Lecture Notes on
Binary Search

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 3
August 31, 2010

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.

Binary search is the first time we see the fundamental principle of *divide-and-conquer*. We will see many other examples of this important principle in future lectures. It refers to the idea of subdividing a given problem into smaller components, each of which can be solved separately. We then combine the results to obtain a solution to the original problem.

We will also once again see the importance of loop invariants in writing correct code. Here is a note by Jon Bentley about binary search:

> *I've assigned [binary search] in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert [its] description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: ninety percent of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found).*
>
> *I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right. But*

they aren't the only ones to find this task difficult: in the history in Section 6.2.1 of his Sorting and Searching, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

—Jon Bentley, Programming Pearls *(1st edition), pp.35–36*

I contend that what these programmers are missing is the understanding of how to use loop invariants in composing their programs. They help us to make assumptions explicit and clarify the reasons *why* a particular program is correct. Part of the magic of pre- and post-conditions as well as loop invariants and assertions is that they *localize* reasoning. Rather than having to look at the whole program, or the whole function, we can focus on individual statements tracking properties via the loop invariants and assertions.

Before we introduce binary search, we discuss linear search.

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. If we have gone through the whole array and still not found such an element, we return `false`.

```
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;
}
```

Could we strengthen the loop invariant, or write a postcondition? For the loop invariant, we might want to express that x is not in $A[0] \ldots A[i-1]$. We would have to express this with 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 because it uses `is_in`, which is the very function we are defining!

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);
{ int i;
  for (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 valid loop invariant.

Loop Entry: 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.

Loop Traversal: 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)) || A[\result] == x;
{ int i;
  for (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 (\result == -1 && !is_in(x, A, 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 (\result == -1 && !is_in(x, A, n)) || A[\result] == x;
{ int i;
  for (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.

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 (\result == -1 && !is_in(x, A, n)) || A[\result] == x;
{ int i;
  for (i = 0; i < n && A[i] <= x; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant i == 0 || A[i-1] < x;
    if (A[i] == x) return i;
  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 $A[i-1] \neq x$. Also, $A[i-1] \leq x$ because the loop condition was `true` on the prior iteration. 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 either $i \geq n$ or $i < n$ and $A[i] > x$. In the first case, we 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.

In the second case ($A[i] > x$) we also know from the loop invariant that either $i = 0$ or $A[i-1] < x$. Because the array is sorted, if $A[0] > x$ then x cannot be in the array. Also, again because the array is sorted, if $A[i-1] < x$ and $A[i] > x$ then x cannot be in the array.

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: $i < n$, $A[i] \leq x$ 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.

6 Binary Search

Can we do better than searching through the array linearly? If you don't know the answer already it might be surprising that, yes, we can do *significantly* better! Perhaps almost equally surprising is that the code is almost as short!

Before we write the code, let us describe the algorithm. We start by examining the *middle element* of the array. If it smaller than x than x must be in the upper half of the array (if it is there at all); if is greater than x then it must be in the lower half. Now we continue by restricting our attention to either the upper or lower half, again finding the middle element and proceeding as before.

We stop if we either find x, or if the size of the subarray shrinks to zero, in which case x cannot be in the array.

Before we write a program to implement this algorithm, let us analyze the running time. Assume for the moment that the size of the array is a power of 2, say 2^k. Each time around the loop, when we examine the middle element, we cut the size of the subarrays we look at in half. So before the first iteration the size of the subarray of interest is 2^k. After the second iteration it is of size 2^{k-1}, then 2^{k-2}, etc. After k iterations it will be $2^{k-k} = 1$, so we stop after the next iteration. Altogether we can have at most $k + 1$ iterations. Within each iteration, we perform a constant amount of work: computing the midpoint, and a few comparisons. So, overall, when given a size of array n we perform $c * \log_2(n)$ operations.[1]

If the size n is not a power of 2, then we can round n up to the next power of 2, and the reasoning above still applies. The actual number of

[1] In general in computer science, we are mostly interested in logarithm to the base 2 so we will just write $\log(n)$ for log to the base 2 from now on unless we are considering a different base.

steps can only be smaller than this bound, because some of the actual subintervals may be smaller than the bound we obtained when rounding up n.

