**LAVANYAA A AIML**

1. **Khaden’s Algorithm**

package dsa;

import java.util.\*;

public class MaxSubArray {

public static long findMaxSum(int[] array, int length) {

long maxSum = Integer.MIN\_VALUE;

long currentSum = 0;

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

currentSum += array[i];

maxSum = Math.max(maxSum, currentSum);

if (currentSum < 0) currentSum = 0;

}

if (maxSum < 0) maxSum = 0;

return maxSum;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int length = scanner.nextInt();

int[] array = new int[length];

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

array[i] = scanner.nextInt();

}

long result = findMaxSum(array, length);

System.out.println(result);

scanner.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Maximum Product SubArray**

package dsa;

import java.util.\*;

public class MaxSubarrayProduct {

public static long findMaxSubarrayProduct(int[] nums, int length) {

long forwardProduct = 1, backwardProduct = 1;

long maxProduct = Integer.MIN\_VALUE;

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

if (forwardProduct == 0) forwardProduct = 1;

if (backwardProduct == 0) backwardProduct = 1;

forwardProduct \*= nums[i];

backwardProduct \*= nums[length - i - 1];

maxProduct = Math.max(maxProduct, Math.max(forwardProduct, backwardProduct));

}

return maxProduct;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int length = scanner.nextInt();

int[] nums = new int[length];

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

nums[i] = scanner.nextInt();

}

long result = findMaxSubarrayProduct(nums, length);

System.out.println(result);

scanner.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Search In Rotated Sorted Array**

package dsa;

import java.util.Scanner;

public class SearchRotatedArray {

public static int findTargetIndex(int[] array, int length, int targetValue) {

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

if (array[i] == targetValue)

return i;

}

return -1;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int length = scanner.nextInt();

int[] array = new int[length];

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

array[i] = scanner.nextInt();

}

int targetValue = scanner.nextInt();

int result = findTargetIndex(array, length, targetValue);

System.out.println(result);

scanner.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Container with Most Water**

package dsa;

import java.util.Scanner;

public class MaxContainer {

public static long calculateMaxWater(int[] heights, int length) {

int left = 0, right = length - 1;

long maxArea = 0;

while (left < right) {

maxArea = Math.max(maxArea, (right - left) \* Math.min(heights[left], heights[right]));

if (heights[left] < heights[right])

left++;

else

right--;

}

return maxArea;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int length = scanner.nextInt();

int[] heights = new int[length];

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

heights[i] = scanner.nextInt();

}

long result = calculateMaxWater(heights, length);

System.out.println(result);

scanner.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Trapping rain water**

package dsa;

import java.util.Scanner;

public class RainWaterTrapping {

public static int calculateTrappedWater(int[] heights, int length) {

int totalWater = 0;

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

int leftMax = 0, rightMax = 0;

for (int j = i; j >= 0; j--) {

leftMax = Math.max(leftMax, heights[j]);

}

for (int j = i; j < length; j++) {

rightMax = Math.max(rightMax, heights[j]);

}

totalWater += Math.min(leftMax, rightMax) - heights[i];

}

return totalWater;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int length = scanner.nextInt();

int[] heights = new int[length];

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

heights[i] = scanner.nextInt();

}

int result = calculateTrappedWater(heights, length);

System.out.println(result);

scanner.close();

}

}

TIME COMPLEXITY: O(n2)

SPACE COMPLEXITY: O(1)

1. **Factorial of a large number**

package dsa;

import java.math.BigInteger;

import java.util.Scanner;

public class LargeFactorialCalculator {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int inputNumber = scanner.nextInt();

BigInteger result = BigInteger.ONE;

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

result = result.multiply(BigInteger.valueOf(i));

}

System.out.println(result);

scanner.close();

}

}

TIME COMPLEXITY: O(n⋅log(n))

SPACE COMPLEXITY: O(log(n))

