Reasoning About Correctness: Binary Search

```
// Pre: assumes data is ordered
// Post:  returns index position (low) such that all
// elements indexed less than low are smaller than val, and
// all elements indexed greater than low are greater than
// or equal to val

int _binarySearch(TYPE * data, int size, TYPE val) {
  int low  = 0;
  int high = size;
  int mid;
//(A)
  while (low < high) {
    mid = (low + high) / 2;
            //mid less than val looking for
    if (LT(data[mid],val))
                                    low  = mid + 1;
    else    high = mid;
//(B)
  }
//(C )
  return low;
}
```

Three important questions…

1. Is the result correct?
2. Does the algorithm terminate?
3. What is the execution time?

Is the result index , low, correct?

To be correct, the resulting index returned from Binary Search must satisfy the following:

1. The result index should be >= 0 and <= the number of elements in the array (ie. it can go anywhere inside the array or at the very end)
2. All elements in positions < the result index must be strictly < value at index
3. All elements in positions >= result index must be >= the value at index

To argue that it's correct, we will:

1. Establish assertions Before, Inside, After the loop (A,B, and C respectively in the code above)
2. Argue paths from each invariant to the next [Note: An assertion in a loop is an invariant]

Invariant#1
The elements at index less than low are themselves strictly less than the argument (argument is the value we're looking for): For all j<low, a[j] < val

Invariant #2
The elements at index >= high are themselves >= value: For all j >= high, a[j] >= val

Do Invariants Hold at A,B, and C?

1. Before the Loop (A)

Low is initialized to 0 and high is initialized to one larger than the very highest legal index (n+1)

Therefore, the sets described by the invariants are EMPTY so the invariants must hold!

2. In the loop (B)

Assume invariants 1&2 hold at the beginning of an iteration and we execute the body of the loop.

If the **a[mid] < val** (ie. condition is true), then all elements at positions **mid** and below must have values < argument so we can safely move **low** to **mid + 1** and still preserve invariant#1.

If the condition is false, then **a[mid] >= val**. Likewise, since these are sorted, the values at positions higher than **mid** must also be >= to val, so we can assign **high** to **mid** and still preserve invariant #2

(C) At the end of the Loop
If the loop executes 0 times, the invariant was true before so must be true after the loop
If the loop executes some number of times, then the invariants must have been true at the last iteration of the loop so they must still be true when we're done since we have not done anything additional after the loop. Thus, invariants 1&2 are still preserved.

So, We've established our invariants and demonstrated that they hold from (A) to (B) [ie. before loop to inside loop], from (B) to (B) [inside the loop] and from (B) to (C) when loop is done.

Finally, let's show that the resulting integer has the desired properties. To do so, we combine our invariants with the observation that when the loop terminates, **high <= low,** however, high can't possibly be less than low, **so high = low.**

Why can't high never be less than low???

1. Remember that mid must be strictly < high (but could be = low)
2. At worst, we have only two elements, low at 0 and high at 1. In this case, mid is 0 , so either low gets set to mid + 1 (or high) , or high get's set to mid (or low) so high= low.

Now, let's tie it all together…

If L=H, then 1, 2, and 3 MUST all be true.

1. The result index should be >= 0 and <= the number of elements in the array (ie. it can go anywhere inside the array or at the very end)
2. All elements in positions < the result index must be strictly < value at index
3. All elements in positions >= result index must be >= the value at index

Therefore we have found the index where either the element is…or where it should be inserted to maintain the sorted order of the array, and our post condition is held.

To Demonstrate Termination, we need to find a quantity that…

1. Is an integer value
2. Changes each iteration through the loop
3. Is decreasing
4. Is strictly positive

In this algorithm, we can't use **high** or **low** because only one of them is changed in each iteration. However, we can consider **high-low**

- It changes each iteration
- It is integer since we're using integer arithmetic
- Is always decreasing because we compute mid as **low+high/2** with integer arithmetic so it must be strictly less than high and we set low either to mid+1 or high to mid so we move low up or high down so high-low MUST decrease.

We've found a quantity that meets all criteria so it MUST terminate.

Execution Time

We divide the array roughly in half at each step. We can do this at most $\log_2 n$ times. So,)_binarySearch is O(logn) [Much, much better than O(n) linear searching….what is log2(1,000,000)?]

Bug?

In 2006, Google found a bug in the _binarySearch algorithm. Can you find it? If not, do some web research to get the story. Can you fix our argument such that it uncovers this bug?

Efficiency Feature

Finally, we can add one feature to potentially speed up our algorithm. That is, if the value is found, we can simply return the value ,terminating the algorithm. Can you make this change to the algorithm? Does it require any changes to the correctness argument?