**CMPSC122 In-Lab 12– Algorithm Analysis**

**Part I. In-Lab Exercise (**10 points)

Submit the solution files of Exercises online before the due date/time. Upload the filled-in InLab12.docx file.

1. Write the big-O estimation of the following functions, using the two-step simplification method we talked about in the lecture.

100n2 + 4000\*n + 50\*log(n) = O(n^2)

20\*log(n) + 500  =O(log(n))

(n+3)2 = O(n^2)

2. In the following, you are given several functions. Your task is to analyze the running time of these functions. You should count how many times each line would be executed, and you add these numbers together to get the running time of the function. Finally, you will use the big-O notation to estimate the algorithm performance (i.e. how fast the running time functions would grow).

The following is an example of analyzing a function line by line. At the end of each line, the number of times the line gets executed is counted. At the end of the function, I add all these numbers together, which is the running time of the function:

**int testPalindrome(char\* targetString, int n) {**

**bool mismatchFound = false; // # of executions: 1**

**for (int i = 0; i < n / 2; i++) { // # of executions: n/2**

**if (targetString[i] != targetString[n - i - 1]) // # of executions: n/2**

**mismatchFound = true; // # of executions: Min:0, Max:n/2**

**}**

**return !mismatchFound; // # of executions: 1**

**}**

**// Min (Best case) Running Time: 1 + n/2 + n/2 + 1 = n+2**

**// Max (Worst case) Running Time: 1 + n/2 + n/2 + n/2 + 1 = (3n/2) + 2**

**// Max (Worst case) Running Time Big-O: O(n)**

Note that, one of the lines above (highlighted green) is special. The number of times it gets executed is not deterministic. It could be executed 0 times if the **if** statement condition is never true, or it could be executed n/2 at max if the if statement condition is always true. In this case, we need to find out the min/max number of times it gets executed. In the end, when we sum up all the numbers, we also need to calculate the min/max total running time (which formally is called the best case/worst case running time). Then we use big-O to estimate the max (worst case) running time.

Your task: analyze the following two functions using the method demonstrated above. Fill in the blanks.

Function #1:

**int findMax(int arr[], int n) {**

**int curMax = arr[0]; // # of executions: \_\_\_\_\_\_1\_\_\_\_\_\_**

**for (int i = 1; i < n; i++) { // # of executions: \_\_\_\_\_\_n-1\_\_\_\_\_\_**

**if (curMax < arr[i]) // # of executions: \_\_\_\_\_Min: 0 Max: n-1\_\_\_\_\_\_\_**

**curMax = arr[i]; // # of executions: Min:\_\_\_0\_\_\_\_\_, Max:\_\_\_\_\_n-1\_\_\_**

**}**

**return curMax; // # of executions: \_\_\_\_1\_\_\_\_\_\_\_\_**

**}**

**// Min (Best case) Running Time: \_\_** **1 + (n - 1) + 1 = n + 1\_\_\_\_\_\_\_\_\_\_\_\_**

**// Max (Worst case) Running Time: \_\_\_\_\_** **1 + (n - 1) + (n - 1) + 1 = 2n - 1\_\_\_\_\_\_\_\_**

**// Max (Worst case) Running Time Big-O: \_\_\_\_** **O(n)\_\_\_**

Function #2:

**int maxPair(int arr[], int n) {**

**int curMax = INT\_MIN; // # of executions: \_\_\_\_\_1\_\_\_\_\_\_\_**

**for (int i = 0; i < n; i++) { // # of executions: \_\_\_\_\_\_n\_\_\_\_\_\_**

**for (int j = i + 1; j < n; j++) { // # of executions: \_\_\_(n-1)**+**(n-2) + ... + 1 = (n-1)(n)/2**

**int sum = arr[i] + arr[j]; // # of executions: (n-1)(n)/2**

**if (sum > curMax) // # of executions: Min:0 Max: (n-1)(n)/2**

**curMax = sum; // # of executions: Min:0, Max:(n-1)(n)/2\_\_\_\_\_\_\_\_**

**}**

**}**

**return curMax; // # of executions: \_\_\_\_1\_\_\_\_\_\_\_\_**

**}**

**// Min (Best case) Running Time: 1 + n + (n-1)(n)/2 + 1 = (n^2 + 3n + 2)/2**

**// Max (Worst case) Running Time: 1 + n + (n-1)(n)/2 + (n-1)(n)/2 + 1 = n^2 + 3n + 1**

**// Max (Worst case) Running Time Big-O: O(n^2)**