(Binary Search)

→ Binary Search is the algorithm which is applicable whenever we search in a sorted area.

Ex :- Dictionary

Pseudo code :-

```
f(arr, n, target)
{
    low=0, high= n-1;
    while (low<=high)
    {
        mid=(low+high)/2;
        if(arr[mid] == target)   return mid;
        else if ( target > arr[mid])   low = mid+1;
        else   high = mid-1;
    }
    return -1;   // if not found
}
```

→ Using Recursion :-

```
f(arr, low, high, target) {
    if(low > high)
        return -1;
    mid = (low+ high)/ 2;
    if(a[mid] == target)
        return mid
    else if(target > a[mid])
        return f(arr, mid+1, high, targ
    else
        ~~f(arr~~
        return f(arr, low, mid-1, target
```

→ Time Complexity :-

$$\begin{array}{ccc} 32 & (1) & 64 \\ \downarrow & & \vdots \\ 16 & (1) & \vdots \\ \downarrow & & \vdots \\ 8 & (1) & \vdots \\ \downarrow & & \vdots \\ 4 & (1) & \vdots \\ \downarrow & & \vdots \\ 2 & (1) & \vdots \\ \downarrow & & \\ 1 & (1) & \\ \hline & \underline{6\ \text{steps}} & \underline{7} \end{array}$$

$32 = 2^5$

$64 = 2^6$

TC :- $O(\log_2 n)$

$\log_2 32$

$\log_2 2^5$

$5 \times \log_2 2$

$5 \times 1 = 5$

Hence - TC :- $O(\log_2 n)$

⇒ overflow case :-

Imagine - $\underset{\text{low, high}}{0}$ ----- search space ------ $\overline{\text{INT-MAX}}$

$$\begin{array}{cc} \downarrow & \downarrow \\ \text{high} & \text{low} \end{array}$$

if you are at last element -

$$\text{mid} = \frac{(\overset{\overset{\text{INT\_MAX}}{\uparrow}}{\text{low}} + \overset{\overset{\text{INT\_MAX}}{\uparrow}}{\text{high}})}{2} = \underline{2 \times \text{INT\_MAX}} \rightarrow \text{long lon}$$

or alternate sol^n :-

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

if search space till INT_MAX

lower bound :- smallest index such that arr[ind]>=n

arr[ ] =	[3,	5,	8,	15,	19]	n=5
	0	1	2	3	4	

n=8
lb=2

n=9
lb=3

n=16
lb=4

n=20
lb=5

ind →	0	1	2	3	4	5	6	7	8	9
arr[ ] =	[1	2	3	3	5	8	8	10	10	11]
	low				↑ mid					high

$\boxed{ans = n}$

Cond^n :- may be an answer
X be an answer

Pseudo code :-

```
f(arr, target, n) {
    low=0, high=n-1
    ans=n
    while (low <=high) {
        mid= $\frac{(low+high)}{2}$
        if (arr[mid]>=n) {
            ans=mid
            high=mid-1
        }
        else {
            low=mid+1
    }
return ans;
```

→ **C++ method :-**

```
                         arr                arr+n
lb = lower_bound (arr.begin(), arr.end(), n)-arr.begin()
```

→ TC :- $O(\log_2 n)$

upper Bound :- smallest index such that arr[ind] > n

```
         0  1  2  3  4  5  6   7   8   9
arr[] = [2  3  6  7  8  8  11  11  11  12]
```

n=10 n=12

ind=6 ub=10

Pseudo code :-

```
if(a[mid] > n)
    ans = mid
    high = mid-1
else
    low = mid+1
```

→ **C++ STL :-**

```
                         arr                arr+n
ub = upper_bound (arr.begin(), arr.end(), n)-arr.begin()
```

→ TC :- $O(\log_2 n)$

Search Insert position :-

find lower bound & insert

i.e. insert at where arr[ind] >= n

arr[] = [1 2 4 7] n = 6

= [1 2 4 6 7]

↑ lb_pos

arr[] = [1 2 4 7] n = 2

= [1 2 2 4 7]

↑ lb_pos

Floor and ceil in sorted array :-

Floor :- largest no. in array <= x

arr[] = [10, 20, 30, 40, 50]

n=25 n=40

f = 20 f=40

Pseudo code :-

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

ceil :- Smallest no. in array >= x

arr[] = [10 20 30 40 50]

n=25 n=40
c = 30 c = 40

Implementation same as lower bound.

