Ode to Binary Search

Greg Zborovsky

1 Introduction

We learned about binary search on 3/3/2021 in the arrays lecture.

To give a short summary, given a sorted array, we can "binary search" through an array to find a certain element by comparing the middle element to our current element and then deciding either that we have found what we are looking for, that what we are looking for is in the left half of the array, or that what we are looking for is in the right half of the array.

Why should we care about this algorithm? Given a sorted list, searching for a certain element is many times faster with binary search than with linear search. Linear search has time complexity $\mathcal{O}(n)$ whereas binary search has time complexity $\mathcal{O}(\log n)$ where n is the number of elements you are searching through.

For reference (Each test was run 100 times, ex: it takes 1.73 seconds to search through 100,000,000 million items 100 times [This is suuuuper fast - Java's optimizations make this way faster than it would be unoptimized. I tested without optimizations as well and it looks roughly 30x slower]):

n	Linear Search	Binary Search	Speed Difference
10^1	0.000011	0.000012	0.95x
10^2	0.000088	0.000037	2.41x
10^3	0.000226	0.000058	3.88x
10^4	0.001670	0.000021	80.28x
10^5	0.002599	0.000033	78.03x
10^6	0.015747	0.000039	409.00x
10^7	0.196274	0.000054	3614.62x
10^8	1.738984	0.000123	14195.78x

As sizes become larger and larger, binary search gains a greater and greater speed advantage, and a significant one at that.

In this post, I'd like to go more in depth with the implementation and applications of binary search.

2 Implementation Details

The implementation from class:

```java
public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;
    while (high >= low) {
        int mid = (low + high) / 2;
        if (key < list[mid])
            high = mid - 1;
        else if (key == list[mid])
            return mid;
        else
            low = mid + 1;
    }
    return -1 - low;
}
```

This implementation is fairly common, but binary search can interestingly be done very similarly with recursion:

```java
public static int binSearchRecursive(int[] list, int key, int
    low, int high) {
    if (high >= low) {
        int mid = (low + high) / 2;
        if (key < list[mid])
            return binSearchRecursive(list, key, low, mid-1);
        else if (key == list[mid])
            return mid;
        else
            return binSearchRecursive(list, key, mid+1, high);
    }
    return -1 - low;
}
```

where *low* and *high* are 0 and list.length-1 respectively on the first call.

Some things to be careful about in either implementation:

- $(low + high)/2$ could cause an overflow if both *low* and *high* are close to their maximum datatype value. $low + (high - low)/2$ is slightly safer, but will still cause issues when high is close to the maximum allowed value and low is close to the minimum allowed value. For reading on an overflow save implementation, refer to http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0811r3.html

- If looking for a duplicate element, the current implementation will return the index of one of the duplicates, but not necessarily the first or the last index of a certain element. (The search can be adjusted to fulfill either goal.)

3 Generalizations

3.1 On What Can it be Used?

We know that we can binary search on any sorted array. More generally, we can binary search on any function which is monotonic within the range we want to search. In the example of binary search introduced in class, our sorted array is monotonically increasing and our function is $f(x) = array[x]$.

Note that this function does not need to be stored in memory, nor in an array. For example, we can binary search on $f(x) = x^2$ in the domain $[0, n]$ where $n \in \mathbb{N}$ by simply evaluating f(x) whenever we need its value.

3.2 Where Should it be Used?

So we can binary search on any monotonic function, but is binary search always the best option?

No. The easiest case is to consider a LinkedList, like SML's default list, for which finding the value of $f(x)$ is, on average, $\mathcal{O}(n)$. If we we were to binary search for the index of a certain value in a LinkedList, it would take $\mathcal{O}(n \log n)$ as opposed to $\mathcal{O}(n)$ for a regular linear search.

Let r = random access time complexity for $f(x)$. Binary search should be used when r is less than $\mathcal{O}(\frac{m}{\log n})$ where n is the size of whatever you are searching over and m is the time needed to iterate over all n elements. The final time complexity for a binary search will be $\mathcal{O}(r \cdot \log n)$. In the case of an array where $r = \mathcal{O}(1)$, the time complexity for binary search is clearly $\mathcal{O}(\log n)$.

3.3 Extending it to Floating Point

There is no restriction that binary search only functions on integers. With a small change to the implementation, binary search can be adjusted to work on floating point data.

A binary search on monotonic floating point data will approach some exact value, but will terminate once $high - low$ is less than some constant ϵ.

The runtime analysis does not change much. We can consider ϵ to be our unit size, thus our entire search will be over roughly $(high - low) \cdot \epsilon$ values, so the time complexity ends up as $\mathcal{O}(\log \epsilon(high - low)) = \mathcal{O}(\log (high - low) + \log \epsilon)$. If we treat ϵ as a constant, the final runtime complexity remains as $\mathcal{O}(\log (high - low))$.

```java
public static double floatingPointBinarySearch(double key,
    double lowVal, double highVal, double epsilon) {
    double low = lowVal;
    double high = highVal;
    while (high - low > epsilon) {
        double mid = low + (high - low) / 2;
        // f is some function which takes in a double and
            returns a double
        // it must be monotonic on [lowVal, highVal]
        if (f(mid) < key)
            low = mid;
        else
            high = mid;
    }
    return low + (high - low) / 2;
}
```

3.4 Without an End Value

It turns out we don't always need a full range to binary search over for an answer. Having just one point is enough data to eventually use binary search.

With a target value t and a given value $f(x)$ evaluated at x, there exist 3 cases.

1. $f(x) = t \implies$ you have the answer.
2. $f(x) > t \implies$ we want to find a value $f(y)$ which is < t.
3. $f(x) < t \implies$ we want to find a value $f(y)$ which is > t.

