# Lab # 3

*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.*

1. **Bit-Wise Operators in C**

   C provides six operators for bit manipulation; these may only be applied to integers. that is `chars` and `ints`.

   - `&` bitwise AND
   - `|` bitwise inclusive OR
   - `^` bitwise exclusive OR
   - `<<` left shift
   - `>>` right shift
   - `~` one's complement (unary)

   The bitwise AND operator & is often used to mask off some set of bits, for example

   ```
   unsigned int n;
   n = n & 0177;
   ```

   sets to zero all but the low-order 7 bits of `n`. The bitwise OR operator `|` is used to turn bits on:

   ```
   unsigned int x;
   x = x | 0xF;
   ```

   sets the low-order 4 bits of `x` to one.

   The operator `^` sets a one in each bit position where its operands have different bits, and zero where they are the same. (One must distinguish the bitwise operators `&` and `|` from the logical operators `&&` and `||`, which imply left-to-right evaluation of a truth value. We'll see these later. For example, if `x` is 1 and `y` is 2, then `x & y` is zero while `x && y` is one.)

   The shift operators `<<` and `>>` perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be non-negative.

Thus `x << 2` shifts the value of `x by` two positions, filling vacated bits with zero; this is equivalent to multiplication by 4. Right shifting an unsigned quantity fits the vacated bits with zero. Right shifting a signed quantity isn't an error, but the result isn't clear. Sometimes C fills in the bits with 0's, sometimes with 1 (depending on the compiler and/or the machine).

The unary operator `~` yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example

```
x = x & ~077
```

sets the last six bits of `x` to zero. Note that `x & ~077` is independent of word length, and is thus preferable to, for example, `x & 0177700`, which assumes that `x` is a 16-bit quantity.

As an illustration of some of the bit operators, consider the function `getbit(x,p)` that returns 0 or 1 depending on whether the `p`-th bit of `x` (from the right) is 0 or 1.

```
# include <stdio.h>

/* getbits: get bit from position p */
unsigned int getbit(unsigned int x, unsigned int p)
{
   return (x >> p) & 1;
}

int main(int argc, char **argv) {

   unsigned int x = atoi(argv[1]);
   unsigned int p = atoi(argv[2]);

   /* print out given bits */
   printf("x: ");
   int k;
   for (k = 8*sizeof(int) - 1; k >= 0; k--) {
      if ((x >> k) & 1)
         printf("1");
      else
         printf("0");
   }
   printf("\n");

   /* print out specified bit */
   printf("%d-th bit: %d\n", p, getbit(x,p));
```

```
}
```

This program prints out:

```
getbit 1023 8
x: 00000000000000000000001111111111
8-th bit: 1

./getbit 1023 9
x: 00000000000000000000001111111111
9-th bit: 1

/getbit 1023 10
x: 00000000000000000000001111111111
10-th bit: 0
```

**Problem**: Write a program `setbit.c` that accepts two integer arguments `s` and `p`. If `s` is equal to 1, it sets all the bits of an integer to 1 except for the `p`-th bit. If `s` is equal to 0, it sets all the bits of an integer to 0 except for the `p`-th bit. Here is the sample output.

```
./setbit 1 0
11111111111111111111111111111110

./setbit 0 0
00000000000000000000000000000001

./setbit 0 10
00000000000000000000010000000000

./setbit 1 10
11111111111111111111101111111111
```

2. **Run Time Complexity**

   In this problem, we'll study the **run-time complexity of programs**. The run-time complexity is a measure of how long it takes a program to execute. It is independent of the machine that the program is actually running on. (Wait until next year and the new PC that you'll get will likely run twice as fast as the old clunker that you have this year.) It doesn't measure the actual time. Rather, run-time complexity refers to the number of steps in an algorithm or program.

   To characterize this, we could track and count every statement (so every line in a C program that's terminated by a semicolon). We would have to include conditional statements (so count every time the conditional part of an `if`, `else` or `while` statement is executed). But this isn't always a well-defined task. What about library functions

such as `atoi()`? How many steps do they entail? What about the `return` statement in a function? Does it count as a step or not?

Instead of such pedantic statement counting, run-time complexity generally entails a highly approximate analysis. In this lab, we are interested in the number of times that `while` loops execute. Whether there are 10 statements or 15 statements in each iteration of the loop doesn't matter. In later labs, when we study recursive function calls, we'll be interested in how many times the functions are called (times the number of iterations in each call, if the functions have `for` or `while` loops).

**Collatz Procedure**

The *Collatz* conjecture is a famous open problem in mathematics, proposed by Lothar Collatz in 1937. Consider the following iterative procedure. 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.

```
# 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.

(c) Write a program that, given a range of inputs, finds the starting value that produces the longest sequence. More specifically, write a program that, given an input of $n$, finds the starting value between 1 and $n$ that produces the longest sequence. Have it print out this value and the length of the sequence. For instance,

- for an input of 15, it should print out 9 19;
- for an input of 16, it should print out 9 19;
- for an input of 17, it should print out 9 19;
- for an input of 18, it should print out 18 20.

So, for starting values up to 17, it is the starting value 9 that produces the longest sequence. This sequence has length 19. Bumping this up to 18, it is 18 that produces the longest sequence. This has length 20.

(d) How high can you go with $n$ in Part 3? How long does it take your computer to finish executing the program (in seconds or minutes) for large values? On my computer, for an input $n$ of 10,000,000, it takes about 5 seconds. My program determines that for numbers up to 10,000,000, a starting value of 7,532,665 produces the longest sequence. This sequence has length 615.