17/02/2023

Searching

We have discussed an algorithm named linear search. Search means to find whether key is present or not. In linear search, in the worst case we will be comparing all the elements. Say the size of array is n, then in worst case n comparasion will happen & the time complexity = O(n)

```
for(int i=0 ; i<n ; i++) {
        if (arr[i] == key) {
            cout << "Found";
            break;
        }
}
```

The above is the code of linear search & has time complexity = O(n)

Let's say in our array we have 1000 elements then in worst case 1000 comparasions is done which is a lot.

Binary Search

If we 1000 elements, then binary search just takes 10 comparasions which is far less than 1000 comparasions.

Note→ To apply binary search, there is one condition which is the elements should be in monotonic order i.e elements should be sorted either

in increasing or decreasing order.

1 2 4 6 9 11 15

The above elements in sorted fashion & currently we are at 9 value & target = 15 which is greater than 9, hence we need to search in the right.

Working of binary search

index → 0 1 2 3 4 5 6 7 target = 15

1	3	7	9	11	13	15	19

1) Initialize start = 0 and end = n - 1 where n is the size of array.

2) Find mid index, $mid = \frac{start + end}{2}$

$$mid = \frac{0+7}{2} = 3$$

arr[3] = 9

3) Compare target value with value at mid index.

15 > 9 } This means search in right array

4) 11 13 15 19

start = 4, end = 7

$mid = \frac{4+7}{2} = 5$

Now compare target with value at mid index.

15 > 13 } Search in right array.

5)

15 19

start = 6, end = 7

mid = $\frac{6+7}{2}$ = 6

Compare target with value present at the mid index.

15 == 15 } return index

If in case, the element is not present, then return - 1 . This is basically achieved after start > end i.e. start goes ahead of end.

Code:

```
int binarySearch (int arr[], int size, int target)
{
    int start = 0;
    int end = size - 1;
    int mid = (start + end)/2;

    while (start <= end) {
        // Return mid index if found
        if (arr[mid] == target) {
            return mid;
        }
        // Search in right part
        else if (target > element arr[mid])
        {
            start = mid + 1;
        }
```

```
//Search in left part
else {
        end = mid-1;
}   mid = (start + end)/2; → Update mid with
}                            updated start & end values.
return -1; //If element not found
}
```

↑ mid is an index.

Issue in `mid = (start + end) / 2;`

There is an issue in this statement & the issue is as follows :

There can be integer overflow here. Let's say

start = $2^{31} - 1$

end = $2^{31} - 1$

$$mid = \frac{(2^{31} - 1 + 2^{31} - 1)}{2}$$ → Numerator will go out of range

Solution

$$mid = s + \frac{(e - s)}{2};$$

s → start

e → end

start = $2^{31} - 1$

end = $2^{31} - 1$

$$2^{31} - 1 + \frac{(2^{31} - 1 - 2^{31} + 1)}{2}$$

$$2^{31} - 1 + 0 \Rightarrow 2^{31} - 1$$

Hence there is no integer overflow.

Note → Other solution is $mid = \frac{start}{2} + \frac{end}{2};$

This will also won't create any integer overflow but we will follow $s + \frac{(e-s)}{2}$ only.

Time complexity of binary search

n size array

1st iteration ⇒ $\frac{n}{2^1}$

2nd iteration ⇒ $\frac{n}{2^2}$

⋮

kth iteration ⇒ $\frac{n}{2^k}$

$\frac{n}{2^k} = 1$ as if only one element is present, then we just have compare it & if found equal, then return index, else return -1.

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$k = \log_2 n$$

Time complexity = $O(k) = O(\log_2 n)$

Note → Binary search stops after we get only 1 element. Everytime the size of array reduces by half.

STL function for binary Search

STL stands for Standard Template Library and this have already predefined function of binary search.

```
vector <int> v {1, 2, 3, 4, 5, 6};
if (binary_search (v.begin(), v.end(), 3)) {
        cout << "Found";
}
else {
        cout << "Not found";
}
```

start, end, key, For Vector

To use binary_search, we need to include <algorithm> in our program.

Note →

```
int arr[] = {1, 2, 3, 4};
int size = 4;
if (binary_search (arr, arr+size, 3)) {
        cout << "Found" << endl;
}
else {
        cout << "Not found" << endl;
}
```

For Array, first index, last index, key

Also note that arr + size will be the last index.

