Naive vs Fast Exponentiation of Big Integers

Out: 1/10
Due: 1/24 by 11:59 PM

Learning Objectives

- ⊙ Setting up your Programming Environment and Tools
- ⊙ Review of the Use of Control Structures
- ⊙ Empirical Analysis of Two Integer Exponentiation 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 functions whose syntaxes and usage information you can find in ANSI/ISO C++ 11 language 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 ANSI/ISO C++ 11. 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 .cpp and .h files, in a zip archive file via a drop box on Moodle for grading.

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.

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." The computation of powers arise in many numerical algorithms.

You will write code to empirically compare the performance of a naive and the *fast* algorithms for computing powers of positive big integer values where the exponent is a non-negative integer. Here is the pseudocode for a naive power algorithm,

Listing 1: Naive Power Algorithm

```
ALGORITHM: NAIVE-POWER(x,n)
   {Input: n - a non-negative integer
           x - a positive integer
    Output: $x^n$
   }
   if n = 0 or x = 1
      return 1
   endif
   y ← 1
   for i  ← 1 to $n$ stepSize ← 1
      y  ← $y \times$ x
   endfor
   return y
endAlgorithm
```

The power of a number can be determined more efficiently using a successive squaring algorithm.

Here is a pseudocode description of the algorithm.

Listing 2: Fast Power Algorithm

```
ALGORITHM: FAST-POWER(x,n)
   {Input: n - a non-negative integer
           x - a positive integer
    Output: $x^n$
   }
   if n = 0 or x = 1
      return 1
   endif
   y ← 1
   while n > 1
      if n mod 2 = 0
         n ← n / 2
      else
         y ← y × x
         n ← (n - 1) / 2
      endif
      x ← x × x
   endwhile
   return y × x
endAlgorithm
```

I have provided *BigInt.h* a header file that contains prototypes for functions of a "Big Integer" library. I have also provided *BigInt.cpp* the implementation file for the header file. These files allow for representing very large integers and performing basic arithmetic operations on them. Do not make any changes to these files.

The BigMath Header File

Complete the implementation of *BigMath.h* so that it contains implementations for the naive and fast exponentiation algorithms.

The PowerAnalyzer Program

The PowerAnalyzer.cpp file will consist of only one function, the *main*. The main function will perform the following tasks:

1. It prompts the user to enter the base of a power.
2. It prompts the user to enter the exponent of a power.
3. It computes the powers using both the fast and naive algorithms and displays the powers.
4. It randomly generates a four-digit positive integer for the base of a power.
5. It randomly generates a two-digit positive integer for the exponent of a power.
6. Similarly, it computes the powers using these randomly generated integers and both the fast and naive algorithms and displays the powers.
7. It prompts the user for a positive integer to be used as the base in measuring and displaying runtimes for generating various powers of the base.
8. For n = $\{16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$, your program will compute, using the naive and fast exponentiation algorithms, powers of the base entered by the user and measure and display the execution times in nanoseconds for these algorithms as shown in the table in the sample run.

.

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. A *Big Integer* calculator at *http://www.javascripter.net/math/calculators/100digitbigintcalculator.htm* can be used to verify that the powers calculated by your program are correct.

Each file should have the following javadocs:

```
/**
 * EXPLAIN THE PURPOSE OF THIS FIlE
 * CSC 3102 Programming Project # 0
 * @author YOUR NAME
 * @since DATE THE FILE WAS LAST MODIFIED
 * @see A LIST OF FILES THAT IT DIRECTLY REFERENCES
 */
```

Also, submit an excel spreadsheet, *powertimes.xls*, containing the data generated by your program and a line graph of execution times of each algorithm in milliseconds (Y-axis) versus the integer exponent (X-axis). Locate the source files, *BigMath.h*, *BigInt.h*, *BigInt.cpp* and *PowerAnalyzer.cpp* and enclose them along with the spreadsheet in a zip file. Name the zip file *YOUR-PAWSID_proj00.zip*, and submit your project for grading using the digital drop box on Moodle.

```
Sample Run:

Enter the base of the power -> 2
Enter the exponent of the power -> 100

Using Naive Algorithm: 2^100 = 1267650600228229401496703205376
Using Fast Algorithm: 2^100 = 1267650600228229401496703205376

Using a random 4-digit base and a random 2-digit exponent:
Using Naive Algorithm: 6362^24 = ....
Using Fast Algorithm: 6362^24 = ....

Enter the base of the powers for the table -> 546879

b = 546879
=========================================================
n             b^n: Fast Power(ns)   b^n: Naive Power (ns)
---------------------------------------------------------
16                  .............            ............
32                  .............            ............
64                  .............            ............
128                 .............            ............
256                 .............            ............
512                 .............            ............
1024                .............            ............
2048                .............            ............
4096                .............            ............
8192                .............            ............
=========================================================
```