CPSC 331 — Assignment #1

Proving the Correctness of Simple Algorithms — and Implementing Them as Java Programs

About This Assignment

This assignment can be completed by groups of up to three students. It is due by 11:59 pm on Wednesday, October 2.

Please read the following ***before*** you begin work on this:

- Assignment Do's and Don't's: Information about what is allowed — and what is ***not*** allowed — when working on assignments in this course
- Expectations about Quality of Submissions for Assignments
- How to Submit an Assignment

Marks will be deducted if submission instructions are not followed — especially if this increases the time spent by a teaching assistant or instructor to mark the submitted work.

The Problem To Be Solved: The MacGonagall Mystery

At the *Hogwarts School of Witchcraft and Wizardry*, significant time is spent studying the mystical properties of various sequence of integers. These include the *MacGonagall Mystery* — a sequence of numbers $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \ldots$ defined as follows: For every integer $i \geq 0$,

$$\mathcal{M}_i = \begin{cases} 1 & \text{if } i = 0, \\ 0 & \text{if } i = 1, \\ 5 & \text{if } i = 2, \\ 8 & \text{if } i = 3, \\ 2\mathcal{M}_{i-1} - 2\mathcal{M}_{i-3} + \mathcal{M}_{i-4} & \text{if } i \geq 4. \end{cases}$$

```
integer smacG (integer n) {
// Assertion:  A non-negative integer n has been given as input.
 1. if (n == 0) {
 2.   return 1
 3. } else if (n == 1) {
 4.   return 0
 5. } else if (n == 2) {
 6.   return 5
 7. } else if (n == 3) {
 8.   return 8
    } else {
 9.   return 2 × smacG(n − 1) − 2 × smacG(n − 3) + smacG(n − 4)
    }
}
```

Figure 1: A Slow Algorithm for the "MacGonagall Mystery Computation" Problem

Consequently, Hermione Granger is interested in the following computational problem:

MacGonagall Mystery Computation

Precondition: A non-negative integer $\mathtt{n}$ is given as input.
Postcondition: The $\mathtt{n}^{\text{th}}$ MacGonagall number, $\mathcal{M}_{\mathtt{n}}$, is returned as output.

A Simple — But Slow — Algorithm for This Computation

Hermione has written a Java program that implements the algorithm shown in Figure 1.

1. Give a reasonably short proof that the function $f(\mathtt{n}) = \mathtt{n}$ is a ***bound function*** for this recursive algorithm.
2. Prove that the `smacG` algorithm correctly solves the "MacGonagall Mystery Computation" problem.

3. Write a Java program, `SMacGonagall.java`, that satisfies the following properties:

   - It is part of the `cpsc331.assignment1` package.
   - Integer inputs and outputs are represented using Java's `int` data type.
   - It includes a method, `smacG`, which receives an integer `n` as input and which computes the $\mathtt{n}^{\text{th}}$ MacGonagall number if $\mathtt{n} \geq 0$, using the algorithm that has now been analyzed, but which actually solves a more general computational problem than the one given above:

     Extended MacGonagall Mystery Computation

     Precondition: An integer `n` is given as input.

     Postcondition: If $\mathtt{n} \geq 0$ then the $\mathtt{n}^{\text{th}}$ MacGonagall number, $\mathcal{M}_{\mathtt{n}}$, is returned as output. An `IllegalArgumentException` is thrown otherwise.

     The method should not be public but should be accessible by other classes in the `cpsc331.assignment1` package.

   - The main method should read its input from the command line. If either the number of inputs is incorrect, or the input is not an integer, then it should return the message

     ```
     Fiddlesticks!  One integer input is required.
     ```

     If there is a single integer input, but it is negative, then the main method should return the message

     ```
     Fiddlesticks!  The integer input cannot be negative.
     ```

     Otherwise —- when given a non-negative integer input `n` – it should report the corresponding MacGonagall number $\mathcal{M}_{\mathtt{n}}$.

