**1.Kadanes Algorithm**

import java.util.Scanner;

public class KadaneAlgorithm {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of elements in the array: ");

int n = scanner.nextInt();

int[] arr = new int[n];

System.out.println("Enter the elements of the array:");

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

arr[i] = scanner.nextInt();

}

int result = kadane(arr);

System.out.println("Maximum sum of a contiguous subarray is: " + result);

scanner.close();}

public int maxSubArray(int[] nums) {

int curr\_max = nums[0];

int global\_max = nums[0];

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

curr\_max = Math.max(nums[i], curr\_max + nums[i]);

global\_max = Math.max(curr\_max, global\_max);

}return global\_max;

}

**TIME COMPLEXITY:O(n)**

**2.MaxProduct of subarray**

import java.util.Scanner;

public class MaximumProductSubarray {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of elements in the array: ");

int n = scanner.nextInt();

int[] arr = new int[n];

System.out.println("Enter the elements of the array:");

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

arr[i] = scanner.nextInt();

}

System.out.println("Maximum product of subarray is: " + maxProduct(arr));

scanner.close();

}

public static int maxProduct(int[] arr) {

if (arr.length == 0) return 0;

int max\_so\_far = arr[0];

int min\_so\_far = arr[0];

int result = arr[0];

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

if (arr[i] < 0) {

int temp = max\_so\_far;

max\_so\_far = min\_so\_far;

min\_so\_far = temp;

}

max\_so\_far = Math.max(arr[i], max\_so\_far \* arr[i]);

min\_so\_far = Math.min(arr[i], min\_so\_far \* arr[i]);

result = Math.max(result, max\_so\_far);

}

return result;

}

}  
**TIME COMPLEXITY:O(n)**

**3. Search in a sorted and rotated Array**

public class RotatedArraySearch {

public static int searchInRotatedArray(int[] arr, int key) {

int start = 0;

int end = arr.length - 1;

while (start <= end) {

int mid = (start + end) / 2;

if (arr[mid] == key) {

return mid;

}

if (arr[start] <= arr[mid]) {

if (arr[start] <= key && key < arr[mid]) {

end = mid - 1;

} else {

start = mid + 1;

}

}

else {

if (arr[mid] < key && key <= arr[end]) {

start = mid + 1;

} else {

end = mid - 1;

}

}

}

return -1;

}

}

**TIME COMPLEXITY:O(logn)**

**4.Container with Water**

class Solution {

public int maxArea(int[] height) {

int left = 0;

int right = height.length - 1;

int maxArea = 0;

while (left < right) {

int currentArea = Math.min(height[left], height[right]) \* (right - left);

maxArea = Math.max(maxArea, currentArea);

if (height[left] < height[right]) {

left++;

}

else {

right--;

}

}

return maxArea;

}

}

**Time Complexity**: O(n)

**5. Find the Factorial of a large number**

import java.math.BigInteger;

java.util.Scanner;

public class LargeFact{

public static void main(String[] args){

Scanner scanner = new Scanner(System.in);

System.out.println("Enter the value:");

int n = scanner.nextInt();

BigInteger factorial = BigInteger.ONE;

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

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

}

System.out.println("Factorial of"+n+"is");

System.out.println(factorial);

scanner.close();

}

**Time Complexity**: O(n⋅logn⋅loglogn)

**6.Trapping water:**

class Solution {

public int trap(int[] height) {

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

int[] left = new int[height.length];

int[] right = new int[height.length];

int max = -1;

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

max = Math.max(max, height[i]);

left[i] = max;

}

max = -1;

for (int i = height.length - 1; i >= 0; i--) {

max = Math.max(max, height[i]);

right[i] = max;

}

int total = 0;

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

total += Math.min(left[i], right[i]) - height[i];

}

return total;

}

}  
**TIME COMPLEXITY:O(n)**

**7. Chocolate Distribution Problem:**

import java.util.Arrays;

import java.util.Scanner;

public class ChocolateDistribution {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of packets: ");

int n = scanner.nextInt();

int[] arr = new int[n];

System.out.println("Enter the number of chocolates in each packet:");

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

arr[i] = scanner.nextInt();

}

System.out.print("Enter the number of students: ");

int m = scanner.nextInt();

int result = findMinDifference(arr, n, m);

System.out.println("The minimum difference is: " + result);

scanner.close();

}

public static int findMinDifference(int[] arr, int n, int m) {

if (m == 0 || n == 0) return 0;

if (n < m) return -1;

Arrays.sort(arr);

int minDiff = Integer.MAX\_VALUE;

for (int i = 0; i + m - 1 < n; i++) {

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

if (diff < minDiff) minDiff = diff;

}

return minDiff;

}

}

**TIME COMPLEXITY: O(nlogn)**

**8. Merge Overlapping Intervals:**

import java.util.Arrays;

class GfG {

// Merge overlapping intervals in-place. We return

// modified size of the array arr.

static int mergeOverlap(int[][] arr) {

// Sort intervals based on start values

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

// Index of the last merged

int resIdx = 0;

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

// If current interval overlaps with the

// last merged interval

if (arr[resIdx][1] >= arr[i][0])

arr[resIdx][1] = Math.max(arr[resIdx][1], arr[i][1]);

