2024

Assignment 1

COMP 2230_02

COLTON ISLES & KAYLEE CROCKER

THOMPSON RIVERS UNIVERSITY

COMP 2230 – Data Structures and Algorithm Analysis

Assignment #1: Analysis of Algorithms

Due Date: Beginning of your Seminar on September 12th, 2024

Assignment:

This first assignment is an analysis of algorithms and understanding growth functions using Big-Oh notation.

1. Explain what O(1) means.

   O (1) indicates a constant execution time regardless of the function size. The function will always run with the same execution time. Ex. Basic mathematic functions (+ - * /), a + b, a / b, etc.

2. If the Big-O calculation for a code segment is $O(2^n)$, what do you conclude about the code segment? Why?

   The segment execution grows exponentially with the larger input. As n increases, the time for execution grows exponentially. Small increases in input(n) would cause substantial time increase. This code is not efficient.

3. How many times will the following code print out Hello World (in terms of the length of the array)? Do the Big O and then check your results by writing code.

```
for(int i = 0; i < array.length; i++) {
        for(int j = 0; j < array.length; j++) {
                for(int k = 0; k < array.length; k++) {
                        System.out.println("Hello World!");
                }
        }
}
```

“Hello World” will print n^3 times with n being the length of the array. Therefore, the efficiency is O(n^3) (f(x) = n*n*n). Array of size 1 prints once and executes in 125700 nanoseconds. Array of size 2 prints 8 times and executes in 209200 nanoseconds. Size 3 prints 27 times and executes in 449300 nanoseconds. Size 10 prints 1000 times and executes in 9231300 nanoseconds.

4. What is the complexity of the following code (in terms of the length of the array), assuming someMethod has a complexity of O(1)? Do the Big O and then check your results by writing code.

```
for(int i = 0; i < array.length; i++)
```

```
                for(int j = 0; j < array.length; j++){
                        someMethod(array[j]);
        }
}
```

The complexity is O(n^2) (f(x) = 1+n^2). size 2 prints 4 times in 159700 nanosecond, size 3 prints 9 time in 207800 nanoseconds. Size 10 prints 100 time in 976100 nanoseconds.

5. What is the complexity of the following code (in terms of the length of the array)? Do the Big O and then check your results by writing code.

```
        for(int i = 0; i < 5; i++)
                System.out.println(array[i]);
```

The complexity is O(1) (f(x) = 1). The loop always loops 5 times and therefore is constant at 5. Always prints 5 times at 153100 nanoseconds.

6. Why do we do algorithm analysis to compare algorithms?
   We do algorithm analysis to compare algorithms so we can determine which algorithms are more efficient over others and predict overall performance of a program
7. Write an **original** code segment (function) that represents an order, for each of the following orders.
    See Below.

   Log **N**
   N
   N Log N
   N^{2}
   N^{3}
   2^{N}

Assignment Submission:

Submit a print-out of the program source code and a sample of the output, for each problem. Note you must follow the marking guidelines as identified in the LabMark document.

