CSCI 1312 (Introduction to Programming for Engineering), Fall 2018

Homework 7

Credit: 40 points.

1 Reading

Be sure you have read (or at least skimmed) the assigned readings from chapter 8 (not including the material on searching and sorting).

2 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. 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 hw 7" or "CS1 hw 7"). 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. (20 points) C, like many programming languages, has a library function (`rand()`) that can be used to generate a "random" sequence of numbers (quotes because it's not truly random, as mentioned in class). Many languages have a similar function that generates "random" numbers in some specified range, useful if for example you're trying to simulate rolling a 6-sided die. C doesn't have such a function, but you can get the same effect using `rand()` and a little additional code. `rand()` itself generates a number between 0 and the library-defined constant value `RAND_MAX`, so to get a value in a smaller range you have to somehow map the larger range to the smaller one. The somewhat obvious way to do this is by computing a remainder (e.g., to map to two possible values, assign even values to 0 and odd values to 1). (I'll call this the "remainder method".) But with some implementations of `rand()` this gives results that aren't very good. The conventional wisdom is therefore to instead try to do a more-direct map (e.g., to map to two possible values, assign values from 0 through `RAND_MAX/2` to 0 and the remaining values to 1). (I'll call this the "quotient method".)

   Your mission for this problem is to complete a C program that, given a number of samples N and a number of "bins" B generates a sequence of N "random" numbers, uses both methods (remainder and quotient) to map each generated number to a number between 0 and *B-1* inclusive, and counts for each method how many elements of the sequence fall into each bin (e.g., for each method bin 0 is how many elements of the sequence map to 0), and prints the result, as in the sample output below. To help you (I hope!) I'm providing a starter program, link below, which you should use as your starting point. Code at the bottom of the program shows how to apply both methods to something returned by `rand()`. The remainder method

is straightforward; the quotient method is less so, but see the footnote[1] if you're curious.

One other thing to know about `rand()` is that by default it always starts with the same value (and produces the same sequence). To make it start with a different value, you can call `srand()` with an integer "seed", so your program should prompt for one of those too.

Sample execution (assuming you call your program `rands` and compile with `make`):

```
[bmassing@diasw04]$ ./rands
seed?
5
how many samples?
1000
how many bins?
6
counts using remainder method:
(0) 154
(1) 188
(2) 171
(3) 161
(4) 155
(5) 171
counts using quotient method:
(0) 172
(1) 175
(2) 183
(3) 150
(4) 168
(5) 152
```

If you feel ambitious you could also have the program print maximum and minimum counts and the difference between them, as a crude measure of how uniform the distribution is:

```
[bmassing@diasw04]$ ./rands
seed?
5
how many samples?
1000
how many bins?
6
counts using remainder method:
```

[1] The formula is cryptic, and expressing it in C more so, but here goes: Suppose n is the value returned by `rand()`. We could convert it to a bin number in two steps: First, we scale it to a floating-point number in the range from 0 up to but not including 1, thus:

$$x = n/(\texttt{RAND\_MAX} + 1)$$

(We divide by `RAND_MAX+1` so the largest possible value maps to something still slightly less than 1. Note also that in the code we have to be sure this addition is done using floating point, since otherwise it could overflow; we can do this by writing the 1 as 1.0.) Then we scale this range of floating-point values to our desired range (still in floating point) by multiplying by B, which gives from 0 up to but not including B. Finally, we convert back to an `int` with a cast, which drops the fractional part, and leaves us with an index between 0 and *B-1* inclusive.

```
(0) 154
(1) 188
(2) 171
(3) 161
(4) 155
(5) 171
min = 154, max = 188, difference 34
counts using quotient method:
(0) 172
(1) 175
(2) 183
(3) 150
(4) 168
(5) 152
min = 150, max = 183, difference 33
```

(You will get an extra-credit point for doing this.)

Here is a starter program that prompts for the seed, generates a few "random" numbers, and illustrates the two methods of mapping to a specified range: http://www.cs.trinity.edu/~bmassing/Classe

Of course, your program should check to make sure all the inputs are positive integers. (Yes, error checking is a pain, but it's an incentive to get better at copy-and-paste?)