1. **Chocolate Distribution Problem**

package dsa;

import java.util.Arrays;

import java.util.Scanner;

public class CandyDistribution {

public static int findMinimumDifference(int[] sweets, int students) {

int totalPackets = sweets.length;

if (totalPackets < students)

return -1;

Arrays.sort(sweets);

int minDifference = Integer.MAX\_VALUE;

for (int i = 0; i <= totalPackets - students; i++) {

int difference = sweets[i + students - 1] - sweets[i];

minDifference = Math.min(minDifference, difference);

}

return minDifference;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int totalPackets = scanner.nextInt();

int students = scanner.nextInt();

int[] sweets = new int[totalPackets];

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

sweets[i] = scanner.nextInt();

}

int result = findMinimumDifference(sweets, students);

System.out.println(result);

scanner.close();

}

}

TIME COMPLEXITY: O(n log n)

SPACE COMPLEXITY: O(1)

1. **Merge overlapping intervals**

package dsa;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.Scanner;

public class IntervalMerger {

public static int[][] consolidateIntervals(int[][] ranges) {

Arrays.sort(ranges, (a, b) -> Integer.compare(a[0], b[0]));

List<int[]> consolidatedList = new ArrayList<>();

int[] currentRange = ranges[0];

consolidatedList.add(currentRange);

for (int[] range : ranges) {

int currentEnd = currentRange[1];

int nextStart = range[0];

int nextEnd = range[1];

if (nextStart <= currentEnd) {

currentRange[1] = Math.max(currentEnd, nextEnd);

} else {

currentRange = range;

consolidatedList.add(currentRange);

}

}

return consolidatedList.toArray(new int[consolidatedList.size()][]);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int intervalCount = scanner.nextInt();

int[][] ranges = new int[intervalCount][2];

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

ranges[i][0] = scanner.nextInt();

ranges[i][1] = scanner.nextInt();

}

int[][] result = consolidateIntervals(ranges);

for (int[] range : result) {

System.out.println(Arrays.toString(range));

}

scanner.close();

}

}

TIME COMPLEXITY: O(n log n)

SPACE COMPLEXITY: O(1)

1. **Boolean Matrix Question**

package dsa;

import java.util.Scanner;

public class MatrixModification {

public static void updateMatrix(int[][] matrix, int rows, int cols) {

boolean firstRowFlag = false, firstColFlag = false;

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

if (matrix[0][col] == 1)

firstRowFlag = true;

}

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

if (matrix[row][0] == 1)

firstColFlag = true;

}

for (int row = 1; row < rows; row++) {

for (int col = 1; col < cols; col++) {

if (matrix[row][col] == 1) {

matrix[row][0] = 1;

matrix[0][col] = 1;

}

}

}

System.out.println("Matrix after marking rows and columns:");

displayMatrix(matrix, rows, cols);

for (int row = 1; row < rows; row++) {

for (int col = 1; col < cols; col++) {

if (matrix[row][0] == 1 || matrix[0][col] == 1)

matrix[row][col] = 1;

}

}

if (firstRowFlag)

for (int col = 0; col < cols; col++)

matrix[0][col] = 1;

if (firstColFlag)

for (int row = 0; row < rows; row++)

matrix[row][0] = 1;

System.out.println("Modified Matrix:");

displayMatrix(matrix, rows, cols);

}

public static void displayMatrix(int[][] matrix, int rows, int cols) {

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

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

System.out.print(matrix[row][col] + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int rows = scanner.nextInt();

int cols = scanner.nextInt();

int[][] matrix = new int[rows][cols];

System.out.println("Enter matrix elements:");

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

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

matrix[row][col] = scanner.nextInt();

}

}

System.out.println("Original Matrix:");

displayMatrix(matrix, rows, cols);

updateMatrix(matrix, rows, cols);

scanner.close();

}

}

TIME COMPLEXITY: O(m\*n)

SPACE COMPLEXITY: O(1)

