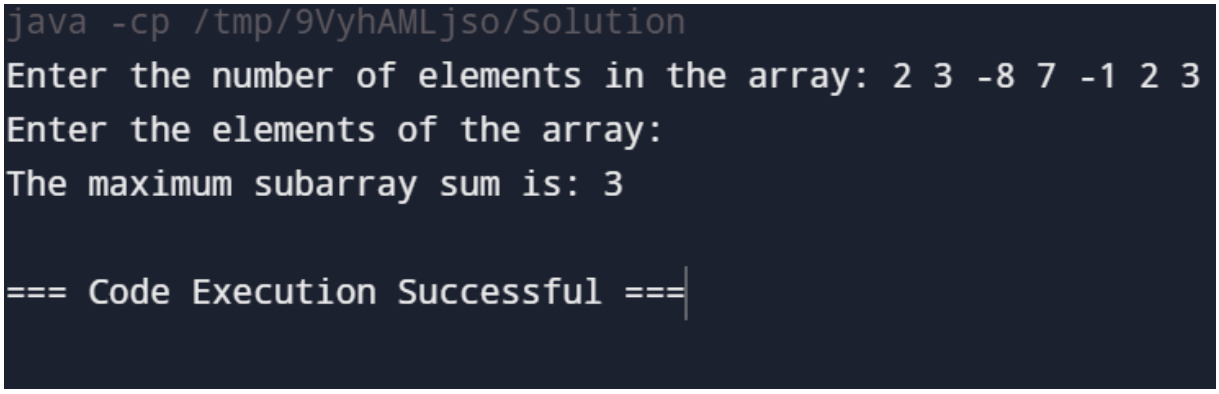
1. Maximum Subarray Sum – Kadane‟s Algorithm:

```
import java.util.Scanner;

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

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

   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 maxSum = maxSubarraySum(arr);
      System.out.println("The maximum subarray sum is: " + maxSum);

      scanner.close();
   }
}
```

Input: arr[] = {2, 3, -8, 7, -1, 2, 3}

Output: 11

Explanation: The subarray {7, -1, 2, 3} has the largest sum 11.



time complexity:O(N)

complexity:O(1)

2.Maximum Product Subarray :

```
import java.util.Scanner;

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

        int maxProd = nums[0];
        int minProd = nums[0];
        int totalMaxProd = nums[0];

        for (int i = 1; i < n; i++) {
            int num = nums[i];
            if (num < 0) {
                int temp = maxProd;
                maxProd = minProd;
                minProd = temp;
```

```
            }

            maxProd = Math.max(num, maxProd * num);
            minProd = Math.min(num, minProd * num);
            totalMaxProd = Math.max(totalMaxProd, maxProd);
        }

        return totalMaxProd;
    }

    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[] nums = new int[n];

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

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

        scanner.close();
    }
}
```

time complexity:O(N) space

complexity:O(1)

3.Search in a sorted and rotated Array :

```
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;
      System.out.println(pivotedSearch(arr1, key1));

      List<Integer> arr2 = Arrays.asList(4, 5, 6, 7, 0, 1, 2);
```

```
        int key2 = 3;
        System.out.println(pivotedSearch(arr2, key2));
    }
}
```

```
4
-1

=== Code Execution Successful ===
```

Time complexity: O(log n)

Space complexity: O(1)

