University of Manitoba  
COMP 2140, Winter 2019  
Assignment 2

Due Date: February 11, at 8:00pm

There is no hurry. We shall get there some day. Rivers know this . . .  
A.A. Milne (Winnie-the-Pooh)

Please pay attentions to the followings when preparing your assignment:

• All problems are written problems. There is no programming component in this assignment.

• You need to submit your solutions electronically via Crowdmark. Let me know if you have any problem accessing questions on Piazza.

• Think of this assignment as an opportunity to learn. The assignment includes long remarks and reviews from the course material. Do not be intimidated by its long length.

• There are 6 problems with a total mark of 85. The last question is a bonus question and is worth 15 marks. Despite being a bonus question, you might find the last question not much harder than a similar question you faced in the quiz. So, think of it as an opportunity to improve your skills and your mark. Your assignments will be marked out of 70. If you get a mark higher than 70, the extra marks will be transferred to other components of the course.

• As always, I encourage you to ask your questions on Piazza. Those students who help others on Piazza will receive bonus marks (by ‘helping’, I mean removing confusions). It is likely that I drop hints on Piazza in response to the questions that are asked publicly. In the pursuit of fairness, I hesitate to drop a hint in response to private questions or emails. So, be active on Piazza!

• This is an individual assignment. You are welcome to discuss questions with your friends (or enemies). But you should write the answers individually, and you should fully understand what you are writing. Please read http://www.cs.umanitoba.ca/~kamalis/winter19/infoCOMP2140.pdf for guidelines on academic integrity.

• If you found the assignment too long or hard, do not panic. We are going to walk together in this course to improve your algorithmic skills. You just need to be committed to the course and follow the right steps. You are always welcome to send share your comments and concerns on Piazza. It can be anonymous.

• Good luck!
Problem 1  Java Review [10 × 2 = 20 marks]

This problem contains short-answer questions that review object oriented concepts of Java. You do not need to justify your choices. But make sure to understand these concepts for your midterm. For True/False questions, there is no penalty for a wrong choice.

(a) True or False: An abstract class can extend a non-abstract (concrete) class.

(b) True or False: An interface can extend a class.

(c) True or False: An abstract class can implement an interface.

(d) True or False: A class can implement two different interfaces.

(e) True or False: An abstract data type can be implemented with different data structures.

(f) Sketch the tree structure of the Animal class hierarchy in the code at the end of this question.

(g) True or False: It is possible to create an instance of Animal using Animal a = new Animal(4).

(h) True or False: An instance of class Mammal can access the value of its heart using super.heart.

(i) True or False: An instance of class Bat can access the value of its eye using super.super.eye.

(j) True or False: In order for the code to compile without error, variable wing in classes Bat and Bird should have the same type.

abstract class Animal
{
    public int brain;
    protected int eye;
    private int heart;
    Animal (int x)
    {
        eye = x;
        brain = heart = 0;
    }
}

class Mammal extends Animal
{
    public int fur;
    private int tail;
}

class Bat extends Mammal
{
    public int wing;
Problem 2  Understanding Big Oh [4 × 4 = 16 marks]

The aim of this problem is to give you an intuition about the meaning of the big-Oh notation. Consider three functions \( f(n) = \log n, \ g(n) = n \log n, \ h(n) = n + \log n, \) and \( z(n) = n^3. \)

(a) Use big-Oh notation to indicate the relationship between these four functions (no need to justify).

(b) Indicate how the values of these four functions grow when we increase \( n \) by a factor of 16. For example, for \( f(n) \), we have \( f(16n) = \log(16n) = \log 16 + \log n = 4 + f(n). \) Repeat this for the other three functions.

(c) Write down the value of \( h(n) \) for \( n_1 = 2^{16} \) and \( n_2 = 2^{32}. \) Note that \( n_2 \) is much, much larger than \( n_1. \) Indicate how the two terms \( n \) and \( \log n \) in \( h \) grow when we increase \( n \) from \( n_1 \) to \( n_2. \) What can we conclude?

