This assignment investigates the limits of various data types in C. A tar file has been provided in hw02.tar.gz. Download and un-tar this file with the command "tar -xvzf hw02.tar.gz" to create the hw02 sub-directory that has three files:

- primeFactor.c
- factorial.c
- Makefile

First, check out primeFactor.c. The code provided will convert the first command line argument to an integer, make sure the result is a positive integer, and then invoke the "nextFactor" function. You will need to code the nextFactor so that it returns the next prime factor of its parameter. (This sounds like a hard job, but here's a hint... if you choose a factor starting at 2, then if that factor divides evenly into the parameter, return it. If not, increment the factor and try again. When you find a number that divides the parameter evenly it must be prime… if not, there is a smaller number that divides the parameter evenly.) The infrastructure of this program is designed to divide n by the returned next factor, and call nextFactor again until nextFactor returns its own argument.

Once you have primeFactor working, move to factorial.c. If you make test_factorial, it will compile and run the program as it stands. As it stands, the factorial.c program will calculate factorials until the factorial no longer fits in a variable defines as "char" (which uses 1 byte or 8 bits.) The function to calculate the factorial is called "fact_char". Factorials are pretty simple… fact_char(4) = 1 * 2 * 3 * 4. It's easiest to code these recursively, using the fact that for n>2, then fact_char(n) = fact_char(n-1).

The "main" function in factorial.c checks to see if the returned factorial is valid and correct. The easiest way to check this for integers is to make sure the result is divisible by 2. Clearly every factorial greater than 2 will have a factor of 2. However, since half of the numbers are divisible by 2, there is still a 50% chance that an invalid result is still divisible by 2. We resolve this problem by noting that everything after 5! must have factors of both 2 and 5, so therefore must be divisible by 10. (There is still a chance for allowing an invalid result, but the chances are much smaller, and it turns out that this is a reasonable test for validity for integer factorials.)

Your job is to copy the fact_char function, and make a fact_short function that uses short integers instead of characters. Make sure that the fact_short argument takes a short integer argument, and returns a short integer result. Then copy the loop in main and change it to invoke fact_short to find out what is the largest valid factorial that fits in a short integer. Make sure that this loop uses a short integer to represent both the argument to fact_short and the result from fact_short.

Do the same thing for integers (fact_int), and long integers (fact_long). Note that when you use long integers, the printf format character "%d" no longer works. You can solve this by replacing "%d" with "%ld", which print understands as "long decimal".

We can get even bigger numbers by creating another new function called "fact_float" that uses floating point numbers. Copy the loop in main for floats as well, but the "%" (remainder) operator does not work on floating point numbers. Therefore, we need to change the definition of "valid". For floating point numbers we can check to see that the result has not become the constant "INFINITY", which indicates that the result is too large to fit in a float. If the result of fact_float is in variable r_float, we can do this by coding “if (r_float == INFINITY) break;”. Note that there may be rounding errors in large numbers less than infinity, but we will ignore these rounding errors for now.

The format character in the printf statements need to be updated as well. We should use “%f” instead of “%d” or “%ld” for floating point results. Actually, it looks best to use:

```
printf("%f! = %g\n",i_float,r_float);
```

… when printing valid factorials. This is because the %g format character tells printf to use scientific notation when printing the results (which is much easier to read when numbers get big.)

Copy your floating point results to fact_double, and create a loop in main to use double precision floating point numbers as well.

It’s interesting how fast factorial numbers grow, and what the effect of the various different data types have on how large numbers can be accommodated.

When you are done testing, run “make submit” to create a file called “hw02_<*userid*>.tar.gz”, where <*userid*> is your gmail userid. Upload this file on myCourses under Content, Homework Submissions, Homework 02 Submission. This assignment is due at 11:59 PM on Sunday, February 05, 2017. You may submit as many times as you wish; but only the latest submission will be graded.

This assignment is worth 10 points. Your grade will be calculated as follows:

- If you submit late without an extension, there will be a two-point deduction for every 24 hours you are late. Extensions are available only in special circumstances, and can only be given by the instructor or a TA.
- There will be a three-point deduction for submissions that do not follow the required format. For instance, if you do not run make submit to create the correct tar file, or if you run make submit on a machine where your userid is incorrect so that the resulting tar file has the wrong name OR wrong contents (your userid is used to create a sub directory that is contained in the tar file).
- There will be a four-point deduction for each C file if there are compiler errors when compiling your code.
- There will be a three-point deduction if your code does not run to completion on any of the test cases used to grade your code. For instance, if your code causes a segmentation violation, there will be a three-point deduction.
- There will be a two-point deduction for each class of compiler warning message issued when compiling your code.
- There will be a two-point deduction if primeFactor does not calculate the correct results for the test case in the Makefile test_primeFactor, and a one-point deduction for each of three unpublished (positive integer) test cases on primeFactor.
- There will be a one-point deduction for each of short, int, long, float, and double which does not correctly identify the first incorrect result (using the criteria for “valid” specified above.)