Project 1. Efficient Polynomial Evaluations

(Due 10/27/Sun)

Description:

In this project, we will compare *three* different algorithms that are used to *evaluate polynomials*. The goal is to *understand the importance of the efficiency of an algorithm*. The first algorithm is the *brute force method* in which we evaluate polynomials in the usual way. The second algorithm applies the *repeated squaring technique* to calculate the *monomials* and by adding up all the terms to get the value of the whole polynomial. The third algorithm uses the *Horner's Rule* to evaluate polynomials. The polynomials to be evaluated have the following general form:

$$P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n.$$

Requirements:

1. (*Format of your program*)

   Three programming languages are accepted: Java, Python, and C/C++. When you submit your project, place your program or programs and the screenshots of the experiments in one folder with your name and appending `Proj1, P1, Project1`, etc. as the folder name, then zip your folder and submit it.

2. (*Observe the overflow problem*)

   When you do computation in a computer, an *overflow* problem may occur without any warning or error message. If we do not pay attention to the potential *overflow problem*, we may get an *incorrect* result without being aware of it. In this project, we would like to experience this *overflow* problem, so that we will prevent this type of errors in our future programming. In order to expose the overflow problem, we choose some special polynomial for the experiment. Let $P_n(x)$ be the following polynomial:

   $$P_n(x) = 1 + x + 2x^2 + 3x^3 + \cdots + nx^n$$

   with n and x to be determined by experiments. Here we choose x as a positive `long` integer. From the formula of $P_n(x)$ above, we can see that $P_n(x)$ should be a positive integer. If you get a negative number for $P_n(x)$, you know that something must be wrong. When you take `n = 25` and `x = 123`, you would get a negative result for $P_n(x)$, which is due to the overflow problem. For those students using Python, you may need to do a simulation for the `long` integers or try some very large numbers to experience the overflow problem. To show that you have done this feature, you only need to output a negative value of $P_n(x)$ for some special data for n and x other than `n = 25` and `x = 123` in a screenshot.

3. (*Eliminate the computation errors and overflow*)

   If we use the *floating-point* numbers for the coefficients of the polynomials and the variable x, we would get computation errors due to the truncation problem of computers. As a result, these three methods usually produce slightly different numbers. In order to make sure that the same polynomial produces the same computation result for these three methods, we take integers for the coefficients of the polynomials and the variable x. However, for a high degree polynomial, even if we use the `long` integer type, we can easily encounter the *overflow* problem. To avoid the *overflow* problem, we apply the Java class `BigInteger`, which supports integer computations with any number of digits. (Note, in Python, the overflow problem for the integer computation has been handled gracefully. You have some special way to detect the overflow. For C/C++, a special header file should be included to avoid the overflow problem.)

4. (*Data description: The polynomial coefficients and the point to be evaluated at*)

   We use two parameters: n and d to control the data we use in the polynomials. We use n for the degree of the polynomial and d for the number of digits (*or bits if it is more convenient*) for the coefficients of the polynomial and the variable x. You can set the values for n and d in your code. In the first round of experiments, your goal is to make sure that your implementations of the three methods are correct. To this end, you only need to use relatively small numbers for n and d, and you can generate the data either on the fly or even using hard-coded data in your program. After you fix all the errors, you do the second round of experiments with relatively large numbers for n and d. In this way, we can see the significant differences in their response times of these three methods.

5. (*Generate the data and store it in an external text file*)

   You can write a function (or a method) to generate the data for the second round of experiments. In order to use the same data in the three different algorithms, we would like to output the generated data into an external text file, so that we can load the data from the same data file before we run each of the three different algorithms. You can choose your own way to store the integer numbers in the text data file. If you want to replace the old data with a new set of data, you just run this function (or method) again. In this way, we can apply the three different algorithms on the same set of data (the polynomial and the variable x) all the time.
   Note: We run these three methods separately, not in sequence to avoid the interference among these three methods. To ensure that they work on the same data, we use the external data file.

6. (*Output and Screenshots*)

   Our main goal is comparing the efficiencies of these three algorithms. To this end, we will make our *user's interface* as simple as possible. You do not need to ask the

user to enter any input. You can output your results on the screen and take the screenshots to show your experiment results.

Specifically, your first experiment result is showing the *overflow* problem as described in Item 2 above, including the values for n and x. Then you do the first round of experiments using relatively small input numbers, for example, $n = 5$ and $d = 2$ or hard-coded coefficients and the value of x. You need to print out the polynomial value $P_n(x)$ and the response time for each algorithm in milliseconds. Since the three result numbers are not too big, we can easily see if they are the same or not.

For the second round of experiments, you run each algorithm on relatively large input numbers from the same data file generated in Item 5 above. How large should those numbers be? You control the numbers for n and d, such that your first algorithm (the brute force method) takes more than 2 seconds and less than 5 seconds. Then you also print out the result numbers and the response times to see their big differences in efficiency. Since the result numbers have a large number of digits, it is not easy to compare if the three big numbers are the same. You can use a function to compare any two big numbers digit by digit (or directly in value), with a returned boolean value to tell if they are the same or not. (Note: This is an option, not required, with no impact on your grade.)

7. (*On the built-in power function*)

   When you implement the *Brute Force Method*, do not use the built-in power function (e.g. `Math.pow()` in Java) directly, because the power function is implemented using the *repeated squaring* method internally. You need to use a for-loop to calculate the monomial x^k instead of using the power function for the first method. In your second method, you need to implement the *repeated squaring* method explicitly (you are allowed to use the *recursion* in this implementation).

   Please include the screenshot(s) for your experiment output in your project folder so as to make the grading faster.

Any updates about this project will be posted in Canvas as announcements.