1. **1Print matrix in spiral form**

package dsa;

import java.util.Scanner;

public class MatrixSpiralTraversal {

public static void printSpiral(int[][] matrix, int rows, int cols) {

int top = 0, bottom = rows - 1, left = 0, right = cols - 1;

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

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

System.out.print(matrix[top][i] + " ");

}

top++;

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

System.out.print(matrix[i][right] + " ");

}

right--;

if (top <= bottom) {

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

System.out.print(matrix[bottom][i] + " ");

}

bottom--;

}

if (left <= right) {

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

System.out.print(matrix[i][left] + " ");

}

left++;

}

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int rows = scanner.nextInt();

int cols = scanner.nextInt();

int[][] matrix = new int[rows][cols];

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

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

matrix[i][j] = scanner.nextInt();

}

}

printSpiral(matrix, rows, cols);

scanner.close();

}

}

TIME COMPLEXITY: O(n\*m)

SPACE COMPLEXITY: O(1)

1. **Check if parenthesis expression is balanced or not**

package dsa;

import java.util.\*;

public class BracketValidator {

public static String validateBrackets(String input) {

int balanceCounter = 0;

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

if (input.charAt(i) == '(') {

balanceCounter++;

} else if (input.charAt(i) == ')') {

balanceCounter--;

}

if (balanceCounter < 0) {

return "Unbalanced";

}

}

if (balanceCounter == 0) {

return "Balanced";

} else {

return "Unbalanced";

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

String inputString = scanner.nextLine();

System.out.println(validateBrackets(inputString));

scanner.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Check two strings are anagrams of each other**

package dsa;

import java.util.Scanner;

public class StringComparer {

public static boolean checkAnagram(String str1, String str2) {

if (str1.length() != str2.length()) {

return false;

}

int[] letterFrequency = new int[26];

for (int i = 0; i < str1.length(); i++) {

letterFrequency[str1.charAt(i) - 'a']++;

letterFrequency[str2.charAt(i) - 'a']--;

}

for (int count : letterFrequency) {

if (count != 0) {

return false;

}

}

return true;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

String str1 = scanner.nextLine();

String str2 = scanner.nextLine();

if (checkAnagram(str1, str2)) {

System.out.println("True");

} else {

System.out.println("False");

}

scanner.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Longest Palindromic Substring**

package dsa;

import java.util.Scanner;

public class LongestPalindromeSubstring {

public static String findLongestPalindrome(String input) {

if (input == null || input.length() < 1) {

return "";

}

int begin = 0, maxLength = 1;

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

int length1 = extendPalindrome(input, i, i);

int length2 = extendPalindrome(input, i, i + 1);

int length = Math.max(length1, length2);

if (length > maxLength) {

maxLength = length;

begin = i - (length - 1) / 2;

}

}

return input.substring(begin, begin + maxLength);

}

private static int extendPalindrome(String input, int left, int right) {

while (left >= 0 && right < input.length() && input.charAt(left) == input.charAt(right)) {

left--;

right++;

}

return right - left - 1;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

String inputString = scanner.nextLine();

System.out.println(findLongestPalindrome(inputString));

scanner.close();

}

}

TIME COMPLEXITY: O(n2)

SPACE COMPLEXITY: O(1)

1. **Longest common prefix using sorting**

package dsa;

import java.util.\*;

public class LongestCommonPrefix {

public static String findLongestCommonPrefix(String[] words) {

if (words.length == 0) {

return "-1";

}

Arrays.sort(words);

String first = words[0], last = words[words.length - 1];

int index = 0;

while (index < first.length() && index < last.length() && first.charAt(index) == last.charAt(index)) {

index++;

}

return index == 0 ? "-1" : first.substring(0, index);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

String[] words = new String[n];

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

words[i] = scanner.next();

}

System.out.println(findLongestCommonPrefix(words));

scanner.close();

}

}

TIME COMPLEXITY: O(n log n + k)

