Programming Assignment #2: Huge Fibonacci

Fall 2017

Due: Sunday, November 12, *before* 11:59 PM

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 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 Fibonacci numbers.

Attachments

Fibonacci.h, testcase{01-05}.c, output{01-05}.txt

Deliverables

Fibonacci.c

(Note: Capitalization of your filename matters!)
(This project has been adapted from Sean Szumlanski's projects)

1. Overview

1.1. Computational Considerations for Recursive Fibonacci

The most straightforward recursive implementation of calculating Fibonacci numbers is prohibitively slow, as there is a lot of repetitive computation. You can find below the top-down implementation of Computing n^{th} Fibonacci number:

```
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);
}
```

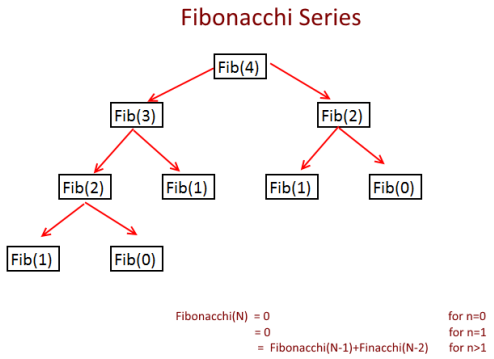


Figure 1. *Fibonacci Recursion*[1]

Now as you can see in the picture above while you are calculating Fibonacci (4), you need Fibonacci (3) and Fibonacci (2). Now for Fibonacci (3), you need Fibonacci (2) and Fibonacci (1) but you notice you have calculated Fibonacci (2) while calculating Fibonacci (4) and again calculating it. So we are solving many sub-problems again and again. The time complexity of the top-down implementation of Fibonacci is:

$$T(n) = T(n-1) + T(n-2) + 1 = 2^n = \mathbf{O(2^n)}$$

To reduce this runtime complexity, we can use dynamic programming for storing the sub-problems result, so that you don't have to calculate again. So first check if solution is already available, if yes then use it else calculate and store it for future. This approach is a bottom-up approach which can be implemented as follows:

```
int fibDP(int n)
{
   // initialization: fib(0) = 0, fib(1) = 1
   int fib[n + 1];
   fib[0] = 0;
   fib[1] = 1;

    for (i = 2; i <= n; ++i)
           fib[i] = fib[i - 1] + fib[i - 2];

   // return of Fibonacci fib[n])
   return fib[n];
}
```

[1] Source: http://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/

The time complexity and space complexity of the bottom-up implementation of Fibonacci is O(n), in which we can achieve linear runtime by building from our base cases, F(0) = 0 and F(1) = 1, toward our desired result, F(*n*). We thus avoid our expensive and exponentially **EX***p*l**Os**IV**e** recursive function calls.

Note: *The former approach is called "top-down" processing, because we work from n down toward our base cases. The latter approach is called "bottom-up" processing, because we build from our base cases up toward our desired result, F(n). In general, the process by which we eliminate repetitive recursive calls by re-ordering our computation is called "dynamic programming".*

1.2. Representing Huge Integers in C

Our 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.[1] The first Fibonacci number to exceed that limit is F(47) = 2,971,215,073.

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).[2] 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.[3] For reasons that will soon become apparent, we will store our

