Department Computer Science and Software Engineering
Concordia University

COMP 352: Data Structures and Algorithms
Fall 2018 - Assignment 1

Name: Tianlin Yang
ID: 40010303

<u>Written Questions (50 marks):</u> Please read carefully: You must submit the answers to all the questions below. However, only one or more questions, possibly chosen at random, will be corrected and will be evaluated to the full 50 marks.

Question 1

a) Given an array *A* of integers of any size, $n \geq 1$, and an integer value *x*, write an algorithm as a pseudo code (not a program!) that would find out the sum of all elements in the array that have values bigger than *x*, as well as the sum of all elements in the array that have values smaller than *x*. For instance, assume that array *A* is as follows:

10	14	3	9	22	35	92	5	9	64

Given *x* = 9, the algorithm would find the two sums as 237 and 8. Your algorithm must handle possible special cases. Finally, the algorithm **must** use the smallest auxiliary/additional storage to perform what is needed.

b) What is the time complexity of your algorithm, in terms of Big-O?

c) What is the space complexity of your algorithm, in terms of Big-O?

Answer:

(a) **Input:** Int x, n, Sumsmall, Sumlarge ,Array //O(1)

Algorithm:

```
for (int i=0, i<n, i++)                              //O(n)

        {if array[i]<x
                Sumsmall = array[i]+Sumsmall         //O(n)
        else
                Sumlarge = array[i]+Sumlarge         //O(n)
        }
```

Output: Sumsmall and Sumlarge

(b) The time complexity should be O(n).

(c) There are x, n, Sumsmall, Sumlarge, array, therefore the space complexity should be O(1).

Question 2

Prove or disprove the following statements, using the relationship among typical growth-rate functions seen in class.

a) $5000000n^5 \log n + n^7$ is $O(n^7 \log n)$
b) $10^{22}n^{21} + 5n^5 + 120n^3$ is $\Theta(n^{29})$
c) n^n is $\Omega(n!)$
d) $0.01n^3 + 0.0000001n^7$ is $\Theta(n^3)$
e) $n^6 + 0.0000001n^5$ is $\Omega(n^5)$
f) $n!$ is $\Theta(2^n)$

Answer:

(a) Correct, but not tight.

$$5000000n^5 \log n + n^7 \le (5000000 + 1)n^7 = cn^7$$
$$for\ c = 5000001\ when\ n \ge n_0 = 1$$

Therefore, is $O(n^7)$ tighter, and $O(n^7 logn) \ge O(n^7)$, $O(n^7)$ also correct.

(b) Correct, but not tight, $\Theta(n^{29})$ too large compare with $\Theta(n^{21})$.

$$10^{22}n^{21} + 5n^5 + 120n^3 \le (10^{22} + 5 + 120)n^{21} \text{ for } n \ge n_0 = 1, \text{ so the } O(n^{21}).$$
$$10^{22}n^{21} + 5n^5 + 120n^3 \ge n^{21} \text{ for } n \ge n_0 = 0, \text{ so the } \Omega(n^{21})$$

Therefore, is $\Theta(n^{21})$, $\Theta(n^{29}) \ge \Theta(n^{21})$, $\Theta(n^{21})$ also correct.

(c) Ture.

$$n^n = n * n * n * \dots\dots\dots\dots * n$$
$$n! = n * (n-1) * (n-2) * \dots\dots\dots\dots * 3 * 2 * 1$$
$$n^n \ge n! \text{ for } n \ge n_0 = 1$$

Therefore, is $\Omega(n!)$.

(d) Ture.

$$0.01n^3 + 0.0000001n^7 \le (0.01 + 0.0000001)n^7 \text{ for } n \ge n_0 = o, \text{ so the } O(n^7).$$
$$0.01n^3 + 0.0000001n^7 \ge 0.0000001n^7 \text{ for } n \ge n_0 = o, \text{ so the } \Omega(n^7).$$

Therefore, is $\Theta(n^7)$.

(e) Correct, but not tight.

$$n^6 + 0.0000001n^5 \ge n^6 \text{ for } n \ge n_0 = o,$$

Therefore, is $\Omega(n^6)$ and $\Omega(n^6) \ge \Omega(n^5)$, $\Omega(n^5)$also correct.

