Gandikota Murali

Gandikotamurali951@gmail.com

Assignment-9

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 Initialize Array.

```
import java.util.Arrays;
public class BruteForceSort {
 public static void main(String[] args) {
int[] arr = initializeArray(10);
System.out.println("Original Array: " + Arrays.toString(arr));
bruteForceSort(arr);
System.out.println("Sorted Array: " + Arrays.toString(arr));
}
 public static void bruteForceSort(int[] arr) {
 for (int i = 0; i < arr.len; i++) {
for (int j = 0; j < arr.len - 1; j++) {
 if (arr[j] > arr[j + 1]) {
 int temp = arr[j]; arr[j] = arr[j + 1];
 arr[j + 1] = temp;
}
 }
 }
 }
```

```
 public static int[] initializeArray(int size) {
 int[] arr = new int[size];
for (int i = 0; i < size; i++) {
 arr[i] = (int) (Math.random() * 100);
}
 return arr;
}
 }
```

Output:

Original Array: [71, 43, 58, 64, 53, 84, 31, 60, 56, 20]

Sorted Array: [20, 31, 43, 53, 56, 58, 60, 64, 71, 84]

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 target) {
for (int i = 0; i < arr.length; i++) {
if (arr[i] == target) {
 return i;
}
 }
 return -1;
 }
public static void main(String[] args) {
 int[] array = {3, 6, 9, 12, 15, 18};
 int target = 15;
```

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

Output:

Element found at index 4

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.HashSet;

public class TwoSum {

   public static boolean hasTwoSum(int[] nums, int target) {
      HashSet<Integer> set = new HashSet<>();

      for (int num : nums) {
         if (set.contains(target - num)) {
            return true;
         }
         set.add(num);
      }

      return false;
   }
```

```
    public static void main(String[] args) {
        int[] nums = {2, 7, 11, 15};
        int target = 9;

        if (hasTwoSum(nums, target)) {
            System.out.println("Two numbers add up to the target");
        } else {
            System.out.println("No two numbers add up to the target");
        }
    }
}
```

Output:

Two numbers add up to the target.

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 SumArrayExample {


// Recursive function to calculate the sum of array elements
public static int SumArray(int[] array, int index) {
    // Base case: if index is out of bounds, return 0
    if (index >= array.length) {
        return 0;
    }
    // Recursive case: return current element + sum of remaining elements
    return array[index] + SumArray(array, index + 1);
}
```

```
    public static void main(String[] args) {
        // Example array
int[] exampleArray = {1, 2, 3, 4, 5};

        // Calculate the sum of the array elements
        int sum = SumArray(exampleArray, 0);

        // Print the result
System.out.println("Sum of array elements: " + sum);
    }
}
```

Explanation

1.Base Case:

•The base case for our recursive function is when the index is greater than or equal to the length of the array. When this condition is met, the function returns 0 because there are no more elements to add.

2.Recursive Case:

•For the recursive case, the function returns the current element at array[index] plus the result of the function call with the next index (index + 1).

3.Example Execution:

•Suppose exampleArray is {1, 2, 3, 4, 5}.

•The initial call is SumArray(exampleArray, 0).

•The first call checks if index >= array.length. Since index is 0 and the array length is 5, it proceeds to return array[0] (which is 1) plus SumArray(exampleArray, 1).

•The next call is SumArray(exampleArray, 1), which returns array[1] (2) plus SumArray(exampleArray, 2), and so on.

•This continues until index reaches 5, at which point SumArray(exampleArray, 5) returns 0.

•The calls then return their results back up the call stack, summing the elements as they go:

•SumArray(exampleArray, 4) returns 5 + 0 = 5

•SumArray(exampleArray, 3) returns 4 + 5 = 9

•SumArray(exampleArray, 2) returns 3 + 9 = 12

•SumArray(exampleArray, 1) returns 2 + 12 = 14

•SumArray(exampleArray, 0) returns 1 + 14 = 15

Conclusion

This example demonstrates how to use recursion to sum the elements of an array. The key points to understand are defining a clear base case to stop the recursion and ensuring each recursive call processes the next element in the array. The function builds up the sum by adding each element in the array to the sum of the elements that follow it.

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 ArraySlicer {

    public static int[] SliceArray(int[] array, int startIndex, int endIndex) {

        if (startIndex < 0 || endIndex > array.length || startIndex > endIndex) {

            return new int[0];

        }


        int length = endIndex - startIndex;

        int[] result = new int[length];
```

```
        System.arraycopy(array, startIndex, result, 0, length);


        return result;
    }


    public static void main(String[] args) {
        int[] myArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int[] slicedArray = SliceArray(myArray, 2, 5);
        System.out.println(Arrays.toString(slicedArray));
    }
}
```

Output:

[3,4,5]

Explanation :

In this implementation, we aim to create a method called **`SliceArray`** that extracts a portion of an array from a given starting index to an ending index. The method takes three parameters: the original array, the starting index, and the ending index. We first validate the provided indices to ensure they are within the valid range. This includes checking if the starting index is non-negative, the ending index does not exceed the length of the array, and the starting index is less than the ending index. If any of these conditions fail, the method returns an empty array, indicating invalid input.

Once the indices are validated, we calculate the length of the new sub-array, which is the difference between the ending index and the starting index. This length determines how many elements will be included in the new array. We then create a new array called **`result`** with this calculated length to hold the sliced elements. Using a **`for`** loop, we copy the elements from the original array to the new array. The loop iterates from 0 to the length of the new array, and in each iteration, it assigns the element at the current index from the original array (starting from the starting index) to the corresponding position in the new array.

After copying the necessary elements, the method returns the new array containing the sliced portion of the original array. This approach ensures that the slicing operation is performed efficiently and correctly, providing a robust solution for array slicing in Java.

To demonstrate the usage of this method, we include a **`main`** method where we define an example array and call the **`SliceArray`** method with specific start and end indices. The resulting sub-array is then printed to the console, showing the elements that fall within the specified range. This implementation highlights key programming concepts such as input validation, array manipulation, and efficient copying of elements, making it a practical example of handling array operations in Java.

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

Program:

```
import java.util.Arrays;

public class Fibonacci {

   public static int fibonacciRec(int n) {

      if (n <= 1) {

         return n;

      }

      return fibonacciRec(n - 1) + fibonacciRec(n - 2);

   }

   public static int[] generateFibonacciSeries(int n) {

      int[] fibSeries = new int[n];

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

         fibSeries[i] = fibonacciRec(i);

      }

      return fibSeries;

   }
```

```
    public static void main(String[] args) {
        int n = 10;
        int[] fibonacciArray = generateFibonacciSeries(n);
        System.out.println("The first " + n + " elements of the Fibonacci sequence are:");
        System.out.println(Arrays.toString(fibonacciArray));
    }
}
```

Output:

The first 10 elements of the Fibonacci sequence are:

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]