Nicholas Colonna

FE 522: Assignment 1

“I pledge my honor that I have abided by the Stevens Honor System.” -ncolonna

Problem 1:

For part a of this problem, I implemented three while loops that terminate once the total amount of rice exceeds the three thresholds of 1000, 1000000 and 1000000000, respectively. The amount of rice for a given square ‘n’ is (n-1)^2, which was added to the total rice count after each pass through the loop. As you can see from the solutions below, it takes 10 squares for 1,000 grains, 20 squares for 1,000,000 grains, and 30 squares for 1,000,000,000 grains.

```
How many squares to give the inventor at least 1000 grains of rice?
Square #1       Rice on Current Square: 1       Rice Total: 1
Square #2       Rice on Current Square: 2       Rice Total: 3
Square #3       Rice on Current Square: 4       Rice Total: 7
Square #4       Rice on Current Square: 8       Rice Total: 15
Square #5       Rice on Current Square: 16      Rice Total: 31
Square #6       Rice on Current Square: 32      Rice Total: 63
Square #7       Rice on Current Square: 64      Rice Total: 127
Square #8       Rice on Current Square: 128     Rice Total: 255
Square #9       Rice on Current Square: 256     Rice Total: 511
Square #10      Rice on Current Square: 512     Rice Total: 1023
```

```
How many squares to give the inventor at least 1,000,000 grains of rice?
Square #1       Rice on Current Square: 1       Rice Total: 1
Square #2       Rice on Current Square: 2       Rice Total: 3
Square #3       Rice on Current Square: 4       Rice Total: 7
Square #4       Rice on Current Square: 8       Rice Total: 15
Square #5       Rice on Current Square: 16      Rice Total: 31
Square #6       Rice on Current Square: 32      Rice Total: 63
Square #7       Rice on Current Square: 64      Rice Total: 127
Square #8       Rice on Current Square: 128     Rice Total: 255
Square #9       Rice on Current Square: 256     Rice Total: 511
Square #10      Rice on Current Square: 512     Rice Total: 1023
Square #11      Rice on Current Square: 1024        Rice Total: 2047
Square #12      Rice on Current Square: 2048        Rice Total: 4095
Square #13      Rice on Current Square: 4096        Rice Total: 8191
Square #14      Rice on Current Square: 8192        Rice Total: 16383
Square #15      Rice on Current Square: 16384       Rice Total: 32767
Square #16      Rice on Current Square: 32768       Rice Total: 65535
Square #17      Rice on Current Square: 65536       Rice Total: 131071
Square #18      Rice on Current Square: 131072      Rice Total: 262143
Square #19      Rice on Current Square: 262144      Rice Total: 524287
Square #20      Rice on Current Square: 524288      Rice Total: 1048575
```

```
How many squares to give the inventor at least 1,000,000,000 grains of rice?
Square #1       Rice on Current Square: 1       Rice Total: 1
Square #2       Rice on Current Square: 2       Rice Total: 3
Square #3       Rice on Current Square: 4       Rice Total: 7
Square #4       Rice on Current Square: 8       Rice Total: 15
Square #5       Rice on Current Square: 16      Rice Total: 31
Square #6       Rice on Current Square: 32      Rice Total: 63
Square #7       Rice on Current Square: 64      Rice Total: 127
Square #8       Rice on Current Square: 128     Rice Total: 255
Square #9       Rice on Current Square: 256     Rice Total: 511
Square #10      Rice on Current Square: 512     Rice Total: 1023
Square #11      Rice on Current Square: 1024        Rice Total: 2047
Square #12      Rice on Current Square: 2048        Rice Total: 4095
Square #13      Rice on Current Square: 4096        Rice Total: 8191
Square #14      Rice on Current Square: 8192        Rice Total: 16383
Square #15      Rice on Current Square: 16384       Rice Total: 32767
Square #16      Rice on Current Square: 32768       Rice Total: 65535
Square #17      Rice on Current Square: 65536       Rice Total: 131071
Square #18      Rice on Current Square: 131072      Rice Total: 262143
Square #19      Rice on Current Square: 262144      Rice Total: 524287
Square #20      Rice on Current Square: 524288      Rice Total: 1048575
Square #21      Rice on Current Square: 1048576     Rice Total: 2097151
Square #22      Rice on Current Square: 2097152     Rice Total: 4194303
Square #23      Rice on Current Square: 4194304     Rice Total: 8388607
Square #24      Rice on Current Square: 8388608     Rice Total: 16777215
Square #25      Rice on Current Square: 16777216        Rice Total: 33554431
Square #26      Rice on Current Square: 33554432        Rice Total: 67108863
Square #27      Rice on Current Square: 67108864        Rice Total: 134217727
Square #28      Rice on Current Square: 134217728       Rice Total: 268435455
Square #29      Rice on Current Square: 268435456       Rice Total: 536870911
Square #30      Rice on Current Square: 536870912       Rice Total: 1073741823
```

