CSE/IT 122: Homework 2

Make sure all code is commented with doxygen style comments, follows the Linux Kernel Coding Style, and has a header. See the file `doxygen.c` on Canvas for correct doxygen usage and the required header for your source code files.

If a function name is given in the directions, please use the declaration of the functions as is. Do not change them. Use a `Makefile` to compile all programs. For output follow the given sample output exactly.

If the problem asks you to type your answers, use Latex, Open Office/Office Equation Editor or similar. You will turn all typed answers in a single pdf file named `cse122_firstname_lastname_hw2.pdf`. Clearly label your answers in the pdf.

Problems

1. You can use the following formula to calculate the n-th Fibonacci term:

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right] \tag{1}$$

   You can approximate F_n by making the observation that $\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$ gets large while $\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$ gets small as n gets large:

$$F_n \approx \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} \right] \tag{2}$$

   You can use the approximation for F_n to determine the largest n that can be calculated before overflow occurs. For `unsigned long`'s the largest value your machine can use is `ULONG_MAX` defined in `limits.h`. In equation (2), substituting `ULONG_MAX` for F_n, solving for n, and taking the floor, gives you an approximation of the largest n you can use before overflow occurs.

   Unfortunately, overflow is not the only issue with calculating Fibonacci terms this way. There is error in the calculation as it involves a $\sqrt{5}$ term, which is irrational. The error also gets compounded as you raise things to a power.

You are going to write a program that finds the largest value of n you can use before overflow occurs and investigate the error between finding Fibonacci terms using equation (1) and finding it using $F_n = F_{n-1} + F_{n-2}$.

Steps

(a) Name your program `fib_error.c`

(b) Write a function

`unsigned max_n(void)`

that determines the largest n you can find before overflow occurs. The function determines n from the equation

$$ULONG\_MAX = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$$

As your solution involves square roots and logs, use `double`s for calculations. Take the floor and cast n to an `unsigned` integer before returning it. Make sure you use `ULONG_MAX` from `limits.h` in your calculation. You may find that the n this function finds is still too large and overflow occurs when you calculate F_n. Just keep subtracting 1 from the value until you find the max n you can calculate without overflow.

(c) Write a function

`double fib(unsigned n)`

that uses the formula

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]$$

to find the n-th Fibonacci term. This function returns a `double`.

(d) Write a function

`unsigned long * fib_array(unsigned n)`

that dynamically allocates an array of `unsigned long`s to store the results of $F_n = F_{n-1} + F_{n-2}$ in the array. Start the array with $F_0 = 0$ and $F_1 = 1$. Return the array of Fibonacci values. Use a `for` loop, not recursion to populate the array with Fibonacci values. Make sure you capture the largest possible Fibinacci value you can in your array.

As a check, you can compare your calculations with known Fibonacci numbers using the website

`http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibtable.html#100`

Like the array, the website uses $F_0 = 0$, while equation (1) assumes $F_0 = 1$. NB equation (1) and the array are off by one in their indexing. Make sure you account for this fact when you compare their values (see below).

(e) In `main()`, print out the array of Fibonacci numbers. One per line.

(f) In `main()`, compare the Fibonacci value from equation (1) with the values found in the array. Cast the `double` to a `unsigned long` before you compare. If they are the same, do nothing, otherwise print an error message stating which term differs, how much they differ by, and the percentage error between the two terms. To find the percentage error, you first find the relative error. Relative error (RE) is defined as

$$RE = \frac{x_o - x}{x}$$

where x is the true value and x_o is the measured value. The percentage error is 100% times the relative error.

(g) The Appendix has sample output for this program. Capture the output to a file named `fib_error.out`

(h) Make sure you free the memory you allocated. Check with valgrind.

2. Write a program (`fib.c`) that calculates the n-th Fibonacci term using command line arguments. The sample program `sample_getopt.c` shows the use of `getopt()` to process command line arguments. Use the array approach to find the n-th Fibonacci term. The array starts with $F_0 = 0$ and $F_1 = 1$. The user runs the program like this:

```
$ ./fib -n 100
n is too large -- overflow will occur
the largest Fibonacci term that can be calculated is 93
$ ./fib -n 93
The 93 Fibonacci term is 12200160415121876738
```

(a) Most of the program is already written. Write a function

`int check_n(unsigned n, unsigned *max_n)`

that returns 0 if the given n is less than or equal to the maximum value and -1 if not. In `main()`, use this function to determine if you can calculate the n-th Fibonacci term or not.

(b) Make sure you free the memory you allocate. Check with valgrind. Also, the size of the array you allocate should be dependent on the n the user enters.

