Homework #1 – 100 pts
Due: Wednesday, April 14th at 11:59pm

Your submission should consist of three files: a .pdf file containing answers to the questions and two .c files containing C code. You do not need to submit your assembly code. Legible, handwritten solutions are acceptable for undergraduate students (472). The handwritten solutions must be electronically scanned and submitted as a PDF file. Graduate students (572) must compose their solutions in MS Word, LaTeX, or some other word processor and submit the results as a PDF file.

Note: The instructions in this homework require you to be connected to **flip.engr.oregonstate.edu** using an SSH client. For help, see the links:

Windows Instructions
Mac/Linux Instructions

In this homework you will be writing code to determine if two integers form an amicable pair. You will then compile that code and compare some performance metrics. In order to emphasize the relevance of a well-designed algorithm, you will write two different .c files, one of which utilizes an array-based lookup and the other of which will utilize a "brunt force" approach to validate the amicable pair.

Let's assume that we have two integers: num_a and num_b. If the numbers form an amicable pair, both of the following points will be true:

- The sum of the proper divisors of num_a is equal to num_b.
- The sum of the proper divisors of num_b is equal to num_a.

Important detail: the proper divisors of a number are all positive divisors except for the number itself. For example, the proper divisors of 24 are: 1, 2, 3, 4, 6, 8, and 12.

Consider this explanation from Wikipedia: "The smallest pair of amicable numbers is (220, 284). They are amicable because the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110, of which the sum is 284; and the proper divisors of 284 are 1, 2, 4, 71 and 142, of which the sum is 220."

In several areas of this assignment you will be asked to execute commands involving GCC. If you are not familiar with the compiler, I recommend that you come to office hours or search online for references that explain the command line parameters.

Note: Anytime that you are comparing performance in this homework, you need to use the relative approach that we discussed in chapter 1 of the textbook. i.e. "Version 2 is 1.26 times faster than version 1."

1. Download the file named **amicable_pairs_v1.c** and review the starter code. Implement an algorithm (of your choosing) inside the check_amicable function to determine whether the two numbers form an amicable pair. You must write your code in such a way that it fulfills the requirements given in the comments.

```
/*
        Returns 1 if the two integers form an amicable pair, 0 otherwise.
        This code needs to function correctly regardless of the ordering.
        For example, check_amicable(220, 284) should return 1, as should
        check_amicable(284, 220).
        This function must work correctly for all integers up to 2 billion.
        Be sure that the function prototype remains intact.
*/
int check_amicable(int a, int b) {
        // YOUR CODE GOES HERE
}


/*
        Short program to determine whether two numbers form an amicable pair.
        Do not change the code below
        (except for the values of num_a and num_b)
*/
int main() {
        const int num_a = 17296;
        const int num_b = 18416;

        return check_amicable(num_a, num_b);
}
```

2. Compile your code and **take the time to ensure that your algorithm works**. You can test various scenarios by executing the following commands at the BASH terminal prompt:

```
# compile the code
gcc -std=c11 ./amicable_pairs_v1.c
# print the return value and see if it is 1 or 0
a.out; echo $?
```

3. Try changing the values of **num_a** and **num_b** (and recompiling the code) to make sure that your program properly detects pairs of amicable numbers. Write your code in such a way that it works for all non-negative integers that are less than 2 billion. Part of your grade will be based on the accuracy of your implementation.

4. Once your program is working properly, set **num_a** and **num_b** to values of **1982313333** and **1892277387**, respectively. Next, run the following commands to generate non-optimized 64-bit x86 code.

```
# generate non-optimized 64-bit x86 code
gcc -m64 -fno-asynchronous-unwind-tables -std=c11 -O0 -S -o ./x86_64bit.asm ./amicable_pairs_v1.c
gcc -m64 -fno-asynchronous-unwind-tables -std=c11 -O0 -o ./x86_64bit.exe ./amicable_pairs_v1.c
```

5. Review the x86_64bit.asm file. How many lines of assembly instructions were utilized inside the **check_amicable** function?
   Note: Throughout this homework, labels (ending with a colon) or directives (statements beginning with a dot) are not assembly instructions and should not be included in the count.

6. Use the time utility (included as part of BASH) to determine the amount of CPU time spent in user mode to execute the x86_64bit.exe file. Round your answer to the nearest millisecond.

   ```
   # measure the CPU time
   time x86_64bit.exe
   ```

7. We will now ask the compiler to generate assembly code that occupies less space.

```
# generate space-optimized 64-bit x86 code
gcc -m64 -fno-asynchronous-unwind-tables -std=c11 -Os -S -o ./x86_64bit_s.asm ./amicable_pairs_v1.c
gcc -m64 -fno-asynchronous-unwind-tables -std=c11 -Os -o ./x86_64bit_s.exe ./amicable_pairs_v1.c
```

8. Review the x86_64bit_s.asm file. How many lines of assembly instructions were utilized inside the **check_amicable** function? How much more efficient (in terms of assembly instructions) is the optimized code?

9. Use the same process as step 6 to measure the user mode CPU time of x86_64bit_s.exe
   Which version of code executes faster? By how much?

10. Run the following commands to generate 32-bit and 64-bit versions that are optimized to minimize execution time:

```
# generate O3-optimized x86 code
gcc -m32 -fno-asynchronous-unwind-tables -std=c11 -O3 -o ./x86_32bit_3.exe ./amicable_pairs_v1.c
gcc -m32 -fno-asynchronous-unwind-tables -std=c11 -O3 -S -o ./x86_32bit_3.asm ./amicable_pairs_v1.c
gcc -m64 -fno-asynchronous-unwind-tables -std=c11 -O3 -o ./x86_64bit_3.exe ./amicable_pairs_v1.c
```

11. As before, use the time utility (from BASH) to measure the user-mode CPU time of x86_32bit_3.exe and x86_64bit_3.exe (generated in step 10). Which version executes faster, and by how much?

12. x86 assembly commonly uses eight 32-bit registers. They are: eax, ebx, ecx, edx, esi, edi, ebp, and esp. Which of these registers are used in the x86_32bit_3.asm file that was generated in step 10?

13. Run the following command to generate MIPS assembly code. The output will stored in a file named sample_mips.s

```
# generate MIPS assembly code
mips64-linux-gnu-gcc -mabi=32 -mips32 -static -std=c11 -Os -S -o ./sample_mips.s ./amicable_pairs_v1.c
```

14. How many lines of MIPS assembly instructions were utilized to implement the **check_amicable** function? How does this compare to the number of instructions used to implement the same function in the x86_32bit_3.asm file (the 32-bit x86 code)?
    Again… do not include labels or directives in the count.

15. As noted in the textbook, MIPS has 32 32-bit registers (see page one of the MIPS reference card). Which of these registers are used in the assembly code generated in step 13?
    Provide your answer by giving the numeric identifiers of each register that is utilized within the code (for example, if you see that register $sp is utilized, this should be written as **$29**).

16. In order to reiterate the importance of an efficient algorithm, you will write one more C file. Download **amicable_pairs_v2.c** and implement the code as instructed. You MUST write your code in such a way that it utilizes the arrays and determines if two numbers form an amicable pair. In this version you will obtain faster execution speed at the expense of memory consumption. You will not receive credit if you fail to implement the code as requested.

17. Once you have ensured that your code is working correctly, compile an optimized 64-bit version (using the -O3 flag as demonstrated in step 10). Be sure to use values of **1982313333** and **1892277387** for **num_a** and **num_b**. Use the BASH time utility to measure the user-mode execution time and compare your results to the values from step 10. How much faster is version 2, compared to version 1?