Some Imp. questions on BS :-

(1) find the first & last occurences of x :-

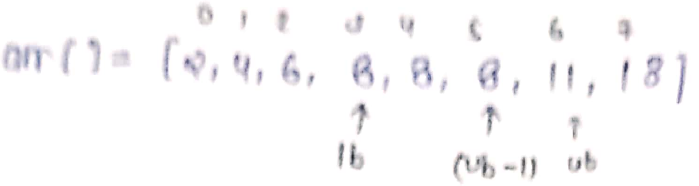
$arr[\,] = [\overset{0}{2}, \overset{1}{4}, \overset{2}{6}, \overset{3}{8}, \overset{4}{8}, \overset{5}{8}, \overset{6}{11}, \overset{7}{13}]$ Ans = [3 , 5]

→ Brute :-

```
first = -1, last = -1
for(i=0→n-1) {
    if(arr[i] == n) {
        if(first == -1) first = i;
        last = i;
    }
}
```

* TC :- O(n)
SC :- O(1)

→ Optimal :-



lower_bound → a[ind] >= x

upper_bound → a[ind] > x

edge case :- if the element does not exist (check lb(value)==x)

for e.g - n=10 (lb → 6) false

if the element does not exist but (lb==n)

for e.g - n=14 (lb - 8) false

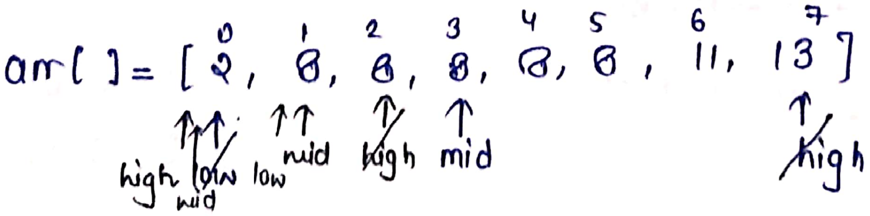
code :-

```
lowerbound ( arr, n, x )
upperbound ( arr, n, x )
occurence ( ) {
    int lb = lowerbound (arr, n, x);
    if (lb ==n || arr[lb] != x)   return {-1, -1};
    return {lb, upperbound (arr, n, K) - 1};
```

→ TC :- 2 x O(log₂ n)

SC :- O(1)

OR find first & last occurence with lower bound & upper bound -



first = ~~1~~ ~~3~~ 1 → return 1

Code :-

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

} first occurence

Note :-
fo( )
if(first ==-1) returns
lo( )

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

} last occurence

TC :- 2x O(logn)
SC :- O(1)

2) Count occurences in Array :-

arr[ ] = [1, 1, 1, 2, 2, 3, 3]

x = 3 → O/P = 2

⇒ return (last occurence − first occurence +1);

3) Search Element in Rotated Sorted Array - I :-
(unique Elements)

arr[ ] = [7, 8, 9, 1, 2, 3, 4, 5, 6] target = 1

→ **Brute :-** (Linear Search)

```
for (i=0 → n) {
    if (arr[i] ==target) {
        return i;
    }
}
```

TC :- O(n)

SC :- O(1)

Note:-

Identify the sorted part

→ **optimal :-** (Binary Search)

