**Assignment Questions 3**

**Q 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).**

# Ans:- Explanation

The problem requires us to find a triplet of numbers in the given array, such that their sum is closest to the given target. We can use the two-pointer approach along with sorting the array to solve this problem.

**Approach:**

1. Sort the given array in non-descending order.
2. Initialize a variable closest\_sum to store the closest sum found so far. Set it initially to the sum of first three elements in the sorted array.
3. Loop over the array from i=0 to i=n-3, where n is the size of the array.
4. For each i, initialize two pointers, left and right, to i+1 and n-1 respectively.
5. While left < right, calculate the sum of the current triplet, sum = nums[i] + nums[left] + nums[right].
6. If sum is equal to the target, we have found the closest sum possible, so we can return it immediately.
7. If sum is less than target, increment left by 1. This will increase the sum, and we may get a closer sum.
8. If sum is greater than target, decrement right by 1. This will decrease the sum, and we may get a closer sum.
9. After each iteration of the inner loop, check if the absolute difference between sum and target is less than the absolute difference between closest\_sum and target. If it is, update closest\_sum to sum.
10. Return closest\_sum after the loop ends.

**Complexity:**

* Time Complexity: Sorting the array takes O(nlogn) time. The two-pointer approach runs in O(n^2) time. Therefore, the overall time complexity of the solution is O(n^2logn).
* Space Complexity: We are not using any extra space in the solution. Therefore, the space complexity of the solution is O(1).

**JAVA**

class Solution {

public int threeSumClosest(int[] nums, int target) {

Arrays.sort(nums);

int n = nums.length;

int closest\_sum = nums[0] + nums[1] + nums[2]; // initialize closest sum

for (int i = 0; i < n - 2; i++) {

int left = i + 1, right = n - 1;

while (left < right) { // two-pointer approach

int 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;

}

}

**Python**

class Solution(object):

def threeSumClosest(self, nums, target):

nums.sort()

n = len(nums)

closest\_sum = nums[0] + nums[1] + nums[2] # initialize closest sum

for i in range(n - 2):

left, right = i + 1, n - 1

while left < right: # two-pointer approach

sum = nums[i] + nums[left] + nums[right]

if sum == target: # sum equals target, return immediately

return sum

elif sum < target:

left += 1

else:

right -= 1

if abs(sum - target) < abs(closest\_sum - target): # update closest sum

closest\_sum = sum

return closest\_sum

**Q2 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]]**

**Ans:- Explanation**

**fix two-pointers and then find the remaining two elements using two pointer technique as the array will be sorted at first.**

**Approach**

**Sort the array, and then fix two pointers, so the remaining sum will be (target – (nums[i] + nums[j])). Now we can fix two-pointers, one front, and another back, and easily use a two-pointer to find the remaining two numbers of the quad. In order to avoid duplications, we jump the duplicates, because taking duplicates will result in repeating quads. We can easily jump duplicates, by skipping the same elements by running a loop.**

