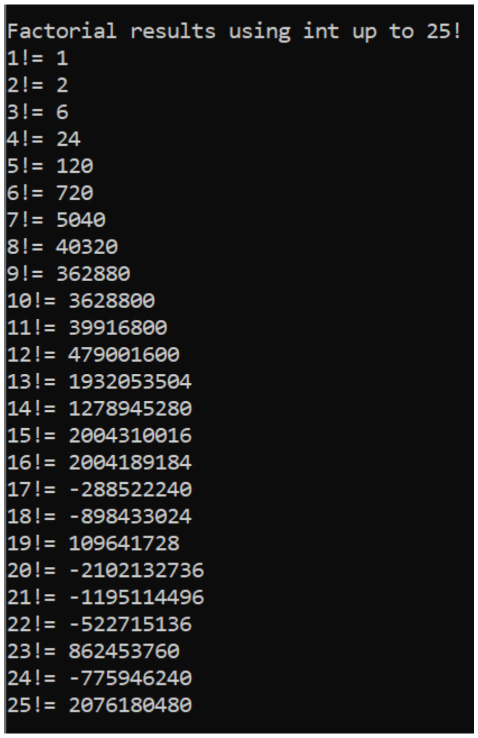
Edward Kwak
01/28/23
ECE 2036

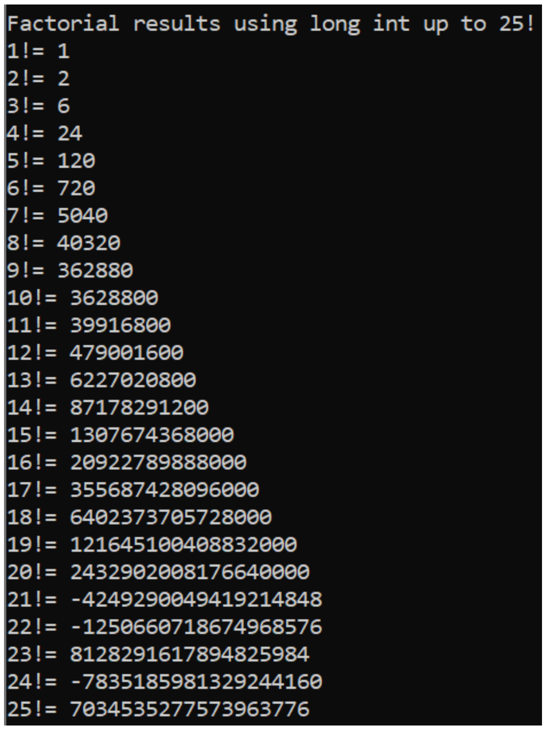
Problem 1: OUT OF RANGE ERROR
The largest factorial that can be represented with regular int type is: 12!

- after 12! it can be seen in the images below that the factorial values while using the int type result in the wrong results.

The largest factorial that can be represented with long int type is 19!

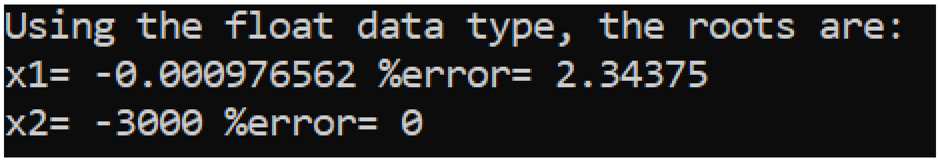


```
Factorial results using int up to 25!
1!= 1
2!= 2
3!= 6
4!= 24
5!= 120
6!= 720
7!= 5040
8!= 40320
9!= 362880
10!= 3628800
11!= 39916800
12!= 479001600
13!= 1932053504
14!= 1278945280
15!= 2004310016
16!= 2004189184
17!= -288522240
18!= -898433024
19!= 109641728
20!= -2102132736
21!= -1195114496
22!= -522715136
23!= 862453760
24!= -775946240
25!= 2076180480
```



