1. **Maximum Subarray Sum – Kadane‟s Algorithm**: Given an array arr[], the task is to find the subarray that has the maximum sum and return its sum.

Input: arr[] = {2, 3, -8, 7, -1, 2, 3} Output: 11 Explanation: The subarray {7, -1, 2, 3} has the largest sum 11.

Input: arr[] = {-2, -4} Output: –2 Explanation: The subarray {-2} has the largest sum -2.

Input: arr[] = {5, 4, 1, 7, 8} Output: 25 Explanation: The subarray {5, 4, 1, 7, 8} has the largest sum 25.

**package** JavaPackage;

**public** **class** SubArray {

**public** **static** **int** SubArray(**int**[] arr) {

**int** maxStart = arr[0];

**int** maxEnding = arr[0];

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

maxEnding = Math.*max*(arr[i],maxEnding+arr[i]);

maxStart = Math.*max*(maxStart,maxEnding);

}

**return** maxStart;

}

**public** **static** **void** main(String[] args) {

**int**[] arr1 = {2,3,-8,7,-1,2,3};

System.***out***.println("SubArray:"+ *SubArray*(arr1));

**int**[] arr2 = {-2,-4};

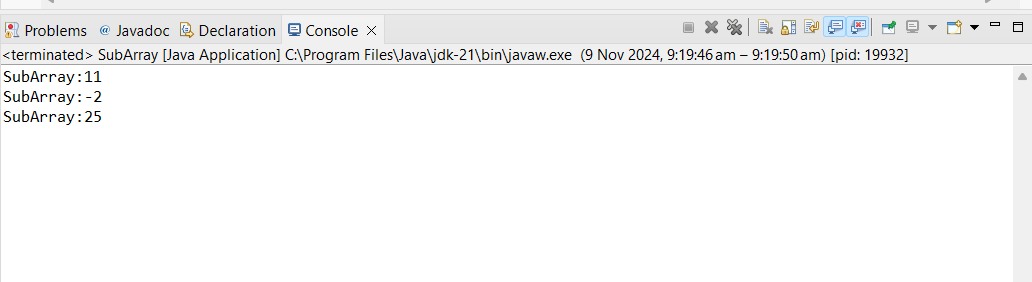
System.***out***.println("SubArray:"+ *SubArray*(arr2));

**int**[] arr3 = {5,4,1,7,8};

System.***out***.println("SubArray:"+ *SubArray*(arr3));

}

}



**Time Complexity**:O(n)

1. **Maximum Product Subarray** Given an integer array, the task is to find the maximum product of any subarray.

Input: arr[] = {-2, 6, -3, -10, 0, 2} Output: 180 Explanation: The subarray with maximum product is {6, -3, -10} with product = 6 \* (-3) \* (-10) = 180

Input: arr[] = {-1, -3, -10, 0, 60} Output: 60 Explanation: The subarray with maximum product is {60}.

**package** JavaPackage;

**public** **class** SubArrayProduct {

**public** **static** **int** Product(**int**[] arr) {

**int** max = arr[0];

**int** min = arr[0];

**int** result = arr[0];

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

**if**(arr[i]<0) {

**int** t = max;

max = min;

min = t;

}

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

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

result = Math.*max*(result,max);

}

**return** result;

}

**public** **static** **void** main(String[] args) {

**int**[] arr1 = {-2, 6, -3, -10, 0, 2};

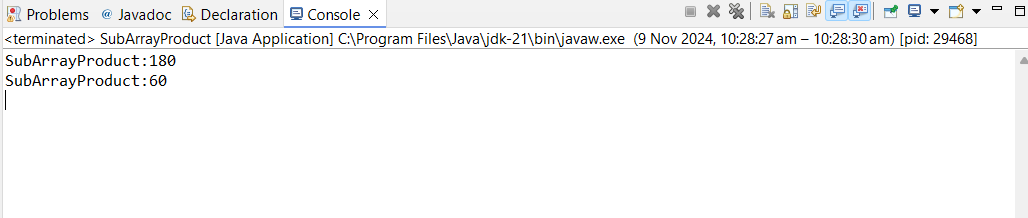
System.***out***.println("SubArrayProduct:"+*Product*(arr1));

**int**[] arr2 = {-1, -3, -10, 0, 60};

System.***out***.println("SubArrayProduct:"+*Product*(arr2));

}

}



**Time Complexity:**O(n)

1. **Search in a sorted and rotated Array** Given a sorted and rotated array arr[] of n distinct elements, the task is to find the index of given key in the array. If the key is not present in the array, return -1.

