# Recitation 1 Notes: Proving Correctness

Jim Sukha

September 9, 2005

## 1 Formal Specification for an Algorithm

To be able to prove that an algorithm is correct, we first have to formally specify the problem. We can do this by specifying a **precondition** and **postcondition** for an algorithm.

- **Precondition**: The assumptions we make about the input.
- **Postcondition**: What the algorithm guarantees if the precondition was satisfied.

The algorithm is considered to be correct if, given any input that satisfies the precondition, the algorithm produces an output that satisfies the postcondition.

### Example

Consider the problem of searching a sorted array of $n$ elements, $A[1..n]$, for a key $x$. Suppose we give an algorithm `Search`$(A[1..n], x)$ for this problem.

- **Precondition**: The array $A[1..n]$ is sorted, i.e., that $A[1] \leq A[2] \leq A[3] \ldots \leq A[n]$.
- **Postcondition**: The search algorithm `Search`$(A[1..n], x)$ should return `true` if and only if $A[i] = x$ for some $1 \leq i \leq n$.

## 2 Iterative Algorithm – Linear Search

The following pseudo-code describes one way to do a linear search of a sorted array.

```
1    LinearSearch(A[1..n], x)
2        i <-- 1;
3
4        while ((A[i] < x) and (i <= n))
5            i <-- i + 1;
6
7        if (i <= n)
8            return (A[i] == x);
9        else
10           return false;
```

To check for correctness, we (at least theoretically) could for every possible input, check that the lines of code that are executed produce the right output. The difficult part about proving correctness of this algorithm is understanding the loop, in lines 4 and 5, because the number of times these lines execute depends on the input.

Instead of trying to figure out how to calculate what happens to $i$ for every possible input, we instead look for a **loop invariant** (call it $L$). The loop invariant is a statement that is always at the beginning and end of the loop ("beginning" is right before line 4 gets run, and "end" is right after line 5 is run).

There are many possible loop invariants that we could show. For example, $L =$ `true` is a correct loop invariant. The trick is finding a useful invariant that captures enough information.

Let $G$ be the statement that is being checked in the while loop in line 4, i.e., $G = (A[i] < x)$ and $i \leq n)$ implies that the loop executes again. Then, we know when the loop terminates, then $G$ is false. When we are proving correctness, imagine that instead of having lines 4 and 5 there, we have a single magical line of code that executes, line 4.5, that after it executes, guarantees the statement

$$L \text{ and (not } G).$$

Thus, a loop invariant of $L =$ `true` isn't useful because we've lost a lot of useful information.

## 2.1 Steps to Prove Correctness of an Iterative Algorithm

To prove an iterative algorithm correct:

1. Guess a loop invariant $L$.
2. Prove by induction that $L$ is actually a loop invariant.
3. Show that if the loop terminates and the loop invariant is true, then postcondition is satisfied.
4. Argue that the loop will actually terminate.

## 2.2 Proof for Linear Search

Let's follow these steps:

1. In this example, what a good loop invariant?

   The `while` loop is designed so that it stops either after finding the first element $A[i]$ that is greater than or equal to the key $x$, or when $i$ has gone past the end of the array. Thus, the element right before the one we stopped at, $A[i-1]$ should be less than $x$. Thus, a good loop invariant to guess is

   - $L$: If $i > 1$, then $A[i-1] < x$.

2. Prove loop invariant via induction on a variable $z$, where $z$ is the number of times the loop has been executed.

   - **Base Case**: When $z = 0$, i.e., before the first iteration, $i = 1$, so $L$ holds automatically.
   - **Inductive Step**:
     For the inductive hypothesis, assume after the $z$th iteration, $L$ holds at the beginning of the loop, i.e., $A[i-1] < x$.
     Suppose that variable $i$ has value $k$, i.e., $i = k$.
     There are two possibilities when we execute the loop:
     - **Case 1**: $A[k] < x$ and $i \leq n$:
       Then we increment $i$ to $k+1$. Then after the loop, we have that $A[i-1] = A[k] < x$.
       **Case 2**: $A[k] \geq x$ or $i > n$:
       Then we don't increment $i$. Since we didn't change $i$, by our inductive hypothesis, we still have $A[i-1] < x$.

     In either case, after the $z+1$st iteration, the loop invariant $L$ is still true.

3. Argue that if the loop terminates, then the running the rest of the algorithm satisfies the postcondition.

   Once the code gets to line 6, i.e., after the loop completes, we know $A[i-1] < x$. In order for the while loop to have terminated in the first place, we must have that ($x \leq A[i]$ or $i > n$).

- **Case 1:** $x \leq A[i]$. Then, if $A[i] = x$, then $x$ is obviously in the list, so the algorithm returns the correct value.

  If $A[i] \neq x$, then $x < A[i]$.

  $$A[i-1] < x < A[i].$$

  Because the array $A$ is sorted, $x$ can't be in $A$. Otherwise, it would be between two consecutive elements in the array.

- If $i > n$, then $i = n + 1$. Then $A[i-1] = A[n]$, and the loop invariant tells us that $A[n] < x$. Since $A$ is sorted, $x$ can't be in the list.

4. Argue that the loop actually does terminate.

   Whenever the loop executes successfully, $i$ increases by 1. This can't happen more than $n$ times.

Note that item 3 also helps us give us an analysis of the worst-case running time of a linear search. Each loop iteration does a constant amount of work, so the running time is $O(n)$.