Problem Solving

Q1 Find the first occurence of an element.

i/p →

1	3	4	4	4	4	4	6	7	9

o/p → 2

Here we can use binary search as elements

are in monotonic fashion.

Algorithm

1) start = 0

end = size - 1 = 9

mid = $\frac{start + end}{2} = \frac{0+9}{2} = 4$

arr[mid] = 4

First of all store the answer as index = 4 and search in the left part.

2)

1	3	4	4

start = 0

end = 3

mid = $\frac{0+3}{2} = 1$

arr[mid] = 3 < 4. Simply search in right part i.e start = mid + 1.

3)

4	4

start = 2

end = 3

mid = $\frac{2+3}{2} = 2$

arr[mid] = 4, answer store as index = 2, & search in left part i.e end = mid - 1.

Now start > end, end loop & answer = 2.

Also if target > arr[mid], then search in right part by start = mid + 1

Code

```
int firstOccurence(vector<int> v, int t){

    int s = 0;
    int e = v.size() - 1;
    int mid = s + (e-s)/2;
    int ans = -1;

    while(s <= e){

        if(v[mid] == t){
            ans = mid; //Store answer
            e = mid-1; //Search in left
        }
        else if(target < v[mid]){
            e = mid-1; //Search in left
        }
        else{
            s = mid+1; //Search in right
        }
        mid = s + (e-s)/2; //Update mid again.
    }
    return ans;
}
```

Time complexity same as binary search i.e O(logn)

Note → We can code the question of last occurence of an element by changing e=mid-1 inside the 1st if condition of the above code to s=mid+1 i.e we

need to search in the right part to find the last occurence of the element.

Code of Last Occurence

```
int lastOccurence (vector<int> v, int t){
                                    t = target
                                    in code
    int s = 0;
    int e = v.size() - 1;
    int mid = s + (e-s)/2;
    int ans = -1;

    while (s <= e) {

        if (v[mid] == target) {
                ans = mid; // Store answer
                s = mid + 1; // Search in right
        }
        else if (target < v[mid]){
                e = mid - 1; // Search in left
        }
        else {
                s = mid + 1; // Search in right
        }
        mid = s + (e-s)/2; // Again update mid
    }
    return ans;
}
```

Time complexity O(logn)

STL functions for first & last occurence

```
lower_bound (v.begin(), v.end(), 20);
                                 ↳ target element
```

Lower bound is used to find the first occurence & it returns an iterator. Similarly upper-bound is used to find the last occurence.

```
upper_bound (v.begin(), v.end(), 20);
                                 ↳ Target element
```

Q2 Total number of occurences of the element.

i/p → 2 4 4 4 4 5 6
o/p → 4

Total occurence = lastOccurence - firstOccurence + 1.

Code

```
int first = firstOccurence (V, 4);
int last = lastOccurence (V, 4);
cout << (last - first + 1) << endl;
```

Q3 Find the missing element.

i/p → 1 2 3 4 6 7 8
o/p → 5 (missing here)

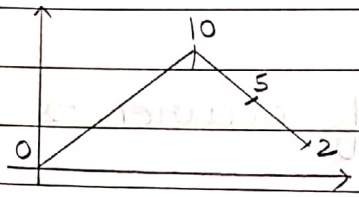
1	2	3	4	6	7	8
0	1	2	3	4	5	6

On left part, there is a pattern which i°+1 == arr[i] but on right side this pattern breaks.

Explore this question & we have to solve it with the help of binary search.

Q4 Peak element in mountain array.

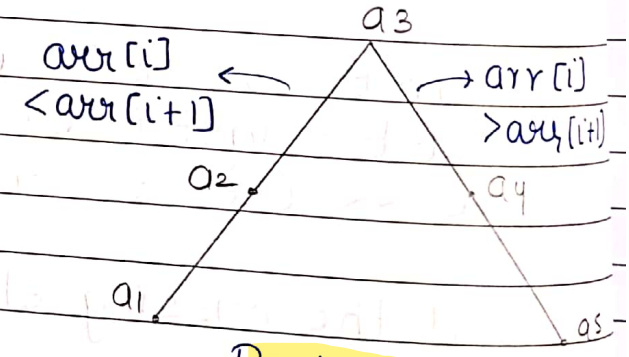
i/p → 0 10 5 2



o/p → 10

A brute force solution can be of linear search & find the maximum array value. This has time complexity = O(n). We can do it with the help of binary search. This approach will have time complexity = O(logn)

Algorithm



Peak element
arr[i] > arr[i+1] && arr[i] > arr[i-1]

1) start = 0
end = 2
mid = $\frac{0+2}{2} = 1$

arr[mid] = 10

2) if (arr[mid] > arr[mid+1]) { }

✓ mid element may be a peak element or in the descending line

✓ mid in line a3, a4, a5.

