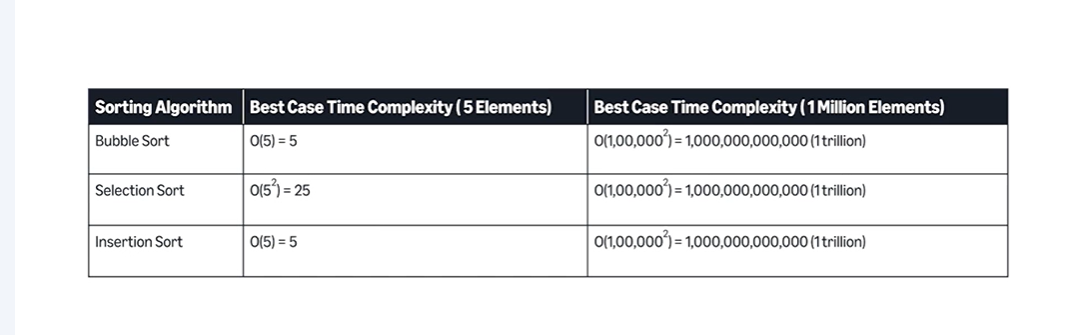
**Coding Problems / Data Structure and algorithm:-**

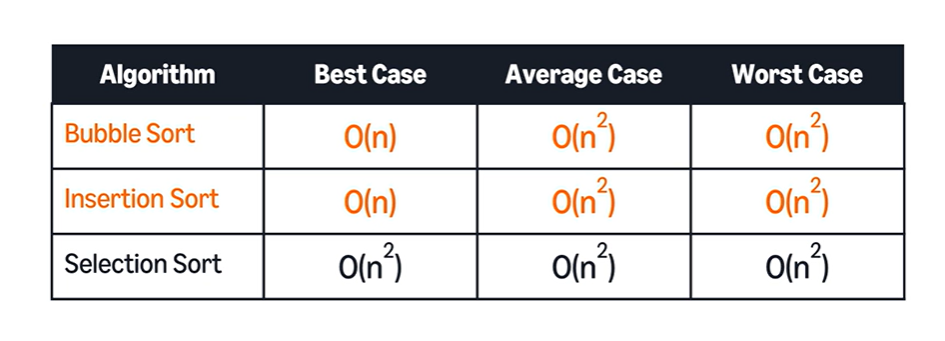
**Link -** [**https://studyalgorithms.com/**](https://studyalgorithms.com/)

**GitHub:-** [**https://github.com/nikoo28/java-solutions/blob/master/src/main/java/leetcode/easy/LengthOfLastWord.java**](https://github.com/nikoo28/java-solutions/blob/master/src/main/java/leetcode/easy/LengthOfLastWord.java)

**JAVA:-** [**https://www.baeldung.com/java-comparator-comparable**](https://www.baeldung.com/java-comparator-comparable)

**JAVA Springboot site:-** [**https://www.baeldung.com/**](https://www.baeldung.com/)





70 Leetcode problems in

YouTube Link**:-** <https://www.youtube.com/watch?v=lvO88XxNAzs&t=18991s>

**📕 Chapters:**

|  |  |  |
| --- | --- | --- |
| Sr No | Problem | Reviewed or not |
| 1 | Steps to get Hired into Tech |  |
| 2 | Big O Notation |  |
| 3 | Problem Solving Techniques |  |
| 4 | SECTION - ARRAYS: Contains Duplicate |  |
| 5 | Missing Number |  |
| 6 | Note: Sorting, Dictionary, Lambdas |  |
| 7 | Find All Numbers Disappeared in an Array |  |
| 8 | Two Sum |  |
| 9 | Note: Java vs Python - Final Value After Operations |  |
| 10 | How Many Numbers Are Smaller Than the Current Number |  |
| 11 | Minimum Time Visiting All Points |  |
| 12 | Spiral Matrix |  |
| 13 | Number of Islands/ DFS |  |
| 14 | SECTION - ARRAYS TWO POINTERS: Best Time to Buy and Sell Stock |  |
| 15 | Squares of a Sorted Array |  |
| 16 | 3Sum |  |
| 17 | Longest Mountain in Array |  |
| 18 | SECTION - ARRAYS SLIDING WINDOW: Contains Duplicate II |  |
| 19 | Minimum Absolute Difference |  |
| 20 | Minimum Size Subarray Sum |  |
| 21 | SECTION - BIT MANIPULATION: Single Number |  |
| 22 | SECTION - DYNAMIC PROGRAMMING: Coin Change |  |
| 23 | Climbing Stairs |  |
| 24 | Maximum Subarray |  |
| 25 | Counting Bits |  |
| 26 | Range Sum Query - Immutable |  |
| 27 | SECTION - BACKTRACKING: Letter Case Permutation |  |
| 28 | Subsets |  |
| 29 | Combinations |  |
| 30 | Permutations |  |
| 31 | SECTION - LINKED LISTS: Middle of Linked List |  |
| 32 | Linked List Cycle |  |
| 33 | Reverse Linked List |  |
| 34 | Remove Linked List Elements |  |
| 35 | Reverse Linked List II |  |
| 36 | Palindrome Linked List |  |
| 37 | Merge Two Sorted Lists |  |
| 38 | SECTION - STACKS: Min Stack |  |
| 39 | Valid Parentheses |  |
| 40 | Evaluate Reverse Polish Notation |  |
| 41 | Stack Sorting |  |
| 42 | SECTION - QUEUES: Implement Stack using Queues |  |
| 43 | Time Needed to Buy Tickets |  |
| 44 | Reverse the First K Elements of a Queue |  |
| 45 | SECTION - BINARY TREES: Average of Levels in Binary Tree |  |
| 46 | Minimum Depth of Binary Tree |  |
| 47 | Maximum Depth of Binary Tree |  |
| 48 | Min/Max Value Binary Tree |  |
| 49 | Binary Tree Level Order Traversal |  |
| 50 | Same Tree |  |
| 51 | Path Sum |  |
| 52 | Diameter of a Binary Tree |  |
| 53 | Invert Binary Tree |  |
| 54 | Lowest Common Ancestor of a Binary Tree |  |
| 55 | SECTION - BINARY SEARCH TREES: Search in a Binary Search Tree |  |
| 56 | Insert into a Binary Search Tree |  |
| 57 | Convert Sorted Array to Binary Search Tree |  |
| 58 | Two Sum IV - Input is a BST |  |
| 59 | Lowest Common Ancestor of a Binary Search Tree |  |
| 60 | Minimum Absolute Difference in BST |  |
| 61 | Balance a Binary Search Tree |  |
| 62 | Delete Node in a BST |  |
| 63 | Kth Smallest Element in a BST |  |
| 64 | SECTION - HEAPS: Kth Largest Element in an Array |  |
| 65 | K Closest Points to Origin |  |
| 66 | Top K Frequent Elements |  |
| 67 | Task Scheduler |  |
| 68 | SECTION - GRAPHS: Breadth and Depth First Traversal |  |
| 69 | Clone Graph |  |
| 70 | Core Graph Operations |  |
| 71 | Cheapest Flights Within K Stops |  |
| 72 | Course Schedule |  |
| 73 | Outro |  |

**For DE – Go for Easy to Medium LC**

* **Arrays**
* **Strings**
* **Binary Search**
* **Recursion**
* **Stacks**
* **Queues**
* **DP(Basic 1D)**
* **Linked Lists**
* **Sliding Window**

