CSCI 1312 (Introduction to Programming for Engineering), Fall 2015

Homework X

Credit: Up to 30 extra-credit points.

1 General Instructions

Do as many (or few) of the following problems as you like. Notice that you can receive at most 30 extra-credit points, but be advised that any points you earn can only help your grade — that is, I will add them to your total points before dividing by the sum of the points on the required assignments.

I am also open to the possibility of giving extra credit for other work — other problems/programs, a report on something course-related, etc. If you have an idea for such a project, let's negotiate (by e-mail).

For this assignment, please work individually, without discussing the problems with other students. If you want to discuss problems with someone, talk to me.

2 Programming Problems

Do as many (or as few) of the following optional programming problems as you like. Submit source code and other files by e-mail, as for previous assignments. (I.e., submit your program source (and any other needed files) by sending mail to `bmassing@cs.trinity.edu`, with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 1312 extra credit"). You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (Up to 5 extra-credit points.) Write a C program that, given the name of a text file as a command-line argument, reads the contents of the file and produces a histogram of word lengths, where a "word" is one or more alphabetic characters. So for example given an input file containing the following text

   ```
   Now is the time for all good persons to come to the aid of their party.

   A really long word, though perhaps not the longest in English,
   is "antidisestablishmentarianism" (28 letters).
   ```

   it would produce the following

   ```
   1 *
   2 ******
   3 ********
   4 *****
   5 **
   6 **
   ```

```
   7 *****
   8
   9
  10
  11
  12
  13
  14
  15
  16
  17
  18
  19
>=20 *
```

(Notice that it groups all words of length at least 20 into a single output line — simpler to code and in my opinion reasonable.)

2. (Up to 5 extra-credit points.) Homework 5 asked you to write a program that compared two ways of assigning "random" numbers to "bins". Output was a list of counts for the two methods. A possibly more useful output would be a (vertical) bar graph showing values of the counters. Improve the program (either your solution or the sample solution) so that it prints such a graph for each of the two methods. Since the height of these bars could be quite large if the number of samples is large, have the program also get as input a "number of samples per row" and use it to create the graph, with the height of each bar equal to the corresponding counter divided by this number of samples per row. Sample execution:

```
seed?
5
number of samples?
1000
number of bins?
6
samples per graph row?
10

counts using remainder method:

(0) 154
(1) 188
(2) 171
(3) 161
(4) 155
(5) 171

....X............
....X..X........X.
....X..X..X.....X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
```

```
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.

counts using quotient method:

(0) 172
(1) 175
(2) 183
(3) 150
(4) 168
(5) 152

.......X..........
.X..X..X..........
.X..X..X.....X....
.X..X..X.....X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
.X..X..X..X..X..X.
```

Hint: You could solve this problem just using the variables in the original program, but it's easier to represent the graph as a 2D array of characters. There is an example of doing something similar if simpler in sample program `array2d.c`.

3. (Up to 5 extra-credit points.) Write a function that evaluates polynomial $p(x)$ given the coefficients of p and one or more values of x. You can prompt for the coefficients or get them from command-line arguments; once you have them, repeatedly prompt for values of x until the user enters something non-numeric. A supposedly efficient way to evaluate a polynomial is with "Horner's rule" (check the Wikipedia article if you're not familiar with this approach), which can be implemented with a loop or recursion. (My program does both.) A sample execution prompting for the coefficients:

```
a.out
degree of polynomial (highest power)?
3
coefficients (starting with highest power)?
2 4 3 5
p(x) = 2.000000(x**3) + 4.000000(x**2) + 3.000000(x**1) + 5.000000
x?
10
iterative version:
p(10.000000) = 2435.000000
recursive version:
p(10.000000) = 2435.000000
x?
100
iterative version:
p(100.000000) = 2040305.000000
recursive version:
p(100.000000) = 2040305.000000
x?
invalid input
```

and one getting them from the command line:

```
a.out 2 4 3 5
p(x) = 5.000000(x**3) + 3.000000(x**2) + 4.000000(x**1) + 2.000000
x?
10
iterative version:
p(10.000000) = 5342.000000
recursive version:
p(10.000000) = 5342.000000
x?
100
iterative version:
p(100.000000) = 5030402.000000
recursive version:
p(100.000000) = 5030402.000000
x?
invalid input
```

4. (Up to 10 extra-credit points.) In most of the programs we wrote in class and for homework

we made some attempt to "validate" user input (e.g., check that inputs are numeric when they're supposed to be, positive when they're supposed to be, etc.). Doing this for many variables is apt to produce a lot of uninterestingly-repetitive code. Also, if the input was not valid we just bailed out of the program rather than trying again. Propose and implement one or more functions that would address one or both of these possible shortcomings, and submit it/them with a short program that could be used to test it/them. Be sure to include comments that describe the function's parameters and behavior (does it exit the program on error or prompt again or what). You might like to have functions for working with input from standard input and also functions that work with command-line arguments.

5. (Up to 10 extra-credit points.) In class I said that getting "a line" of character data (a sequence of characters read from a file or standard input ending with the end-of-line character) was surprisingly difficult and error-prone in C. Propose and implement a function or functions that gets a full line of character data in a way that does not limit the length of the input data but also does not risk overflowing an array, and submit it/them with a short program that could be used to test it/them. (You will almost surely need `malloc` to make this work.)

6. (Up to 10 extra-credit points.) The textbook presents code for various sorting algorithms, some suggestions for testing them, and an analysis of how the amount of computation involved (estimated as the number of comparisons) depends on the number of elements being sorted. One way to test that a particular implementation of one of these algorithms is correct and also check the claim about amount of computation goes as follows: Generate an array of N values using `rand()`, sort them, and have the program check that the resulting values are in order. During the sort, count the number of comparisons and print that at the end. Your mission for this problem is to write such a program. Input to the program is a seed (to pass to `srand`) and a count (number of values to generate/sort). Output is a message saying whether the sort worked (i.e., the values are in order) and a count of comparisons. You *could* prompt for the input, but if instead you get it from command-line arguments you can more easily call the program repeatedly for different input sizes. And you *could* use a fixed-size or variable-length array for the data, but if you want to allow for running the program with large numbers of elements it's probably better to allocate space for the array with `malloc`. How much credit you get for this problem depends on how much of this advice you follow in addition to whether the program does what it's supposed to do. Below are some sample executions of such a program, written to get its arguments from the command line:

```
a.out 5 10
sort of 10 values (seed 5) succeeded, 45 comparisons
a.out 5 100
sort of 100 values (seed 5) succeeded, 4950 comparisons
a.out 5 1000
sort of 1000 values (seed 5) succeeded, 499500 comparisons
a.out 5 10000
sort of 10000 values (seed 5) succeeded, 49995000 comparisons
```