15-122: Principles of Imperative Computation

Recitation 6 — Josh Zimmerman

Binary search

Binary search lets us search arrays *substantially* more quickly than linear search does.

The basic idea behind binary search is that if we're searching for `x`, we look in the middle of a sorted array and compare that element to `x`. If that element is smaller than `x`, we look in the top half of the array and if that element is bigger than `x`, we look in the bottom half of the array. (If that element is equal to `x`, then we're done.)

We're going to work through a few examples on the board to illustrate the number of steps binary search takes on arrays in practice, but the theoretical view is as follows: On every iteration of the loop, we roughly cut in half the amount of the array that we still have to look at—at every step, we throw out half what's left of the array.

So, we look at half of the array, and we then look at half of that, and so on. How many halvings will it take until we're looking at 1 element?

Here's the code for binary search. We're going to look at a proof of its correctness.

```
int binsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
      || ((0 <= \result && \result < n) && A[\result] == x);
 @*/
{
  int lower = 0;
  int upper = n;
  while (lower < upper)
  //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
  //@loop_invariant lower == 0 || A[lower-1] < x;
  //@loop_invariant upper == n || A[upper] > x;
  {
    int mid = lower + (upper-lower)/2;
    if (A[mid] < x) {
      // We can ignore the bottom half of the array now, since we
      // know that every thing in that half must be less than x
      lower = mid+1;
    } else if (A[mid] > x) {
      // We can ignore the upper half of the array, since we know
      // that everything in that half must be greater than x
      upper = mid;
    } else {
      //@assert A[mid] == x;
      return mid;
    }
  }
  //@assert lower == upper;
  return -1;
}
```

It's not immediately obvious from looking at this code that it works. So, let's prove that it does, by showing that the precondition implies the loop invariant will be true at the start of the first loop, that if the loop invariant is correct after one iteration of the loop it will be correct after the next iteration, that if the loop terminates and the loop invariants hold, then the postcondition holds, and that the loop does terminate.

Binary search for integer square root

Recall the linear search for integer square root function we discussed last recitation. Here's a function that binary searches instead. (Note: this function has a bug related to integer overflow with sufficiently large inputs.)

```
#use <util>
int bin_search_sqrt (int n)
//@requires n > 0;
//@ensures \result * \result <= n;
//@ensures n < (\result+1) * (\result+1) || (\result+1)*(\result+1) < 0;
//@ensures \result == isqrt(n);
{
  int lower = 1;
  int upper = n;
  int mid = upper/2;
  int mid_plus_one_square = (mid + 1) * (mid + 1);
  while (!(mid * mid <= n
       && ((mid_plus_one_square > n) || mid_plus_one_square < 0)))
  // Note that the <= is necessary here because isqrt rounds down.
  //@loop_invariant lower <= isqrt(n);
  //@loop_invariant upper >= isqrt(n);
  {
    mid = lower + (upper - lower)/2;
    int square = mid * mid; // Only compute once for efficiency
    if ((mid != 0 && mid >= int_max() / mid) || square > n) {
      upper = mid;
    }
    else if (square < n) {
      lower = mid;
    }
    else {
      //@assert mid * mid == n;
      return mid;
    }
    mid_plus_one_square = (mid + 1) * (mid + 1);
  }
  return mid;
}
```

Note the similarities between this and binary search on an array. If you consider an array `A` of all nonnegative integers, where the ith entry of the array is i^2, we're simply searching for the i such that `A[i] <= n && A[i + 1] > n`. We could implement the function this way, but it would be a large waste of memory.

If we take the square roots of n 5,000,000 times, the UNIX utility `time` reports the following:

n	Binary search time (s)	Linear search time (s)
10,000	1.091	1.413
20,000	1.300	1.934
30,000	1.322	2.374
40,000	1.161	2.863
50,000	1.217	3.187
60,000	1.443	3.447
70,000	1.446	3.777
80,000	1.450	4.032
90,000	1.462	4.292

As you can see, in practice linear search takes much longer than binary search and the amount of time linear search takes increases far more quickly than binary search (as number of elements we're taking the square root of goes up). We'll formalize the notion of linear search being slower than binary search next lecture when we talk about big-O notation.