3. You are given the following code fragments:

I.
```
sum = 0;
for (i = 0; i < n; i++)
    sum++;
```

II.
```
sum = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
       sum++;
```

III.
```
sum = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n * n; j++)
        sum++;
```

IV.
```
sum = 0;
for (i = 0; i < n; i++)
    for(j = 0; j < i; j++)
       sum++;
```

V.
```
sum = 0;
for (i = 0; i < n; i++)
    for(j = 0; j < i * i; j++)
        for(k = 0; k < j; k++)
            sum++;
```

VI.
```
sum = 0;
for (i = 1; i < n; i++)
    for(j = 1; j < i * i; j++)
        if (j % i == 0)
            for(k = 0; k < j; k++)
                sum++;
```

(a) For each fragment, except VI, find the running time $T(n)$. For your analysis, make a table of the line, its cost, and number of times each line is executed. Type your answers. Be *exact* in your counts.

(b) Write each fragment as a function. Name your functions `foo1(), foo2()`, etc. Name your source code file `foo.c`. Use `unsigned long`s for all integers in the program. You are going to time each fragment using the same approach you did in Homework 0.

For each function, in `main()` run the function for various values of n and time the results. Print out the values using the format `n|sum|seconds`. You are going to run the functions for the following maximum values of n. Start at 1 and increment n by a power of 10. That is time the functions for $1, 10, \ldots, max\ n$.

Run the functions for the following maximum n's.

```
foo1: max n = 1000000000
foo2: max n = 100000
foo3: max n = 1000
foo4: max n = 100000
foo5: max n = 100
foo6: max n = 100
```

(c) The Appendix has sample output for this program. Capture the output to a file named `foo.out` using redirection.

(d) Now, for each function change the value of each max n by a value of 10 and run the program again. **However, do the test for one function at a time.** Name your output `foo1.out, foo2.out, etc`. Rather than using powers of 10 to increment, start the loop with the previous max n term and increment by that value, stopping at the new value. For example, for `foo3{}`, the loop is

```
for (n = 1000; n <= 10000; n += 1000)
```

and for `foo5()` the loop is

```
for (n = 100; n <= 1000; n += 100).
```

As before time the function for each value of `n` and report it in the same format `n|sum|time`. If after a hour, the code hasn't stop executing kill it with `CTRL + C`. Now copy the `stdout` to the desired text file. Leave the ^C in the output. **Do not use redirection** as when you kill the program the file will be empty. Keep all values recorded for an hour or however long it takes to finish, even if overflow occurs and the sum is wrong.

As both steps (b) and (d) involve similar code except the for loops and how you run the program, you can code this with conditional compilation flags. Copy the code for step (b), modify the for loops, and wrap the the blocks of code in conditonal flags. Your 113 book, *C Programming: A Modern Approach* has a section on conditional compilation starting at page 333.

(e) For each function, plot *time* vs n ($T(n)$ vs n). Combine the data from both steps (b) and (d). Use only valid timing data (i.e. data where overflow doesn't occur). Use a spreadsheet or any other plotting software of your choice. For each function, create a pdf of the plot. Name your plots `foo1.pdf, foo2.pdf`, etc. To see the shape of the running time more clearly, create a line plot of the data.

(f) Answer the following question. Does the shape of the plot correspond to the running time you found in step (a)? In other words, if $T(n) = 3n + 6$, is the corresponding plot linear as well? Based on your plot of function VI, what is the running time – linear, quadratic, cubic, quartic, etc – of that function? Type your answers.

Submission

Tar your source code files, your Makefile, the out files, and the pdf files into a file named
`cse122_firstname_lastname_hw2.tar.gz`
Upload the file to Canvas before the due date.