3) if (arr[mid] < arr[mid+1]) { }

✓ mid element can not be peak element & check in the right part i.e s = mid+1;

```
else {
    e = mid;
}
```

Code

```
int peakElement (int arr[], int size){

        int s = 0;
        int e = size - 1;
        int mid = s + (e-s)/2;
        while(s<e){
            if(arr[mid] < arr[mid+1]){
                s = mid + 1;    //right of mid
            }
            else{
                e = mid;
            }
            mid = s + (e-s)/2;
        }
        return s; //We can return e; also
}
```

Note → if (arr[mid] < arr[mid+1]) { -- }
else { -- }

↳ This means that we are on either peak element or on the descending line.

If we write e = mid - 1, then if we are on peak element, then we might loose it & hence e = mid is written.

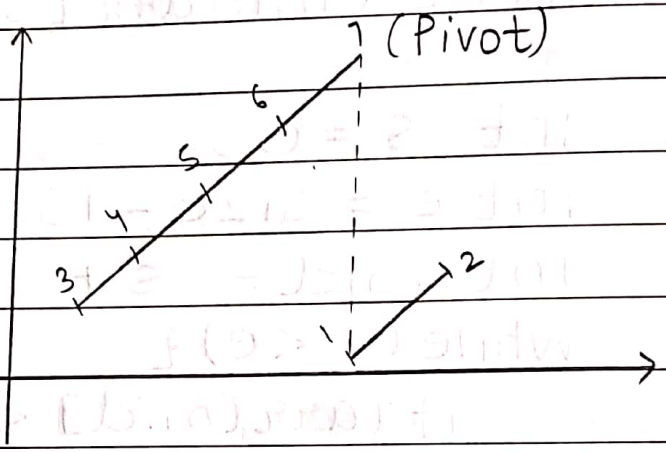
s = mid; } while (s <= e) { -- }
e = mid;

Infinite Loop

18/02/2023

Q1 Find pivot element in an array

i/p →	3	4	5	6	7	1	2



O/p → 7 or 1

Order breaks at the pivot element. At some platform 7 is pivot & at some 1 is the pivot element. Here we will consider 7 is the pivot element

The element at which the monotonic order breaks is the pivot element.

Brute force approach can be the linear search, but this approach has time complexity of O(n) but can we do in O(log n).

Here our code can get stuck & hence we can explicitly handle these 2 cases. or numbers

Algorithm

1) Start = 0
end = size - 1 = 7 - 1 = 6
mid = $\frac{0+6}{2}$ = 3
arr[mid] = 6

Suppose that our mid comes at 4th index, then arr[mid] = 7 & arr[mid+1] = 1

```
if (arr[mid] > arr[mid+1]){
        return mid;
}
if (arr[mid-1] > arr[mid]){
        return mid-1;
}
```

→ These are the 2 cases which we are explicitly handling.

Also we have to make sure mid+1 & mid-1 is a valid index.

2) Now only 2 conditions are left i.e. to search in right part or left part. We know that pivot is the maximum element. Compare with starting element.

```
if (arr[s] >= arr[mid]) {
        e = mid - 1; // Left part
}
else if (arr[s] < arr[mid]) {
        s = mid + 1; // Right part
}
```