For part b, the body of the while loop was identical to part a, but the while loop would terminate once 64 squares of rice were calculated. For the integer loop, I noticed that at square 32, the rice for the current square became negative and the rice total was negative 1. The rice for the current square remained the same all the way until the end, while the total amount of rice kept alternating between -1 and the maximum integer value, 2147483647. You are only able to calculate the exact amount of up to and including square 31, after that, it is no longer possible using integers.

```
Square #55      Rice on Current Square: -2147483648    Rice Total: 2147483647
Square #56      Rice on Current Square: -2147483648    Rice Total: -1
Square #57      Rice on Current Square: -2147483648    Rice Total: 2147483647
Square #58      Rice on Current Square: -2147483648    Rice Total: -1
Square #59      Rice on Current Square: -2147483648    Rice Total: 2147483647
Square #60      Rice on Current Square: -2147483648    Rice Total: -1
Square #61      Rice on Current Square: -2147483648    Rice Total: 2147483647
Square #62      Rice on Current Square: -2147483648    Rice Total: -1
Square #63      Rice on Current Square: -2147483648    Rice Total: 2147483647
Square #64      Rice on Current Square: -2147483648    Rice Total: -1
```

When using doubles for the same loop, the behavior was slightly different. Starting at square number 20, the values become so large that the program begins using scientific notation, expressing numbers such as 1.04858e+06. This behavior continues all the way through square 64, which means you are able to calculate the approximate number of total grains for all squares in this problem, including 64 (the largest). I ran a test to see the maximum it could go beyond this problem, which I found to be 1023 squares, after that, the rice total becomes 'inf'.

```
Square #55      Rice on Current Square: 1.80144e+16    Rice Total: 3.60288e+16
Square #56      Rice on Current Square: 3.60288e+16    Rice Total: 7.20576e+16
Square #57      Rice on Current Square: 7.20576e+16    Rice Total: 1.44115e+17
Square #58      Rice on Current Square: 1.44115e+17    Rice Total: 2.8823e+17
Square #59      Rice on Current Square: 2.8823e+17     Rice Total: 5.76461e+17
Square #60      Rice on Current Square: 5.76461e+17    Rice Total: 1.15292e+18
Square #61      Rice on Current Square: 1.15292e+18    Rice Total: 2.30584e+18
Square #62      Rice on Current Square: 2.30584e+18    Rice Total: 4.61169e+18
Square #63      Rice on Current Square: 4.61169e+18    Rice Total: 9.22337e+18
Square #64      Rice on Current Square: 9.22337e+18    Rice Total: 1.84467e+19
```

```
Square #1015      Rice on Current Square: 1.75556e+305      Rice Total: 3.51112e+305
Square #1016      Rice on Current Square: 3.51112e+305      Rice Total: 7.02224e+305
Square #1017      Rice on Current Square: 7.02224e+305      Rice Total: 1.40445e+306
Square #1018      Rice on Current Square: 1.40445e+306      Rice Total: 2.8089e+306
Square #1019      Rice on Current Square: 2.8089e+306    Rice Total: 5.61779e+306
Square #1020      Rice on Current Square: 5.61779e+306      Rice Total: 1.12356e+307
Square #1021      Rice on Current Square: 1.12356e+307      Rice Total: 2.24712e+307
Square #1022      Rice on Current Square: 2.24712e+307      Rice Total: 4.49423e+307
Square #1023      Rice on Current Square: 4.49423e+307      Rice Total: 8.98847e+307
Square #1024      Rice on Current Square: 8.98847e+307      Rice Total: inf
```

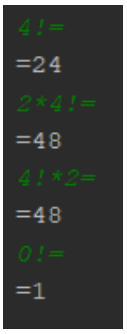
Problem 2:

For part a, the first step for adding {} to the calculator was adding a case for each bracket to the get() function where all of the other symbols are. The second and final step for adding the brackets was in the primary() function. I added a case for {, which was identical the case for (, except I replaced all of the references of parenthesis to brackets. This adequately added handling for brackets as well as checking to ensure all brackets have a matching closing pair.

```
{(4+5)*6}/(3+4)=
=7.71429
{{1+1}}=
=2
{2+2=
Please enter a character to exit
error: '}' expected
```

For part b, the first step to adding the factorial operator to the calculator was adding a case for ! to the get() function where all the other symbols are. To adjust the grammar of the program to ensure correct order of operations and handling of factorials, I wrote a function called factorial_first(). Since the factorial operator appears after a number, this function checks if a factorial operator is present before attempting to use any other operators. This means that all references to the primary() function in the term() function were changed to call factorial_first() instead. This ensures that factorial will always be

checked for each number. The factorial operation was implemented with a simple for loop that looped from 1 to x, multiplying each increment to the total.