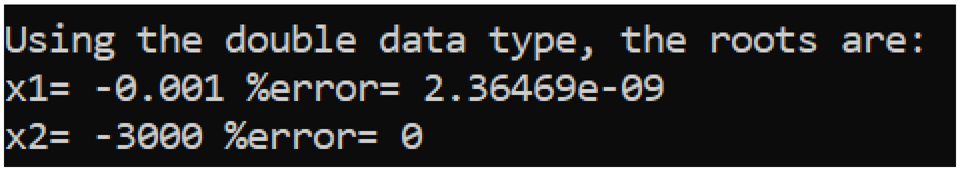
```
Factorial results using long int up to 25!
1!= 1
2!= 2
3!= 6
4!= 24
5!= 120
6!= 720
7!= 5040
8!= 40320
9!= 362880
10!= 3628800
11!= 39916800
12!= 479001600
13!= 6227020800
14!= 87178291200
15!= 1307674368000
16!= 20922789888000
17!= 355687428096000
18!= 6402373705728000
19!= 121645100408832000
20!= 2432902008176640000
21!= -4249290049419214848
22!= -1250660718674968576
23!= 8128291617894825984
24!= -7835185981329244160
25!= 7034535277573963776
```

Problem 2: ROUND-OFF ERROR
Using a float



```
Using the float data type, the roots are:
x1= -0.000976562 %error= 2.34375
x2= -3000 %error= 0
```

Using a double



```
Using the double data type, the roots are:
x1= -0.001 %error= 2.36469e-09
x2= -3000 %error= 0
```

- 2.36469e^-09 is practically = 0

Given the fact that each root can easily be represented in a float. Why do you think that there was error using the floating point? Be as specific as you can. You might write your answer on the back of this sheet.

- The roots themselves (-0.001 and -3000) can easily be represented in a float. However, there were intermediary calculations completed prior to storing the values (x1 and x2) into a float. It must be that during these calculations, some of the values exceeded what is allowed to be carried in a float thus causing the error. My prediction is that a double, being approximately twice more accurate compared to a float, was able to handle the intermediary calculations.

Problem 3: TRUNCATION ERROR

How many terms do you need to include in the power series expansion until the digital value remains unchanged?

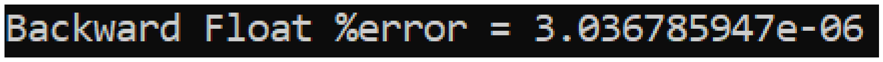
Answer for float type : 11 Terms (percent error stopped changing after 11 terms)

Answer for double type: 18 Terms (percent error stopped changing after 18 terms)

Problem 4: ERROR PUZZLE

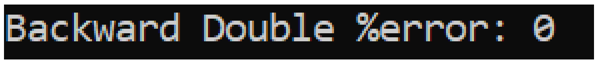
What is the error for the summing for the first 100 terms in the power series from largest to smallest?

Answer for float type:



```
Backward Float %error = 3.036785947e-06
```

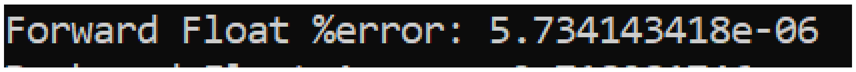
Answer for double type:



```
Backward Double %error: 0
```

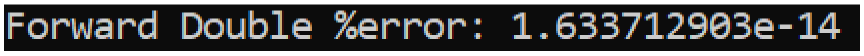
What is the error for summing the first 100 terms in the power series from smallest to largest?

Answer for float type:



```
Forward Float %error: 5.734143418e-06
```

Answer for double type:



```
Forward Double %error: 1.633712903e-14
```

Why do you think it makes a difference as to whether you sum forward or backwards?

- It seems that the difference in this case is that the forward sums ended up with higher percent errors for both float and double categories. My best guess is that when we sum backwards, we are adding the largest numbers first and the truncation of smaller values is less significant. The opposite would be true for forward sums. I am not sure if I recall correctly, but in calculus I believe a backwards sum for riemann sum is more accurate in certain cases as well. This could be a similar scenario.