**Complexity**

**Time complexity:**

**O(n^3)-->**

**2 nested for loops and the front pointer as well as the right pointer (Third nested loop)**

**Space complexity:**

**O(1)-->**

**Generally the space complexity that is used to return the answer is ignored**

**Code**

**class Solution {**

**public List<List<Integer>> fourSum(int[] nums, int target) {**

**int n=nums.length;**

**Arrays.sort(nums);**

**List<List<Integer>> ans=new ArrayList<>();**

**if(n==0||n<3){**

**return ans;**

**}**

**if(target==-294967296 || target==294967296){**

**return ans;**

**}**

**for(int i=0;i<n;i++){**

**for(int j=i+1;j<n;j++){**

**int low=j+1;**

**int high=n-1;**

**int sum=target-nums[i]-nums[j];**

**while(low<high){**

**if(nums[low]+nums[high]==sum){**

**List<Integer> temp=new ArrayList<>();**

**temp.add(nums[i]);**

**temp.add(nums[j]);**

**temp.add(nums[low]);**

**temp.add(nums[high]);**

**ans.add(temp);**

**while(low<high&&nums[low]==nums[low+1]){**

**low++;**

**}**

**while(low<high&&nums[high]==nums[high-1]){**

**high--;**

**}**

**low++;**

**high--;**

**}**

**else if(nums[low]+nums[high]<sum){**

**low++;**

**}**

**else{**

**high--;**

**}**

**}**

**while(j+1<n&&nums[j+1]==nums[j]){**

**j++;**

**}**

**}**

**while(i+1<n&&nums[i+1]==nums[i]){**

**i++;**

**}**

**}**

**return ans;**

**}**

**}**

**Q3 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]**

# Ans:- Explanation:

The problem requires finding the next permutation in lexicographic order of the given array. If the given permutation is the largest possible permutation, then the function should return the smallest possible permutation by rearranging the array in ascending order.

**Approach:**

The algorithm finds the first pair of adjacent elements in the array that satisfy nums[k] < nums[k+1] from the right end of the array. If such a pair does not exist, then the entire array is sorted in descending order, and we need to reverse the entire array to obtain the smallest possible permutation.

Otherwise, the algorithm finds the smallest element nums[l] to the right of nums[k] such that nums[l] > nums[k]. We swap nums[k] and nums[l], and then reverse the subarray starting at nums[k+1] to obtain the next lexicographic permutation of the array.

**Complexity:**

* Time complexity: The algorithm performs two passes over the array. The first pass has time complexity O(n), while the second pass also has time complexity O(n). Therefore, the overall time complexity of the algorithm is O(n).
* Space complexity: The algorithm uses constant extra space, hence the space complexity of the algorithm is O(1).

**JAVA**

class Solution {

public void nextPermutation(int[] nums) {

int n = nums.length, k = n - 2, l = n - 1;

while (k >= 0 && nums[k] >= nums[k + 1]) {

k--;

}

if (k < 0) {

reverse(nums, 0, n - 1);

} else {

while (l > k && nums[l] <= nums[k]) {

l--;

}

swap(nums, k, l);

reverse(nums, k + 1, n - 1);

}

}

private void reverse(int[] nums, int i, int j) {

while (i < j) {

swap(nums, i++, j--);

}

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

**Python**

class Solution(object):

def nextPermutation(self, nums):

n = len(nums)

k, l = n - 2, n - 1

while k >= 0 and nums[k] >= nums[k + 1]:

k -= 1

if k < 0:

nums.reverse()

else:

while l > k and nums[l] <= nums[k]:

l -= 1

nums[k], nums[l] = nums[l], nums[k]

nums[k + 1:n] = reversed(nums[k + 1:n])

**Q 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**

# Ans:- Explanation:

The given code implements a binary search algorithm to find the index where a target number should be inserted into a sorted array. If the target number is already present in the array, the function returns its index. If not, it returns the index where the target should be inserted to maintain the sorted order.

**Approach:**

1. Initialize the start pointer to 0, representing the first index of the array, and the end pointer to nums.size() - 1, representing the last index of the array.
2. Set the answer variable ans to nums.size(), which is the default answer when the target is greater than all elements in the array.
3. Enter a while loop while the start pointer is less than or equal to the end pointer.
4. Calculate the mid index using the formula: mid = start + (end - start) / 2. This formula ensures that the mid index is always rounded down to the nearest integer.
5. Compare the element at the mid index, nums[mid], with the target:  
   a. If they are equal, return the mid index because the target is found in the array.  
   b. If nums[mid] is less than the target, update the start pointer to mid + 1 because the target should be on the right side of the mid index.  
   c. If nums[mid] is greater than the target, update the answer ans to the current mid index and update the end pointer to mid - 1 because the target should be on the left side of the mid index.
6. If the while loop exits without finding the target, return the answer ans, which represents the index where the target should be inserted to maintain the sorted order.

**Complexity:**

* The time complexity of this solution is O(log n) because the binary search algorithm divides the search space in half at each step.
* The space complexity is O(1) since the algorithm uses only a constant amount of extra space.

