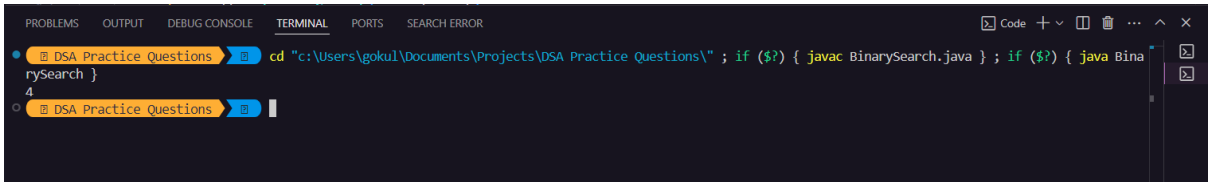
1.Given an array **arr[]** and an integer **k** where k is smaller than the size of the array, the task is to find the **kth smallest** element in the given array.

Solution:

```
class Solution {
    public static int kthSmallest(int[] arr, int k) {
        int max = Integer.MIN_VALUE;
        for(int i:arr) max = Math.max(i,max);
        int [] freq = new int[max+1];
        //for calculating the frequency
        for(int i=0;i<arr.length;i++){
            freq[arr[i]]+=1;
        }
        //find the Kth largest element
        int count =0;
        for(int i=0;i<freq.length;i++){
            count+=freq[i];
            if(count>=k) return i;
        }
        return 0;
    }
}
```



Time Complexity:O(n)

Space Complexity:O(maxelement)

2. Given an array **arr[]** denoting heights of **N** towers and a positive integer **K.**

For **each** tower, you must perform **exactly one** of the following operations **exactly once**.

- **Increase** the height of the tower by **K**

- **Decrease** the height of the tower by **K**

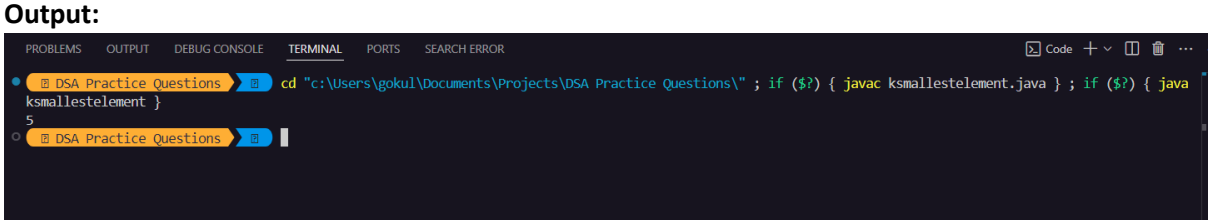
Find out the **minimum** possible difference between the height of the shortest and tallest towers after you have modified each tower.

You can find a slight modification of the problem here.
Note: It is **compulsory** to increase or decrease the height by K for each tower. **After** the operation, the resultant array should **not** contain any **negative integers**.

Solution:

```
import java.util.Arrays;


public class ksmallestelement {


    public static int getMinDiff(int[] arr, int k) {
        Arrays.sort(arr);
        int height = arr[arr.length-1]-arr[0];
        for(int i=0;i<arr.length-1;i++){
          int sh = Math.min(arr[0]+k,arr[i+1]-k);
          int th = Math.max(arr[arr.length-1]-k,arr[i]+k);
            if(sh<0) continue;
            height = Math.min(height,th-sh);
        }
        return height;
    }
    public static void main(String[] args) {
        int res= getMinDiff(new int[]{2,4,3,9,9,10,9,7,1,2},4);
        System.out.println(res);
    }
}
```

Output:



Time Complexity:O(n log n)

Space Complexity:O(1)

3. Given a string s, composed of different combinations of '(' , ')', '{', '}', '[', ']', verify the validity of the arrangement.
An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Solution:

```
class Solution
{
   boolean valid(String s)
   {
      Stack<Character> st = new Stack<>();
      for(int i=0;i<s.length();i++){
         char ch = s.charAt(i);
         if(ch=='['||ch=='{'||ch=='(') st.push(ch);
         else if(ch==']'||ch=='}'||ch==')'){
            if(st.isEmpty()){
               return false;
            }
            else{
               char top = st.pop();
               if((ch==']'&&top!='[')||(ch=='}'&&top!='{')||(ch==')'&&top!='(')){
                  return false;
               }
            }
         }
      }
      return st.isEmpty();
   }
}
```