4. Container with Most Water

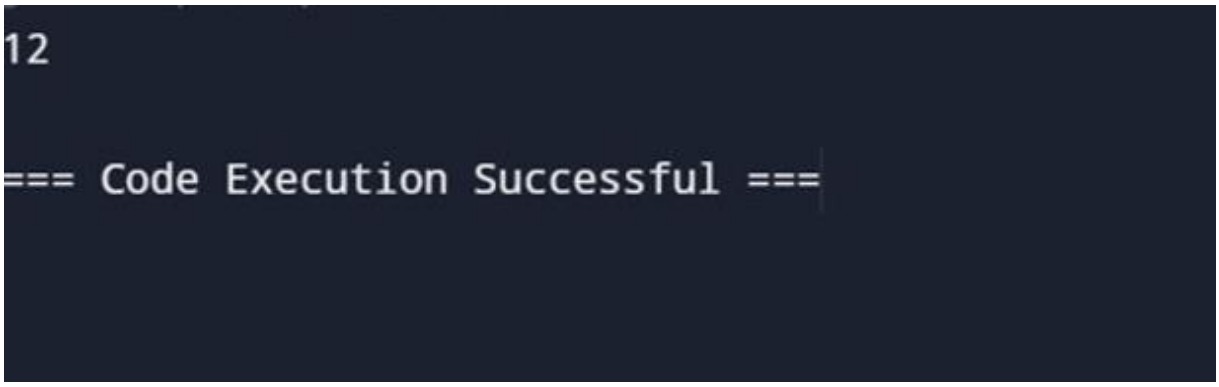
```
 import java.util.Scanner;

class Solution {
    public int maxArea(int[] heights) {
        int maxArea = Integer.MIN_VALUE;
        int left = 0;
        int right = heights.length - 1;

        while (left < right) {
            int minHeight = Math.min(heights[left], heights[right]);
            int width = right - left;
            maxArea = Math.max(maxArea, minHeight * width);

            while (left < right && heights[left] <= minHeight) left++;
            while (left < right && heights[right] <= minHeight) right--;
        }

        return maxArea;
    }

    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[] heights = new int[n];
        System.out.println("Enter the elements of the array:");
```

```
        for (int i = 0; i < n; i++) {
            heights[i] = scanner.nextInt();
        }

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

        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



Time Complexity: O(n) **Space Complexity**: O(1)

5. Find the Factorial of a large number

```
class GFG {
    static void factorial(int n) {
        int result[] = new int[500];
        result[0] = 1;
        int resultSize = 1;
        for (int x = 2; x <= n; x++) {
            resultSize = multiply(x, result, resultSize);
        }
```

```
        System.out.println("Factorial of given number is ");
        for (int i = resultSize - 1; i >= 0; i--) {
            System.out.print(result[i]);
        }
    }

    static int multiply(int x, int result[], int resultSize) {
        int carry = 0;
        for (int i = 0; i < resultSize; i++) {
            int prod = result[i] * x + carry;
            result[i] = prod % 10;
            carry = prod / 10;
        }
        while (carry != 0) {
            result[resultSize] = carry % 10;
            carry = carry / 10;
            resultSize++;
        }
        return resultSize;
    }

    public static void main(String args[]) {
        factorial(100);
    }
}
```

Hidden TestCases:

Input: 200

Output: Factorial of given number is

788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113 4

628572707633172373969889439224456214516642402540332918641312274282948532775242424075739032403212574055795686602260319041703240623517008587961789222227896237038973747200000000000000000000000000000000000000000000000000

```
Factorial of given number is
788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664
```

Time Complexity: O(N^2.logn)

Space Complexity: O(n.logn)

6. Trapping Rainwater :

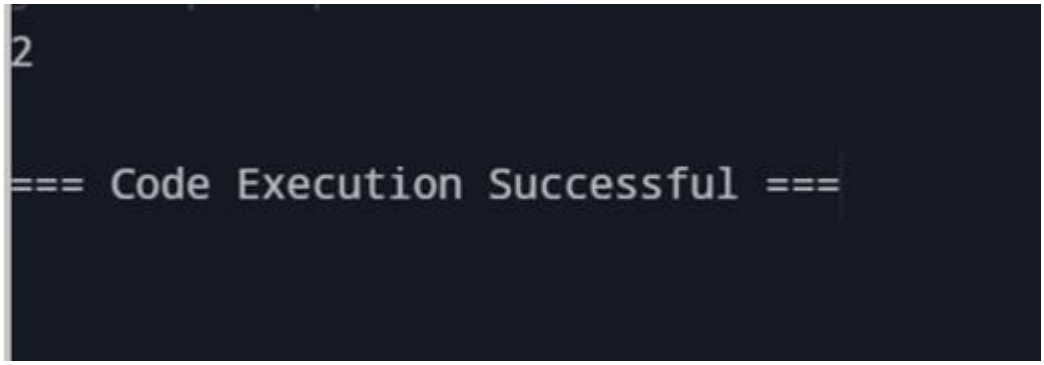
```
import java.util.Scanner;


public class Solution {
    public static int trap(int[] height) {
        int n = height.length;
        if (height == null || n == 0) {
            return 0;
        }

        int left = 0, right = n - 1;
        int maxLeft = 0, maxRight = 0;
        int totalWater = 0;

        while (left < right) {
            if (height[left] < height[right]) {
                if (height[left] >= maxLeft) {
                    maxLeft = height[left];
                } else {
                    totalWater += maxLeft - height[left];
                }
                left++;
            } else {
```

