**ECE 4318 Final Part 2 Group**

**Date:** 12/5/20 Saturday

**Due Date:** 12/10/20, Thursday, 12 p.m. (noon)

**Total Points:**  147 points

**Total Grade Points:** 10 grade points, 100 points = 10 grade points

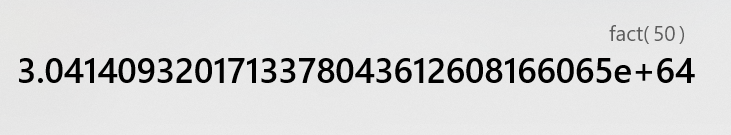
**This is ECE 4318 final part 2. You are allowed to work as group**

**Submit to Blackboard**

**Classes (Object Oriented Programming)**

1. **(97%) Class Huge Integer or Big Integer etc.**

The intrinsic integer data type of C / C++ / C# / Java allows int type to be of signed 4 bytes (with maximum = 231– 1 = 2,147,483,647 or about 2 billions) and long type to be of signed 8 bytes (with maximum = 2 63– 1 = 9,223,372,036,854,775,808 of 19 digits). If you try to calculate 50! (fifty factorial), C/C++/C#/Java will show overflow (so is MATLAB, Excel etc.) and calculator will show this



Displaying that as a floating number of 64 digits.

Python probably can do the above calculations because Python (and maybe Javascript) does not fixed data type.

The following is from my C++ manual / book that I wrote in 2013

44! = 2658271574788448768043625811014615890319638528000000000

45! = 119622220865480194561963161495657715064383733760000000000

46! = 5502622159812088949850305428800254892961651752960000000000

47! = 258623241511168180642964355153611979969197632389120000000000

48! = 12413915592536072670862289047373375038521486354677760000000000

49! = 608281864034267560872252163321295376887552831379210240000000000

50! = 30414093201713378043612608166064768844377641568960512000000000000

My 50! was displayed as a big integer (probably with 64 or 65 digits with all the digits displayed, not displayed as a floating number like calculator.

This was based on a class HugeInteger (that I got idea from Deitel long time ago, but they had probably only 20 digits). Java / C++ / C# etc. later on has the idea of a Big Integer class where the number of digits is theoretically unlimited (but you of course *can NOT try to compute 10000000!* Which may take away all your computer resources and also may not be meaningful to do that).

1. (7%) **Build** a class myHugeInteger starting with say an int array of 20 entries (every entry here now represents a digit). Make your constructor and variable definitions etc. (after you add, subtract multiply etc. you’ll find that 20 entries are NOT enough). Note the trivial +-\*/ for integers become a little bit strange when you use arrays where every entry is limited to 0 to 9. You need a constructor that you can set a common integer like 4318 as a myHugeInteger (so 4318 may be represented as either {4, 3, 1, 8} of 4 entries or {8, 1, 3, 4} of 4 entries as a muHugeInteger of 4 entries; or actually the remaining 16 entries are all 0s like {0, 0, 0, …., 4, 3, 1, 8} or {8, 1, 3, 4, 0, 0, …, 0}.
2. (15%) **Add** the method Add to add two myHugeIntegers.

Remember that when you add the ith digit of two Huge Integers , the sum can NOT be bigger than 10 (so if you add 5 from the 6th entry of A and 7 from the 6th entry of B, the sum is 12, which is an overflow or carry, you’ll keep 2, and send 1 to the next higher digit).

When you add two Huge Integers of 20 digits, the sum can be 21 digits, in that case, you need to expand the array from 20 entries to 21 entries.

