

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.

```
import java.util.Scanner;
```

```
public class Solution {
    public static int maxSubarraySum(int[] arr) {
        int curr_max = arr[0];
        int max = arr[0];

        for (int i = 1; i < arr.length; i++) {
            curr_max = Math.max(arr[i], arr[i] + curr_max);
            max = Math.max(max, curr_max);
        }
        return max;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Ask for the number of elements in the array
        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();

        int[] arr = new int[n];

        // Ask for each element
        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }

        // Get the maximum subarray sum and print the result
        int maxSum = maxSubarraySum(arr);
        System.out.println("The maximum subarray sum is: " + maxSum);

        scanner.close();
    }
}
```

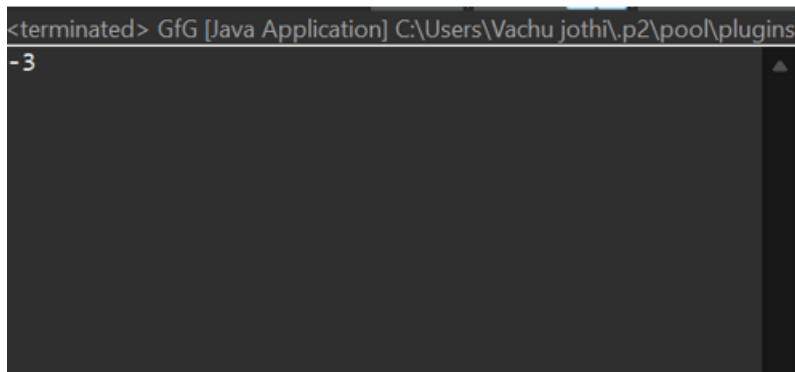
Hidden test case:

Input: {1,6,3,8,4}

Output: 22

Input: {3}

Output: -3



time complexity: $O(N)$

space complexity: $O(1)$

## 2. 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}.

```
import java.util.Scanner;
```

```
public class Solution {  
    public static int maxProduct(int[] nums) {  
        int n = nums.length;  
  
        if (n == 0) {  
            return 0;  
        }  
    }  
}
```

```

int max_pro = nums[0];
int min_pro = nums[0];
int tot_max_pro = nums[0];

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

    // Swap max and min when the current number is negative
    if (num < 0) {
        int temp = max_pro;
        max_pro = min_pro;
        min_pro = temp;
    }

    max_pro = Math.max(num, max_pro * num);
    min_pro = Math.min(num, min_pro * num);

    tot_max_pro = Math.max(tot_max_pro, max_pro);
}

return tot_max_pro;
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Prompt the user for the number of elements in the array
    System.out.print("Enter the number of elements in the array: ");
    int n = scanner.nextInt();

```

```
int[] nums = new int[n];

// Prompt the user for each element
System.out.println("Enter the elements of the array:");
for (int i = 0; i < n; i++) {
    nums[i] = scanner.nextInt();
}

// Calculate and display the maximum product of a subarray
int result = maxProduct(nums);

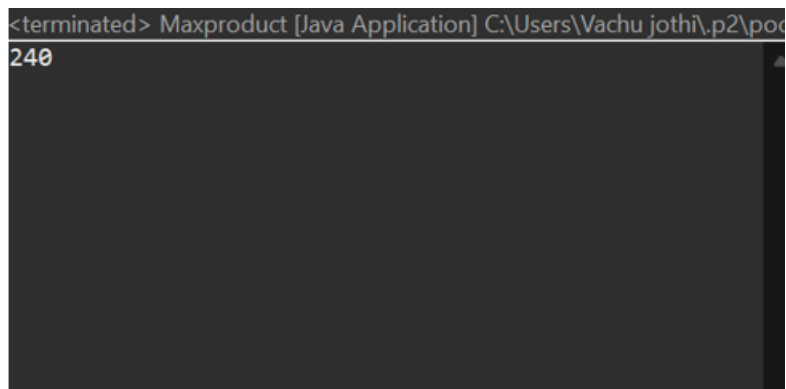
System.out.println("The maximum product of a subarray is: " + result);

scanner.close();
}
}
```

Hidden test case:

