CPSC 319 Assignment 1: Comparing Algorithms

Due Date: Friday February 7th, 2020 11:59pm
Weight: (30 * 0.25)% [Assignment 1 of 4]

Collaboration
IMPORTANT

Discussing the assignment requirements with others is a reasonable thing to do, and an excellent way to learn. However, the work you hand-in must ultimately be your work. This is essential for you to benefit from the learning experience, and for the instructors and TAs to grade you fairly. Handing in work that is not your original work, but is represented as such, is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. **Collaborative coding is strictly prohibited.** Your assignment submission must be strictly your code. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing code itself, or modelling code after another student's algorithm. **You can not (even with citation) use another student's code.**
2. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code that it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's console, then this code is not yours.
3. Everything that you hand in must be your original work, except for the code copied from the textbook, lecture material (i.e., slides, notes), web, or that supplied by your TA. When someone else's code is used like this, you must acknowledge the source explicitly, citing the sources in a scientific way (i.e., including author(s), title, page numbers, URLs, etc.)
4. Cite all sources of code that you hand-in that are not your original work. You can put the citation into comments in your program. For example, if you find and use code found on a web site, include a comment that says, for example:
   ```
   # the following code is from
   https://www.quackit.com/python/tutorial/python_hello_world.cfm.
   ```
   Use the complete URL so that the marker can check the source.
5. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. However, you may still get a low grade if you submit code that is not primarily developed by yourself.
6. We will be looking for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
7. Copying another student's work constitutes academic misconduct, a very serious offense that will be dealt with rigorously in all cases. Please read the sections of the University Calendar under the heading "Student Misconduct". If you are in doubt whether a certain form of aid is allowed, ask your instructor!

Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor to get help than it is to plagiarize.

Late Penalty:

LATE ASSIGNMENTS WILL NOT BE ACCEPTED

Goal

The goal of this assignment is to implement and compare different algorithms that solve the same problem. In this case, the problem is to compute the n^{th} Fibonacci number.

Fibonacci numbers are defined by the Fibonacci numbers – denoted F_n – form a sequence where each number is the sum of the two preceding ones, starting from 0 and 1. Fibonacci numbers are defined as follows:

$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2,} & n > 1 \end{cases}$$

which generates the Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … The algorithms range from one based on the recursive definition above to a more sophisticated algorithm that runs much faster for large values of *n*.

The Algorithms

Here are descriptions of the five algorithms.

Algorithm 1: fibRec(n)	Compute F_n using the **rec**ursive definition given above using recursive function calls.
Algorithm 2: fibMem(n)	Compute F_n using **mem**oization in an array. Memoization means to create an array to 'cache' the results of previous recursive calls. Pass this array down the recursive function calls. Before doing a recursive call for some n-1 or n-2 check if a value exists in the 'cache'. If it does use it, if not do the recursive call and after it returns store the value in the cache. (array should only be maintained through recursion process and should be new for each new execution of algorithm)
Algorithm 3: fibDyn(n)	Compute F_n using an array from the bottom up (**dyn**amic programming). Create an array to store the answer in (size n+1). Store 0 and 1, then loop upwards to n, assigning each subsequent value.
Algorithm 4: fibIter(n)	Initialize two variables (no array) to represent the first two numbers in the Fibonacci sequence. Then, in an **iter**ative loop, compute each member of the sequence from the previous two. Stop when the n^{th} number is reached.
Algorithm 5: fibMat(n)	Using **mat**rix exponentiation (next page):

Function **FibMat**
Input: n (integer);
Output: F_n (integer);

Begin

If (n = 0)

return 0;

Initialize $FM = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

Call **MatrixPower** (n – 1, FM);

Return FM[0][0];

End

Procedure **MatrixPower**
Input: n (integer), FM(2x2 matrix);
Output: *none*

Begin
If (n > 1)
Begin
Call **MatrixPower** (n / 2);
// matrix multiplication
Update $FM = FM * FM$

If (n is **odd**)
// matrix multiplication
Update $FM = FM * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

End
End

Instructions

1. Write a Java program that implements each of the five algorithms as a function that computes the n^{th} Fibonacci number. It is recommended that you get your program to print a table that allows you to verify that each of the algorithms is coded correctly as you develop them. *(If you want to calculate higher Fibonacci numbers you may want to explore using long, or other, Java storage structures for numbers. This is not required, and if you overflow the type you can still continue and pretend your calculation result is still valid.).*

2. Modify your program so that it can measure the execution time for each of the algorithms. Make sure to measure the performance of your algorithms for different ranges of Fibonacci sequences F_n. For instance, from F_0 to F_{100}, carefully observing the performance of each of the three algorithms for different intervals – e.g., for *n* < about 20, for 20 < *n* <= 50 (approximately), and for all *n* larger than that. **(Ex. you may find that you have to pick a different upper limit for input n for Algorithm 1 versus the next 4)**. Your ranges may have to go much higher than 100, like the powers of 2 seen in lecture.

