Homework 2 - KW26 - using 40 hexadecimal digits

Michael McAlpin
Instructor - COP3502 - CS-1
Fall 2017
CECS-UCF
michael.mcalpin@ucf.edu

October 13, 2017

Assignment due date: November 5, 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)*. The calculation of the *F(n)* will provide a *key* to encrypt a simple text message. 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 program for computing huge 40 hexadecimal digit sequences of Fibonacci numbers that also encrypts short text messages.

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. This 40 digit number will be used as an encryption key for a text message, read from a user specified file.

Attachments

kw26.h, kw26-m{1-6}.c, kw26-m{1-6}.log, kw26-m4.err

Deliverables

kw26.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 has 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(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 Int40
{
  // a dynamically allocated array to hold a 40
  // digit integer, stored in reverse order
  int *digits;
} Int40;
```

2 Attachments

2.1 Header File (kw26.h)

This assignment includes a header file, `kw26.h`, which contains the definition for the `Int40 struct`, as well as functional prototypes for all the required functions in this assignment. You should `#include` this header file from your `kw26.c` source file, like so:

```
#include "kw26.h"
```

2.2 Test Cases

This assignment comes with multiple sample main files `(kw26-m1-6.c)`, which you can compile with your `kw26.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 `(kw26-main1-6.log & kw26-m4.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, `kw26.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.

```
Int40 *kw26Add(Int40 *p, Int40 *q);
```

Description: Return a pointer to a new, dynamically allocated Int40 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 Int40 struct, or NULL in the special cases mentioned above.

```
Int40 *i40Destroyer(Int40 *p);
```

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

Returns: NULL

```
Int40 *parseString(char *str);
```

Description: Convert a number from string format to Int40 format. (For example function calls, see kw26-m1.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 Int40 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.

```
Int40 *fibKw26(int n, Int40 *first, Int40 *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 Int40 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 fibKw26(...).

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 *Int40* representing F(n), or NULL if dynamic memory allocation fails.

```
Int40 *encrypt(Int40 *key, Int40 *inputText);
```

Description: This is the encryption function. It will take the *inputText* pointer and `XOR` the *inputText* data with the *key* that had been generated using the *fibKw26* function. This process will produce an *encrypted* message.

Special Notes: It is interesting to note that *encrypted* data can produce the ***original plain text*** by simply `XOR'ing` the *encrypted* data with the *key*, which this assignment demonstrates can be reproduced on demand.

Returns: A pointer to an *Int40* representing the ***encrypted plain text***.

```
void kw26Rating();
```

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 *kw26RatingStruct* which is defined in the *kw26.h* include file. Make sure to output those items to STDERR. Each element should be terminated or delimited by a **";"**.

```
Int40* loadHwConfigVariable(int seed);
```

Returns: A pointer to an Int40 array of 40 digits. If the input variable `seed` is not zero, the random number generator will be seeded with the time of the *epoch* which is expressed as an integer value of the number of seconds since `01-01-1970`, otherwise the random number generator will not be used. In the event the `seed` is **zero**, the function should return 40 digits, each with the value of 1. In the event the `seed` is greater than 0, the 40 digits will be initialized in 8 groups of the same 5 random digits. For example, if the five unique random digits were 16597, the result would be those 5 digits concatenated 8 times to produce:

1659716597165971659716597165971659716597

Returns NULL if there is an error during creation or initialization of the *hwConfigVariable*.

Note: It is acceptable to use decimal values for the 5 random digits, that is, there is **no requirement** to use 5 random *hexadecimal* digits.

```
Int40* loadCryptoVariable(char *cryptoVariableFilename);
```

Returns: A pointer to an Int40 array of 40 random hexadecimal digits read in from the *cryptoVariableFilename*. Returns NULL if there is an error during initialization of the *cryptoVariable* or in the file I/O.

```
Int40* loadPlainText(char *plainTextFilename);
```

Note: Reading the input text will require *casting* the *character* or text to an integer compatible format. Therefore it is likely to have an internal function roughly equivalent to the previously define funtion, *parseString*.

