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

class InterpolationSearch {

public static int interpolationSearch(int[] arr, int target) {

int low = 0, high = arr.length - 1;

while (low <= high && target >= arr[low] && target <= arr[high]) {

// Prevent division by zero and calculate pos

if (low == high) {

if (arr[low] == target) return low;

return -1;

}

// Estimate the position

int pos = low + ((target - arr[low]) \* (high - low)) / (arr[high] - arr[low]);

// Check if the target is found

if (arr[pos] == target) {

return pos;

}

// If the target is larger, move the low pointer

if (arr[pos] < target) {

low = pos + 1;

}

// If the target is smaller, move the high pointer

else {

high = pos - 1;

}

}

return -1; // Target not found

}

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90};

int target = 70;

int index = interpolationSearch(arr, target);

if (index != -1) {

System.out.println("Element found at index: " + index);

} else {

System.out.println("Element not found.");

}

}

}

You are given a **0-indexed** array of integers nums of length n. You are initially positioned at nums[0].

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j] where:

* 0 <= j <= nums[i] and
* i + j < n

Return the minimum number of jumps to reach nums[n - 1]. The test cases are generated such that you can reach nums[n - 1].

**Example 1:**

**Input:** nums = [2,3,1,1,4]

**Output:** 2

**Explanation:** The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

**Input:** nums = [2,3,0,1,4]

**Output:** 2

import java.util.\*;

class Solution {

public int jump(int[] nums) {

int n = nums.length;

int[] dp = new int[n];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

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

for (int j = 1; j <= nums[i] && i + j < n; j++) {

dp[i + j] = Math.min(dp[i + j], dp[i] + 1);

}

}

return dp[n - 1];

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Test Case 1

int[] nums1 = {2, 3, 1, 1, 4};

int result1 = solution.jump(nums1);

System.out.println("Minimum jumps needed: " + result1);

// Test Case 2

int[] nums2 = {2, 3, 0, 1, 4};

int result2 = solution.jump(nums2);

System.out.println("Minimum jumps needed: " + result2);

// Test Case 3

int[] nums3 = {1, 2, 3};

int result3 = solution.jump(nums3);

System.out.println("Minimum jumps needed: " + result3);

// Test Case 4 (Edge Case)

int[] nums4 = {0};

int result4 = solution.jump(nums4);

System.out.println("Minimum jumps needed: " + result4);

}

}

Given an array of strings strs, group the

anagrams

 together. You can return the answer in **any order**.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**Explanation:**

* There is no string in strs that can be rearranged to form "bat".
* The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.
* The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

import java.util.\*;

class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

Map<String, List<String>> map = new HashMap<>();

for (String word : strs) {

char[] chars = word.toCharArray();

Arrays.sort(chars);

String sortedWord = new String(chars);

if (!map.containsKey(sortedWord)) {

map.put(sortedWord, new ArrayList<>());

}

map.get(sortedWord).add(word);

}

return new ArrayList<>(map.values());

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Test Case 1

String[] strs1 = {"eat", "tea", "tan", "ate", "nat", "bat"};

List<List<String>> result1 = solution.groupAnagrams(strs1);

System.out.println("Grouped Anagrams: " + result1);

// Test Case 2

String[] strs2 = {""};

List<List<String>> result2 = solution.groupAnagrams(strs2);

System.out.println("Grouped Anagrams: " + result2);

// Test Case 3

String[] strs3 = {"a"};

List<List<String>> result3 = solution.groupAnagrams(strs3);

System.out.println("Grouped Anagrams: " + result3);

// Test Case 4

String[] strs4 = {"abc", "bca", "cab", "xyz", "yxz"};

List<List<String>> result4 = solution.groupAnagrams(strs4);

System.out.println("Grouped Anagrams: " + result4);

}

}

You have intercepted a secret message encoded as a string of numbers. The message is **decoded** via the following mapping:

"1" -> 'A'  
"2" -> 'B'  
...  
"25" -> 'Y'  
"26" -> 'Z'

However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes ("2" and "5" vs "25").

For example, "11106" can be decoded into:

* "AAJF" with the grouping (1, 1, 10, 6)
* "KJF" with the grouping (11, 10, 6)
* The grouping (1, 11, 06) is invalid because "06" is not a valid code (only "6" is valid).

Note: there may be strings that are impossible to decode.  
  
Given a string s containing only digits, return the **number of ways** to **decode** it. If the entire string cannot be decoded in any valid way, return 0.

The test cases are generated so that the answer fits in a **32-bit** integer.