Code

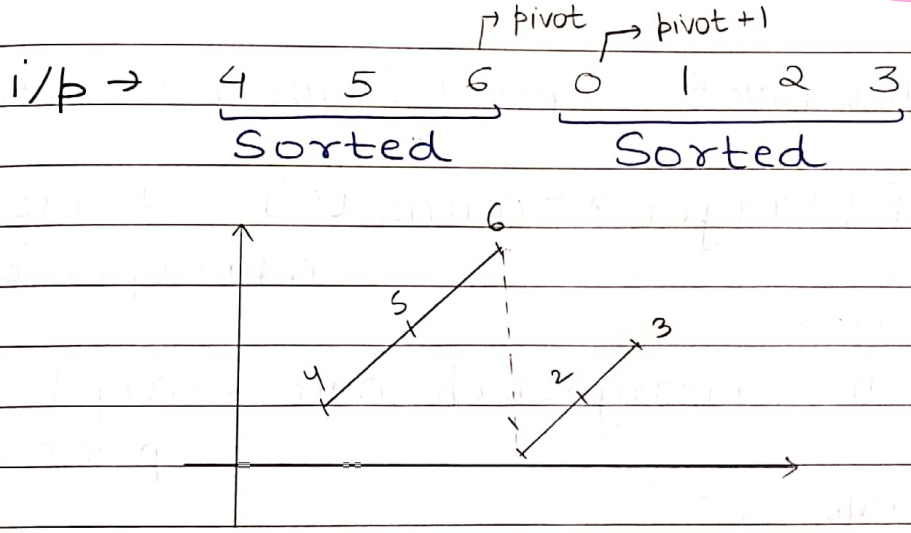
```
int pivotElement (vector <int> arr){

    int s = 0;
    int e = arr.size() - 1;
    int mid = s + (e-s)/2;

    while (s < e) {
        // valid index
        if (mid+1 < arr.size() && arr[mid] > arr[mid+1])
            return mid;
        // valid index
        if (mid-1 >= 0 && arr[mid-1] > arr[mid])
        {
            return mid-1;
        }
        if (arr[s] >= arr[mid]) {
            e = mid-1;
        }
```

```
else if (arr[s] < arr[mid]) {
        s = mid ; // As in while we
                    have used s<e
}
mid = s + (e-s)/2;
}
return s ; → For single element in array
}          & not gone in while loop.
```

Q2 Search in rotated sorted array.



Suppose that we need to find 2 in the array with the help of binary search.

Algorithm

1) Find pivot element which is 6 here. Now compare key = 2 which we need to compare with pivot.

2 < 6, hence we need to search for the element in sorted array - 2

4 5 6 | 0 1 2 3

array - 1 | array - 2

2) We can reuse the code of pivot element & then decide in which array we need to search the element & once we get to know the array, then simply apply the binary search.

Code

```
int search (vector<int>& nums, int t){

    int pivot = pivotElement (nums);

    if (target >= nums [0] && target <=
                        nums [pivot]) {
  // Search in left array
 int ans = binarySearch (nums, target, 0,
                        pivot);

  return ans;
 }
   // valid index
if (pivot +1 < nums.size() && target
                    >= nums [pivot+1] &&
            target <= nums [nums.size()
                        -1]) {

   // Search in right array
  int ans = binarySearch (nums, target,
                pivot+1, nums.size()-1);

    return ans;
}
return -1;
}
```

The question now comes to our mind that what is rotation in the array.

0 1 2 3 4 5 6 → sorted array

1st rotation

6 0 1 2 3 4 5

2nd rotation

5 6 0 1 2 3 4

3rd rotation

4 5 6 0 1 2 3

We can do further rotations on the array.

Note → Other way is that we can apply binary search on both arrays & the one who is returning index != -1 will be the answer.

Q3 Square root of a number using binary search

i/p → 12
o/p → 3 (only integer part)

Square root of a number n will lie within 0 to n. This is known as the search space

n = 10

Search Space: 0 1 2 3 4 5 6 7 8 9 10

s ↑ mid ↑ e

for sqrt of 12

1) start = 0
end = size - 1 = 10
mid = 5

arr [mid] → 5
5 * 5 = 25
25 > target = 10
So we need to search in left part.

2) start = 0
end = 4
mid = 2

arr [mid] = 2
2 x 2 = 4
4 < target = 10
Store the answer and then search in the right part of array.

3) start = 3
end = 4
mid = $\frac{3+4}{2}$ = 3
arr [mid] = 3

3 * 3 = 9
9 < 10
Store answer & search in right part.

4) start = 4 } Both becomes equal.
end = 4

Code

```
int squareRoot (int n) {
    int ans = -1
    int s = 0;
    int e = n;
    int mid = s + (e-s)/2;
    int target = n;
    while (s <= e) {

        if (mid * mid == target) {
            return mid;
        }
        if (mid * mid > target) {
            e = mid - 1;        //search in left
        }
        else {
            ans = mid;          //searchg in right + store
            s = mid + 1;
        }
        mid = s + (e - s)/2;
    }
    return ans;
}
```

Note → We are storing the ans = mid because it might happen that if we go to right & we won't get the answer, then we get the ans = -1 only which is wrong answer.

