Q1.import java.util.Arrays;

public class ThreeSumClosest {

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

Arrays.sort(nums);

int n = nums.length;

int closestSum = Integer.MAX\_VALUE;

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

int left = i + 1;

int right = n - 1;

while (left < right) {

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

if (sum == target) {

return target;

}

if (Math.abs(sum - target) < Math.abs(closestSum - target)) {

closestSum = sum;

}

if (sum < target) {

left++;

} else {

right--;

}

}

}

return closestSum;

}

public static void main(String[] args) {

ThreeSumClosest solution = new ThreeSumClosest();

int[] nums = {-1, 2, 1, -4};

int target = 1;

int closestSum = solution.threeSumClosest(nums, target);

System.out.println("Sum closest to target: " + closestSum);

}

}

Q2.import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class FourSum {

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

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(nums);

int n = nums.length;

for (int i = 0; i < n - 3; i++) {

if (i > 0 && nums[i] == nums[i - 1]) {

continue;

}

for (int j = i + 1; j < n - 2; j++) {

if (j > i + 1 && nums[j] == nums[j - 1]) {

continue;

}

int left = j + 1;

int right = n - 1;

while (left < right) {

int sum = nums[i] + nums[j] + nums[left] + nums[right];

if (sum == target) {

result.add(Arrays.asList(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--;

} else if (sum < target) {

left++;

} else {

right--;

}

}

}

}

return result;

}

public static void main(String[] args) {

FourSum solution = new FourSum();

int[] nums = {1, 0, -1, 0, -2, 2};

int target = 0;

List<List<Integer>> result = solution.fourSum(nums, target);

System.out.println("Unique quadruplets:");

for (List<Integer> quadruplet : result) {

Q3.public class NextPermutation {

public void nextPermutation(int[] nums) {

int n = nums.length;

int i = n - 2;

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

i--;

}

if (i >= 0) {

int j = n - 1;

while (j > i && nums[j] <= nums[i]) {

j--;

}

swap(nums, i, j);

}

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

}

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

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

private void reverse(int[] nums, int start, int end) {

while (start < end) {

swap(nums, start, end);

start++;

end--;

}

}

public static void main(String[] args) {

NextPermutation solution = new NextPermutation();

int[] nums = {1, 2, 3};

System.out.println("Original permutation: " + Arrays.toString(nums));

solution.nextPermutation(nums);

System.out.println("Next permutation: " + Arrays.toString(nums));

}

}

Q4.def searchInsert(nums, target):

left = 0

right = len(nums) - 1

while left <= right:

mid = (left + right) // 2

if nums[mid] == target:

return mid

elif nums[mid] < target:

left = mid + 1

else:

right = mid - 1

return left

Q5.def plusOne(digits):

n = len(digits)

carry = 1 # Start with a carry of 1

# Traverse the array from right to left

for i in range(n - 1, -1, -1):

digits[i] += carry # Add the carry to the current digit

carry = digits[i] // 10 # Update the carry

digits[i] %= 10 # Update the current digit

# If there is no carry, we can stop and return the updated array

if carry == 0:

return digits

# If there is still a carry after traversing the entire array,

# we need to add an additional digit at the beginning

if carry > 0:

digits.insert(0, carry)

return digits

Q6def singleNumber(nums):

result = 0

for num in nums:

result ^= num

return result

Q7.def findMissingRanges(nums, lower, upper):

result = []

prev = lower - 1

for i in range(len(nums) + 1):

curr = nums[i] if i < len(nums) else upper + 1

if prev + 1 <= curr - 1:

result.append(formatRange(prev + 1, curr - 1))

prev = curr

return result

def formatRange(start, end):

if start == end:

return str(start)

else:

return str(start) + "->" + str(end)

Q8.def canAttendMeetings(intervals):

intervals.sort(key=lambda x: x[0]) # Sort the intervals by the start time

for i in range(1, len(intervals)):

if intervals[i][0] < intervals[i-1][1]:

return False

return True