COMP 2230 - Data Structure, Algorithm Analysis, and Program Design

Lab/Homework No. 4

Total: 50 Marks

Due date and time for submission: midnight of Sunday, October 19, 2019.
Method of submission: Moodle.tru.ca

Algorithm Complexity Analysis

This is an "experimental" lab in which you will experiment with codes to understand algorithm complexities. The objective is to write programs with code snippets of different time complexities and test how long each code snippet takes to run on your machine for different values of input n.

Please do not use "wall-clock" time, instead you can use the following code template to obtain the execution time. The reason is that you are not comparing your execution times with others. Rather, it is an experiment to see how the function grows for different values of the input size on your own machine.

```
long startTime, endTime, executionTime;
startTime = System.currentTimeMillis();

//include the code snippet (or call to the method) here

endTime = System.currentTimeMillis();
executionTime = endTime - startTime;

//print executionTime
```

The above segment will give the time for executing the code segment in milliseconds.

Exercise 1.

a) **[10 Marks**] Write a program to implement the **bubble sort, insertion sort, selection sort, merge sort and quick sort** algorithms (*you may use the class codes that I have uploaded in Moodle, but make sure your comment those codes and fix any errors that might be there*). One way is to create separate methods for each sorting algorithms. The sorting algorithms should take an array of size *n* and sort them in ascending order. For example, the bubble sort algorithm takes an array of size *n* and sorts them it in *(n-1)* passes, making (n-1) comparisons in each pass.

b) **[5 Marks]** Write the complexity for each of the sorting algorithms in the following table:

Algorithm	**Worst Case Complexity**	**Average Case Complexity**
Bubble Sort		
Insertion Sort		
Selection Sort		
Merge Sort		
Quick Sort		

c) **[10 Marks]** Write a driver program with main method that creates an array of integers of size *n*, where *n* is specified by the user. Fill the array with random integers.
Note: `Math.random()` generates a random number in the range 0.0 and 1.0. If you want a random integer within a specific range, say 1 to 500, here's how you do it:
int num = (int)(Math.random()*500) + 1;

Call the sorting method to sort the array using different sorting algorithms (all five), and record the execution time of each algorithms.

Test the execution time for different values of *n* (say n = 10000, 50000, 100000, 200000, … 1000,000)

For each algorithm, record your values in a table shown below:

i. Bubble Sort

Degree of polynomial (n)	Evaluation time (millisecs)

ii. Insertion Sort

Degree of polynomial (n)	Evaluation time (millisecs)

iii. Selection Sort

Degree of polynomial (n)	Evaluation time (millisecs)

iv. Merge Sort

Degree of polynomial (n)	Evaluation time (millisecs)

v. Quick Sort

Degree of polynomial (n)	Evaluation time (millisecs)

Using the above tables, draw graphs of the evaluation time (y axis) vs input size *n* (x axis).
You may notice that for small values of n (even up to 10000) the execution time is close to 0 millisecs. Also, the times may vary depending upon individual computer speeds, the network speed, etc. You need to generate results with enough data points to show the growth curve.

d) **[5 Marks]** Now, analyze the graph and write a short report (500 words) based on your observation of comparing the algorithms.

Exercise 2. [10 Marks] The following code snippet has an algorithm time complexity of $O(n^3)$.

Write a method to multiply two matrices. The header of the method is as follows:

```
public static double [][] multiplyMatrix(double[][] a, double[][] b)
```

Assume that the two input matrices are square (that is, they are *n* x *n* matrices).
To multiply matrix a by matrix b, where c is the result matrix, use the formula:

```
a11  a12  a13          b11  b12  b13          c11  c12  c13
a21  a22  a23     X    b21  b22  b23     =    c21  c22  c23
a31  a32  a33          b31  b32  b33          c31  c32  c33
```

where `cij = ai1 X b1j + ai2 X b2j + ai3 X b3j`

Since we are only interested in the execution time, you may assume that all the elements of matrices a and b are identical. A sample screen dialog and outputs are given below:

```
Enter the size of each matrix: 100
Enter the matrix element (all elements of the matrices are assumed to be
the same): 3.5

Execution time: 16 millisecs
```

The template of the program with the main method is given below. It includes the code segment to get the execution time. Fill in the method to compute the product of the two matrices.

```
//Multiplication of two square matrices of size n X n each
import java.util.Scanner;

public class MatrixMult {
  /** Main method */
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
      int n;
    double num;

    System.out.print("Enter the size of each matrix: ");
    n = keyboard.nextInt();
    System.out.println("Enter the matrix element");
    System.out.print("All elements of the matrices are assumed to be the
same: ");
    num = keyboard.nextDouble();

    double[][] matrix1 = new double[n][n];
    for (int i = 0; i < n; i++)
```

```
      for (int j = 0; j < n; j++)
        matrix1[i][j] = num;

    double[][] matrix2 = new double[n][n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        matrix2[i][j] = num;


    long startTime, endTime, executionTime;
      startTime = System.currentTimeMillis();

      double[][] resultMatrix = multiplyMatrix(matrix1, matrix2);

      endTime = System.currentTimeMillis();
      executionTime = endTime - startTime;

      System.out.println("Execution time: " + executionTime + "
millisecs");

  }

  /** The method for multiplying two matrices */
  public static double[][] multiplyMatrix(double[][] m1, double[][] m2)
  {
      //include your code here
  }


}
```

Test the execution time for different values of n (say n = 100, 200, 300, …. 1000)

Record your values in a table

Size of matrix (n)	Multiplication time (millisecs)

Draw a graph of execution time vs n.

Again, use appropriate data sizes in order to generate the curve.
You may notice that if n exceeds 1500 or 2000, the execution time takes longer and longer. You may also get an error such as:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at MatrixMult.multiplyMatrix(MatrixMult.java:45)
    at MatrixMult.main(MatrixMult.java:32)
```

Explore why this happens (you can use web resources for this) and write a brief answer.

Exercise 3. **[10 Marks]** The third code has an exponential time complexity (2^n), which means that it grows very fast even for small values of n.

Write a code that generates 2 power n binary numbers for a given value of n.
For example, if n is 3, the binary numbers that are generated are 000, 001, 010, 011, 100, 101, 110, 111 (which are equivalent to 0 to 7 in decimal).

A sample screen dialog and output are given below:

```
Enter the value of n: 17
Execution time to generate 2^17 binary numbers: 32 millisecs
```

Note:
`(int)Math.pow(2,n)` gives 2 power n as an integer.

```
String sb = Integer.toBinaryString(i);
```

converts an integer i into a binary number.

Note: You need not print the binary numbers – you just need to generate them and measure the execution time.

Therefore, all you need to do in the program is to call the above statement
`String sb = Integer.toBinaryString(i);`
2 power n times using a for loop (the loop goes from 0 to $2^n - 1$

Test the execution time for different values of n (say n = 10, 11, 15, 17, 20, etc.). *Word of caution:* The program will take a really long time to complete for values of n even larger than 20 or 30. If the program doesn't end in a few minutes, just report that it as **"hangs"**.

Record your values in a table

Value of n	Time to generate 2 power n binary numbers (millisecs)

Draw the graphs.

Grading Rubric

1. Code compiled without any error	*20%*
2. Good Comments	*20%*
3. All requirements are met	*40%*
4. Screenshots of several test runs are included	*10%*
5. Apply OOP principals • *Modularity* • *Good use of classes* • *Encapsulation*	*10%*

Note: All problems will be graded based on the above-mentioned rubric.