**Example 1:**

**Input:** s = "12"

**Output:** 2

**Explanation:**

"12" could be decoded as "AB" (1 2) or "L" (12).

class Solution {

public int numDecodings(String s) {

if (s == null || s.length() == 0 || s.charAt(0) == '0') {

return 0;

}

int n = s.length();

int[] dp = new int[n + 1];

dp[0] = 1; // Base case for an empty string

dp[1] = 1; // Base case for the first character (if not 0)

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

// Check if the current character is valid (non-zero)

if (s.charAt(i - 1) != '0') {

dp[i] += dp[i - 1];

}

// Check if the last two characters form a valid number (10 to 26)

int twoDigit = Integer.parseInt(s.substring(i - 2, i));

if (twoDigit >= 10 && twoDigit <= 26) {

dp[i] += dp[i - 2];

}

}

return dp[n];

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Test Case 1

String s1 = "226";

System.out.println("Number of ways to decode '" + s1 + "': " + solution.numDecodings(s1));

// Test Case 2

String s2 = "12";

System.out.println("Number of ways to decode '" + s2 + "': " + solution.numDecodings(s2));

// Test Case 3

String s3 = "06"; // Invalid encoding

System.out.println("Number of ways to decode '" + s3 + "': " + solution.numDecodings(s3));

// Test Case 4

String s4 = "11106";

System.out.println("Number of ways to decode '" + s4 + "': " + solution.numDecodings(s4));

}

}

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return the **maximum** profit you can achieve.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 7

**Explanation:** Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Total profit is 4 + 3 = 7.

class Solution {

public int maxProfit(int[] prices) {

int sum = 0;

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

if (prices[i] < prices[i+1]){

sum = sum + prices[i+1] - prices[i];

}

}

return sum;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Test Case 1: Prices increase then decrease

int[] prices1 = {7, 1, 5, 3, 6, 4};

System.out.println("Max Profit for prices1: " + solution.maxProfit(prices1)); // Output: 7

// Test Case 2: Prices always increasing

int[] prices2 = {1, 2, 3, 4, 5};

System.out.println("Max Profit for prices2: " + solution.maxProfit(prices2)); // Output: 4

// Test Case 3: Prices always decreasing

int[] prices3 = {7, 6, 4, 3, 1};

System.out.println("Max Profit for prices3: " + solution.maxProfit(prices3)); // Output: 0

// Test Case 4: No transactions possible (single price)

int[] prices4 = {5};

System.out.println("Max Profit for prices4: " + solution.maxProfit(prices4)); // Output: 0

}

}

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

**Input:** grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

**Output:** 1

class Solution {

public int numIslands(char[][] grid) {

int rows = grid.length;

int cols = grid[0].length;

int numberofIslands = 0;

for(int i = 0; i < rows; i++){

for(int j = 0; j < cols; j++){

if (grid[i][j] == '1'){

numberofIslands++;

bfs(grid,i,j);

}

}

}

return numberofIslands;

}

private void bfs(char[][] grid, int x, int y){

int rows = grid.length;

int cols = grid[0].length;

Deque<int []> dq = new LinkedList<>();

int [][] dir = { {1,0},{-1,0},{0,1},{0,-1}};

dq.offer(new int[] {x,y});

grid[x][y] = '0';

while(!dq.isEmpty()){

int [] cell = dq.poll();

int r = cell[0];

int c = cell[1];

for(int [] d : dir){

int nr = r + d[0];

int nc = c + d[1];

if (nr >=0 && nr <rows && nc >=0 && nc < cols && grid[nr][nc] == '1'){

dq.offer(new int[] {nr,nc});

grid[nr][nc] = '0';

}

}

}

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Test Case 1: Small grid with islands

char[][] grid1 = {

{'1', '1', '0', '0', '0'},

{'1', '1', '0', '0', '0'},

{'0', '0', '1', '0', '0'},

{'0', '0', '0', '1', '1'}

};

System.out.println("Number of islands in grid1: " + solution.numIslands(grid1)); // Output: 3

// Test Case 2: Single island

char[][] grid2 = {

{'1', '1', '1', '1', '0'},

{'1', '1', '0', '1', '0'},

{'1', '1', '0', '0', '0'},

{'0', '0', '0', '0', '0'}

};

System.out.println("Number of islands in grid2: " + solution.numIslands(grid2)); // Output: 1

// Test Case 3: No islands

char[][] grid3 = {

{'0', '0', '0'},

{'0', '0', '0'},

{'0', '0', '0'}

};

System.out.println("Number of islands in grid3: " + solution.numIslands(grid3)); // Output: 0

// Test Case 4: All land

char[][] grid4 = {

{'1', '1'},

{'1', '1'}

};

System.out.println("Number of islands in grid4: " + solution.numIslands(grid4)); // Output: 1

// Test Case 5: Mixed grid

char[][] grid5 = {

{'1', '0', '1', '0', '1'},

{'0', '1', '0', '1', '0'},

{'1', '0', '1', '0', '1'}

};

System.out.println("Number of islands in grid5: " + solution.numIslands(grid5)); // Output: 9

}

}

You are given an integer **mountain** array arr of length n where the values increase to a **peak element** and then decrease.

