Lecture Notes on Linear Search

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 5
January 29, 2013

1 Introduction

One of the fundamental and recurring problems in computer science is to find elements in collections, such as elements in sets. An important algorithm for this problem is *binary search*. We use binary search for an integer in a sorted array to exemplify it. As a preliminary study in this lecture we analyze *linear search*, which is simpler, but not nearly as efficient. Still it is often used when the requirements for binary search are not satisfied, for example, when we do not have the elements we have to search arranged in a sorted array.

In term of our learning goals, we discuss the following:

Computational Thinking: We will see the first time the power of *order* in various algorithmic problems.

Algorithms and Data Structures: We will see a simple linear search in a fixed-size array.

Programming: We will practice *deliberate programming* together in lectures. We also emphasize the importance of contracts for testing and reasoning, both about safety and correctness.

2 Linear Search in an Unsorted Array

If we are given an array of integers A without any further information and have to decide if an element x is in A, we just have to search through it,

element by element. We return `true` as soon as we find an element that equals x, `false` if no such element can be found.

```
bool is_in(int x, int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
{
  for (int i = lower; i < upper; i++)
    //@loop_invariant lower <= i && i <= upper;
    {
      if (A[i] == x) return true;
    }
  return false;
}
```

We used the statement `i++` which is equivalent to `i = i+1` to step through the array, element by element.

The precondition is very common when working with arrays. We pass an array, and we also pass bounds – typically we will let $lower$ be 0 and $upper$ be the length of the array. The added flexibility of allowing $lower$ and $upper$ to take other values will be useful if we want to limit search to the first n elements of an array and do not care about the others. It will also be useful later to express invariants such as *x is not among the first k elements of A*, which we will write in code as `!is_in(x, A, 0, k)` and which we will write in mathematical notation as $x \notin A[0, k)$.

The loop invariant is also typical for loops over an array. We examine every element (i ranges from $lower$ to $upper-1$). But we will have $i = upper$ after the last iteration, so the loop invariant which is checked *just before the exit condition* must allow for this case.

Could we strengthen the loop invariant, or write a postcondition? We could try something like

```
//@loop_invariant !is_in(x, A, lower, i);
```

where `!b` is the negation of b. However, it is difficult to make sense of this use of recursion in a contract or loop invariant so we will avoid it.

This is small illustration of the general observation that some functions are basic specifications and are themselves not subject to further specification. Because such basic specifications are generally very inefficient, they are mostly used in other specifications (that is, pre- or post-conditions, loop invariants, general assertions) rather than in code intended to be executed.

3 Sorted Arrays

A number of algorithms on arrays would like to assume that they are sorted. Such algorithms would return a correct result only if they are actually running on a sorted array. Thus, the first thing we need to figure out is how to specify sortedness in function specifications. The specification function `is_sorted(A,lower,upper)` traverses the array A from left to right, starting at $lower$ and stopping just before reaching $upper$, checking that each element is smaller or equal to its right neighbor. We need to be careful about the loop invariant to guarantee that there will be no attempt to access a memory element out of bounds.

```
bool is_sorted(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
{
  for (int i = lower; i < upper-1; i++)
    //@loop_invariant lower <= i;
    if (!(A[i] <= A[i+1])) return false;
  return true;
}
```

The loop invariant here does not have an upper bound on i. Fortunately, when we are inside the loop, we know the loop condition is true so we know $i < upper - 1$. That together with $lower \leq i$ guarantees that *both* accesses are in bounds.

We could also try $i \leq upper - 1$ as a loop invariant, but this turns out to be false. It is instructive to think about why. If you cannot think of a good reason, try to prove it carefully. Your proof should fail somewhere.

Actually, the attempted proof already fails at the initial step. If $lower = upper = 0$ (which is permitted by the precondition) then it is *not* true that $0 = lower = i \leq upper - 1 = 0 - 1 = -1$. We could say $i \leq upper$, but that wouldn't seem to serve any particular purpose here since the array accesses are already safe.

Let's reason through that. Why is the acccess $A[i]$ safe? By the loop invariant $lower \leq i$ and the precondition $0 \leq lower$ we have $0 \leq i$, which is the first part of safety. Secondly, we have $i < upper - 1$ (by the loop condition, since we are in the body of the loop) and $upper \leq \mathsf{length}(A)$ (by the precondition), so i will be in bounds. In fact, even $i + 1$ will be in bounds, since $0 \leq lower \leq i < i + 1$ (since i is bounded from above) and $i + 1 < (upper - 1) + 1 = upper \leq \mathsf{length}(A)$.

Whenever you see an array access, you must have a very good reason why the access must be in bounds. You should develop a coding instinct where you *deliberately pause* every time you access an array in your code and verify that it should be safe according to your knowledge at that point in the program. This knowledge can be embedded in preconditions, loop invariants, or assertions that you have verified.

