# Binary Search Problems

| | |
|---|---|
| Created | @July 15, 2022 7:58 PM |
| Class | |
| Type | |
| Materials | |
| Reviewed | ☐ |

## Problem 1

### Given a sorted array and an element X, check if the element is repeated.

Example: [1,2,3,3,4,5,5,6,7,7,8,8,8,9,11,13,13] , X = 13 → ans yes

#### Hint1: Element is already given to you

#### Hint2: Figure out the position of repeating element

### Solution:

#### Brute Force:

Just apply a basic linear search and check for any index `i` the adjacent index value is same or not ?

```
// O(n)
function checkDuplicate(arr, x) {
  for(let i = 1; i < arr.length; i++) {
    if(arr[i] == x && arr[i-1] == x) return true;
  }
  return false;
}

function checkDuplicate1(arr, x) {
  for(let i = 0; i < arr.length - 1; i++) {
    if(arr[i] == x && arr[i+1] == x) return true;
  }
  return false;
}
```

## Can we optimise ?

The only thing that problem demands is to check whether X is present or not and if present whether it is repeating itself or not ?

- So to check if X is present or not in a sorted array, we can use `Binary Search`
- And we know if the element X will be repeating then if X is at index `i` then the other occurrence will be at either `i-1` or `i+1`

So while applying binary if we got the element then we can check whether the adjacent index is having X or not ?

```
function checkDuplicationOptimised(arr, x) {
  let lo = 0, hi = arr.length - 1;
  while(lo <= hi) {
    let mid = lo + Math.floor((hi - lo)/2);
    if(arr[mid] == x) {
      return true;
    }
    else if (arr[mid] > x) {
      hi = mid - 1;
    } else {
      lo = mid + 1;
    }
  }
  return false;
}
```

The above implementation will check if x is present or not in the array ?

If we want to check if x is repeating or not then we will check adjacent indexes of mid also

```
function checkDuplicationOptimised(arr, x) {
  /*
  * Time: O(logn)
  * Space: O(1)
  */
  let lo = 0, hi = arr.length - 1;
```

```
    while(lo <= hi) {
      let mid = lo + Math.floor((hi - lo)/2);
      if(arr[mid] == x) {
        if(mid - 1 >= 0 && arr[mid - 1] == x) return true;
        if(mid + 1 < arr.length && arr[mid + 1] == x) return true;
        return false; // because x is not repeating
      }
      else if (arr[mid] > x) {
        hi = mid - 1;
      } else {
        lo = mid + 1;
      }
    }
    return false; // x was not present
  }
```

So in order to check the left and right adjacent we will also if the indexes `mid - 1` and `mid + 1` are valid or not ?

# Problem 2

## Given a sorted array where every element is present twice except one element. Find the single occurring element.  (Element is not given)

Example: [1,1,4,4,5,5,11,17,17] → ans 11

Note: Try to do it in less than O(n)

## Brute Force Solution :

We can prepare a frequency map and then check the element with frequency 1. Time: O(n) and Space: O(n)

There can two more solutions of O(n) time where we can solve the question without extra space.

1. We can do a linear search. For every element at index `i` check if `i+1` and `i-1` are both different or not. If both are different then element at index `i` is the answer. Time: O(n) Space: O(1)
2. We can use the property of xor, that is xor of two same elements is 0 and xor of 0 and x is x.

We will take the xor of all the elements, and then what will happen is repeated elements will eliminate each other as they are present twice only, and the unique element will be left. This solution works even for unsorted arrays.

```
function findUnique(arr){
  // Time: O(n) Space: O(1)
  let ans = 0;
  for(let i = 0; i < arr.length; i++) {
    ans = ans ^ arr[i];
  }
  return ans;
}
```

## Can we optimise ?

Now because of the fact that array is sorted, may be we can try to find a property based on which we can divide our array into two parts such that the two parts are different based on that property. If we are able to do so, then Binary Search may be applicable.

Example: [1,1,4,4,5,5,11,17,17, 20, 20]

If you see the elements before the unique element, then for every repeating pair, the first occurrence of the element of the repeating pair is at even index and the second occurrence is at odd index.

Whereas after the unique element, the first occurrence of the repeating pair, is at odd index and the second occurrence is at even.

Explanation with example: Before 11, we have 1, 4 and 5 repeating. The first occurrence of 1 is at index 0 and the second occurrence is at index 1

| element | First occurrence | Second Occurrence |
|---|---|---|
| 1 | 0 even | 1 odd |
| 4 | 2 even | 3 odd |
| 5 | 4 even | 5 odd |

After 11, the values have the following indexes

| element | First Occurrence | Second Occurrence |
|---|---|---|
| 17 | 7 odd | 8 even |
| | | |

| | | |
|---|---|---|
| 20 | 9 odd | 10 even |
| | | |

Now we have to implement binary search such that, we can detect whether we are on the left side of unique element or on the right side of the unique element or we are sitting on the unique element.

```
function uniqueElement(arr) {
  // Time: O(logn) Space: O(1)
  let lo = 0, hi = arr.length - 1;
  while(lo <= hi) {
    let mid = lo + Math.floor((hi - lo) / 2);
    if(mid - 1 >= 0 && mid + 1 < arr.length && arr[mid] != arr[mid+1] && arr[mid] != arr[m
id-1]) {
      // we are on the unique element
      return arr[mid];
    }
    if(mid == 0 && mid + 1 < arr.length && arr[mid] != arr[mid+1]) {
      return arr[mid]; // [1,2,2,3,3,4,4]
    }
    if(mid == arr.length - 1 && mid - 1 >= 0 && arr[mid] != arr[mid-1]) {
      return arr[mid]; // [1,1,2,2,3,3,4]
    }
    if((mid+1 < arr.length && arr[mid] == arr[mid+1] && mid%2 == 0) ||
        (mid - 1 >= 0 && arr[mid-1] == arr[mid] && (mid-1)%2 == 0)) {
      // we are on the left of unique element
      lo = mid + 1;
    } else {
      // we are on right of unique element
      hi = mid - 1;
    }
  }
  return undefined;
}
```

