Joseph A. Boyle

This program will, given a positive integer **n**, compute the long form of:

$$(1+x)^n$$

This calculation is done by computing: nCr = **n!/(r!(n-r)!)** for all values of r between 1 and n (inclusive), producing the r'th term of the equation: $nCr*x^r$.

This program was written in 32 bit assembly. Thus, the maximum value it can handle is n = 13, as 13! is approximately 6.2M, which is larger than 2^{32}. Thus, we cannot compute it with simple 32-bit arithmetic.

How it Works

The program converts the user input into an integer using atoi, and then iterates from 1 to this value (inclusively). It performs a Factorial calculation on the user input to check if the maximum value which will be calculated is too large and therefore overflows. If it does, the program stops execution.

nCr Calculation

With n and r, we can compute nCr as follows:

Input: Two integers, **n** and **r**

1. Compute **n!** And store it in memory
2. Compute **r!** And store it in memory
3. Compute **(r-n)!** And store it in memory
4. Retrieve the values of **(r-n)!** And multiply it by the value of **r!**
5. Divide **n!** By this value.

Since we are working with the **divl** x86 instruction, we are doing the division of a 64-bit integer by a 32-bit integer. As such, we must be careful to set the values of the **edx** register to 0 so that we do not calculate an incorrect result. We store the result of step 4 in the **eax** register, and the result from step 1 in the **ebx** register. We can then simply call **divl ebx** to perform edx:eax / ebx.

Factorial

Factorial computes the factorial of a given number, **n**, by continuously multiplying a variable, **val** which is initially set to one, by a number **r**, initially set to **n**, a total of **n** times. Each time a multiplication occurs, **r** is set to **r-1**. Essentially, this equates to: n*(n-1)*(n-2)*...(1), which is equivalent to **n!**.

Efficiency

The algorithm runs in $O(N^2)$ time.

Factorial: Factorial performs **n** multiplications for a given argument **n**, thus O(N) time efficiency.

nCr: Since we are simply performing 3 factorial calls, one multiplication call, and one division call, the efficiency of this is limited by factorial, which is O(N).

Overall: Since we must call nCr n times, the overall running time is $O(N^2)$

In terms of space efficiency, we are using 5 local variables on the stack with every value of **r** between 1 and **n** (inclusive). These variables are consumed once the function call goes away, thus the space usage is constant: O(5) ~= O(1).