Now we need to find the decimal part.

ans = 3

3.1 → 3.1 * 3.1 <= 10 } Now check
3.2 → 3.1 * 3.1 > 10
3.3
3.4
⋮
3.9

ans = 3.1

3.11 3.11 * 3.11 <= 10
3.12 3.12 * 3.12 <= 10
3.13 ⋮
⋮ 3.16 * 3.16 <= 10 ans = 3.16
3.17 * 3.17 > 10
3.19

Additional code — val after decimal

```
double ans = squareRoot(10);
int precision = 3;
double step = 0.1;

for (int i=0; i<precision; i++){
    for (double j=ans; j*j<=n; j=j+step)
        ans = j;
    }
    step = step/10;    // changing decimal places every time
}
cout << ans << endl;
```

Q4 Binary Search in 2D matrix

i/p →

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

n → no. of rows
m → no. of columns

Algorithm

start = 0

end = n * m - 1 = 5 * 5 - 1 = 25 - 1 = 24

$mid = \frac{start + end}{2} = \frac{0 + 24}{2} = 12$

arr [ ] [ ]
↳ row Index ↳ col Index

row Index → mid / cols (cols → m)
col Index → mid % cols
} Formulae

element = arr [rowIndex] [colIndex];

if (element == target) return true;
if (element > target) e = mid - 1; Left part
else s = mid + 1; Right part

Code

```
bool binarySearch (int arr [][5], int r,
                   int c, int t) {

int s = 0;
int e = r * c - 1;          //row*col -1
```

```
int mid = s + (e-s)/2;

while (s<=e) {

    int rowIndex = mid/cols;
    int colIndex = mid%cols;
    int element = arr[rowIndex][colIndex];
    if(element == t) {
        return true;
    }

    if (element < t) {
        s = mid+1; // left part
    }
    else {
        e = mid-1; // Right part
    }
}
return false;
}
```

19/02/2023

Q1 Binary Search in a nearly sorted array.

i/p → 10 3 40 20 50 80 70

First of all we need to know what is a nearly sorted array.

If the above array was sorted, then elements would be

3 10 20 40 50 70 80

In the nearly sorted array, the element present at the i^{th} index in the sorted array can be present in 3 places i.e $(i-1)^{th}$ index or i^{th} index or $(i+1)^{th}$ index.

$i=0$ in sorted array can be present at -1 or 0 or 1st index in nearly sorted array & is found at 1st index.

$i=1$ in sorted array can be present at 0, 1 or 2nd index in the nearly sorted array & hence is found at 0^{th} index.
Similarly we can verify for all the elements as the condition will always be true.

Approach can be that we can apply linear search however it has time complexity = $O(n)$ but can we solve in the $\log n$ approach / time complexity.
Other approach can be like we can sort the array & then apply binary search but the complexity of this solution will be $O(n \log n)$.

Algorithm

Sorted	Nearly sorted
1) Find mid = $\frac{s+e}{2}$ & compare arr[mid] and target.	Find mid & compare target with value at mid, mid-1 or mid+1 index and return index in each case.

2) target > arr[mid], search in right side i.e s = mid + 1

target > arr[mid] search in right side i.e s = mid + 2. We added 2 as we have already checked mid + 1 index.

3) target < arr[mid], Search in left side i.e e = mid - 1

Here we will search in left side i.e e = mid - 2 as we have already checked mid - 1 index.

Code

```
int binarySearch (vector <int> arr , int target){

    int s = 0;
    int e = arr.size() - 1;
    int mid = s + (e-s)/2;
    while (s <= e) {
        if (arr[mid] == target) {
                return mid;
        }        ↱ valid index check
        if (mid-1 >= 0 && arr[mid-1] == target){
                return mid-1;
        }
        if (mid+1 < arr.size() && arr[mid+1] == target){
                return mid+1;
        }
```

```
if (target > arr[mid]) {
        s = mid + 2;
}
else {
        e = mid - 2;
}
mid = s + (e-s)/2;
}
return -1;
}
```

Note → We can optimize the above code by modifying the condition of valid index check to other condition.