Input: { -2, 2,-3,4,5,-1 }

Output: 240



time complexity: $O(N)$

space complexity: $O(1)$

### 3. 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

```
import java.util.*;
```

```
public class GFG {  
    public static int pivotedSearch(List<Integer> arr, int key) {  
        int low = 0, high = arr.size() - 1;  
  
        while (low <= high) {  
            int mid = low + (high - low) / 2;  
            if (arr.get(mid) == key)  
                return mid;  
            if (arr.get(mid) >= arr.get(low)) {  
                if (key >= arr.get(low) && key < arr.get(mid))  
                    high = mid - 1;  
                else  
                    low = mid + 1;  
            }  
            else {  
                if (key > arr.get(mid) && key <= arr.get(high))  
                    low = mid + 1;  
                else  
                    high = mid - 1;  
            }  
        }  
  
        return -1;  
    }  
  
    public static void main(String[] args) {  
        List<Integer> arr1 = Arrays.asList(4, 5, 6, 7, 0, 1, 2);  
        int key1 = 0;  
        int result1 = pivotedSearch(arr1, key1);  
        System.out.println(result1);  
  
        List<Integer> arr2 = Arrays.asList(4, 5, 6, 7, 0, 1, 2);  
        int key2 = 3;  
        int result2 = pivotedSearch(arr2, key2);  
        System.out.println(result2);  
    }  
}
```

```

java -cp /tmp/dRQgesfgbi/GFG
4
-1

=== Code Execution Successful ===

```

Time complexity:  $O(\log n)$

Space complexity:  $O(1)$

#### 4. Container with Most Water

Given  $n$  non-negative integers  $a_1, a_2, \dots, a_n$  where each represents a point at coordinate  $(i, a_i)$ . ' $n$ ' vertical lines are drawn such that the two endpoints of line  $i$  is at  $(i, a_i)$  and  $(i, 0)$ . Find two lines, which together with x-axis forms a container, such that the container contains the most water.

The program should return an integer which corresponds to the maximum area of water that can be contained (maximum area instead of maximum volume sounds weird but this is the 2D plane we are working with for simplicity).

**Note:** You may not slant the container.

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 =  $4 * 3 = 12$

```
import java.util.Scanner;
```

```

class Solution {
    public int maxArea(int[] h) {
        int max = Integer.MIN_VALUE;

        int i = 0;
        int j = h.length - 1;

        while (i < j) {

```

```

        int min = Math.min(h[i], h[j]);

        int diff = j - i;

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

        while (i < j && h[i] <= min) i++;
        while (i < j && h[j] <= min) j--;
    }

    return max;
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter the number of elements in the array:");
    int n = scanner.nextInt();

    int[] h = new int[n];
    System.out.println("Enter the elements of the array:");
    for (int k = 0; k < n; k++) {
        h[k] = scanner.nextInt();
    }

    Solution solution = new Solution();
    int result = solution.maxArea(h);

    System.out.println("Maximum area of water that can be contained: " + result);

    scanner.close();
}
}

```

Hidden Test Case:

Input: {3, -1,3,5,6,7}

Output: 15

```
java -cp /tmp/21aQbVIfLw/GfG  
12  
  
=== Code Execution Successful ===
```

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

**Space Complexity:**  $O(1)$

5. Find the Factorial of a large number

Input: 100 Output:

933262154439441526816992388562667004907159682643816214685929638952175999932299  
1560894146397615651828625369792082722375825118521091686400000000000000000000 00

Input: 50 Output: 30414093201713378043612608166064768844377641568960512000000000000

```
class GFG {  
    static void factorial(int n)  
    {  
        int res[] = new int[500];  
        res[0] = 1;  
        int res_size = 1;  
        for (int x = 2; x <= n; x++)  
            res_size = multiply(x, res, res_size);  
  
        System.out.println("Factorial of given number is ");  
        for (int i = res_size - 1; i >= 0; i--)  
            System.out.print(res[i]);  
    }  
}
```



```
java -cp /tmp/3Wzuncvln8/GFG  
Factorial of given number is  
78865786736479050355236321393218506229513597768717326329474253324435944  
9963403342920304284011984623904177212138919638830257642790242637105  
0619266249528299311134628572707633172373969889439224456214516642
```

**Time Complexity:**  $O(N^2 \cdot \log n)$

**Space Complexity:**  $O(n \cdot \log n)$

#### 6. 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  $0 + 0 + 5 + 0 = 5$

```
import java.util.Scanner;

public class Solution {

    public static int trap(int[] height) {

        int h = height.length;

        if (height == null || h == 0) {

            return 0;

        }

        int left = 0;

        int right = h - 1;

        int max_left = 0;

        int max_right = 0;

        int tot_area = 0;

        while (left < right) {

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

                if (height[left] >= max_left) {

                    max_left = height[left];

                } else {
```

```

        tot_area += max_left - height[left];
    }
    left++;
} else {
    if (height[right] >= max_right) {
        max_right = height[right];
    } else {
        tot_area += max_right - height[right];
    }
    right--;
}
}
return tot_area;
}

