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

Homework 5

Credit: 20 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 homework 5" or "CS1 hw5"). 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. (10 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). This function can be used to generate a value in a specified range (e.g., between 0 and 5 inclusive if you're trying to simulate rolling a 6-sided die). `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 (see starter program below), 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 just two possible values, assign values from 0 through `RAND_MAX/2` to 0 and the remaining values to 1).

   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 to map each generated number to a number between 0 and $B - 1$ inclusive, and shows the distribution of results 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.

   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:

```
[bmassing@diasw04]$ ./a.out
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]$ ./a.out
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
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: rands.c[1].

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?)

2. (10 points) In class we wrote 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 will likely notice that the one using recursion can be quite slow, especially if you change the type of the element from `int` to something that can represent larger values and try it for larger 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 `int` 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 fibonacci-starter.c[3]. Sample output for an input of 46:

```
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".

[1] `http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2015fall/Homeworks/HW05/Problems/rands.c`

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

[3] `http://www.cs.trinity.edu/~bmassing/Classes/CS1312_2015fall/Homeworks/HW05/Problems/fibonacci-starter.c`

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.) The program should do something reasonable if this array is not 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 do this 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. Notice 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!