Lecture Notes on Binary Search

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 6
January 27, 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. We started in the last lecture by discussing *linear search* and giving some background on the 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.

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]

[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

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.

different base.

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, n);
/*@ensures (-1 == \result && !is_in(x, A, n))
        || ((0 <= \result && \result < 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 (-1 == \result && !is_in(x, A, 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`. We assert that the midpoint is indeed between $lower$ and $upper$. That assertion is easy to see, because $lower \leq upper$ and therefore $upper - lower > 0$. Furthermore, $lower + (upper - lower) = upper$, so if we divide the second summand

by 2 (which truncates towards 0), we will have $lower + (upper - lower)/2 < upper$.

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)
//@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 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. 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 (-1 == \result && !is_in(x, A, 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;
      // ...??...
   }
  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 (-1 == \result && !is_in(x, A, 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;
}
```

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.

4 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[3]: 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.

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

5 Big-O Notation

Our brief analysis so far already indicates that linear search should take about n iterations of a loop while binary search take about $\log_2(n)$ iterations, with a constant number of operations in each loop body. This suggest that binary search should more efficient. In the design and analysis of algorithms we try to make this mathematically precise by deriving so-called *asymptotic complexity measures* for algorithms. There are two fundamental principles that guide our mathematical analysis.

1. We only care about the behavior of an algorithm *in the long run*, that is, on larger and larger inputs. It is when the inputs are large that differences between algorithms become really pronounced. For example, linear search on a 10-element array will be practically the same as binary search on a 10-element array, but once we have an array of, say, a million entries the difference will be huge.

2. We do not care about *constant factors* in the mathematical analysis. For example, in analyzing the search algorithms we count how often we have to iterate, not exactly how many operations we have to perform on each iteration. In practice, constant factors make a big difference, but they are influenced by so many factors (compiler, runtime system, machine model, available memory, etc.) that at the abstract, mathematical level a precise analysis is neither appropriate nor feasible.

Let's see how these two fundamental principles guide us in the comparison between functions that measure the running time of an algorithm.

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

[3] and was suggested first in lecture

Let's say we have functions f and g that measure the number of operations of an algorithm as a function of the size of the input. For example $f(n) = 3 * n$ measures the number of comparisons performed in linear search for an array of size n, and $g(n) = 3 * \log(n)$ measures the number of comparisons performed in binary search for an array of size n.

The simplest form of comparison would be

> $g \leq_0 f$ *if for every* $n \geq 0$, $g(n) \leq f(n)$.

However, this violates principle (1) because we compare the values and g and f on all possible inputs n.

We can refine this by saying that *eventually*, g will always be smaller or equal to f. We express "eventually" by requiring that there be a number n_0 such that $g(n) \leq f(n)$ for all n that are greater than n_0.

> $g \leq_1 f$ *if there is some* n_0 *such that for every* $n \geq n_0$ *we have* $g(n) \leq f(n)$.

This now incorporates the first principle (we only care about the function in the long run, on large inputs), but constant factors still matter. For example, according to the last definition we have $3 * n \leq_1 5 * n$ but $5 * n \not\leq_1 3 * n$. But if constants factors don't matter, then the two should be equivalent. We can repair this by allowing the right-hand side to be multiplied by an arbitrary constant.

> $g \leq_2 f$ if there is a constant $c > 0$ and some n_0 such that for every $n \geq n_0$ we have $g(n) \leq c * f(n)$.

This definition is now appropriate.

The less-or-equal symbol $\leq$ is already overloaded with many meanings, so we write instead:

> $g \in O(f)$ if there is a constant $c > 0$ and some n_0 such that for every $n \geq n_0$ we have $g(n) \leq c * f(n)$.

This notation derives from the view of $O(f)$ as a set of functions, namely those that eventually are smaller than a constant times f.[4] Just to be explicit, we also write out the definition of $O(f)$ as a set of functions:

> $O(f) = \{g \mid$ there are $c > 0$ and n_0 s.t. for all $n \geq n_0$, $g(n) \leq c * f(n)\}$

[4] In textbooks and research papers you may sometimes see this written as $g = O(f)$ but that is questionable, comparing a function with a set of functions.

With this definition we can check that $O(f(n)) = O(c * f(n))$.

When we characterize the running time of a function using big-O notation we refer to it as the *asymptotic complexity* of the function. Here, *asymptotic* refers to the fundamental principles listed above: we only care about the function in the long run, and we ignore constant factors.

The asymptotic time complexity of linear search is $O(n)$, which we also refer to as *linear time*. The asymptotic time complexity of binary search is $O(\log(n))$, which we also refer to as *logarithmic time*. *Constant time* is usually described as $O(1)$, expressing that the running time is independent of the size of the input.

Some brief fundamental facts about big-O. For any polynomial, only the highest power of n matters, because it eventually comes to dominate the function. For example, $O(5*n^2+3*n+83) = O(n^2)$. Also $O(\log(n)) \subseteq O(n)$, but $O(n) \not\subseteq O(\log(n))$. Logarithms to different (constant) bases are asymptotically the same: $O(\log_2(n)) = O(\log_b(n))$ because $\log_b(n) = \log_2(n)/\log_2(b)$.

As a side note, it is mathematically correct to say the running time of binary search is $O(n)$, because $\log(n) \in O(n)$. It is, however, a looser characterization than saying that the running time of binary search is $O(\log(n))$, which is also correct. Of course, it would be incorrect to say that the running time is $O(1)$. Generally, when we ask you to characterize the worst-case running time of an algorithm we are asking for the tightest bound in big-O notation.

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.