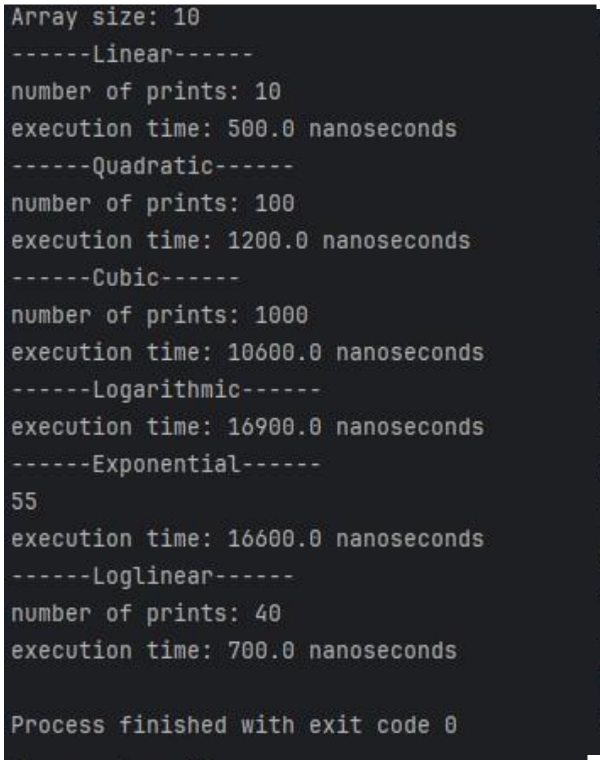
```
import static java.util.Arrays.binarySearch;

public class Ass1_2230 {
   public static void main(String[] args){
      int size = 40;
      int[] num = new int[size];
      for (int i = 0; i < size; i++) {
            num[i] = i + 1;
         }
```

```
    System.out.println("Array size: " + num.length);

    linear(num);
    quadratic(num);
    cubic(num);
    log(num, 15);
    exponential(size);
    logLin(num);
}

/** example of linear order
 * @param n array of n size
 */
public static void linear(int [] n){
    System.out.println("------Linear------");
    double startTime = System.nanoTime();
    int count = 0;

    for(int i = 0; i < n.length; i++) {
        //System.out.println(n[i]);
        count++;
    }
    double endTime = System.nanoTime();
    double duration = endTime - startTime;

    System.out.println("number of prints: " + count);
    System.out.println("execution time: " + duration + " nanoseconds");
}

/** example of quadratic order
 * @param n array of n size
 */
public static void quadratic(int [] n){
    System.out.println("------Quadratic------");
    double startTime = System.nanoTime();
    int count = 0;

    for(int i = 0; i < n.length; i++){
        for(int j = 0; j < n.length; j++){
            //System.out.println(n[j]);
            count++;
        }
    }
    double endTime = System.nanoTime();
    double duration = endTime - startTime;

    System.out.println("number of prints: " + count);
    System.out.println("execution time: " + duration + " nanoseconds");
}
```

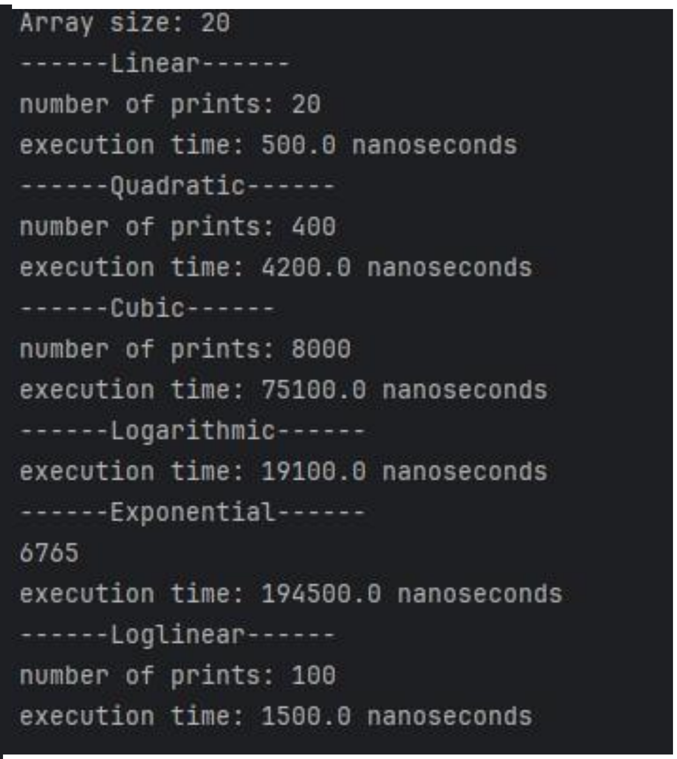
```
/** example of cubic order
 * @param n array of N length
 */
public static void cubic(int[] n){
    System.out.println("------Cubic------");
    double startTime = System.nanoTime();
    int count = 0;
    for(int i = 0; i < n.length; i++){
        for(int j = 0; j < n.length; j++){
            for(int k = 0; k < n.length; k++){
                count++;
            }
        }
    }
    double endTime = System.nanoTime();
    double duration = endTime - startTime;

    System.out.println("number of prints: " + count);
    System.out.println("execution time: " + duration + " nanoseconds");
}

/** example of logarithmic order
 * @param n array of N length
 * @param m key integer to search within the array
 */
public static void log(int[] n, int m){
    System.out.println("------Logarithmic------");
    double startTime = System.nanoTime();

    binarySearch(n, m);

    double endTime = System.nanoTime();
    double duration = endTime - startTime;

    //System.out.println("number of prints: " + count);
    System.out.println("execution time: " + duration + " nanoseconds");
}

/** Handler for the exp() method. prints the execution time
 * @param n integer of n size
 */
public static void exponential(int n){
    System.out.println("------Exponential------");
    double startTime = System.nanoTime();
   // int count = 0;
    System.out.println(exp(n));
    double endTime = System.nanoTime();
    double duration = endTime - startTime;
```

```
        //System.out.println("number of prints: " + count);
        System.out.println("execution time: " + duration + " nanoseconds");
    }

    /** example oh exponential Order
     * @param n integer with size n
     * @return Value after the addition to use in the recursive step
     */
    public static int exp(int n){
      if(n <= 1){
          return n;
      }else{
          return exp(n - 1) + exp(n - 2);
      }

    }

    /** Example of LogLinear Order
     * @param n array of N length
     */
    public static void logLin(int [] n){
        System.out.println("------Loglinear------");
        double startTime = System.nanoTime();
        int count = 0;
        for(int i = 0; i < n.length; i++){
            for(int j = n.length; j > 0; j/=2){
                count++;
            }
        }
        double endTime = System.nanoTime();
        double duration = endTime - startTime;

        System.out.println("number of prints: " + count);
        System.out.println("execution time: " + duration + " nanoseconds");

    }
}
```

