Binary Search and Friends

Suhas Arehalli

COMP221 - Spring 2026

1 Divide and Conquer & Binary Search

Recall that Divide and Conquer algorithms consist of 3 steps: a Division of a problem into subproblems, a Conquering of those subproblem (typically through recursion), and a recombination of subproblem solutions into a solution for the initial problem. Binary search is a special case of this kind approach: One where we can reduce a large problem to a *single* subproblem about *half the size*. This makes the idea of binary search particularly powerful, not in the least because it represents a class of algorithms that run asymptotically faster than just scanning through the input! However, in its standard form, Binary Search is a bit limited: You need to have one input that is array-like and sorted, and then you return when you find an element equal to the other input. We can fix this by distilling down the reasoning that makes binary search correct:

> Given S, a set of viable solutions to our problem. If we can write an $O(1)$ operation that tells us that a valid solution (if it exists) must be in $S' \subseteq S$ with $|S'| \approx \frac{1}{2}|S|$, we can write a binary search to solve our problem.

Typical binary search has S as the space of possible indices (or $NULL$), and the $O(1)$ operation is a comparison between e and $A[\frac{n}{2}]$, which only works because our input A is sorted. However, this framework can apply to a variety of stranger situations:

2 Binary Search Variants

2.1 Binary Search with Repeated Values

We often think of binary search in cases where values are unique, but things get a bit odd in cases where we allow for duplicates of e — the element we're searching for. By our standard definition of the sorted search problem, we now no longer have unique solutions — *any* i s.t. $A[i] = e$ is a valid return value for our function!

This might motivate us to ask a slightly harder version of the question. For instance:

Problem Statement (Sorted Search — First)**.**
Input: A sorted array $A[1 \dots n]$
Output: The smallest i such that $A[i] = e$

Here we want to find the *first* appearance of e in the array (the smallest index i, or, super formally, $\min\{i \in \{1, \ldots, n\} \mid A[i] = e\}$). How can we modify Binary Search to find this index?

Well, we clearly need to remove the *early exit* — i.e., the return if $A[m] = e$ conditional — from binary search, as finding an index that contains e is no longer sufficient. Instead, we have to determine (1) what a new return condition should be and (2) what recursive call we should make if we find an m such that $A[m] = e$.

The best way to think about this is by (as always) working through an example: If we start with $A = [1, 2, 3, 3, 3, 4, 5]$ and $e = 3$, we determine $m = 4$ and $A[m] = 3$. This means the leftmost e must be somewhere in $A[1 \ldots 4]$! We can repeat and find that $A[1 \ldots 4] = [1.2.3, 3]$, $m = 2$, and $A[m] = 2$. As usual, this means that we recurse right and add $m = 2$ to the resulting index. We then have $A[3 \ldots 4] = [3, 3]$, $m = 1$, and $A[m] = 3$. We again know that the first occurance might be m, but it could be further left, so we again search $A[1 \ldots m] = A[1 \ldots 1] = [3]$. Here, we are left with exactly one possible option for the furthest left e, and so it must be it! We return 1, remember to add back in the 2 from when we recursed to the right, and get that the first instance is at index $2 + 1 = 3$ as we wanted!

What is important is to ensure that (1) We always reduce the size of our input when we recurse (no infinite loops!) and, at the same time, (2) we only remove elements when we *know* they cannot be our solution. This gets us to Alg. **??**.

---

function FIRSTBS($A[1 \ldots n]$, e)
  if $n \leq 0$ **then**
    return $NULL$
  end if
  $m \leftarrow \lceil \frac{n}{2} \rceil$
  if $A[m] = e$ **then**
    if $n = 1$ **then**
      return 1
    end if
    return FIRSTBS($A[1 \ldots m]$, e)
  end if
  if $A[m] > e$ **then**
    return FIRSTBS($A[1 \ldots m - 1]$, e)
  else
    return m + FIRSTBS($A[m + 1 \ldots n]$, e)
  end if
end function

2.2 Root Finding and the Bisection Method

If you recall the quadratic formula or have taken a calculus class, you're probably aware that a common problem in mathematical modeling is finding the *roots* of a function — i.e., for a function f, a root/zero is an x such that $f(x) = 0$. It turns out one can use Binary Search to find zeroes under some reasonable conditions!

Consider the following problem:

Problem Statement.

Input: a continuous function f and a, b such that $a \leq b$ and $f(a) < 0$, $f(b) > 0$
Output: x such that $f(x) = 0$

We're not going to solve it! Instead, we're going to *approximate* its solution using binary search!

Now, this isn't a calculus class, so here are some useful intuitions to have if terms like *continuous* are unfamiliar: Think of continuity as the idea of being able to draw the function without raising your pen (or marker, in my case). This leads to a natural result: The *Intermediate Value Theorem* (IVT) tells us that if a continuous function takes values $f(a)$ and $f(b)$ at points a, b, then for any $f(a) \leq c \leq f(b)$, there is some $a \leq x \leq b$ such that $f(x) = c$. I.e., I can't draw the function between a and b without reaching every value between $f(a)$ and $f(b)$.

