Programming Assignment #2: Fibonacci

COP 3502, Summer 2013

Due: Sunday, June 23, 11:59 PM on Webcourses2@UCF

Abstract

In this programming assignment, you will implement a recursive Fibonacci function that avoids repetitive computation by memoizing results as they are derived. 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 advanced experience working with recursion and reinforce your understanding of dynamic memory management in C. You will also learn to process command line arguments at runtime. In the end, you will have a very fast and awesome program for computing huge Fibonacci numbers.

Attachments

Fibonacci.h, HugeInteger.h, HugeInteger.c, fib01.txt, fib02.txt, fib03.txt

Deliverables

Fibonacci.c

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(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);
}
```

One way to improve the runtime of this function is to use a process called *memoization*. Essentially, we store every $F(n)$ value we ever calculate. That way, we never have to perform any repetitive computation. If we want to calculate $F(5)$, we calculate $F(4)$, $F(3)$, and $F(2)$ exactly once (and of course, $F(1)$ and $F(0)$ are already given). So, our function becomes:

```
int fib(n, memo)
{
   // base cases: F(0) = 0, F(1) = 1
   if (n < 2)
      return n;

   // check memory
   if (memo[n] != -1)
      return memo[n];

   // these calls ensure that F(n-1) and F(n-2) end up in memo
   fib(n–1, memo);
   fib(n–2, memo);

   // we're now ready to compute this result and store it in memo
   memo[n] = memo[n-1] + memo[n-2];
   return memo[n];
}
```

One problem that is immediately obvious here is that we aren't checking whether n is out of bounds when accessing `memo[n]`. We could pass the length of the `memo` array to our function as an additional parameter, but in this program, we'll wrap up the array and its length in a struct so they can both be

passed to functions easily, in one nice, neat package:

```
typedef struct Memo
{
   // a dynamically allocated array of integers
   // to store our Fibonacci numbers
   int *F;

   // the current length (i.e., capacity) of this array
   int length;
} Memo;
```

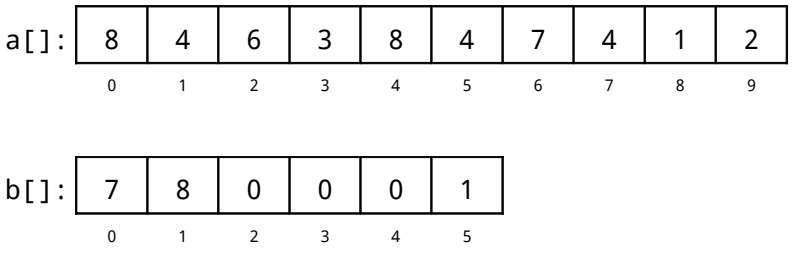
This isn't our final definition for the struct, though. There's another problem to deal with first....

1.2. Representing Huge Integers in C

A second problem with our Fibonacci function, which is perhaps less obvious, is that 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 can only be counted on 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 datatypes.

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



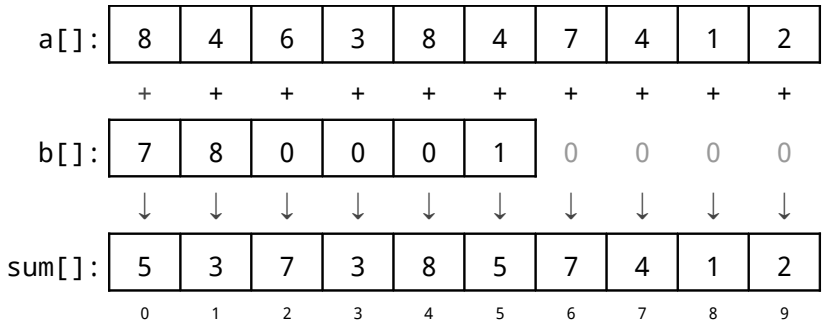
Storing these integers in reverse order makes it *really* easy to add two of them together. The ones digits for both integers are stored at index [0] in their respective arrays, the tens digits are at index [1], the hundreds digits are at index [2], and so on. How convenient!

---

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

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

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 length of the array (i.e., number of digits
   // in the huge integer)
   int length;
} HugeInteger;
```

Because integers will be stored using `HugeInteger` structs, we have to modify our `Memo` struct to hold an array of these things. The final `Memo` struct definition becomes:

```
typedef struct Memo
{
   // a dynamically allocated array of huge integers
   // to store our Fibonacci numbers
   HugeInteger *F;

   // the current length (i.e., capacity) of this array
   int length;
} Memo;
```

Get excited about your life, because in this assignment, we're not making you write the code to add two of these huge integers together. We're providing a function that will do that for you. But, you will have to write a few functions that process and deal with these structs.

