Binary Search Slides

Problem

Suppose we have a **sorted** array of n numbers (which we will call ‘a’). We want to see if m is in a (or find the index of m). We want to do this as fast as we can. How might we do this?

Example:

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

First Approach

Let us visit every element until we find m. Starting from index 0 and going to n-1, we check every element and see if it is equal to m. If it is, we stop and return the index. If we never find it, we report so.

This is known as *linear search*, and it works on any array! However, it is O(n). This is because if m was a large value (say m=20), we wouldn't find it until towards the end.

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Consideration

Let m = 16.

Suppose, for no particular reason, we decided to check a[6] = 12 first.

Given this fact, is there any reason to check a[0] to a[5] now?

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Consideration (Part 2)

The answer is **no**. This is because the array is sorted, and since m = 15 > 12, we know m cannot appear before i = 6. This eliminates a large part of the array!

But: i = 6 is arbitrary. And if m < a[6], we would've only eliminated a little bit. So, what index should we pick such that we eliminate a lot every single time?

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Binary Search

The idea is to pick the ***middle index***. This way, we get rid of half of the array. Once we get rid of half of the array, we can pick the middle of the remaining section again, and again, and so on. This way, we eliminate half of what remains every single time. This gives us a complexity of O(log n).

This only works because the array is sorted.

Example

Let’s say we have the same array and we want to find m=12. Intuitively, this is how we would try to find it.

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

a	1	3	4	8	[illegible]	11	12	15	16	20
i	0	1	2	3		5	6	7	8	9

a	1	3	4	8	[illegible]	11	12	15	16	20
i	0	1	2	3	[illegible]	5	6	7	8	9

Implementation

There is a clear problem with that example, how do we decide where “half” truly is, especially when there isn’t a clear half? How do we even know what remains? This is where the implementation comes in.

We should be careful on how we decide to implement it. It should be noted that deleting half of the array is **O(n)** which defeats the purpose.

Instead of trying to do crazy computation to find where the middle is, how about we simply track where we are bounding the array to? We will keep track of the lowest index and highest index we are still considering. We can easily obtain the middle from that information, and when we eliminate half of the array we move either the low or high index to wherever the middle is to get rid of that half.

Python Example

```
def binsearch(a,n,m):
    low = 0
    high = n - 1
    while low <= high:
        mid = (low + high) // 2 # integer division
        if a[mid] == m:
            return mid
        if a[mid] < m:
            low = mid + 1
        if a[mid] > m:
            high = mid - 1
    return -1 # not found
```

Implementation Example

Try to trace it on your own with m=12. Note that n=10.

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Implementation Example Solution

Here is the solution. Remember that m=12

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Low Mid High

Mid = (Low+High)//2 = (0+9)//2 = 4
a[Mid] < 12 so Low = Mid + 1

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Low Mid High

Mid = (Low+High)//2 = (5+9)//2 = 7
a[Mid] > 12 so High = Mid - 1

Implementation Example Solution (Part 2)

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Low High
Mid

Mid = (Low+High)//2 = (5+6)//2 = 5
a[Mid] < 12 so Low = Mid + 1

a	1	3	4	8	10	11	12	15	16	20
i	0	1	2	3	4	5	6	7	8	9

Low
Mid
High

Mid = (Low+High)//2 = (6+6)//2 = 6
a[Mid] = 12

Comparison

O(log n) is MUCH faster than O(n).

Here is one example of a random trial comparing the number of iterations on an array of length 10^5: Linear Search: 38787; Binary search: 17

The advantage is clear. Here is f(n) = n vs. g(n) = log n on a graph

A Few Disclaimers

1. Remember, binary search only works if the array is sorted!!!
2. However, it does not have to be sorted least to greatest. If has some sort of consistent order that spans the entire array (i.e., greatest to least), it can still work! You will have to slightly tweak the algorithm.
3. You can also use binary search to find the greatest element smaller than a bound, or the smallest element greater than a bound.