The logarithm grows much slower than the linear function that we obtained when analyzing linear search. As before, consider that we are doubling the size of the input, $n' = 2*n$. Then the number of operations will be $c*\log(2*n) = c*(\log(2)+\log(n)) = c*(1+\log(n)) = c + c*\log(n)$. So the number of operations increases only by a constant amount c when we double the size of the input. Considering that the largest representable positive number in two's complement representation is $2^{31} - 1$ (about 2 billion) binary search even for unreasonably large arrays will only traverse the loop 31 times! So the maximal number of operations is effectively bounded by a constant if it is logarithmic.

7 Implementing Binary Search

The specification for binary search is the same as for linear search.

```
int binsearch(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)) || A[\result] == x;
  ;
```

We have two variables, `lower` and `upper`, which hold the lower and upper end of the subinterval in the array that we are considering. We start with `lower` as 0 and `upper` as n, so the interval includes `lower` and excludes `upper`. This often turns out to be a convenient choice when computing with arrays.

The `for` loop from linear search becomes a `while` loop, exiting when the interval has size zero, that is, `lower == upper`. We can easily write the first loop invariant, relating `lower` and `upper` to each other and the overall bound of the array.

```
int binsearch(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)) || A[\result] == x;
{ int lower = 0;
  int upper = n;
  while (lower < upper)
    //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
   {
    // ...??...
   }
  return -1;
}
```

In the body of the loop, we first compute the midpoint `mid`. Then we check if $A[mid] = x$. If so, we have found the element and return mid.

```
int binsearch(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)) || A[\result] == x;
```

```
{ int lower = 0;
  int upper = n;
  while (lower < upper)
    //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
    //@loop_invariant ...??...
    { int mid = (lower + upper)/2;
      if (A[mid] == x) return mid;
      // ...??...
    }
  return -1;
}
```

Now comes the hard part. What is the missing part of the invariant? The first instinct might be to say that x should be in the interval from $A[lower]$ to $A[upper]$. But that may not even be true when the loop is entered the first time. Looking back at linear search we notice that the invariant was somewhat different: we expressed that x could not be outside of the chosen interval. We say that here by saying that $A[lower - 1] < x$ and $A[upper] > x$. The asymmetry arises because the interval under consideration includes $A[lower]$ but excludes $A[upper]$.

As in linear search, we have to worry about the boundary condition when $lower = 0$ or $upper = n$, in which case we have not yet excluded any part of the array. And, again, we use disjunction and exploit short-circuit evaluation to put these together.

```
int binsearch(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)) || 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)/2;
      if (A[mid] == x) return mid;
      // ...??...
   }
  return -1;
}
```

At this point, let's check if the loop invariant is strong enough to imply the postcondition of the function. If we return from inside the loop because $A[mid] = x$ we return mid, so `A[\result] == x` as required.

If we exit the loop because $lower < upper$ is false, we know $lower = upper$, by the first loop invariant. Now we have to distinguish some cases.

1. If $A[lower-1] < x$ and $A[upper] > x$, then $A[lower] > x$ (since $lower = upper$). Because the array is sorted, x cannot be in it.

2. If $lower = 0$, then $upper = 0$. By the second conjunct, then either $n = 0$ (and so the array has no elements and we must return -1), or $A[upper] = A[lower] = A[0] > x$. Because A is sorted, x cannot be in A if its first element is already strictly greater than x.

3. If $upper = n$, then $lower = n$. By the first conjunct, then either $n = 0$ (and so we must return -1), or $A[n - 1] = A[upper - 1] = A[lower - 1] < x$. Because A is sorted, x cannot be in A if its last element is already strictly less than x.

Notice that we could verify all this without even knowing the complete program! As long as we can finish the loop to preserve the invariant and terminate, we will have a correct implementation! This would again be a good point for you to interrupt your reading and to try to complete the loop, reasoning from the invariant.

We have already tested if $A[mid] = x$. If not, then $A[mid]$ must be less or greater than x. If it is less, then we can keep the upper end of the interval as is, and set the lower end ot $mid + 1$. Now $A[lower - 1] < x$ (because $A[mid] < x$ and $lower = mid + 1$), and the condition on the upper end remains unchanged.