     A few sample runs of the program should therefore look like the following.

     ```
     > java cpsc331.assignment1.SMacGonagall 0
     > 1

     > java cpsc331.assignment1.SMacGonagall 1
     > 0

     > java cpsc331.assignment1.SMacGonagall 2
     > 5

     > java cpsc331.assignment1.SMacGonagall 3
     > 8

     > java cpsc331.assignment1.SMacGonagall 4
     > 17
     ```

```
> java cpsc331.assignment1.SMacGonagall -1
> Fiddlesticks!  The input integer cannot be negative.

> java cpsc331.assignment1.SMacGonagall
> Fiddlesticks!  One integer input is required.

> java cpsc331.assignment1.SMacGonagall xyz
> Fiddlesticks!  One integer input is required.
```

The following files are available and can be used before you submit your program for assessment:

- `test_smacG.java`: A program that can be executed using `JUnit` to perform unit tests of the method `smacG`. See Java Development Exercise #5 for information about how to use this.
- `test_SMacGonagall.sh`: A Unix shell script that can be used to perform testing of the main method. See Java Development Exercise #6 for information about how to use this.

4. Let $T_{\text{smacG}}(n)$ be the number of steps included in the execution of the algorithm, `smacG`, shown in Figure 1 on input n, for a non-negative integer n — assuming that the ***uniform cost criterion*** is used to define this and the only steps counted are the numbered steps shown in Figure 1.

   Give a ***recurrence*** for $T_{\text{smacG}}(n)$.

   Note: If your answer is correct then you should be able to use your recurrence to show that $T_{\text{smacG}}(0) = 2$, $T_{\text{smacG}}(1) = 3$, $T_{\text{smacG}}(2) = 4$, $T_{\text{smacG}}(3) = 5$, $T_{\text{smacG}}(4) = 15$, and $T_{\text{smacG}}(5) = 27$.

5. Use your recurrence to prove that $T_{\text{smacG}}(n) \geq \left(\frac{3}{2}\right)^n$ for every non-negative integer n. Thus the number of steps executed by the `smacG` algorithm is at least ***exponential*** in its input.

An Asymptotically Faster Algorithm for This Computation

After a discovery of what is considered when solving the above problem, Ms. Granger realized that this algorithm is far too slow to be used on Muggles computers to compute the n^{th} MacGonagall number, $\mathcal{M}_n$, when n is very large. Indeed, one should not even try to use it to compute $\mathcal{M}_{1000}$.

```
integer fmacG (integer n) {
// Assertion:  A non-negative integer n has been given as input.
 1. if (n == 0) {
 2.   return 1
 3. } else if (n == 1) {
 4.   return 0
 5. } else if (n == 2) {
 6.   return 5
 7. } else if (n == 3) {
 8.   return 8
    } else {
 9.   int[] M := new int[n + 1]
10.   M[0] := 1
11.   M[1] := 0
12.   M[2] := 5
13.   M[3] := 8
14.   int i := 4
15.   while (i ≤ n) {
16.     M[i] := 2 × M[i − 1] − 2 × M[i − 3] + M[i − 4]
17.     i := i + 1
      }
18.   return M[n]
    }
}
```

Figure 2: Another Algorithm for the "MacGonagall Mystery Computation" Problem

Happily, Hermione was familiar with one of the Muggles arts that you will learn about in CPSC 413 — ***Dynamic Programming***.[1] Another algorithm for the "MacGonagall Mystery Computation" problem, making use of this technique, is shown in Figure 2. This algorithm is ***not*** recursive; instead, it creates and accesses an array.

[1] Ms. Granger wonders why Muggles insist on giving such confusing names to simple ideas — like storing a solution for an instance of a problem and reusing this solution, instead of solving the same instance of the problem over and over again, as if you hadn't already solved it before!

