Naive vs Fast ln Approximation

Out: 1/16
Due: 1/28 by 11:59 PM

Learning Objectives

- ⊙ Setting up your Programming Environment and Tools
- ⊙ Review of the Use of Control Structures
- ⊙ Empirical Analysis of Two ln Series Approximation Algorithms
- ⊙ Familiarization with the Course Programming Conventions

The purpose of the first programming exercise is to familiarize you with the programming standards and coding conventions that are required for this course. All students taking this course are expected to have a certain level of proficiency which includes the ability to read and understand code written by others. Students are also expected to write code that efficiently implements and tests various algorithms using programming constructs studied in your programming classes and methods whose syntaxes and usage information you can find in the Java™Platform Standard Edition 8 API documentation.

In this exercise and all subsequent programming projects, you will be expected to properly document your code using the coding conventions for this course, program in multiple files so that there is a wall between the public interface and implementation of your algorithms as well as the application that uses the implementation. This assignment will test your ability to follow this approach to programming no matter what IDE (integrated development environment) that you use to write your code. Your code will be tested using Java™Platform Standard Edition 8. Your program will be considered acceptable for grading only if it compiles without any syntax errors. You will generally submit only your source code, the .java files, for grading in a zip archive file via a drop box on Moodle.

Definition 1. Empirical algorithmics is a computer science area of specialization that involves the use of experimental and statistical, rather than theoretical, methods to analyze the behavior of algorithms.

The goals of empirical algorithmics are characterization of an algorithm based on its performance, enhancing the performance of an algorithm using empirical techniques and the comparative analysis of the various algorithms that solve the same problem within a given context.

A Naive Power Algorithm

Definition 2. A **power** is an exponent to which a given quantity is raised. The expression x^n is therefore known as "x to the n^{th} power." Computations involving powers arise in many numerical algorithms.

You will write code to implement a naive power algorithm whose base is a real number and whose exponent is an integer.

base(b)	exponent(n)	b^n
0	n is less than or equal to 0	indeterminate
0	n is greater than 0	0
b	0	1
1	n	1
b	1	b
b	-1	1/b
-1	n is even	1
-1	n is odd	-1
b	n is positive	$\underbrace{b \times b \times b \times \cdots \times b}_{n\ factors}$
b	n is negative	$\frac{1}{\underbrace{b \times b \times b \times \cdots \times b}_{n\ factors}}$

Table 1: Cases When Computing Integer Powers

Natural Log Series Approximation

The natural logarithm of a number x, denoted $\ln(x)$ where $x > 0$, is given by the series below.

$$\ln(x) = 2\sum_{i=1}^{\infty}\left[\frac{1}{2i-1}\left(\frac{x-1}{x+1}\right)^{2i-1}\right] \qquad \text{(eq1)}$$

You will write a program to empirically compare the performance of a *naive* and a *fast* algorithm for approximating the natural logarithm of numbers using the series in Equation eq1.

For the *naive* algorithm, you will compute the series up to its first n terms using a loop. For each term you will evaluate the summand $\left[\frac{1}{2i-1}\alpha^{2i-1}\right]$, where $\alpha = \frac{x-1}{x+1}$. To evaluate α^{2i-1} for each term, use your implementation of naive power method described in Table 1. Do not use any standard Java$^{\text{TM}}$Math class method in this project.

For the *fast* algorithm, you will also compute the series up to its first n terms using a loop. However, except for the first term, you will generate each term from its predecessor without your user-defined power method, any standard Math class method or a nested loop. It might be useful to expand the series for a few terms to see how the α component of a term can be derived from the α component of its predecessor. Also, observe how the denominator of the coefficient of the α component of a term can be derived from the denominator of the coefficient of the α component of its predecessor. As each succeeding term is generated they are added to compute the series.

The NaturalLogger Class

Complete the implementation of *NaturalLogger.java* so that it contains implementations for five static methods: an implementation for the *naive* power algorithm described in Table 1, an implementation for the *naive* ln algorithm and its wrapper method and an implementation for the *fast* ln algorithm and its wrapper method. See the file for additional details on each method.

The NaturalLogProfiler Program

Create the NaturalLogProfiler.java source file consisting of only one class that contains one method, the *main*. The *main* will perform these tasks:

1. It prompts the user to enter a number whose natural logarithm is to be approximated.

2. It computes an approximation for the natural log of the number using versions of the fast and naive ln method with an arity of one and displays approximate natural logs.

3. It randomly generates a three-digit positive integer.

4. Similarly, it computes an approximation for the natural log of the randomly generated three-digit number using both the fast and naive methods with an arity of one and displays the approximate natural logs.

5. It prompts the user to enter a real number whose approximate natural log will be calculated using varying numbers of terms of the series in Equation eq1 with your implementations of the fast and naive algorithms. It displays the runtimes for generating various approximations.

6. For number of terms $n \in \{1000i, i \in \mathbb{Z} \,|\, 1 \leq i \leq 10\}$, the program will compute, using the naive and fast algorithms, approximate natural logs of the number entered by the user and measure and display the execution times in nanoseconds for these algorithms as shown in the table in Listing 1.

.

Additional Requirements

Test the program to ensure that it works correctly and generates its output in the same format as shown in the sample run. You may use the natural log calculator at https://www.symbolab.com/solver/logarithms-calculator to obtain the exact value for the natural log of a number.

Complete the preamble documentation in the starter code. Do not delete the GNU GPL v2 license agreement in the documentation, where applicable.

```
/**
 * EXPLAIN THE PURPOSE OF THIS FIlE
 * @author YOUR NAME
 * @see A LIST OF FILES THAT IT DIRECTLY REFERENCES
 * <pre>
 * CSC 3102 Programming Project # 0
 * Date: DATE PROGRAM WAS LAST MODIFIED
 * </pre>
 */
```

I have provided a spreadsheet template *lntimes.xls*. Copy the data generated by your into the spreadsheet so that it creates a line graph of execution times of each algorithm in nanoseconds (Y-axis) versus the number of terms used in generating the approximation (X-axis). When generating the Excel plot, use the data generated when $x = 872.53$ is used as input for the table. Locate the source files, *NaturalLogger.java*, and *NaturalLogProfiler.java* and enclose them along with the spreadsheet in a zip file. Name the zip file *YOURPAWSID_proj00.zip*, and submit your project for grading using the digital drop box on Moodle. YOURPAWSID is the prefix of your LSU email address, left of the @ sign.

Listing 1: Sample Run

```
Enter a number to approximate its natural log -> 15
Naive Algorithm: ln(15.0) ~ 2.708050
Fast Algorithm: ln(15.0) ~ 2.708050

For a random 3-digit positive integer:
Naive Algorithm: ln(952) ~ 6.858565
Fast Algorithm: ln(952) ~ 6.858565

Enter x to generate approximations for ln(x) -> 872.53

x = 872.53
```

#Terms	Fast ln(x) (ns)	Naive ln(x) (ns)
1000	.....	.....
2000	.....	.....
3000	.....	.....
4000	.....	.....
5000	.....	.....
6000	.....	.....
7000	.....	.....
8000	.....	.....
9000	.....	.....
10000	.....	.....