Lecture Notes on Binary Search

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 6
January 31, 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. We started in the last lecture by discussing *linear search* and giving some background on the problem. This lecture clearly illustrates the power of *order* in algorithm design: if an array is sorted we can search through it very efficiently, much more efficiently than when it is not ordered.

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.

2 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. For example, is $n = 13$ we round it up to $16 = 2^4$. The actual number of 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.

3 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, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
        || ((0 <= \result && \result < n) && A[\result] == x);
  @*/
  ;
```

We declare 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 (but see Exercise 1).

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

[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.

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, 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;
   {
    // ...??...
   }
  return -1;
}
```

In the body of the loop, we first compute the midpoint `mid`. By elementary arithmetic it is indeed between $lower$ and $upper$.

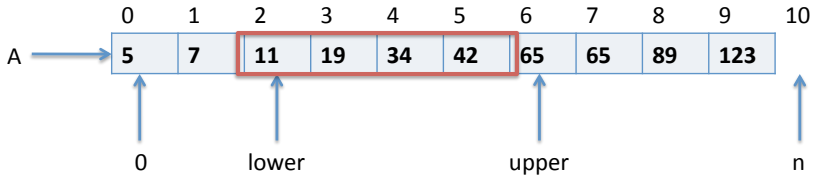
Next in the loop body we check if $A[mid] = x$. If so, we have found the element and return mid.

```
int binsearch(int x, int[] A, int n)
// ... contract elided ...
{ int lower = 0;
  int upper = n;
  while (lower < upper)
    //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
    //@loop_invariant ...??...
      int mid = lower + (upper-lower)/2;
      //@assert lower <= mid && mid < upper;
      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

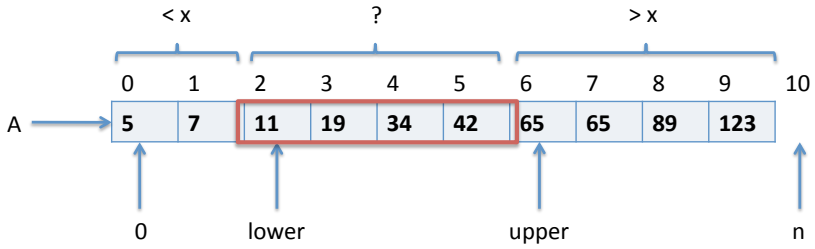
$A[lower]$ to $A[upper]$. But that may not even be true when the loop is entered the first time.

Let's consider a generic situation in the form of a picture and collect some ideas about what might be appropriate loop invariants. Drawing diagrams to reason about an algorithm and the code that we are trying to construct is an extremely helpful general technique.



The red box around elements 2 through 5 marks the segment of the array still under consideration. This means we have *ruled out* everything to the right of (and including) $upper$ and to the left of (and not including) $lower$. Everything to the left is ruled out, because those values have been recognized to be strictly less than x, while the ones on the right are known to be strictly greater than x, while the middle is still unknown.

We can depict this as follows:



We can summarize this by stating that $A[lower-1] < x$ and $A[upper] > x$. This implies that x cannot be in the segments $A[0..lower)$ and $A[upper..n)$ because the array is sorted (so all array elements to the left of $A[lower-1]$ will also be less than x and all array elements to the right of $A[upper]$ will also be greater than x). For an alternative, see Exercise 2.

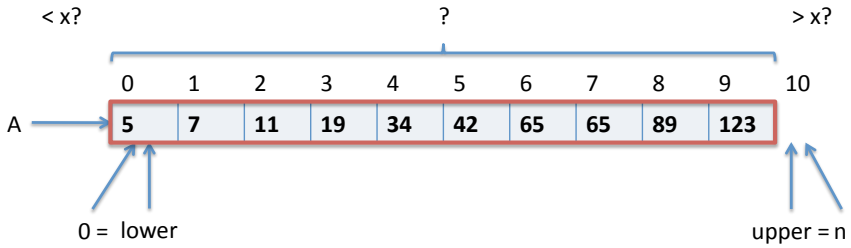
We can postulate these as invariants in the code.

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

Now a very powerful programming instinct should tell you someting is fishy. Can you spot the problem with the new invariants even before writing any more code in the body of the loop?

Whenever you access an element of an array, you must have good reason to know that the access will be in bounds!

In the code we blithely wrote $A[lower - 1]$ and $A[upper]$ because they were in the middle of the array in our diagram. But initially (and potentially through many iterations) this may not be the case. Fortunately, it is easy to fix, following what we did for linear search. Consider the following picture when we start the search.



In this case all elements of the array have to be considered candidates. All elements strictly to the left of 0 (of which there are none) and to the right of n (of which there are none) have been ruled out. As in linear search, we can add this to the our invariant using disjunction.