$mid-1 >= 0 \rightarrow mid-1 >= s$

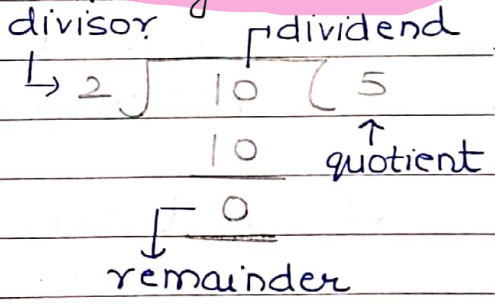
$mid+1 < arr.size() \rightarrow mid+1 <= e$

Time complexity : O(logn) same as that of binary search.

Q2 Divide two numbers using binary search

i/p → dividend = 10
divisor = 2
quotient = ?

o/p → 5



Now the question is how can we use binary search algorithm to find the quotient. The similar kind of approach of finding the square root of a number using binary search.

Here we will take the search space from 0 to dividend.

Dividend = Quotient * Divisor + Remainder
Also we can modify the above formulae to

quotient * remainder <= dividend

Algorithm for 10/2

1) start = 0
end = 10
mid = $\frac{0+10}{2}$ = 5

mid * divisor = 5 * 2 = =10 return mid;

Algorithm for 22/7

1) Start = 0
end = 22
mid = $\frac{0+22}{2}$ = 11

mid * divisor = 11 * 7 = 77 > 22
move to the left side by e = mid-1.

2) Start = 0
end = 10
mid = $\frac{0+10}{2}$ = 5

mid * divisor = 5 * 7 = 35 > 22
Again search in the left side by e = mid-1

3) start = 0
end = 4
mid = $\frac{0+4}{2}$ = 2

mid * divisor = 2 * 7 = 14 <= 22
(i) Store ans as ans = mid as it might be the answer.

(ii) Search in right part
s = mid + 1;

4) start = 3
end = 4
mid = $\frac{3+4}{2}$ = 3

mid * divisor = 3 * 7 = 21
(i) Store ans as ans = mid
ans = 3
(ii) Search in right part
s = mid + 1

5) start = 4
end = 4
mid = $\frac{4+4}{2}$ = 4

mid * divisor = 4 * 7 = 28 > 22
e = mid - 1 ; } Search in the left part

Now start > end, exit the loop.

Code
```
int solve (int dividend, int divisor) {
```

```
int s = 0;
int e = abs(dividend);
int mid = s + (e-s)/2;
int ans = 1;

while (s <= e) {

if (abs(mid * divisor) == abs(dividend)){
        ans = mid; // Got the answer
        break;
}

if (abs(mid * divisor) > abs(dividend)){
        e = mid - 1; // Search in left
}
else {
    ans = mid; // Store answer
    s = mid + 1; // Search in right
}
mid = s + (e-s)/2;
}
// To handle the -ve case
if ((divisor < 0 && dividend < 0) || (divisor > 0 &&
                                      dividend > 0)){
        return ans;
}
else {
    return -ans;
}
}
```

both +ve

both -ve

one is -ve

Note→ abs(-3); → 3
abs(3); → 3

abs() always return the +ve value.

Q3 Find the odd occuring element in the array.

i/p → 1 1 2 2 3 3 4 4 3 600 600 4 4

In this question all the elements occur even number of times except one. Also all the repeating occurence of element appear in the pairs and the pairs are not adjacent. Also note that there can not be more than 2 consecutive occurence of any element. We have to find the element that appears odd number of times.

Algorithm - 1

Brute force approach can be doing XOR of all the elements of the array. Time complexity of this approach is O(n) but can we

Algorithm - 2

1	1	2	2	3	3	4	4	3	600	600	4	4
index 0	1	2	3	4	5	6	7	ans 8	9	10	11	12

Left of ans → pair

First value on the even index

Second value on the odd index

Right of ans → pair

First value on odd index

Last/second value on even index.

Also from observation, ans always lies on the even index.

