

Due: 4-29-2015 (April 29, 2015)

Filename: `divisibles.c`, `fibonacci.c`, `quadratics.c`, `powers.c`, `pascal.c`, `factorization.c`,
`piseries.c`

Add your Name and UC Davis ID number as the first comment of each file to make it easier for TA's to identify your files.

All programs should compile with no errors. Warnings are okay, as long as execution is still correct. Compile programs using `gccx filename.c`, then execute with `./a.out` or specify your executable filename during compilation using `gccx -o outputFile filename.c` and execute with `./outputFile`.

Use `diff` with the `*.out` files to compare the output of your executable files with the official output files. (Syntax: `diff file1 file2`) `diff` will compare two files line by line, and list the changes that need to be made to the first file to make it identical to the second. If there are no differences between the two files, then `diff` outputs nothing.

Submitting files

Usage: `handin cs30 assignmentName files...`

The assignmentNames are Proj1, Proj2, Proj3, Proj4, Proj5. **This is Proj2.** Do not hand in files to directories other than Proj2, they will not be graded. The files are the names of the files you wish to submit. For this assignment, your files will be `divisibles.c`, `fibonacci.c`, `quadratics.c`, `powers.c`, `pascal.c`, `factorization.c`, `piseries.c`

Therefore,

you would type from the proper directory:

\$ handin cs30 Proj2 programName.c

Do NOT zip or compress the files in any way. Just submit your source code.

\$ handin cs30 Proj2 `divisibles.c`, `fibonacci.c`, `quadratics.c`, `powers.c`, `pascal.c`, `factorization.c`,
`piseries.c`

If you resubmit a file with the same name as one you had previously submitted, the new file will overwrite the old file. This is handy when you suddenly discover a bug after you have used `handin`. You have up to 4 late days to turn in files, with a 10% deduction for each late day. The deadline for each project (and late day) is 11:59pm.

TA comments:

If it makes sense to put code into a function, use function definition. Otherwise, you can write your code in main, you do not HAVE to use functions for each of these problems. Just to be clear, this does not mean hard code the print statements into your main function. It just meant that if it makes sense to move part of your code into a separate function, do so, otherwise, it's okay to keep all the code in main.

Some of you have been enabling c99 in your compilers so you can do things like:

```
for (int i=0; i< NUM; i++)  
instead of:  
    int i;  
    for (i=0;i<NUM;i++)
```

Do NOT do this. For consistency we will not be enabling c99 during your grading scripts, so this will result in an error and you **WILL LOSE POINTS**. Please just write the extra line of code.

OUTPUT FILES and DIFF:

For this project we are giving .out files for all the problems, and input files when needed. These are **EXECUTABLE FILES, NOT TEXT** files. You will need to pipe the outputs of these files to an output text file as well as the outputs of your programs into a separate output text file for diff comparison. For example:

If you are working on hello.c:

Assume the official provided file is **hello.out**, and your hello.c file generates **hello**.

To compare the outputs of the two .out files, you need to:

```
$ ./hello > myHello.txt  
$ ./hello.out > hello.txt  
$ diff myHello.txt hello.txt
```

If there were no differences, nothing will print when you run ‘diff’. If you get a “Permission denied” error for any of the files (ex. hello.out), try `$ chmod 777 hello.out` in the console/terminal.

For the problem with required input file, do the following (ex. inchtocm.c):

```
$ ./inchtocm < inchtocm_input > myInchtocm.txt  
$ ./inchtocm.out < inchtocm_input > inchtocm.txt  
$ diff myInchtocm.txt inchtocm.txt
```

`$ diff -w myInchtocm.txt inchtocm.txt` (ignore differences of white spaces) - used for grading

Checking with diff is very important as we will be using scripts to grade your homeworks, and even the smallest difference will result in your not getting points for a problem.

*Since this project also deals with a lot of formatting, use these executable files to correctly map spacing from our results to yours. (i.e., **DO NOT** use the `-bB` or `-w` arguments with diff). Some comments about how to properly format printing are included below as “TA comments”.*

FOR THE GRAPHICS PROGRAMS: All the graphics programs are optional. You should still do them, but they will **NOT** be graded. Each program that is optional will have the word **(OPTIONAL)** next to it. **There are no graphics programs in Project 2.**

Programming project #2: Ch. 4: 4, 8; Ch. 5: 2, 6, 12; Ch. 6: 2, 7

Programs reproduced below for your convenience: (from the Programming Exercises section of each chapter)

CH.4: (pg. 133-136)

4. Filename: **divisibles.c** (use divisible.out to test your solution)

Why is everything either at sixes or at sevens?