```

```

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Input length of the height array
    System.out.print("Enter the number of elements in height array: ");
    int n = scanner.nextInt();

    // Input elements of the height array
    int[] height = new int[n];
    System.out.println("Enter the elements of height array:");
    for (int i = 0; i < n; i++) {
        height[i] = scanner.nextInt();
    }

    // Calculate and print the total trapped water
    int result = trap(height);
}

```

```
System.out.println("Total trapped water: " + result);
```

```
scanner.close();
```

```
}
```

```
}
```

Hidden Test cases:

Input: { 2,3,5,3,6,1 }

Output: 2

```
java -cp /tmp/enunSPnzww/GfG
2
=== Code Execution Successful ===
```

Time complexity:  $O(n)$

Space complexity:  $O(1)$

## 7. Chocolate Distribution Problem :

Given an array `arr[]` of  $n$  integers where `arr[i]` represents the number of chocolates in  $i$ th 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  $9 - 2 = 7$ .

```
import java.util.*;
```

```
public class ChocolateDistribution {
```

```
    static int findMinDiff(int[] arr, int n, int m) {
```

```
        if (m == 0 || n == 0) {
```

```
            return 0;
```

```
        }
```

```
        Arrays.sort(arr);
```

```

    if (n < m) {
        return -1;
    }

    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);
    }

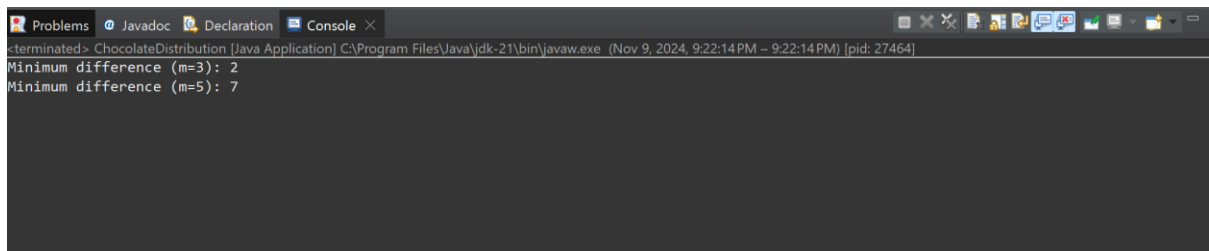
    return minDiff;
}

public static void main(String[] args) {
    int arr1[] = {7, 3, 2, 4, 9, 12, 56};
    int m1 = 3;
    int n1 = arr1.length;
    System.out.println("Minimum difference (m=3): " + findMinDiff(arr1, n1, m1));
    int arr2[] = {7, 3, 2, 4, 9, 12, 56};
    int m2 = 5;
    int n2 = arr2.length;
    System.out.println("Minimum difference (m=5): " + findMinDiff(arr2, n2, m2));
}
}

```

Hidden Test cases:

Input: {7, 3, 2, 4, 9, 12, 56}, m=4



```

<terminated> ChocolateDistribution [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Nov 9, 2024, 9:22:14 PM - 9:22:14 PM) [pid: 27464]
Minimum difference (m=3): 2
Minimum difference (m=5): 7

```

Time complexity:  $O(n \log n)$

Space complexity:  $O(1)$

## 8. Merge overlapping interval

Given an array of time intervals where  $arr[i] = [start_i, end_i]$ , 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  $[1, 6]$

```
import java.util.*;

public class MergeIntervals {

    public static int[][] mergeIntervals(int[][] intervals) {

        if (intervals == null || intervals.length == 0) {

            return new int[0][0];

        }

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

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

        merged.add(intervals[0]);

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

            int[] current = intervals[i];

            int[] lastMerged = merged.get(merged.size() - 1);

            if (current[0] <= lastMerged[1]) {

                lastMerged[1] = Math.max(lastMerged[1], current[1]);

            } else {

                merged.add(current);

            }

        }

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

    }

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

```

int[][] intervals1 = {{1, 3}, {2, 4}, {6, 8}, {9, 10}};

int[][] result1 = mergeIntervals(intervals1);

System.out.println("Merged Intervals 1: " + Arrays.deepToString(result1));

int[][] intervals2 = {{7, 8}, {1, 5}, {2, 4}, {4, 6}};