Case 1 is trivial.

Case 2 and 3 are essentially equivalent. I will cover case 2. The idea is to create a variable y equal to x and while($f(y) > t$), double the distance of y from x. This does require $f(y)$ to eventually cross t or the process will not terminate. Because the distance from y to x increases at an exponential rate, it will only be doubled a logarithmic number of times with respect to the size of the final search region. Overall complexity of the binary search including this step remains at $\mathcal{O}(r \cdot \log n)$ where $r =$ random access time complexity for $f(x)$.

Case 3 is equivalent to Case 2 with the only difference being that you repeat while($f(y) < t$).

Note: I have seen this concept used often where you aren't initially given too much information, but it does require monotonicity across the entire range that y expands across.

4 Applications

I'll discuss some interesting binary search related problems here.

4.1 Guessing Games

You can ask me at most 20 questions. With no other information, what day and year is my birthday on?

The oldest human is still younger than 130 years old, and every year has at most 366 days. Thus, my birthday must be on one of $366 \cdot 130 = 47580$ days (between today and 47580 days ago). We cannot ask 47580 questions, so we must do something a bit more clever.

You can consider this is as a binary search between [-47580, 0]. You can first ask whether I was born before or on $(47580 + 0)/2 = 23790$ days ago. If I say yes, then the binary search would continue within the range [-47580, -23790]. Otherwise, it would continue within the range [-23789, 0].

Each step reduces the number of days we have to consider. In total, we will need exactly $\lceil \log_2 47580 \rceil = 16$ questions, which is actually less than 20.

In this game our function is:

$$f(x) = \begin{cases} 0 & \text{if } x <= birthday \\ 1 & \text{if } x > birthday \end{cases}$$

Despite having many duplicate values, our function is indeed monotonically increasing. The target date, my birthday, would be the index of the final 0 in the sequence. It turns out this technique of having a monotonic function which only returns 0 and 1, or true and false, is fairly applicable to many competitive programming problems.

4.2 Calculating Inverse CDF

People taking AMS 310 right now (Survey of Probability and Statistics) are probably familiar with CDFs (Cumulative Distribution Functions). Let $F(x)$ be a CDF and let $f(x)$ be a PDF (Probability Distribution Function).

Since

$$F(x) = \sum_{n=0}^{x} f(x)$$

and

$$f(x) >= 0, \forall x \in \mathbb{Q},$$

F(x) must be a monotonically increasing function, and thus we can binary search over it.

To calculate $F^{-1}(y)$, we can binary search over $F(x)$ where $x \in$ [min(domain of distribution), max(domain of distribution)] while searching for the value y. We can choose some arbitrary yet small enough ϵ value such as $\epsilon = 10^{-6}$ and use $(low + high)/2$ as the final value of $F^{-1}(y)$ at termination.

4.3 Calculating the Inverse of Monotonic Functions

I previously described how to calculate the values of the inverse CDF function, but binary search can be used to find the values of the inverse of any monotonic function by applying similar reasoning.

One more common example of this is calculating the square root of a value. There is no obvious way to calculate a square root, but calculating the square of a value is trivial.

To calculate $sqrt(x)$, we can binary search over $n \in [0, x]$ and calculate the value of $n \cdot n$. If $n \cdot n > x$, we can replace the high value with the middle value.

Otherwise, we can replace the low value with the middle value. Again, once $high - low$ is less than some value ϵ, we can return $(low + high)/2$.

In the case of sqrt(x), this method isn't worth it. On modern processors, calculating sqrt(x) is pretty fast, but more difficult operations or functions that aren't already implemented could benefit from this method.

4.4 Finding the Size of a Dartboard

This question actually comes from a Google Codejam Competition, which is a competition for competitive programmers. You can think of it as a math team contest but for programmers.

https://codingcompetitions.withgoogle.com/codejam/round/000000000019fef2/00000000002d5b63#problem

This problem doesn't require binary search, but using binary search makes the problem much simpler. The link I've given has both the problem statement and the solution/analysis.

This problem will require several different problem solving techniques and using binary search multiple times in different dimensions. This is a difficult problem, only recommended for people who are really interested in spending some time on solving a problem.

5 Informal Ramblings

A bit of a lighter section—some more of my own thoughts on binary searching as a concept.

Even in real life, people tend to use binary search to solve small life problems. Looking through a dictionary? You're probably using binary search. Dictionaries are ordered alphabetically, so you're probably using some kind of binary search to flip through pages faster. Broke some code but you don't know where? It's pretty common for people to test the first half, then the first 3/4, then the first 7/8, etc of their code, just to find where a bug is; this is also a form of binary search. Interestingly enough, git supports this type of debugging through git bisect, which will binary search over a range of commits with the help of the user to find where a branch turns from good to bad.

Another real life example, although hidden away, lives within switch statements. In the JVM, memory-efficient switches with a small ranges of indices will be often be implemented through an $\mathcal{O}(1)$ call to a table, but if the switch cases are sparse, the JVM will switch to a form of binary search to find the matching case. More info here: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.html#jvms-3.10

Many other CS problems cannot be solved with binary search, even some problems that require or employ ordered data. They usually still employ concepts similar to those we find in binary search, particularly the idea that at every level, or node, we either disregard the left or right half of something, and continue going down through levels. Many data structures in computer science rely on this concept in some way, and particularly tree-like data structures.

There are definitely some other neat tricks and applications of binary search that I haven't covered, but I hope you found this document interesting.

6 More Resources

Some of the resources I've used or seen:

- Binary Search on Wikipedia
- Binary Search Tutorial by Errichto
- Binary Search Concepts on Leetcode
- Some people's implementations of Binary Search on Codeforces