-Gilbert and Sullivan, H.M.S. Pinaforte, 1878

Write a program that displays the integers between 1 and 100 that are divisible by either 6 or 7.

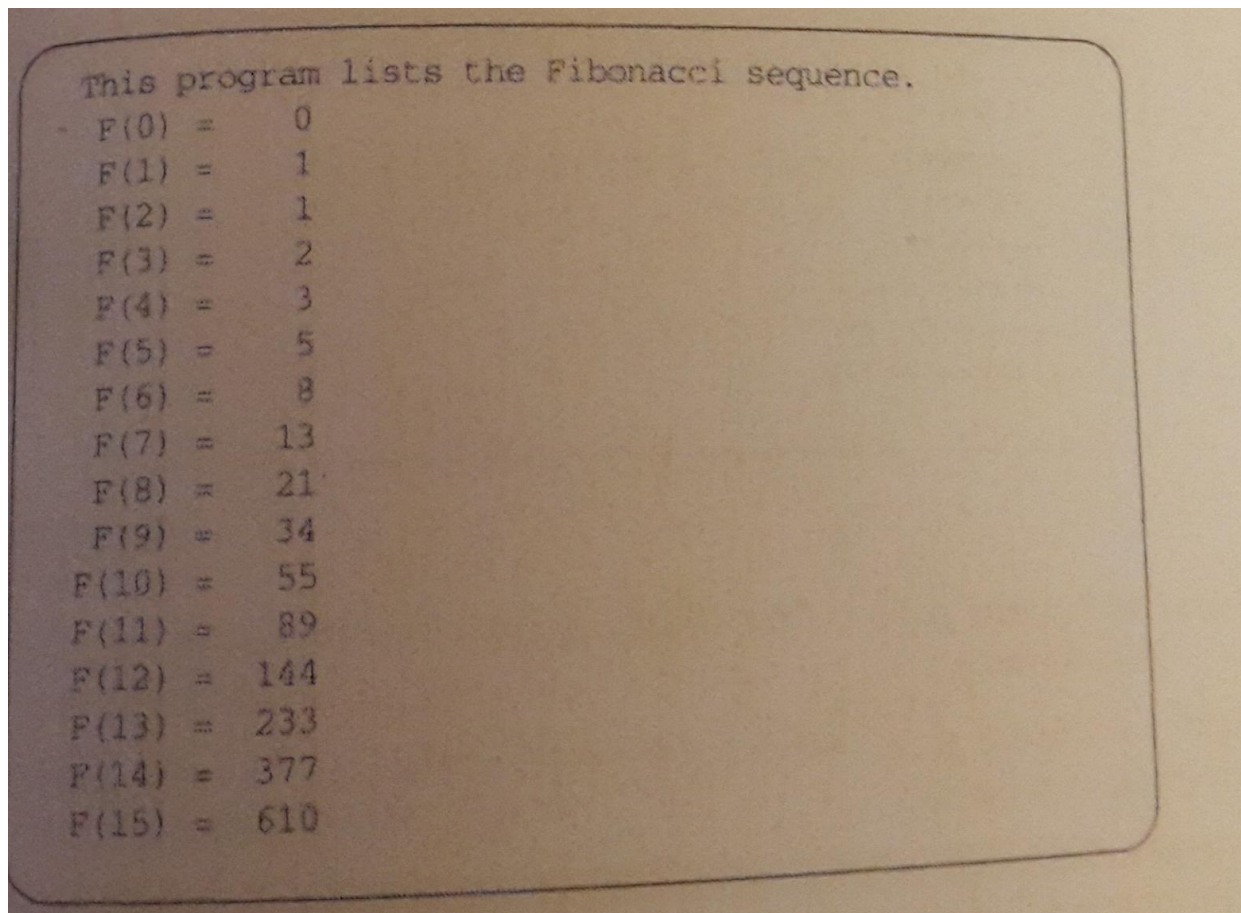
TA comment: Use “%4d” to print the integers.

8. filename: **fibonacci.c** (use fibonacci.out to test your solution)

In mathematics, there is a famous sequence of numbers called the Fibonacci sequence after the thirteenth-century Italian mathematician Leonardo Fibonacci. The first two terms in this sequence are 0 and 1, and every subsequent term is the sum of the preceding two. Thus the first several numbers in the Fibonacci sequence are as follows:

$$\begin{array}{rcl} F_0 & = & 0 \\ F_1 & = & 1 \\ F_2 & = & 1 \quad (0 + 1) \\ F_3 & = & 2 \quad (1 + 1) \\ F_4 & = & 3 \quad (1 + 2) \\ F_5 & = & 5 \quad (2 + 3) \\ F_6 & = & 8 \quad (3 + 5) \end{array}$$

Write a program to display the values in this sequence from F_0 through F_{15} . Make sure the values line up as shown in the following sample run:



TA comment: Right-align "F(x)" where $x < 10$, with $11 < x < 15$ by adding an extra space character.

