Single element that appears in sorted array with duplicate

BRUTE FORCE :

xor all the elements the duplicate elements cancel each other while only the element that is not a duplicate will remain in ans.

```
int singleNonDuplicate(vector<int>& arr)
{
  // Write your code here
  int xr=0;
  for(auto num:arr)
  {
    xr^=num;
  }
  return xr;
}
```

- Time Complexity : O(N)
- Space Complexity : O(1)

Optimal Approach :

As it is sorted we need a search space to determine the element.

A small observation using example :

1	1	2	2	3	4	4
0	1	2	3	4	5	6

Here when we look at the single element it partitions the array into two halves left half and right half

in the left half at even index we find the first instance of repeating elem and at odd index we find the second instance this is vice versa in right half.

- left index:

1st index - even

2nd index - odd

- right index:

1st index : odd

2nd index : even

The intuition is that when we see an element, if we know its index and whether it is the first instance or the second instance, we can decide whether we are presently in the left array or right array. Moreover, we can decide which direction we need to move to find the breakpoint. We need to find this breakpoint between our left array and the right array.

low=0 and high =second last element because our motive is to shrink the left half so that the value we get at low will be the non repeating element.

2nd step is to find mid. for mid we will check the left and right of the mid element for this example mid is 3 and element is 2. We need to shrink left and right half based on the fulfilment of this condition. In this case the mid is odd and first instance is found in left half so we are currently in left half and we need to shrink it so we shift the low to mid+1.This goes on until low crosses high.

consider mid is always the first instance

Code :

```
int singleNonDuplicate(vector<int>& arr)
{
  // Write your code here
  int n=arr.size();
  int low=0,high=n-2;
  while(low<=high)
  {
    int mid=(low+high)/2;
    if(arr[mid]==arr[mid^1])
    {
      low=mid+1;
    }
    else
    {
      high=mid-1;
    }

  }
  return arr[low];
}
```

In the code above instead of checking whether mid is even or odd and we are in left or right half we simply xor with 1 to find the next instance if mid is even then it will given the next odd number and if it is not equal we definitely know it is right half and we shift high and vice versa so the comparision for mid is even or odd and to check if next instance is at next index we make use of xor.

//previous code that could be written insteadof xor condition.

```
if (mid % 2 == 0)
{
    if (nums[mid] != nums[mid + 1])
    //Checking whether we are in right half

        high = mid - 1; //Shrinking the right half
    else
        low = mid + 1; //Shrinking the left half
    } else {

    //Checking whether we are in right half
    if (nums[mid] == nums[mid + 1])
        high = mid - 1; //Shrinking the right half
    else
        low = mid + 1; //Shrinking the left half
}
```