Programming Assignment: Exponentiation with Big Integers
CSC 207, “Algorithms and Object-Oriented Design”
Department of Computer Science
Grinnell College
December 3, 2018

The goal of this assignment is to design, write, and test a Java program that calculates optimal methods for performing exponentiation on values of the BigInteger data type.

This programming assignment will be due at the beginning of class on Monday, December 10.

To submit your work log in on MathLAN and open a terminal window, create a new directory within your home directory, copy or move the files you want to submit into that directory, and then run the command

```
/home/reseda/executables/submit-207 directory-name
```

putting the name of the directory you created in place of directory-name.

Computing Large Powers

The java.util.BigInteger class includes a method, pow, for raising BigInteger values to arbitrary non-negative powers. When the exponent is also large, computing the result by iterated multiplication, as in

```java
BigInteger result = BigInteger.ONE;
for (int step = 0; step < exponent; step++)
    result = result.multiply(base);
```

is unnecessarily slow. For most exponents, it is faster to combine some multiplications by the base with some squaring operations. For instance, to compute the 18th power of base, we might use a plan based on the following observations:

\[
\text{base}^{18} = (\text{base}^9)^2 \\
= (\text{base}^8 \times \text{base})^2 \\
= ((\text{base}^4)^2 \times \text{base})^2 \\
= (((\text{base}^2)^2)^2 \times \text{base})^2
\]

or, in Java,

```java
BigInteger result = BigInteger.ONE;
result = result.multiply(base);
result = result.multiply(result);
result = result.multiply(result);
result = result.multiply(result);
result = result.multiply(base);
result = result.multiply(result);
```

We get the same result, but we perform only six multiplications instead of eighteen.
Drawing Up the Plan

Of course, this means that we have to figure out, for each separate exponent, an efficient plan for interleaving squaring operations and multiplications by base to get exactly the exponent we want. In some cases, it may not be obvious which of the possible plans is the most efficient.

If we have a lot of exponentiation problems to do, involving many different exponents, one approach to finding efficient is to set up a directed graph in which the vertices are possible exponents and each edge connects an exponent either to the next greater exponent (representing a multiplication by the base) or to an exponent twice as large (representing a squaring operation). So, for instance, the exponent 93 would have two edges leading away from it — one to the exponent 94, and one to the exponent 186. Of course, to keep the graph finite, we would have to impose a maximum on the exponents that can be represented inside the graph, and leave out any edges that would take us to unrepresented exponents. (For instance, if we set a maximum of 1000, we wouldn’t have an edge from 647 to 1294, because there would be no vertex for 1294.)

Finding the most efficient plan would then simply entail finding the shortest path from vertex 0 to the desired exponent in this graph.

Determining the Edge Weights

Normally, we assume that multiplication is a constant-time operation, which would correspond in our graph representation to the assumption that all the edge weights should be the same. But this isn’t true in the BigInteger type. It’s much more likely that the running times of the multiplications that we are contemplating depend on the sizes of the numbers that we are trying to multiply together.

A more realistic way to estimate the running time for a multiplication by the base would be to label each edge from an exponent $e$ to the next greater exponent $e + 1$ with $e$ itself. And a more realistic way to assess the running time for a squaring operation would be to label each edge from an exponent $e$ to its double $2e$ with $e \cdot (1 + \log_2 e)$, reflecting the complexity of a fast algorithm for multiplying large numbers. This definition doesn’t work for 0, since $\log_2 0$ is undefined, but there’s no reason to have a “squaring” edge from 0 to itself anyway.

The Assignment

Design and implement a Java program that builds an exponent graph, with appropriate edge weights determined by the formulas shown above, for exponents ranging from 0 to 1000, and then finds and prints the shortest path from vertex 0 to every other vertex in the graph.