Q1. Given an array of integer nums and an integer target, return indices of the two numbers such that they add up to the target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example:
Input: nums = [2,7,11,15], target = 9
Output0 [0,1]

Explanation: Because nums[0] + nums[1] == 9, we return [0, 1]

Answer:
The approach towards this solution lies behind the difference between the target number and the iterating or current number.
For example: In the given array nums = [2,7,11,15] and target = 9
Let's say we select the first number i.e nums[0] = 2, then we need to find the number that suits the difference between the current number and target,
Number to find = target - num[0]
i.e 9 - 2 = 7
The rest of the program will try to find whether the array has the number 7 or not. If the number 7 is not available, it will move to the next index and repeats the same process.

Time Complexity and space complexity in this program will be O(n) as we iterate over every single item using the map to store the pairs.

Code:

```
var twoSum = function (nums, target) {
   let mp = new Map()

   for (let i = 0; i < nums.length; i++) {
    const diff = target - nums[i]

    if (diff in mp) return [i, mp[diff]]

    mp[nums[i]] = i
   }
}
```

Q2. Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums that are not equal to val.

Consider the number of elements in nums that are not equal to val be k, to get accepted, you need to do the following things:

- Change the array nums such that the first k elements of nums contain elements that are not equal to val. The remaining elements of nums are not important as well as the size of nums.
- Return k.

Example : Input: nums = [3,2,2,3], val = 3 Output: 2, nums = [2,2,*,*]

Explanation: Your function should return k = 2, with the first two elements of nums being 2. It does not matter what you leave beyond the returned k (hence they are underscores)

Answer

The question says to remove all the occurrences of the given value or 'val' from the array 'nums' while returning the new array and new length (k) after the entire removal process.
In this process, apart from array and val, we will declare a variable 'count', that will store the new array exclusive of the removed occurrences.
So the approach is that the count variable keeps the tracks on all the elements left after excluding the val and one loop will go through the original array matching the elements with val. If the value is equal, the value will be ignored and iteration will increment. But if the value is not equal, the value should be placed in the 'count' array and the index will be incremented by 1 for the next iteration.
The time complexity of this method is O(n) as we are looping only once. The space complexity is O(1) as we are not using any additional space.

Code

```
function removeElement(nums, val) {
    // Counter for keeping track of elements other than val
    let count = 0;
    // Loop through all the elements of the array
    for (let i = 0; i < nums.length; i++) {
        // If the element is not val
        if (nums[i] !== val) {
            nums[count++] = nums[i];
        }
    }
    return count;
}
```

Q3. Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Example 1:
Input: nums = [1,3,5,6], target = 5

Output: 2

Answer:
To find the index where a target number is stored we should be initiating with a binary search algorithm with the sorted array. If the number is already present in the array, the function will directly return its index. Otherwise, it returns the index place where the target should possibly get inserted while maintaining that sorted order.
Initially, the pointer will be placed at the start of the index array and other pointers will be placed at the end position of the index. The target is to find the mid of the array. If the pointer matches the current value, it will return the index. If the mid is less than the target the pointer should be updated to +1 and otherwise update the index value to the target number. If their is no target available on the array then the index should be returned where the target should be inserted while maintaining the sorted order.

The time complexity of this solution would be O(logn) as it uses binary search algorithm that divides the space into half. The space complexity would be O(1) since it uses the constant amount of space.

Code

```
var searchInsert = function(nums, target) {
    let start = 0, end = nums.length - 1;
    let ans = nums.length; // Default answer when the target is greater
than all elements

    while (start <= end) {
        let mid = Math.floor((start + end) / 2);

        if (nums[mid] === target) {
            return mid;
        } else if (nums[mid] < target) {
            start = mid + 1;
        } else {
            ans = mid; // Update the answer to the current index
```

```
            end = mid - 1;
        }
    }

    return ans;
};
```

Q4. You are given a large integer represented as an integer array of digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

Example 1: Input: digits = [1,2,3] Output: [1,2,4]

Explanation: The array represents the integer 123.

Incrementing by one gives 123 + 1 = 124. Thus, the result should be [1,2,4].

Answer: In the problem statement it is mentioned that the number is a large integer representation. That is why we will use BigInt value representation that is capable of holding a large number of primitives. Before moving forwards, it is important to mention that bigInt primitives are appended with a ‘n’ at the end of the value.
For example :

```
const bigNumber = BigInt(2244996633)
//2244996633n
```

Now coming back to our problem, we are going to take the digits initially and add ‘1n’ to the end. Later on, we want to split the array into subarray and for that, we need to break the array into strings using toString() method.