6. State a ***loop invariant*** for the `while` loop at lines 15–17 of this algorithm.

   For full marks (or even very many part marks) your answer should have the following properties:

   - It is *correct*. That is it really *is* a loop invariant for this `while` loop.
   - It is *complete* enough to be used to establish the partial correctness of this algorithm.
   - It is *not* necessary to make any guesses about the value of $\mathcal{M}_n$, for $n \geq 4$, in order to see that this is the case.

7. Prove that your answer for the previous question really *is* a loop invariant for the `while` loop in this algorithm.

8. Use this to prove that this algorithm is ***partially correct***.

9. State a ***bound function*** for the `while` loop in this algorithm and prove that your answer is correct.

10. Prove that if this algorithm is executed when the precondition for the "MacGonagall Mystery Computation" problem is satisfied, and the `while` loop is reached and executed, then the execution of the loop eventually ends.

11. Using what has been proved so far, complete a proof that the `fmacG` algorithm correctly solves the "MacGonagall Mystery Computation" problem.

12. Let $T_{\text{fmacG}}(n)$ be the number of steps executed by the algorithm `fmacG` when executed on input n, for a natural number n — as defined using the Uniform Cost Criterion (and counting only the numbered steps shown in Figure 2) — so that, for example, $T_{\text{fmacG}}(0) = 2$, because the steps at lines 1 and 2 are carried out when this algorithm is executed on input 0.

    Using techniques introduce in this course, give an ***upper bound*** for $T_{\text{fmacG}}(n)$, as a function of n, that is as precise as you can. Your final answer should be "in closed form", so that it does not include any summations or recurrences.

13. Write a Java program, `FMacGonagall.java`, that satisfies the following properties:

    - It is part of the `cpsc331.assignment1` package.
    - Integer inputs and outputs are represented using Java's `int` data type.
    - It includes a method, `fmacG`, which receives an integer `n` as input and which computes the $\texttt{n}^{\text{th}}$ MacGonagall number if $\texttt{n} \geq 0$, using the algorithm which has now been analyzed, but which solves the more general "Extended MacGonagall Mystery Computation" problem that has also been defined. The method should not be

public but should be accessible by other classes in the `cpsc331.assignment1` package.

- The main method should read its input from the command line. If either the number of inputs is incorrect, or the input is not an integer, then it should return the message

  ```
  Fiddlesticks!  One integer input is required.
  ```

  If there is a single integer input, but it is negative, then the main method should return the message

  ```
  Fiddlesticks!  The integer input cannot be negative.
  ```

  Otherwise —- when given a non-negative integer input `n` – it should report the corresponding MacGonagall number $\mathcal{M}_{\mathtt{n}}$.

  A few sample runs of the program should therefore look like the following.

  ```
  > java cpsc331.assignment1.FMacGonagall 0
  > 1

  > java cpsc331.assignment1.FMacGonagall 1
  > 0

  > java cpsc331.assignment1.FMacGonagall 2
  > 5

  > java cpsc331.assignment1.FMacGonagall 3
  > 8

  > java cpsc331.assignment1.FMacGonagall 4
  > 17

  > java cpsc331.assignment1.FMacGonagall -1
  > Fiddlesticks!  The input integer cannot be negative.

  > java cpsc331.assignment1.FMacGonagall
  > Fiddlesticks!  One integer input is required.

  > java cpsc331.assignment1.FMacGonagall xyz
  > Fiddlesticks!  One integer input is required.
  ```

The following files are available and can be used before you submit your program for assessment:

- `test_fmacG.java`: A program that can be executed using `JUnit` to perform unit tests of the method `smacG`.

- `test_FMacGonagall.sh`: A Unix shell script that can be used to perform testing of the main method.

Finding a Closed Form

14. Find an expression for the $\texttt{n}^{\text{th}}$ MacGonagall number, as a function of `n`, in ***closed form*** — so that it does not include a summation or recurrence — and prove that your answer is correct.

    Hint: Compare $\mathcal{M}_n$ to a few simple functions, including n, n^2, and 2^n. Look for a pattern.