CSCI 1730 Breakout Lab 02

Hamming Distance

Learning Outcomes

- Understand how unsigned integers are represented and stored in memory by converting unsigned integers from base-10 to binary.
- Use emacs to write programs in a Unix environment.
- Compile, link, and run programs via the command line.

Problem / Exercise

The Hamming distance between two equal length bit strings is the number of positions where the bits do not match, and it is an important metric used in information theory and various scientific disciplines. We can calculate the Hamming distance between two non-negative integers by converting them to their unsigned binary (bit string) representation and counting the number of bit positions that do not match. If the two bit strings are not of the same length, then we can add leading zeros to the shorter bit string to make it the same length as the longer bit string. Adding leading zeros to an unsigned bit string does not change their value (e.g. 101 = 0101 = 00101 = 000101). Think about how you would solve this problem with a C program, and write a general algorithm on paper and pencil before you write any source code. You may assume the user of your program will input, via command line arguments, two non-negative integers as shown in the Examples section that can be converted to long values using calls to the atol function. Consider the following questions and read over the Examples section to get an idea on how your program should work.

- How do you get command line arguments in C?
- How do you convert command line arguments to a long integer?
- How do you convert a non-negative integer to its bit string (binary)?
- How do you add leading zeros if the bit strings are not equal?
- How do you compute the Hamming distance between two equal length bit strings?

Examples

Your C source code should be in a file called `lab02.c`, and it should be compiled into an executable called `lab02.out`. Your C program must look exactly like the examples below when run on the command line on odin, it must compile correctly, and it must run correctly with any valid sequence of inputs provided as command line arguments. You may assume that the command line arguments for the strings that represent integers will always be non-negative and they can be successfully converted to longs using calls to the atol function. However, you may not assume that these inputs will be in order. Each example is a separate execution of a correct program for this assignment.

```
./lab02.out 5 8
0101 is the bit string for 5
1000 is the bit string for 8
3 is the Hamming distance between the bit strings
```

```
./lab02.out 1730 30
11011000010 is the bit string for 1730
00000011110 is the bit string for 30
7 is the Hamming distance between the bit strings
```

```
./lab02.out 15 0
1111 is the bit string for 15
0000 is the bit string for 0
4 is the Hamming distance between the bit strings

./lab02.out 28 66
0011100 is the bit string for 28
1000010 is the bit string for 66
5 is the Hamming distance between the bit strings

./lab02.out 66 28
1000010 is the bit string for 66
0011100 is the bit string for 28
5 is the Hamming distance between the bit strings

./lab02.out 31 62
011111 is the bit string for 31
111110 is the bit string for 62
2 is the Hamming distance between the bit strings

./lab02.out 13 13
1101 is the bit string for 13
1101 is the bit string for 13
0 is the Hamming distance between the bit strings
```

1 C Program

1.1 General Requirements

1. Use emacs to write all of the C source code for this assignment in a file named lab02.c. See the Examples section to see what your C program should output. Compile your C program using the following command. If there are any warnings or errors from the compiler, then you should fix them before you submit this assignment to us for grading.

   ```
   gcc -Wall lab02.c -o lab02.out
   ```

2. Place the files for this assignment in a directory called `lab02`.
3. Your program's source file(s) must compile on odin using the required compiler for this course (`gcc version 11.2.0`).
4. Do *NOT* include `math.h` in your source code. Do *NOT* call any functions in `math.h`.
5. All written and verbal instructions stated by the teaching staff (lecture instructor, lab instructor(s), etc.) must be followed for this assignment. Failure to follow instructions may result in losing points.

1.2 Coding Style Requirements

1. All functions must be commented. Comments must include a brief description of what the function does, its input(s), its output, and any assumptions associated with calling it. If your function has a prototype and an implementation separated into different parts of your source code, then you only need to put the comments for that function above its prototype (there is no need to comment both the prototype and implementation; commenting the prototype is sufficient).
2. All structs, unions, and enums must be commented.
3. All global variables and static variables must be commented.
4. All identifiers must be named well to denote their functionality. Badly named identifiers are not allowed. For example, identifiers like a, aaa, b, bbb, bbbb are bad names for identifiers.
5. Every line of source code must be indented properly and consistently.

1.3 README.txt File Requirements

Make sure to include a `README.txt` file (use a .txt file extension) that includes the following information presented in a reasonably formatted way:

- Your First and Last Name (as they appear on eLC) and your 810/811#
- Instructions on how to compile and run your program.

2 Submission

Before the date and time stated on eLC, you need to submit your files on eLC under the Assignment named Lab 02. The submission on eLC must include only the following files.

1. All source files required for this lab: lab02.c
2. A README.txt file filled out correctly

Do not submit any compiled code. Also, do not submit any zipped files or directories. We only need the files mentioned above with the file extensions aforementioned.

3 Breakout Lab Attendance: 3 points

Breakout lab attendance is required for this assignment, and it is worth three points. Students must physically attend the entire period of the breakout lab section that they registered for during the week of this assignment to earn these three points for attendance. If you complete the assignment before the end of your breakout lab period during the week of this assignment, then you are still required to attend the entire breakout lab period to earn attendance points, and you may use this time for independent study for the next exam in this course.

4 Grading: 7 points

If your program does not compile on odin using the required compiler for this course (`gcc version 11.2.0`), then you'll receive a grade of zero on this part of the assignment. Otherwise, your program will be graded using the criteria below.

Program runs correctly with various test cases on odin	7 points
README.txt file missing or filled out incorrectly	-1 point
Not submitting to the correct Assignment on eLC	-1 point
Late penalty for submitting 0 hours to 24 hours late	-2 points
One or more compiler warnings	-2 points
Not adhering to one or more coding style requirements	-2 points
Not submitting this assignment before its late period expires	-7 points
Penalty for not following instructions (invalid I/O, etc.)	Penalty decided by grader

You must test, test, and retest your code to make sure it compiles and runs correctly on odin with any given set of valid inputs. This means that you need to create many examples on your own (that are different than the aforementioned examples) to ensure you have a correctly working program. Your program's I/O must match the examples given in the Examples section. We will only test your program with valid sets of inputs.