*"There is no algorithm for creativity." ~ Andy Hargreaves*

Dear reader, welcome to a new problem based on Binary Search (Divide & Conquer Strategy). The problem name is *‘First and Last Index*’.

***Problem Statement***

You are given a sorted array ‘a’ of n elements. Given a number d, you have to find it’s occurrence in the array as well as its last occurrence. The array may contain duplicates.

Important Links: [Problem Link](https://www.pepcoding.com/resources/online-java-foundation/function-and-arrays/first-index-last-index-official/ojquestion), [Question Video](https://www.youtube.com/watch?v=PFaoQ72ziNk), [Solution Video](https://www.youtube.com/watch?v=9NXZccIWNqU)

***Example***

Find first & last occurrence of 33 in the given array:

Array a = {1, 5, 10, 15, 22, ***33***, 33, 33, 33, ***33***, 40, 42, 55, 66, 77}.

Hence, the answer will be 5 and 9 respectively (0-based indexing).

***Deducing Algorithm***

Let us forget about binary search for now. How can you solve this question?

Simply, we can apply a ***linear search*** on the array. We can maintain two variables *first* and *last* which will store the indices. We traverse through the array from left to right, as soon as we get our first element equal to d, we update *first* = current index. Now, we will keep on traversing (and updating *last* along with it) until we get a higher element than d, or we have reached the end of the array. In this manner, *last* will point to the last occurrence of d.

But isn’t the solution too trivial? Yes, it is having ***O(N) worst case time*** complexity, (and O(1) auxiliary space is required).

Now, let us try to implement ***binary search*** to solve this problem.

We have previously solved how to find the *floor* and the *ceil* of a number in a sorted array using binary search in the problem: [Broken Economy](https://www.pepcoding.com/resources/online-java-foundation/function-and-arrays/ceil-floor-official/ojquestion). Please try to think in the same line.

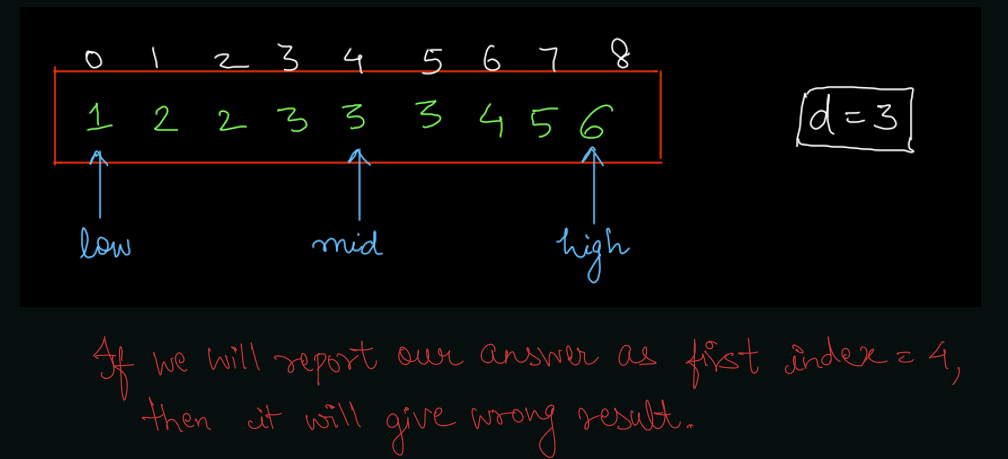
In simple binary search, we update low and high pointers only when we have not found our element, i.e. make low point to mid + 1 if element at mid is less than element we are searching, or, make high point to mid - 1 if element at mid is greater than element we are searching for. We break out from our recursion/while loop whenever we find an equal element.

We have to modify our trivial binary search algorithm to tackle the problem that we should not break out whenever we find an equal element. Even if we find an equal element, we should continue our binary search, as we have to find it’s leftmost and rightmost occurrence.

We will solve for the first and last index by applying binary search two times in the array. I will tell you, why can’t we find both in a single binary search, i.e. simultaneously. Let us consider the *first* index first.

