Lab # 2

Collaboration is encouraged. You may discuss the problems with other students, but you must **write up your own solutions**, *including all your C programs, by yourself. If you submit identical or nearly identical solutions to someone else, this will be considered a violation of the code on academic honesty.*

In Lab 1, we saw examples of *iterative* programs to compute functions like raising-to-a-power and factorial. An alternative approach for computing many types of functions is *recursion*. A recursive function is one that calls itself.

Consider the following implementation of the factorial function.

```
/* factorial function */
int factorial(int n)
{
   if (n <= 1) {
      return 1;
   } else {
     return n * factorial(n - 1);
   }
}
```

Note that there must be a base case to the recursion – otherwise, the function will continue calling itself foreover.

1. **Fibonacci Numbers**

   Consider the numbers:

   `0, 1, 1, 2, 3, 5, 8, 13, 21, 34`

   What comes next?

Add the missing line so that the following code computes the Fibonacci sequence.

```
# include <stdio.h>

int fibonacci(int n) {
   if (n <= 1) {
       return n;
   } else {
      // ADD THIS LINE
      printf("%d\n", m);
      return m;
   }
}

int main(int argc, char **argv) {
   fibonacci(atoi(argv[1]));
}
```

2. **Ackermann Function**

   The Ackermann Function is a mind-boggling recursive function. It is called with two integer arguments:

```
# include <stdio.h>

int ackermann(int m, int n) {
   printf("%d, %d\n", m, n);
    if (m == 0)
        return n + 1;
    else if (n == 0)
        return ackermann(m - 1, 1);
    else
        return ackermann(m - 1, ackermann(m, n - 1));
}

int main() {
   printf("%d\n", ackermann(4, 2));
}
```

   This function always terminates with an integer value, yet it can take remarkably long to do so, even when called with small arguments. To see how the Ackermann function grows so quickly, it helps to expand out some simple expressions using the rules in the original definition. For example, we can fully evaluate A(1,2) in the following way:

$$\begin{aligned}
A(1,2) &= A(0,A(1,1)) \\
&= A(0,A(0,A(1,0))) \\
&= A(0,A(0,A(0,1))) \\
&= A(0,A(0,2)) \\
&= A(0,3) \\
&= 4.
\end{aligned}$$

To demonstrate how A(4,3)'s computation results in many steps and in a large number:

$$\begin{aligned}
A(4,3) &= A(3,A(4,2)) \\
&= A(3,A(3,A(4,1))) \\
&= A(3,A(3,A(3,A(4,0)))) \\
&= A(3,A(3,A(3,A(3,1)))) \\
&= A(3,A(3,A(3,A(2,A(3,0))))) \\
&= A(3,A(3,A(3,A(2,A(2,1))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(2,0)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(1,1)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(1,0))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(0,1))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,2)))))) \\
&= A(3,A(3,A(3,A(2,A(1,3))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(1,2)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(1,1))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(1,0)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(0,1)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,2)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,3)))))) \\
&= A(3,A(3,A(3,A(2,A(0,4))))) \\
&= A(3,A(3,A(3,A(2,5)))) \\
&= ... \\
&= A(3,A(3,A(3,13))) \\
&= ... \\
&= A(3,A(3,65533)) \\
&= ... \\
&= A(3,2^{65536}-3) \\
&= ... \\
&= 2^{2^{65536}}-3.
\end{aligned}$$

Written as a power of 10, this is roughly equivalent to $10^{6.03110^{19,727}}$.

Problem Find arguments to the ackermann function that result in more than 500 recursive calls but less than 1000.

3. The runtime of iterative functions can also be difficult to analyze. The *Collatz* conjecture is a famous open problem in mathematics, proposed by Lothar Collatz in 1937. For any positive integer x,

   - if $x = 1$ stop;
   - else if x is odd, let $x = 3x + 1$;
   - else let $x = x/2$.

   For instance, starting with $x = 5$, one follows the sequence 16, 8, 4, 2 and 1. Starting from $x = 27$, one follows the sequence 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, ... Does this sequence ever end? Yes. But does every such sequence end?

   Most mathematicians who study this problem believe so. The conjecture is that, starting with any positive integer x, the procedure always terminates with $x = 1$. Proving this is evidently difficult. Paul Erdös said about the conjecture: "Mathematics is not yet ready for such problems". Like Fermat's Last Theorem, it is striking that a problem that is so easy to state could be so hard to prove.

   You are *not* asked to prove the *Collatz* conjecture on this homework. (But hey, if you do, you'll immediately get a PhD and a Field's Medal – the mathematical equivalent of a Nobel Prize). Rather you are asked to study the run-time of a program that implements the procedure.

```c
# include <stdio.h>

int main(int argc, char **argv)
{
    int n = atoi(argv[1]);
    while (n > 1) {
       printf("%d\n", n);
       if (n % 2 == 0) {
          n /= 2;
       } else {
          n = 3*n + 1;
       }
    }
}
```

   Problem

   (a) What sequence does the program print out given inputs of 7, 15, 27, and 121?

   (b) Modify the program so that it prints out the length of the sequence instead of the sequence itself. For instance,

- for an input of 7, it should print out 16;
- for an input of 15, it should print out 17;
- for an input of 27, it should print out 111;
- for an input of 121, it should print out 95.