Homework 2 - Fibonacci numbers, 40 hexadecimal digits at a time

Michael McAlpin
Instructor - COP3402 - CS-1
Summer 2017
CS-UCF
michael.mcalpin@ucf.edu

July 1, 2017

Assignment due date: July 16, 2017

Abstract

In this programming assignment, you will implement a Fibonacci function that avoids repetitive computation by computing the sequence linearly from the bottom up: *F(0)* through *F(n)*. You will also overcome the limitations of C's 32-bit integers by storing very large integers in arrays of individual digits.

By completing this assignment, you will gain experience crafting algorithms of moderate complexity, develop a deeper understanding of integer type limitations, become acquainted with unsigned hexadecimal integers, and reinforce your understanding of dynamic memory management in C. In the end, you will have a very fast and awesome program for computing huge 40 hexadecimal digit sequences of Fibonacci numbers.

Interestingly, this problem will be limited to 40 hexadecimal digit numbers, from the outset, thru the whole program. This will mimic the performance constraints of some old cryptographic equipment (KW-26) that generated key strings based on a 2 number input to start a continuous chain of *some type of* calculations to generate long apparently random number sequences.

Attachments

big40.h, big40-main{01-04}.c, big40-main{01-04}.log, big40-main04.err

Deliverables

big40.c

(Note: Pay attention - as the filename matters!)

1 Overview

1.1 Computational Considerations for Recursive Fibonacci

We've seen in class that calculating Fibonacci numbers with the most straightforward recursive implementation of the function is prohibitively slow, as there is a lot of repetitive computation:

```
int fib(int n)
{
  //base cases: F(0) = 0, F(1) = 1
  if (n < 2)
     return n;
  //definition of Fibonacci: F(n) = F(n - 1) + F(n - 2)
  return fib(n - 1) + fib(n - 2);
}
```

This recursive function sports an exponential runtime. It is possible to achieve linear runtime by building from the base cases, $F(0) = 0$ and $F(1) = 1$, toward the desired result, $F(\mathrm{n})$. This avoids the expensive and exponentially *explosive* recursive function calls.

This assignment will emulate some aspects of hardware encryption from the 1960s, specifically the KW-26, while the KW-26 used 45 digit round-robin counters and a bit of other hardware, this assignment will use *40 hexadecimal digit* counters, with two initialization *vectors*, the *cryptoVariable* and the *hwConfigVariable*. Each of these *vectors* will be 40 hexadecimal digits. All subsequent products will be 40 hexadecimal digits. In the event of *overflow* the *overflow* product will be ignored.

Once the *cryptoVariable* and the *hwConfigVariable* have been read and created, respectively, they will be decimally added to produce a *Fibonacci* sum. All subsequent 40 hexadecimal digit integers will be the sum of the two previous 40 hexadecimal digit integers. (This ensures that the digits after $F(2)$ will be unique and the full 40 digits.) The math is shown below.

$$
\begin{aligned}
f_0 &= hwConfigVariable \\
f_1 &= cryptoVariable \\
f_2 &= f_1 + f_0 \\
&\vdots \\
f_n &= f_{n-1} + f_{n-2}
\end{aligned}
$$

Note that 40 hexadecimal digits does **not** fit into any standard C variable data type. (See *Section 7, Representing huge integers in C* for a detailed explanation on how to add large integers using a created data type.)

Careful review shows that by placing the 40 hexadecimal digit integers into an array, with the **least significant digit** in the leading digit it will be possible to add the two 40 digit numbers together, if added one digit at a time from the first element in the array to the last element in the

array. Arithmetically speaking the **most significant digit** will be in the **most significant slot in the array**.

