Day 4

Prajwalhonashetti143@gmail.com

Task 1: Array Sorting and Searching

a) **Implement a function called BruteForceSort that sorts an array using the brute force approach. Use this function to sort an array created with InitializeArray.**

```
import java.util.Arrays;
import java.util.Random;

public class BruteForceSort {

   public static void main(String[] args) {
      int arraySize = 10;
      int[] myArray = InitializeArray(arraySize);

      System.out.println("Original array: " + Arrays.toString(myArray));

      BruteForceSort(myArray);

      System.out.println("Sorted array: " + Arrays.toString(myArray));
   }

   public static int[] InitializeArray(int size) {
      int[] arr = new int[size];
      Random rand = new Random();
      for (int i = 0; i < size; i++) {
         arr[i] = rand.nextInt(100) + 1;
      }
      return arr;
   }

   public static void BruteForceSort(int[] arr) {
      int n = arr.length;
      for (int i = 0; i < n; i++) {
         for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[i]) {
               // Swap arr[i] and arr[j]
               int temp = arr[i];
               arr[i] = arr[j];
               arr[j] = temp;
            }
         }
```

```
        }
    }
}
```

Explanation:

- **int[] arr = new int[size];:** Creates a new integer array of the specified size.
- **Random rand = new Random**();: Creates a new instance of Random to generate random numbers.
- **for (int i = 0; i < size; i++)** { ... }: Iterates over the array and assigns each element a random integer between 1 and 100 using **rand.nextInt(100) + 1**.
- **return arr;:** Returns the initialized array.

BruteForceSort Method:

- The BruteForceSort method implements a brute force approach (specifically, selection sort) to sort an integer array.
- **int n = arr.length;:** Gets the length of the array.
- The nested **for** loops iterate over the array:
- **for (int i = 0; i < n; i++):** Outer loop selects each element of the array.
- **for (int j = i + 1; j < n; j++):** Inner loop compares the selected element (**arr[i])** with each subsequent element (**arr[j]).**
- If **arr[j] < arr[i],** the elements are swapped to ensure the smaller element comes before the larger one.
- **Time Complexity: O(n^2)** where n is the number of elements in the array. This makes it inefficient for large arrays but suitable for demonstration purposes.

Method Details:

- **int arraySize = 10;:** Sets the size of the array to be initialized**.**
- **int[] myArray = InitializeArray(arraySize);:** Initializes an array of size 10 with random integers using InitializeArray method**.**
- System.out.println**("Original array: " + Arrays.toString(myArray));:** Prints the original array.
- **BruteForceSort(myArray);:** Sorts the array in-place using the **BruteForceSort** method**.**
- System.out.println**("Sorted array: " + Arrays.toString(myArray));:** Prints the sorted array.

- **Output Explanation**
  Original array: [42, 51, 32, 78, 82, 68, 96, 2, 88, 99]
  Sorted array: [2, 32, 42, 51, 68, 78, 82, 88, 96, 99]

b) **Write a function named PerformLinearSearch that searches for a specific element in an array and returns the index of the element if found or -1 if not found.**

```
public class LinearSearch {
    public static int performLinearSearch(int[] arr, int element) {
        // Iterate through the array
        for (int i = 0; i < arr.length; i++) {
            // If element found, return its index
            if (arr[i] == element) {
                return i;
            }
        }

        return -1;
    }

    public static void main(String[] args) {
        int[] array = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
        int elementToFind = 9;

        int index = performLinearSearch(array, elementToFind);
        if (index != -1) {
            System.out.println("Element " + elementToFind + " found at index " + index +
".");
        } else {
            System.out.println("Element " + elementToFind + " not found in the array.");
        }
    }
}
```

Explanation:

performLinearSearch method:

- It accepts an integer array arr and an integer element to search for.
- It iterates through each element of the array using a for loop.
- It compares each element of the array with the element parameter.
- If a match is found (arr[i] == element), it returns the index i.
- If no match is found after iterating through the entire array, it returns -1.

main method:

- This is where the function is demonstrated.
- An example array array is defined, and an elementToFind is set to 9.
- It calls performLinearSearch with array and elementToFind, and prints the result.

Example Output:
Element 9 found at index 5.

Task 2: Two-Sum Problem

a) **Given an array of integers, write a program that finds if there are two numbers that add up to a specific target. You may assume that each input would have exactly one solution, and you may not use the same element twice. Optimize the solution for time complexity.**

```
import java.util.*;

public class TwoSum {

    public static int[] findTwoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }
            map.put(nums[i], i);
        }

        throw new IllegalArgumentException("No two sum solution");
    }

    public static void main(String[] args) {
        int[] nums = { 2, 7, 11, 15 };
        int target = 9;
        int[] result = findTwoSum(nums, target);
        System.out.println("Indices: " + result[0] + ", " + result[1]);
    }
}
```

Explanation:
HashMap Approach:

- We use a single pass approach with a HashMap to store previously seen elements and their indices.
- For each element **nums[i**], compute **complement = target - nums[i].**
- Check if complement exists in the HashMap:

i. If it does, we found the pair and return the indices.

ii. If it doesn't, add nums[i] to the HashMap.

Time Complexity:

- The time complexity is O(n), where n is the number of elements in the array.
- This is because we traverse the list containing n elements only once

Space Complexity:
The space complexity is O(n) due to the extra space used by the HashMap.

Edge Cases:

- The solution handles the case when there are no solutions by throwing an exception, which is appropriate for the problem constraints .