**JAVA**

class Solution {

public int searchInsert(int[] nums, int target) {

int start = 0, end = nums.length - 1;

int ans = nums.length; // Default answer when target is greater than all elements

while (start <= end) {

int mid = start + (end - start) / 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;

}

}

**Python**

class Solution(object):

def searchInsert(self, nums, target):

start, end = 0, len(nums) - 1

ans = len(nums) # Default answer when target is greater than all elements

while start <= end:

mid = (start + end) / 2

if nums[mid] == target:

return mid

elif nums[mid] < target:

start = mid + 1

else:

ans = mid # Update the answer to the current index

end = mid - 1

return ans

**Q 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].**

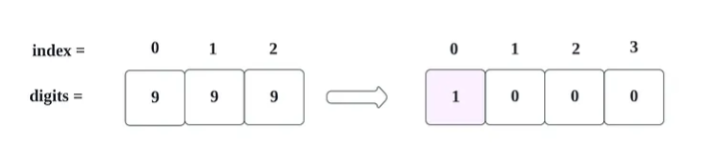
# Ans:- Solution: →

This is a little tricky problem. The interface requires to return **int[]**, but you are not sure what’s the length for the returning array. Given input array,**int[]** digits, it could be **digits.length** or **digits.length + 1**.

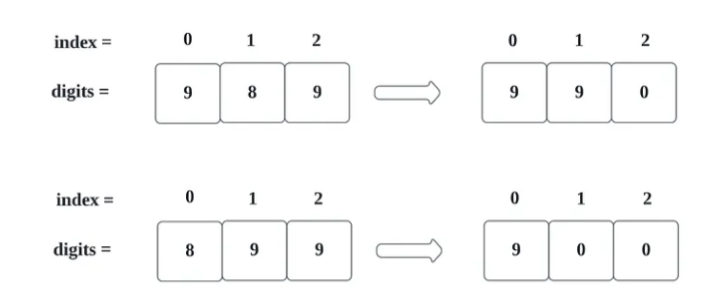
To get this solution, lets go step by step.

1. First, lets find out in what condition we will require **digits.length + 1.**

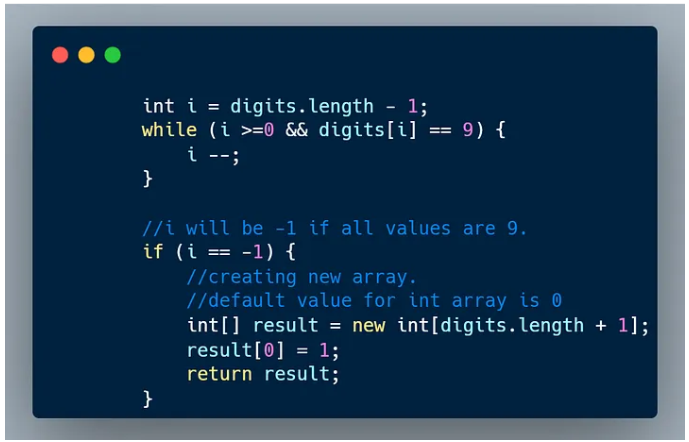
Here, by checking different values, we can see that if**all digits are 9**, in that case only we require **digits.length + 1**

****

If any digit value, other then 9 then we don’t need to take an array with size **digits.length + 1.**

****

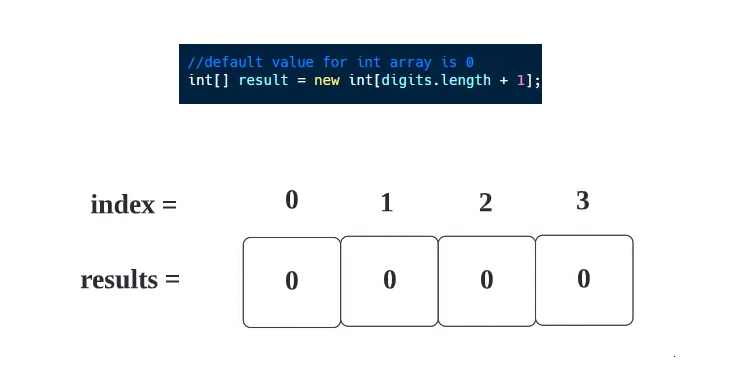
For above condition, we can use following logic,