Problem 3:

For this problem, the 5 random number distributions I chose were Uniform Distribution, Binomial Distribution, Exponential Distribution, Normal Distribution, and Geometric Distribution. For each distribution, I created the distribution using parameters of my choice. Then, I ran a loop 1000 times for each, where I generated a random number using a default_random_engine variable and proceeded to save the value into an array of size 1000 for each. I also implemented two functions, one to calculate the mean and another to calculate the standard deviation. After calling those functions for each, I was ready to output to the text file. Using ofstream, I created a text file called 'randomResults.txt' and outputted a table with the results for each distribution, which can be seen below.

For the example shown, the following distributions were used:

Uniform(0, 10), Binomial(10, 0.5), Exponential(5), Normal(5, 2), Geometric(0.4)

Distribution	Mean	Standard Deviation
Uniform	4.91993	2.85087
Binomial	4.97	1.59659
Exponential	0.199295	0.203634
Normal	4.88124	2.07179
Geometric	1.524	1.9752

Problem 4:

The Money class I implemented begins with creating a long variable for cents and a string variable for currency. There is then a default constructor and main constructor following. The rest of the Money class is full of operator overloading functions, as specified in the instructions. For the addition and subtraction operators, I check to ensure that the currencies of the two Money object match each other. If they match, the calculation is done and returned, but if they don't match, an error is printed, and -1 cents and currency 'None' are returned. For multiplication overloading, I ensure that the number we are multiplying by is greater than 0, and if it isn't, an error is printed and -1 cents and 'None' currency are returned. For integer multiplication, I simply returned the multiplication. For the double multiplication, I

returned the rounded result. For division overloading, I ensure that the number we are dividing by is greater than 0, and if it isn't, an error is printed and -1 cents and 'None' currency are returned. For both integer and double division, I simply return the rounded result.

Outside of the Money class, there is the input and output operator definitions. For input, I save the currency entered and the dollars entered. From there, I convert the dollar amount into cents by multiplying by 100 and return the input. For the output operator, I calculate the amount of whole dollars by dividing cents by 100. From there, I calculate the remaining cents by subtracting cents by the whole dollar amount in cents. Finally, I structure the output to show the currency and the amount in dollars, formatted to print 2 decimals.

Below are some tests done on the Money class. The integer 2 and the double 1.5 are hard-coded for each test case for simplicity, but the class works for any integer double pairs.

```
Entry 1: Enter the currency and dollar amount: USD 100
Entry 2: Enter the currency and dollar amount: USD 50.50

Entry1: USD 100.00
Entry2: USD 50.50

Entry1 + Entry2 = USD 150.50
Entry1 - Entry2 = USD 49.50
Entry1 *  2 = USD 200.00
Entry1 * 1.5 = USD 75.75
Entry1 / 2 = USD 50.00
Entry1 / 1.5 = USD 33.67
```

```
Entry 1: Enter the currency and dollar amount: EUR 22.34
Entry 2: Enter the currency and dollar amount: EUR 47.21

Entry1: EUR 22.34
Entry2: EUR 47.21

Entry1 + Entry2 = EUR 69.55
Entry1 - Entry2 = EUR -24.87
Entry1 *  2 = EUR 44.68
Entry1 * 1.5 = EUR 70.82
Entry1 / 2 = EUR 11.17
Entry1 / 1.5 = EUR 31.47
```

Below are some test cases in which error checking detects an error.

```
Entry 1: Enter the currency and dollar amount: USD 100
Entry 2: Enter the currency and dollar amount: EUR 50.50

Entry1: USD 100.00
Entry2: EUR 50.50

Cannot add different currencies together.
Entry1 + Entry2 = None -0.01
Cannot subtract different currencies from each other.
Entry1 - Entry2 = None -0.01
Entry1 *  2 = USD 200.00
Entry1 * 1.5 = EUR 75.75
Entry1 / 2 = USD 50.00
Entry1 / 1.5 = EUR 33.67
```

```
Entry 1: Enter the currency and dollar amount: USD 100
Entry 2: Enter the currency and dollar amount: USD 50.50

Entry1: USD 100.00
Entry2: USD 50.50

Entry1 + Entry2 = USD 150.50
Entry1 - Entry2 = USD 49.50
Cannot multiply by 0 or a negative number.
Entry1 *  -2 = None -0.01
Cannot multiply by 0 or a negative number.
Entry1 * -1.5 = None -0.01
Cannot divide by 0 or a negative number.
Entry1 / -2 = None -0.01
Cannot divide by 0 or a negative number.
Entry1 / -1.5 = None -0.01
```

