Lecture Notes on
Linear Search

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 5
January 25, 2011

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.

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 not such element can be found.

```
bool is_in(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
{ int i;
  for (i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    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 n which we generally think of as its length. However, for the test to work correctly it need not be the length—it could be smaller. That's useful if we want to express an invariant such as *x is not among the first k elements of A*, which would be `!is_in(x, A, k)`.

The loop invariant is also typical for loops over an array. We examine every element (i ranges from 0 to $n-1$). But we will have $i = n$ 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, 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. We begin with a specification of this property. The function `is_sorted(A,n)` traverses the array A from left to right, 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 n)
//@requires 0 <= n && n <= \length(A);
{
  for (int i = 0; i < n-1; i++)
    //@loop_invariant n == 0 || (0 <= i && i <= n-1);
    if (!(A[i] <= A[i+1])) return false;
  return true;
}
```

The loop invariant here is a disjunction: either $n = 0$ or i is between 0 and $n - 1$. This is a simple *or* on boolean values (`true` and `false`) and not an *exclusive or*, even though we will often pronounce it as *either* . . . *or* . . ..

Why is it necessary? If we ask if a zero-length array is sorted, then before the check the exit condition of the loop the first time we have $i = 0$ and $n = 0$, so it is not the case that $i \leq n - 1$. Therefore the second part of the loop invariant does not hold. We account for that explicitly by allowing n to be 0.

For an example of reasoning with loop invariants, we verify in some detail why this is a valid loop invariant.

Initially: Upon loop entry, $i = 0$. We distinguish two cases. If $n = 0$, then the left disjunction `n == 0` holds. If $n \neq 0$ then $n > 0$ because the precondition of the function requires $n \geq 0$. But if $n > 0$ and $i = 0$ then $i \leq n - 1$. We also have $0 \leq i$ so `0 <= i && i <= n-1` holds.

Preservation: Assume the loop invariant holds before the test, so either $n = 0$ or $0 \leq i \leq n - 1$. Because we do not exit the loop, we also have $i < n - 1$. The step statement in the loop increments i so we have $i' = i + 1$.

Since $i' = i + 1$ and $0 \leq i$ we have $0 \leq i'$. Also, since $i < n - 1$ and $i' = i + 1$ we have $i' - 1 < n - 1$ and so $i' < n$. Therefore $i' \leq n - 1$.

So $0 \leq i' \leq n - 1$ and the loop invariant is still satisfied because the right disjunct is true for the new value i' of i.

One pedantic point (and we *do* want to be pedantic in this class when assessing function correctness, just like the machine is): from $0 \leq i$ and $i' = i + 1$ we inferred $0 \leq i'$. This is only justified in modular arithmetic if we know that $i + 1$ does not overflow. Fortunately, we also know $i < n - 1$, so $i < n$ and i is bounded from above by a positive integer. Therefore incrementing i cannot overflow.

We generally do not verify loop invariants in this amount of detail, but it is important for you do know how to reason attentively through loop invariants to uncover errors, be they in the program or in the loop invariant itself.

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 linsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A,n);
/*@ensures (\result == -1 && !is_in(x, A, 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 linsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A,n);
/*@ensures (-1 == \result && !is_in(x, A, n))
        || ((0 <= \result && \result < n) && A[\result] == x);
  @*/
{ int i;
  for (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 linsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A,n);
/*@ensures (-1 == \result && !is_in(x, A, 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 linsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, n);
/*@ensures (-1 == \result && !is_in(x, A, 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, 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, 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 linsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, n);
/*@ensures (-1 == \result && !is_in(x, A, 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;
    }
  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 $A[i-1] < x$.

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 next lecture.