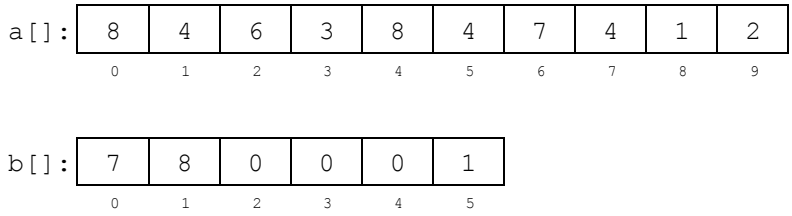
---

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

2 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);`

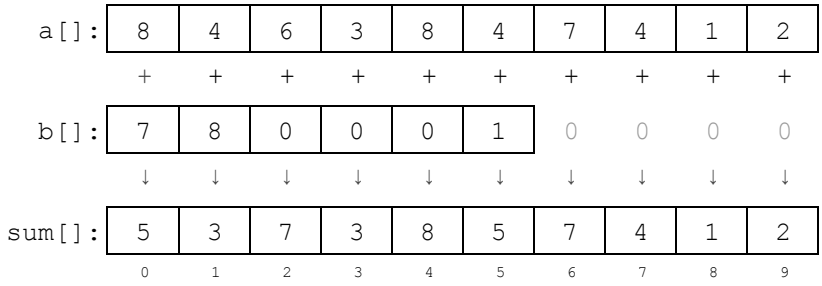
3 Admittedly, there is a lot of wasted space with this approach. We only need 4 bits to represent all the digits in the range 0 through 9, yet the `int` type on most modern systems is 32 bits. Thus, we're wasting 28 bits for every digit in the huge integers we want to represent! Even C's smallest data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

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!

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:



In this program, we will use this array representation for integers. The arrays will be allocated dynamically, and we will stuff each array inside a struct that also keeps track of the array's length:

```
typedef struct HugeInteger
{
   // a dynamically allocated array to hold the digits
   // of a huge integer, stored in reverse order
   int *digits;

   // the number of digits in the huge integer (which is
   // approximately equal to the length of the array)
   int length;
} HugeInteger;
```

1.3. Unsigned Integers and `limits.h`

There's one final curve ball you have to deal with: there are a few places where your program will utilize unsigned integers. This is no cause to panic. An unsigned integer is just an integer that can't be negative. (There's no "sign" associated with the value. It's always positive.) You declare an unsigned integer like so:

```
unsigned int n;
```

Because an `unsigned int` is typically 32 bits (like the normal `int` data type), but doesn't need to use any of those bits to signify a sign, it can eke out a higher maximum positive integer value than a normal `int`.

For at least one function in this assignment, you'll need to know what the maximum value is that you can represent using an `unsigned int` on the system where your program is running. That value is defined in your system's `limits.h` file, which you should `#include` from your `Fibonacci.c` source file, like so:

```
#include <limits.h>
```

`limits.h` defines a value called `UINT_MAX`, which is the maximum value an `unsigned int` can hold. It also defines `INT_MAX` (the maximum value an `int` can hold), `UINT_MIN`, `INT_MIN`, and many others that you might want to read up on in your spare time.

If you want to print an `unsigned int`, the correct conversion code is `%u`. For example:

```
unsigned int n = UINT_MAX;
printf("Max unsigned int value: %u\n", n);
```

Note that (`UINT_MAX + 1`) necessarily causes integer overflow, but since an `unsigned int` can't be negative, (`UINT_MAX + 1`) just wraps back around to zero. Try this out for fun:

```
unsigned int n = UINT_MAX;
printf("Max unsigned int value (+1): %u\n", n + 1);
```

Compare this, for example, to the integer overflow caused by the following:

```
int n = INT_MAX;
printf("Max int value (+1): %d\n", n + 1);
```

2. Attachments

2.1. Header File (Fibonacci.h)

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

```
#include "Fibonacci.h"
```

2.2. Test Cases

This assignment comes with multiple sample main files (`testcase{01-05}.c`), which you can compile with your `Fibonacci.c` source file. For more information about compiling projects with multiple source files, see Section 4, "Compilation and Testing (CodeBlocks)," and 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 (`output{01-05}.txt`).

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, `Fibonacci.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.

```
HugeInteger *hugeAdd(HugeInteger *p, HugeInteger *q);
```

Description: Return a pointer to a new, dynamically allocated `HugeInteger` struct that contains the result of adding the huge 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. You might find it helpful to make the array *slightly* larger than is absolutely necessary in some cases. As long as that extra overhead is bounded by a very small constant, that's okay. (In this case, the struct's `length` field should reflect the number of meaningful digits in the array, not the actual length of the array, which will necessarily be a bit larger.)