****

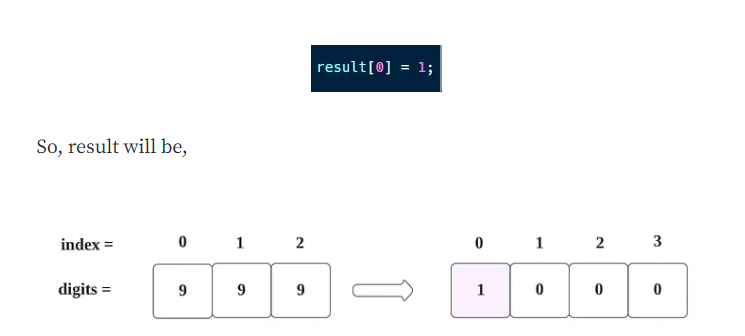
Here, we are moving from Right To Left, as if we find last element’s value **9**, then it will be **0** as 9+1 = 10.

So, with above logic, below result will be achieved **if all values are 9**.

1. New array will be created with total **digits.length + 1** size.

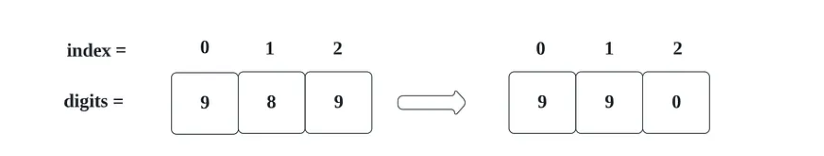
****

1. We will assign 1, to our first index.

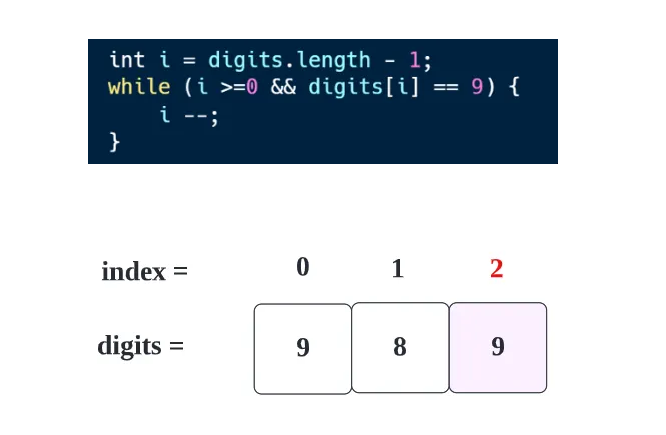
****

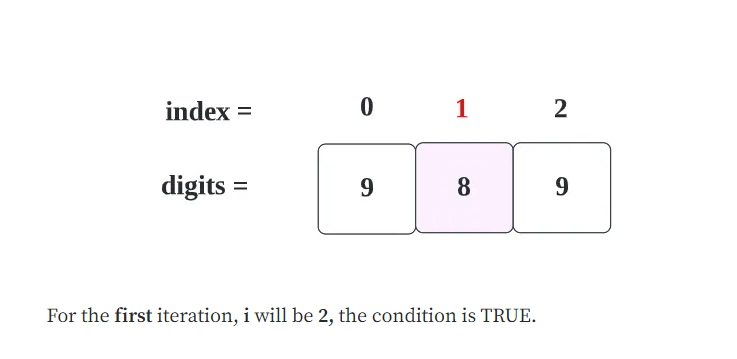
Now, moving forward to other conditions where array size is not getting incremented.

