W7 - Data Structure & Algorithms

Problem 1: Sorting Large Arrays with Merge Sort and Quick Sort

Overview: Implement Merge Sort and Quick Sort algorithms to sort an array of 1,000,000 random integers. Compare the running times of both algorithms. Discuss the practicality of using Selection Sort or Bubble Sort for sorting an array of this size.

Merge Sort Algorithm:

- **Divide and Conquer Approach**: Recursively divide the array into halves, sort each half, and then merge them.
- **Stable and Efficient for Large Data Sets**: Particularly efficient for sorting large arrays.

Quick Sort Algorithm:

- **Divide and Conquer with a Pivot**: Choose a pivot element and partition the array around the pivot. Recursively apply the same logic to the partitions.
- **Generally Faster, but Less Stable**: Often faster than Merge Sort, but can degrade to O(n^2) in the worst case.

Algorithm Implementations in Java:

1. **Merge Sort**:

```
public class MergeSort {

    public static void sort(int[] array) {
        if (array.length < 2) {
            return;
        }
        int mid = array.length / 2;
        int[] left = new int[mid];
        int[] right = new int[array.length - mid];

        System.arraycopy(array, 0, left, 0, mid);
        System.arraycopy(array, mid, right, 0, array.length - mid);

        sort(left);
        sort(right);

        merge(array, left, right);
    }

    private static void merge(int[] array, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                array[k++] = left[i++];
            } else {
                array[k++] = right[j++];
            }
        }
        while (i < left.length) {
            array[k++] = left[i++];
        }
        while (j < right.length) {
            array[k++] = right[j++];
        }
    }
}
```

2. **Quick Sort**:

```
public class QuickSort {

   public static void sort(int[] array) {
       quickSort(array, 0, array.length - 1);
   }

   private static void quickSort(int[] array, int start, int end) {
       if (start < end) {
           int partitionIndex = partition(array, start, end);
           quickSort(array, start, partitionIndex - 1);
           quickSort(array, partitionIndex + 1, end);
       }
   }

   private static int partition(int[] array, int start, int end) {
       int pivot = array[end];
       int i = (start - 1);

       for (int j = start; j < end; j++) {
           if (array[j] <= pivot) {
               i++;
               int temp = array[i];
               array[i] = array[j];
               array[j] = temp;
           }
       }

       int temp = array[i + 1];
       array[i + 1] = array[end];
       array[end] = temp;
       return i + 1;
   }
}
```

3. **Generating and Sorting the Array**:
    - **Generate Random Array**: Create an array of 1,000,000 random integers.
    - **Sort Using Both Algorithms**: Use Merge Sort and Quick Sort to sort the array.
    - **Measure Running Times**: Compare the time taken by each algorithm to sort the array.

```
import java.util.Random;

public class SortingComparison {
   public static void main(String[] args) {
       int[] array = new int[1000000];
       Random random = new Random();
       // Fill the array with random integers
       for (int i = 0; i < array.length; i++) {
           array[i] = random.nextInt(Integer.MAX_VALUE);
       }
       // Copy the array for fair comparison
       int[] arrayForMergeSort = array.clone();
       int[] arrayForQuickSort = array.clone();
       // Measure time for Merge Sort
       long startTime = System.currentTimeMillis();
       MergeSort.sort(arrayForMergeSort);
       long endTime = System.currentTimeMillis();
       System.out.println("Merge Sort Time: " + (endTime - startTime) + " ms");
```

```
        // Measure time for Quick Sort
        startTime = System.currentTimeMillis();
        QuickSort.sort(arrayForQuickSort);
        endTime = System.currentTimeMillis();
        System.out.println("Quick Sort Time: " + (endTime - startTime) + " ms");
    }
}
```

Problem 2

Problem 2A: Convert Sorted Array to Balanced Binary Search Tree (BST)

Overview: Given a sorted array, the task is to convert it into a height-balanced BST. In a balanced BST, the depth of the two subtrees of every node never differs by more than one.