Note → 0th index will be considered as even in this question.

```
s = 0
e = n-1
mid = (s+e)/2   } Change in code to s+(e-s)/2;
while (s<=e){
 if (s==e)                  } Only one element
      return s;             } remaining case

Case-1 if (mid%2==0){ → Even index
                    ↳ left side of answer
     if (arr[mid]==arr[mid+1]){
          //Search in the right part
          s=mid+2;
                    ↳ As mid+1 already
                      checked.
     }
     else{
          e=mid; // if we do e=mid-1, then
     }              we might loose the answer
}                   as mid may be an answer.

Case-2 else { → odd index
                ↳ left part as mid is odd
     if (arr[mid-1]==arr[mid]){
          //Search in right part
          s=mid+1;
                    ↳ Here 1 & not 2 because
                      mid+1 is not explored
     }
```

```
else {
    e = mid-1; // Here mid can not be answer as the mid index is odd but answer is at even index.
}
```

Code

```
int oddOccurence (vector <int> v) {
    int s = 0;
    int e = v.size() -1;
    int mid = s + (e-s)/2;
    while (s <= e) {
        if (s == e)                    // one element left in array
            return s;

        if (mid % 2 == 0) {            //even index

            if (v[mid] == v[mid+1]){ //left part
                s = mid + 2;           //move to right
            }
            else { // we are in right part
                e = mid;               //move left
            }                          //and preserve mid
        }
        else {                         //odd index
            if (v[mid] == v[mid-1]) {//left part
                s = mid + 1;           //move right
            }
            else { // we are in right part
                e = mid - 1;           //move left
            }
        }
}
```

```
        mid = s + (e-s)/2;
    }
    return -1;
}
```

Types of questions in binary search

1) Classic Questions like lower bound, upper bound etc.

2) Search space predicate function question such as aggressive cows, book allocation etc.

3) Observing index value like the question of finding odd occuring element.

24/02/2023

Selection Sort

First of all we need to understand what is sorting. Sorting is basically rearranging elements in either increasing or decreasing order.

1, 3, 5, 4, 7, 2 → These elements are not in sorted order as initially it was increasing but 5 to 4 is of decreasing order & hence we can say it is not sorted.

In selection sort, an element is picked & then is placed at correct place. In this we have to place the minimum element at the right place.

Algorithm of Selection Sort

10	1	4	8	5	7

Step-1 Find the minimum element from the array & place at $i = 0$.

```
min = 1
swap (arr[0], arr[minIndex]);
```

1	10	4	8	5	7

Step-2 Find minimum element in subarray start from index = 1 to n-1 & place at 1st index.

```
min = 4
swap (arr[1], arr[minIndex]);
```

1	4	10	8	5	7

Step-3 Find minimum in sub-array start from index = 2 to n-1 & place at 2nd index.

```
min = 5
swap (arr[2], arr[minIndex]);
```

1	4	5	8	10	7

Step-4 Find minimum in sub-array start from index = 3 to n-1 & place at 3rd index

```
min = 7
swap (arr [3], arr [minIndex]);
```

1	4	5	7	10	8

Step-5 Find minimum in sub-array start from index = 4 to n-1 & place at 4th index

```
min = 8
swap (arr [4], arr [minIndex]);
```

1	4	5	7	8	10

Now only 1 element remaining, so need to check it as there is no element on its right and hence we get the sorted array.

Note → Every step we do is termed as round or parse.

Step-1 = Round-1 = Parse-1

We can observe that for 6 elements, 5 rounds are required to sort the array. Hence for n elements, n-1 rounds are required to sort the array via selection Sort.

Code

```
void selectionSort (vector <int> arr){

int n = arr.size();
for (int i=0; i<n-1; i++) {
      // Assuming minimum element to be current element
      int minIndex = i;
      for (int j=i+1; j<n; j++){
       // Comparing with minimum Index.
            if (arr[minIndex] > arr[j]){
                  minIndex = j;
            }       //Updating minimum
      }
      // Placing min element at correct position
      swap (arr[minIndex], arr[i]);
}
}
```

Note → We have started inner loop from i+1 as we have already taken minIndex as i & does not make sense to compare with itself.

Time complexity
$O(n^2)$ as the loops are nested, hence time complexity gets multiplied.

Space complexity
$O(1)$ as only variables have been created & we don't have to consider space of the input.

Bubble Sort

The logic behind this sorting technique is that in the i^{th} round, i^{th} largest element will be placed at the right position.

Algorithm for bubble sort

10 1 7 6 14 9

Round - 1

* 10 > 1 → swap
  1 10 7 6 14 9
* 10 > 7 → swap
  1 7 10 6 14 9
* 10 > 6 → swap
  1 7 6 10 14 9
* 10 > 14 → no swap
* 14 > 9 → swap
  1 7 6 10 9 14

At right place ↑

Round - 2

1 7 6 10 9 | 14

Sorted boundary

* 1 > 7 → no swap
* 7 > 6 → swap
  1 6 7 10 9 14
* 7 > 10 → no swap
* 10 > 9 → swap
  1 6 7 9 10 14

Round - 3

1 6 7 9 | 10 14

Sorted boundary

* 1 > 6
* 6 > 7
* 7 > 9

} No swap in any case.