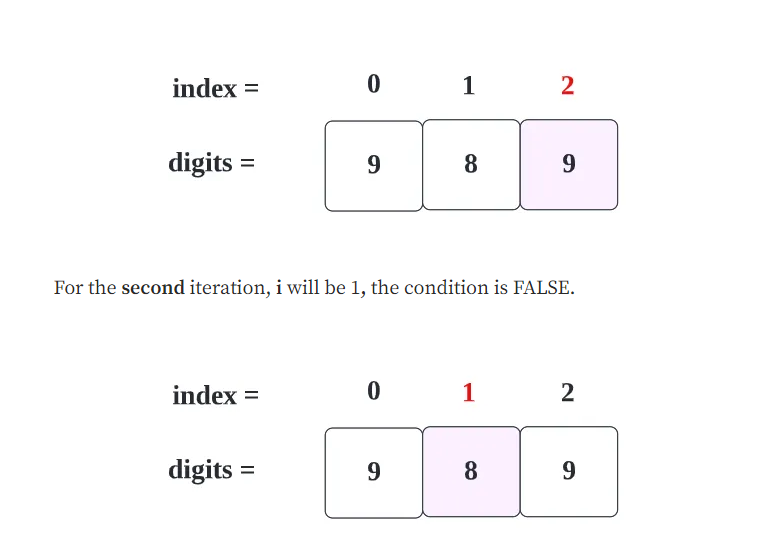
For ex,

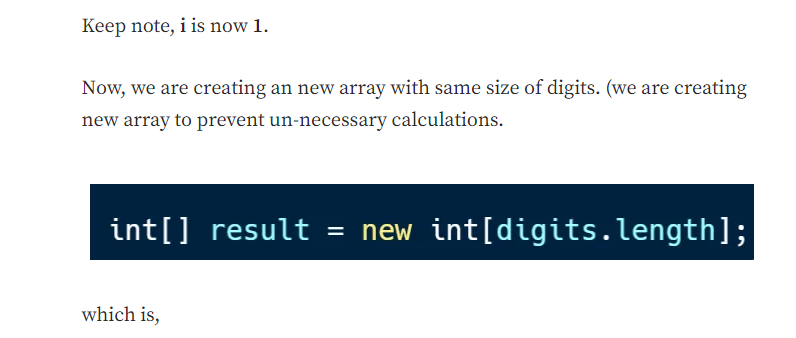
****

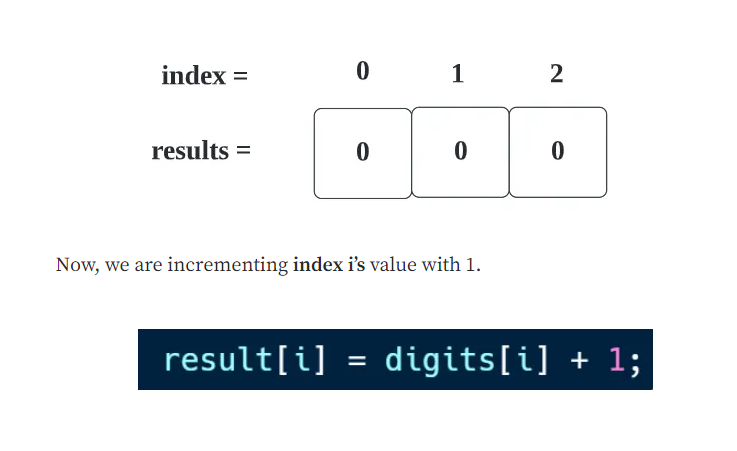
In this case also, we need to pass trough first condition where we are checking **9**.