Code

```
var plusOne = function(digits) {
  let num = BigInt(digits.join("")) + 1n
      let str = num.toString()
      digits = str.split('')
      return digits
};
```

Q5. You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

Merge nums1 and nums2 into a single array sorted in non-decreasing order.

The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

Example 1: Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3 Output: [1,2,2,3,5,6]

Explanation: The arrays we are merging are [1,2,3] and [2,5,6]. The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

Answer:
We are given the two arrays with non-decreasing order nums1 and nums2 of the respective sizes of m and n. The task is to merge two arrays into a single sorted array and the result should be stored inside nums1, thus making the size of nums1 to be m+n. The approach would be to iterate the second array from the end and place the larger element at the end on nums1. After that sort the entire nums1 using the sort() function.
The time complexity would be O((m+n)log(m+n)) due to the sorting functions and the space complexity would be O(1) since we are not using any extra space.

```
var merge = function(nums1, m, nums2, n) {
    for (let i = m, j = 0; j < n; i++, j++) {
        nums1[i] = nums2[j];
    }
    nums1.sort((a, b) => a - b);
};
```

Q6. Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1: Input: nums = [1,2,3,1]

Output: true

Answer: For this problem, we are going to use a set in Javascript. A Javascript set is a collection of unique values where one value can occur once in a set. So initially we are going to make a set from the given array and compare their sizes. If the size of the set is less than the array then it obviously has a duplicate number. The time and space complexity would be O(n) for both.

Code:

```
var containsDuplicate = function(nums) {
const s = new Set(nums);
return s.size !== nums.length
};
```

Q7. Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the nonzero elements.

Note that you must do this in place without making a copy of the array.

Example 1: Input: nums = [0,1,0,3,12] Output: [1,3,12,0,0]

Answer: To solve this problem, we need a pointer method in javascript. We create one pointer which will iterate over the array and if zero is not found then the number will come to the index and pointe will be updated by 1.

Though only one loop is iterating, the time complexity is O(n) and the space complexity is (1).

Code:

```
var moveZeroes = function(nums) {
     let pointer = 0
    for (let i in nums){
        if (nums[i] !== 0){
            [nums[pointer], nums[i]] = [nums[i], nums[pointer]]
            pointer++
        }
    }
};
```

Q8. You have a set of integers s, which originally contains all the numbers from 1 to n. Unfortunately, due to some error, one of the numbers in s got duplicated to another number in the set, which results in repetition of one number and loss of another number.

You are given an integer array nums representing the data status of this set after the error.

Find the number that occurs twice and the number that is missing and return them in the form of an array.

Example 1: Input: nums = [1,2,2,4] Output: [2,3]

Answer: To solve this problem, we have to get back to the basics of arithmetic progression where the sum of a series 1 to n can be calculated as n*(n+1)/2.

Suppose we have a series from 1 to n in an arithmetic progression. We can find the sum of the series by the formula. Now, the problem says that the few numbers have been repeated which means subtracting the sum of arithmetic progression(without mistake) with the present array(with mistakes) will get us the deficit in the number. But before that, we have to transfer all the unique numbers to different arrays using the Set method in Javascript and find their sum.

For example:

Let's say a series a = [1,2,3,4,4]

Transferred to set b = [1,2,3,4]

We know the sum of AP of length 5 should be

SumOriginal = 1+2+3+4+5 = 15

Sum of mistaken array SumA = 1+2+3+4+4= 14

But, the sum of b

SumB= 1+2+3+4 = 10

Number that occurred twice = SumA - SumB = 14 - 10 = 4

Number that is missing = SumOriginal - SumB = 15 - 10 = 5

This is how we find both the number.

The time complexity would be O(n) as it iterates over the array and space complexity would be O(n).

Code:

```
var findErrorNums = function(nums) {

    const n = nums.length
    const except = n * (n + 1) / 2  // Find sum of AP

    const set = new Set(nums) // The unique set
    let setSum = 0
    set.forEach(val => setSum+=val) // Sum of unique set
```

```
    const numSum = nums.reduce((curr,acc) => curr + acc) // Sum of mistakes array
    return [numSum - setSum, except - setSum]
};
```