CSC 360 Assignment # 5
Total Points: 40

Computing Powers

This problem is a case study in using recursive thinking to improve the efficiency of an iterative algorithm. You will write a sequence of methods for the exponentiation of floating point numbers. Parts I and II involve writing rather simple-minded iterative and recursive methods for the task. In Parts III and IV you use more sophisticated recurrence relations in order to write more efficient recursive methods. In Part V you convert your method from Part IV into an iterative method that is far more efficient than the Part I iterative method.

You may put all of your code for this project into a **single class file – PowersUsername.java**. All of your methods should be static.

Part I:
Write an iterative method called *power1* to compute b^n, where b is of type double and n is an integer ≥ 0. Use a simple for-loop that repeatedly (n times) multiplies an accumulator variable by b.

Part II:
Write a recursive method *power2* that accomplishes the same task as *power1*, but is based on the following recurrence relation:

$b^0 = 1$
$b^n = b * b^{n-1}$ if $n > 0$

Part III:
Write a recursive method *power3* that is identical to *power2* except that it is based on this recurrence relation

b0 = 1
bn = (bn/2)2 if n > 0 and n is even
bn = b * (bn/2)2 if n > 0 and n is odd (Note: This equation is not true in math. Why is it true in Java?)

Note: If n is a large exponent, then *power3* should perform far fewer multiplications than *power2*. In particular, when computing something like $(b^{n/2})^2$, there is no need to compute $b^{n/2}$ twice. Rather, compute it once, store it in a variable, and then compute the result of multiplying the variable by itself.

Part IV:
Write a <u>tail recursive</u> helper method called *multPow*, that computes the value of $a*b^n$. Base your implementation on the following recurrence relation:

a*b0 = a
a*bn = a*(b2)n/2 if n > 0 and n is even
a*bn = (a*b)*(b2)n/2 if n > 0 and n is odd

Then write a method called *power4* that computes b^n simply by making the call *multPow(1, b, n)*. Note that in this approach, the extra parameter *a* used by the helper method is serving as an accumulator for the result.

Part V:
Write an iterative method *power5* to compute b^n. Write it in such a way that the number of multiplications performed by *power5* is no more than the number performed by *power4*. (Hint: Base your solution to Part V on your solution to Part IV. Declare *a* as a local variable that is initialized to 1 and that eventually accumulates the result of the calculation.) Note that, in general, *power5* requires far fewer multiplications than *power1*.

Main method:
Write a main method to test your methods from Parts I – V. It should ask the user for *b* and *n*, then compute and display the results. Call the *Math.pow* method in order to check your results. Display the results from *Math.pow* first. Then display the results from your methods. Also display the number of multiplications performed by each of your methods. In order to count the number of multiplications, you may use a "global variable" that is modified by the power methods as a side-effect, as demonstrated below:

```
public class Powers
{
	private static int multiplications;  // "global variable" for counting the
	                                     //    number    of    multiplications
	                                     performed // by each method
	public static void main(String[] args)
	{
		...
		multiplications = 0;
		System.out.println("\npower1(" + base + ", " + n + ") = " + power1(base,
		n));
		System.out.println("Multiplications = " + multiplications);

		multiplications = 0;
		...
	}

	public static double power1(double base, int n)
	// Returns base to the n-th power.
	// Iterative method.
	{
		...
		for (...)
		{
			multiplications++;
			result *= base;
		}
		return result;
	}
	...
}
```

Note: It is a good idea to test code as you develop it. For instance, I recommend that you write the main method code that tests Part I as soon as (or even before) Part I is finished.

Sample session with a completed program:

Enter a decimal number: 1.001
Enter a non-negative integer exponent: 1000

Computing 1.001 to the power 1000:
Math.pow(1.001, 1000) = 2.7169239322355936

power1(1.001, 1000) = 2.7169239322355985
Multiplications = 1000

power2(1.001, 1000) = 2.7169239322355985
Multiplications = 1000

power3(1.001, 1000) = 2.716923932235485
Multiplications = 16

power4(1.001, 1000) = 2.7169239322355203
Multiplications = 16

power5(1.001, 1000) = 2.7169239322355203
Multiplications = 16

Note:
In Parts I-V, your code should ***not*** call any methods from java.lang.Math. For example, in those methods, if you need to compute b^2, **do not** write Math.pow(b, 2). Instead, write b * b, and count the multiplication.

Think about after Problem 2 is done:
We assumed above that $n \geq 0$. What would be an easy way to handle cases where $n < 0$?

What to turn in:
Submit only the **PowersUsername.java** file on Blackboard, by substituting the Username with your NKU username.