Algorithm for two cases: when element at mid < d, and element at mid > d, will remain same. The only difference is to modify the case when element = d. The mid index is a ***potential candidate***, but not our final answer.

*WHY*? Consider the following example:

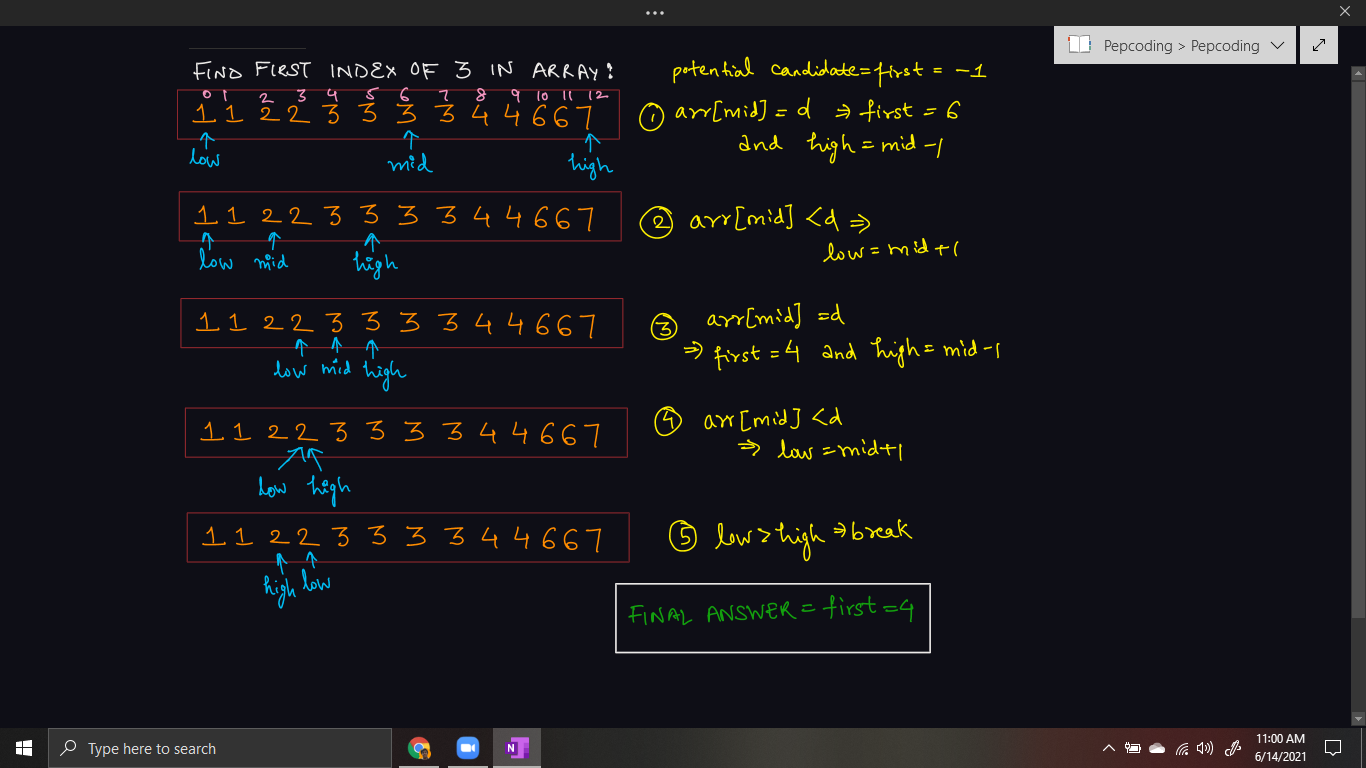


As you can see, as and when we encountered arr[mid] = d, breaking out of the loop can result in wrong output.

So, what can you modify? Some of you might think that our finding mid (whose value = d), why not go as backwards as possible using linear search. But, the problem in this case is, what if the entire array is made up of elements = d? You will linearly traverse upto 0th index from mid, which will bring you back to O(N) time complexity. We do not want it, right?