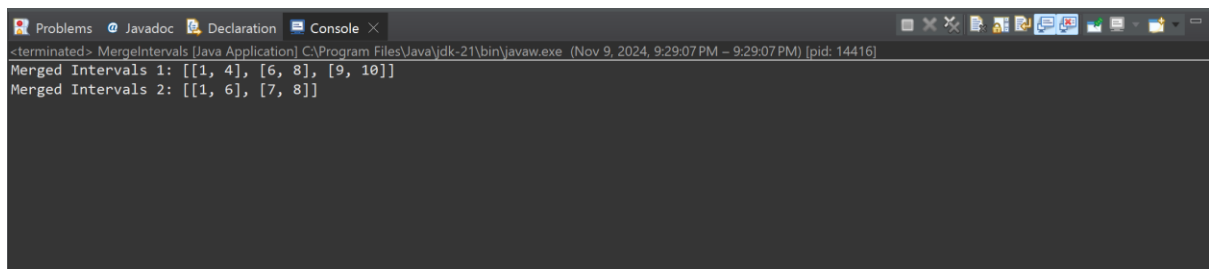
int[][] result2 = mergeIntervals(intervals2);

System.out.println("Merged Intervals 2: " + Arrays.deepToString(result2));

}

}

```



```

<terminated> MergeIntervals [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Nov 9, 2024, 9:29:07 PM - 9:29:07 PM) [pid: 14416]
Merged Intervals 1: [[1, 4], [6, 8], [9, 10]]
Merged Intervals 2: [[1, 6], [7, 8]]

```

Time Complexity:  $O(n \log n)$

Space Complexity:  $O(n)$

## 9.BooleanMatrix

Given a boolean matrix `mat[M][N]` of size  $M \times N$ , modify it such that if a matrix cell `mat[i][j]` is 1 (or true) then make all the cells of *i*th row and *j*th 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 sample1;

import java.util.*;

public class BooleanMatrix {

    public static void modify(int[][] mat) {

        int M = mat.length;

        int N = mat[0].length;

        boolean[] rowFlag = new boolean[M];

```

```

boolean[] colFlag = new boolean[N];
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        if (mat[i][j] == 1) {
            rowFlag[i] = true;
            colFlag[j] = true;
        }
    }
}

for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        if (rowFlag[i] || colFlag[j]) {
            mat[i][j] = 1;
        }
    }
}

public static void print(int[][] mat) {
    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat[i].length; j++) {
            System.out.print(mat[i][j] + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    int[][] mat1 = {{1, 0}, {0, 0}};

    System.out.println("Modified Matrix 1:");
    modify(mat1);
    print(mat1);
}

```



```

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

System.out.println("Modified Matrix 2:");

modify(mat2);

print(mat2);

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

System.out.println("Modified Matrix 3:");

modify(mat3);

print(mat3);

}

}

```

```

Modified Matrix 1:
1 1
1 0
Modified Matrix 2:
0 0 1
1 1 1
Modified Matrix 3:
1 1 1 1
1 1 1 1
1 0 1 1

```

Time Complexity:  $O(M*N)$

Space Complexity:  $O(M+N)$

## 10. Spiral Matrix

Given an  $m \times n$  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 sample1;

public class SpiralMatrix {

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

        int m = matrix.length;

        int n = matrix[0].length;
    }
}

```

```

int top = 0, bottom = m - 1, left = 0, right = n - 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) {

```

```

    int[][] matrix1 = {
        {1, 2, 3, 4},

```

```

        {5, 6, 7, 8},

        {9, 10, 11, 12},

        {13, 14, 15, 16}

    };

    System.out.println("Spiral Form of Matrix 1:");

    printSpiral(matrix1);

    System.out.println();

    int[][] matrix2 = {

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

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

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

    };

    System.out.println("Spiral Form of Matrix 2:");

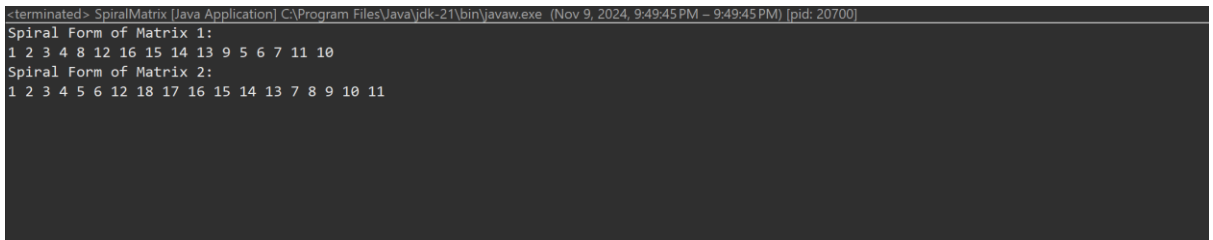
    printSpiral(matrix2);