(c) We know that \( 2n + \log n \in O(n). \) To formally prove this, we need to find an \( n_0 \) such that for large values of \( n \) (those larger than \( n_0 \)) we have \( 2n + \log n \leq M \cdot n \) for some value of \( M. \) We know for \( n \geq 2 \) we have \( \log n < n. \) Use this inequality to indicate some values of \( n_0 \) and \( M \) that show \( 2n + \log n \in O(n). \)

Problem 3  Algorithm Analysis [3 × 4 = 12 marks]

In this question, we review the basics for analyzing a piece of code. We would like to analyze the following piece of Java code and give a bound on its running time as a function of \( n. \) The algorithm receives two matrices \( A \) and \( B, \) each of size \( n \times n, \) and stores their product in a new array \( C \) which is later returned.

Remark: When writing the time complexity, we do not care about the actual value of the involved constants. For example the time complexity of algorithm remains the same if the statement \( C[i][k] += A[i][j]*B[j][k] \) is counted as “1 statement” or “2 operations” (a multiplication and an addition) or “9 memory access” (to indices and pointer destination). Regardless of what we really count, the statement takes a constant time and the complexity of the function remains the same when we use Big-Oh notation.

```java
public int[][] multiply(int[][] A, int[][] B, int n) {
    int[][] C = new int[n][n];
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            if(A[i][j] != 0){
                for(int k=0; k<n; k++){
                    C[i][k] += A[i][j]*B[j][k];
                }
            }
        }
    }
    return C;
}
```
return C;
}

(a) What is the best-case time complexity of multiply as a function of n? You should consider a scenario in which the time complexity of the algorithm is minimized. Use big-Oh notation to summarize the time complexity.

(b) What is the worst-case time complexity of multiply as a function of n? You should consider a scenario in which the time complexity of the algorithm is maximized. Use big-Oh notation to summarize the time complexity.

(c) Consider the following power functions which takes a matrix A of size $n \times n$ and repeatedly calls the function multiply to find the $A^{2019}$, that is, the 2019th product of A. What is the worst-case time complexity of power as a function of n? Use big-Oh notation to summarize the time complexity.

Remark: When a function $f$ calls another function $g$ for $k$ times, the time complexity of $f$ will be added by $k$ times the complexity of $g$. For example, if a function $f$ calls binary-search function on $n$ numbers for $n^2$ times, the time complexity of $f$ will be added by $n^2 \times O(\log n) = O(n^2 \log n)$.

public int[][] power(int[][] A, int n) {
    // initiate res as the second power of A
    int[][] res = multiply(A, A, n);
    for (int i=0; i<2018; i++)
        res = multiply(res, A, n);
    return res;
}

Problem 4 Recursive Insertion Sort [8+4 = 12 marks]

In the class, we saw an iterative implementation of the insertion sort. There is also an easy recursive interoperation of the algorithm. Assume we want to sort an array $A$ of size $n$; we can recursively sort the array up-to index $n-1$ and then ‘insert’ the last element into its right position by checking and shifting elements before it (the last step is similar to the inner loop in the iterative interoperation).

(a) write a recursive function void recInsertionSort(int[] A, int n) that sorts the first $n$ elements of an input array A of integers. Do not forget to cover the base of recursion.

public static void recInsertionSort(int[] A, int n) {
    if (n < 2)
        return;
    recInsertionSort(A, n-1);
    int key = A[n-1];
    int j = n-2;
    while (j >= 0 && key < A[j])
    {
        j --;
    }
    A[j+1] = key;
}
(b) sketch the recursion tree for the above recursive method. Do you think the time complexity of the algorithm is better than the iterative version (a short explanation is sufficient; no formal justification is required).

Problem 5 Palindrome Numbers [2 + 8 = 10 marks]

A string is said to be palindrome if it reads identically left-to-right and right-to-left. For example, “abba”, “abcba” and “amanaplanacanalpanama” are palindromes.