1. **Problem: Arrays Contains Duplicate**

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

///////////Python Solution

def containsDuplicate(nums):

    return len(nums) != len(set(nums))

# Example usage

nums = [1, 2, 3, 1]

print(containsDuplicate(nums))  # Output: True

////////Java Solution

import java.util.HashSet;

public class ContainsDuplicate {

    public static boolean containsDuplicate(int[] nums) {

        HashSet<Integer> set = new HashSet<>();

        for (int num : nums) {

            if (set.contains(num)) {

                return true;

            }

            set.add(num);

        }

        return false;

    }

    public static void main(String[] args) {

        int[] nums = {1, 2, 3, 1};

        System.out.println(containsDuplicate(nums));  // Output: true

    }

}

### ****Problem: Missing Number****

**Description:**  
Given an array nums containing n distinct numbers in the range [0, n], return the missing number.

///////Python

def missingNumber(nums):

    n = len(nums)

    expected\_sum = n \* (n + 1) // 2  # Sum of first n natural numbers

    actual\_sum = sum(nums)

    return expected\_sum - actual\_sum

# Example usage

nums = [3, 0, 1]

print(missingNumber(nums))  # Output: 2

////////Java Solution

public class MissingNumber {

    public static int missingNumber(int[] nums) {

        int n = nums.length;

        int expectedSum = n \* (n + 1) / 2;  // Sum of first n natural numbers

        int actualSum = 0;

        for (int num : nums) {

            actualSum += num;

        }

        return expectedSum - actualSum;

    }

    public static void main(String[] args) {

        int[] nums = {3, 0, 1};

        System.out.println(missingNumber(nums));  // Output: 2

    }

}

### ****Note: Sorting, Dictionary, Lambdas (On hold Video needs to be checked)****

**Description:**  
Given an array nums containing n distinct numbers in the range [0, n], return the missing number.

### ****Problem: Find All Numbers Disappeared in an Array****

**Description:**  
Given an array nums of length n where nums contains numbers from 1 to n, return an array of all the numbers that do not appear in nums.

//////////Python Solution

def findDisappearedNumbers(nums):

    n = len(nums)

    num\_set = set(nums)

    return [i for i in range(1, n + 1) if i not in num\_set]

# Example usage

nums = [4, 3, 2, 7, 8, 2, 3, 1]

print(findDisappearedNumbers(nums))  # Output: [5, 6]

//////////////JAVA

import java.util.\*;

public class FindDisappearedNumbers {

    public static List<Integer> findDisappearedNumbers(int[] nums) {

        int n = nums.length;

        HashSet<Integer> numSet = new HashSet<>();

        for (int num : nums) {

            numSet.add(num);

        }

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

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

            if (!numSet.contains(i)) {

                missingNumbers.add(i);

            }

        }

        return missingNumbers;

    }

    public static void main(String[] args) {

        int[] nums = {4, 3, 2, 7, 8, 2, 3, 1};

        System.out.println(findDisappearedNumbers(nums));  // Output: [5, 6]

    }

}

### ****Problem: Two Sum****

**Description:**  
Given an array of integers nums and an integer target, return the **indices** of the two numbers such that they add up to target.  
Assume that each input has **exactly one solution**, and you may not use the same element twice.

/////Python Function

def twoSum(nums, target):

    num\_map = {}  # Dictionary to store seen numbers and their indices

    for i, num in enumerate(nums):

        complement = target - num

        if complement in num\_map:

            return [num\_map[complement], i]  # Return indices of the two numbers

        num\_map[num] = i  # Store current number with its index

# Example usage

nums = [2, 7, 11, 15]

target = 9

print(twoSum(nums, target))  # Output: [0, 1]

/////JAVA Solution

import java.util.HashMap;

public class TwoSum {

    public static int[] twoSum(int[] nums, int target) {

        HashMap<Integer, Integer> numMap = new HashMap<>();

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

            int complement = target - nums[i];

            if (numMap.containsKey(complement)) {

                return new int[]{numMap.get(complement), i};  // Return indices

            }

            numMap.put(nums[i], i);  // Store number with its index

        }

        return new int[]{};  // Should never reach here

    }

    public static void main(String[] args) {

        int[] nums = {2, 7, 11, 15};

        int target = 9;

        int[] result = twoSum(nums, target);

        System.out.println("[" + result[0] + ", " + result[1] + "]");  // Output: [0, 1]

    }

}

### ****Problem: Final Value After Operations****

**Description:**  
You are given an array of strings operations where each operation is "++X", "X++", "--X", or "X--".

* "++X" and "X++" increase the value of X by 1.
* "--X" and "X--" decrease the value of X by 1.

Return the **final value** of X after performing all operations, assuming X starts at 0.

//////Python

def finalValueAfterOperations(operations):

    x = 0

    for op in operations:

        if "+" in op: ////// op[1] == "+" instead you can check this too.

            x += 1

        else:

            x -= 1

    return x

# Example usage

operations = ["--X", "X++", "X++"]

print(finalValueAfterOperations(operations))  # Output: 1

////Java Solution

public class FinalValueAfterOperations {

    public static int finalValueAfterOperations(String[] operations) {

        int x = 0;

        for (String op : operations) {

            if (op.contains("+")) {  //////op[1] == "+" instead you can check this too.

                x += 1;

            } else {

                x -= 1;

            }

        }

        return x;

    }

    public static void main(String[] args) {

        String[] operations = {"--X", "X++", "X++"};

        System.out.println(finalValueAfterOperations(operations));  // Output: 1

    }

}

### ****Problem: How Many Numbers Are Smaller Than the Current Number****

**Description:**  
Given an array nums, return an array where each element at index i contains the count of numbers in nums that are **smaller** than nums[i].

**Solution**:- Sorting and removing duplicates from the array is the main solution, then the index of the num is returned from sorted/distinct array items.

/////Python Solution

def smallerNumbersThanCurrent(nums):

    sorted\_nums = sorted(nums)  # Sort the numbers

    rank\_map = {num: i for i, num in enumerate(sorted\_nums) if num not in rank\_map}  # Store first occurrence index

    return [rank\_map[num] for num in nums]

# Example usage

nums = [8, 1, 2, 2, 3]

print(smallerNumbersThanCurrent(nums))  # Output: [4, 0, 1, 1, 3]

/////Java Solution

import java.util.Arrays;

import java.util.HashMap;

public class SmallerNumbersThanCurrent {

    public static int[] smallerNumbersThanCurrent(int[] nums) {

        int[] sortedNums = nums.clone();

        Arrays.sort(sortedNums);  // Sort the array

        HashMap<Integer, Integer> rankMap = new HashMap<>();

        for (int i = 0; i < sortedNums.length; i++) {

            rankMap.putIfAbsent(sortedNums[i], i);  // Store first occurrence index

        }

        int[] result = new int[nums.length];

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

            result[i] = rankMap.get(nums[i]);

        }

        return result;

    }

    public static void main(String[] args) {

        int[] nums = {8, 1, 2, 2, 3};

        System.out.println(Arrays.toString(smallerNumbersThanCurrent(nums)));  // Output: [4, 0, 1, 1, 3]

    }

}

### ****Minimum Time Visiting All Points:-****

### The problem **"Minimum Time Visiting All Points"** can be solved using the **Chebyshev distance**, where the time taken to move from one point to another is determined by the maximum of the absolute differences in x and y coordinates.

