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](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.
   Answer: False and True, it is possible but not commonly used.
   Update: It turns out that there is no syntax issue with an abstract class extending a concrete one. That is, you do not see an error. However, in practice, it is very uncommon for an abstract class to extend a concrete one. Having that said, you should receive the mark for this question if you answered it “True” or “False”.

(b) True or False: An interface can extend a class.
   Answer: False, a class implements an interface.

(c) True or False: An abstract class can implement an interface.
   Answer: True, it is possible for an interface to be implemented by a class (regardless of the class being abstract or concrete).

(d) True or False: A class can implement two different interfaces.
   Answer: True; it is possible.

(e) True or False: An abstract data type can be implemented with different data structures.
   Answer: True; think of a Stack being implemented with an array or a linked list.

(f) Sketch the tree structure of the Animal class hierarchy in the code at the end of this question.
   Answer: A tree with Animal in the root and Mammal and Bird as its two children. Mammal indeed has Bat as its child.

(g) True or False: It is possible to create an instance of Animal using Animal a = new Animal(4).
   Answer: False. It is not possible because Animal is an abstract class.

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

(i) True or False: An instance of class Bat can access the value of its eye using super.super.eye.
   Answer: True. Because eye is protected and hence accessible to subclasses.

(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.
   Answer: False. The two classes are unrelated and there is no restriction with respect to their variables as they are hidden from each other.

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;
}

class Bird extends Animal
{
  public double wing;
}

Answer:

**Marking Scheme:** 2 marks for each correct answer and 0 otherwise.

### 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).

**Answer:** \( f(n) \in O(h(n)) \in O(g(n)) \in O(z(n)) \).

**Marking Scheme:** 4 marks for correct ordering; -2 mark for each mistake; 0 for two mistakes or more. No justification is needed.

(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.

**Answer:** We have \( g(16n) = 16n \log 16n = 64n + 16n \log n = 16g(n) + 64n \). Similarly, \( h(16n) = 16n + \log 16n = 16n + \log n + 4 = 16h(n) - 15 \log n + 4 \). Finally, \( z(16n) = (16n)^3 = 4096z(n) \). Note that the functions with larger asymptotic value grow faster when the input size grows.

**Marking Scheme:** 4 marks for following the correct algebra in all cases. -1 mark for each mistake on algebra. It is not necessary to achieve the most simplified form (the first step in above calculations is enough).

(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?

**Answer:** \( h(n_1) = 2^{16} + 16 \) and \( h(n_2) = 2^{32} + 32 \). The value of \( n \) grows by a huge multiplicative factor of \( 2^{16} \) while the the value of \( \log n \) is increased by a constant 16. We can conclude the growth of \( n \) is much faster than \( \log n \), that is, for large values of \( n \), the term \( \log n \) can be ignored in the asymptotic sense as it contributes little to the actual value.

**Marking Scheme:** 2 marks for correct mentioning of the growth rates. 2 marks for correct conclusion from that observation.
(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 $2n + \log n \in O(n)$. Answer: Assume $n > 2$ we have $\log n < n$, hence, $2n + \log n < 3n$. So, any value of $n_0 \geq 2$ and $M \geq 3$ show that $2n + \log n \in O(n)$ (e.g., $n_0 = 2$ and $M = 3$).

**Marking Scheme:** 2 marks for following the correct definition, and 2 marks for correct values of $n_0$ and $M$ (note that any $n_0 \geq 2$ and any $M \geq 3$ work. There are many correct solutions!