For the input array nums = [2, 7, 11, 15] and target = 9:

The program will output Indices: 0, 1, since nums[0] + nums[1] = 2 + 7 = 9.

Task 3: Understanding Functions through Arrays

a) Write a recursive function named SumArray that calculates and returns the sum of elements in an array, demonstarte with example.

```
public class SumArray {

  public static int sumArray(int[] arr, int n) {
    if (n == 1) {
      return arr[0];
    } else {
      return arr[n-1] + sumArray(arr, n-1);
    }
  }

  public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 5};
    int sum = sumArray(arr, arr.length);
    System.out.println("Sum of array elements: " + sum);

  }
}
```

Explanation:

Recursive Function sumArray:

- The function **sumArray** takes an array **arr** and an integer n as parameters.
- n represents the number of elements to consider in the array.
- Base Case: If **n == 1**, the function returns **arr[0],** which is the sum of the single element in the array.
- Recursive Case: If **n > 1**, the function returns **arr[n-1] + sumArray(arr, n-1).**

i. arr[n-1] is the last element of the array.
ii. sumArray(arr, n-1) recursively calculates the sum of the first n-1 elements of the array.
iii. This effectively adds up all elements from arr[0] to arr[n-1].

Main Method:

- In the main method, an example array arr = {1, 2, 3, 4, 5} is used.
- sumArray(arr, arr.length) is called to calculate the sum of all elements in the array.
- The result is printed to the console.

Output:

The program outputs **Sum of array elements: 15.**

This is because **1 + 2 + 3 + 4 + 5 = 15**, which is the sum of all elements in the array.

Task 4: Advanced Array Operations

a) **Implement a method SliceArray that takes an array, a starting index, and an end index, then returns a new array containing the elements from the start to the end index.**

```
import java.util.Arrays;

public class SliceArray {

  public static int[] sliceArray(int[] arr, int start, int end) {
    if (start < 0 || start >= arr.length || end < start || end > arr.length) {
      throw new IllegalArgumentException("Invalid start or end indices");
    }

    int length = end - start;
    int[] result = new int[length];
    for (int i = 0; i < length; i++) {
      result[i] = arr[start + i];
```

```
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int start = 1;
        int end = 4;

        int[] slicedArray = sliceArray(arr, start, end);

        System.out.println("Original Array: " + Arrays.toString(arr));
        System.out.println("Sliced Array from index " + start + " to " + end + ": " + Arrays.toString(slicedArray));
    }
}
```

Explanation:
Method sliceArray:

- **sliceArray takes three parameters:**
- **arr:** The original array from which elements will be sliced.
- s**tart:** The starting index (inclusive).
- **end**: The ending index (exclusive).
- The method first checks if the provided start and end indices are valid:
- **start** should be within the bounds **[0, arr.length).**
- **end** should be within the bounds [**start, arr.length].**
- It calculates the length of the resulting array as **end - start.**
- It creates a new array **result** of the calculated length.
- It copies elements from the original array **arr** to the new array **result** starting from index start up to but not including index end.
- Finally, it returns the result array.

Main Method:

- In the main method, an example array **arr = {1, 2, 3, 4, 5**} is used.
- **start =** 1 and **end = 4** are specified to slice the array from index 1 to 3.
- The s**liceArray** method is called with these parameters, and the result is stored in **slicedArray.**
- The original and sliced arrays are printed using **Arrays.toString()** for clarity.

```
Output:
Original Array: [1, 2, 3, 4, 5]
Sliced Array from index 1 to 4: [2, 3, 4]
```

b) Create a recursive function to find the nth element of a Fibonacci sequence and store the first n elements in an array.

```
import java.util.Arrays;

public class Fibonacci {
  public static int fibonacci(int n) {
    if (n <= 0) {
      throw new IllegalArgumentException("n must be greater than zero");
    }

    if (n == 1) {
      return 0;
    } else if (n == 2) {
      return 1;
    } else {

      return fibonacci(n - 1) + fibonacci(n - 2);
    }
  }
  public static int[] fibonacciArray(int n) {
    if (n <= 0) {
      throw new IllegalArgumentException("n must be greater than zero");
    }

    int[] fibArray = new int[n];
          fibArray[i] = fibonacci(i + 1);
    }

    return fibArray;
  }

  public static void main(String[] args) {
    int n = 10;
        int nthFib = fibonacci(n);
    System.out.println("The " + n + "th Fibonacci number is: " + nthFib);

    int[] fibArray = fibonacciArray(n);
    System.out.println("The first " + n + " Fibonacci numbers are: " +
Arrays.toString(fibArray));
```

```
    }
}
```

Explanation:

Recursive Function fibonacci:

- fibonacci(int n) is a recursive function that calculates the nth Fibonacci number.

Base Cases:

- If n == 1, return 0 (the first Fibonacci number).
- If n == 2, return 1 (the second Fibonacci number).

Recursive Case:

- For n > 2, return the sum of the previous two Fibonacci numbers: fibonacci(n - 1) + fibonacci(n - 2).

Function fibonacciArray:

- fibonacciArray(int n) generates an array containing the first n Fibonacci numbers.
- It calls fibonacci(i + 1) for each index i to populate the array.

Main Method:

- In the main method, an example value n = 10 is used to find the 10th Fibonacci number and generate the first 10 Fibonacci numbers.
- The nth Fibonacci number and the first n Fibonacci numbers are printed to the console.

Output:

```
The 10th Fibonacci number is: 34
The first 10 Fibonacci numbers are: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```