## Can the above solution work for unsorted arrays ?

Now if we technically see the solution, we are not using the property of array being sorted anywhere. The solution will not work for any kind of unsorted array, it will only work when the pairs exist together.

Example: [1,1,-1,-1,3,3,99,13,13,6,6] → in this kind of unsorted arrangement the algorithm works because the pairs are sorted.

Where as for something like [1,2,11,2,13,1,13] → It will not work because the property of pairs starting from even index from left of unique and odd index from right of unique doesn't exist here.

Hence we can see, it doesn't matter whether the array is sorted or not, binary search is applicable if the property that we are able to divide our array in two parts and both of them are distinguishable based on the property then we can apply binary search.

## Problem 3

### Given an array of length n, where elements are sorted, all the elements are unique but one element is present twice, and the elements lie in the range [0,n-2]. Find the repeating element.

Example: [0,1,2,3,3,4,5,6,7,8] → ans is 3

Note: try to solve in less than O(n)

#### Hint: Can we relate the elements with their indexes ?

### Brute force:

We can use a frequency map, and if we don't want to use extra space, we can do linear search.

### How to optimise

We can observe a simple relation between elements and indexes before the repeating element and after the repeating element.

#### Before the repeating element

| Element | index |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| | |

## After the repeating element

| Element | index |
|---|---|
| 4 | 5 |
| 5 | 6 |
| 6 | 7 |
| 7 | 8 |
| 8 | 9 |

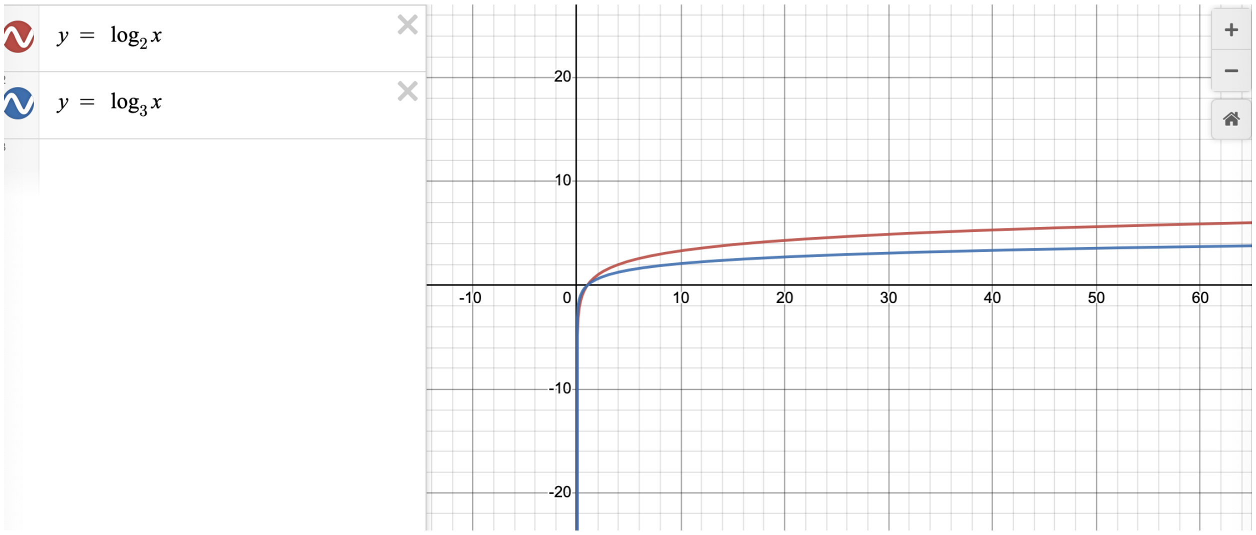
So according to the observation, everything on the left of repeating element has a property that `arr[i] == i` and everything to the right of repeating element has a property that `arr[i] == i-1`

So based on this property we can divide our search space into two parts and apply binary search.

```
function findRepeating(arr) {
  /**
  * Time: O(logn) Space: O(1)
  */
  let lo = 0, hi = arr.length - 1;
  while(lo <= hi) {
    let mid = lo + Math.floor((hi - lo) / 2);
    if(mid + 1 < arr.length && arr[mid] == arr[mid+1]) {
      return arr[mid];
    }
    if(arr[mid] == mid-1) {
      // we are on the right side of repeating element
      hi = mid - 1;
    } else {
      // we are on the left side of repeating element
      lo = mid + 1;
    }
  }
  return undefined;
}
```

## Homework

NOTE: log3n < log2n



Now knowing the above fact that, log base 3 is smaller, think that, binary search when divides the array into two parts leads to log2n comparisons, what if instead of dividing into two parts we start dividing the array into 3 parts ? Do you think that will be faster as log3n < log2n, because the search space will form smaller buckets this time.