TA comment: Use "%d" to print the index and "%4d" for printing the resulting values of fibonacci.

CH.5: (pg. 178-183)

2. filename: **quadratics.c** (use quadratics.out with input files quadratics_input1, quadratics_input2, quadratics_input3)

In high-school algebra, you learned that the standard quadratic equation

$$ax^2 + bx + c = 0$$

has two solutions given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The first solution is obtained by using + in place of \pm ; the second is obtained by using - in place of \pm .

Write a C program that accepts values for a , b , and c , and then calculates the two solutions. If the quantity under the square root sign is negative, the equation has no real solutions, and your program should display a message to that effect. You may assume that the value for a is nonzero. Your program should be able to duplicate the following sample run:

Enter coefficients for the quadratic equation:

a: 1 ↵

b: -5 ↵

c: 6 ↵

The first solution is 3

The second solution is 2

TA comment: the library `math.h` has the `sqrt` function. Include `math.h` in your code. The syntax for the `sqrt` function is:

`double sqrt (double val)`

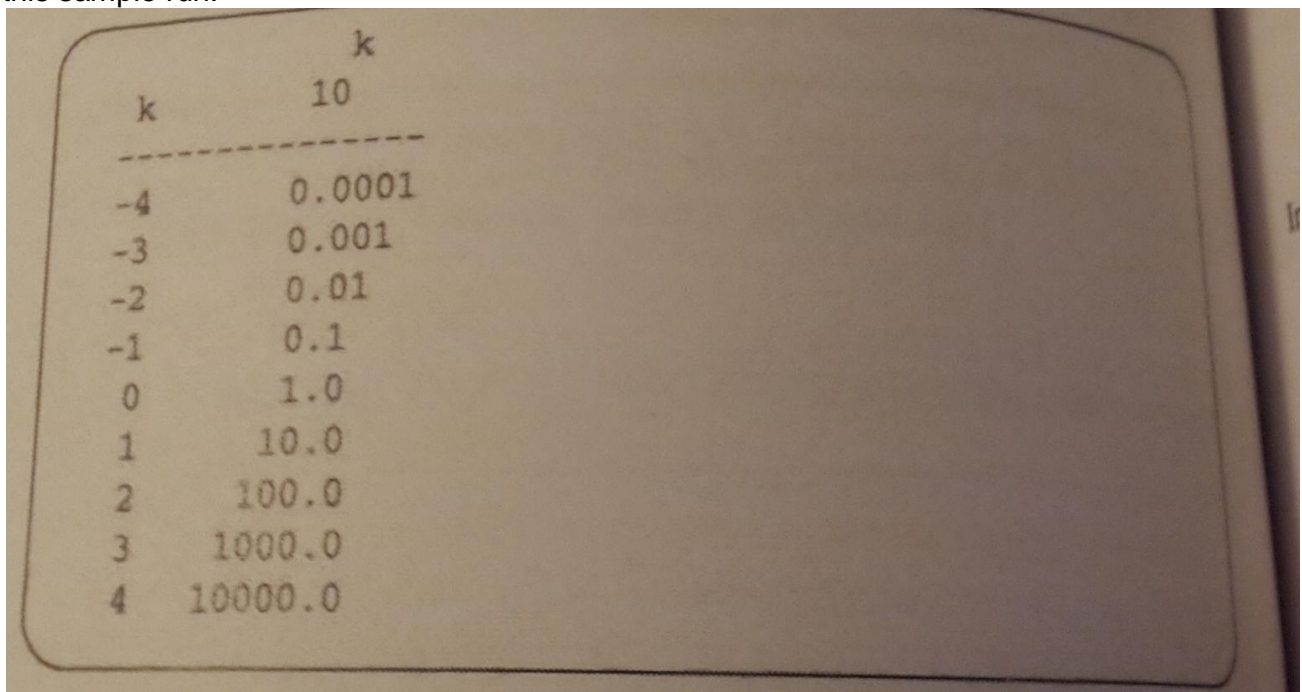
Also, you can assume that a, b, c are real values. Print the solutions using `%g`. If there are no solutions, print the statement "The equation has no real solutions.\n". If there is only one solution, print the statement: "The only solution is X \n", where X is the number.