**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;
}
```

(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. Answer: In the best case, all values of $A$ are 0. Hence, the inner-most loop is never executed. The running time will be $\sum_{i=0}^{n} \sum_{j=0}^{n} c$ for some constant $c$. Basically $c$ is added up $n \times n$ times which gives a total time complexity of $cn^2 \in O(n^2)$.

**Marking Scheme:** 2 marks for mentioning what the best case is and 2 marks for correct running time for this case.

(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. Answer: In the worst case, the value of all elements of $A$ are non-zero. The inner loop always executes. The
running time will be \( \sum_{i=0}^{n} \sum_{j=0}^{n} (c_1 + \sum_{k=0}^{n} c_2) \) for some constants \( c_1 \) and \( c_2 \). Basically \( c_1 \) is added up \( n^2 \) times while \( c_2 \) is added \( n^3 \) times. This gives a total time complexity of \( c_1 n^2 + c_2 n^3 \in O(n^3) \).

**Marking Scheme:** 2 marks for mentioning what the worst case is and 2 marks for correct running time for this case.

(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 \( n^2 \times O(\log n) = O(n^2 \log n) \).

**Answer:**
`power` calls `multiply` 2019 times. Other components of `power` take a constant time \( c_3 \). From the previous part, we know `multiply` takes \( O(n^3) \) in the worst-case. So, the time complexity of `power` is \( c_3 + 2019 O(n^3) = O(n^3) \). In a sense, calling a function one time or a constant number of times does not make a difference in time complexity.

**Marking Scheme:** 2 marks for mentioning \( 2019 n^3 \) or \( 2019 \cdot O(n^3) \) and two marks for mentioning \( O(n^3) \).

```java
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.

```java
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;
}
```
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);
}

Answer: If the length of a palindrome input is an even integer, i and j ‘pass’ each other without causing a break in the loop (they never become equal). So, the values of i and j do not become equal and the while loop does not end. To fix this issue, it suffices to replace i != j with i<j.

Marking Scheme: 1 mark for correct report of the issue and 1 mark for correct fixing.

(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 isPlaindromeRec(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., isPlaindromeRec("canada",1,3) returns true). Develop isPlaindromeRec 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);
}
```

**Answer:**

**Marking Scheme:** -3 marks if the parent (public function is missing; deduce marks if it is calling the private function with incorrect parameters). -2 marks if the base of recursion (checking \( i > j \)) is missing (it can be \( i \geq j \) or something similar depending on the algorithm). -2 marks for each mistake in the recursive call. -2 marks if the recursion is not a tail recursion (a computation is made after the recursive call). -2 marks if the check \( s[i] \) is not checked with \( s[j] \). The recursive function should receive two indices as input (your answer cannot be correct if they are missing, and you will get more than half of the max-mark in that case). You get partial marks for partially correct answers.

**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, if \( (n \mod 2 == 0) \) \{ \( n = n / 2 \} \).
- If \( n \) is divisible by 3, divide it by 3 (that is, if \( (n \mod 3 == 0) \) \{ \( n = n / 3 \} \).

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.

**Answer:** We have \( F(n) = 1 + \min\{F(n-1), F(n/2), F(n/3)\} \) for \( (n > 1) \) and \( F(1) = 0 \).

\[
F(n) = \begin{cases} 
1 + \min\{F(n-1), F(n/2), F(n/3)\} & \text{if } n\%6 = 0 \\
1 + \min\{F(n-1), F(n/2)\} & \text{if } n\%6 \neq 0 \text{ and } n\%2 = 0 \\
1 + \min\{F(n-1), F(n/3)\} & \text{if } n\%6 \neq 0 \text{ and } n\%3 = 0 \\
1 + F(n-1) & \text{otherwise}
\end{cases}
\]
Marking Scheme: This part has 5 marks. You get 3 out of 5 if only include the first line of the above recursion. We tend to be a bit harsher when marking bonus questions.

(b) In order to compute $F(n)$ efficiently, you will need memorization. Write a recursive algorithm, named `recPathToOne` that receives an input integer $n$ and uses 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 `pathToOne(int n)` that initiates the memory and calls `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;
}
```

Answer: 0 marks if the solutions has fundamental issues (we are being generous for bonus questions). -2 marks if the base of reduction is not covered. -4 marks if there is an issue with the memorization (if an array like memo is not passed or if it is not checked for negative values. -3 marks for each missing recursive call (there should be 3). -3 marks if the result is not incremented or memo is not updated.