```
            if (height[right] >= maxRight) {
                maxRight = height[right];
            } else {
                totalWater += maxRight - height[right];
            }
            right--;
          }
        }
        return totalWater;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of elements in height array: ");
        int n = scanner.nextInt();

        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();
        }

        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



Time complexity:O(n)

Space complexity:O(1)

7. Chocolate Distribution Problem :

```
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

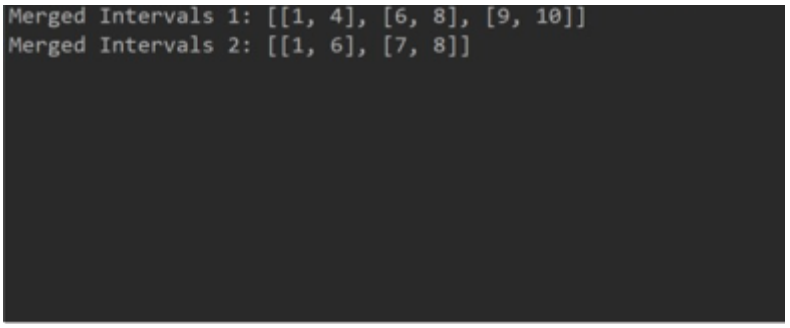
Time complexity: O(n log n)

Space complexity:O(1)

8. Merge overlapping interval

```
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));
    }
}
```



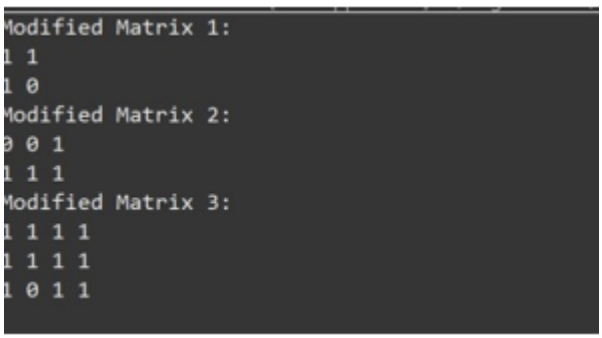
Time Complexity: O(n log n)

Space Complexity:O(n)

9.BooleanMatrix

```
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);
    }
}
```



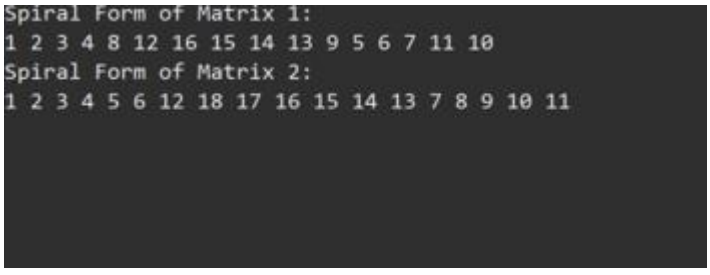
Time Complexity: O(M*N)

Space Complexity:O(M+N)

10. Spiral Matrix

```
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();
    }
}
```



Time Complexity: O(m*n)

Space Complexity:O(1)

13. Check if the given parenthesis string is balanced or not

```
import java.util.*;

public class Parenthesis {
   public static String check(String s) {
      Stack<Character> stack = new Stack<>();
      for (char c : s.toCharArray()) {
         if (c == '(') {
            stack.push(c);
         } else if (c == ')') {
            if (stack.isEmpty()) {
               return "Not Balanced";
            } else {
               stack.pop();
            }
         }
      }
      return stack.isEmpty() ? "Balanced" : "Not Balanced";
   }

   public static void main(String[] args) {
      String res1 = check("(((()))()()");
      System.out.println(res1);

      String res2 = check("())((())");
      System.out.println(res2);
   }
}
```

Time Complexity: O(n)

Space Complexity:O(n)

14. Two Strings are anagram or not

```
import java.util.*;

public class ValidAnagram {
   public static boolean valid(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);
   }

   public static void main(String[] args) {
      String s1 = "geeks";
      String s2 = "skeeg";
      System.out.println(valid(s1, s2)); // true

      String s3 = "allergy";
      String s4 = "allergic";
      System.out.println(valid(s3, s4)); // false

      String s5 = "g";
      String s6 = "g";
      System.out.println(valid(s5, s6)); // true
   }
}
```

