Adeeb Kabir
CS211 Computer Architecture
Instructor: Professor Russell
30 July 2017

1 Writing formula.c and nCr.s

The troubles of this project came down to mostly learning how to read assembly in the first place. However, once I learned to recognize the tells of reading and writing out of memory, understanding the arithmetic operations between moving the data became really easy. In addition to doing all of the required pieces, I implemented detection of overflow, a calculation for the time it takes for the program to calculate the expression, and support for negative input.

My solution to factorial is iterative since I did not want to store anything on the stack for that function, and nCr just consists of repeatedly taking factorials and then multiplying or dividing the results. Checking for overflow and the time it took to compute was just a matter of adding a few lines of conditional jumps to my existing code. I also handled negative numbers fairly well, but I think that was an extraneous worry all in all. Luckily, the computation of $n!$ does not run in factorial time. Given that my loops run based on the size of n, the program runs in linear time $\mathcal{O}(n)$. I wrote all my test cases in `testformula.sh`.

2 Reading mystery.s

Mystery ended up taking more time to think about since I had to take my newfound knowledge and apply it to *someone else's* generated code. I kept pen and paper out while commenting every line on mystery.s to understand what it did. After running it a few times and playing around with inputs, I found that it was a Fibonacci Sequence calculator.

If the input is n, then the calculator output the nth term of the sequence. It makes the computation faster by breaking down the number into the seed values of 0 and 1 on the zeroeth and first terms, adding these together to get second term, and repeating the process until the program reaches the nth term. This became obvious after just once manual stack trace on paper, and the assembly code generated by the `gcc` compiler is very close to the provided assembly file. The program iterates through the loops in `dothething` n times twice while executing, so it also runs in linear time $\mathcal{O}(n)$.

The given assembly code is in `mystery.given.s`.
My unoptimized generated assembly code is in `mystery.unoptimized.s`.
My optimized generated assembly code is in `mystery.s`.
My test cases are detailed in `testmystery.sh`.