Input : arr[] = {4, 5, 6, 7, 0, 1, 2}, key = 0 Output : 4

Input : arr[] = { 4, 5, 6, 7, 0, 1, 2 }, key = 3 Output : -1

Input : arr[] = {50, 10, 20, 30, 40}, key = 10 Output : 1**package** JavaPackage;

**public** **class** BinarySearch {

**public** **static** **int** Search(**int**[]arr,**int** k) {

**int** low = 0;

**int** high = arr.length-1;

**while** (low<=high) {

**int** mid = (low+high)/2;

**if**(arr[mid] == k) {

**return** mid;

}

**if**(arr[low]<= arr[mid]) {

**if**(arr[low] <= k && k<arr[mid]) {

high = mid -1;

}

**else** {

low = mid + 1;

}

}**else** {

**if** (arr[mid] < k && k <= arr[high]) {

low = mid + 1;

}**else** {

high = mid -1;

}

}

}

**return** -1;

}

**public** **static** **void** main(String[] args) {

**int**[] arr1 = {4, 5, 6, 7, 0, 1, 2};

**int** k1 = 0;

System.***out***.println( *Search*(arr1, k1));

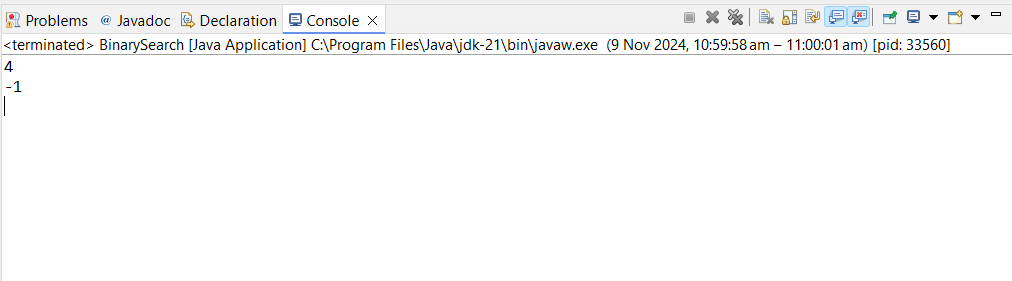
**int**[] arr2 = { 4, 5, 6, 7, 0, 1, 2 };

**int** k2 = 3;

System.***out***.println( *Search*(arr2, k2));

}

}



**Time Complexity:**O(logn)

1. **Container with Most Water**

Input: arr = [1, 5, 4, 3] Output: 6 Explanation: 5 and 3 are distance 2 apart. So the size of the base = 2. Height of container = min(5, 3) = 3. So total area = 3 \* 2 = 6

Input: arr = [3, 1, 2, 4, 5] Output: 12 Explanation: 5 and 3 are distance 4 apart. So the size of the base = 4. Height of container = min(5, 3) = 3. So total area = 4 \* 3 = 12

**package** JavaPackage;

**public** **class** MostWater {

**public** **static** **void** main(String[] args) {

**int**[] arr1 = {1, 5, 4, 3};

System.***out***.println(*maxArea*(arr1));

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

System.***out***.println(*maxArea*(arr2));

}

**public** **static** **int** maxArea(**int**[] height) {

**int** left = 0;

**int** right = height.length - 1;

**int** maxArea = 0;

**while** (left < right) {

**int** width = right - left;

**int** minHeight = Math.*min*(height[left], height[right]);

**int** currentArea = minHeight \* width;

maxArea = Math.*max*(maxArea, currentArea);

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

left++;

} **else** {

right--;

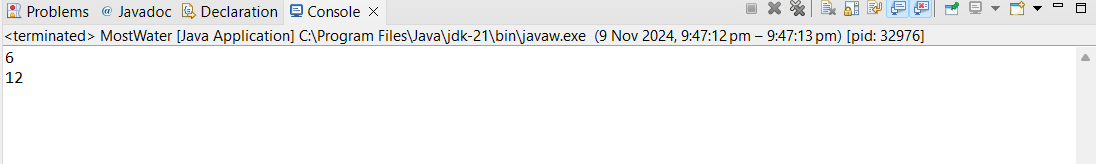
}

}

**return** maxArea;

}

}



**Time Complexity**:O(n)

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

Input: 100 Output: 933262154439441526816992388562667004907159682643816214685929638952175999932299 156089414639761565182862536979208272237582511852109168640000000000000000000000 00

Input: 50 Output:

30414093201713378043612608166064768844377641568960512000000000000

**package** JavaPackage;

**import** java.math.BigInteger;

**public** **class** Factorial {

**public** **static** **void** main(String[] args) {

**int** n = 100;

BigInteger fact = **new** BigInteger("1");

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

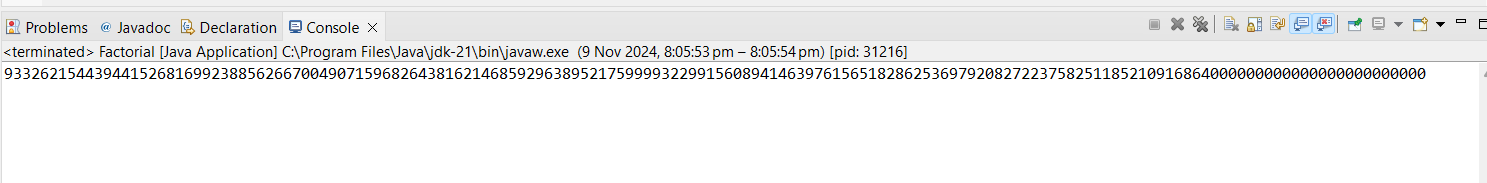
fact = fact.multiply(BigInteger.*valueOf*(i));

}

System.***out***.println(fact);

}

}



**Time Complexity**:O(n)

1. **Trapping Rainwater Problem states** that given an array of n non-negative integers arr[] representing an elevation map where the width of each bar is 1, compute how much water it can trap after rain.

Input: arr[] = {3, 0, 1, 0, 4, 0, 2} Output: 10 Explanation: The expected rainwater to be trapped is shown in the above image.

Input: arr[] = {3, 0, 2, 0, 4} Output: 7 Explanation: We trap 0 + 3 + 1 + 3 + 0 = 7 units.

Input: arr[] = {1, 2, 3, 4} Output: 0 Explanation : We cannot trap water as there is no height bound on both sides

Input: arr[] = {10, 9, 0, 5} Output: 5 Explanation : We trap 0 + 0 + 5 + 0 = 5

**package** JavaPackage;

**public** **class** Rainwater {

**public** **static** **void** main(String[] args) {

**int**[] arr1 = {3, 0, 1, 0, 4, 0, 2};

System.***out***.println(*trap*(arr1));

**int**[] arr2 = {3, 0, 2, 0, 4};

System.***out***.println(*trap*(arr2));

**int**[] arr3 = {1, 2, 3, 4};

System.***out***.println(*trap*(arr3));

**int**[] arr4 = {10, 9, 0, 5};

System.***out***.println(*trap*(arr4));

}

**public** **static** **int** trap(**int**[] height) {

**if** (height == **null** || height.length == 0) {

**return** 0;

}

**int** left = 0;

**int** right = height.length - 1;

**int** leftMax = 0;

**int** rightMax = 0;

**int** totalWater = 0;

**while** (left < right) {

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

**if** (height[left] >= leftMax) {

leftMax = height[left];

} **else** {

totalWater += leftMax - height[left];

}

left++;

} **else** {

**if** (height[right] >= rightMax) {

rightMax = height[right];

} **else** {

totalWater += rightMax - height[right];

}

right--;

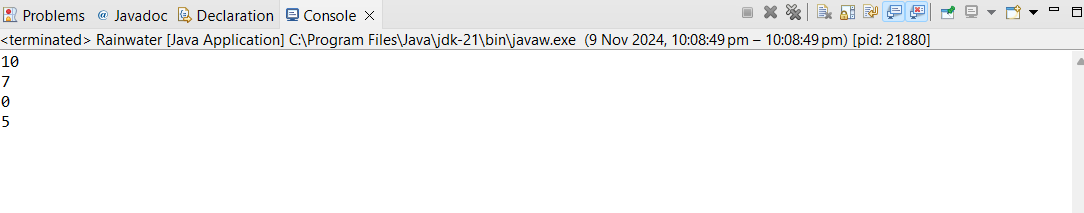
}

}

**return** totalWater;

}

}



**Time Complexity**:O(n)

1. **Chocolate Distribution Problem** Given an array arr[] of n integers where arr[i] represents the number of chocolates in ith packet. Each packet can have a variable number of chocolates. There are m students, the task is to distribute chocolate packets such that: Each student gets exactly one packet. The difference between the maximum and minimum number of chocolates in the packets given to the students is minimized.