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

```
HugeInteger *hugeDestroyer(HugeInteger *p);
```

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

Returns: `NULL`

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

Description: Convert a number from string format to `HugeInteger` format. (For example function calls, see `testcase01.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`, but be careful to avoid memory leaks when you do so. You may assume the string will only contain ASCII digits '0' through '9', and that there will be no leading zeros in the string.

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

```
HugeInteger *parseInt(unsigned int n)
```

Description: Convert the unsigned integer `n` to `HugeInteger` format.

Special Notes: 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 the newly allocated `HugeInteger` struct, or `NULL` if dynamic memory allocation fails at any point.

```
unsigned int *toUnsignedInt(HugeInteger *p);
```

Description: Convert the integer represented by `p` to a dynamically allocated `unsigned int`, and return a pointer to that value. If `p` is `NULL`, simply return `NULL`. If the integer represented by `p` exceeds the maximum `unsigned int` value defined in `limits.h`, return `NULL`.

Note: The sole reason this function returns a pointer instead of an `unsigned int` is so that we can return `NULL` to signify failure in cases where `p` cannot be represented as an `unsigned int`.

Returns: A pointer to the dynamically allocated unsigned integer, or `NULL` if the value cannot be represented as an unsigned integer (including the case where `p` is `NULL`).

```
HugeInteger *fib(int n);
```

Description: This is your Fibonacci function; this is where the magic happens. Implement an iterative solution that runs in O(*nk*) time and returns a pointer to a `HugeInteger` struct that contains F(*n*). (See runtime note below.) Be sure to prevent memory leaks before returning from this function.

Runtime Consideration: In the O(*nk*) runtime restriction, *n* is the parameter passed to the function, and *k* is the number of digits in F(n). So, within this function, you can make a total of *n* calls to a function that is O(*k*) (or faster).

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 `fib()`.

Special Notes: 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 a `HugeInteger` representing F(*n*), or `NULL` if dynamic memory allocation fails.

```
double difficultyRating(void);
```

Returns: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Returns: An estimate (greater than zero) of the number of hours you spent on this assignment.

4. Compilation and Testing (CodeBlocks)

The key to getting multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, importing `Fibonacci.h`, your `Fibonacci.c` file (even if it's just an empty file so far), and any of the sample main files included with this writeup.

1. Start CodeBlocks.
2. Create a New Project (*File -> New -> Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "Fibonacci").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **each** file and choose "Add file to active project."

   – or –

2. Go to *Project -> Add Files....* Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

5. Compilation and Testing (Linux/Mac Command Line)

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

```
gcc Fibonacci.c testcase01.c
```

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

```
./a.out
```

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

```
gcc Fibonacci.c testcase01.c -o fib.exe
```

...and then run the program using:

```
./fib.exe
```

Running the 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 command, which will create a file called `whatever.txt` that contains the output from your program:

```
./fib.exe > whatever.txt
```

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

```
diff whatever.txt output01.txt
```

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

```
navid@ubuntu:~$ diff whatever.txt output01.txt
navid@ubuntu:~$ _
```

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

```
navid@ubuntu:~$ diff whatever.txt output01.txt
6c6
< F(5) = 3
---
> F(5) = 5
navid@ubuntu:~$ _
```

6. Deliverables

Submit a single source file, named `Fibonacci.c`, via Canvas. 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 `Fibonacci.h` header file. Your program must compile and run 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 Fibonacci.c testcase01.c
```

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

7. Grading

The *expected* scoring breakdown for this programming assignment is:

75%	Correct output for test cases used in grading
15%	Implementation details (manual inspection of your code)
10%	Comments and whitespace

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.

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `hugeDestroyer()` function to see that it is actually freeing up memory properly, or your `fib()` function to see that it has no memory leaks.

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(*nk*), 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. That won't be enough time for a traditional recursive implementation of the Fibonacci function to compute results for the large values of *n* that I will pass to your programs.

Your `Fibonacci.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")).