The IVT actually tells us all we need to do to set up our binary search variant! If $f(a) < 0 < f(b)$, then I know my root is somewhere within the interval (a, b). My binary search idea is to see if I can halve that interval: If I compute $f(\frac{b-a}{2})$, it's sign should tell me (via the IVT) which half of the interval a root is guaranteed in! Consider

1. If $f(\frac{b-a}{2}) < 0$, then I know $f(\frac{b-a}{2}) < 0 < f(b)$ and so by the IVT my answer is in $(\frac{b-a}{2}, b)$
2. If $f(\frac{b-a}{2}) > 0$, then I know $f(a) < 0 < f(\frac{b-a}{2})$ and so by the IVT my answer is in $(a, \frac{b-a}{2})$

Of course, my stopping criterion here is a little bit trickier, because I may never actually find a value x where $f(x) = 0$. Instead, what we want to do is design an algorithm where we find smaller and smaller intervals with each iteration and stop when the interval is sufficiently small. We can also frame this in terms of *error*: If I really want to return a single value, I might be OK providing an x where I may not know that $f(x) = 0$, but I do know that there exists an x^* close to x (i.e., $|x - x^*| < \varepsilon$ for some small ε) such that $f(x^*) = 0$. Informally, you can determine how many iterations you should run depending on how close you want x to be to the right answer! The activity helps you reason about this framing!

2.3 One-Sided Binary Search

Suppose you have an unbounded sequence $\{x_i\}_{i \geq 1} = x_1, x_2, \ldots$ that is *monotonic increasing* — i.e., for any $i \geq 1$, $x_{i+1} \geq x_i$.[1] Suppose you want to see if some value e is in this sequence. If you have a right bound on the sequence — i.e., some k such that $e \leq x_k$, then we can do binary search on the sorted sequence $x_1, \ldots, x_k$. We can frame the problem in shorthand as follows:

Problem Statement (Monotonic unbounded search)**.**
Input: an infinite, monotonic increasing sequence $\{x_i\}_{i \geq 1}$, some element e
Output: j such that $x_j = e$ if j exists, $NULL$ otherwise

How can we solve this problem as efficiently as possible? Given this framing, it's difficult to avoid outcomes that never terminate[2], but let's suppose we know there exists some k such that $x_k \geq e$ — i.e., a right bound of the sort that will reduce this to a normal binary search. How can we find it fast? We can find k using a *one-sided* binary search which doubles rather than halves! take a look at Alg. 1.

And now we've *reduced* the problem of monotonic unbounded search to a normal binary search! We just need to find the first *power-of-two* j such that $x_j > e$. This way, we can approach this

[1] This is our unbounded equivalent of being sorted!
[2] This would take enforcing that values are unique and integers, for instance

Algorithm 1 A One-Sided Binary Search.

```
function getRightBound({x_i}_{i≥1}, e)
    j ← 1
    while x_j ≤ e do
        j ← 2j
    end while
    return j
end function
function OSBinarySearch({x_i}_{i≥1}, e)
    k ← getRightBound({x_i}_{i≥1}, e)
    return BinarySearch([x_1, ..., x_k], e)
end function
```

bound at an *exponential* rate! That is, if we know a right bound k exists, we need only $\lceil \log k \rceil$ loops to get the bound!

Note again the structure of binary search: Since the sequence is monotonic increasing, if $x_k > e$ — the condition to break the loop — $\forall i \geq k$, $x_i > e$. This means that if $\exists j$ such that $x_j = e$, $j < k$, and thus doing binary search on $x_1, \ldots, x_k$ is sufficient to determine if e is in the sequence!

2.4 Binary Search in a Rotated Array

Now to a classic interview problem! Suppose you have a *rotated* sorted array and want to find an element e within it. That is, an array that *would* be sorted, but has been shifted by some unknown amount. For example, the array $[1, 2, 3, 4, 5]$ may have been rotated to $[3, 4, 5, 1, 2]$.

A naive way to solve this problem would be to figure out how much the array is shifted, rotate it back, and then do a normal binary search. Unfortunately, shifted the array back is guaranteed to be an $O(n)$ operation — slow! Instead, one approach is to find the shift in $O(\log n)$ time *and then do binary search directly on the shifted array.*

Let p — the *pivot* — be the index of the smallest element in a rotated sorted array. How can we find the index of the pivot quickly with a finite number of comparisons? Let's consider an example: the array $A = [3, 4, 5, 1, 2]$, with pivot $p = 4$. Using the normal binary search strategy of inspecting the mid-point $A[3] = 5$ doesn't give us much information, so we'll need to probe more indices. At the same time, we need to make sure we're always looking at a *constant* number of indices, lest our algorithm take longer than $O(\log n)$ time. Our trick (and a common pattern to remember!) is to consider the left and right bounds (here, $A[1]$ and $A[5]$) in addition to the midpoint. Here, consider knowing $A[1] = 3$, $A[3] = 5$, and $A[5] = 2$. What does this tell us? Well, the subarray $A[1 \ldots 3]$ is in sorted order, and is thus "normal" within a rotated subarray. $A[3 \ldots 5]$ is *not* — $A[5] < A[3]$! At some point past index 3, we *decrease* by increasing the index. Why? Because we reach the pivot! Thus, the pivot must be in $A[3 \ldots 5]$!

This gives us the pivot-finding scheme in Alg. 2. Correctness here is tedious, but not particularly interesting to show, but again relies on the fact that the comparison in question tells us where the minimum element must be — in the out-of-order half, if there is one, and the minimum element (certainly in the left subarray) if they're both in-order. From here, one simply needs to find a way to do binary search on the array $A[p \ldots n] + A[1 \ldots p-1]$ (where $+$ is concatenation) without doing the shift explicitly. This requires clever indexing work, which you see in the activity!

Algorithm 2 A method to find the *pivot* (the smallest element) in a rotated sorted array

```
function findPivot(A[1...n])
    if n = 1 then
        return 1
    end if
    m ← ⌈n/2⌉
    if A[m] > A[n] then
        return m+findPivot(A[m+1...n])
    else
        return findPivot(A[1...m])
    end if
end function
```