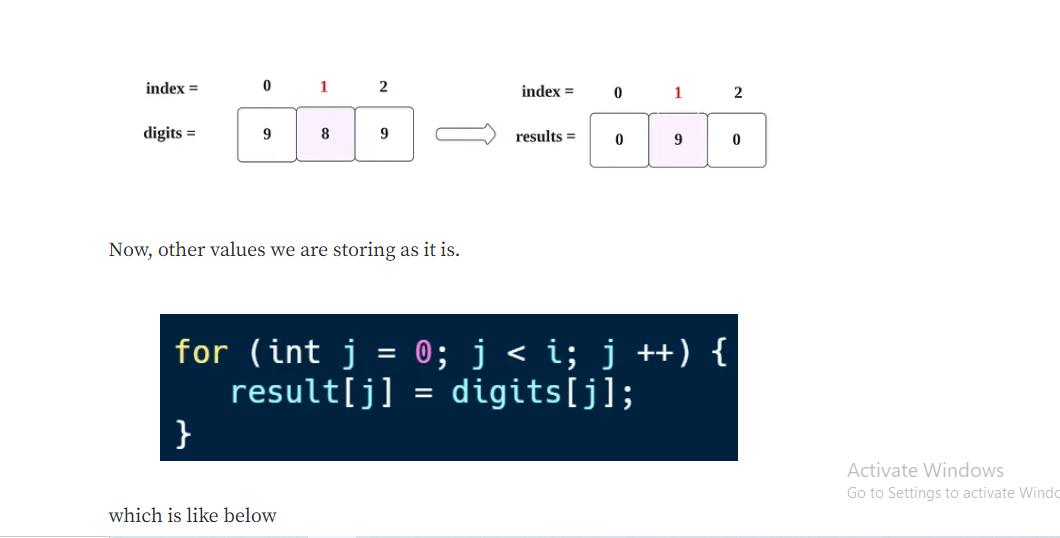
****

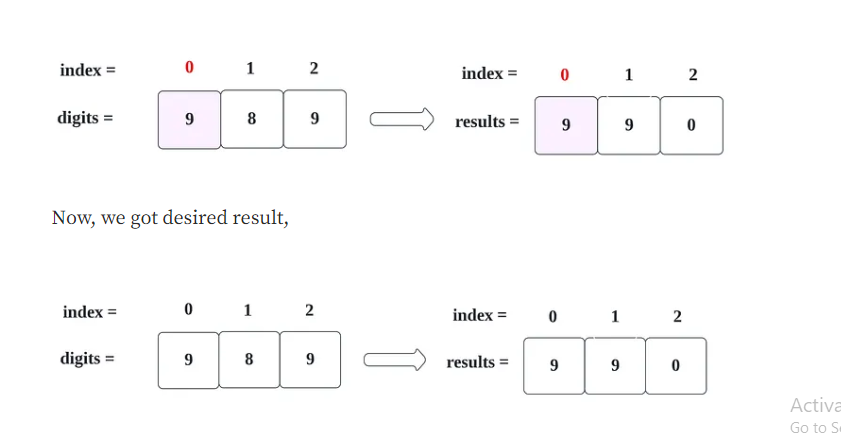
****

****

****

****

****

****

Lets see whole code

**Code (Java): →**

|  |
| --- |
| class Solution { |
|  | public int[] plusOne(int[] digits) { |
|  |  |
|  | int i = digits.length - 1; |
|  |  |
|  | while (i >=0 && digits[i] == 9) { |
|  | i --; |
|  | } |
|  |  |
|  |  |
|  | if (i == -1) { |
|  | int[] result = new int[digits.length + 1]; |
|  | result[0] = 1; |
|  | return result; |
|  | } |
|  |  |
|  |  |
|  | int[] result = new int[digits.length]; |
|  |  |
|  | result[i] = digits[i] + 1; |
|  | for (int j = 0; j < i; j ++) { |
|  | result[j] = digits[j]; |
|  | } |
|  |  |
|  | return result; |
|  |  |
|  | } |
|  | } |

**Code (Python): →**

|  |
| --- |
| class Solution(object): |
|  | def plusOne(self, digits): |
|  | """ |
|  | :type digits: List[int] |
|  | :rtype: List[int] |
|  | """ |
|  | digit\_length = len(digits) |
|  |  |
|  | i = digit\_length - 1 |
|  |  |
|  | while digits[i] == 9 and i >= 0: |
|  | i -= 1 |
|  |  |
|  | if i == -1: |
|  | results = [0]\*(digit\_length + 1) |
|  | results[0] = 1 |
|  | return results |
|  |  |
|  | results = [0]\*(digit\_length) |
|  |  |
|  | results[i] = digits[i] + 1 |
|  |  |
|  | for j in range(i-1, -1, -1): |
|  | results[j] = digits[j] |
|  |  |
|  | return results |

**Time Complexity**

We are traversing an whole array (from right to left) until we find space, so time complexity will be **O(n)**.