4 Linear Search in a Sorted Array

Next, we want to search for an element x in an array A which we know is sorted in ascending order. We want to return -1 if x is not in the array and the index of the element if it is.

The pre- and postcondition as well as a first version of the function itself are relatively easy to write.

```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A,0,n);
/*@ensures (\result == -1 && !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;
   if (A[i] == x) return i;
  return -1;
}
```

This does not exploit that the array is sorted. We would like to exit the loop and return -1 as soon as we find that $A[i] > x$. If we haven't found x already, we will not find it subsequently since all elements to the right of i will be greater or equal to $A[i]$ and therefore strictly greater than x. But we have to be careful: the following program has a bug.

```
int 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; A[i] <= x && i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    if (A[i] == x) return i;
  return -1;
}
```

Can you spot the problem? If you cannot spot it immediately, reason through the loop invariant. Read on if you are confident in your answer.

The problem is that the loop invariant only guarantees that $0 \leq i \leq n$ before the exit condition is tested. So it is possible that $i = n$ and the test `A[i] <= x` will try access an array element out of bounds: the n elements of A are numbered from 0 to $n - 1$.

We can solve this problem by taking advantage of the so-called *short-circuiting evaluation* of the boolean operators of conjunction ("and") `&&` and disjunction ("or") `||`. If we have condition `e1 && e2` (e_1 *and* e_2) then we do not attempt to evaluate e_2 if e_1 is `false`. This is because a conjunction will always be false when the first conjunct is false, so the work would be redundant.

Similarly, in a disjunction `e1 || e2` (e_1 *or* e_2) we do not evaluate e_2 if e_1 is `true`. This is because a disjunction will always be true when the first disjunct it true, so the work would be redundant.

In our linear search program, we just swap the two conjuncts in the exit test.

```
int 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 && A[i] <= x; i++)
    //@loop_invariant 0 <= i && i <= n;
   if (A[i] == x) return i;
  return -1;
}
```

Now `A[i] <= x` will only be evaluated if $i < n$ and the access will be in bounds since we also know $0 \leq i$ from the loop invariant.

Alternatively, and perhaps easier to read, we can move the test into the loop body.

```
int 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;
    {
      if (A[i] == x) return i;
      else if (A[i] > x) return -1;
    }
  return -1;
}
```

This program is not yet satisfactory, because the loop invariant does not have enough information to prove the postcondition. We *do* know that if we return directly from inside the loop, that $A[i] = x$ and so `A[\result] == x` holds. But we cannot deduce that `!is_in(x, A, 0, n)` if we return -1.

Before you read on, consider which loop invariant you might add to guarantee that. Try to reason why the fact that the exit condition must be false and the loop invariant true is enough information to know that `!is_in(x, A, 0, n)` holds.

Did you try to exploit that the array is sorted? If not, then your invariant is most likely too weak, because the function is incorrect if the array is not sorted!

What we want to say is that *all elements in A to the left of index i are smaller than x*. Just saying `A[i-1] < x` isn't quite right, because when the loop is entered the first time we have $i = 0$ and we would try to access $A[-1]$. We again exploit shirt-circuiting evaluation, this time for disjunction.

```
int 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 i == 0 || A[i-1] < x;
    {
      if (A[i] == x) return i;
      else if (A[i] > x) return -1;
      //@assert A[i] < x;
    }
  return -1;
}
```

It is easy to see that this invariant is preserved. Upon loop entry, $i = 0$. Before we test the exit condition, we just incremented i. We did not return while inside the loop, so $A[i-1] \neq x$ and also $A[i-1] \leq x$. From these two together we have $A[i-1] < x$. We have added a corresponding assertion to the program to highlight the importance of that fact.

Why does the loop invariant imply the postcondition of the function? If we exit the loop normally, then the loop condition must be false. So $i \geq n$. know $A[n-1] = A[i-1] < x$. Since the array is sorted, all elements from 0 to $n-1$ are less or equal to $A[n-1]$ and so also strictly less than x and x can not be in the array.

If we exit from the loop because $A[i] > x$, we also know that $A[i-1] < x$ so x cannot be in the array since it is sorted.

5 Analyzing the Number of Operations

In the worst case, linear search goes around the loop n times, where n is the given bound. On each iteration except the last, we perform three comparisons: $i < n$, $A[i] = x$ and $A[i] > x$. Therefore, the number of comparisons is almost exactly $3 * n$ in the worst case. We can express this by saying that the running time is *linear* in the size of the input (n). This allows us to predict the running time pretty accurately. We run it for some reasonably large n and measure its time. Doubling the size of the input $n' = 2 * n$ mean that now we perform $3 * n' = 3 * 2 * n = 2 * (3 * n)$ operations, twice as many as for n inputs.

We will introduce more abstract measurements for the running times in the lecture after next.