Tutorial 2

Problem 1

a) niceFunction:

The function contains a for loop that runs n times. Each iteration of the loop takes a constant amount of time, and the time taken for each iteration does not depend on the value of n.

Since the number of iterations is proportional to the value of n, the time complexity of this function is O(n).

d) suspiciousFunction:

The reason for this is that the function is calling itself twice, once with n/2 and once with n/2, and each time the size of n is halved. This means that each time the function is called, the number of operations to be performed is reduced by half.

In terms of complexity, this is similar to the divide and conquer approach used in algorithms such as binary search and merge sort, which have a time complexity of O(log n).

Since the function calls itself twice each time, the time complexity of this function is O(n log n).

e) badFunction:

The function calls itself twice with inputs n-1 and n-2, which means that the number of calls to the function grows rapidly as n increases. Each call to the function performs a constant amount of work, which in this case is adding 0.

$T(n) \leq 2T(n -1) + O(1)$

1 + 2 + 3 +4 + 5 + … 2^n = O(2 ^n) as 2 ^n is twice as large as the number before.

Tighter bound : O(ϕ^n)

The number of calls to the function grows exponentially with n, as seen in the Fibonacci sequence. This means that the time complexity is O(2^n), as the number of operations performed is proportional to the exponential growth of the number of calls to the function.

f)

O(n^2)

1 + 2 + 3 + 4 + … (n -1)

Note: sb.append(”?”) takes O(1) time. Its like adding to an array.

Use string builder

Problem 2: Sorting Review

a)

```
import java.util.Arrays;

public class insertionSort {

    private int[] targetArr;
    private int arrLength;

    public insertionSort (int n, int[] array) {
        targetArr = new int[n];
        targetArr = array;
        arrLength = n;
    }
    //n is the length of the array.
    private void sortArray (int n) {
        if (n > 1) {
            // Recurse to sort first n-1 elements
            sortArray(n-1);

            // Insert last element (the one to sort) at its correct position in sorted array.
            int elementToSort = targetArr[n-1];
            int j = n-2; //element no. n-1

            // While loop to help place the target element in its correct place
            //Relies on the loop invariant that the first n-1 elements are already sorted.
            while (j >= 0 && targetArr[j] > elementToSort)
            {
                targetArr[j+1] = targetArr[j];
                j--;
            }
            targetArr[j+1] = elementToSort;
        }

    }

    public static void main(String[] args) {
        int[] inputArr = {13, 22, 3, 4, 1, 341, 324, 900};
        int n = inputArr.length;
        insertionSort testSort = new insertionSort(n, inputArr);
        testSort.sortArray(testSort.arrLength);
        System.out.println(Arrays.toString(testSort.targetArr));
    }
}
```

Recursion relation = T(n) = T(n-1) + O(n) = O(n^2)

b)

First, we use SelectionSort to sort the tied pairs based on b. Since the length of an array of tied pairs is at most the length of the main array, SelectionSort can deal more effectively with smaller arrays.

Then, use MergeSort to sort the pairs based on a. This is because we are unaware of how many pairs will be in the array, and we must sort all pairs. MergeSort is more efficient when dealing with large arrays than SelectionSort.

MergeSort is stable and preserves the ordering of b (which is already sorted when a has the same value. So we sort b first.

c)

```
import java.util.Arrays;

public class iterativeMergeSort {
    int[] targetArr;
    int n;

    public iterativeMergeSort (int[] targetArr){
        this.targetArr = targetArr;
        this.n = targetArr.length;
    }

    private void sort() {
        // for each size of sub-array
        for (int subArrSize = 1; subArrSize < n; subArrSize *= 2) { //*= 2 because it is merge sort
            // for each sub-array of size 'size'
            for (int begin = 0; begin < n - subArrSize; begin += subArrSize * 2) {
                // merge the sub-array from 'start' to 'start + size - 1'
                // with the sub-array from 'start + size' to 'Math.min(start + size * 2 - 1, n - 1)'
                merge(begin, begin + subArrSize - 1, Math.min(begin + subArrSize * 2 - 1, n - 1));
            }
        }
    }
    private void merge(int begin, int mid, int end) {
        int[] tempArr = new int[end - begin + 1];

        int i = begin, j = mid + 1, k = 0;
        while (i <= mid && j <= end) {
            if (targetArr[i] <= targetArr[j]) {
                tempArr[k++] = targetArr[i++];
            } else {
                tempArr[k++] = targetArr[j++];
            }
        }

        // copy the remaining elements of left sub-array
        while (i <= mid) {
            tempArr[k++] = targetArr[i++];
        }

        // copy the remaining elements of right sub-array
        while (j <= end) {
            tempArr[k++] = targetArr[j++];
        }

        // copy the sorted temp array to the original array
        for (int p = 0; p < tempArr.length; p++) {
            targetArr[begin + p] = tempArr[p];
        }
    }

    public static void main(String[] args) {
        int[] input = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

        iterativeMergeSort testSort = new iterativeMergeSort(input);
        testSort.sort();

        System.out.println(Arrays.toString(testSort.targetArr));
    }
}
```

The space complexity of the iterative implementation of MergeSort is O(n), where n is the number of elements in the input array. This is because the iterative approach uses an array of size n to store the sorted elements during each iteration, as well as to store the target array.