6. filename: `powers.c` (use `powers.out` to test your solution)

Write a function `RaiseRealToPower` that takes a floating-point value x and an integer k and returns x^k . Implement your function so that it can correctly calculate the result when k is negative, using the relationship

$$x^{-k} = \frac{1}{x^k}$$

Use your function to display a table of values of 10^k for all values of k from -4 to 4, as shown in this sample run:



k	10 ^k
-4	0.0001
-3	0.001
-2	0.01
-1	0.1
0	1.0
1	10.0
2	100.0
3	1000.0
4	10000.0

Note: There is no single *printf* format code that will correctly display each of the output lines in this table. To write a main program that produces precisely this output, you need to use a different format specification when the value of *k* is negative.

TA comment: As the note says, there is no one *printf* to get both the 0/positives and negatives to align perfectly. Use this link as a **hint**: <http://stackoverflow.com/questions/1000556/what-does-the-s-format-specifier-mean> for one of these print formats. For the second format, use %7.1f.

Also, there is no way to print 10^k so print the *k* on the first line, and the 10 on the second line. Use %2d to print the *k* values.

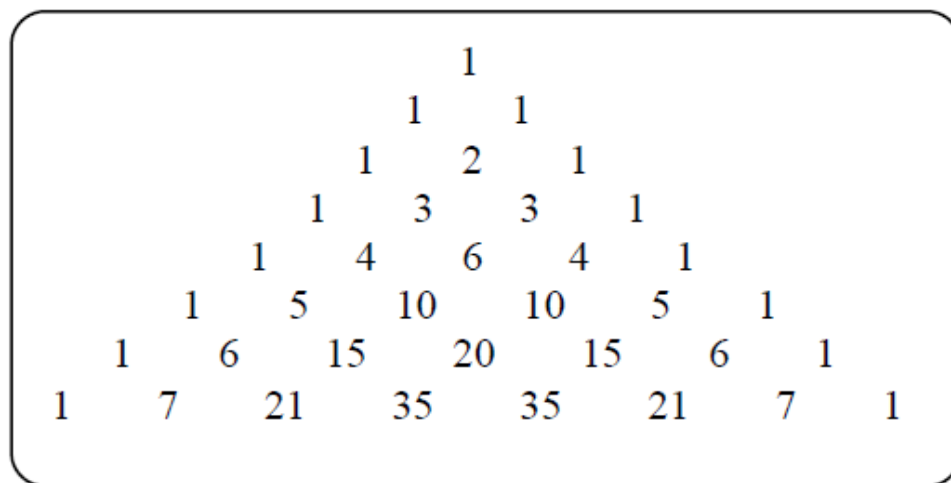
12. filename: **pascal.c** (use pascal.out to test your solution)

The values of the *Combinations* function used in the text are often displayed in the form of a triangle using the following arrangement:

```
      C(0,0)
    C(1,0) C(1,1)
  C(2,0) C(2,1) C(2,2)
C(3,0) C(3,1) C(3,2) C(3,3)
C(4,0) C(4,1) C(4,2) C(4,3) C(4,4)
```

and so on. this figure is called Pascal's Triangle after the seventeenth-century French mathematician Blaise Pascal, who invented it. Pascal's Triangle has the interesting property that every interior entry is the sum of the two entries above it.

Write a C program to display the first eight rows of Pascal's Triangle like this:



TA comment: To figure out the spacing for each line, think about how many characters each line occupies. Then analyze how that character number relates to the row number.

CH.6: (pg. 214-218)

2. filename: **factorization.c** (use factorization.out with input files factorization_input1, factorization_input2)

In many cases, it is not enough to know whether a number is prime; sometimes, you need to know its factors. Every positive integer greater than 1 can be expressed as a product of prime numbers. This factorization is unique and is called the **prime factorization**. For example, the number 60 can be decomposed into the factors $2 \times 2 \times 3 \times 5$, each of which is prime. Note that the same prime can appear more than once in the factorization.

Write a program to display the prime factorization of a number n . The following is a sample run of the program:

Enter number to be factored: **60** ↵

2 * 2 * 3 * 5

TA comment: the <stdbool.h> library might be useful. Also, you might need to define TRUE and FALSE to be 1 and 0, respectively.

7. **piseries.c** (use piseries.out to test your solution)

The technique of series approximation can be used to compute approximations of the mathematical constant π . One of the simplest series that involves π is the following:

$$\frac{\pi}{4} \cong 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

Write a program that calculates an approximation of π consisting of the first 10,000 terms in the series above.

TA comment: Use %12.10lf to print the value of pi.