1 Introduction

A *loop invariant* is a condition that is true at the beginning and end of every loop iteration. When you write a loop that works correctly, you are at least implicitly relying on a loop invariant. Knowing what a loop invariant is and thinking explicitly about loop invariants will help you write correct, efficient code and to develop tricky algorithms.

2 Binary search via tail recursion

```
/** Returns an index i such that a[i] == k.
    Requires: k is in a[l..r], elements in a are in ascending sorted
    order, l <= r.
    Performance: O(lg(n)) where n = r-l+1
    */
int search(int[] a, int l, int r, int k) {
    if (l==r) return l;
    int m = (l + r)/2;
    if (k <= a[m])
        return search(a, l, m, k);
    else
        return search(a, m+1, r, k);
}
```

This algorithm is deceptively tricky to get right. It's pretty easy to get close, but how do we know we got the computation of `m` right? Why is it `k <= a[m]` and not `k < a[m]`? Why `m` and `m+1` in the two recursive calls to `search`? If we change any of these decisions, the algorithm will fail to find the correct element and may fail to terminate.

3 Binary search via iteration

Since the recursive algorithm is tail-recursive, we can convert it into code that uses a loop.

```
int search(int[] a, int l, int r, int k) {
    while (l < r) {
        int m = (l+r)/2;
        if (k <= a[m]) r = m;
        else l = m+1;
    }
    return l;
}
```

The precondition of the recursive implementation becomes a loop invariant with three clauses:

1. $k \in a[l..r]$
2. a is sorted in ascending order
3. $l \leq r$

4 Using loop invariants to show code is correct

Loop invariants can help us convince ourselves that our code, especially tricky code is correct. They also help us develop code to be correct in the first place, and they help us write efficient code.

To use a loop invariant to argue that code does what we want, we use the following steps:

1. Show that the loop invariant is true at the very beginning of the loop.
2. Show that if we assume the loop invariant is true at the beginning of the loop, it is also true at the end of the loop. (Other than coming up with the loop invariant in the first place, this is usually the trickiest step.)
3. Show that if the loop guard is false (so the loop exits) and the loop invariant still holds, the loop must have achieved what is desired.

These three steps allow us to conclude that the loop satisfies *partial correctness*, which means that if the loop terminates, it will succeed. To show *total correctness*, meaning that the loop will terminate, there is a fourth step:

4. Show that some quantity decreases on every loop iteration, and that (assuming the loop invariant holds), it cannot decrease indefinitely without making the loop guard false.

5 Example: Exponentiation by squaring and multiplication

```
/** Returns: x^e.
    Performance: O(lg(e))
*/
int pow(int x, int e) {
    int r = 1;
    int b = x;
    int y = e;
    // loop invariant: r · b^y = x^e
    while (y > 0) {
        if (y % 2 == 1) r = r * b;
        y = y/2;
        b = b*b;
    }
}
```

6 Example: Insertion sort