    System.out.println();

}

}

```



```

<terminated> SpiralMatrix [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Nov 9, 2024, 9:49:45 PM - 9:49:45 PM) [pid: 20700]
Spiral Form of Matrix 1:
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
Spiral Form of Matrix 2:
1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11

```

Time Complexity:  $O(m*n)$

Space Complexity:  $O(1)$

### 13. Check if the given parenthesis string 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 sample1;

import java.util.*;

public class Paranthesis {

    static String check(String s) {

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

        for(char i:s.toCharArray()) {

            if(i=='(') {

                st.push(i);

            }

            else if(i==')'){

                if(!st.isEmpty()) {

                    st.pop();

                }

                else {

                    return "not Balanced";

                }

            }

        }

        return st.isEmpty()?"Balanced":"not Balanced";

    }

    public static void main(String[] args) {

        String res=check("()");

        System.out.println(res);

        String res1=check("((()))()");

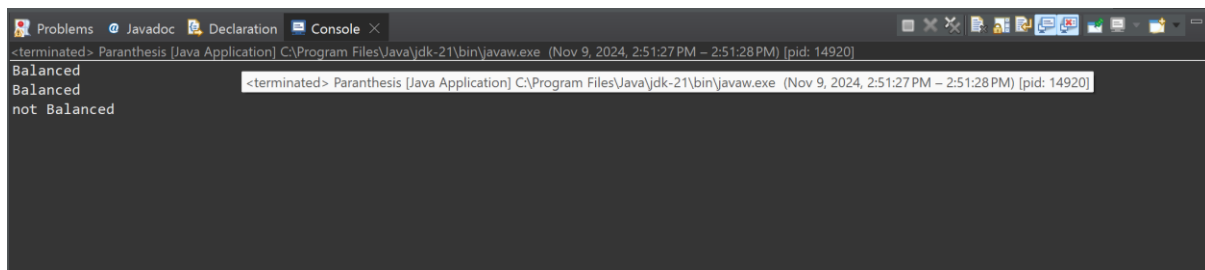
        System.out.println(res1);

        String res2=check("()((()))");

        System.out.println(res2);

    }

}
```



```
<terminated> Paranthesis [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Nov 9, 2024, 2:51:27 PM - 2:51:28 PM) [pid: 14920]
Balanced
Balanced
not Balanced
<terminated> Paranthesis [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Nov 9, 2024, 2:51:27 PM - 2:51:28 PM) [pid: 14920]
```

Time Complexity:  $O(n)$

Space Complexity:  $O(n)$

#### 14. Two Strings are anagram or not

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:  $s1 = \text{"geeks"}$   $s2 = \text{"kseeeg"}$  Output: true Explanation: Both the string have same characters with same frequency. So, they are anagrams.

Input:  $s1 = \text{"allergy"}$   $s2 = \text{"allergic"}$  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:  $s1 = \text{"g"}$ ,  $s2 = \text{"g"}$  Output: true Explanation: Characters in both the strings are same, so they are anagrams.

```
import java.util.*;

public class ValidAnagram {

    public static boolean valid(String s1, String s2) {

        char[] lst1=s1.toCharArray();

        char[] lst2=s2.toCharArray();

        if(lst1.length!=lst2.length) {

            return false;

        }

        Arrays.sort(lst1);

        Arrays.sort(lst2);

        return Arrays.equals(lst1,lst2);

    }

}
```

```

    }

    public static void main(String[] args) {

        String s = "geeks";

        String t = "skeeg";

        System.out.println(valid(s,t));


        String s1 = "allergy";

        String t1 = "allergic";

        System.out.println(valid(s1,t1));

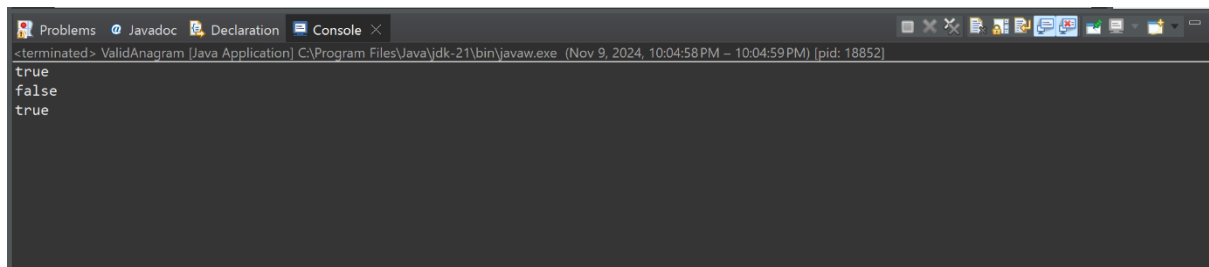

        String s2 = "g";

        String t2 = "g";

        System.out.println(valid(s2,t2));

    }
}