Hints:

- If you find the problem description confusing, maybe an example will clarify a bit: Suppose input specifies 100 samples and 2 bins. The program should generate a sequence of 100 "random" numbers. For the remainder method, bin 0 will be how many elements of this sequence are even, while bin 1 will be how many are odd. For the quotient method, bin 0 will be how many elements of this sequence are in the first half of the range from 0 through `RAND_MAX`, while bin 1 will be how many are in the second half.

2. (20 points) In class we have written two programs to compute an element of the Fibonacci sequence, one using a loop (iteration) and one using recursion. If you compile and run both programs, you may notice that the one using recursion can be quite slow with all but small inputs. The reason for that is fairly obvious if you think about how it works — the recursive function does quite a lot of duplicate computation. One way to improve the performance of such a function is with a technique referred to as *memoization*[2], in which every time you compute a result you save it for possible reuse. Because I was curious myself about how the iterative and recursive versions compared, and how much memoization might help, I wrote a program that defines functions for all three approaches (using `long long` rather than `long` for the elements of the sequence, to allow computing larger values) and times them. For me only the simple recursive function took more than trivial time, so I added code to also count the number of recursive calls. You can find the result — minus the actual computation of values! — in http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2018fall/Homeworks/HW07/Problems/fibonac
Sample output for an input of 46:

[2]No, that's not a typo — there really is no "r" in this word!

```
which index?
computing fibonacci(46)

iterative version:
result 2971215073 (time 0.00000000)

recursive version:
result 2971215073 (time 46.40543509, count 5942430145)

memoized recursive version:
result 2971215073 (time 0.00000095, count 91)
```

Your mission is to fill in the blanks in this starter program so that it performs the desired computation. (Look for comments with the word `FIXME`.) For the iterative and simple recursive approaches, you can get the code from the course "sample programs" page; you will just need to change some data types from `int` to `long long`.

For the memoized recursive approach, a simple way to use this technique is to have an array for saving previously-computed results, with the n-th element of the Fibonacci sequence stored as the n-th element of the array, and a value of 0 meaning "has not been previously computed". This array could be an additional argument for the function, or it could be a global variable. (Usually global variables are discouraged, but for this problem they might make sense.) If you take the global-variable approach, your program should do something reasonable if it isn't big enough, perhaps just rejecting any input that would overflow it.

The only changes you should need to make in the starter program are:

- Fill in the body of functions `fibonacci_iterate()` and `fibonacci_recurse()`. It's okay to use code from the "sample programs" page (edited so it computes a `long long`), and in fact that is what I strongly recommend. (I didn't just do this for you myself because I hope that copying and pasting will encourage you to look at the copied/pasted code.)
- Fill in the body of function `fibonacci_recurse_m()`. You can add additional parameters if you like, or use global variables for the saved values.

Turn in the resulting code. Note that all three functions should return the same value for the n-th element of the Fibonacci sequence; the only difference should be execution time and count of recursive calls. I'm not asking you (at this point?) to formally collect results that would show how the count of recursive calls, and the execution time, increases as the input variable increases, but you may find it interesting to try some different values and observe!

3 Honor Code Statement

Include the Honor Code pledge or just the word "pledged", plus *at least one of the following* about collaboration and help (as many as apply).[3] Text *in italics* is explanatory or something for you to

[3] Credit where credit is due: I based the wording of this list on a posting to a SIGCSE mailing list. SIGCSE is the ACM's Special Interest Group on CS Education.

fill in. For programming assignments, this should go in the body of the e-mail or in a plain-text file `honor-code.txt` (no word-processor files please).

- This assignment is entirely my own work. *(Here, "entirely my own work" means that it's your own work except for anything you got from the assignment itself — some programming assignments include "starter code", for example — or from the course Web site. In particular, for programming assignments you can copy freely from anything on the "sample programs page".)*
- I worked with *names of other students* on this assignment.
- I got help with this assignment from *source of help — ACM tutoring, another student in the course, the instructor, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*
- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*
- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

4 Essay

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what about the assignment you found interesting, difficult, or otherwise noteworthy. For programming assignments, it should go in the body of the e-mail or in a plain-text file `essay.txt` (no word-processor files please).