# Programming Assignment #3: ListyFib

**COP 3502, Fall 2020**

**Due:** Sunday, October 18, *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.

**Important note:** In your assignments, you can use any code I've posted in Webcourses, as long as you leave a comment saying where the code came from. However, you cannot use code posted by other professors, and you should never incorporate or refer to code from online resources or from other individuals. Of course, you will learn more if you try to implement everything from scratch without copying any of my code from Webcourses.

**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 **EXplOsIVe** 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] 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.

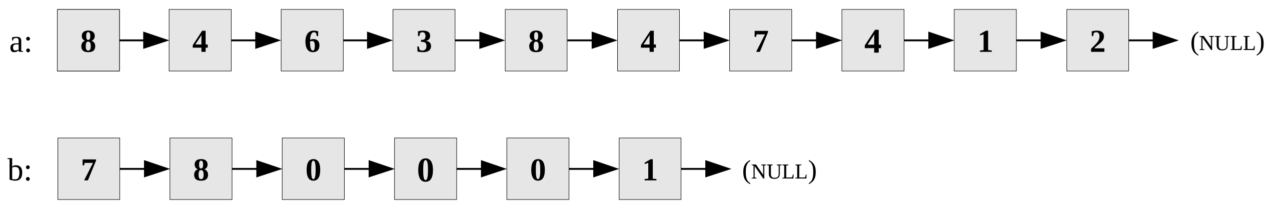
To overcome this limitation, we will represent integers in this program using linked lists, where each node holds a single digit of an integer.[3] For reasons that will soon become apparent, we will store our integers in *reverse*

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

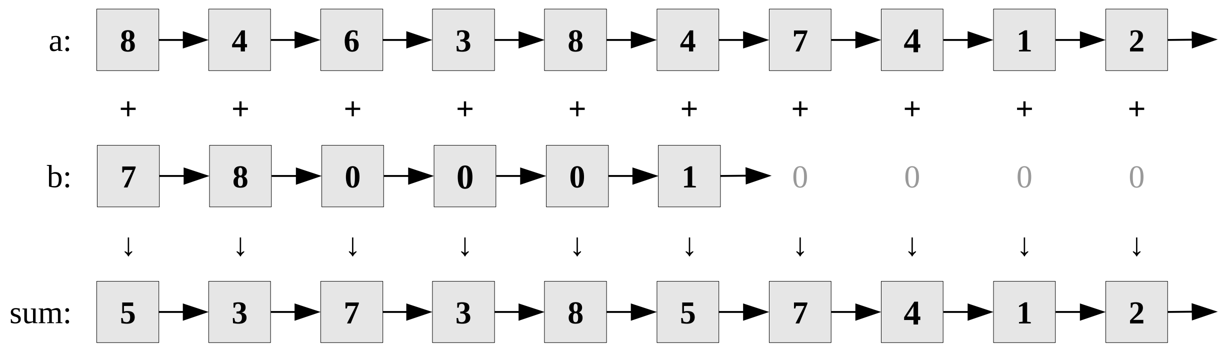
3 Admittedly, there is a lot of wasted space with this approach. We only need 4 bits to represent all the digits in the range 0 through 9, yet the *int* type on most modern systems is 32 bits. Thus, we're wasting 28 bits for every digit in the huge integers we want to represent (not to mention the space 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.

*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 nodes 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. ListyFib.h (*Super Important!*)

For your linked lists, you must use the *ListyInt* and *Node* structs we have specified in *ListyFib.h* without any modifications. Do **not** add those struct definitions to your code, though. Instead, you **must** *#include* the header file that contains those struct definitions by adding the following line of code to your *ListyFib.c* source file:

```
#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 in O(*k*) time (where *k* is the length of *str*). (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.

*Continued on the following page...*

```
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. The runtime for this function must not exceed O(*n*) (where *n* is the length of the list).

**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. If any dynamic memory allocation functions fail within this function, return NULL, but be careful to avoid memory leaks when you do so. Your runtime for this function cannot exceed O(*k*), where *k* is the number of digits in *n*.

**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 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);
```

**Description:** 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);
```

**Description:** Returns a reasonable and realistic estimate (greater than zero) of the number of hours you spent on this assignment.

## 3. Running All Test Cases on Eustis (*test-all.sh*)

The test cases included with this assignment are designed to show you 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.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *ListyFib.c*, *ListyFib.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm is fairly straightforward, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*ListyFib.c*, *ListyFib.h*, all the test case files, and the *sample_output* folder).