```
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) 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 third loop invariant, 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 second loop invariant, 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 to $mid + 1$. 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 that 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, 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;
      //@assert lower <= mid && mid < upper;
      if (A[mid] == x) return mid;
      else if (A[mid] < x) lower = mid+1;
      else /*@assert(A[mid] > x);@*/
        upper = mid;
    }
  return -1;
}
```

Let's set up the proof of the loop invariants more schematically.

Init: When the loop is first reached, we have $lower = 0$ and $upper = n$, so the first loop invariant follows from the precondition to the function. Furthermore, the first disjunct in loop invariants two (`lower == 0`) and three (`upper == n`) is satisfied.

Preservation: Assume the loop invariants are satisfied and we enter the loop:

$0 \leq lower \leq upper \leq n$	(Inv 1)
$(lower = 0$ or $A[lower - 1] < x)$	(Inv 2)
$(upper = n$ or $A[upper] < x)$	(Inv 3)
$lower < upper$	(loop condition)

We compute $mid = lower + \lfloor (upper - lower)/2 \rfloor$. Now we distinguish three cases:

$A[mid] = x$**:** In that case we exit the function, so we don't need to show preservation. We do have to show the postcondition, but we already considered this earlier in the lecture.

$A[mid] < x$**:** Then

$$lower' = mid + 1$$
$$upper' = upper$$

The first loop invariant $0 \leq lower' \leq upper' \leq n$ follows from the formula for mid, our assumptions, and elementary arithmetic.

For the second loop invariant, we calculate:

$A[lower' - 1]$	$=$	$A[(mid + 1) - 1]$	since $lower' = mid + 1$
	$=$	$A[mid]$	by arithmetic
	$<$	x	this case ($A[mid] < x$)

The third loop invariant is preserved, since $upper' = upper$.

$A[mid] > x$**:** Then

$$lower' = lower$$
$$upper' = mid$$

Again, by elementary arithmetic, $0 \leq lower' \leq upper' \leq n$.

The second loop invariant is preserved since $lower' = lower$.

For the third loop invariant, we calculate

$A[upper']$	$=$	$A[mid]$	since $upper' = mid$
	$>$	x	since we are in the case $A[mid] > x$

4 Termination

Does this function terminate? If the loop body executes, 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.

5 One More Observation

You might be tempted to calculate the midpoint with

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

but that is in fact incorrect. Consider this change and try to find out why this would introduce a bug.

Were you able to see it? It's subtle, but somewhat related to other problems we had. When we compute `(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 mid = lower + (upper-lower)/2;   // as shown in binary search
//@assert lower <= mid && mid < upper;
```

Let us convince ourselves why the assert is correct. The division by two will round to zero, which will round down to 0 here, because $upper - lower > 0$. Thus, $0 \leq (upper - lower)/2 < upper - lower$, because dividing a positive number by two will make it strictly smaller. Hence,

$$mid = lower + (upper - lower)/2 < lower + (upper - lower) = upper$$

Since dividing positive numbers by two will still result in a nonnegative number, the first part of the assert is correct as well.

$$mid = lower + (upper - lower)/2 \geq lower + 0 = lower$$

Other operations in this binary search take place on quantities bounded from above by the `int` n and thus cannot overflow.

Why did we choose to look at the middle element and not another element at all? Because, whatever the outcome of our comparison to that middle element may be, we maximize how much we have learned about the contents of the array by doing this one comparison. If we find the element, we are happy because we are done. If the middle element is smaller than what we are looking for, however, we are happy as well, because we have just learned that the lower half of the array has become irrelevant. Similarly, if the middle element is bigger, then we have made substantial progress by learning that we never need to look at the upper half of the array anymore. There are other choices, however, where binary search will also still work in essentially the same way.

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

6 Some Measurements

Algorithm design is an interesting mix between mathematics and an experimental science. Our analysis above, albeit somewhat preliminary in 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 stake. See the file find-time.c0 on the course web pages. We compile this file, together with the our implementation from this lecture with the `cc0` command below. We can get an overall end-to-end timing with the Unix `time` command. Note that we do not use the `-d` flag, since that would dynamically check contracts and completely throw off our timings.

```
% cc0 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.

Exercises

Exercise 1 *Rewrite the binary search function so that both lower and upper bounds of the interval are* inclusive. *Make sure to rewrite the loop invariants and the loop body appropriately, and proof that the correctness of the new loop invariants. Also explicitly prove termination by giving a measure that strictly decreases each time around the loop and is bounded from below.*

Exercise 2 *Rewrite the invariants of the binary search function to use* is_in(x, A, l, u) *which returns true if and only if there is an i such that $x = A[i]$ for $l \leq i < u$.* is_in *assumes that $0 \leq l \leq u \leq n$ where n is the length of the array.*

Then prove the new loop invariants, and verify that they are strong enough to imply the function's postcondition.

Exercise 3 *Binary search as presented here may not find the leftmost occurrence of x in the array in case the occurrences are not unique. Given an example demonstrating this.*

Now change the binary search function and its loop invariants so that it will always find the leftmost occurrence of x in the given array (if it is actually in the array, -1 as before if it is not).

Prove the loop invariants and the postconditions for this new version, and verify termination.

Exercise 4 *If you were to replace the midpoint computation by*

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

then which part of the contract will alert you to a flaw in your thinking? Why? Give an example showing how the contracts can fail in that case.

Exercise 5 *In lecture, we used design-by-invariant to construct the loop body implementation from the loop invariant that we have identified before. We could also have maintained the loop invariant by replacing the whole loop body just with*

```
// .... loop_invariant elided ....
{
  lower = lower;
  upper = upper;
}
```

Prove the loop invariants for this loop body. What is wrong with this choice? Which part of our proofs fail, thereby indicating why this loop body would not implement binary search correctly?