(f) False.

$$n! = n * (n-1) * (n-2) * \dots\dots\dots\dots * 3 * 2 * 1$$

When n=1,2,3,4

$$1! = 1, 2! = 2, 3! = 6, 4! = 24$$
$$2^1 = 1, 2^2 = 4, 2^3 = 8, 2^4 = 16$$
$$n! \ge 2^n \text{ for } n \ge n_0 = 4, \text{ so the } \Omega(2^n).$$
$$n! \le 2^n \text{ for } 0 \le n \le n_0 = 3, \text{ so not the } O(2^n)$$

Question 3

a) Given an ***unsorted*** array A of integers of any size, n ≥ 3, and an integer value *x*, write an algorithm as a pseudo code (not a program!) that would find out if there exist **EXACTLY** 3 occurrences in the array with value *x*.

Answer:

(a) Input: For example Int array[1,1,5,2,1] arraylength(5) x=1,

Algorithm:

```
    Boolean OccurenceCount(int x) {
    for ( i =0; i< arrayLength; i++ )                //O(n) time
            {if( array[ i ] ==x) count++;            //O(n) time
                    if(count==3)
                            return true;             //1 time
            }
    return false;                                    //1 time
}
```

Output: Count = 3 or False

b) What is the time complexity of your algorithm, in terms of Big-O?
O(n)

c) What is the space complexity of your algorithm, in terms of Big-O?
O(1)

d) Will time complexity change if *A* was given as a ***sorted*** array? If yes; give a new algorithm that achieves this better complexity (indicate the time complexity as of that algorithm). If no, explain why such new constraints/conditions cannot lead to a better time complexity.

YES, The searching in a sorted array is changed to O(log n).
Input an example sorted array from low to high value [1,1,1,2,5] Low = 1, High = 5.
Input: Array, int x, int low, int high
Algorithm:

```
If (low > high)
    Return false;
else {int mid = (low+high)/2;
    if (x == array[mid])
            if (array[mid-1] == array[mid+1] || array[mid-1] == array[mid-2]
              || array[mid+1] == array[mid+2])
                    count = 3
                    return ture;
            else
                    printf ('can't find 3 occurrences');
                    System.exit(-1);
    else if (x < array[mid])
            return OccurenceCount(data, target, low, mid -1);
    else
            return OccurenceCount(data, target, mid+1, high);
```

Output: Count = 3 or False

<u>Programming Questions (50 marks):</u>

In class, we discussed about the two versions of Fibonacci number calculations: BinaryFib(n) and LinearFibonacci(n) (refer to your slides and the text book). The first algorithm has exponential time complexity, while the second one is linear.

a) In this programming assignment, you will design an algorithm (in pseudo code), and implement (in Java), two recursive versions of an *Oddonacci* calculator (similar to the ones of Fibonacci) and experimentally compare their runtime performances. Oddonacci numbers are inspired by Fibonacci numbers but start with three predetermined values, each value afterwards being the sum of the preceding three values. The first few Oddonacci numbers are:

   1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355, 653, 1201, 2209, 4063, 7473, 13745, 25281, 46499, ...

For that, with each implemented version you will calculate Oddonacci(5), Oddonacci (10), etc. in increments of 5 up to Oddonacci(100) (or higher value if required for your timing measurement) and measure the corresponding run times. For instance, Oddonacci(10) returns 105 as value. You can use Java's built-in time function for this purpose. You should redirect the output of each program to an *out.txt* file. You should write about your observations on the timing measurements in a separate text file. You are required to submit the two fully commented Java source files, the compiled files, and the text files.

Pseudo code for Binary Oddonacci Calculator:

```
    if (input < 1)                       //Output Warning for input less than 1
    {OddonacciNumber = 0;
    return OddonacciNumber; }            //return a empty value
else
    if(input == 1 || input == 2 ||input == 3)    //Generate initial value {1,1,1}
    {OddonacciNumber = 1;}
    else                                 //Generate number for n>3
    {OddonacciNumber = BinaryOddonacci(input-1) +
                       BinaryOddonacci(input-2) +
                       BinaryOddonacci(input-3);}
    return OddonacciNumber;              //Return current Onumber
```

Pseudo code for Linear Oddonacci Calculator:

```
    int[] ResultArray;
    int first;
    int second;
    int third;
    if (input==1,==2,==3)
    return ResultArray = {1} or {1,1} or {1,1,1}    //For first three values are special
      ResultArray = linearCalculator(input - 1);    //Linear for n,n-1,n-2
      first = ResultArray[0];
      second = ResultArray[1];
      third = ResultArray[2];                       //Extract {n-1,n-2,n-3}
      System.out.print((first+second+third) + " ");     //Output Oddonacci value for 1 time run
```

```
ResultArray[0] = first + second + third;
ResultArray[1] = first;
ResultArray[2] = second;                    //Save the current time Result as{n,n-1,n-2}
return (ResultArray);               //Tail of Oddonacci, which is Linear
 // this will return (Oddaonacci(n),Oddaonacci(n-1), Oddaonacci(n-2))
```

b) Briefly explain why the first algorithm is of exponential complexity and the second one is linear (more specifically, how the second algorithm resolves some specific bottleneck(s) of the first algorithm). You can write your answer in a separate file and submit it together with the other submissions.