(a) The following algorithm is aimed to determine whether an input is palindrome. Unfortunately the algorithm has a logic error. Find the error and explain what problem it will cause. Then describe a short fix to the error (write the correct statements).

```java
boolean isPalindrome(String s) {
    int i = 0, j = s.length() - 1;
    while (i != j && s.charAt(i) == s.charAt(j)) {
        i ++;
        j --;
    }
    return (i == j);
}
```

(b) We are going to develop another function isPalindrome2(String s) which indicates whether an input string is palindrome. As shown below, isPalindrome2 calls a recursive algorithm isPalindromeRec(String s, int i, int j) that indicates whether the substring starting at index i and ending at index j of input string s is palindrome (e.g., isPalindromeRec("canada",1,3) returns true). Develop isPalindromeRec using tail recursion.

```java
boolean isPalindrome2(String s) {
    return isPalindromeRec(s, 0, s.length() - 1);
}

boolean isPalindromeRec(String s, int i, int j) {
    if (i > j)
        return true; // i and j have passed each other.
    if (s[i] != s[j])
        return false;
    return isPalindromeRec(s, i+1, j-1);
}
```

Problem 6 [Bonus] Recursion & Memorization [15 marks]

Assume you are given a positive integer n. Given this value, you can perform any one of the following 3 operations:
• Decrement \( n \) (that is \( n = n - 1 \)).

• If \( n \) is an even number, divide it by 2 (that is, \( \text{if } (n \% 2 == 0) \{ n = n / 2 \} \)).

• If \( n \) is divisible by 3, divide it by 3 (that is, \( \text{if } (n \% 3 == 0) \{ n = n / 3 \} \)).

We are interested in designing an algorithm that, given a positive integer \( n \), outputs the minimum number of steps that take \( n \) to 1. Let \( F(n) \) denote that number.

For example, for \( n = 1 \) the algorithm returns 0, for \( n = 4 \) the algorithm returns 2 (the operations are \( 4/2 = 2 \) followed by \( 2/2 = 1 \)), and for \( n = 7 \) the algorithm returns 3 (the operations are \( 7 - 1 = 6 \) followed by \( 6/3 = 2 \) followed by \( 2/2 = 1 \)). Note that the algorithm that always chooses the operation that makes \( n \) as small as possible does not always return the correct answer. For example, for \( n = 10 \), if we divide it by 2 (the operation that makes it as small as possible), then we will need 4 steps to get to 1 (\( 10/2 = 5 \) followed by \( 5 - 1 = 4 \), followed by \( 4/2 = 2 \) followed by \( 2/2 = 1 \)). However, the correct solution requires 3 steps (\( 10 - 1 = 9 \) followed by \( 9/3 = 3 \) followed by \( 3/3 = 1 \)). So, we will need to develop a solution that considers all possibilities.

(a) Derive a recursion definition for \( F(n) \). Your recursive function should call itself at most 3 times.

(b) In order to compute \( F(n) \) efficiently, you will need memorization. Write a recursive algorithm, named \texttt{recPathToOne} that receives an input integer \( n \) and uses \texttt{memorization} to compute \( F(n) \). Note that the algorithm needs to receive another parameter (in addition to \( n \)) for memorization. As always, do not forget to cover the base of recursion. Similar to other memorization examples, you need to also have a parent function \texttt{pathToOne(int n)} that initiates the memory and calls \texttt{recPathToOne} for the first one. This parent function is public and is called from outside.

```java
public static int pathToOne(int n)
{
    int[] memo = new int[n+1];
    for (int i = 0; i < n+1; i++)
        memo[i] = -1;
    return getMinSteps(n, memo);
}

private static int getMinSteps(int n, int[] memo)
{
    if (n == 1)
        return 0; // base case
    if (memo[n] != -1)
        return memo[n]; // we have solved it already :)

    int a, b, c; // three possible paths to 1
    a = b = c = -1;
    a = getMinSteps(n-1, memo); // '-1' path.
    if(n%2 == 0)
        b = getMinSteps(n/2, memo); // '/2' path
    if(n%3 == 0)
        c = getMinSteps(n/3, memo); // '/3' path

    int res = a;
    if (b != -1 && b < res)
        res = b;
```

if (c != -1 && c < res)
    res = c;

res ++; // adding the last operation
    memo[n] = res; // save the result. If you forget this step, then its
    same as plain recursion.

    return res;
}