Read Me File for PA3:

To design my mystery.c file I first looked at the assembly code to determine what was happening within the code. First I looked at the .globl functions because they indicated the separated functions. I saw three functions including: add, dosomething, and main. I then broke down the functions down further by looking at the code within each function. In add I noticed that there were two movl indicating two variables (constants) with %eax indicating their addresses. I then realized that addl was being called indicating that an item was being added to another address. Meaning line 4 was being added to line 3. The result was then being returned.

```
add:
        pushl %ebp
        movl  %esp, %ebp
        movl  12(%ebp), %eax
        addl   8(%ebp), %eax
        popl   %ebp
        ret
        .size   add, .-add
```

dothething was a very long piece of code. I was unfamiliar with what was going on at first but I saw that there the same function was being called within the same function. Meaning dosomething was calling itself which gave me the immediate belief that it was a recursive function. I then saw that the multiple divisions were a representation if statements. I saw that a variable called num was being referred to several times in different conditions. When I compared my unoptimized and optimized code, I noticed a major difference. To begin with I saw that lines of codes with movl, movq, and several other statements were significantly condensed. I think the compiler made those changes to make the code more efficient. Instead of doing the same computations multiple times, I believe that the compiler stored the variable in memory to refer to it at a later time. When running the optimized versus the unoptimized code in C for mystery.s, I saw many differences. The optimized code was pushing and popping data from a stack, while my unoptimized was doing computations with leal. I believe that the compiler used a stack frame to do computations in order to save the contents of the register to memory and by popping the compiler could restore the contents of the register. The optimized code seemed to implement an array called num was called to store the registers. My unoptimized code simply did continuous computations by calling the Fibonacci function. Also the optimized program used the test register which indicated that the result was not being stored in any destination operand. It was simply computing the operands, while my unoptimized code was calling add, moves, etc.