Input: arr[] = {7, 3, 2, 4, 9, 12, 56}, m = 3 Output: 2 Explanation: If we distribute chocolate packets {3, 2, 4}, we will get the minimum difference, that is 2.

Input: arr[] = {7, 3, 2, 4, 9, 12, 56}, m = 5 Output: 7 Explanation: If we distribute chocolate packets {3, 2, 4, 9, 7}, we will get the minimum difference, that is 9 – 2 = 7

**package** JavaPackage;

**import** java.util.Arrays;

**public** **class** ChocolateDistribution {

**public** **static** **void** main(String[] args) {

**int**[] arr1 = {7, 3, 2, 4, 9, 12, 56};

**int** m1 = 3;

System.***out***.println(*minDifference*(arr1, m1));

**int**[] arr2 = {7, 3, 2, 4, 9, 12, 56};

**int** m2 = 5;

System.***out***.println(*minDifference*(arr2, m2));

}

**public** **static** **int** minDifference(**int**[] arr, **int** m) {

**int** n = arr.length;

**if** (n == 0 || m == 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];

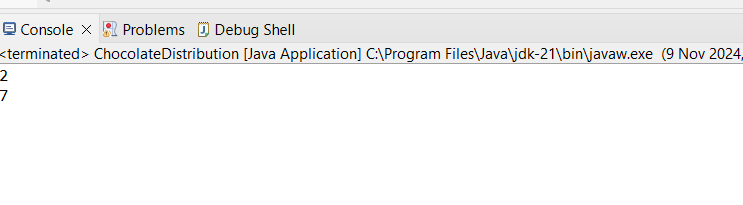
minDiff = Math.*min*(minDiff, diff);

}

**return** minDiff;

}

}



**Time Complexity**:O(n)

1. **Merge Overlapping Intervals** Given an array of time intervals where arr[i] = [starti, endi], the task is to merge all the overlapping intervals into one and output the result which should have only mutually exclusive intervals.

Input: arr[] = [[1, 3], [2, 4], [6, 8], [9, 10]] Output: [[1, 4], [6, 8], [9, 10]] Explanation: In the given intervals, we have only two overlapping intervals [1, 3] and [2, 4]. Therefore, we will merge these two and return [[1, 4}], [6, 8], [9, 10]].

Input: arr[] = [[7, 8], [1, 5], [2, 4], [4, 6]] Output: [[1, 6], [7, 8]] Explanation: We will merge the overlapping intervals [[1, 5], [2, 4], [4, 6]] into a single interval [1, 6].

**package** JavaPackage;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** MergeIntervals {

**public** **static** **void** main(String[] args) {

**int**[][] arr1 = {{1, 3}, {2, 4}, {6, 8}, {9, 10}};

System.***out***.println(*mergeIntervals*(arr1));

**int**[][] arr2 = {{7, 8}, {1, 5}, {2, 4}, {4, 6}};

System.***out***.println(*mergeIntervals*(arr2));

}

**public** **static** List<List<Integer>> mergeIntervals(**int**[][] intervals) {

**if** (intervals.length == 0) {

**return** **new** ArrayList<>();

}

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

List<List<Integer>> merged = **new** ArrayList<>();

**int**[] currentInterval = intervals[0];

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

**if** (currentInterval[1] >= intervals[i][0]) {

currentInterval[1] = Math.*max*(currentInterval[1], intervals[i][1]);

} **else** {

merged.add(Arrays.*asList*(currentInterval[0], currentInterval[1]));

currentInterval = intervals[i];

}

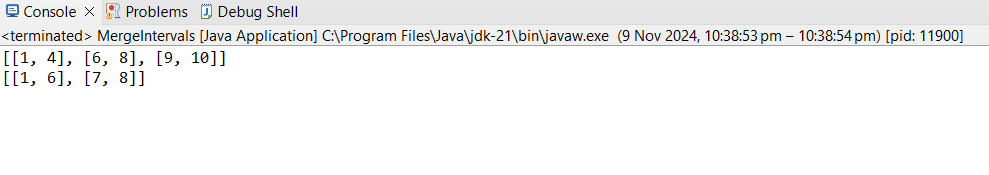
}

merged.add(Arrays.*asList*(currentInterval[0], currentInterval[1]));

**return** merged;

}

}



**Time Complexity**:O(logn)

1. **A Boolean Matrix** Question Given a boolean matrix mat[M][N] of size M X N, modify it such that if a matrix cell mat[i][j] is 1 (or true) then make all the cells of ith row and jth column as 1.