SPACE COMPLEXITY: O(1)

1. **Delete the middle element of stack**

package dsa;

import java.util.\*;

public class StackRemoveMiddle {

public static void removeMiddleElement(Stack<Integer> stack, int mid) {

if (mid == 0) {

System.out.println("Removing middle element: " + stack.peek());

stack.pop();

return;

}

int topElement = stack.pop();

System.out.println("Removed element: " + topElement);

removeMiddleElement(stack, mid - 1);

stack.push(topElement);

System.out.println("Restored element: " + topElement);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

int size = sc.nextInt();

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

int element = sc.nextInt();

stack.push(element);

}

System.out.println("Original Stack: " + stack);

int middleIndex = size / 2;

removeMiddleElement(stack, middleIndex);

System.out.println("Modified Stack: " + stack);

sc.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(n)

1. **Next greater element in every element in given array**

package dsa;

import java.util.\*;

public class NextLargerElement {

public static void findNextLargerElement(int[] nums) {

int len = nums.length;

int[] nextLarger = new int[len];

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

for (int i = len - 1; i >= 0; i--) {

while (!stack.isEmpty() && stack.peek() <= nums[i]) {

stack.pop();

}

nextLarger[i] = stack.isEmpty() ? -1 : stack.peek();

stack.push(nums[i]);

}

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

System.out.println(nums[i] + " -> " + nextLarger[i]);

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int size = scanner.nextInt();

int[] numbers = new int[size];

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

numbers[i] = scanner.nextInt();

}

findNextLargerElement(numbers);

scanner.close();

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(n)

1. **Print Right view of Binary Tree**

package dsa;

import java.util.\*;

class Node {

int data;

Node leftChild;

Node rightChild;

Node(int data) {

this.data = data;

this.leftChild = null;

this.rightChild = null;

}

}

public class RightSideView {

public static void traverse(Node node, int depth, List<Integer> result) {

if (node == null) return;

if (result.size() == depth) {

result.add(node.data);

}

traverse(node.rightChild, depth + 1, result);

traverse(node.leftChild, depth + 1, result);

}

public static List<Integer> getRightView(Node root) {

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

traverse(root, 0, result);

return result;

}

public static void main(String[] args) {

Node root = new Node(25);

root.leftChild = new Node(20);

root.rightChild = new Node(36);

root.leftChild.leftChild = new Node(10);

root.leftChild.leftChild.leftChild = new Node(5);

root.leftChild.rightChild = new Node(22);

root.leftChild.leftChild.rightChild = new Node(12);

root.rightChild.leftChild = new Node(30);

root.rightChild.leftChild.leftChild = new Node(28);

root.rightChild.rightChild = new Node(40);

System.out.println(getRightView(root));

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(n)

1. **Maximum Depth or Height of Binary Tree**

package dsa;

import java.util.\*;

class Node {

int value;

Node leftChild;

Node rightChild;

Node(int value) {

this.value = value;

this.leftChild = null;

this.rightChild = null;

}

}

public class TreeHeight {

public static int calculateDepth(Node node) {

if (node == null) return 0;

int leftDepth = calculateDepth(node.leftChild);

int rightDepth = calculateDepth(node.rightChild);

return Math.max(leftDepth, rightDepth) + 1;

}

public static void main(String[] args) {

Node rootNode = new Node(25);

rootNode.leftChild = new Node(20);

rootNode.rightChild = new Node(36);

rootNode.leftChild.leftChild = new Node(10);

rootNode.leftChild.leftChild.leftChild = new Node(5);

rootNode.leftChild.rightChild = new Node(22);

rootNode.leftChild.leftChild.rightChild = new Node(12);

rootNode.rightChild.leftChild = new Node(30);

rootNode.rightChild.leftChild.leftChild = new Node(28);

rootNode.rightChild.rightChild = new Node(40);

System.out.println(calculateDepth(rootNode));

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(n)