So, now here is an eye opener: We will mark arr[mid] (=d) as our potential candidate (it may or may not be our final answer), and continue our binary search for the left subarray, and try to find if there is any element with index less than mid, which is also equal to d. We will keep on doing this, until the binary search terminates (low becomes greater than high).

Let us take an example:



For finding the last index, the algorithm will be almost the same, just in the third case when arr[mid] = d, we will update our potential candidate and modify low = mid + 1, as we have to continue for the right subarray after finding an equal element. You can try yourself to trace the working for it using the same test case, as it will be very similar.

One question, which is left unanswered is, why can’t we find both the first and last index in a single binary search? As you can see for the first index, we are modifying high = mid - 1, whereas for the last index, we are modifying low = mid + 1 in the third case. How will you update both pointers in single binary search? Hence, a simple choice to calculate them separately.

***Pseudo Code/ Algorithm***

***First Index***:

* Set potential candidate *first* = -1
* Take low = 0, high = size of array - 1. Apply binary search until low <= high
  + Find mid = (low + high)/2
  + If arr[mid] < d, then update low = mid + 1
  + Else if arr[mid] > d, then update high = mid - 1
  + Else (arr[mid] = d)
    - Update Potential candidate first = mid
    - Update high = mid - 1
* If first = -1, then no occurrence of d in array, else return *first*.

***Last Index***:

* Set potential candidate *last* = -1
* Take low = 0, high = size of array - 1. Apply binary search until low <= high
  + Find mid = (low + high)/2
  + If arr[mid] < d, then update low = mid + 1
  + Else if arr[mid] > d, then update high = mid - 1
  + Else (arr[mid] = d)
    - Update Potential candidate last = mid
    - Update low = mid + 1
* If last = -1, then no occurrence of d in array, else return *last*.

Please try to code this without taking help of the video solution. It will help you develop an insight about binary search.

***Implementation (Java)***

**import java.io.\*;**

**import java.util.\*;**

***public* class Main{**

***public* *static* void main(String[] args) throws Exception {**

**BufferedReader br = *new* BufferedReader(*new* InputStreamReader(System.in));**

**int n = Integer.parseInt(br.readLine());**

**int[] arr = *new* int[n];**

***for*(int i = 0; i < n; i++){**

**arr[i] = Integer.parseInt(br.readLine());**

**}**

**int data = Integer.parseInt(br.readLine());**

**int left = 0;**

**int right = arr.length - 1;**

**int fi = -1; *// First index = fi***

***while*(left <= right){**

**int mid = (left + right) / 2;**

***if*(data > arr[mid]){**

**left = mid + 1;**

**} *else* *if*(data < arr[mid]){**

**right = mid - 1;**

**} *else* {**

**fi = mid;**

**right = mid - 1;**

**}**

**}**

**left = 0;**

**right = arr.length - 1;**

**int li = -1; *// last index = li***

***while*(left <= right){**

**int mid = (left + right) / 2;**

***if*(data > arr[mid]){**

**left = mid + 1;**

**} *else* *if*(data < arr[mid]){**

**right = mid - 1;**

**} *else* {**

**li = mid;**

**left = mid + 1;**

**}**

**}**

**System.out.println(fi);**

**System.out.println(li);**

**}**

**}**

This code is written and explained by our team in [this video](https://www.youtube.com/watch?v=9NXZccIWNqU) from *[6:55, 10:30]*. Please refer to it if you are stuck somewhere.

***Time & Space Complexity Analysis***

Come on friend, Give it a try!

**Time Complexity** - We are doing binary search only. Even if we are continuing our binary search after finding an element, it is still reducing the search space by half of the original size. Hence, the time complexity will remain as ***O(log2 n)***.

**Space Complexity** - O(1) auxiliary space is required to store three integer pointers, low, high and mid.

**Extra Gyaan(Knowledge)**: Have you heard of the lower bound and upper bound of an element in an sorted array. Lower bound is nothing but finding the first index, and upper bound is finding the first greater element (element next to last index), i.e. the ceil of a number. Try to implement it too!

I hope you enjoyed solving the problem with me. I will see in the next problem: Subsets of Array, until then *sayō nara*!

Contributor : Archit Aggarwal