### 

### ****Problem Statement:****

Given an array points where points[i] = [x\_i, y\_i] represents the coordinates of a point on a 2D plane, you need to visit all points in order. You can move 1 step **horizontally, vertically, or diagonally** in **one second**. Find the minimum time required to visit all points.

### ****Approach:****

To move from (x1, y1) to (x2, y2), the number of steps required is max(abs(x2 - x1), abs(y2 - y1)) because diagonal movement allows covering both axes in one step when possible.

/////Python

from typing import List

def minTimeToVisitAllPoints(points: List[List[int]]) -> int:

    time = 0

    for i in range(1, len(points)):

        x1, y1 = points[i - 1]

        x2, y2 = points[i]

        time += max(abs(x2 - x1), abs(y2 - y1))

    return time

# Example usage

points = [[1,1], [3,4], [-1,0]]

print(minTimeToVisitAllPoints(points))  # Output: 7

//////Java Solution:

class Solution {

    public int minTimeToVisitAllPoints(int[][] points) {

        int time = 0;

        for (int i = 1; i < points.length; i++) {

            int x1 = points[i - 1][0], y1 = points[i - 1][1];

            int x2 = points[i][0], y2 = points[i][1];

            time += Math.max(Math.abs(x2 - x1), Math.abs(y2 - y1));

        }

        return time;

    }

    public static void main(String[] args) {

        Solution sol = new Solution();

        int[][] points = {{1, 1}, {3, 4}, {-1, 0}};

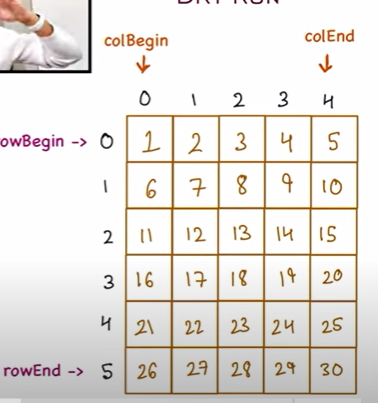
        System.out.println(sol.minTimeToVisitAllPoints(points));  // Output: 7

    }

}

### ****Problem: Spiral Matrix****

**Description:**  
Given an m x n matrix, return all elements in **spiral order** (clockwise traversal starting from the top-left corner).



### ****Approach:****

1. Maintain **four boundaries**:
   * top (starting row)
   * bottom (ending row)
   * left (starting column)
   * right (ending column)
2. Traverse **right** → **down** → **left** → **up**, updating boundaries.
3. Stop when all elements are visited.

**Explanation By Neha:-**

             TOP                              Bottom

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

          LC     RC

          CB     CE

matrix =     LC     RC

      Top [[1, 2, 3],

          [4, 5, 6],

          [7, 8, 9],

          [10, 11, 12]]

        Bottom

LC = 0

RC = len(matrix[0])-1

TOP = 0

Bottom = len(matrix)-1

output = 1, 2, 3, 6, 9, 12,11,10,7,4,

while left<right and top<bottom

for loop left to right- matrix[top][i]

    print first element of matrix (1,2,3)

    matrix[top][i]

    Top=Top+1

for loop - Top to Bottom matrix[i][Rc] end element of rest of the matrix (6,9,12)

    matrix[i][Rc]

    RC = RC-1

if top<bottom // To check if rows are left

    for RC, Left, -1    (11,10)

        matrix[bottom][i]

        Bottom=bottom-1

if left < right // To check if columns are left

    for Bottom, top-1, -1  (7,4)

        matrix[i][LC]

        LC=LC+1

**Python Solution:-**

def spiralOrder(matrix):

    if not matrix or not matrix[0]:

        return []

    result = []

    ##row\_begin,row\_end,col\_begin, col\_end

    top, bottom, left, right = 0, len(matrix) - 1, 0, len(matrix[0]) - 1

    while top <= bottom and left <= right:

        # Move right

        for i in range(left, right + 1):

            result.append(matrix[top][i])

        top += 1  # Move the top boundary down

        # Move down

        for i in range(top, bottom + 1):

            result.append(matrix[i][right])

        right -= 1  # Move the right boundary left

        if top <= bottom:

            # Move left

            for i in range(right, left - 1, -1):

                result.append(matrix[bottom][i])

            bottom -= 1  # Move the bottom boundary up

        if left <= right:

            # Move up

            for i in range(bottom, top - 1, -1):

                result.append(matrix[i][left])

            left += 1  # Move the left boundary right

    return result

# Example usage

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(spiralOrder(matrix))  # Output: [1, 2, 3, 6, 9, 8, 7, 4, 5]

**JAVA Solution:-**

//JAVA Solution

import java.util.\*;

public class SpiralMatrix {

    public static List<Integer> spiralOrder(int[][] matrix) {

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

        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return result;

        int top = 0, bottom = matrix.length - 1;

        int left = 0, right = matrix[0].length - 1;

        while (top <= bottom && left <= right) {

            // Move right

            for (int i = left; i <= right; i++)

                result.add(matrix[top][i]);

            top++; // Move the top boundary down

            // Move down

            for (int i = top; i <= bottom; i++)

                result.add(matrix[i][right]);

            right--; // Move the right boundary left

            if (top <= bottom) {

                // Move left

                for (int i = right; i >= left; i--)

                    result.add(matrix[bottom][i]);

                bottom--; // Move the bottom boundary up

            }

            if (left <= right) {

                // Move up

                for (int i = bottom; i >= top; i--)

                    result.add(matrix[i][left]);

                left++; // Move the left boundary right

            }

        }

        return result;

    }

    public static void main(String[] args) {

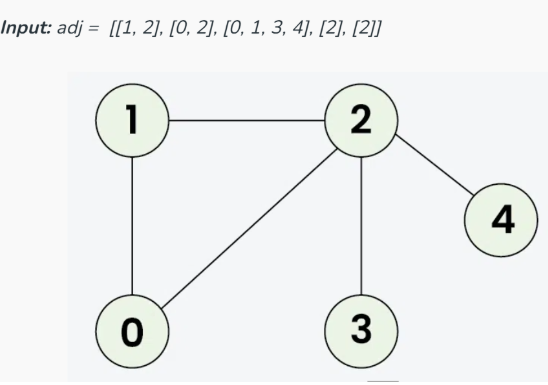
        int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

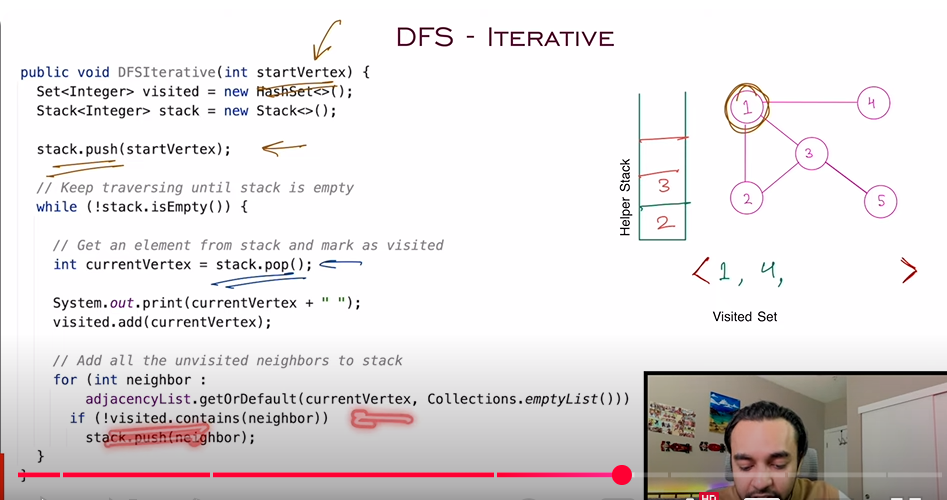
        System.out.println(spiralOrder(matrix));  // Output: [1, 2, 3, 6, 9, 8, 7, 4, 5]