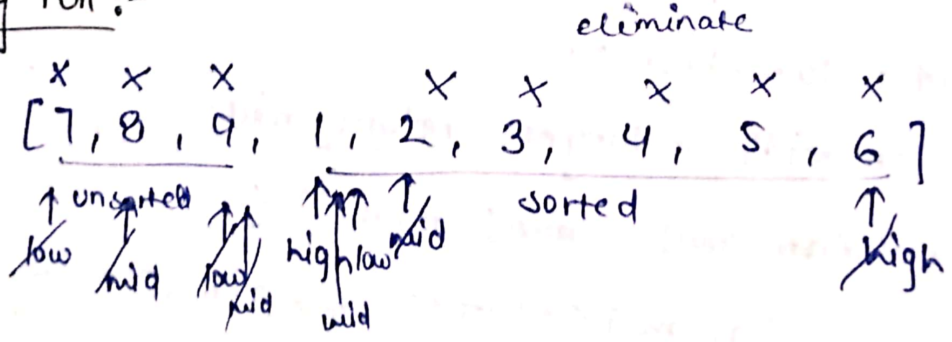
```
~~for (i=0 → n) {~~

    ~~if (arr~~
while (low <= high) {
        mid = (low + high)/2;
        if (arr[mid] == target) return mid;
left sorted → else if (arr[low] <= ~~mid~~ arr[mid]) {
                if (a[low] <=target && a[mid] > target) {
                    high = mid-1;
                }
            else {
                    low = mid +1;
            }
    }
```

```
right sorted->else{
        if(a[mid]<target && a[high]>=target){
                low = mid+1;
        }
        else{
                high = mid-1;
        }
}
```

Note:- Instead of checking low & high at both sides of array to eliminate one portion of array -
Just find the sorted & unsorted portion of array & check whether the element lies array & check whether the element lies the sorted or unsorted part; & eliminate the other part.

* TC:- O(logn)
  SC:- O(1)

* Dry run:-



return 3

4) Search in Rotated Sorted Array - II (duplicates) :-

arr[ ] = [7 8 1 2 3 3 3 4 5 6] target = 3

→ **Brute :-**

linear search

→ **optimal :-**

edge case :- [3, 1, 2, 3, 3, 3, 3]

↑ low ↑ mid ↑ high

```
ondn :- if (arr[low] == arr[mid] == arr[high]) {
            low++ , high ++
        apply on I prob.
```

TC :- $O(\log_2 n)$

SC :- $O(1)$

5) Minimum in Rotated Sorted array :-

→ **Brute :-**

Linear Search

→ **optimal :-**

```
if (a[low] <= a[~~hto~~ mid]) {
        ans = min(ans, arr[low])
        low = mid + 1;
}
else {
        ans = min(ans, arr[mid])
        high = mid - 1;
}
```

Note :-

Identify the sorted half → sorted half may or may not be the answer

TC :- $O(\log_2 n)$

SC :- $O(1)$

6) Find out how many times array is rotated :-

       0 1 2 3 4
arr( ) = ($\underbrace{3, 4, 5}$, 1, 2]   ans = 3

original arr( ) = [1, 2, $\underline{3, 4, 5}$]

→ like previous one find the index of minim
element and return.