Algorithm Steps for Problem 2A:

1. **Find Middle**: Identify the middle element of the array.
2. **Create Node**: The middle element becomes the root of the BST.
3. **Recursive Structure**: Recursively apply the above steps to the left and right halves of the array to create left and right subtrees.
4. **Code**:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}

public class SortedArrayToBST {
    public static TreeNode sortedArrayToBST(int[] nums) {
        if (nums == null || nums.length == 0) {
            return null;
        }
        return constructBSTRecursive(nums, 0, nums.length - 1);
    }

    private static TreeNode constructBSTRecursive(int[] nums, int left, int right) {
        if (left > right) {
            return null;
        }
        int mid = left + (right - left) / 2;
        TreeNode node = new TreeNode(nums[mid]);
        node.left = constructBSTRecursive(nums, left, mid - 1);
        node.right = constructBSTRecursive(nums, mid + 1, right);
        return node;
    }
}
```

Problem 2B: Convert Sorted Linked List to Balanced Binary Search Tree

Overview: Convert a sorted linked list to a balanced BST. The challenge is to do this without converting the linked list to an array first.

Algorithm Steps:

1. **Find Middle Element**: Use the "slow and fast pointer" technique to find the middle of the linked list.

2. **Recursive Construction**:
    - Use the middle node as the root.
    - Recursively build the left subtree from the left half of the list.
    - Recursively build the right subtree from the right half of the list.
3. **Code**:

```
class ListNode {
   int val;
   ListNode next;

   ListNode(int x) {
       val = x;
   }
}

class SortedListToBST {
   public static TreeNode sortedListToBST(ListNode head) {
       if (head == null) {
           return null;
       }
       return constructBSTRecursive(head, null);
   }

   private static TreeNode constructBSTRecursive(ListNode start, ListNode end) {
       if (start == end) {
           return null;
       }

       ListNode slow = start;
       ListNode fast = start;
       while (fast != end && fast.next != end) {
           slow = slow.next;
           fast = fast.next.next;
       }

       TreeNode node = new TreeNode(slow.val);
       node.left = constructBSTRecursive(start, slow);
       node.right = constructBSTRecursive(slow.next, end);
       return node;
   }
}
```

Problem 3: Counting Inversions in an Array Using Merge Sort

Overview: Given an array of integers `A`, count the number of inversions, where two elements `A[i]` and `A[j]` form an inversion if `i < j` and `A[i] > A[j]`. This problem is solved efficiently using a divide and conquer approach, similar to Merge Sort.

Divide and Conquer Approach:

- **Divide**: Split the array into two halves.
- **Conquer**: Recursively count inversions in both halves.
- **Combine**: During the merge step of Merge Sort, count the inversions where elements from the left half are greater than elements from the right half.

Algorithm Steps:

1. **Divide the Array**: Split the array into two halves.
2. **Recursively Count Inversions**: Recursively count inversions in the left half and the right half of the array.

3. **Count and Merge**: During the merge phase, count the inversions where elements in the left half are greater than those in the right half. Then, merge the two halves as in Merge Sort.
4. **Code**:

```
public class InversionCount {

   // Method to use mergeSort to count inversions
   static int countInversions(int[] array) {
       int[] temp = array.clone();
       return mergeSortAndCount(array, temp, 0, array.length - 1);
   }

   // Merge Sort and Count function
   private static int mergeSortAndCount(int[] array, int[] temp, int left, int right) {
       int mid, inv_count = 0;
       if (right > left) {
           // Divide the array into two parts
           mid = (right + left) / 2;

           // Count inversions in the left and right parts
           inv_count += mergeSortAndCount(array, temp, left, mid);
           inv_count += mergeSortAndCount(array, temp, mid + 1, right);

           // Merge the two parts and count the inversions during merge
           inv_count += mergeAndCount(array, temp, left, mid + 1, right);
       }
       return inv_count;
   }

   // Merge and count function
   private static int mergeAndCount(int[] array, int[] temp, int left, int mid, int right) {
       int i, j, k;
       int inv_count = 0;

       i = left;    // Initial index of left subarray
       j = mid;     // Initial index of right subarray
       k = left;    // Initial index of merged subarray

       while ((i <= mid - 1) && (j <= right)) {
           if (array[i] <= array[j]) {
               temp[k++] = array[i++];
           } else {
               temp[k++] = array[j++];

               // Inversion will occur
               inv_count = inv_count + (mid - i);
           }
       }

       // Copy remaining elements of left subarray
       while (i <= mid - 1) {
           temp[k++] = array[i++];
       }

       // Copy remaining elements of right subarray
       while (j <= right) {
           temp[k++] = array[j++];
       }
```

```
        // Copy back the merged elements to the original array
        for (i = left; i <= right; i++) {
            array[i] = temp[i];
        }
        return inv_count;
    }

    public static void main(String[] args) {
        int[] arr = { 1, 20, 6, 4, 5 };
        System.out.println("Number of inversions are: " + countInversions(arr));
    }
}
```