The time complexity of the iterative implementation of MergeSort is O(n * log n), where n is the number of elements in the input array. This is because the algorithm splits the input array into smaller sub-arrays and sorts them in a bottom-up manner, combining them into a single sorted array. The time complexity of each step of the merging process is proportional to the size of the sub-arrays being merged, which is O(n). The total number of merging steps is log n, so the overall time complexity is O(n * log n).

Problem 3

a)

```
class Stack {
    private int[] stackArray;
    private int top;
    private int size;

    public Stack(int size) {
        stackArray = new int[size];
        top = -1;
        this.size = size;
    }

    public void push(int value) {
        if (top != size - 1) {
            top++;
            stackArray[top] = value;
        } else {
            System.out.println("Stack full");
        }
    }

    public int pop() {
        if (top != -1) {
            int poppedValue = stackArray[top];
            top--;
            return poppedValue;

        } else {
            System.out.println("Stack is empty");
            return -1;
        }

    }

    public int peek() {
        if (top == -1) {
            System.out.println("Stack is empty");
            return -1;
        }
        return stackArray[top];
    }

    boolean isEmpty() {
        return top == -1;
    }
}
```

```
class Queue {
    private int[] queueArray;
    private int start;
    private int end;
    private int size;

    public Queue(int size) {
        queueArray = new int[size];
        start = 0;
        end = -1;
        this.size = size;
    }

    private void enqueue(int value) {
        if (end == size - 1) {
            end++;
            queueArray[end] = value;
        } else {
            System.out.println("Queue is full");
        }
    }

    private int dequeue() {
        if (start > end) {
            System.out.println("Queue is empty");
            return -1;
        }
        int dequeuedValue = queueArray[start];
        start++;
        return dequeuedValue;
    }

    private int peek() {
        if (start > end) {
            System.out.println("Queue is empty");
            return -1;
        }
        return queueArray[start];
    }

}
```

For queue: wrong: you need to wrap around as you will run out of space

b)

Dequ is used to undo/redo function, website history, rate limiter (prevent ddos).

Implement it similarly to a queue. However, the variable “start” now points at the middle element of the fixed array. When items are added to the front, start-=1 and the item is placed in that slot.

c) Check if the maximum size of the array has been met when we try to assign items to the array, check if array is empty when we remove/ peek at items,

d)

Can also use variable to store open parenthesis (keep count)

```
public class checkBalance {
    public static boolean isBalanced(String input) {
        int inputLength = input.length();
        Stack targetStack = new Stack(inputLength);
        for (int i = 0; i < inputLength; i++) {
            char c = input.charAt(i);
            if (c == '(') {
                targetStack.push(c);
            } else if (c == ')') {
                if (targetStack.isEmpty()) {
                    return false;
                } else {
                    targetStack.pop(); // If we find a ) can just rem the ( in the stack to signify that it is cancelled out
                }
            }
        }
        return targetStack.isEmpty();
    }

    public static void main(String[] args) {
        String input = ")(())(";
        System.out.println(isBalanced(input));
    }
}
```

Problem 4:

Maintain a decreasing monotonic stack.

```
public class stacNCue {
    public static int evacuation(int[] houses) {
        int noOfHouses = houses.length;
        Stack trackingStack = new Stack(noOfHouses);
        int flooded = 0;

        // iterate through the houses
        for (int i = 0; i < noOfHouses; i++) {
            //Decides whether we should start popping
            boolean shouldPop = false;
            //Checks if a house of elevation similar or larger than current has appeared
            for (int p = 0; p < i; p ++ ) {
                if (houses[p] >= houses[i]) {
                    shouldPop = true;
                }
            }
            // pop the indexes from the stack until the current house is equal or higher
            // peek returns the elevation of the previous house added
            while (!trackingStack.isEmpty() && trackingStack.peek() < houses[i] && shouldPop) {
                    trackingStack.pop();
                    flooded++;
            }
            trackingStack.push(houses[i]);
        }

        return flooded;
    }

    public static void main(String[] args) {
        System.out.println("We need to evacuate " + evacuation(new int[]{1, 0, 1, 2, 3, 1, 2, 3, 0}) + " houses");
    }
}
```

5) Just like merge sort iterative : sort 1, then 2, then 4, then 8 etc.