// Move to the next interval

else {

resIdx++;

arr[resIdx] = arr[i];

}

}

// Returns size of the merged intervals

return (resIdx + 1);

}

public static void main(String[] args) {

int[][] arr = {{7, 8}, {1, 5}, {2, 4}, {4, 6}};

// Get the new size of the array after merging

int newSize = mergeOverlap(arr);

// Print the merged intervals based on the new size

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

System.out.println(arr[i][0] + " " + arr[i][1]);

}

}  
**Time Complexity**: O(nlogn)

**9. Boolean Matrix Question:**  
class Solution {

public void booleanMatrix(int[][] mat) {

int rows = mat.length;

int cols = mat[0].length;

boolean[] rowFlag = new boolean[rows];

boolean[] colFlag = new boolean[cols];

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

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

if (mat[i][j] == 1) {

rowFlag[i] = true;

colFlag[j] = true;

}

}

}

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

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

if (rowFlag[i] || colFlag[j]) {

mat[i][j] = 1;

}

}

}

}

}

**Time Complexity**: O(M×N)

**10. Print a Matrix in Spiral Form:**

import java.util.\*;

class Solution {

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

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

if (matrix.length == 0) return result;

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

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

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

for (int i = left; i <= right; i++) result.add(matrix[top][i]);

top++;

for (int i = top; i <= bottom; i++) result.add(matrix[i][right]);

right--;

if (top <= bottom) {

for (int i = right; i >= left; i--) result.add(matrix[bottom][i]);

bottom--;

}

if (left <= right) {

for (int i = bottom; i >= top; i--) result.add(matrix[i][left]);

left++;

}

}

return result;

}

}

**Time Complexity**:O(M×N)  
**13. Check if Parentheses Expression is Balanced:**  
import java.util.\*;

class Solution {

public boolean isBalanced(String str) {

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

for (char ch : str.toCharArray()) {

if (ch == '(') {

stack.push(ch);

} else if (ch == ')') {

if (stack.isEmpty()) return false;

stack.pop();

}

}

return stack.isEmpty();

}

}  
**Time Complexity**: O(N)  
**14. Check if Two Strings are Anagrams:**import java.util.Arrays;

class Solution {

public boolean areAnagrams(String s1, String s2) {

if (s1.length() != s2.length()) return false;

char[] arr1 = s1.toCharArray();

char[] arr2 = s2.toCharArray();

Arrays.sort(arr1);

Arrays.sort(arr2);

return Arrays.equals(arr1, arr2);

}

}  
**Time Complexity**:O(nlogn)

**15.Longest Palindromic Substring:**

public class LongestPalindromicSubstring {

public static String longestPalindrome(String str) {

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

return "";

}

int start = 0;

int end = 0;

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

int len1 = expandAroundCenter(str, i, i);

int len2 = expandAroundCenter(str, i, i + 1);

int len = Math.max(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return str.substring(start, end + 1);

}

private static int expandAroundCenter(String str, int left, int right) {

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

left--;

right++;

}

return right - left - 1;

}

**16. Longest Common Prefix using Sorting:**

import java.util.Arrays;

class Solution {

public String longestCommonPrefix(String[] arr) {

if (arr == null || arr.length == 0) return "-1";

Arrays.sort(arr);

String first = arr[0];

String last = arr[arr.length - 1];

int i = 0;

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

i++;

}

String prefix = first.substring(0, i);

return prefix.isEmpty() ? "-1" : prefix;

}

}  
**Time Complexity**: O(nlogn+m)  
**17. Delete Middle Element of a Stack**

import java.util.Stack;

class Solution {

public void deleteMiddle(Stack<Integer> stack) {

int middleIndex = stack.size() / 2;

deleteMiddleHelper(stack, middleIndex);

}

private void deleteMiddleHelper(Stack<Integer> stack, int k) {

if (k == 0) {

stack.pop();

return;

}

int top = stack.pop();

deleteMiddleHelper(stack, k - 1);

stack.push(top);

}

}

**Time Complexity**: O(n)

**18. Next Greater Element (NGE) for Every Element in Given Array**

import java.util.Stack;

import java.util.HashMap;

class Solution {

public int[] nextGreaterElement(int[] arr) {

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

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

for (int i = arr.length - 1; i >= 0; i--) {

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

stack.pop();

}

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

stack.push(arr[i]);

}

return result;

}

}

**Time Complexity**: O(n)

**19. Print Right View of a Binary Tree**

import java.util.\*;

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int val) {

this.val = val;

}

}

class Solution {

public List<Integer> rightView(TreeNode root) {

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

if (root == null) return rightView;

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

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

TreeNode node = queue.poll();

if (i == levelSize - 1) {

rightView.add(node.val);

}

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

}

return rightView;

}

}

**Time Complexity**: O(n)  
**20. Maximum Depth or Height of Binary Tree**  
class TreeNode {

int val;

TreeNode left, right;

TreeNode(int val) {

this.val = val;

this.left = null;

this.right = null;

}

}

class Solution {

public int maxDepth(TreeNode root) {

if (root == null) return 0; // Base case: an empty tree has depth 0

int leftDepth = maxDepth(root.left);

int rightDepth = maxDepth(root.right);

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

}

}  
**Time Complexity**: O(n)