Round - 4

1 6 7 | 9 10 14

Sorted boundary

* 1 > 6
* 6 > 7

} No swap in any case

Round - 5

1 6 | 7 9 10 14

Sorted boundary

* 1 > 6 → no swap

Now we don't have to go in round - 6 as only one element is left & hence we can't compare it with any element. Hence we get sorted array in n-1 rounds where n is the no. of elements.

Optimization in Bubble Sort

Also we can observe that in round - 3 there was no swap done & hence the array gets already sorted & We don't have to go in further rounds.

Code

```
void bubbleSort (vector <int> & arr) {

    int n = arr.size();
    for (int i=0; i<n-1; i++) {
        bool swapped = false;
        for (int j = 1; j < n-i; j++) {

            if (arr[j-1] > arr[j]) {
                swapped = true;
                swap (arr[j-1], arr[j]);
            }
        }
        if (!swapped) {
            break;
        }
    }
}
```

Best case Time Complexity
Best case occurs when the array is already sorted. Here the time complexity is $O(n)$ as only $n-1$ comparasions will be done.
Normal case Time complexity $O(n^2)$
Worst case Time complexity
Worst case occurs when the array is reverse sorted & hence time complexity here is $O(n^2)$.

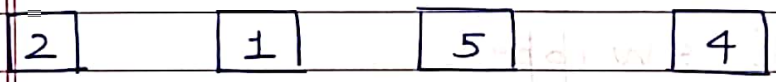
By optimization, best case time complexity

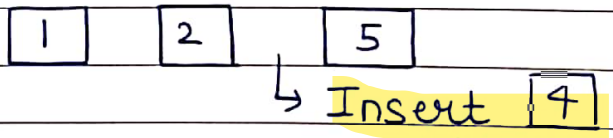
reduces to O(n).

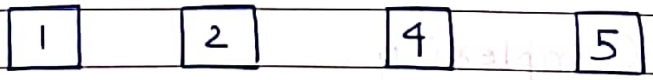
Space complexity
O(1) as only variables are created.

Note → Use case of selection sort is incase of small arrays.
Use case of bubble sort is when ith largest elements to be put at correct place.

Insertion Sort
This means to insert the element at right place.







→ Sorted array

Algorithm for insertion sort

10 1 7 6 14 9

1) 10 is already at right place or will automatically be placed at right place.

2) Pick element at index = 1

1 < 10, shift 10 & place 1 there. There is no swapping involved.

1 10 7 6 14 9

Sorted

3) Pick element at index = 2

7 < 10
7 > 1
} 7 should come in blw 1 & 10.

So shift 10 & copy 7 at the empty place.

1 7 10 6 14 9

Sorted

4) Pick element at index = 3

6 < 10
6 < 7
6 > 1

Shift 10, then shift 7 & then copy 6 at the empty place

1 6 7 10 14 9

Sorted

5) Pick element at index = 4

14 > 10 → nothing to do.

1 6 7 10 14 9

Sorted

6) Pick element at index = 5

9 < 14
9 < 10
9 > 7

Shift 14, then shift 10 & then copy 9 to

the empty place

1 6 7 9 10 14

Hence the array is sorted now.

Code

```
void insertionSort (vector <int> arr) {

        int n = arr.size();
        for (int i=1; i<n; i++) {
            int val = arr[i]; // Pick element
            int j = i-1;
            for( ; j>=0; j--) {
                //Compare element
                if(arr[j] > val) {
  //shifting operation  arr[j+1] = val;
                }
                else {
                    break;
                }
            }
  // Copy step arr[j+1] = val;
        }
}
```

Time complexity

$O(n^2)$ in normal & worst case.

$O(n)$ in the best case i.e already sorted.

Space complexity

$O(1)$ as only variables are created.

Use case of insertion sorted is in case of small arrays or when array is partially sorted.

Inbuilt sort function
sort (arr.begin(), arr.end()); is used to sort the vector.
We need to include algorithm header file to use this inbuilt function.