    }

}

1. **DFS (Before starting Number of Islands problem):- DFS(Depth first search)**





**The image represents an undirected graph with an adjacency list given as:**

**adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]**

### Understanding the adjacency list:

* Node 0 is connected to nodes [1, 2]
* Node 1 is connected to nodes [0, 2]
* Node 2 is connected to nodes [0, 1, 3, 4]
* Node 3 is connected to node [2]
* Node 4 is connected to node [2]

**DFS using Stack :-**

//DFS using stack

import java.util.\*;

public class Graph {

    private Map<Integer, List<Integer>> adjacencyList = new HashMap<>();

    public void addEdge(int u, int v) {

        adjacencyList.computeIfAbsent(u, k -> new ArrayList<>()).add(v);

        adjacencyList.computeIfAbsent(v, k -> new ArrayList<>()).add(u); // For undirected graph

    }

    public void DFSIterative(int startVertex) {

        Set<Integer> visited = new HashSet<>();

        Stack<Integer> stack = new Stack<>();

        stack.push(startVertex);

        while (!stack.isEmpty()) {

            int currentVertex = stack.pop();

            if (!visited.contains(currentVertex)) {

                System.out.print(currentVertex + " ");

                visited.add(currentVertex);

            }

            for (int neighbor : adjacencyList.getOrDefault(currentVertex, Collections.emptyList())) {

                if (!visited.contains(neighbor)) {

                    stack.push(neighbor);

                }

            }

        }

    }

    public static void main(String[] args) {

        Graph graph = new Graph();

        graph.addEdge(1, 2);

        graph.addEdge(1, 3);

        graph.addEdge(1, 4);

        graph.addEdge(2, 3);

        System.out.println("DFS Traversal:");

        graph.DFSIterative(1);

    }

}

**The graph created with edges:-**

**graph.addEdge(1, 2);**

**graph.addEdge(1, 3);**

**graph.addEdge(1, 4);**

**graph.addEdge(2, 3);**

This results in the following **adjacency list**:

| **Vertex** | **Neighbors** |
| --- | --- |
| 1 | [2, 3, 4] |
| 2 | [1, 3] |
| 3 | [1, 2] |
| 4 | [1] |

**Graph Visualization**

**1**

**/|\**

**2 3 4**

**\|**

**3**

## **2. Step-by-Step DFS Execution (Starting from 1)**

### ****Initialization****

* **Stack:** [1]
* **Visited:** {}

### ****Iteration 1****

* **Pop 1** from stack.
* **Visit 1**, mark as visited: {1}
* **Push neighbors** [2, 3, 4] into stack (order matters).
* **Stack:** [2, 3, 4]
* **Visited:** {1}
* **Output:** 1

### ****Iteration 2****

* **Pop 4** from stack.
* **Visit 4**, mark as visited: {1, 4}
* **Push neighbors** of 4 ([1]), but 1 is already visited.
* **Stack:** [2, 3]
* **Visited:** {1, 4}
* **Output:** 1 4

### ****Iteration 3****

* **Pop 3** from stack.
* **Visit 3**, mark as visited: {1, 3, 4}
* **Push neighbors** [1, 2] (only 2 is unvisited).
* **Stack:** [2, 2]
* **Visited:** {1, 3, 4}
* **Output:** 1 4 3

### ****Iteration 4****

* **Pop 2** from stack.
* **Visit 2**, mark as visited: {1, 2, 3, 4}
* **Push neighbors** [1, 3] (both are already visited).
* **Stack:** [2]
* **Visited:** {1, 2, 3, 4}
* **Output:** 1 4 3 2

### ****Iteration 5****

* **Pop 2** again.
* 2 is already visited, so do nothing.
* **Stack:** [] (empty)
* **Visited:** {1, 2, 3, 4}
* **Final Output:** 1 4 3 2

**Final DFS Traversal Order**

**1 4 3 2**

## **Summary of DFS Execution**

| **Step** | **Stack** | **Visited Set** | **Output** |
| --- | --- | --- | --- |
| 1 | [2, 3, 4] | {1} | 1 |
| 2 | [2, 3] | {1, 4} | 1 4 |
| 3 | [2, 2] | {1, 3, 4} | 1 4 3 |
| 4 | [2] | {1, 2, 3, 4} | 1 4 3 2 |
| 5 | [] (empty) | {1, 2, 3, 4} | 1 4 3 2 |

**DFS using Arraylist** (Java Solution)

This solution uses an **ArrayList** to represent the adjacency list and implements **Depth-First Search (DFS) recursively**.

Java Code:-

import java.util.\*;

public class Graph {

    private List<List<Integer>> adjacencyList;

    public Graph(int vertices) {

        adjacencyList = new ArrayList<>();

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

            adjacencyList.add(new ArrayList<>()); // Initialize adjacency list

        }

    }

    public void addEdge(int u, int v) {

        adjacencyList.get(u).add(v);

        adjacencyList.get(v).add(u); // For an undirected graph

    }

    public void DFS(int startVertex) {

        boolean[] visited = new boolean[adjacencyList.size()];

        System.out.print("DFS Traversal: ");

        DFSRecursive(startVertex, visited);

    }

    private void DFSRecursive(int vertex, boolean[] visited) {

        visited[vertex] = true;

        System.out.print(vertex + " ");

        for (int neighbor : adjacencyList.get(vertex)) {

            if (!visited[neighbor]) {

                DFSRecursive(neighbor, visited);

            }

        }

    }

    public static void main(String[] args) {

        Graph graph = new Graph(5); // 5 vertices (0 to 4)

        // Adding edges

        graph.addEdge(0, 1);

        graph.addEdge(0, 2);

        graph.addEdge(1, 2);

        graph.addEdge(2, 3);

        graph.addEdge(2, 4);

        // Performing DFS

        graph.DFS(0);

    }

}

## **Step-by-Step Execution for DFS(0)**

### ****Graph Representation****

Adjacency List:

0 -> [1, 2]

1 -> [0, 2]

2 -> [0, 1, 3, 4]

3 -> [2]

4 -> [2]

### ****Step-by-Step DFS Traversal from**** 0

1. Start at 0, mark as visited {0}.
2. Visit 1 (first neighbor of 0), {0, 1}.
3. Visit 2 (next neighbor of 1), {0, 1, 2}.
4. Visit 3 (neighbor of 2), {0, 1, 2, 3}.
5. Backtrack to 2, visit 4, {0, 1, 2, 3, 4}.