# 3 Recursive Algorithm – Binary Search

A linear search does not take full advantage of the fact that the data is in order. The following algorithm does a binary search of the subarray, $A[a..b]$, for a key $x$.

```
1    BinarySearch(A[a .. b], x)
2       if (b < a) return false;
3
4       m <-- floor((a + b)/2);
5       if (x == A[m])
6           return true;
7       else if (x < A[m])
8           return BinarySearch(A[a .. m-1], x);
9       else
10          return BinarySearch(A[m+1 .. b], x);
```

## 3.1 Proving Correctness for a Recursive Algorithm

Often, for a recursive algorithm, we can just use induction more directly. For binary search, we can use induction on the length of the array $A[a..b]$ that we are searching.

## 3.2 Proof

To prove correctness, show that if the input satisfies the precondition, then the output of the algorithm satisfies the postcondition. The claim is that `BinarySearch`$(A[a..b], x)$ returns true if and only if $x = A[i]$ for some $i$ satisfying $a \leq i \leq b$, i.e., if and only if $x$ is in the subarray $A[a..b]$.

To prove correctness of a recursive algorithm, we can often use induction directly. In this case, we can do strong induction a variable $z$ that is size of the array being searched i.e., on $z = (b - a) + 1$.

- **Base Case**: If $z = 0$, then $b < a$. Then, $A[a..b]$ trivially has no elements. Thus, `false` is the correct answer for `BinarySearch`.

- **Inductive Step**:

  Assume that `BinarySearch` returns the correct answer for all $z \leq k$. Show that `BinarySearch` will be true for $z = k + 1$.

  We know that $k + 1 > 0$, so $b - a + 1 > 0$, or $a < b + 1$, or $a \leq b$.

  Consider the different cases for what can happen in lines 4 through 10.

– **Case 1**: $x = A[m]$.
It is always true that

$$a \leq m = \left\lfloor \frac{a+b}{2} \right\rfloor \leq b.$$

Thus, if $x = A[m]$, then `x` is in the subarray $A[a..b]$. Thus, line 6 returns the correct answer.
**Case 2**: $x < A[m]$.

* Suppose $x = A[i]$ for some $a \leq i \leq b$. According to the precondition that the list is sorted, $A[m] < A[m+1] < A[m+2] \ldots A[b]$. Thus, $x < A[i]$ for any $m \leq i \leq b$.
Therefore, $x = A[i]$ could only be true for some $i$ such that $a \leq i \leq m-1$.
* Suppose $x \neq A[i]$ for any $a \leq i \leq b$. Then certainly $x \neq A[i]$ for any $a \leq i \leq m-1$.

From this, we conclude that in this case `BinarySearch`$(A[a..b], x)$ is true if and only if $x$ is in the array $A[a..m-1]$. By the inductive hypothesis, this occurs exactly when `BinarySearch`$(A[a..m-1], x)$ is true (it can be shown that $(m-1-a)+1 < (b-a)+1$, i.e, we are recursively solving a smaller subproblem).

– **Case 3**: $x > A[m]$. This is symmetric to Case 2.
We know $A[a] \leq A[a+1] \leq A[a+2] \ldots A[m] < x$. Thus $x$ is in $A[a..b]$ if and only if $x$ is in $A[m+1..b]$. By the inductive hypothesis, this happens exactly when `BinarySearch`$(A[m+1..b], x)$ is true.

# 4 Horner's Rule

We can specify an $n$th degree polynomial specified by giving $A[i]$, the coefficient of the $x^i$ term. Thus, a sequence of coefficients $A[0], A[1], A[2], \ldots A[n]$ corresponds to a polynomial

$$P(x) = A[0] + A[1]x^1 + A[2]x^2 + \ldots A[n]x^n.$$

The problem is, given an array $A[0..n]$ and a value for $x$, compute $P(x)$.
Horner's rule can do this computation in $O(n)$ time, using $O(1)$ additional space.

```
1  Horner(A[0..n], x)
2      y <-- A[n];
3      i <-- n;
4      while (i > 0)
5          i <-- i - 1;
6          y <-- x*y + A[i];
7      return y
```

Follow the method for proving correctness of an iterative algorithm:

1. What is a good loop invariant $L$? Look at the value of $y$ after a few iterations:

$$\begin{aligned} i = n &\implies y = A[n] \\ i = n-1 &\implies y = A[n] \cdot x + A[n-1] \\ i = n-2 &\implies y = A[n] \cdot x^2 + A[n-1] \cdot x + A[n-2] \\ &\vdots \end{aligned}$$

Try a loop invariant

$$y = \sum_{j=i}^{n} A[j]x^{j-i}.$$

2. Proof the invariant holds by induction on $z$, the number of times the loop has run. Note that $z = n - i$.

   - **Base Case**: When $z = 0$, $i = n$, $y = A[n]$.
   - **Inductive Step**:
     Assume that the loop invariant $L$ holds after the $z$th iteration. Suppose that after the $z$th iteration, $i$ has value $k$. Then the $L$ guarantees that $y = \sum_{j=k}^{n} A[j]x^{j-k}$.
     During the $z+1$st iteration, $i$ changes to $k-1$, and we compute a new $y' = x * y + A[k-1]$. After the $z+1$st iteration, we have

$$\begin{aligned} y' &= x \cdot \left( \sum_{j=k}^{n} A[j]x^{j-k} \right) + A[k-1] \\ &= \left( \sum_{j=k}^{n} A[j]x^{j-k+1} \right) + A[k-1] \\ &= \left( \sum_{j=k}^{n} A[j]x^{j-(k-1)} \right) + A[k-1]x^0 \\ &= \sum_{j=k-1}^{n-1} A[j]x^{j-i} \end{aligned}$$

     Therefore, $L$ is true after the $z+1$st iteration.

3. When the loop terminates, $i = 0$, so $y = \sum_{j=0}^{n} x^j = P(x)$. This is exactly the value we were trying to compute.

4. The while loop always gets executed exactly $n$ times, as long as $n$ is nonnegative.