CSCI 2500 — Computer Organization
Lab 02 (document version 1.0)

- This lab is due by the end of your lab session on Wednesday, September 12, 2018.
- This lab is to be completed **individually**. Do not share your code with anyone else.
- You **must** show your code and your solutions to a TA or mentor to receive credit for each checkpoint.
- Labs are available by 6:00PM on Mondays before your lab sessions. Plan to start each lab early and ask questions during office hours, in the discussion forum on Submitty, and during your lab session.

1. **Checkpoint 1:** The concept of **endianness** plays an important role on your laptop (or any computer hardware architecture). Specifically, endianness dictates whether the most significant byte (i.e., "big-end") or least significant byte (i.e., "little-end") is stored as the first byte of the value being stored at a given address. Some architectures are big-endian (e.g., Motorola 68K), while other architectures are little-endian (e.g., x86 and its descendants).

   Start by typing in the code below. Attempt to understand what's happening in the given code as you type it in. Key to understanding this code is that an `int` data type in C is stored as four bytes.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  /* Insert your four bytes of ASCII for your secret message */
  int z = 0xFFFFFFFF;
    /* The 0x prefix above indicates a hexadecimal number */

  char * c = (char *)&z;

  printf( "%c", *c++ );
  printf( "%c", *c++ );
  printf( "%c", *c++ );
  printf( "%c\n", *c++ );

  return EXIT_SUCCESS;
}
```

   Compile and run this code, then replace the hexadecimal value of variable `z` with your own secret four-letter word. To do so, look up each letter in an ASCII table and convert them to their corresponding hexadecimal values. You can find an ASCII table at the following URL: `https://en.wikipedia.org/wiki/ASCII#ASCII_control_code_chart`

   Based on your output, what is the endianness of your laptop?

2. **Checkpoint 2:** The greatest common divisor (GCD) of two numbers can be determined using Euclid's algorithm. Implement the recursive algorithm to determine the GCD of various values.

3. **Checkpoint 3:** As we saw in Lab 01, the Fibonacci sequence is calculated recursively by summing the previous two values of the sequence, i.e., `fib(n)=fib(n-1)+fib(n-2)`. As with Lab 01, assume that this sequence starts with `0` and `1` as its first two elements.

   For this checkpoint, write code to calculate the nth Fibonacci number for up to 10 inputs given by the user on the command-line (hint: use `argc` to determine the number of inputs and `atoi()` to convert each input into an integer). Use a naive approach in which you simply calculate each Fibonacci number in turn using either a recursive or iterative approach. An example of program execution is as follows:

   ```
   bash$ ./a.out 4 11 8
   fib(4) is 3
   fib(11) is 89
   fib(8) is 21
   ```

   Repeatedly calculating the nth Fibonacci number with varying values of n requires a lot of repetitive computation. A better approach is to *store* results of previous computations for future use. This technique is called *memoization*.

   Next, write an optimized version that stores previously computed Fibonacci numbers in a global array of values that you dynamically allocate at program startup. Pre-fill this array with sentinel values of zero, which will let you check whether the requested value has already been computed or not. As an example, if `fib(n)` is non-zero, then you know that that value has already been computed and you can return that as the solution. Otherwise, you must compute `fib(n)`, though you can possibly utilize some pre-computed values.

   Once you've tested your solutions, prepend the `time` command to your normal command line for both your naive and optimized versions, as in:

   ```
   bash$ time ./a.out 4 11 8 13 9 5
   ```

   Was there a noticeable difference? What was the tradeoff here?

   And is there a further optimized version you could write?