Input: {{1, 0}, {0, 0}} Output: {{1, 1} {1, 0}}

Input: {{0, 0, 0}, {0, 0, 1}} Output: {{0, 0, 1}, {1, 1, 1}}

Input: {{1, 0, 0, 1}, {0, 0, 1, 0}, {0, 0, 0, 0}} Output: {{1, 1, 1, 1}, {1, 1, 1, 1}, {1, 0, 1, 1}}

**package** JavaPackage;

**import** java.util.Arrays;

**public** **class** BooleanMatrix {

**public** **static** **void** main(String[] args) {

**int**[][] mat1 = {{1, 0}, {0, 0}};

*modifyMatrix*(mat1);

System.***out***.println(Arrays.*deepToString*(mat1));

**int**[][] mat2 = {{0, 0, 0}, {0, 0, 1}};

*modifyMatrix*(mat2);

System.***out***.println(Arrays.*deepToString*(mat2));

**int**[][] mat3 = {{1, 0, 0, 1}, {0, 0, 1, 0}, {0, 0, 0, 0}};

*modifyMatrix*(mat3);

System.***out***.println(Arrays.*deepToString*(mat3));

}

**public** **static** **void** modifyMatrix(**int**[][] mat) {

**int** M = mat.length;

**int** N = mat[0].length;

**boolean**[] row = **new** **boolean**[M];

**boolean**[] col = **new** **boolean**[N];

**for** (**int** i = 0; i < M; i++) {

**for** (**int** j = 0; j < N; j++) {

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

row[i] = **true**;

col[j] = **true**;

}

}

}

**for** (**int** i = 0; i < M; i++) {

**for** (**int** j = 0; j < N; j++) {

**if** (row[i] || col[j]) {

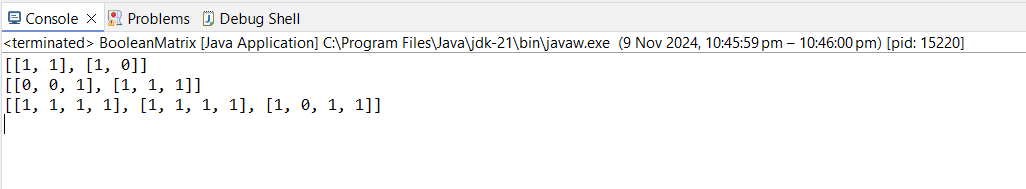
mat[i][j] = 1;

}

}

}

}

}

**Time Complexity**:O(MN)

1. **Print a given matrix in spira**l form Given an m x n matrix, the task is to print all elements of the matrix in spiral form.

Input: matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16 }} Output: 1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

Input: matrix = { {1, 2, 3, 4, 5, 6}, {7, 8, 9, 10, 11, 12}, {13, 14, 15, 16, 17, 18}} Output: 1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11 Explanation: The output is matrix in spiral format.

**package** JavaPackage;

**public** **class** SpiralMatrix {

**public** **static** **void** main(String[] args) {

**int**[][] matrix1 = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12},

{13, 14, 15, 16}

};

*printSpiral*(matrix1);

**int**[][] matrix2 = {

{1, 2, 3, 4, 5, 6},

{7, 8, 9, 10, 11, 12},

{13, 14, 15, 16, 17, 18}

};

*printSpiral*(matrix2);

}

**public** **static** **void** printSpiral(**int**[][] matrix) {

**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++) {

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] + " ");

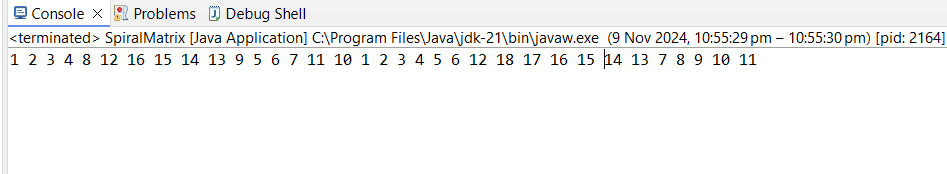
}

left++;

}

}

}

}

**Time Complexity**:O(MN)

**13.** Check if given **Parentheses expression** is balanced or not Given a string str of length N, consisting of „(„ and „)„ only, the task is to check whether it is balanced or not.