Also, one or both of the two Huge Integers can be negative. You have to handle the case 5 + (-8) when both are say the 10th entry of HugeInteger A and HugeInteger B

1. (7%) **Add** a similar method Subtract to compute the difference of two myHugeIntegers.
2. (25%) **Add** a harder method **Multiply** to find the product of two myHugeIntegers. You need to think about long multiplication you learned in kindergarten. It is a little bit like convolution in Fourier series etc., every digit can be calculated from many neighboring digits. Also do not forget that when you do A.multiply(B), either A or B can be negative.
3. (20%) **Add** a not that hard but still hard method **Divide** for two myHugeIntegers. Just like 205 / 13 = 15 quotient with remainder = 205 % 13 or 8, the Divide method needs to produce both Quotient and Remainder.
4. (8%)Add a method numberOfDigits to find the number of digits for the big integer or myHugeInteger. For example A.numberofDigits() should return 3 if A is the common integer 720 (= 6!) since 720 is 3 digits long (but 720 is expressed as either {7, 2, 0} or {0, 2, 7} since we use array to express integer here.
5. **(15%+) Test** your class and methods. **Verify** that for smaller numbers, they agree with the common integer for example 4318 + 4317 =8635 using common integer arithmetic; when representing 4318 and 4317 as myHugeInteger, their sum using the add method of part (b) like A.add(B) (where A is *4318* and B is *4317* should show something like 8635 or {0, 0, 0…, 0, 8, 6, 3, 5} or {5, 3, 6, 8, 0, 0, …}. **Verify** your add, subtract, multiply, divide and numberofdigits methods using some small integers.

**Compute** in particular 50! using your Multiply method. Does that agree with my answer.?

Find the number of digits in 50! Using your method.

Now try 100! and if you are brave enough 1000! (one thousand factorial). By Java HugeInteger, 1000! Is about 2500 digits. So do not display that, but compute the number of digits.

1. (0% - 6%) Compare your output with some OOP like Java, C++, C# or even Python.
2. **(50%) Class Polynomial**

Polynomial is actually an array, but every entry is given a different meaning like the power of that entry. For example {1, 2, 3, 4} can mean x3 + 2x2 + 3x + 4 if we order them from the highest power down or 1 + 2x + 3x2 + 4 x3 if we start from the constant term.

Every polynomial also has a degree, the power of the highest power term. So a polynomial f(x) = x3 + 2x2 + 3x + 4 has a power of 3. Constant polynomials are usually assigned a degree of 0. (it is somehow controversial on whether number 0 is assigned degree 0 or minus infinity -∞. We will NOT discuss that here).

MATLAB is especially superior in handling polynomials since polynomials are just arrays, which are matrices (MATLAB stands for Matrices LAB).

1. (4%) Build a class myPolynomial starting with probably a double polynomial of degree 3.
2. (6%) Add the method degree for the polynomial object. Add also a method print or display so that you can print your polynomial on the screen.
3. (4%) Add the method Add to add a second polynomial. Note the sum of two polynomials can have lower degree since the highest degree term may cancel. Note deg (f + g) <= max (deg (f), deg (g))
4. (2%) Add the method Subtract (or Difference)
5. (8%) Add the method Multiply (or product) to compute the product of two polynomials. Note degree (fg) = deg (f) + deg (g). So the starting degree 3 and polynomial array of 3+1 = 4 entries may NOT suffice.
6. (10%) Add the method Division to divide polynomial A by polynomial B. Note we have quotient and remainder as you learned in high school (or kindergarten)
7. (16%) Test your results.

First, test that Multiply and Division work. So if you divide a(x) by b(x) and you get a(x) = b(x) \* q(x) + r(x) using your division method of class myPolynomial, then b(x) \* q(x) using your Multiply from part (e) and then b(x) \* q(x) + r(x) using your Add method from part (c), your b(x) \* q(x) + r(x) should be equal to a(x).

**Use for example a(x) corresponding to 4318 or array / polynomial {4, 3, 1, 8}** and b(x) corresponding to 480 or 3310.

**To test your Multiply, test with (1+x) 3 and (1+x) 10** (that we can verify using Pascal’s triangle).

Note (1+x) 3 just means you multiply 1+x three times using your method Multiply in part (e), so is (1+x) 10 (you probably need a for loop).