Programming Questions:

1) (20 points) We would like to write a function for computing the value of a polynomial of the form:

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_nx^n \quad (1)$$

The naive way of solving this problem is to substitute the values directly in equation (1) and compute the value of p(x).

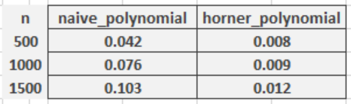
The other way is to use Horner's method to compute p(x) as in equation (2). This algorithm is based on a transform and conquer technique.

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + xa_n)\ldots))) \quad (2)$$

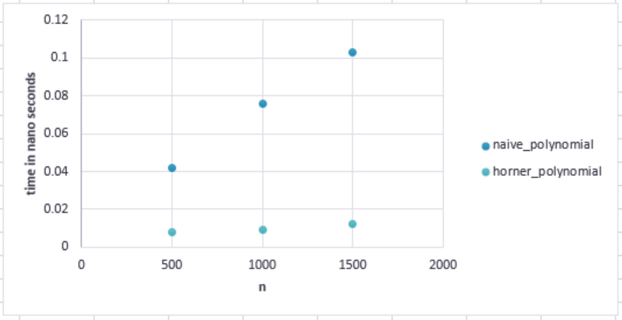
(a) (10 points) Implement the naive method and Horner's method to compute the value of a polynomial and test your code to ensure it's working

```
private static int naive_polynomial(int[] a,int x) {
        int n = a.length;
        double p;
        int pOfX=0;
        for(int i = 0; i < n;i++) {
                p=a[i] * (Math.pow(x,i));
                pOfX+=p;
        }
        return pOfX;
}
private static int horner_polynomial(int[] a,int x) {
        int p=a[a.length-1];
        for(int i = a.length-2; i >= 0;i--) {
                p=a[i] + (x * p);
        }
        return p;
}
```

(b) (10 points) Experimentally, estimate the running time for each of the methods. Note that you will need to experiment with different values of n. You can use arrays to store values of the coefficients (not zero) and then pick a value for x that is not zero. Create a spreadsheet and graph your results



n	naive_polynomial	horner_polynomial
500	0.042	0.008
1000	0.076	0.009
1500	0.103	0.012



2) (20 points)

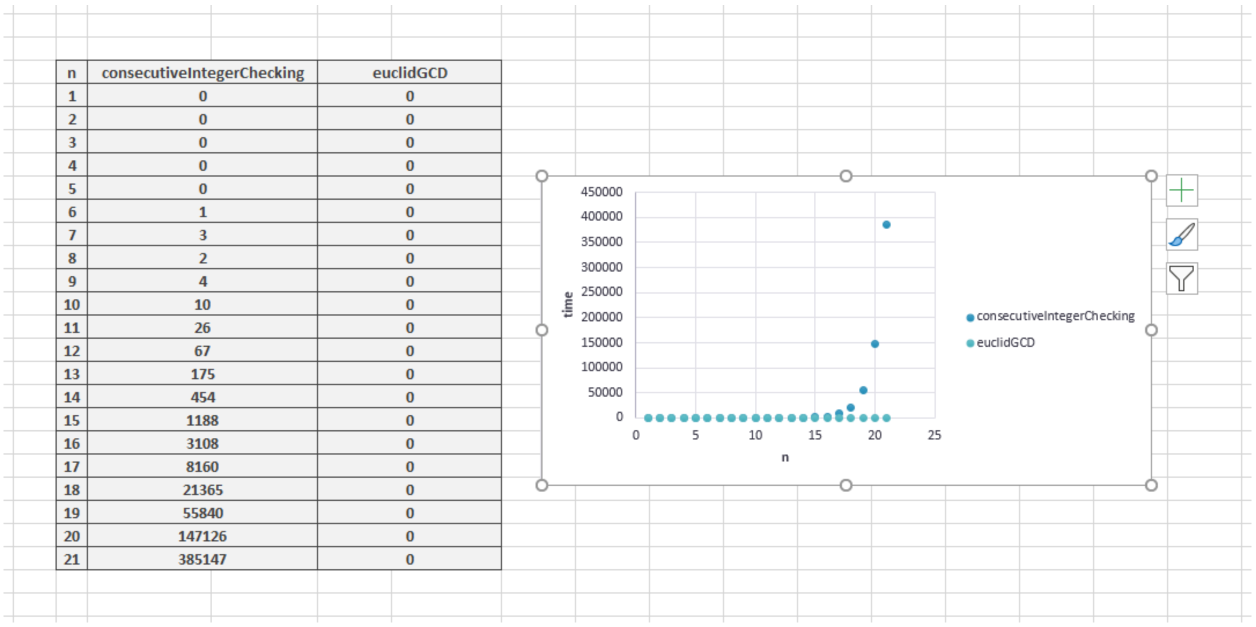
(a) (10 points) Implement the consecutive integer checking and the Euclid's method algorithms for computing the greatest common divisor algorithms described in class.