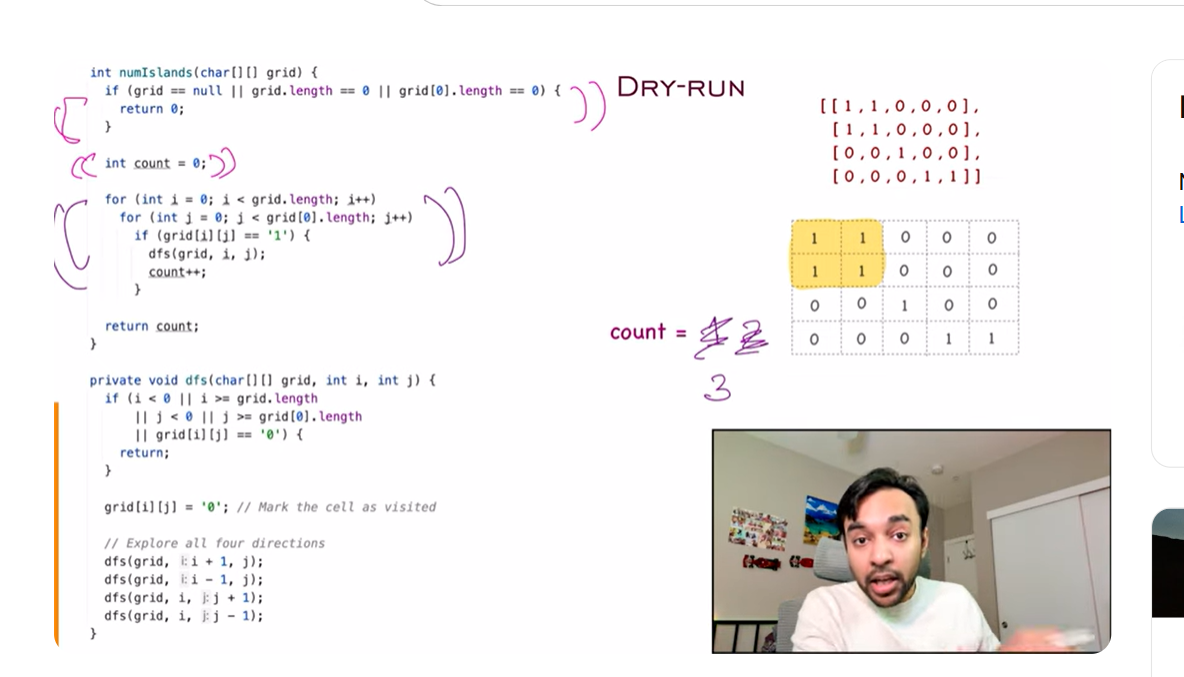
Final DFS Order:-

0 1 2 3 4

## **DFS Traversal Breakdown (Recursive Call Stack)**

| **Step** | **Call Stack** | **Visited Nodes** |
| --- | --- | --- |
| 1 | DFS(0) | {0} |
| 2 | DFS(1) | {0,1} |
| 3 | DFS(2) | {0,1,2} |
| 4 | DFS(3) | {0,1,2,3} |
| 5 | DFS(4) | {0,1,2,3,4} |
| 6 | Backtrack | {0,1,2,3,4} |

**Number of islands :-**



* + **Search will go through each**

**I = number of lists within the list – length(list)**

**List = [[1,1,0,0,0], [1,1,0,0,0], [0,0,1,0,0], [0,0,0,1,1]]**

**J= each item in the individual list – length(list[0])**

**If grid[i][j] == 1{**

**Dfs(I,j)**

**Count = count++**

**}**

public class NumberOfIslands {

    public static void main(String[] args) {

        char[][] grid = {

            {'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: " + numIslands(grid));

    }

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

        if (grid == null || grid.length == 0) return 0;

        int count = 0;

        int rows = grid.length, cols = grid[0].length;

        // Traverse the entire grid

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

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

                if (grid[i][j] == '1') {  // Found an unvisited island

                    count++;  // Increase island count

                    dfs(grid, i, j);  // Sink the island

                }

            }

        }

        return count;

    }

    private static void dfs(char[][] grid, int i, int j) {

        int rows = grid.length, cols = grid[0].length;

        // Boundary & water check

        if (i < 0 || i >= rows || j < 0 || j >= cols || grid[i][j] == '0') return;

        grid[i][j] = '0'; // Mark as visited (sink the island)

        // Explore all 4 directions

        dfs(grid, i + 1, j); // Down 1,0 -->

        dfs(grid, i - 1, j); // Up -1,0 --> 0,0

        dfs(grid, i, j + 1); // Right  0,1 --> for next recursion down --> 1,1

        dfs(grid, i, j - 1); // Left 0,-1 --> 0,0

    }

} Try to check this with respect to matrix. See how up , down , right and left works recursively.

## **Step-by-Step DFS Calls**

At (0,0), we recursively visit all 4 possible directions:

* **Down:** (1,0)
* **Up:** (-1,0) (out of bounds, ignored)
* **Right:** (0,1)
* **Left:** (0,-1) (out of bounds, ignored)

### ****Step 1: DFS at (0,0)****

* **Mark (0,0) as 0**
* Call dfs(1,0) (Down)
* Call dfs(0,1) (Right)

### ****Step 2: DFS at (1,0)****

* **Mark (1,0) as 0**
* Call dfs(2,0) (Down) → 0, return
* Call dfs(0,0) (Up) → Already 0, return
* Call dfs(1,1) (Right)

### ****Step 3: DFS at (1,1)****

* **Mark (1,1) as 0**
* Call dfs(2,1) (Down) → 0, return
* Call dfs(0,1) (Up) → Already 0, return
* Call dfs(1,2) (Right) → 0, return
* Call dfs(1,0) (Left) → Already 0, return

### ****Step 4: DFS at (0,1)****

* **Mark (0,1) as 0**
* Call dfs(1,1) (Down) → Already 0, return
* Call dfs(-1,1) (Up) → Out of bounds, return
* Call dfs(0,2) (Right) → 0, return
* Call dfs(0,0) (Left) → Already 0, return

**Updated Grid After First Island is Processed**

0 0 0 0 0

0 0 0 0 0

0 0 1 0 0

0 0 0 1 1

 **Now, the outer loop continues scanning the grid.**

 It will detect the next 1 at (2,2) and start another DFS.

1. **SECTION - ARRAYS TWO POINTERS: Best Time to Buy and Sell Stock**

/////Python/////

def maxProfit(prices):

    min\_price = float('inf')

    max\_profit = 0

    for price in prices:

        if price < min\_price:

            min\_price = price

        else:

            max\_profit = max(max\_profit, price - min\_price)

    return max\_profit

//Example

prices = [7, 1, 5, 3, 6, 4]

print(maxProfit(prices))  # Output: 5

/////Java Solution://///

class Solution {

    public int maxProfit(int[] prices) {

        int minPrice = Integer.MAX\_VALUE;

        int maxProfit = 0;

        for (int price : prices) {

            if (price < minPrice) {

                minPrice = price;

            } else {

                maxProfit = Math.max(maxProfit, price - minPrice);

            }

        }

        return maxProfit;

    }

    public static void main(String[] args) {

        Solution sol = new Solution();

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

        System.out.println(sol.maxProfit(prices)); // Output: 5

    }

}

**Same solution. Selling price should be greater to get the profit, if not find the next max value.**

def maxProfit(prices):

    buy = 0  # Start of the sliding window

    max\_profit = 0

    for sell in range(1, len(prices)):  # Expanding the window

        if prices[sell] > prices[buy]:

            max\_profit = max(max\_profit, prices[sell] - prices[buy])

        else:

            buy = sell  # Shift window to the new minimum price

    return max\_profit

# Example

prices = [7, 1, 5, 3, 6, 4]

print(maxProfit(prices))  # Output: 5

1. **Squares of a Sorted Array**

#######Python########

        LP            RP

nums = [-4, -1, 0, 3, 10]

result= [0, 0, 0, 0, 0]

pos = 4

def sortedSquares(nums):

    left, right = 0, len(nums) - 1

    result = [0] \* len(nums)

    pos = len(nums) - 1  # Fill from the back

    while left <= right:

        left\_sq, right\_sq = nums[left] \*\* 2, nums[right] \*\* 2

        if left\_sq > right\_sq:

            result[pos] = left\_sq

            left += 1

        else:

            result[pos] = right\_sq

            right -= 1

        pos -= 1  # Move to next position

    return result

# Example Usage

nums = [-4, -1, 0, 3, 10]

print(sortedSquares(nums))  # Output: [0, 1, 9, 16, 100]

