**Note: All the Code is in JavaScript Programming Language**

**Question 1**

Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to the target. Return the sum of the three integers.

You may assume that each input would have exactly one solution.

**Example 1:**

Input: nums = [-1,2,1,-4], target = 1

Output: 2

**Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

**Answer:** In the above solution it is better to use 2 pointer approach after the sorting of the array. Let’s say we have the sorted array in ascending order. One variable will find the sum of the first three elements of the sorted array while each pointer will reside on either side of the loop that is one pointer will be at starting and the other will be at the end of the array. Till right > left pointer we will calculate the sum of all triplets and suppose it is equal to the target, we will return the numbers. Otherwise, if it is more than the target we will decrement right by 1 and if it is less than the target we will increment left by 1. This is how we are getting closer to the sum. We will check the absolute difference between the sum and the target and return the closest sum after the entire iterations

The time complexity for sorting will be O(n log n) while two pointer approach takes O(n^2). Therefore the overall time complexity would be O(n^2 logn). Space complexity would be O(1) as no extra space is taken in the solution.

| var threeSumClosest = function(nums, target) {  nums.sort((a, b) => a - b);  let n = nums.length;  let closest\_sum = nums[0] + nums[1] + nums[2]; // initialize closest sum  for (let i = 0; i < n - 2; i++) {  let left = i + 1, right = n - 1;  while (left < right) { // two-pointer approach  let sum = nums[i] + nums[left] + nums[right];  if (sum == target) { // sum equals target, return immediately  return sum;  } else if (sum < target) {  left++;  } else {  right--;  }  if (Math.abs(sum - target) < Math.abs(closest\_sum - target)) { // update closest sum  closest\_sum = sum;  }  }  }  return closest\_sum; }; |
| --- |

**Question 2**

Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that: ● 0 <= a, b, c, d < n ● a, b, c, and d are distinct. ● nums[a] + nums[b] + nums[c] + nums[d] == target

You may return the answer in any order.

Example 1:

Input: nums = [1,0,-1,0,-2,2], target = 0

Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

**Answer:** This is basically a 4-sum problem where we can use the same method as above. Like the 3 sum solution, we can follow the same approach where we first sort the arrays and then we use the 2 pointer approach to find the quadruplet whole target value is equal to the sum. The time complexity would be O(n^3) where n is the length of the input array. The space complexity would be O(1) because we are not using any extra space.

| var fourSum = function(nums, target) {  nums.sort((a, b) => a - b);  const quadruplets = [];  const n = nums.length;  for (let i = 0; i < n - 3; i++) {  if (i > 0 && nums[i] === nums[i - 1]) {  continue;  }  for (let j = i + 1; j < n - 2; j++) {  if (j > i + 1 && nums[j] === nums[j - 1]) {  continue;  }  let left = j + 1;  let right = n - 1;  while (left < right) {  const sum = BigInt(nums[i]) + BigInt(nums[j]) + BigInt(nums[left]) + BigInt(nums[right]);  if (sum < target) {  left++;  } else if (sum > target) {  right--;  } else {  quadruplets.push([nums[i], nums[j], nums[left], nums[right]]);  while (left < right && nums[left] === nums[left + 1]) {  left++;  }  while (left < right && nums[right] === nums[right - 1]) {  right--;  }  left++;  right--;  }  }  }  }  return quadruplets; }; |
| --- |

**Question 3** A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container.

If such an arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

● For example, the next permutation of arr = [1,2,3] is [1,3,2]. ● Similarly, the next permutation of arr = [2,3,1] is [3,1,2]. ● While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums. The replacement must be in place and use only constant extra memory.

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

**Answer:** The approach would to find the decreasing index from end to the start of nums. Then swap num 1 with the next large nums to its right that is 2. Then reverse or sort nums to the right. But if no next permutation is found reverse the array.

| var nextPermutation = function(nums) {    for(let i = nums.length-1; i >= 0; i--) {  if(nums[i] < nums[i+1]) {  const large = nextLarge(i);  swap(i, large);  reverse(i+1);  return;  }  }    // If there is no next permutation reverse the arr  nums.reverse()    function swap(i, j) {  [nums[i], nums[j]] = [nums[j], nums[i]];  }    function reverse(idx) {  let start = idx, end = nums.length-1;    while(start < end) {  swap(start, end);  start++;  end--;  }  }    function nextLarge(idx) {  for(let i = nums.length-1; i > idx; i--) {  if(nums[i] > nums[idx]) return i;  }  } }; |
| --- |

The time complexity would be O(n) as it iterated over the length of array that is ‘n’. The space complexity would be O(1) as it doesn’t took any extra space.