   You will likely find that computing multiple runs for each algorithm and averaging them, as well as inter-leaving the execution of each algorithm can help reduce the process/memory-based execution time disruptions that will become evident in your timing attempts. However, these concerns should be reduced as the size of n increases.

Your program should print out data in the following format:

0 time to compute F_0 with alg. 1
1 time to compute F_1 with alg. 1
⋮
blank line
2 time to compute F_0 with alg. 2
3 time to compute F_1 with alg. 2
⋮
blank line
4 time to compute F_0 with alg. 3
5 time to compute F_1 with alg. 3
⋮
etc.

3. Use **gnuplot** (http://www.gnuplot.info/), Excel, Google Sheets, R, or a similar program to produce plots showing execution time for each algorithm versus *n*. Note that to get illustrative plots useful, you may have to alter the range of *n* used for each algorithm, the number of values of *n* that are tried, and so on, as described above in item (2). It is up to you to explore your data!
4. Answer the questions at the bottom of this document in a report.

NOTE:	The functions **System.currentTimeMillis()** and **System.nanoTime()** are handy for timing execution. **Make sure you experiment with both functions!** See Section 4.1, Code Fragment 4.1 in the textbook for examples using System.currentTimeMillis() – also refer to the lecture slide set "1Analysis", slide 18.
	Your functions may run faster than the resolution of the CPU clock. In that case, you must execute the function several times and divide the time by the number of executions to get a good measurement of the time.

Hand-In (digitally to D2L dropbox)

1. A written report (PDF format) with your name and ID number in the grading summary, including the answers to the questions and your plots. Keep your answers short and to the point. The maximum length for a report is 2 pages plus a maximum of **five plots**.

2. Your source code file that generates the data you used for your plots (in specified format [Java8]). The README.md file should include description of which class begins execution of your submitted code. For example, "CPSC319W20A1.java" is my main class and should be executed, after compiling with "Makefile", with no arguments.

3. Include all of the above items in a zipped folder (standard ZIP) titled
   CPSC319W20A1-LASTNAME-ID.zip
   before submission to the **D2L dropbox** for assignment 1.

Grading

- Assignment grades will be base equally on the two submitted components as described on grading summary

- Source code that does not compile with the supplied 'Makefile' (zipped folder “CPSC319W20A1-Makefile”) or produces run-time errors will receive a grade of 0%.

- Code that produces incorrect output (Ex. wrong values for Fibonacci sequence) will be penalized appropriately.

- Your written answers should include well structured sentences, proper grammar, and arguments of reasoning that can be understood. Giving singular formula answers, single word statements, or leaving details implicit will be marked accordingly. It is not the job of the TA to assume you meant something you didn’t say.

The conversion between a percentage grade and letter grade that will be posted online is as follows.

Letter	A+	A	A-	B+	B	B-	C+	C	C-	D+	D	F
Min. Perc.	95%	90%	85%	80%	75%	70%	65%	60%	55%	50%	45%	Below 45%

Assignment grading and questions continue on the next pages.

GRADING SUMMARY

Name: _____

ID: _____

PROGRAM:		/20
REPORT:		/20
TOTAL		**/40**

PROGRAM

Program builds correctly	Yes	No
Runs: completes with zero run-time errors	Yes	No

IMPORTANT: Source code that does not compile or produces run-time errors will receive 0%

Criteria	TOTAL
Functionality: produces correct output	
Readability: code is clear and easy to read	
Modularity: code is easy to extend, or use	
Generality: easy to extend to more sophisticated applications	
Documented: code classes/functions/inline commented	
TOTAL:	**/20**

REPORT

Criteria	TOTAL
Plots: show important trends in data	/4
Question #1/#2/#3:	/4
Question #4/#5/#6:	/4
Question #7:	/4
Question #8:	/4
TOTAL:	**/20**

Questions

1. Concerning algorithm 1:
    a. Does it perform redundant calculations (yes or no)?
    b. Draw its recursion tree for the computation of F_6
    c. How many times are F_2, F_3, … evaluated (recursion called) in the computation of F_6?

2. Concerning algorithm 2:
    a. Does it perform redundant calculations (yes or no)?
    b. Draw its recursion tree for the computation of F_6
    c. How many times are F_2, F_3, … evaluated (recursion called) in the computation of F_6?

3. Cost-benefit comparison of algorithm 1 and algorithm 2 based on Q1 and Q2.

4. What is the big-O running time for algorithm 3? Why?

5. What is the big-O running time for algorithm 4? Why?

6. Cost-benefit comparison of algorithm 3 and algorithm 4 based on Q4 and Q5.

7. What is the big-O running time for algorithm 5? Why?

8. Under what situations would you use each of the algorithms? Answer your question according to the experimental studies results (i.e., after computing the timing execution). You don't have to repeat verbatim answers to Q3/Q6 here but should likely reference them in making your determinations.