7) Single Element in sorted Array :-

arr[ ] = [1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6]   ans = 4

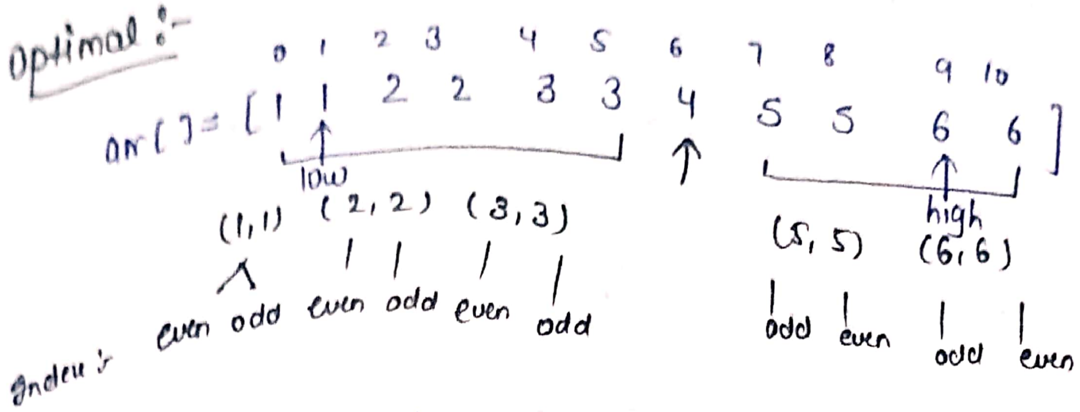
→ Brute :-

```
if (n == 1)   return arr[0];
for (i=0 → n) {
    if (i == 0) {
        if (arr[i] != arr[i+1]   return arr[i];
    }
    else if (i == n-1) {
        if (arr[i] != arr[i-1]   return arr[i];
    }
    else {
        if (arr[i] != arr[i+1]  && arr[i] != arr[i-1]) {
                return arr[i];
        }
}
```

TC :- O(N)

SC :- O(1)

→ Optimal :-



(even, odd) → element is on right half (eliminate left)

(odd, even) → element is on left half (eliminate right)

code :-

```
if (arr.size() == 1)  return arr[0];
if (arr[0] != arr[1])  return arr[0];
if (arr[n-1] != arr[n-2])  return arr[n-1];
low = 1 , high = n-2;
While (low <= high) {
    mid = (low + high)/2;
    if (arr[mid] != arr[mid+1] && arr[mid] != arr(mid-1])
            return Arr[mid];
    if ((mid%2 == 1  && arr[mid-1] == arr[mid]) ||
        (mid % 2 == 0  && arr[mid] == arr [mid+1]) {
                low = mid + 1; }  // eliminate left half
    else {
        high = mid-1;  // eliminate right half
        }
}.
return -1;
}
```

8) find Peak element :- arr[i-1] < arr[i] > arr[i+1]

arr[ ] = [1 2 3 4 5 6 7 8 (5) 1] ans = 5

arr[ ] = [1 (2) 1 3 5 (6) 4] ans = 2 or 6

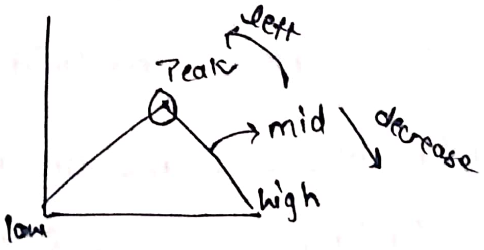
arr[ ] = -∞[1 2 3 4 5] -∞ ans = 5

arr[ ] = -∞[5 4 3 2 1] -∞ ans = 5

→ **Brute :-** (compare each & every digit)

```
if(arr.size()==1)   return arr[0];
for(i=0→n){
    if((i==0 || arr[i]>arr[i-1]) && (i==n-1 || arr[i]>arr[i+1])){
        return (arr[i]);
    }
}
return -1;
}
```

→ optimal :-



arr[ ] = [1 2 1 3 5 6 4]

⇒ arr[mid] > arr[mid-1] = Peak on right half | eliminate left half

arr[mid] > arr[mid+1] = Peak on left half | eliminate right hal

→ optimal :-

```
if(arr.size()==1) return arr[0];
if(arr[0] > arr[1]) return arr[0];
if(arr[n-1] > arr[n-2]) return arr[n-1];
while(low <= high) {
        mid=(low + high)/2;
        if(arr[mid]>arr[mid+1] && arr[mid] > arr[mid-1]){
            return arr[mid];
        }

        else if (arr[mid] > arr[mid-1]){      // eliminate left half
            low=mid+1;
        }

        else {
            high = mid-1;          // eliminate right half
        }
}
return -1;
```