Programming Assignment #3: Listy Fibonacci

COP 3502, Summer 2018

Due: Sunday, July 8, *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 linked lists with nodes that contain individual digits.

By completing this assignment, you will gain experience crafting algorithms of moderate complexity, develop a deeper understanding of integer type limitations, become better acquainted with unsigned integers, and reinforce your understanding of dynamic memory management in C. You will also master the craft of linked list manipulation! In the end, you will have a very fast and awesome program for computing huge Fibonacci numbers.

Deliverables

ListyFib.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

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) for n > 1
   return fib(n – 1) + fib(n – 2);
}
```

This recursive function sports an exponential runtime. We saw in class that 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**O*s***IV**e** recursive function calls.

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," and is a topic we will explore in more depth in COP 3503 (Computer Science II).

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^{31}-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.

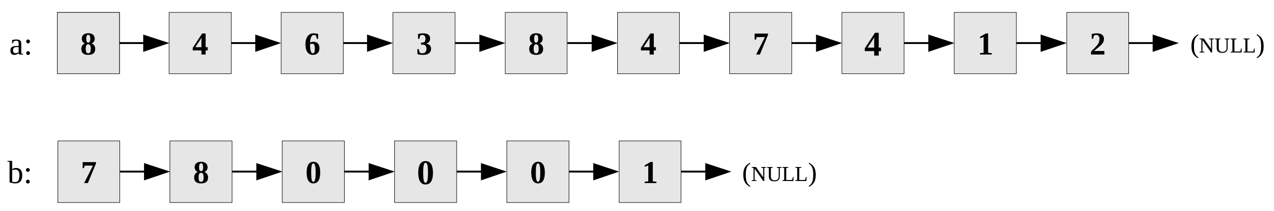
To overcome this limitation, we will represent integers in this program using linked lists, where each node holds a single digit of an integer.[3] For reasons that will soon become apparent, we will store our

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);`

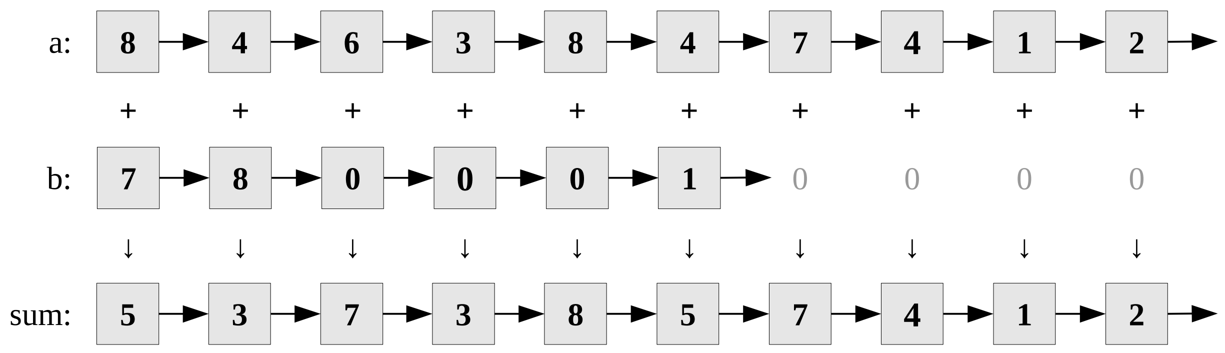
3 Admittedly, there is a lot of wasted space with this approach. We only need 4 bits to represent all the digits in the range 0 through 9, yet the `int` type on most modern systems is 32 bits. Thus, we're wasting 28 bits for every digit in the huge integers we want to represent (not to mention the space taken up by all those nodes' `next` pointers)! 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 linked lists. So, for example, the numbers 2,147,483,648 and 100,087 would be represented as:



Storing these integers in reverse order makes it *much* easier to add two of them together than if we stored them the other way around. The ones digit for each integer is stored in the first node in its respective linked list, the tens digit is stored in the second node, the hundreds digit is stored in the third node, and so on. How convenient!

So, to add these two numbers together, we add the values in the first nodes (8 + 7 = 15), throw down the 5 in the first node of some new linked list where we want to store the sum, carry the 1, add it to the values in the second nods of our linked lists (1 + 4 + 8 = 13), and so on:



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

```
typedef struct ListyInt
{
   // the head of a linked list that holds the digits
   // of an integer, stored in reverse order
   Node *head;

   // the number of digits in the integer (which is
   // equal to the number of nodes in the list)
   int length;
} ListyInt;
```

This struct is defined in ListyFib.h. There is, of course, a corresponding node struct defined in ListyFib.h, as well.

1.3. Header File (ListyFib.h)

This assignment includes a header file, `ListyFib.h`, which contains the definition for the `ListyInt` and `Node` structs, as well as functional prototypes for all the required functions in this assignment. You **must** `#include` this header file from your `ListyFib.c` source file, like so:

```
#include "ListyFib.h"
```

1.4. 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 non-negative.) As we've seen in class, 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 `ListyFib.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:[4]

```
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);
```

4 Here, "fun" is a relative term.

2. Function Requirements

In the source file you submit, `ListyFib.c`, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Note that none of your functions should print to the screen, and the file you submit should **not** have a `main()` function.