Returns: A pointer to an *Int40* array of 40 characters read in from the *plainTextFilename*. Returns NULL if there is an error during conversion of the characters of the **`plain text`** or in the file I/O.

Special Cases: There are two special cases.

- In the event there are less than 40 characters in the *plainTextFilename* this function should pad with the equivalent of spaces.
- In the event there are more than 40 characters in the *plainTextFilename*, the trailing characters should be deleted.

4 Compilation and Testing (Linux/Mac Command Line)

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

```
gcc kw26.c kw26-m1.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 kw26.c kw26-m1.c -o kw26-01.exe
```

...and then run the program using:

```
./kw26-01.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:

```
./kw26-01.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 kw26-output01.txt
```

If the contents of `whatever.txt` and `kw26-output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
mcalpin@eustis:~$ diff whatever.txt kw26-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 kw26-m2.log kw26-m2.bogus.log
2c2
< 19427194271942719427194271942719427194271942719427
---
> 49427194271942719427194271942719427194271942719427
mcalpin@eustis:~$ _
```

4.1 Deliverables

Submit a single source file, named **kw26.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 kw26.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 kw26.c kw26-m1.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 Int40 (both numeric and input text)
- 5	Cannot create the correct Int40 from loadCrypto-Variable
- 10	Cannot create an Int40 from loadHwConfigVari-able(w/o seed)
- 5	Creates an Int40 from loadHwConfigVariable(w/o seed), but it is not a repeating pattern
- 10	Cannot create an Int40 from loadHwConfigVari-able(w/ seed)
- 5	Creates the same Int40 from subsequent calls to loadHwConfigVariable(w/ seed)
- 10	Cannot create a valid array of hexadecimal repre-sentations of plain text data
- 10	Does not correctly encrypt plain text with the asso-ciated F[N] keystream
- 10	Cannot manage memory for Int40s - (no partial credit)
- 20	Crashes when adding Int40 numbers
- 10	Adds Int40 numbers, but gets the wrong answer
- 10	Adds Int40y numbers correctly, but cannot calcu-late fibKw26
- 10	Calculates fibKw26, 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 kw26Rating data or outputs data without the **;** delimiter
- 5	Outputs kw26Rating 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 exe-

cution. 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 *kw26.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 hexadecimal digit number calculation.
This problem is exacerbated by the fact that **all** the numbers used in this problem will be 40 hexadecimal digits long. The maximum value of 16^{40} is $4_{bitsPerHexdigit} \times 40$ digits or 160 bits. The decimal equivalent is:

$$16^{40} = 1.4615 \times 10^{48}$$

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:

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!*

[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 hexadecimal digits in the range 0 through F, yet the `int` type on most modern systems is 32 bits.

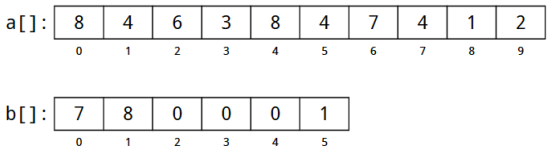

Figure 1: Two numbers stored in array - LSD first

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:

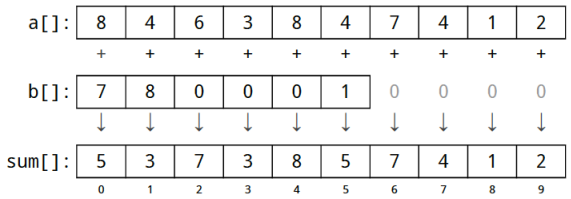

Figure 2: Calculating the sum of two numbers (LSD first)

Note that the examples shown are for small sequences of digits. For **all** numbers in this program, we will use this array representation for integers containing 40 hexadecimal digits. The arrays will be allocated *dynamically*.