```



The screenshot shows a Java IDE window with a console tab. The console output displays the results of the `valid` method for three different input pairs: `(s, t)`, `(s1, t1)`, and `(s2, t2)`. The output is as follows:

```

<terminated> ValidAnagram [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Nov 9, 2024, 10:04:58 PM - 10:04:59 PM) [pid: 18852]
true
false
true

```

Time Complexity:  $O(n \log n)$

Space Complexity:  $O(n)$

## 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", "eeksske" etc. But the substring "geeksskeeg" is the longest among all.

Input: str = "Geeks" Output: "ee"

Input: str = "abc" Output: "a" Input: str = "" Output: ""

```
import java.util.*;

public class longPalindrome {

    public static String longestPalindrome(String str) {

        if (str == null || str.length() == 0) {
            return "";
        }

        int n = str.length();

        int start = 0;

        int maxLength = 1;

        boolean[][] dp = new boolean[n][n];

        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        for (int length = 2; length <= n; length++) {
            for (int i = 0; i < n - length + 1; i++) {
                int j = i + length - 1;
                if (str.charAt(i) == str.charAt(j)) {
                    if (length == 2 || dp[i + 1][j - 1]) {
                        dp[i][j] = true;

                        if (length > maxLength) {
                            maxLength = length;
                        }
                    }
                }
            }
        }

        return str.substring(start, start + maxLength);
    }
}
```

```

        start = i;
    }
}
}
}
}

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

public static void main(String[] args) {

    String str1 = "forgeeksskeegfor";

    System.out.println("Longest Palindromic Substring: " + longestPalindrome(str1));

    String str2 = "Geeks";

    System.out.println("Longest Palindromic Substring: " + longestPalindrome(str2));

    String str3 = "abc";

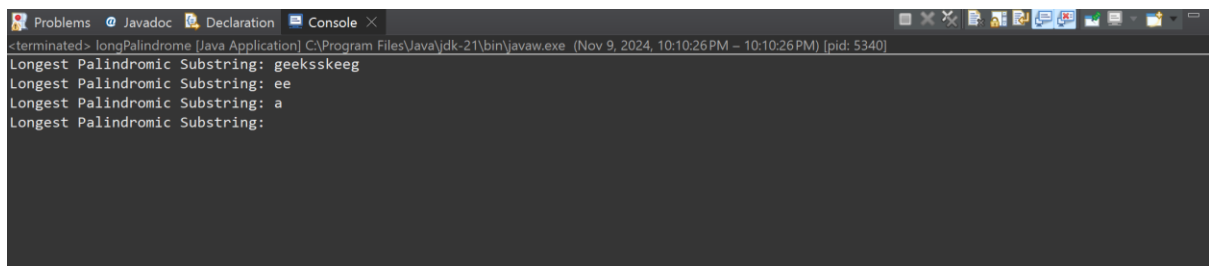
    System.out.println("Longest Palindromic Substring: " + longestPalindrome(str3));

    String str4 = "";

    System.out.println("Longest Palindromic Substring: " + longestPalindrome(str4));

}
}

```



The screenshot shows a Java IDE window with a console output. The console displays the results of the program for four different input strings. The output is as follows:

```

<terminated> longPalindrome (Java Application) C:\Program Files\Java\jdk-21\bin\javaw.exe (Nov 9, 2024, 10:10:26 PM ~ 10:10:26 PM) [pid: 5340]
Longest Palindromic Substring: geeksskeeg
Longest Palindromic Substring: ee
Longest Palindromic Substring: a
Longest Palindromic Substring:

```

Time Complexity:  $O(n^2)$

Space Complexity:  $O(N^2)$



## 16. Longest Common prefix

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 sample1;

import java.util.*;

public class CommonPrefix {

    static String Common(String[] arr) {

        Arrays.sort(arr);

        String f=arr[0];

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

        int n=Math.min(f.length(),l.length());

        String res="";

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

            if(f.charAt(i)==l.charAt(i)){

                res+=f.charAt(i);