Appendix - Sample Output

Sample output for `fib_error`

```
f[0] = 0
f[1] = 1
f[2] = 1
f[3] = 2
f[4] = 3
f[5] = 5
f[6] = 8
f[7] = 13
f[8] = 21
f[9] = 34
f[10] = 55
f[11] = 89
f[12] = 144
f[13] = 233
f[14] = 377
f[15] = 610
f[16] = 987
f[17] = 1597
f[18] = 2584
f[19] = 4181
f[20] = 6765
f[21] = 10946
f[22] = 17711
f[23] = 28657
f[24] = 46368
f[25] = 75025
f[26] = 121393
f[27] = 196418
```

```
f[28] = 317811
f[29] = 514229
f[30] = 832040
f[31] = 1346269
f[32] = 2178309
f[33] = 3524578
f[34] = 5702887
f[35] = 9227465
f[36] = 14930352
f[37] = 24157817
f[38] = 39088169
f[39] = 63245986
f[40] = 102334155
f[41] = 165580141
f[42] = 267914296
f[43] = 433494437
f[44] = 701408733
f[45] = 1134903170
f[46] = 1836311903
f[47] = 2971215073
f[48] = 4807526976
f[49] = 7778742049
f[50] = 12586269025
f[51] = 20365011074
f[52] = 32951280099
f[53] = 53316291173
f[54] = 86267571272
f[55] = 139583862445
f[56] = 225851433717
f[57] = 365435296162
f[58] = 591286729879
f[59] = 956722026041
f[60] = 1548008755920
f[61] = 2504730781961
f[62] = 4052739537881
f[63] = 6557470319842
f[64] = 10610209857723
f[65] = 17167680177565
f[66] = 27777890035288
f[67] = 44945570212853
f[68] = 72723460248141
f[69] = 117669030460994
f[70] = 190392490709135
f[71] = 308061521170129
f[72] = 498454011879264
f[73] = 806515533049393
f[74] = 1304969544928657
```

```
f[75] = 2111485077978050
f[76] = 3416454622906707
f[77] = 5527939700884757
f[78] = 8944394323791464
f[79] = 14472334024676221
f[80] = 23416728348467685
f[81] = 37889062373143906
f[82] = 61305790721611591
f[83] = 99194853094755497
f[84] = 160500643816367088
f[85] = 259695496911122585
f[86] = 420196140727489673
f[87] = 679891637638612258
f[88] = 1100087778366101931
f[89] = 1779979416004714189
f[90] = 2880067194370816120
f[91] = 4660046610375530309
f[92] = 7540113804746346429
f[93] = 12200160415121876738

error in 72 term of formula Fibonacci
terms differ by 1
percentage error 0.00000000000238237%

error in 73 term of formula Fibonacci
terms differ by 2
percentage error 0.00000000000247980%

error in 74 term of formula Fibonacci
terms differ by 3
percentage error 0.00000000000229890%

error in 75 term of formula Fibonacci
terms differ by 5
percentage error 0.00000000000248640%

error in 76 term of formula Fibonacci
terms differ by 8
percentage error 0.00000000000248796%

error in 77 term of formula Fibonacci
terms differ by 14
percentage error 0.00000000000253259%

error in 78 term of formula Fibonacci
terms differ by 24
percentage error 0.00000000000268324%
```

```
error in 79 term of formula Fibonacci
terms differ by 39
percentage error 0.00000000000276389%

error in 80 term of formula Fibonacci
terms differ by 59
percentage error 0.00000000000256227%

error in 81 term of formula Fibonacci
terms differ by 102
percentage error 0.00000000000274486%

error in 82 term of formula Fibonacci
terms differ by 161
percentage error 0.00000000000260987%

error in 83 term of formula Fibonacci
terms differ by 279
percentage error 0.00000000000274208%

error in 84 term of formula Fibonacci
terms differ by 464
percentage error 0.00000000000279127%

error in 85 term of formula Fibonacci
terms differ by 743
percentage error 0.00000000000283409%

error in 86 term of formula Fibonacci
terms differ by 1207
percentage error 0.00000000000289389%

error in 87 term of formula Fibonacci
terms differ by 2014
percentage error 0.00000000000301224%

error in 88 term of formula Fibonacci
terms differ by 3157
percentage error 0.00000000000290886%

error in 89 term of formula Fibonacci
terms differ by 5171
percentage error 0.00000000000287644%

error in 90 term of formula Fibonacci
terms differ by 8584
```

```
percentage error 0.000000000000302215%

error in 91 term of formula Fibonacci
terms differ by 14523
percentage error 0.000000000000307636%

error in 92 term of formula Fibonacci
terms differ by 22595
percentage error 0.000000000000298775%

error in 93 term of formula Fibonacci
terms differ by 37118
percentage error 0.000000000000302160%
```

Sample output for `foo`.

```
foo1
1|1|0.000000
10|10|0.000000
100|100|0.000000
1000|1000|0.000000
10000|10000|0.000000
100000|100000|0.000000
1000000|1000000|0.000000
10000000|10000000|0.030000
100000000|100000000|0.180000
1000000000|1000000000|1.890000
foo2
1|1|0.000000
10|100|0.000000
100|10000|0.000000
1000|1000000|0.010000
10000|100000000|0.180000
100000|10000000000|18.390000
foo3
1|1|0.000000
10|1000|0.000000
100|1000000|0.000000
1000|1000000000|1.820000
foo4
1|0|0.000000
10|45|0.000000
100|4950|0.000000
1000|499500|0.000000
10000|49995000|0.090000
100000|4999950000|9.250000
```

```
foo5
1|0|0.000000
10|7524|0.000000
100|975002490|1.800000
foo6
1|0|0.000000
10|870|0.000000
100|12087075|0.020000
```