Input: str = “((()))()()” Output: Balanced

Input: str = “())((())” Output: Not Balanced

**package** JavaPackage;

**public** **class** BalancedParantheses {

**public** **static** **void** main(String[] args) {

String str1 = "((()))()()";

System.***out***.println(*isBalanced*(str1) ? "Balanced" : "Not Balanced");

String str2 = "())((())";

System.***out***.println(*isBalanced*(str2) ? "Balanced" : "Not Balanced");

}

**public** **static** **boolean** isBalanced(String str) {

**int** balance = 0;

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

**if** (ch == '(') {

balance++;

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

balance--;

}

**if** (balance < 0) {

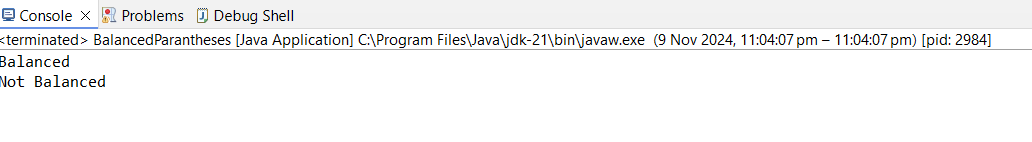
**return** **false**;

}

}

**return** balance == 0;

}

}

**Time Complexity**:O(n)

14. Check if two Strings are Anagrams of each other Given two strings s1 and s2 consisting of lowercase characters, the task is to check whether the two given strings are anagrams of each other or not. An anagram of a string is another string that contains the same characters, only the order of characters can be different.

Input: s1 = “geeks” s2 = “kseeg” Output: true Explanation: Both the string have same characters with same frequency. So, they are anagrams.

Input: s1 = “allergy” s2 = “allergic” Output: false Explanation: Characters in both the strings are not same. s1 has extra character „y‟ and s2 has extra characters „i‟ and „c‟, so they are not anagrams. Input: s1 = “g”, s2 = “g” Output: true Explanation: Characters in both the strings are same, so they are anagrams

**package** JavaPackage;

**import** java.util.Arrays;

**public** **class** AnagramChecker {

**public** **static** **void** main(String[] args) {

String s1 = "geeks";

String s2 = "kseeg";

System.***out***.println(*areAnagrams*(s1, s2));

String s3 = "allergy";

String s4 = "allergic";

System.***out***.println(*areAnagrams*(s3, s4));

String s5 = "g";

String s6 = "g";

System.***out***.println(*areAnagrams*(s5, s6));

}

**public** **static** **boolean** areAnagrams(String s1, String s2) {

**if** (s1.length() != s2.length()) {

**return** **false**;

}

**char**[] arr1 = s1.toCharArray();

**char**[] arr2 = s2.toCharArray();

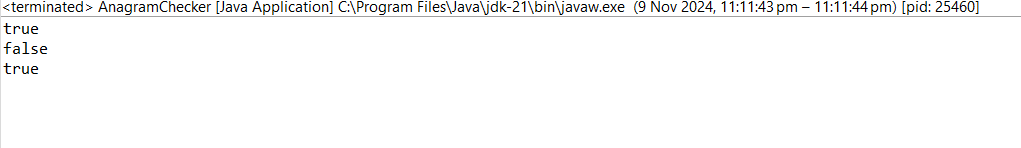
Arrays.*sort*(arr1);

Arrays.*sort*(arr2);

**return** Arrays.*equals*(arr1, arr2);

}

}



**Time Complexity:O(N logN)**

15. **Longest Palindromic Substring** Given a string str, the task is to find the longest substring which is a palindrome. If there are multiple answers, then return the first appearing substring.

Input: str = “forgeeksskeegfor” Output: “geeksskeeg” Explanation: There are several possible palindromic substrings like “kssk”, “ss”, “eeksskee” etc. But the substring “geeksskeeg” is the longest among all.

Input: str = “Geeks” Output: “ee” Input: str = “abc” Output: “a” Input: str = “” Output: “”

**package** JavaPackage;

**public** **class** LongestPalindrom {

**public** **static** **void** main(String[] args) {

String str1 = "forgeeksskeegfor";

System.***out***.println(*longestPalindromicSubstring*(str1));

String str2 = "Geeks";

System.***out***.println(*longestPalindromicSubstring*(str2));

String str3 = "abc";

System.***out***.println(*longestPalindromicSubstring*(str3));

String str4 = "";

System.***out***.println(*longestPalindromicSubstring*(str4));

}

**public** **static** String longestPalindromicSubstring(String str) {

**if** (str == **null** || str.length() == 0) {

**return** "";

}

**int** start = 0, maxLength = 1;

**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 > maxLength) {

maxLength = len;

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

}

}

**return** str.substring(start, start + maxLength);

}

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

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

left--;

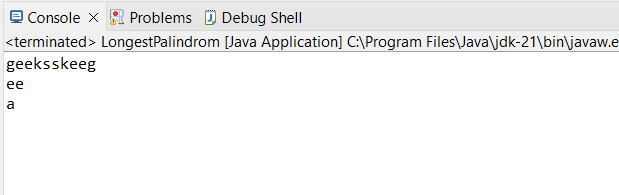
right++;

}

**return** right - left - 1;

}

}



**Time Complexity**:O(N^2)

**16. Longest Common Prefix** using Sorting Given an array of strings arr[]. The task is to return the longest common prefix among each and every strings present in the array. If there‟s no prefix common in all the strings, return “-1”.