For example, the decimal number 12,567 would be parsed one digit at a time into an array named x containing 7 in $x[0]$, 6 in $x[1]$, 5 in $x[2]$, 2 in $x[3]$, and 1 in $x[4]$. All 40 hexadecimal digits will be stored in an array using the following data structure to hold the pointer to the *malloc'ed* buffer of 40 digits.

```
typedef struct Integer40
{
  // a dynamically allocated array to hold a 40
  // digit integer, stored in reverse order
  int *digits;
} Integer40;
```

2 Attachments

2.1 Header File (big40.h)

This assignment includes a header file, `big40.h`, which contains the definition for the `Integer40` `struct`, as well as functional prototypes for all the required functions in this assignment. You should `#include` this header file from your `big40.c` source file, like so:

```
#include "big40.h"
```

2.2 Test Cases

This assignment comes with multiple sample main files `(big40-main01-04.c)`, which you can compile with your `big40.c` source file. For more information about compiling projects with multiple source files, see Section 5, "Compilation and Testing (Linux/Mac Command Line)."

2.3 Sample Output Files

Also included are a number of sample output files that show the expected results of executing your program `(big40-main01-04.log & big40-main04.err)`.

2.4 Disclaimer

The test cases included with this assignment are by no means comprehensive. Please be sure to develop your own test cases, and spend some time thinking of "edge cases" that might break each of the required functions.

3 Function Requirements

In the source file you submit, `big40.c`, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Notice that none of your functions should print anything to the screen or STDOUT.

```
Integer40 *big40Add(Integer40 *p, Integer40 *q);
```

Description: Return a pointer to a new, dynamically allocated Integer40 struct that contains the result of adding the 40 digit integers represented by p and q.

Special Notes: If a NULL pointer is passed to this function, simply return NULL. If any dynamic memory allocation functions fail within this function, also return NULL, but be careful to avoid memory leaks when you do so.

Hint: Before adding two huge integers, you will want to create an array to store the result. Remember that all integers in this problem are 40 digits long. In the event that the most significant digits (MSD) result in a carry, the carry will be ignored. For example, if the MSD of the two inputs are 9 and 7, the resultant MSD will be 6 with a carry of 1 for the MSD + 1 digit. ($9_{16} + 7_{16} = 10_{16}$, therefore 6 is the MSD and the 1 is ignored.)[1]

Returns: A pointer to the newly allocated Integer40 struct, or NULL in the special cases mentioned above.

```
Integer40 *i40Destroyer(Integer40 *p);
```

Description: Destroy any and all dynamically allocated memory associated with p. Avoid segmentation faults and memory leaks.

Returns: NULL

```
Integer40 *parseString(char *str);
```

Description: Convert a number from string format to Integer40 format. (For example function calls, see big40-main01.c.)

Special Notes: If the empty string ("") is passed to this function, treat it as a zero ("0"). If any dynamic memory allocation functions fail within this function, or if str is NULL, return NULL, be careful to avoid memory leaks when you do so. You may assume the string will only contain ASCII digits '0' through '9' and the letters 'A' thru 'F' in either upper or lower case, for a minimum of 40 digits. In the event that 40 digits are not in the input string, print an error message to STDERR and fill with leading zeroes. Also, if there are more than 40 digits in the input string use the **first** 40 digits in the string.

Returns: A pointer to the newly allocated Integer40 struct, or NULL if dynamic memory allocation fails or if the input *str* is NULL.

[1] The subscript of 16 indicate base 16. However the two expressions 10_{16} and 10_x are equivalent and used equally often.

```
Integer40 *fibBig40(int n, Integer40 *first, Integer40 *second);
```

Description: This is your Fibonacci function. Pay careful attention the F(0) is initialized with the *hwConfigVariable* and F(1) is initialized with the *cryptoVariable*. Implement an iterative solution that runs in O(n) time and returns a pointer to a Integer40 struct that contains F(n). Be sure to prevent memory leaks before returning from this function.

Space Consideration: When computing F(n) for large n, it's important to keep as few Fibonacci numbers in memory as necessary at any given time. For example, in building up to F(10000), you won't want to hold Fibonacci numbers F(0) through F(9999) in memory all at once. Find a way to have only a few Fibonacci numbers in memory at any given time over the course of a single call to fibBig40(...).