The binary version calculator is exponential complexity due to the multi-recursion with run time shown below, t1 is when n=1 call method 1 time, t2 to t5 similar.

t1=1
t2=1
t3=1
t4=1+t1 +t2+t3 =1+1+1+1 =4
t5=1+t2+t3+t4 =1+1+1+4 =7
……

For each recursion it calls itself 3 times, therefore it's big $O(3^n)$. And its not be the tail recursive.

The linear version calculator is using tail recursive algorithm, with some additional parameters to save temp values of f(n-3), f(n-2) and f(n-1). Which could be used for the next call. It will reduce the time complexity to big $O(n)$.

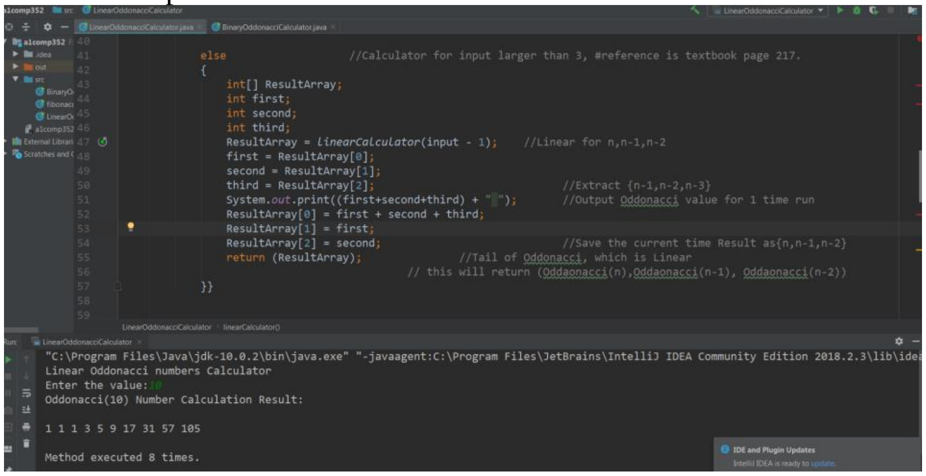
c) Do any of the previous two algorithms use tail recursion? Why or why not? Explain your answer. If your answer is "No" then:

Can a tail-recursive version of Oddonacci calculator be designed?

i. If yes; write the corresponding pseudo code for that tail recursion algorithm and implement it in Java and repeat the same experiments as in part (a) above.
ii. If no, explain clearly why such tail-recursive algorithm is infeasible.

No for the first algorithm, is not tail recursive. Because it has exponential run time.

Yes for the second algorithm, is tail recursion. It's the method calls itself at the end ("tail") of the method in which no computation is done after the return of recursive call.



a1comp352
src
LinearOddonacciCalculator
LinearOddonacciCalculator.java
BinaryOddonacciCalculator.java
.idea
out
LinearOddonacciCalculator
linearCalculator()
Run:
IDE and Plugin Updates
IntelliJ IDEA is ready to update.

```
        else                    //Calculator for input larger than 3, #reference is textbook page 217.
        {
            int[] ResultArray;
            int first;
            int second;
            int third;
            ResultArray = linearCalculator(input - 1);    //Linear for n,n-1,n-2
            first = ResultArray[0];
            second = ResultArray[1];
            third = ResultArray[2];                        //Extract {n-1,n-2,n-3}
            System.out.print((first+second+third) + " ");      //Output Oddonacci value for 1 time run
            ResultArray[0] = first + second + third;
            ResultArray[1] = first;
            ResultArray[2] = second;                       //Save the current time Result as{n,n-1,n-2}
            return (ResultArray);               //Tail of Oddonacci, which is Linear
                                        // this will return (Oddaonacci(n),Oddaonacci(n-1), Oddaonacci(n-2))

        }}
```

```
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2018.2.3\lib\idea
Linear Oddonacci numbers Calculator
Enter the value:10
Oddonacci(10) Number Calculation Result:

1 1 1 3 5 9 17 31 57 105

Method executed 8 times.
```