Input: arr[] = [“geeksforgeeks”, “geeks”, “geek”, “geezer”] Output: gee Explanation: “gee” is the longest common prefix in all the given strings.

Input: arr[] = [“hello”, “world”] Output: -1 Explanation: There‟s no common prefix in the given strings.

**package** JavaPackage;

**import** java.util.Arrays;

**public** **class** LongestCommonPrefix {

**public** **static** **void** main(String[] args) {

String[] arr1 = {"geeksforgeeks", "geeks", "geek", "geezer"};

System.***out***.println(*longestCommonPrefix*(arr1));

String[] arr2 = {"hello", "world"};

System.***out***.println(*longestCommonPrefix*(arr2));

}

**public** **static** 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** minLength = Math.*min*(first.length(), last.length());

StringBuilder commonPrefix = **new** StringBuilder();

**for** (**int** i = 0; i < minLength; i++) {

**if** (first.charAt(i) == last.charAt(i)) {

commonPrefix.append(first.charAt(i));

} **else** {

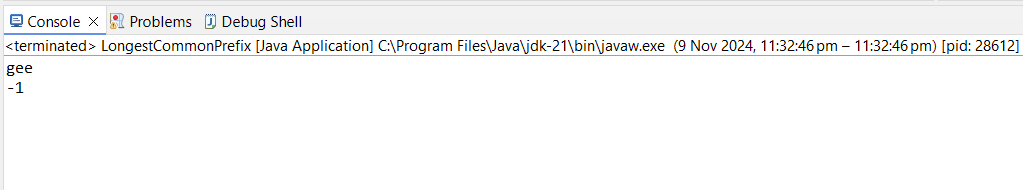
**break**;

}

}

**return** commonPrefix.length() > 0 ? commonPrefix.toString() : "-1";

}

}

**Time Complexity**:O(N logN)

**17. Delete middle element** of a stack Given a stack with push(), pop(), and empty() operations, The task is to delete the middle element of it without using any additional data structure.

Input : Stack[] = [1, 2, 3, 4, 5] Output : Stack[] = [1, 2, 4, 5]

Input : Stack[] = [1, 2, 3, 4, 5, 6] Output : Stack[] = [1, 2, 4, 5, 6]

**package** JavaPackage;

**import** java.util.Stack;

**public** **class** DeleteMiddleElement {

**public** **static** **void** main(String[] args) {

Stack<Integer> stack1 = **new** Stack<>();

stack1.push(1);

stack1.push(2);

stack1.push(3);

stack1.push(4);

stack1.push(5);

*deleteMiddle*(stack1);

System.***out***.println(stack1);

Stack<Integer> stack2 = **new** Stack<>();

stack2.push(1);

stack2.push(2);

stack2.push(3);

stack2.push(4);

stack2.push(5);

stack2.push(6);

*deleteMiddle*(stack2);

System.***out***.println(stack2);

}

**public** **static** **void** deleteMiddle(Stack<Integer> stack) {

**int** size = stack.size();

**if** (size == 0) {

**return**;

}

**int** middleIndex = size / 2;

*deleteMiddleUtil*(stack, middleIndex);

}

**private** **static** **void** deleteMiddleUtil(Stack<Integer> stack, **int** middleIndex) {

**if** (middleIndex == 0) {

stack.pop();

**return**;

}

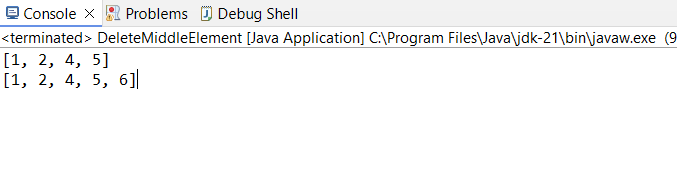
**int** temp = stack.pop();

*deleteMiddleUtil*(stack, middleIndex - 1);

stack.push(temp);

}

}



**Time Complexity**:O(n)

**18. Next Greater Element** (NGE) for every element in given Array Given an array, print the Next Greater Element (NGE) for every element. Note: The Next greater Element for an element x is the first greater element on the right side of x in the array.