Special Notes: Remember that **n** is the second parameter passed as an input argument to the program. You may assume that **n** is a non-negative integer. If any dynamic memory allocation functions fail within this function, return NULL, but be careful to avoid memory leaks when you do so.

Returns: A pointer to an *Integer40* representing F(n), or NULL if dynamic memory allocation fails.

```
void big40Rating();
```

STDERR output: Outputs the following items to STDERR, delimited by a semicolon **";"**:

1. NID
2. A difficulty rating of how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).
3. Duration, in hours, of the time you spent on this assignment.

The first argument to this function is the pointer to the *big40RatingStruct* which is defined in the *big40.h* include file. Make sure to output those items to STDERR. Each element should be terminated or delimited by a **";"**.

```
Integer40* loadHwConfigVariable(int seed);
```

Returns: A pointer to an Integer40 array of 40 random digits. If the input variable `seed` is set, the random number generator will be seeded, otherwise not. Regardless, the 40 digits will be initialized in 10 unique groups of 4 random digits. Returns NULL if there is an error during creation or initialization of the *hwConfigVariable*.

```
Integer40* loadCryptoVariable(char *cryptoVariableFilename);
```

Returns: A pointer to an Integer40 array of 40 random digits read in from the *cryptoVariableFilename*. Returns NULL if there is an error during initialization of the *cryptoVariable* or in the file I/O.

4 Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (.c files) at the command line:

```
gcc big40.c big40-main01.c
```

By default, this will produce an executable file called `a.out` that you can run by typing, e.g.:

```
./a.out
```

If you want to name the executable something else, use:

```
gcc big40.c big40-main01.c -o b40-01.exe
```

...and then run the program using:

```
./b40-01.exe
```

Running your program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following:

```
./b40-01.exe > whatever.txt
```

This will create a file called `whatever.txt` that contains the output from your program.

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt b40-output01.txt
```

If the contents of `whatever.txt` and `b40-output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
mcalpin@eustis:~$ diff whatever.txt b40-output01.txt
mcalpin@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
mcalpin@eustis:-$ diff big40-main02.log big40-main02.bogus.log
2c2
< 19427194271942719427194271942719427194271942719427
---
> 49427194271942719427194271942719427194271942719427
mcalpin@eustis:~$ _
```

4.1 Deliverables

Submit a single source file, named big40.c, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work.

Your source file must not contain a main() function. Do not submit additional source files, and do not submit a modified big40.h header file. Your program must **compile and run on Eustis** to receive credit. Programs that do not compile will receive an automatic zero. Specifically, your program must compile without any special flags, as in:

```
gcc big40.c big40-main01.c
```

Be sure to include your name and NID as a comment at the top of your source file.

5 Grading

Scoring will be based on the following rubric:

Table 1: Grading Rubric

Percentage	Description
-100	Cannot compile on *Eustis*
- 10	Late
- 30	Cannot convert a string to the correct Integer40
- 5	Cannot create the correct Integer40 from loadCryptoVariable
- 10	Cannot create an Integer40 from loadHwConfigVariable(w/o seed)
- 5	Creates an Integer40 from loadHwConfigVariable(w/o seed), but it is not a repeating pattern
- 10	Cannot create an Integer40 from loadHwConfigVariable(w/ seed)
- 5	Creates the same Integer40 from subsequent calls to loadHwConfigVariable(w/ seed)
- 10	Cannot manage memory for Integer40s - (no partial credit)
- 20	Crashes when adding Integer40 numbers
- 10	Adds Integer40 numbers, but gets the wrong answer
- 10	Adds Integer40 numbers correctly, but cannot calculate fibBig40
- 10	Calculates fibBig40, but uses the wrong base cases
- 10	The large n case must be forced to stop
- 5	The large n case finishes, but takes longer than 5 seconds
- 10	Does not output any big40Rating data
- 5	Outputs big40Rating data, but not to stderr

Your grade will be based primarily on your program's ability to compile and produce the exact output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly.