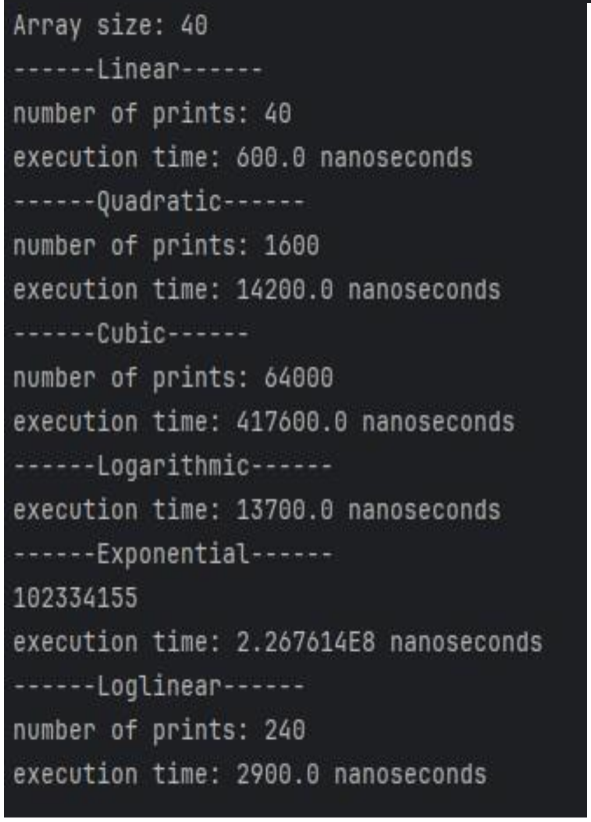
Test cases:



```
Array size: 10
------Linear------
number of prints: 10
execution time: 500.0 nanoseconds
------Quadratic------
number of prints: 100
execution time: 1200.0 nanoseconds
------Cubic------
number of prints: 1000
execution time: 10600.0 nanoseconds
------Logarithmic------
execution time: 16900.0 nanoseconds
------Exponential------
55
execution time: 16600.0 nanoseconds
------Loglinear------
number of prints: 40
execution time: 700.0 nanoseconds

Process finished with exit code 0
```



```
Array size: 20
------Linear------
number of prints: 20
execution time: 500.0 nanoseconds
------Quadratic------
number of prints: 400
execution time: 4200.0 nanoseconds
------Cubic------
number of prints: 8000
execution time: 75100.0 nanoseconds
------Logarithmic------
execution time: 19100.0 nanoseconds
------Exponential------
6765
execution time: 194500.0 nanoseconds
------Loglinear------
number of prints: 100
execution time: 1500.0 nanoseconds
```



```
Array size: 40
------Linear------
number of prints: 40
execution time: 600.0 nanoseconds
------Quadratic------
number of prints: 1600
execution time: 14200.0 nanoseconds
------Cubic------
number of prints: 64000
execution time: 417600.0 nanoseconds
------Logarithmic------
execution time: 13700.0 nanoseconds
------Exponential------
102334155
execution time: 2.267614E8 nanoseconds
------Loglinear------
number of prints: 240
execution time: 2900.0 nanoseconds
```