Lecture Recap: Linear Search

(The is_in and is_sorted functions used here are the same as defined in class)

```c
int lin_search(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); @

    for (int i = 0; i < n; i++)
        //@loop_invariant 0 <= i && i <= n;
        //@loop_invariant !is_in(x, A, 0, i);
    {
        if (A[i] == x) return i; // We found what we were looking for!
        else if (x < A[i]) return -1; // Can’t possibly be to the right
        //@assert A[i] < x;
    }
    return -1;
}
```

Checkpoint 0

Work on your own or with other people to follow the four-step process to proving that linear search (the code above) works. Use the guidelines below:

**Solution:**

**Loop invariants hold initially**

Loop invariant 1: we initialize i to 0, so 0 <= i. By the precondition, 0 <= n, so i <= n initially as well.

Loop invariant 2: we initialize i to 0, so we’re checking to see if anything is in an empty chunk of the array. Nothing is, since it’s empty, so the loop invariant holds.

**Preservation of loop invariants**

Loop invariant 1: By the loop exit condition, i < n when we start the iteration, so when we exit the iteration, i + 1 == i’ <= n. Further, i’ > i >= 0, so i’ >= 0 (since i’ <= n, we know there wasn’t overflow)

Loop invariant 2: By the loop invariant, x \notin A[0...i]. If A[i] == x, we would have exited the loop on line 11. Thus, A[i] != x after we finish this iteration of the loop, so x \notin A[0,i+1). Since i’ == i + 1, we know that x \notin A[0,i’).
Loop invariants imply postcondition

There are several cases in which we can return. We need to address all of them.

Case 1: We return on line 11. In this case, we return a value which by the loop invariant is between 0 and n. Further, we know that $A[i] == x$ by the condition on line 11.

Thus, the second clause of the postcondition is satisfied, and so the postcondition is satisfied.

Case 2: We return on line 12. We know that $\text{result} == -1$, so we want to show $\neg \text{is_in}(x, A, 0, n)$. We know by the loop invariant that $\neg \text{is_in}(x, A, 0, i)$. Further, we know that $A[i] > x$, and that $A$ is sorted. Since $A$ is sorted, we know that everything in the segment $A[i, n)$ is also greater than $x$. Thus, $x$ is not in the array. We returned -1, so the first clause of the postcondition is satisfied.

Case 3: We return on line 15. In this case, we know we’ve exited the loop, so $i >= n$ by the negation of the loop guard and $i <= n$ by the loop invariant. Thus, $i == n$.

So, $\neg \text{is_in}(x, A, 0, i)$, which is equivalent to $\neg \text{is_in}(x, A, 0, n)$. Further, we return -1, so the first clause of the postcondition is satisfied.

Termination

The loop starts with $i$ being nonnegative. We increment $i$ once per iteration of the loop and terminate once $i >= n$, which must happen eventually since $0 <= n$.

Linear search for integer square root

We can apply the same concept of linear search to find the integer (since C0 doesn’t have floats!) square root of a given number. The integer square root of $n$ is defined to be the greatest non-negative integer $m$, such that $m^2 <= n$.

```
1 int isqrt (int n)
2 //@requires n >= 0;
3 //@ensures result * result <= n;
4 //@ensures n < (result+1) * (result+1) || (result+1) * (result+1) < 0;
5 {
6   int i = 0;
7   int k = 0;
8   while (0 <= k && k <= n)
9     //@loop_invariant i * i == k;
10    //@loop_invariant i == 0 || (i > 0 && (i-1)*(i-1) <= n);
11      {
12      // Note: (i + 1)*(i + 1) == i * i + 2*i + 1 and k == i * i
13        k = k + 2*i + 1;
14        i = i + 1;
15      }
16      // This subtraction is necessary since we know k > n now
17    // and i * i == k. i is barely too large to be the square root of n
18    return i - 1;
19  }
```

Note that this function is very similar to the linear search function we discussed. It’s essentially equivalent to searching through a sorted array containing all non-negative ints less than $n$, looking for the square root of $n$. There is a similar improved algorithm that we’ll discuss on Friday.
**Checkpoint 1**

Prove the correctness of this integer square root function

**Solution:**

**Loop invariants hold initially**

Loop invariant 1: we initialize \(i\) and \(k\) to 0, so \(i^2 = 0^2 = 0 = k\).

Loop invariant 2: we initialize \(i\) to 0, so the first part of the or is true, therefore the loop invariant holds.

**Preservation of loop invariants**

Loop invariant 1: \(k' = k + 2i + 1\). By the loop invariant, this is \(i^2 + 2i + 1 = (i + 1)^2 = i'^2\).

Loop invariant 2: By the loop invariant, \(i = 0\) \&\& \((i > 0 \&\& (i - 1)^2 \leq n)\). In the first case, \(i = 0\). Thus \(i' = 1\), so \(i > 0\) and \((i - 1)^2 = 0^2 = 0 \leq n\) by the precondition, so the loop invariant holds.

In the second case \(i > 0\) and \((i - 1)^2 \leq n\) by the loop invariant. We know by the loop condition that \(k \leq n\), and by the first loop invariant, we know that \(i^2 = (i' - 1)^2 = k\), thus by transitivity, \((i' - 1)^2 \leq n\).

**Loop invariants imply postcondition**

Post-condition 1: We return \(i - 1\). By the negation of the loop guard, we know that \(k > n\), and by the loop invariant, we know that \(i^2 = k\). Thus, \(i^2 > n \implies i \neq 0\). Thus, by the second loop invariant, we know that \((i - 1)^2 \leq n\).

Post-condition 2: We return \(i - 1\). By the negation of the loop guard we have \(k > n\), and by the loop invariant we have \(i^2 = k\), thus \((i - 1 + 1)^2 = i^2 > n\).

**Termination**

At each iteration, we have \(k' = k + 2i + 1\). From the second loop invariant, we know that \(i\) is always non-negative, so \(k\) is always increasing. Thus \(n - k\) is decreasing, and once it reaches 0, we will terminate.

**Checkpoint 2**

A water main break in GHC has, confusingly, broken the C0 compiler’s -d option! C0 contracts are now being treated as comments, and the only way to generate assertion failures is with the `assert()` statements.

Insert `assert()` statements into the code below so that, when the code runs, all operations (C0 statements, conditional checks, and assertions) are performed at runtime in the exact same sequence that would have occurred if we compiled with -d. Not all of the blanks need to be filled in.
int mult(int x, int y)
//@requires x >= 0 && y >= 0;
//@ensures \result == x*y;
{
/* 1 */
   /* 1 */ CHECK((x >= 0) && (y >= 0));
int k = x; int n = y;
int res = 0;

/* 2 */
   /* 2 */ CHECK(x * y == (k * n + res));
while (n != 0)
   //@loop_invariant x * y == k * n + res;
   {
      /* 3 */
         /* 3 */ // do nothing here!
      if ((k & 1) == 1) res = res + n;
      k = k >> 1;
      n = n << 1;
      /* 4 */
         /* 4 */ CHECK(x * y == (k * n + res));
   }
/* 5 */
   /* 5 */ // do nothing here!
/* 6 */
   /* 6 */ CHECK(res == x * y);
return res;
/* 7 */
   /* 7 */ // do nothing here!
}
int main() {
   int a;
   /* 8 */
      /* 8 */ // do nothing here!
a = mult(3,4);
   /* 9 */
      /* 9 */ // do nothing here!
   return a;
}