1.3. Processing Command Line Arguments

There's one final curve ball for you to deal with: your program must handle command line arguments. For example, if we want to calculate $F(12)$ and $F(8)$, we will execute your program in Linux by calling:

```
./a.out 12 8
```

It's easy to get command line arguments (like the numbers 12 and 8 in this example) into your program. We just have to change the function signature for `main()` a bit. Whereas we have typically seen `main()` defined using `int main(void)`, we will now use the following function signature instead:

```
int main(int argc, char **argv)
```

Within `main()`, `argc` is now an integer representing the number of command line arguments passed to the function (including the name of the executable itself). `argv` is an array of strings that stores all those command line arguments. `argv[0]` stores the name of the program being executed. Here's a simple program to print out all the command line arguments passed to a program:

```
#include <stdio.h>
int main(int argc, char **argv)
{
   int i;
   for (i = 0; i < argc; i++)
      printf("arg[%d]: %s\n", i, argv[i]);
   return 0;
}
```

If we compiled this code into an executable file called `a.out` and ran it from the command line by typing `./a.out 12 8`, we would see the following output:

```
arg[0]: ./a.out
arg[1]: 12
arg[2]: 8
```

To convert an argument from a string to an integer, we can use the `atoi()` function from `stdlib.h`:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
   int i;
   for (i = 1; i < argc; i++)
      printf("arg[%d]: %d\n", i, atoi(argv[i]));
   return 0;
}
```

2. Attachments

2.1. Header Files

This assignment includes two header files, `Fibonacci.h` and `HugeInteger.h`, which you should `#include` in your `Fibonacci.c` source file, like so (although the order is irrelevant):

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

2.2. Source Files

This assignment comes with a source file, `HugeInteger.c`, which must be compiled with your `Fibonacci.c`. 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

We have included a number of sample output files that show the expected results of executing your program. There are no sample input files. Instead, each sample output file begins with a line that shows exactly how the function was called. That line is part of the output that your program will produce, as well.

We encourage you to develop your own test cases. Ours are by no means comprehensive.

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.

```
int main(int argc, char **argv);
```

Description: First and foremost, after your variable declarations, call the `foo()` function defined in `HugeInteger.c`. The proper function call is:

```
foo(argc, argv);
```

Then, if no command line arguments are given, terminate immediately (return 0) with no output. Otherwise:

1. Call your `createMemo()` function to dynamically create a new `Memo` struct to hold memoized Fibonacci numbers.

2. For each command line argument, *n* (not including the program name):

   a. Call your Fibonacci function, `fib(n)`, to compute *F*(*n*).

   b. Print "`F(<N>): `" to the screen (without quotes). `<N>` is just the integer argument, without the angled brackets. Notice the SPACE at the end of this string.

   c. Call `HugePrint()` on the return value of `fib(n)` in order to print the result.

3. Before exiting the program, call `destroyMemo()` to destroy your `Memo` struct.

Note: You might find it easier to implement the command line stuff last in this assignment. When you start coding and testing, you might just want to `scanf()` some integer values in a for-loop or hard-code an array of integers that you loop through to call your Fibonacci function.

Returns: 0

```
void HugePrint(HugeInteger *p);
```

Description: This function takes a pointer to a `HugeInteger` struct and prints the huge integer it represents, followed by a newline character, '`\n`'. If `p` is `NULL`, print "`(undefined)`" (without quotes), followed by a newline character, '`\n`'.

```
int HugeInit(HugeInteger *p, int n);
```

Description: This function takes a pointer to an *existing* `HugeInteger` struct (i.e., it does not dynamically allocate a `HugeInteger` struct). Set up the struct so that it represents the non-negative, single-digit integer *n*. (Hint: Since *n* is a single-digit integer, you will have to dynamically allocate `p→digits` to be an array of 1 integer.) This function will be called by the `createMemo()` function to initialize your Fibonacci base cases.

Panic: If any calls to `malloc()` fail within this function, call `panic()` (defined for you in `HugeInteger.c`) with the following string argument: "`ERROR: out of memory in HugeInit()\n`".

Returns: Return 1 if the function successfully initializes the fields of `p`. If `p` is `NULL`, or if `n` is *not* a non-negative, single-digit integer, return 0 without attempting to modify any fields within the struct.