```
ListyInt *listyAdd(ListyInt *p, ListyInt *q);
```

Description: Return a pointer to a new, dynamically allocated `ListyInt` struct that contains the result of adding the 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.

Another Special Note: If an argument passed to this function is non-`NULL`, you can assume that the `head` pointer inside that struct is also non-`NULL` and that the `length` is greater than zero. You may also assume that any non-`NULL` `ListyInt` struct passed to the functions in this assignment will be well-formed. For example, if the `length` field of a `ListyInt` is set to 3, the list will have exactly three nodes, and the last node's `next` pointer will be set to `NULL`.

Runtime Restriction: The runtime of this function must be no worse than O(m + n), where m is the length of `p`, and n is the length of `q`.

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

```
ListyInt *destroyListyInt(ListyInt *listy);
```

Description: Destroy any and all dynamically allocated memory associated with `listy` in O(n) time (where n is the length of the list). Avoid segmentation faults and memory leaks.

Returns: `NULL`

```
ListyInt *stringToListyInt(char *str);
```

Description: Convert a number from string format to `ListyInt` 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 `ListyInt` struct, or `NULL` if dynamic memory allocation fails or if `str` is `NULL`.

`char *listyIntToString(ListyInt *listy);`

Description: Convert the integer represented by `listy` to a dynamically allocated string, and return a pointer to that string (i.e., return the base address of the `char` array). Be sure to properly terminate the string with a null sentinel ('\0'). If `listy` is `NULL`, or if any calls to `malloc()` fail, simply return `NULL`.

Returns: A pointer to the dynamically allocated string, or `NULL` if dynamic memory allocation fails at any point or if `listy` is `NULL`.

`ListyInt *uintToListyInt(unsigned int n);`

Description: Convert the unsigned integer `n` to `ListyInt` 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 `ListyInt` struct, or `NULL` if dynamic memory allocation fails at any point.

`unsigned int *listyIntToUint(ListyInt *listy);`

Description: Convert the integer represented by `listy` to a dynamically allocated `unsigned int`, and return a pointer to that value. If `listy` is `NULL`, simply return `NULL`. If the integer represented by `listy` 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 `listy` 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 `listy` is `NULL`).

`void plusPlus(ListyInt *listy);`

Description: Increment the value held in `listy` by one. If `listy` is `NULL`, simply return. You don't necessarily have to use this function anywhere else in your program. It's just here as an additional exercise.

Special Note: If any dynamic memory allocation functions fail within this function, simply leave the value in `listy` unmodified. (Such a failure is unlikely anyway, and will not be explicitly tested for this particular function.)

Returns: Nothing. This is a void function.

`ListyInt *fib(unsigned 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 `ListyInt` 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 O(*n*) number of calls to any 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: Notice that *n* will always be 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 `ListyInt` 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: A reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

3. Test Cases and the test-all.sh Script

We've included multiple test cases with this assignment to show some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, `test-all.sh`, that will compile and run all test cases for you. **Super Important:** Using the `test-all.sh` script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting. However, as you know, these test cases are not comprehensive. You should create additional test cases to test your code thoroughly.

You can run the test-all script on Eustis by placing it in a directory with `ListyFib.c`, `ListyFib.h`, the `sample_output` directory, and all the test case files, and then typing:

```
bash test-all.sh
```

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

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

```
gcc ListyFib.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 ListyFib.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 sample_output/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:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.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 sample_output/output01.txt
6c6
< F(5) = 3
---
> F(5) = 5
seansz@eustis:~$ _
```

5. Deliverables

Submit a single source file, named `ListyFib.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 `ListyFib.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 ListyFib.c testcase01.c
```

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

6. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

70%	Correct output for test cases used in grading
10%	No memory leaks (passes `valgrind` test cases)
10%	Implementation details (manual inspection of your code)
10%	Comments and whitespace

Important Restrictions: Do not include a `main()` function in your code; do not use global variables, `goto` statements, or system calls (e.g., `system("pause")`); and do not read or write to any files.

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 `listyAdd()` function to see that it handles `malloc()` failures properly.

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 we will pass to your programs.

Start early. Work hard. Good luck!