Problem 5:

In the EuropeanOption class, I added a default constructor, as well as the main constructor. I also created a helper function to return the CDF of the standard normal distribution called N(). The most important function of this class was getPrice(), which returns the price of an option using Black &

Scholes. Utilizing the inputs from the user, the getPrice() function calculates d1, which uses the N() function, and d2. From there, I used an if statement to determine whether or not the option is a put or call. The corresponding option price is returned based off the decision of the if statement.

In the main function, I made the program prompt the user for all 6 parameters and then used while loops to error check those inputs. For option type, I checked to ensure the user entered either put or call, and returned an error otherwise. For spot price, I ensured the price was greater than or equal to zero. For the strike price, I ensured the price was greater than or equal to zero. For interest rate, I prompted the user to enter the rate in %, I then ensured that the rate was greater than or equal to 0% and less than or equal to 100%. For volatility, I prompted the user to enter the vol in %, and then I ensured the vol was greater than or equal to 0%. Finally, for time to maturity, I checked to ensure that it was greater than 0 years. After that, it was a simple call to create the corresponding option, call the getPrice() function and print the results.

For the examples below, I checked that the solutions were correct using an online option price calculator.

```
Enter the type of option (put or call): put
Enter the spot price: 100
Enter the strike price: 150
Enter the interest rate (in %): 20
Enter the volatility (in %): 60
Enter the time to maturity (in years): 1

The price of the option is: 39.1504
```

```
Enter the type of option (put or call): call
Enter the spot price: 100
Enter the strike price: 150
Enter the interest rate (in %): 20
Enter the volatility (in %): 60
Enter the time to maturity (in years): 1

The price of the option is: 16.3408
```

Below are examples of all of the error checking.

```
Enter the type of option (put or call): test
Error: Option type can only be 'put' or 'call'
```

```
Enter the spot price: -10
Error: Spot price can't be negative.
```

```
Enter the strike price: -5
Error: Strike price can't be negative.
```

```
Enter the interest rate (in %): -5
Error: Interest rate can't be negative.
Enter the interest rate (in %): 120
Error: Interest rate can't be greater than 100%
```

```
Enter the volatility (in %): -20
Error: Volatility can't be negative.
```

```
Enter the time to maturity (in years): -5
Error: Time to maturity can't be negative.
```

Problem 6:

For this problem, the EuropeanOption class remained identical to problem 5, the changes all occurred in the main() function. First, I used ifstream to read in the TSV file called ‘optionsData.txt’ with the options data. From there, I looped through data row by row, saving the data to a vector of vector strings. Once the data was imported, I looped through the data vector and changed spot price, strike price, interest rate, volatility and time to maturity variable for each option to type double. Now, all the variables are in the correct form to call the EuropeanOption class constructor. I created a vector of EuropeanOptions and looped through the inputted data row by row, creating the EuropeanOption object and appending it

to this vector. In the same loop, I also created a vector of doubles and calculated the option price for each row of data.

One of the trickiest parts was after all of the calculations were completed was to create a vector of vector strings called 'output_vec' which would store all of the data from the original dataset, but also appending a new column called price that would have the corresponding option price for each row. Utilizing nested for loops and stringstream, I looped through the original data array, essentially copying over the original data. However, output_vec includes one more column and for that column, I used various if statements to help format the option prices to strings and appended to the final vector. Finally, I used ofstream to output a file called 'results.txt', where I looped through the output_vec vector of vector strings and formatted it to be tab delimited, like the original file.

The Input File: (optionsData.txt)

type	spt	str	rate	vol	ttm
call	25	50	25	90	1
put	25	50	25	90	1
call	50	40	15	35	2
put	50	40	15	35	2
call	95	120	20	60	3
put	95	120	20	60	3
call	72	62	12	40	2
put	72	62	12	40	2
call	150	166	22	28	1
put	150	166	22	28	1

The Output File: (results.txt)

type	spt	str	rate	vol	ttm	Price
call	25	50	25	90	1	5.34086
put	25	50	25	90	1	19.2809
call	50	40	15	35	2	21.7621
put	50	40	15	35	2	1.39482
call	95	120	20	60	3	48.2422
put	95	120	20	60	3	19.0996
call	72	62	12	40	2	28.0034
put	72	62	12	40	2	4.77431
call	150	166	22	28	1	25.5413
put	150	166	22	28	1	8.75941