If $A[mid] > x$ we can set *upper* to *mid* and keep *lower* the same. We do not need to test this last condition, because the fact the tests $A[mid] = x$ and $A[mid] < x$ both failed implies that $A[mid] > x$. We note this in an assertion.

```
int binsearch(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)) || 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)/2;
      if (A[mid] == x) return mid;
      else if (A[mid] < x) lower = mid+1;
      else /*@assert(A[mid] > x);@*/ upper = mid;
    }
  return -1;
}
```

Does this function terminate? If proceed to the loop body, that is, $lower < upper$, then the interval from *lower* to *upper* is non-empty. Moreover, the intervals from *lower* to *mid* and from $mid + 1$ to *upper* are both strictly smaller than the original interval. Unless we find the element, the difference between *upper* and *lower* must eventually become 0 and we exit the loop.

8 One More Bug

The function so far, even though we "proved" it correct, still contains a bug. Again, consider the function and see if you can spot the problem. Do not give up too early!

Where you able to see it? It's subtle, but somewhat similar to the problem we had in our very first example, the integer square root. Where we compute

```
int mid = (lower + upper)/2;
```

we could actually have an overflow, if $lower + upper > 2^{31} - 1$. This is somewhat unlikely in practice, since $2^{31} = 2G$, about 2 billion, so the array would have to have at least 1 billion elements. This is not impossible, and, in fact, a bug like this in the Java libraries[2] was actually exposed.

Fortunately, the fix is simple: because $lower < upper$, we know that $upper - lower > 0$ and represents the size of the interval. So we can divide that in half and add it to the lower end of the interval to get its midpoint.

```
int binsearch(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)) || 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; // (lower + upper)/2 could overflow
      if (A[mid] == x) return mid;
      else if (A[mid] < x) lower = mid+1;
      else /*@assert(A[mid] > x);@*/ upper = mid;
    }
  return -1;
}
```

Other operations in this program take place on quantities bounded from above by n and thus cannot overflow.

9 Some Measurements

Algorithm design is an interesting mix between mathematics and an experimental science. Our analysis above, albeit somewhat preliminary in

[2]see Joshua Bloch's Extra, Extra blog entry

nature, allow us to make some predictions of running times of our implementations. We start with linear search. We first set up a file to do some experiments. We assume we have already tested our functions for correctness, so only timing is at stage. See the file find-time.c0 on the course web pages. We compile this file, together with the our implementation from this lecture and a random number generator with the `cc0` command below. We can get an overall end-to-end timing with the Unix `time` command.

```
% cc0 -lconio rand.h0 lcg.c0 find.c0 find-time.c0
% time ./a.out
```

When running linear search 2000 times (1000 elements in the array and 1000 random elements) on 2^{18} elements (256 K elements) we get the following answer

```
Timing 1000 times with 2^18 elements
0
4.602u 0.015s 0:04.63 99.5% 0+0k 0+0io 0pf+0w
```

which indicates `4.602` seconds of user time.

Running linear search 2000 times on random arrays of size 2^{18}, 2^{19} and 2^{20} we get the timings on our MacBook Pro

array size	time (secs)
2^{18}	4.602
2^{19}	9.027
2^{20}	19.239

The running times are fairly close to doubling consistently. Due to memory locality effects and other overheads, for larger arrays we would expect larger numbers.

Running the same experiments with binary search we get

array size	time (secs)
2^{18}	0.020
2^{19}	0.039
2^{20}	0.077

which is much, much faster but looks suspicously linear as well.

Reconsidering the code we see that the time might increase linearly because we actually must iterate over the whole array in order to initialize it with random elements!

We comment out the testing code to measure only the initialization time, and we see that for 2^{20} elements we measure 0.072 seconds, as compared to 0.077 which is insignificant. Effectively, we have been measuring the time to set up the random array, rather than to find elements in it with binary search!

This is a vivid illustration of the power of divide-and-conquer. Logarithmic running time for algorithms grow very slowly, a crucial difference to linear-time algorithms when the data sizes become large.