Date :- 02/02/2026
Day :- Monday

Binary Search

Real Life example

Take dictionary

→ In Dictionary we have all words Suppose you need to find the meaning of enlarge then first you will split into two halves in dictionary and you check which words are these on that paper and you seen 5 Letter words are these and now you will remove right part and split the left part and Like this you will do.

Linear Search

we search the target element by checking one after another in the array

ex:- 2, 6, 4, 1, 3, 9

tar = 9

Here you will check 2 is equal to target not equal so you will go to next element.

For this time Complexity is O(n)

ex:- 1, 3, 6, 2, 8 tar = 1

Here you will search and you will get

It is the best case Time Complexity O(1)

Binary search

for binary search the array must be in sorted form

ex:- i = 0, 1, 2, 3, 4, 5, 6

2, 4, 5, 7, 8, 13, 19

low = 0

tar = 8

high = 6

→ The search space is always between Low and high.

→ First you will split them into two parts by calculating mid

$$mid = \frac{(low + high)}{2}$$

→ Then after we check

if a[mid] == tar

we return idx (index)

else if a[mid] < tar

we go to right part and removes left part

low = mid + 1

else

removes right part and goes to left part

high = mid - 1

Here when low crosses high then there is no search space so element not found.

Search the target element in the array

0	1	2	3	4	5	6
2	4	5	8	13	17	19

arr = [2, 4, 5, 8, 13, 17, 19] tar = 17

Iterative

```
public int binarySearch(int[] arr, int tar)
{
    int n = arr.length;
    int low = 0;
    int high = n-1;
    while (low <= high)
    {
        int mid = low + (high-low)/2;

        if (arr[mid] == tar)
            return mid;
        else if (arr[mid] < tar)
            low = mid+1;
        else
            high = mid-1;
    }
    return -1;
}
```

Recursive

I will pass (arr, 0, 6, tar)
0 6

```
public int binarysearch (int[] a, int low, int high, int tar)
{
    if (low > high)   search space not exist
        return -1;
    int mid = (low+high)/2;
    if (a[mid] == tar)
        return mid;
    else if (a[mid] < tar)
        return binarysearch (a, mid+1, high, tar);
                                  Here it returns 5
    else
        return binarysearch (a, low, mid-1, tar);
}
```

binarysearch(a, 0, 6, 17)
it calls another function

```
public int binarysearch (a, 4, 6, 17) {
{
    if (low>high)
        return -1;
    int mid = (low+high)/2;   5
    if (a[mid] == tar)        17==17
        return mid;           so it returns 5
    else if (a[mid] < tar)
        return binarysearch (a, mid+1, high, tar);
    return binarysearch (a, low, mid-1, tar);
}
```

this will return back to where it is called

Time Complexity

for ex:-

length of the array is 32

Here you will split into two halves

32 – 1
↓
16 – 1
↓
8 – 1
↓
4 – 1
↓
2 – 1
↓
1 – 1

⑥

total 6

$32 \Rightarrow 2^5$

5 times we are dividing

$\log_2 n$ is the time complexity for binary search

$\log_2 2^5$

$5 \log_2 2$

$5(1)$

$= 5$

overflow

we will use this formula

for mid

$$mid = low + \frac{(high - low)}{2}$$

Lower Bound

→ We have to find the smallest index that is greater than or equal to x.

0	1	2	3	4	5	6
1	3	4	6	9	11	12

a = [1, 3, 4, 6, 9, 11, 12]

Given: x = 7

Iterative way

```
public int Lowerbound(int[] a, int tar)
{
    int ans = a.length;
    int low = 0;
    int high = n-1;
    while (low <= high)
    {
        int mid = low + (high - low)/2;
        if (a[mid] >= tar)
            ans = mid;
            high = mid - 1;
        else
            low = mid + 1;
    }
    return ans;
}
```

→ Here when we got an element that is greater than target then we will store that index and go to left part until search space exist

Recursion

```
public int binarysearch (int[] a, int low, int high,
                         int ans, int tar){

    if (low > high)
        return ans;

    int mid = low + (high-low)/2

    if (a[mid] >= tar)
        ans = mid;
        return binarysearch(a, low, mid-1, ans,
                            tar);
    else
        return binarysearch(a, mid+1, high, ans, tar);

}
```

upper bound

→ Here we find the smallest index that

a[mid] > x x is target
↓
index

Same Like Lower bound

but there is no =

Floor

→ largest element that is less than or equal to x.