```
true
false
true
```

Time Complexity: O(n log n)

Space Complexity:O(n)

15. Longest Palindromic substring

```
public class LongestPalindrome {
   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];

      // Every single character is a palindrome
      for (int i = 0; i < n; i++) {
         dp[i][i] = true;
      }

      // Check for substrings of length 2
      for (int i = 0; i < n - 1; i++) {
         if (str.charAt(i) == str.charAt(i + 1)) {
            dp[i][i + 1] = true;
            start = i;
            maxLength = 2;
         }
      }

      // Check for lengths greater than 2
      for (int length = 3; length <= n; length++) {
         for (int i = 0; i < n - length + 1; i++) {
            int j = i + length - 1;

            // Check if the substring from ith index to jth index is a palindrome
            if (str.charAt(i) == str.charAt(j) && dp[i + 1][j - 1]) {
```

```
                dp[i][j] = true;
                if (length > maxLength) {
                    start = i;
                    maxLength = length;
                }
            }
        }
    }

    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));
  }
}
```

```
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

```
public class LongestPalindrome {
  public static String findLongestPalindrome(String s) {
    if (s == null || s.length() == 0) {
      return "";
    }
```

```
    int len = s.length();
    int start = 0;
    int maxLen = 1;
    boolean[][] dp = new boolean[len][len];

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

    for (int i = 0; i < len - 1; i++) {
        if (s.charAt(i) == s.charAt(i + 1)) {
            dp[i][i + 1] = true;
            start = i;
            maxLen = 2;
        }
    }

    for (int l = 3; l <= len; l++) {
        for (int i = 0; i < len - l + 1; i++) {
            int j = i + l - 1;

            if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
                dp[i][j] = true;
                if (l > maxLen) {
                    start = i;
                    maxLen = l;
                }
            }
        }
    }

    return s.substring(start, start + maxLen);
}
```

```
    public static void main(String[] args) {
        String s1 = "forgeeksskeegfor";
        System.out.println("Longest Palindromic Substring: " + findLongestPalindrome(s1));

        String s2 = "Geeks";
        System.out.println("Longest Palindromic Substring: " + findLongestPalindrome(s2));

        String s3 = "abc";
        System.out.println("Longest Palindromic Substring: " + findLongestPalindrome(s3));

        String s4 = "";
        System.out.println("Longest Palindromic Substring: " + findLongestPalindrome(s4));
    }
}
```

```
gee
-1
```

Time Complexity: O(n)

Space Complexity:O(n)

17.Delete Middle element of the stack

```
import java.util.*;

public class MidDel {
    static void deleteMiddle(Stack<Integer> stack, int n, int curr) {
        if (stack.isEmpty() || curr == n) {
            return;
        }

        int x = stack.pop();
```

```
    deleteMiddle(stack, n, curr + 1);

    if (curr != n / 2) {
        stack.push(x);
    }
}

static void deleteMiddle(Stack<Integer> stack) {
    int n = stack.size();
    deleteMiddle(stack, n, 0);
}

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);

    System.out.println("Original stack: " + stack1);
    deleteMiddle(stack1);
    System.out.println("Stack after deleting middle element: " + stack1);

    Stack<Integer> stack2 = new Stack<>();
    stack2.push(1);
    stack2.push(2);
    stack2.push(3);
    stack2.push(4);
    stack2.push(5);
    stack2.push(6);
```

```
        System.out.println("Original stack: " + stack2);
        deleteMiddle(stack2);
        System.out.println("Stack after deleting middle element: " + stack2);
    }
}
```

```
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

```
import java.util.*;

public class NextGreat {
    static void findNextGreater(int[] arr) {
        int len = arr.length;
        int[] nge = new int[len];
        Stack<Integer> stack = new Stack<>();
        Arrays.fill(nge, -1);

        for (int i = len - 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 < len; 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) + ":");
        findNextGreater(arr1);

        System.out.println();

        int[] arr2 = {13, 7, 6, 12};
        System.out.println("Next Greater Elements for the array " + Arrays.toString(arr2) + ":");
        findNextGreater(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

```
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 count = queue.size();
         for (int i = 0; i < count; i++) {
            Node curr = queue.poll();
            if (i == count - 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);
  }
}
```

Time complexity: O(n)

Space Complexity:O(n)

20.Maximum height or depth of the binary tree

```
import java.util.*;

class Node {
   int val;
   Node left, right;

   Node(int val) {
      this.val = val;
      left = 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.