2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

   ```
   scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   **Warning:** Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the ***entire contents*** of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

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

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd ListyProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

**Warning:** When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type (`cd project\ 3`), or the entire name needs to be wrapped in double quotes.

## 4. Running the Provided Test Cases Individually

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. Here's how to do that:

1. Place all the test case files released with this assignment in one folder, along with your *ListyFib.c* file.

2. At the command line, *cd* to the directory with all your files for this assignment, and compile your source file with one of our test cases (such as *testcase01.c*) like so:

   ```
   gcc ListyFib.c testcase01.c
   ```

3. To run your program and redirect the output to *output.txt*, execute the following command:

   ```
   ./a.out > output.txt
   ```

4. Use *diff* to compare your output to the expected (correct) output for the program:

   ```
   diff output.txt sample_output/testcase01-output.txt
   ```

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

   ```
   seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
   6c6
   < F(5) = 3
   ---
   > F(5) = 5
   seansz@eustis:~$ _
   ```

If the contents of *output.txt* and *testcase01-output.txt* are exactly the same, *diff* won't have any output:

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
seansz@eustis:~$ _
```

***Super Important:*** *Remember, 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.*

## 5. Testing for Memory Leaks with Valgrind

Part of the credit for this assignment will be awarded based on your ability to implement the program without any memory leaks. To test for memory leaks, you can use a program called *valgrind*, which is installed on Eustis.

Valgrind will **not** guarantee that your code is completely free of memory leaks. It will only detect whether any memory leaks occur when you run your program. So, if you have a function called *foo()* that has a nasty memory leak, but you run your program in such a way that *foo()* never gets called, *valgrind* won't be able to find that potential memory leak.

The *test-all.sh* script will automatically run your program through all test cases and use *valgrind* to check whether any of them result in memory leaks. If you want to run *valgrind* manually, simply compile your program with the *-g* flag, and then run it through *valgrind*, like so:

```
gcc ListyFib.c testcase01.c -g
valgrind --leak-check=yes ./a.out
```

In the output of *valgrind*, the magic phrase you're looking for to indicate that no memory leaks were detected is:

```
All heap blocks were freed -- no leaks are possible
```

For more information about *valgrind*'s output, see: http://valgrind.org/docs/manual/quick-start.html

## 6. Special Restrictions (*Super Important!*)

You must abide by the following restrictions in the *ListyFib.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

★ Do not read or write to files (using, e.g., C's *fopen()*, *fprintf()*, or *fscanf()* functions). Also, please do not use *scanf()* to read input from the keyboard.

★ Do not declare new variables part way through a function. All variable declarations should occur at the *top* of a function, and all variables must be declared inside your functions or declared as function parameters.

★ Do not use *goto* statements in your code.

★ Do not make calls to C’s *system()* function.

★ Do not write malicious code, including code that attempts to open files it shouldn’t be opening, whether for reading or writing. (I would hope this would go without saying.)

★ No crazy shenanigans.

## 7. Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

★ Any time you open a curly brace, that curly brace should start on a new line.

★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.

★ Be consistent with the amount of indentation you’re using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you’re using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.

★ Please avoid block-style comments: /* *comment* */

★ Instead, please use inline-style comments: // *comment*

★ Always include a space after the “//” in your comments: “// *comment*” instead of “//*comment*”

★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *<u>above</u>* your *#include* statements.

★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *<u>above</u>* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.

★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.

★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.

★ When defining a function that doesn’t take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.

★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(int argc, char **argv)* instead of *int main (int argc, char **argv)*. Similarly, use *printf("...")* instead of *printf ("...")*.

- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if( condition )*.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do <u>not</u> have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ Use meaningful variable names that convey the purpose of your variables. It's fine to use single-letter variable names for short functions (e.g., a simple *max* function, such as *int max(int a, int b)*, where it would be silly to try to come up with more meaningful variable names for those two input parameters), for control variables in your *for* loops (where *i*, *j*, and *k* are common variable name choices), or for sizes and lengths of certain inputs (e.g., using *n* for the length of an array). Otherwise, please try to use variable names that convey the intended use of your variables. Names like *cheeseburger* and *pizza* are not good choices for this particular program.

## 8. 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. Be sure to include your name and NID in a header comment at the top of your source file. Also, don't forget to *#include "ListyFib.h"* in your source code (with correct capitalization).

Do not submit additional source files, do not submit a modified *ListyFib.h* header file, and <u>**do not**</u> include a *main()* function in your *ListyFib.c* source file. Your source file must work with the *test-all.sh* script, and it must also compile on Eustis in both of the following ways:

```
gcc -c ListyFib.c
gcc ListyFib.c testcase01.c
```

## 9. Grading

***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

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

| | |
|---|---|
| 60% | Passes test cases with 100% correct output formatting. |
| 20% | Passes *valgrind* test cases (no memory leaks). |

| | |
|---|---|
| 10% | Implementation details and adherence to the special restrictions imposed on this assignment. This will likely involve some manual inspection of your code. |
| 10% | Follows all style restrictions; has adequate comments and whitespace; source file is named correctly and includes your name and NID (*not* your UCF ID) in a header comment. We will likely impose huge penalties for small deviations from style restrictions because we really want you to develop good style habits in this class. |

***Note!*** Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely 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. Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Additional points may 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!*