a[mid] <= x

Ceil

→ Smallest element that is greater than or equal to x a[mid] >= x

ceil is similar to Lower bound

Floor

Given an array

$$a = [2, 5, 6, 13, 14, 17, 19] \quad x = 16$$

(indices 0 1 2 3 4 5 6)

we need to find Largest element that is less than or equal to x

low=0 high=6

mid=3

a[mid]<=x

13<=16

so I will go right side to get Largest ele

low=mid+1

and every time I will store in ans

when low crosses high then search space not exist so we return ans

```
public int floor(int[] a, int n, int x)
{
    int low=0;
    int high=n-1;
    int ans=-1;
    while(low<=high)
    {
        int mid= low+(high-low)/2;
        if(a[mid]<=x)
            ans=a[mid];
            low=mid+1;
        else
            high=mid-1;
    }
    return ans;
```

Find the first and last occurence of given element.

Given array

```
          0  1  2  3  4  5  6  7  8  9
a = [2, 3, 5, 7, 11, 11, 11, 13, 15, 18]
```

or

```
          0  1  2  3  4  5  6
a = [1, 2, 4, 4, 4, 6, 8]
```

first occurence is 2

Last occurence is 4

to find first occurence we use Lower bound and to find last occurence we use upperbound - 1

and there is another approach

you will take two variables

first = -1 and last = -1

for first

```
int low=0
int high=n-1
while(low<=high)
{
        int mid= low+(high-low)/2;
    if(a[mid]==x)
        first=mid
        high=mid-1;
    else if(a[mid]<x)
        low=mid+1
    else
        high=mid-1;
}
return first
```

similarly for last we go right side

if (first == -1)
    we print (-1, -1)
otherwise
    first, last

if first == -1
then there is no element.

Search in a rotated sorted array

```
      0  1  2  3  4  5  6  7
a = [6  7  8  1  2  3  4  5]
```

Given tar = 8

Here previously we are calculating mid

if a[mid] < tar
then we are thinking tar is in the right side
otherwise we are going to the left side

→ But Here it is not Like that suppose you take 7 and mid ele is 1 but here 7 is not in the right side it is in the Left side of that element

→ so here we need to think which part we should remove either Left part or right part

```
0  1  2  3  4  5  6  7        tar=8
6  7  8  1  2  3  4  5
            mid=3
         high=7
```

first low=0
mid=3
if (1 == 8) false
so here we check which one part have to remove

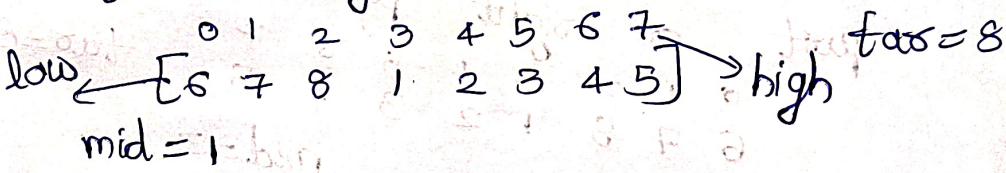
for that
we check if left part is sorted or not
for these we check by the condition

if (a[low] <= a[mid])
then it is sorted (left part is sorted)
otherwise
it is not sorted
(Right part is sorted)

if left part is sorted
then I would check if the target is in that part or not.

if (a[low] <= tar && tar <= a[mid])
then we have element here
→ So we go to left side until that is equal to tar
and if we go right side then it will be wrong you are seeing in unsorted array.