///////Java Solution///////////////

import java.util.Arrays;

class Solution {

    public int[] sortedSquares(int[] nums) {

        int left = 0, right = nums.length - 1;

        int[] result = new int[nums.length];

        int pos = nums.length - 1;

        while (left <= right) {

            int leftSq = nums[left] \* nums[left];

            int rightSq = nums[right] \* nums[right];

            if (leftSq > rightSq) {

                result[pos] = leftSq;

                left++;

            } else {

                result[pos] = rightSq;

                right--;

            }

            pos--;

        }

        return result;

    }

    public static void main(String[] args) {

        Solution sol = new Solution();

        int[] nums = {-4, -1, 0, 3, 10};

        System.out.println(Arrays.toString(sol.sortedSquares(nums)));

        // Output: [0, 1, 9, 16, 100]

    }

}

1. **3SUM**

**3Sum solutions to multiple problems:-** <https://www.geeksforgeeks.org/find-triplets-array-whose-sum-equal-zero/>

def threeSum(nums):

    nums.sort()  # Step 1: Sort the array

    result = []

    for i in range(len(nums) - 2):

        if i > 0 and nums[i] == nums[i - 1]:  # Skip duplicate elements

            continue

        left, right = i + 1, len(nums) - 1  # Two pointers

        while left < right:

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

            if total < 0:

                left += 1  # Increase sum by moving left pointer

            elif total > 0:

                right -= 1  # Decrease sum by moving right pointer

            else:

                result.append([nums[i], nums[left], nums[right]])  # Found a triplet

left += 1

                right -= 1

                # Skip duplicate values for left and right pointers

                while left < right and nums[left] == nums[left + 1]:

                    left += 1

                while left < right and nums[right] == nums[right - 1]:

                    right -= 1

    return result

# Example

nums = [-1, 0, 1, 2, -1, -4]

print(threeSum(nums))  # Output: [[-1, -1, 2], [-1, 0, 1]]

**Examples of Iterations:-**

**After Sorting:-** nums = [-4, -1, -1, 0, 1, 2]

### ****First Iteration (****i = 0****)****

* **i = 0**, nums[i] = -4
* **Left Pointer (left = i + 1 = 1)** → nums[left] = -1
* **Right Pointer (right = len(nums) - 1 = 5)** → nums[right] = 2

**Pointers:-**

i = 0 → nums[i] = -4

left = 1 → nums[left] = -1

right = 5 → nums[right] = 2

### ****Second Iteration (****i = 1****)****

* **i = 1**, nums[i] = -1
* **Left Pointer (left = i + 1 = 2)** → nums[left] = -1
* **Right Pointer (right = 5)** → nums[right] = 2

**Pointers:**

i = 1 → nums[i] = -1

left = 2 → nums[left] = -1

right = 5 → nums[right] = 2

### ****Third Iteration (****i = 2****)****

* **i = 2**, nums[i] = -1 (Duplicate of previous nums[i], so skipped)
* **Skip to next iteration**

### ****Fourth Iteration (****i = 3****)****

* **i = 3**, nums[i] = 0
* **Left Pointer (left = i + 1 = 4)** → nums[left] = 1
* **Right Pointer (right = 5)** → nums[right] = 2

Pointers:-

i = 3 → nums[i] = 0

left = 4 → nums[left] = 1

right = 5 → nums[right] = 2

**Summary:-**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **i** | **nums[i]** | **left** | **nums[left]** | **right** | **nums[right]** |
| 1 | 0 | -4 | 1 | -1 | 5 | 2 |
| 2 | 1 | -1 | 2 | -1 | 5 | 2 |
| 3 | 2 | -1 (Skipped) | - | - | - | - |
| 4 | 3 | 0 | 4 | 1 | 5 | 2 |

**JAVA:-**

import java.util.\*;

public class ThreeSum {

    public static List<List<Integer>> threeSum(int[] nums) {

        Arrays.sort(nums);

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

        int n = nums.length;

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

            if (i > 0 && nums[i] == nums[i - 1]) continue;  // Avoid duplicates

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

            while (left < right) {

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

                if (total == 0) {

                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));

                    left++;

                    right--;

                    while (lefsst < right && nums[left] == nums[left - 1]) left++; // Skip duplicates

                    while (left < right && nums[right] == nums[right + 1]) right--; // Skip duplicates

                }

                else if (total < 0) {

                    left++;

                }

                else {

                    right--;

                }

            }

        }

        return result;

    }

    public static void main(String[] args) {

        int[] nums = {-1, 0, 1, 2, -1, -4};

        System.out.println(threeSum(nums));

    }

}

Longest Mountain in Array

1. **Longest Mountain in Array**

/////Longest Mountain in Array - Step by Step Explanation

/////Problem Statement

Given an array arr, find the length of the longest mountain.

A mountain is defined as:

At least 3 elements long.

Strictly increasing sequence followed by a strictly decreasing sequence.

Example 1

Input - arr = [2, 1, 4, 7, 3, 2, 5]

Output - 5

Explanation

The longest mountain is [1, 4, 7, 3, 2] with length 5.

//////Python Solution

def longestMountain(arr):

    if len(arr) < 3:

        return 0

    max\_length = 0

    n = len(arr)

    for i in range(1, n - 1):

        if arr[i - 1] < arr[i] > arr[i + 1]:  # Peak found

            left, right = i, i

            # Move left to count increasing part

            while left > 0 and arr[left - 1] < arr[left]:

                left -= 1

            # Move right to count decreasing part

            while right < n - 1 and arr[right] > arr[right + 1]:

                right += 1

            # Calculate mountain length

            max\_length = max(max\_length, right - left + 1)

    return max\_length if max\_length >= 3 else 0

# Example Usage

arr = [2, 1, 4, 7, 3, 2, 5]

print(longestMountain(arr))  # Output: 5

////Java Solution

public class LongestMountain {

    public static int longestMountain(int[] arr) {

        if (arr.length < 3) return 0;

        int maxLength = 0, n = arr.length;

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

            if (arr[i - 1] < arr[i] && arr[i] > arr[i + 1]) {  // Peak found

                int left = i, right = i;

                // Move left to count increasing part

                while (left > 0 && arr[left - 1] < arr[left]) left--;

                // Move right to count decreasing part

                while (right < n - 1 && arr[right] > arr[right + 1]) right++;

                // Calculate mountain length

                maxLength = Math.max(maxLength, right - left + 1);

            }

        }

        return maxLength >= 3 ? maxLength : 0;

    }

    public static void main(String[] args) {

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

        System.out.println(longestMountain(arr));  // Output: 5

    }

}

1. **SECTION - ARRAYS SLIDING WINDOW: Contains Duplicate II**

//Basic Approach

def check\_duplicates\_within\_k(arr, k):

    n = len(arr)

    # Traverse for every element

    for i in range(n):