Please note that you will not receive credit for test cases that call your Fibonacci function if that function's runtime is worse than O(n), or if your program has memory leaks that slow down execution. In grading, programs that take longer than a fraction of a second per test case (or perhaps

a whole second or two for very large test cases) will be terminated.

Your *big40.c* must **not** include a main() function. If it does, your code will fail to compile during testing, and you will receive zero credit for the assignment.

Special Restrictions: As always, you must avoid the use of global variables, mid-function variable declarations, and system calls (such as system("pause")).

6 Submission Instructions

The assignment shall be submitted via *WebCourses*.

7 Representing huge integers in C

Any linear Fibonacci function has a big problem, though, which is perhaps less obvious than the original runtime issue: when computing the sequence, we quickly exceed the limits of C's 32-bit integer representation. On most modern systems, the maximum int value in C is $2^{32} - 1$, or 2,147,483,647. [2] The first Fibonacci number to exceed that limit is F(47) = 2,971,215,073. Obviously, this will not support a 40 digit number calculation.
This problem is exacerbated by the fact that **all** the numbers used in this problem will be 40 decimal digits long. The maximum value of 10^{40} requires 163 bits. The 163 bits is derived by solving the following equation:

$$10^{50} = 49 * log_{10}10/log_{2}10 \approx 162.7744$$

Even C's 64-bit `unsigned long long int` type is only guaranteed to represent non-negative integers up to and including 18,446,744,073,709,551,615 which is $2^{64} - 1$.[3] The Fibonacci number F(93) is 12,200,160,415,121,876,738, which can be stored as an `unsigned long long int`. However, F(94) is 19,740,274,219,868,223,167, which is too big to store in any of C's extended integer data types.

To overcome this limitation, we will represent integers in this program using arrays, where each index holds a single digit of an integer.[4] For reasons that will soon become obvious, we will store integers in reverse order in these arrays. So, for example, the numbers 2,147,483,648 and 10,0087 would be represented as:

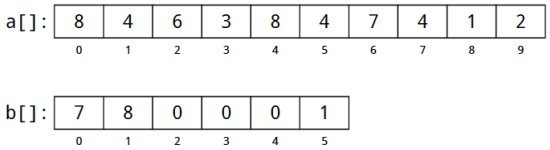

Figure 1: Two numbers stored in array - LSD first

Storing these integers in reverse order makes it *really* easy to add two of them together. The ones digits for both integers are stored at index [0] in their respective arrays, the tens digits are at index [1], the hundreds digits are at index [2], and so on. *How convenient!*

So, to add these two numbers together, we add the values at index [0] (8 + 7 = 15), throw down the 5 at index [0] in some new array where we want to store the sum, carry the 1, add it to the values at index [1] in our arrays (1 + 4 + 8 = 13), and so on:

Note that the examples shown are for small sequences of digits. For **all** numbers in this program,

[2] To see the upper limit of the `int` data type on your system, `#include <limits.h>`, then `printf("%d\n", INT_MAX);`

[3] To see the upper limit of the `unsigned long long int data` type on your system, `#include <limits.h>`, then `printf("%llu\n", ULLONG_MAX);`

[4] Yes, there is a lot of wasted space with this approach. We only need 4 bits to represent all the digits in the range 0 through 9, yet the `int` type on most modern systems is 32 bits. Thus, we're wasting 28 bits for every digit in the huge integers we want to represent! Even C's smallest data type utilizes at least one byte (8 bits), giving us at least 4 bits of unnecessary overhead.

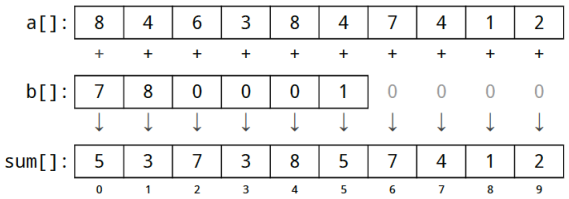

Figure 2: Calculating the sum of two numbers (LSD first)

we will use this array representation for integers containing 40 digits. The arrays will be allocated *dynamically*.