            }

            else{

                break;

            }

        }

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

    }

    public static void main(String[] args) {

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

        System.out.println(Common(arr1));

    }

}
```

```

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

        System.out.println(Common(arr2));

    }
}

```



Time Complexity: O(n)

Space Complexity:O(n)

### 17.Delete Middle element of the 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 sample1;

import java.util.*;

public class MidDel {

    static void deleteMiddle(Stack<Integer> stack, int n, int curr) {

        // If the stack is empty or all items are traversed
        if (stack.isEmpty() || curr == n) {

            return;

        }

        // Remove the current item
        int x = stack.pop();

        // Recursively reach the middle element
        deleteMiddle(stack, n, curr + 1);
    }
}

```

```
// Only push the element back if it's not the middle one
if (curr != n / 2) {
    stack.push(x);
}
}
```

```
// Wrapper function to start recursion
static void deleteMiddle(Stack<Integer> stack) {
    int n = stack.size();
    deleteMiddle(stack, n, 0);
}
```

```
public static void main(String[] args) {
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);
    stack.push(5);

    System.out.println("Original stack: " + stack);
    deleteMiddle(stack);
    System.out.println("Stack after deleting middle element: " + stack);

    Stack<Integer> stackEven = new Stack<>();
    stackEven.push(1);
    stackEven.push(2);
    stackEven.push(3);
    stackEven.push(4);
    stackEven.push(5);
}
```

```

stackEven.push(6);

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

deleteMiddle(stackEven);

System.out.println("Stack after deleting middle element: " + stackEven);
}
}

```

```

Original stack: [1, 2, 3, 4, 5]
Stack after deleting middle element: [1, 2, 4, 5]
Original stack: [1, 2, 3, 4, 5, 6]
Stack after deleting middle element: [1, 2, 4, 5, 6]

```

Time Complexity: O(n)

Space Complexity: O(n)

## 18. Next Greater Element

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 sample1;

import java.util.*;

public class NextGreat {

    static void Greater(int[] arr) {

        int n = arr.length;

        int[] nge = new int[n];

```

```

Stack<Integer> stack = new Stack<>();
Arrays.fill(nge, -1);
for (int i = n - 1; i >= 0; i--) {
    while (!stack.isEmpty() && stack.peek() <= arr[i]) {
        stack.pop();
    }
    if (!stack.isEmpty()) {
        nge[i] = stack.peek();
    }
    stack.push(arr[i]);
}
for (int i = 0; i < n; i++) {
    System.out.println(arr[i] + " --> " + nge[i]);
}
}

public static void main(String[] args) {
    int[] arr1 = {4, 5, 2, 25};

    System.out.println("Next Greater Elements for the array " + Arrays.toString(arr1) + ":");
    Greater(arr1);

    System.out.println();

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

    System.out.println("Next Greater Elements for the array " + Arrays.toString(arr2) + ":");
    Greater(arr2);
}
}

```

```

Next Greater Elements for the array [4, 5, 2, 25]:
4 --> 5
5 --> 25
2 --> 25
25 --> -1

Next Greater Elements for the array [13, 7, 6, 12]:
13 --> -1
7 --> 12
6 --> 12
12 --> -1

```

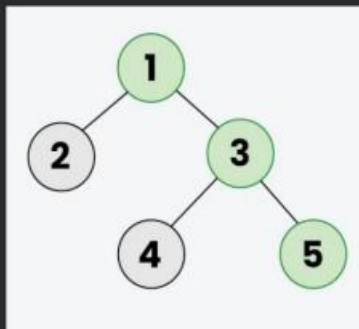
Time Complexity:  $O(n)$

Space Complexity:  $O(n)$

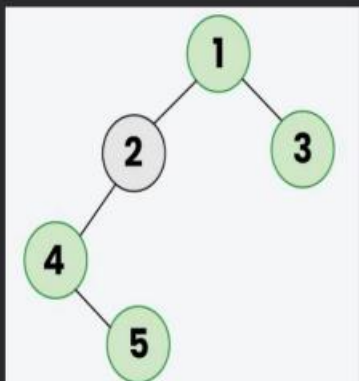
### 19. Right view of the binary tree

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.

*Example 1: The **Green** colored nodes (1, 3, 5) represents the Right view in the below Binary tree.*



*Example 2: The **Green** colored nodes (1, 3, 4, 5) represents the Right view in the below Binary tree.*



```
package sample1;

import java.util.*;

class Node {
    int val;
    Node left, right;