Elements for which no greater element exist, consider the next greater element as -1.

Input: arr[] = [ 4 , 5 , 2 , 25 ] Output: 4 –> 5 5 –> 25 2 –> 25 25 –> -1 Explanation: Except 25 every element has an element greater than them present on the right side

Input: arr[] = [ 13 , 7, 6 , 12 ] Output: 13 –> -1 7 –> 12 6 –> 12 12 –> -1 Explanation: 13 and 12 don‟t have any element greater than them present on the right side

**package** JavaPackage;

**import** java.util.Stack;

**public** **class** NextGreaterElement {

**public** **static** **void** main(String[] args) {

**int**[] arr1 = {4, 5, 2, 25};

*printNextGreaterElements*(arr1);

**int**[] arr2 = {13, 7, 6, 12};

*printNextGreaterElements*(arr2);

}

**public** **static** **void** printNextGreaterElements(**int**[] arr) {

**int** n = arr.length;

**int**[] nge = **new** **int**[n];

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

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

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

nge[stack.pop()] = arr[i];

}

stack.push(i);

}

**while** (!stack.isEmpty()) {

nge[stack.pop()] = -1;

}

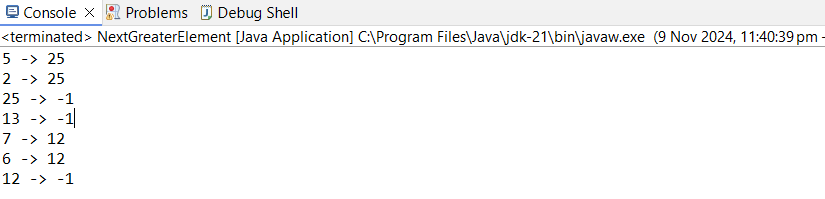
**for** (**int** i = 0; i < n; i++) {

System.***out***.println(arr[i] + " -> " + nge[i]);

}

}

}



**Time Complexity**:O(n)

**19. Print Right View of a Binary Tree** Given a Binary Tree, the task is to print the Right view of it. The right view of a Binary Tree is a set of rightmost nodes for every level

**package** JavaPackage;

**import** java.util.\*;

**class** TreeNode {

**int** data;

TreeNode left, right;

TreeNode(**int** data) {

**this**.data = data;

left = right = **null**;

}

}

**public** **class** BinaryTree {

TreeNode root;

**public** **void** rightView(TreeNode root) {

**if** (root == **null**) **return**;

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

queue.add(root);

**while** (!queue.isEmpty()) {

**int** size = queue.size();

**for** (**int** i = 1; i <= size; i++) {

TreeNode current = queue.poll();

**if** (i == size) {

System.***out***.print(current.data + " ");

}

**if** (current.left != **null**) queue.add(current.left);

**if** (current.right != **null**) queue.add(current.right);

}

}

}

**public** **static** **void** main(String[] args) {

BinaryTree tree = **new** BinaryTree();

tree.root = **new** TreeNode(1);

tree.root.left = **new** TreeNode(2);

tree.root.right = **new** TreeNode(3);

tree.root.left.right = **new** TreeNode(5);

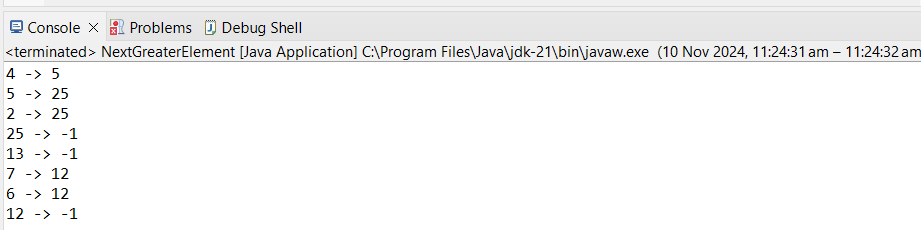
tree.root.right.right = **new** TreeNode(4);

System.***out***.print("Right view of the binary tree: ");

tree.rightView(tree.root);

}

}



**Time Complexity**:O(n)

20. Maximum Depth or Height of Binary Tree Given a binary tree, the task is to find the maximum depth or height of the tree. The height of the tree is the number of vertices in the tree from the root to the deepest node