**Question 4**

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:** The approach to this solution lies behind the binary search where we assign high to the end of the array and low to the start of the array. We will try to find the mid of the array and compare it with the target. If mid is greater than the target, then the mid will become high otherwise mid+1 will become the next low. The iteration continues till the condition of target > mid is true. Otherwise, it will return to the low where we can place the new integer.

**Code:**

| var searchInsert = function(nums, target) {  let lo = 0, hi = nums.length; // we might need to inseart at the end  while(lo < hi) { // breaks if lo == hi  let mid = lo + Math.floor((hi-lo)/2); // always gives the lower mid  if (target > nums[mid]) {  lo = mid + 1 // no way mid is a valid option  } else {  hi = mid // it might be possibe to inseart @ mid  }  }  return lo; }; |
| --- |

The time complexity would be O(log n) for binary search and the space complexity would be O(1) as we haven’t use any extra space.

**Question 5** You are given a large integer represented as an integer array 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 };** |
| --- |

**Question 6**

Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

**Example 1:**

Input: nums = [2,2,1] Output: 1

**Solution:** Here, in this solution, we are going to use the concept of XOR which has the following properties.

* XOR operation with O gives the same number (2 XOR 0 = 2)
* XOR operation with the same number gives 0 (a XOR a = 0)
* XOR is associative ((2 XOR 3) XOR 4 = 2 XOR (3 XOR 4))

With the help of these properties, we will perform the iteration over the nums and return the unique number that has appeared only once. The space complexity would be O(1) as no extra space has been takes. The time complexity would be O(n) as iteration over the n length has been run.

| var singleNumber = function(nums) {  // Initialize the unique number...  let uniqNum = 0;  // TRaverse all elements through the loop...  for (let id = 0; id < nums.length; id++) {  // Concept of XOR...  uniqNum = uniqNum ^ nums[id];  } return uniqNum; // Return the unique number... }; |
| --- |

**Question 7**

You are given an inclusive range [lower, upper] and a sorted unique integer array nums, where all elements are within the inclusive range.

A number x is considered missing if x is in the range [lower, upper] and x is not in nums.

Return the shortest sorted list of ranges that exactly covers all the missing numbers. That is, no element of nums is included in any of the ranges, and each missing number is covered by one of the ranges.

Example 1: Input: nums = [0,1,3,50,75], lower = 0, upper = 99

Output: [[2,2],[4,49],[51,74],[76,99]]

Explanation: The ranges are: [2,2] [4,49] [51,74] [76,99]

**Answer:** The approach to this question is such that we are going to initialize a variable that iterates to find the next number in the array. If the number from the array is less than the next, we will jump to the next num to chech if it’s in the range. The nums is equal to next, it means we found the first num in range and increment the next target by one. But if nums is greater than next, then the missing range would be next till the nums - 1. Then update next value to nums + 1. At last we need to check if still next is less than the upper range, that means we still have a missing range (next to upper) to add. This is how we solve the problem.

| class targetRange{  findMissingRanges(nums, lower, upper) {  let res = [];  let next = lower;    for (let i = 0; i < nums.length; i++) {  if (lower === Number.MAX\_VALUE) return res;    if (nums[i] < next) {  continue;  }    if (nums[i] === next) {  next++;  continue;  }    res.push(this.getRange(next, nums[i] - 1));    if (nums[i] === Number.MAX\_VALUE) return res;    next = nums[i] + 1;  }    if (next <= upper) {  res.push(this.getRange(next, upper));  }    return res;  }    getRange(n1, n2) {  return n1 === n2 ? String(n1) : `${n1}->${n2}`;  } } |
| --- |

The time complexity of the solution would be O(n) where n is the length of nums array and space complexity would be O(1) as it uses a constant space.

**Question 8**

Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

**Example 1:** Input: intervals = [[0,30],[5,10],[15,20]]

Output: false

**Answer:** In this question, we have to find the intersection of intervals and check it there is an overlap. If the overlapping is there, return false as the person cannot attent interview. If we breakdown the solution, we find there are basically two cases here.Let say there are two intervals a and b, if the start position of a is greater than or equal to the start position of b, and start positon of a is less than the end position of b, then there must be an overlap. The second situation is that a and b exchange positions. If the start position of b is greater than or equal to the start position of a, and the start position of b is less than the end position of a, then there must an overlap.

| function canAttendMeetings(intervals) {  intervals.sort((a, b) => a[0] - b[0]);  for (let i = 1; i < intervals.length; ++i) {  if (intervals[i][0] < intervals[i - 1][1]) {  return false;  }  }  return true; } |
| --- |

The time complexity would be O(n log n) as it is using the sort function and iterating over the start to end of the index. The space complexity would be O(1) indicating constant space usage.