    Node(int val) {
        this.val = val;
        left = right = null;
    }
}
```

```

    }
}

public class BinaryTreeRight {

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

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

        if (root == null) {

            return result;

        }

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

        queue.add(root);

        while (!queue.isEmpty()) {

            int cap = queue.size();

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

                Node curr = queue.poll();

                if (i == cap - 1) {

                    result.add(curr.val);

                }

                if (curr.left != null) {

                    queue.add(curr.left);

                }

                if (curr.right != null) {

                    queue.add(curr.right);

                }

            }

        }

        return result;

    }

}

```

```

public static void main(String[] args) {

    Node root = new Node(1);

    root.left = new Node(2);

```

```

root.right = new Node(3);

root.left.right = new Node(5);

root.right.right = new Node(4);

List<Integer> rightView = rightView(root);

System.out.println("Right view of the binary tree: " + rightView);
}
}

```

```

Right view of the binary tree: [1, 3, 4]

```

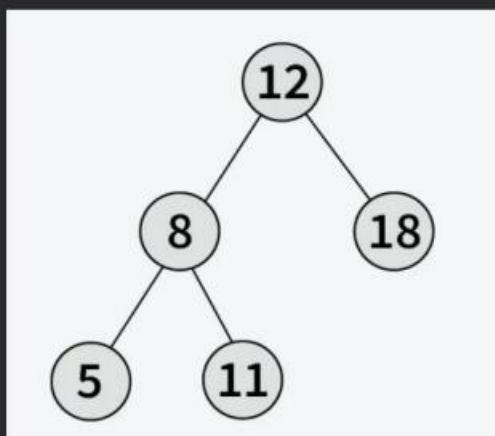
Time complexity:  $O(n)$

Space Complexity:  $O(n)$

## 20. Maximum height or depth of the binary tree

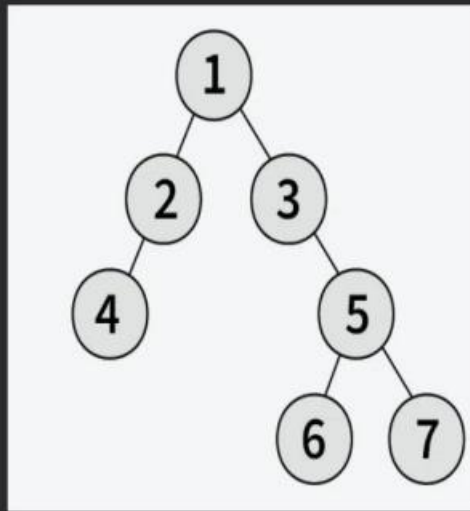
Given a binary tree, the task is to find the maximum depth or height of the tree. The height of the tree is the number of vertices in the tree from the root to the deepest node

*Example 1: The height of the below binary tree is 3.*





*Example 2: The height of the below binary tree is 4*



```
package sample1;

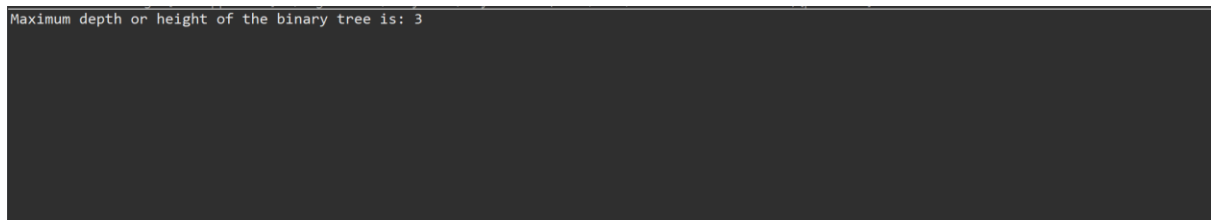
import java.util.*;

class Node {
    int val;
    Node left, right;
    Node(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}

public class TreeHeight {
    static int maxDepth(Node root) {
        if (root == null) {
            return 0;
        }
        int leftDepth = maxDepth(root.left);
        int rightDepth = maxDepth(root.right);
        return Math.max(leftDepth, rightDepth) + 1;
    }

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

```
Node root = new Node(1);  
root.left = new Node(2);  
root.right = new Node(3);  
root.left.left = new Node(4);  
root.left.right = new Node(5);  
System.out.println("Maximum depth or height of the binary tree is: " + maxDepth(root));  
}  
}
```



```
Maximum depth or height of the binary tree is: 3
```

Time Complexity:  $O(n)$

Space Complexity:  $O(h)$ , where  $h$  is the height of the tree.