which takes $\Theta(n^3)$ time, is to understand the loop invariant of the outer loop. The invariant contains nested quantifiers:

"For any i and any k, $V[i,k]$ is the least weight of any path from i to k whose interior does not contain any vertex of index greater than j."

At line 20, $j = n$, and the loop invariant holds vacuously, because there is no vertex of index greater than j. Thus the code is correct.