Output

Time Complexity:O(n)

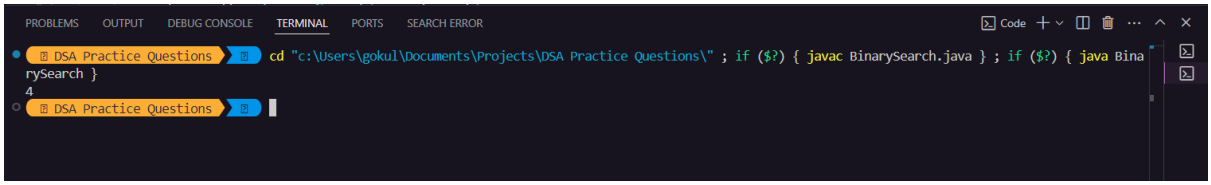
Space Complexity:O(n)

4**. Binary Search**

Solution:

```
public class BinarySearch {

    public static int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) return mid;
            if (arr[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        int target = 5;
        int result = binarySearch(arr, target);
        System.out.println(result);
    }
}
```

Output:



Time Complexity:O(log n)

Space Complexity:O(1)

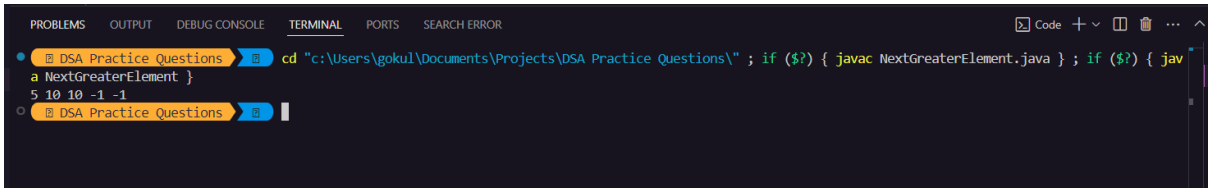
5. Given an array **arr[ ]** of integers, the task is to find the next greater element for each element of the array in order of their appearance in the array. Next greater element of an element in the array is the nearest element on the right which is greater than the current element.
If there does not exist next greater of current element, then next greater element for current element is -1. For example, next greater of the last element is always -1.

Solution:

```
import java.util.Stack;


public class NextGreaterElement {


    public static int[] nextGreaterElement(int[] nums) {
        int[] result = new int[nums.length];
        Stack<Integer> stack = new Stack<>();


        for (int i = nums.length - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= nums[i]) {
                stack.pop();
            }
            result[i] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(nums[i]);
        }


        return result;
    }


    public static void main(String[] args) {
        int[] nums = {4, 5, 2, 10, 8};
```

```
        int[] result = nextGreaterElement(nums);
        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

Ouput:



Time Complexity:O(n)

Space Complexity:O(n)

6. Given an array **arr** of non-negative numbers. The task is to find the first **equilibrium point** in an array. The equilibrium point in an array is an index (or position) such that the sum of all elements before that index is the same as the sum of elements after it.

Note: Return equilibrium point in 1-based indexing. Return -1 if no such point exists.

Solution

```
class Solution {

  // Function to find equilibrium point in the array.
    public static int equilibriumPoint(int arr[]) {
        int n = arr.length;
        if (n == 1) return 1;

        int totalSum = 0;
        for (int num : arr) {
            totalSum += num;
        }

        int leftSum = 0;
```

```
        for (int i = 0; i < n; i++) {
            // Subtract the current element from totalSum to get the right sum
            totalSum -= arr[i];
            // Check if leftSum equals rightSum
            if (leftSum == totalSum) {
                return i + 1;  // Return the 1-based index of the equilibrium point
            }

            // Update leftSum for the next position
            leftSum += arr[i];
        }

        return -1;  // No equilibrium point found
    }
}
```

Time Complexity:O(n)

Space Complexity:O(1)