## Bugs

1. The size of data structure (int) is too small for the given problem
2. The condition for recursion was wrong
3. When the struct is passed in factorial the changes are not saved
4. The function reduce factorial turns every value
5. Performance issue: The program calculated the factorial every time the passed number evenb though it already had it.

## Solution

The size of each of the numbers eas set to long double so it fits all the numbers processed,

The function factorial was removed and integrated inside computefactorial creating a condition where if i is different than 0 it will take i and multiply for the last value in the array, if its 0 it will assign 1. Additionally, the input of the function was changed to a struct pointer so the changes modify in the real struct.

ReduceFactorial was removed since it didnt represent any value for program.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define MAX_FACTORIALS 10000
#define NUM_FACTS 100
struct FactorialResults
{
	long double results[MAX_FACTORIALS];
	int numResults;
};

void computeFactorials(struct FactorialResults *results, int numFactorials)
{
	int i;
	for (i = 0; i < numFactorials; i++)
	{
		results->results[i] = (i) ? i * results->results[i - 1] : 1;
	}
	results->numResults = numFactorials;
}
int main(void)
{
	struct FactorialResults results = { {0}, 0 };
	int i;
	computeFactorials(&results, NUM_FACTS);
	for (i = 0; i < NUM_FACTS; i++)
```

```
        {
                printf("%5d %12Lf\n", i, results.results[i]);
        }
        return 0;
}
```

## Reflextion

- ➔ For me the three techniques that represented the most value were brute forcing (Debugger & prints), program slacking and deductive cause of elimination. Starting with brute force is one of the most commune way of testing reeling of line by line code examination and looking at the change of variable as the program runs, this allows to know when an unexpected output is presented, another reason is they facilitate to implement and repeat sign it doesn't require a deep analysis of error leaks. Finally, together comes program slashing and deductive error finding, these techniques allow the tester to determine where the error may be just by reading the code, specifying where the error could be by looking at where the data is changed and filtering the test in those specific areas.

- ➔ What are the largest integer and double values you can store?
  - ◆ For regular whole numbers (integers), the biggest one depends on your computer's "word size." On many computers, it's around 2 billion or 9 quintillion, depending on whether you're using a 32-bit or 64-bit system. For really big decimal numbers (like those with lots of digits after the decimal point), you can use double-precision floating-point values. The biggest double number you can store is around 1.8 with 308 zeros after it.

- ➔ Why is there a limit on the maximum value you can store in a variable?
  - ◆ Computers use a fixed number of bits to store values. These limits help manage memory and make sure programs run smoothly. If there were no limits, programs could quickly eat up all available memory, causing crashes and slowdowns.

- ➔ If you exceed the maximum value an integer can hold, what happens? Explain why the format causes this to happen.
  - ◆ If you go beyond the limit for integers, you'll experience what's called an overflow. Think of it like an odometer in a car. When it rolls over from 999,999 back to 0, that's an overflow. In C, when you overflow an integer, the result isn't guaranteed, and it depends on your computer and compiler. It might wrap around to negative numbers or do something else unpredictable.
- ➔ What is the format for the storage of double-precision floating-point values?
  - ◆ Double-precision floating-point values use a standard format called IEEE 754. It's like scientific notation for computers. It has three parts: a sign (positive or

negative), an exponent (how many times to multiply the number by 2), and a fraction (the actual number with its decimal part).

➔ The default stack size given when a C++/C program is 1MB and the heap memory is usually 1MB but can change since its not assigned by visual but by the computer (windows) itself.
Because of the big account of memory use for factorial a warning is sent saying that we should consider move the data to the pile, but it didn't stop the program from running. Even though changing the size of stack and would solve the issue of the program there are better ways to solve the program. Changing the sizes would represent a giant memory loses affecting the performance of the program and the machine itself, to avoid we can make better logical decisions regarding the way we use the memory like storing each digit in an array instead of the whole number and not saving every result of the factorial since once they are printed they are no longer necessary.