        # Traverse next k elements

        for c in range(1, k + 1):

            j = i + c

            # If we find one more occurrence within k

            if j < n and arr[i] == arr[j]:

                return True

    return False

# Driver method to test above method

arr = [10, 5, 3, 4, 3, 5, 6]

print("Yes" if check\_duplicates\_within\_k(arr, 3) else "No")

**Python:-**

// Python

def containsNearbyDuplicate(nums, k):

    window = set()

    for i in range(len(nums)):

        if nums[i] in window:

            return True  # Found a duplicate within k distance

        window.add(nums[i])

        if len(window) > k:

            window.remove(nums[i - k])  # Remove oldest element

    return False

# Example Usage

nums = [1, 2, 3, 1, 2, 3]

k = 2

print(containsNearbyDuplicate(nums, k))  # Output: False

**JAVA:-**

// Java

import java.util.HashSet;

public class ContainsDuplicateII {

    public static boolean containsNearbyDuplicate(int[] nums, int k) {

        HashSet<Integer> window = new HashSet<>();

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

            if (window.contains(nums[i])) {

                return true; // Found a duplicate within k distance

            }

            window.add(nums[i]);

            if (window.size() > k) {

                window.remove(nums[i - k]); // Remove oldest element

            }

        }

        return false;

    }

    public static void main(String[] args) {

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

        int k = 2;

        System.out.println(containsNearbyDuplicate(nums, k));  // Output: False

    }

}

1. **Minimum Absolute Difference**

Given an array of distinct integers arr, find all pairs of elements with the minimum absolute difference of any two elements.

Return a list of pairs in ascending order(with respect to pairs), each pair [a, b] follows

a, b are from arr

a < b

b - a equals to the minimum absolute difference of any two elements in arr

**Example 1:**

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

**Output:** [[1,2],[2,3],[3,4]]

Explanation: The minimum absolute difference is 1. List all pairs with difference equal to 1 in ascending order.

**Example 2:**

Input: arr = [1,3,6,10,15]

Output: [[1,3]]

**Example 3:**

Input: arr = [3,8,-10,23,19,-4,-14,27]

Output: [[-14,-10],[19,23],[23,27]]

Constraints:

2 <= arr.length <= 105

-106 <= arr[i] <= 106

/////JAVA

import java.util.\*;

class Solution {

    public List<List<Integer>> minimumAbsDifference(int[] arr) {

        Arrays.sort(arr);  // Step 1: Sort the array

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

        int minDiff = Integer.MAX\_VALUE;

        // Step 2: Find the minimum difference

        for (int i = 1; i < arr.length; i++) {

            int diff = arr[i] - arr[i - 1];

            minDiff = Math.min(minDiff, diff);

        }

        // Step 3: Collect pairs with the minimum difference

        for (int i = 1; i < arr.length; i++) {

            if (arr[i] - arr[i - 1] == minDiff) {

                result.add(Arrays.asList(arr[i - 1], arr[i]));

            }

        }

        return result;

    }

    public static void main(String[] args) {

        Solution sol = new Solution();

        int[] arr = {4, 2, 1, 3};

        System.out.println(sol.minimumAbsDifference(arr));

    }

}

///////Python

def minimum\_abs\_difference(arr):

    arr.sort()  # Step 1: Sort the array

    min\_diff = float('inf')

    result = []

    # Step 2: Find the minimum difference

    for i in range(1, len(arr)):

        min\_diff = min(min\_diff, arr[i] - arr[i - 1])

    # Step 3: Collect pairs with the minimum difference

    for i in range(1, len(arr)):

        if arr[i] - arr[i - 1] == min\_diff:

            result.append([arr[i - 1], arr[i]])

    return result

# Example usage

arr = [4, 2, 1, 3]

print(minimum\_abs\_difference(arr))

///////////////////////////////////////////////////////////////////////

1. **Minimum Size Subarray Sum (Sliding Window Approach)**

# Problem Statement

Given an array of positive integers nums and a positive integer target, the goal is to return the minimal length of a subarray whose sum is greater than or equal to target. If there is no such subarray, return 0 instead.

# Examples

1. Input: target = 7, nums = [2, 3, 1, 2, 4, 3] Output: 2 Explanation: The subarray [4, 3] has the minimal length under the problem constraint.
2. Input: target = 4, nums = [1, 4, 4] Output: 1 Explanation: The subarray [4] has the minimal length.
3. Input: target = 11, nums = [1, 1, 1, 1, 1, 1, 1, 1] Output: 0 Explanation: There is no subarray whose sum is greater than or equal to 11.

**Step-by-Step Execution**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Step** | **right Index** | **nums[right]** | **Window (nums[left:right+1])** | **Sum (total)** | **Shrink Window?** | **min\_length** |
| **1** | 0 | 2 | [2] | 2 | No (2 < 7) | ∞ |
| **2** | 1 | 3 | [2, 3] | 2 + 3 = 5 | No (5 < 7) | ∞ |
| **3** | 2 | 1 | [2, 3, 1] | 5 + 1 = 6 | No (6 < 7) | ∞ |
| **4** | 3 | 2 | [2, 3, 1, 2] | 6 + 2 = 8 | Yes (8 >= 7) | 4 (right - left + 1) |
| **5** | Shrink | left = 1 | [3, 1, 2] | 8 - 2 = 6 | No (6 < 7) | 4 |
| **6** | 4 | 4 | [3, 1, 2, 4] | 6 + 4 = 10 | Yes (10 >= 7) | 4 |
| **7** | Shrink | left = 2 | [1, 2, 4] | 10 - 3 = 7 | Yes (7 >= 7) | 3 |
| **8** | Shrink | left = 3 | [2, 4] | 7 - 1 = 6 | No (6 < 7) | 3 |
| **9** | 5 | 3 | [2, 4, 3] | 6 + 3 = 9 | Yes (9 >= 7) | 3 |
| **10** | Shrink | left = 4 | [4, 3] | 9 - 2 = 7 | Yes (7 >= 7) | 2 |
| **11** | Shrink | left = 5 | [3] | 7 - 4 = 3 | No (3 < 7) | 2 |

//////Minimum Size Subarray Sum (Sliding Window Approach)

//Python

def minSubArrayLen(target, nums):

    left = 0

    total = 0

    min\_length = float('inf')

    for right in range(len(nums)):

        total += nums[right]  # Expand window

        while total >= target:  # Shrink window

            min\_length = min(min\_length, right - left + 1)

            total -= nums[left]

            left += 1

    return min\_length if min\_length != float('inf') else 0

# Test Case

target = 7

nums = [2, 3, 1, 2, 4, 3]

print(minSubArrayLen(target, nums))  # Output: 2

/////Java Implementation

public class MinSizeSubarraySum {

    public static int minSubArrayLen(int target, int[] nums) {

        int left = 0, total = 0, minLength = Integer.MAX\_VALUE;

        for (int right = 0; right < nums.length; right++) {

            total += nums[right];  // Expand window

            while (total >= target) {  // Shrink window

                minLength = Math.min(minLength, right - left + 1);

                total -= nums[left];

                left++;

            }

        }

        return (minLength == Integer.MAX\_VALUE) ? 0 : minLength;

    }

    public static void main(String[] args) {

        int target = 7;

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

        System.out.println(minSubArrayLen(target, nums));  // Output: 2

    }

}

1. **SECTION - BIT MANIPULATION: Single Number**

class Solution {

    public int singleNumber(int[] nums) {

        int result = 0; // Initialize result as 0

        // XOR each element in the array

        for (int num : nums) {

            result ^= num; // XOR logic

        }

        return result; // The single number will remain

    }

    // Example usage

    public static void main(String[] args) {

        Solution solution = new Solution();