similarly for right part also



mid = 1

1 == 8 false
then we check which one is sorted

6<=1 false

So right part is sorted

I will check 8 is present in that sorted array

1<=8 && 8<=5

false

So we go to left side

by high=mid-1

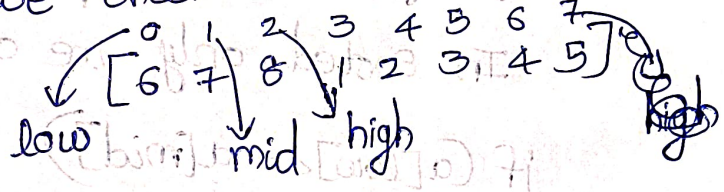
↓

3

low=0 high=2

mid=1 a[1] 7==8 X false

again we check which one is sorted



~~6<=7~~ 6<=7

yes so it is sorted

here we dont go right side

and we will check go left side

if the tar is present in that part for that condition

6<=8 && 8<=7 false

So we go right side

low=mid+1

low = 1+1

low = 2

low=2 high=2

mid=2

a[2] = = 8 true

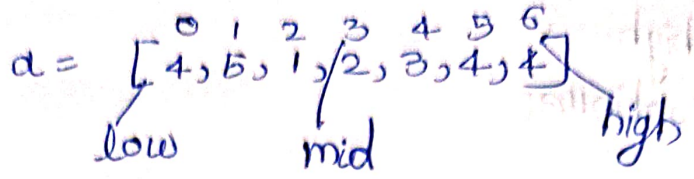
8

return mid

```
public int searchrotate (int[] a, int n, int tar)
{
    int low=0;
    int high=n-1;
    while (low<=high)
    {
        int mid= low+(high-low)/2;
        if (a[mid]==tar)
            return mid;
        otherwise
        we don't which part to remove
        In sorted only we check
        if (a[low]<= a[mid])
        {
            if (a[low]<=tar && tar<= a[mid])
            {
                high=mid-1;
            }
            else{
                low=mid+1;
            }
        }
        else{
            if (a[mid]<=tar && tar<= a[high])
            {
                low=mid+1;
            }
            else{
                high=mid-1;
            }
        }
    return -1;
}
```

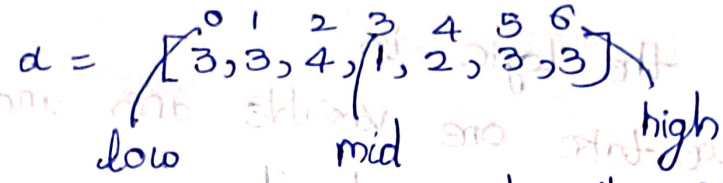
Search in rotated Sorted Array - II

Here we Contain duplicates in the array



For this test case
it passes

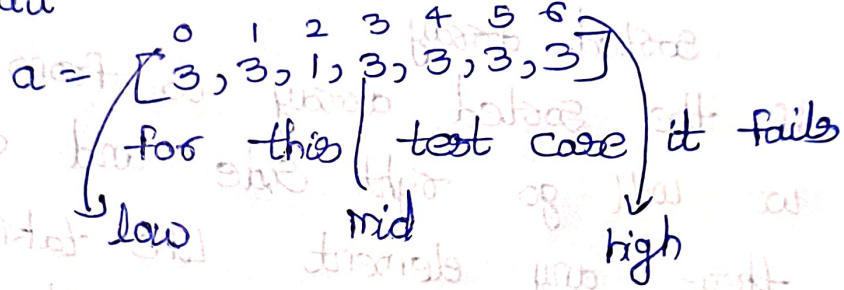
→ But there is one edge case



for this also it passes

0 1 2 3 4 5
3, 3, 1, 2, 3, 3, 3, 3, 3

wait



Here a[low] = a[mid] = a[high]

all 3 are equal so we can't
choose which one to remove

for this edge case to overcome this
we shrink from both sides

low++ high--

and skip the remaining process

it will go as

low = 1 high = 5

in while loop

```
if(a[low]==a[mid] && a[mid]==a[high]){
        low++;
        high--;
        continue;
}
```

Find the minimum element in the sorted Array

	0	1	2	3	4	5
a =	5	6	1	2	3	4

Here the Logic is

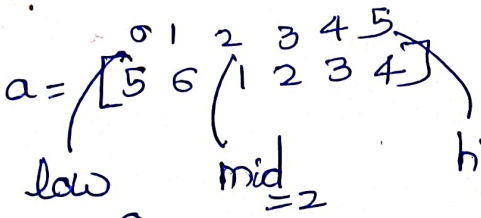
We take one variable ans and we assign max value to it

→ After that we check for the sorted array and take the first elements in that sorted array

if the sorted array is from Left side we will go right side and check is there any element Like taking first element and checking with minimum if it is less than the minimum value we update it.

```
public int findMin(int[] a, int n){
    int low=0;
    int high=n-1;
    int ans= Integer.MAX_VALUE;
    while(low<=high)
    {
        int mid= low+(high-low)/2;
```

```
if (a[low]<= a[mid])
        ans= Math.min(ans, a[low]);
        low=mid+1;
    else
        ans= Math.min(ans, a[low]);
        high=mid-1;
}
return ans;
}
```



low =0

5≤1

not sorted so we go right side

rightside sorted

in right part we take first ele

and ans=1 → and goes to a[mid] left part

high= mid-1

high = 1

low=0 high=1

mid=0

5≤6 sorted so

we take first ele

nd go right side

low=mid+1 high=1

=1 mid=1

ans = min(1,6)

=1

low=2

Crosses and while fails

and In answer we have min=1