```
Memo *createMemo(void);
```

Description: Create and initialize a `Memo` struct:

1. Dynamically allocate space for a new `Memo` struct.
2. Within the `Memo` struct, dynamically allocate an array of `INIT_MEMO_SIZE` number of `HugeInteger` structs. `INIT_MEMO_SIZE` is defined in `Fibonacci.h`.

   Note: This is an array of *actual structs*, not pointers to structs. So, you cannot set the individual elements of this array to `NULL`, and you cannot `free()` them.
3. Once the array is created, initialize `memo→length` appropriately.
4. Make two calls to `HugeInit()` to initialize `F[0]` and `F[1]` (your two Fibonacci base cases) within the `Memo` struct.
5. For all remaining `F[i]`, initialize the `digits` field to `NULL` and the `length` field to 0 to indicate that `F[i]` has not yet been memoized.

Panic: If any calls to `malloc()` fail within this function, call `panic()` (defined in `HugeInteger.c`) with the following string argument: "`ERROR: out of memory in createMemo()\n`".

Returns: A pointer to the new `Memo` struct.

`int expandMemo(Memo *memo, int n);`

Description: This function is called by your Fibonacci function, `fib()`, when it wants to memoize $F(n)$, but n exceeds the bounds of the array within `memo`. Dynamically allocate a new array of `HugeInteger` structs. The array's length should be the greater of the following values:

`(memo→length * 2 + 1)` --*or*-- `(n * 2 + 1)`

Copy the contents of the old array into the new one, and properly initialize all remaining elements in the new array. (Refer to the `createMemo()` function description for a refresher on how to initialize a `HugeInteger` struct that does not yet contain a memoized value.) Update all fields of the `memo` struct so that it contains the new array, and get rid of the old array without creating any memory leaks.

Note (1/2): You should only call `malloc()` and `free()` *once* in this function. You should be able to copy the old array's contents into the new array without calling `malloc()` inside your for-loop.

Note (2/2): This is probably the most difficult function to implement in this assignment, so you might want to get your program working for smaller values of n (i.e., n `< INIT_MEMO_LENGTH`) and come back to this one last.

Output: "`-> Expanded memo capacity to <N>.`" (Output should not include the quotes. Terminate the line with a newline character, '`\n`'. `<N>` should be the new length of the array, without the angled brackets.)

Panic: If any calls to `malloc()` fail within this function, call `panic()` (defined in `HugeInteger.c`) with the following string argument: "`ERROR: out of memory in expandMemo()\n`".

Returns: The length of the new array.

`Memo *destroyMemo(Memo *memo);`

Description: Free all dynamically allocated memory associated with or contained within `memo`. If `memo` is `NULL`, simply return `NULL`.

Returns: `NULL`

`struct HugeInteger *fib(int n, Memo *memo);`

Description: This is your recursive Fibonacci function; this is where the magic happens. Implement a recursive solution that checks whether $F(n)$ has already been memoized and, if so, returns $F(n)$ without making any recursive calls. (Recall that the base cases $F(0)$ and $F(1)$ are

memoized when we initialize `memo` in our `createMemo()` function.) There are a few more specifics to consider when writing this function:

1. When checking your base cases, be careful to ensure that you never access an array index that is out of bounds.

2. If *n* exceeds the bounds of the array in `memo`, call `expandMemo()`. For more information on what parameters to pass to `expandMemo()`, refer to its function description above.

3. In order to compute and memoize *F*(*n*), you must call the `HugeAdd()` function that is defined in `HugeInteger.c`. The function requires *F*(*n* - 1) and *F*(*n* - 2) to be memoized already. `HugeAdd()` will memoize *F*(*n*) if you call it correctly (i.e., it will store the result in `memo`). Part of the challenge and fun here is to examine the `HugeAdd()` function and figure out how to call it properly!

Returns: The address of the struct within `memo` that holds *F*(*n*). If `memo` is `NULL` or `n` is less than 0, return `NULL` without performing any recursive calls and without calling `expandMemo()`.

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`, `HugeInteger.h`, `HugeInteger.c`, and the `Fibonacci.c` file you've created (even if it's just an empty file so far).

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 **<u>each</u>** 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).

Even if you develop your code with CodeBlocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 5, "Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

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

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

```
gcc Fibonacci.c HugeInteger.c
```

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

```
./a.out 12 0 4 -2 18
```

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

```
gcc Fibonacci.c HugeInteger.c -o Fibonacci.exe
```

...and then run the program using:

```
./Fibonacci.exe 12 0 4 -2 18
```

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:

```
./Fibonacci.exe 12 0 4 -2 18 > 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 fib01.txt
```

If the contents of `whatever.txt` and `fib01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt fib01.txt
seansz@eustis:~$ _
```

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

```
seansz@eustis:~$ diff whatever.txt fib01.txt
6c6
< F(-2): undefined
---
> F(-2): (undefined)
seansz@eustis:~$ _
```

6. Grading Criteria and Submission

6.1. Deliverables

Submit a single source file, named `Fibonacci.c`, through Webcourses2@UCF. 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.

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

6.2. Additional Restrictions: No Global Variables

Please do not use global variables in this program. Doing so may result in a huge loss of points.

6.3. Grading

The expected scoring breakdown for this programming assignment is:

75%	Correct Output for Test Cases (Autograder)
15%	Implementation Details (Manually Graded)
10%	Comments and Whitespace

Your program must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

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 throughly.

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `destroyMemo()` function to see that it is actually freeing up memory properly.

Please note that you will not receive credit for test cases if you do not use memoization. The autograder will not let your program run for more than a fraction of a second per test case (or perhaps a bit longer for very large test cases), which won't be enough time for a traditional recursive implementation of the Fibonacci function to compute results for the large values of n we will pass to your program.