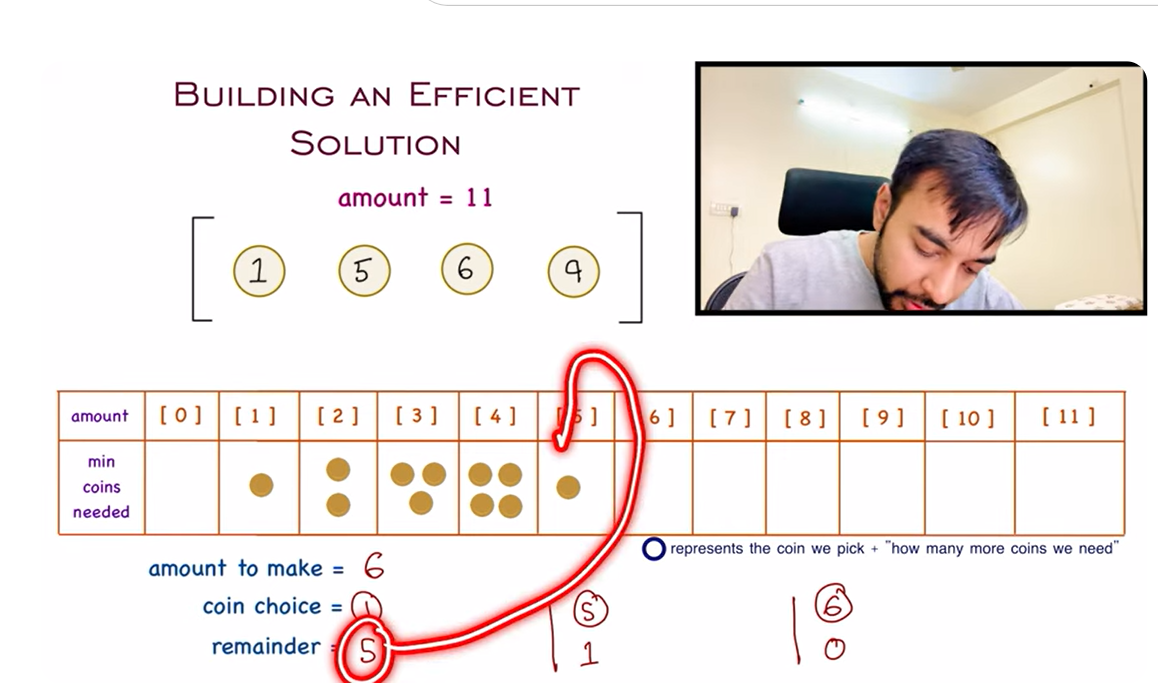
        System.out.println(solution.singleNumber(new int[]{4, 1, 2, 1, 2})); // Output: 4

        System.out.println(solution.singleNumber(new int[]{2, 2, 1}));       // Output: 1

    }

}

1. **SECTION - DYNAMIC PROGRAMMING: Coin Change(Check on the Greedy approach for this)**



import java.util.Arrays;

class Solution {

    public int coinChange(int[] coins, int amount) {

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

        Arrays.fill(dp, amount + 1); // Step 1: Fill DP array with a large value

        dp[0] = 0; // Base case: 0 coins needed for amount 0

        for (int i = 1; i <= amount; i++) { // Step 2: Loop through all amounts

            for (int coin : coins) { // Step 3: Loop through all coins

                if (i >= coin) { // Step 4: Check if we can use the coin

                    dp[i] = Math.min(dp[i], dp[i - coin] + 1); // Step 5: Update DP array

                    // here +1 is added to the number of coins needed to make the amount

                    //+1 for calculating coin itself

                }

            }

        }

        return dp[amount] > amount ? -1 : dp[amount]; // Step 6: Return result

    }

}

**Python**

// In Python

def coinChange(coins, amount):

    dp = [float('inf')] \* (amount + 1)

    dp[0] = 0  # Base case

    for i in range(1, amount + 1):

        for coin in coins:

            if i >= coin:

                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

We iterate over each i from 1 to 11 and check all coin denominations [1, 2, 5].

1. **For i = 1**
   * We can use coin = 1:  
     dp[1] = min(dp[1], dp[1 - 1] + 1) = min(12, 0 + 1) = 1
   * Updated DP:

csharp

CopyEdit

[0, 1, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12]

1. **For i = 2**
   * We can use coin = 1:  
     dp[2] = min(dp[2], dp[2 - 1] + 1) = min(12, 1 + 1) = 2
   * We can use coin = 2:  
     dp[2] = min(dp[2], dp[2 - 2] + 1) = min(2, 0 + 1) = 1
   * Updated DP:

csharp

CopyEdit

[0, 1, 1, 12, 12, 12, 12, 12, 12, 12, 12, 12]

1. **For i = 3**
   * Using 1: dp[3] = min(12, dp[3-1] + 1) = min(12, 1+1) = 2
   * Using 2: dp[3] = min(2, dp[3-2] + 1) = min(2, 1+1) = 2
   * Updated DP:

csharp

CopyEdit

[0, 1, 1, 2, 12, 12, 12, 12, 12, 12, 12, 12]

1. **For i = 4**
   * Using 1: dp[4] = min(12, dp[4-1] + 1) = min(12, 2+1) = 3
   * Using 2: dp[4] = min(3, dp[4-2] + 1) = min(3, 1+1) = 2
   * Updated DP:

csharp

CopyEdit

[0, 1, 1, 2, 2, 12, 12, 12, 12, 12, 12, 12]

1. **For i = 5**
   * Using 1: dp[5] = min(12, dp[5-1] + 1) = min(12, 2+1) = 3
   * Using 2: dp[5] = min(3, dp[5-2] + 1) = min(3, 2+1) = 3
   * Using 5: dp[5] = min(3, dp[5-5] + 1) = min(3, 0+1) = 1
   * Updated DP:

csharp

CopyEdit

[0, 1, 1, 2, 2, 1, 12, 12, 12, 12, 12, 12]

1. **For i = 6 to 10**, similar updates occur.
2. **For i = 11**
   * Using 1: dp[11] = min(12, dp[11-1] + 1) = min(12, 3+1) = 4
   * Using 2: dp[11] = min(4, dp[11-2] + 1) = min(4, 3+1) = 4
   * Using 5: dp[11] = min(4, dp[11-5] + 1) = min(4, 2+1) = 3
   * Updated DP:

csharp

CopyEdit

[0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 2, 3]

### ****Step 3: Return the Answer****

The final dp[11] = 3, so the minimum number of coins needed is **3**.

**Final Answer:** 3

### ****Time & Space Complexity****

* **Time Complexity:** O(n×m)O(n \times m)O(n×m), where n = amount, m = number of coins
* **Space Complexity:** O(n)O(n)O(n) (since we store dp array of size amount + 1)

### ****Final Takeaways****

* We use **Dynamic Programming (Bottom-Up)** to find the minimum number of coins.
* **Key Formula:** dp[i]=min⁡(dp[i],dp[i−coin]+1)dp[i] = \min(dp[i], dp[i - coin] + 1)dp[i]=min(dp[i],dp[i−coin]+1)
* **Edge Cases**:
  + If amount = 0, return 0.
  + If no combination exists, return -1.

# **20 . Climbing Stairs (LeetCode 70) | Full solution with animations | Dynamic Easy | Study Algorithms**