a)

```
private static long consecutiveIntegerChecking(long m, long n) {
        long t = Math.min(m, n);
        for(long i = t; i > 0;i--) {
                if(t%i==0 && n%i==0) {
                        return i;
                }
        }
        return 1;
}
private static long euclidGCD(long m, long n) {
        if (m == 0) {
                return n;
        }
        return euclidGCD(n%m, m);
}
```

(b) (10 points) Experimentally, estimate the running time for each of the methods. You can use the attached file which contains a list of Fibonacci numbers to compute running times. Testing consecutive Fibonacci values will yield the most time since they are relatively prime. Therefore you should use them to test both algorithms.

The input file provided includes a sequence of Fibonacci numbers. You will read the file and then test gcd method using consecutive Fibonacci numbers. For example gcd(89, 144), gcd(144, 233), gcd(233, 377),...etc. Once you have computed run times for each of these executions you can put them in a spread sheet and graph it.



n	consecutiveIntegerChecking	euclidGCD
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	1	0
7	3	0
8	2	0
9	4	0
10	10	0
11	26	0
12	67	0
13	175	0
14	454	0
15	1188	0
16	3108	0
17	8160	0
18	21365	0
19	55840	0
20	147126	0
21	385147	0

3) (20 points) Implement the two algorithms to compute the value of n x^n. (a) Brute

force: $x^n = \prod x$ This is just the product $(x \times x \times \ldots \times x)$ n times i=1

```
private static int bruteForce(int x,int n) {
            int result=1;
            for(int i = 0; i < n;i++) {
                    result*=x;
            }
            return result;
    }
```

(b) Decrease and conquer algorithm: Apply the following definition of the power until you get to the base case (when $n = 1$)

$x_n = \{x(x(x_{(nn-/21)})_{)/22})_2$ *if inf ins oisdedven}*

You don't need to provide experimental results for run time.

```
private static int decreaseAndConquer(int x,int n) {
        int result=1;
        if(n% 2 == 0) {
                x=x*x;
                for(int i = 0; i < n/2;i++) {
                        result*=x;
                }
        }
        else {
                int originalX=x;
                x=x*x;
                for(int i = 0; i < (n-1)/2;i++) {
                        result*=x;
                }
                result*=originalX;
        }
        return result;
}
```

Measuring Execution Time

Measuring the timing of a single execution of an algorithm is usually imprecise, since the run time (especially for small inputs) may be on the order of the precision of the system timer. For performance measurement, we'll normally run an algorithm many times and then compute the average time for each execution.

Put your method call (the algorithm) in a loop that runs a number of times and take currentTime before and after the loop. Then when determining the running time of the algorithm you divide the time you got by the number of executions. For example

```
long startTime = System.currentTimeMillis(); for(int
i=0; i < 1000; i++)
{
    myAlgorithm(someInputSize);
}
```

```
long endTime = System.currentTimeMillis();
```

Then your runtime would be (endTime - startTime) / 1000.0

Note that this will give you run time for some input size and then you will need to repeat the experiment for different sizes of input in order to produce the graph the data would look something like:

N	run time
500	0.4
1000	0.6
1500	0.8
...	

Then you can copy the data into a spreadsheet to graph the result which will be your runtime function.