**Space Complexity**

Since we have used an extra array with +1 more space, the space complexity will be ***O(n+1)***.

**Q 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**

# Ans:- Approach

The code is a Java implementation of a function that takes an integer array nums as input and returns the single integer in the array that appears only once. The function assumes that all other integers in the array appear exactly twice.

The implementation sorts the input array using the Arrays.sort() method. Sorting the array ensures that all duplicate integers are adjacent to each other. Then, the implementation iterates through the sorted array using a loop that increments the index by 2 in each iteration. Inside the loop, the implementation checks if the current integer and the next integer are not equal. If they are not equal, the implementation returns the current integer as the single integer that appears only once.

If the loop completes without finding a single integer that appears only once, the implementation returns the last element of the sorted array. This is because the last element is the only remaining element in the array, and it must be the single integer that appears only once.

**Complexity**

* Time complexity:  
  The Arrays.sort method used at the beginning of the function has a time complexity of O(n log n), where n is the length of the input array. This is because it uses a variant of the quicksort algorithm, which has an average time complexity of O(nlogn).
* Space complexity:  
  The space complexity of the function is O(1), as it uses only a constant amount of additional memory to store the loop variable and some temporary variables for the sorting operation. The sorting operation is performed in-place, so it does not require any additional memory allocation.

**Code**

class Solution {

public int singleNumber(int[] nums) {

Arrays.sort(nums);

if(nums.length==1){

return nums[0];

}

for(int i=0;i<nums.length-1;i+=2){

if(nums[i]!=nums[i+1]){

return nums[i];

}

}

return nums[nums.length-1];

}

}

**Q 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]**

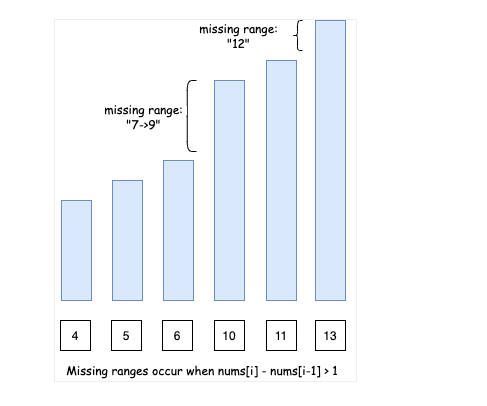
**[76,99]**

**Ans:-** Approach 1: Linear Scan

Intuition and Algorithm

Since the input array, nums, is sorted ascendingly and all the elements in it are within the given [lower, upper] bounds, we can simply check consecutive elements to see if they differ by one or not. If they don't, then we have found a missing range.

* When nums[i] - nums[i-1] == 1, we know that there are no missing elements between nums[i-1] and nums[i].
* When nums[i] - nums[i-1] > 1, we know that [nums[i-1] + 1, nums[i] - 1] range of elements are missing.

****

However, there are two edge cases:

* Case 1: If we don't start with lower as the first element of the array, we will need to consider [lower, num[0] - 1] missing range as well.

****

* Case 2: Similarly, if we don't end with upper as the last element of the array, we will need to consider [nums[n-1] + 1, upper] missing range as well. Note n here is the length of the input array nums.