Return the index of the peak element.

Your task is to solve it in O(log(n)) time complexity.

**Example 1:**

**Input:** arr = [0,1,0]

**Output:** 1

**Example 2:**

**Input:** arr = [0,2,1,0]

**Output:** 1

**Example 3:**

**Input:** arr = [0,10,5,2]

**Output:** 1

class Solution {

public int peakIndexInMountainArray(int[] arr) {

int l = 0, r = arr.length - 1;

while (l < r) {

int m1 = l + (r - l) / 3;

int m2 = r - (r - l) / 3;

if (arr[m1] < arr[m2]) {

l = m1 + 1;

} else {

r = m2 - 1;

}

// Uncomment the following line if you want to see the range during execution

// System.out.println("l: " + l + ", r: " + r);

}

return l; // or r, since l == r when the loop exits

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Test Case 1: Standard mountain array

int[] arr1 = {1, 3, 5, 7, 6, 4, 2};

System.out.println("Peak index in arr1: " + solution.peakIndexInMountainArray(arr1)); // Output: 3 (index of 7)

// Test Case 2: Single peak at the middle

int[] arr2 = {0, 2, 4, 6, 5, 3, 1};

System.out.println("Peak index in arr2: " + solution.peakIndexInMountainArray(arr2)); // Output: 3 (index of 6)

// Test Case 3: Edge case with small array

int[] arr3 = {0, 10, 5};

System.out.println("Peak index in arr3: " + solution.peakIndexInMountainArray(arr3)); // Output: 1 (index of 10)

// Test Case 4: Longer array

int[] arr4 = {0, 1, 3, 5, 9, 12, 15, 14, 8, 4, 2, 0};

System.out.println("Peak index in arr4: " + solution.peakIndexInMountainArray(arr4)); // Output: 6 (index of 15)

// Test Case 5: Edge case with only one possible peak

int[] arr5 = {3, 5, 3};

System.out.println("Peak index in arr5: " + solution.peakIndexInMountainArray(arr5)); // Output: 1 (index of 5)

}

}

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

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

**Example 1:**

**Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** nums = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

class Solution {

public int search(int[] arr, int target) {

int low = 0, high = arr.length - 1;

while (low <= high && target >= arr[low] && target <= arr[high]) {

// Prevent division by zero and calculate pos

if (low == high) {

if (arr[low] == target) return low;

return -1;

}

// Estimate the position

int pos = low + ((target - arr[low]) \* (high - low)) / (arr[high] - arr[low]);

// Check if the target is found

if (arr[pos] == target) {

return pos;

}

// If the target is larger, move the low pointer

if (arr[pos] < target) {

low = pos + 1;

}

// If the target is smaller, move the high pointer

else {

high = pos - 1;

}

}

return -1; // Target not found

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Test Case 1: Standard case where the target exists in the array

int[] arr1 = {10, 20, 30, 40, 50, 60, 70, 80, 90};

int target1 = 60;

System.out.println("Index of " + target1 + ": " + solution.search(arr1, target1)); // Output: 5

// Test Case 2: Target exists at the beginning of the array

int[] arr2 = {1, 3, 5, 7, 9};

int target2 = 1;

System.out.println("Index of " + target2 + ": " + solution.search(arr2, target2)); // Output: 0

// Test Case 3: Target exists at the end of the array

int[] arr3 = {2, 4, 6, 8, 10};

int target3 = 10;

System.out.println("Index of " + target3 + ": " + solution.search(arr3, target3)); // Output: 4

// Test Case 4: Target is not present in the array

int[] arr4 = {1, 3, 5, 7, 9, 11, 13};

int target4 = 6;

System.out.println("Index of " + target4 + ": " + solution.search(arr4, target4)); // Output: -1

// Test Case 5: Edge case with an empty array

int[] arr5 = {};

int target5 = 5;

System.out.println("Index of " + target5 + ": " + solution.search(arr5, target5)); // Output: -1

}

}