**class** **Solution** {

**public** List<String> **findMissingRanges**(**int**[] nums, **int** lower, **int** upper) {

**int** n = nums.length;

**if** (n == 0) {

**return** Collections.singletonList(formatRange(lower, upper));

}

List<String> missingRanges = **new** ArrayList<>();

// Edge case 1) Missing ranges at the beginning

**if** (nums[0] > lower) {

missingRanges.add(formatRange(lower, nums[0] - 1));

}

// Missing ranges between array elements

**for** (**int** i = 1; i < n; ++i) {

**if** (nums[i] - nums[i - 1] > 1) {

missingRanges.add(formatRange(nums[i - 1] + 1, nums[i] - 1));

}

}

// Edge case 2) Missing ranges at the end

**if** (nums[n - 1] < upper) {

missingRanges.add(formatRange(nums[n - 1] + 1, upper));

}

**return** missingRanges;

}

// formats range in the requested format

String **formatRange**(**int** lower, **int** upper) {

**if** (lower == upper) {

**return** String.valueOf(lower);

} **else** {

**return** lower + "->" + upper;

}

}

}

Complexity Analysis

* Time complexity : O(N)O(N), where NN is the length of the input array. This is because we are only iterating over the array once.
* Space complexity : O(N)O(N) if we take the output into account and O(1)O(1) otherwise, where NN is the length of the input array. This is because we could have a missing range between each of the consecutive element of the input array. Hence, our output list that we need to return will be of size NN.

**Q8 Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.**

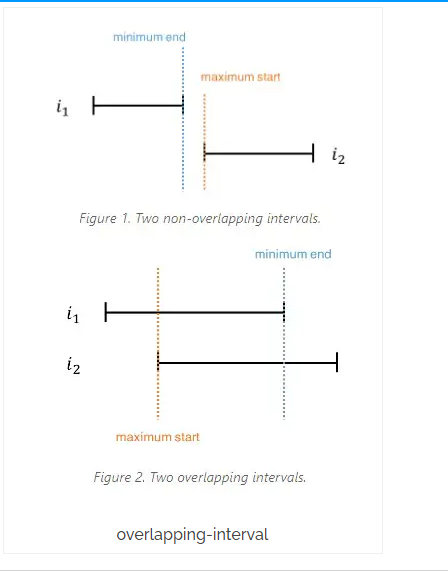
**Example 1:**

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

**Output: false**

### Ans:- BRUTEFORCE ALGORITHM TO CHECK IF ANY INTERVALS OVERLAPPING

If any two [meetings](https://helloacm.com/algorithms-to-check-if-any-two-intervals-overlap/) (intervals) overlap, then we can’t attend all the meetings. Thus we can bruteforce all pairs of meetings in O(N^2) to see if they overlap. To find out if two meetings overlap, we check their ends:

****

The overall time complexity is O(N^2).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **class** Solution:  **def** canAttendMeetings(self, intervals: List[List[int]]) -> bool:          n = len(intervals)    **def** overlap(a, b):  **return** min(a[1], b[1]) > max(a[0], b[0])    **for** i **in** range(n):  **for** j **in** range(i):  **if** overlap(intervals[i], intervals[j]):  **return** False  **return** True |

**Time complexity=O(N^2)**

**Space complexity=O(N)**

### SORTING ALGORITHM TO CHECK IF ANY INTERVALS OVERLAPPING

If we sort the intervals by their start, we can check if the current meeting is [overlapping](https://helloacm.com/the-overlapping-rectangles-using-css-and-javascript/) with the next one.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **class** Solution:  **def** canAttendMeetings(self, intervals: List[List[int]]) -> bool:          n = len(intervals)          intervals.sort(key=**lambda** x:x[0]) *# or just intervals.sort()*  **for** i **in** range(len(intervals) - 1):  **if** intervals[i][1] > intervals[i + 1][0]:  **return** False  **return** True |

Sorting takes O(NLogN) time complexity, we can rewrite above using one-liner via all or any method:

|  |  |
| --- | --- |
| 1  2 | **return** all(intervals[i][1] <= intervals[i + 1][0] **for** i **in** range(n - 1))  **return** **not** any(intervals[i][1] > intervals[i + 1][0] **for** i **in** range(n - 1)) |

**Time complexity=O(NLogN)**

**Space complexity=O(1)**