Adam Jean-Laurent

CSC 412 Prog02 Report

C Programs:

Since the 3 C programs have fundamentally similar structure, I'll summarize in general how I implemented the solution, and touch on specifics of part 2 and the extra credit.

Part 1:

Steps:

1. Validate User Input
2. Calculate Fibonacci Sequence
3. Print Fibonacci Sequence

Validate Input:

- I made sure to break all functionality that is eligible for its own container in it's own function. This way I can make my code easier to read, debug, and understand.
- The first step in the program is to validate the user input. I needed to ensure that the user had the correct arguments. So the first thing I do is validate the input in the main function.
- I have a function `int validateUserInput(char* arguments[], int numOfArguments);`
- This function takes in argv and argc, and checks that argv == 4 and calls the function `int allStringsAreNumbers(char* str1, char* str2, char* str3);`
- Which takes in str1(F1), str2(F2), and str3(n), it verifies that all the strings are valid positive integers, by using atoi to convert them, and checks that F2 > F1.
- There are two error codes that can come out of verifying the input, -1 which means the incorrect number of arguments were passed, and -2 which means F2 <= F1.

Calculate Fibonacci Sequence:

- After verifying the user input, I knew it was safe at that point to convert the arguments to integers and perform calculation.
- I pass the parsed parameters to a function `void printFibonacciSequence(int sequenceLength, int F1, int F2);`
- The purpose of this function is to get the calculated fibonacci sequence and print it to the terminal in the output specified in the PDF directions.
- To get the calculated Fibonacci sequence, a call to `int* getFibonacciSequence(int sequenceLength, int F1, int F2);` Is Made.
- The purpose of getFibonaciSequence is to return an array of the fibonacci sequence for the given parameters. I allocated an array of the size of sequenceLength with calloc, and used the iterative top-down approach to the fibonacci sequence. This is a dynamic programming algorithm and uses memoization to have a constant space complexity, which is more efficient than the N stack frames needed in the recursive implementation that is taught in CSC 211 / 212.

Print Fibonacci Sequence:

- All that happens after the sequence is calculated is to loop through the sequence, printing the values, and freeing the dynamically allocated array that holds the sequence at the end.

Test Performance:

- Overall the program passed all of the tests I gave it, invalid number of arguments, F1 >= F2, zeroes, and negative numbers. The only abusable behavior is calculating giant numbers which will obviously cause overflow. To combat this I would probably use a long long instead of int datatype to hold calculated values, so that overflow happens less, but regardless overflow could happen if the user provided very large input numbers, because the assignment didn't specify and I feel I didn't have enough time I didn't put a

safeguard for this, but I would do something where you couldn't enter large enough numbers to overflow, or add some defined behavior in the case of an overflow.

-

Part 2:

Steps:

1. Validate User Input
2. Calculate Fibonacci Sequences
3. Filter Duplicates
4. Print Fibonacci Sequences and distinct values

Validate Input:

- Same process as in part 1, expect `Bool validateUserInput(char* userInput[], int length);` Now validates a list of triplets rather than just one.
- It does this by looping through 3 items at a time and calling `Bool validateTriplet(char* F1, char* F2, char*n);` For each triplet. Which does exactly what allStringsAreNumbers did in part 1. It validates that the triplet contains positive integers and F1 < F2.
- Here there is only 1 error code, I'm using Bools this time

```
-
- typedef enum Bool {
-    FALSE,  //  0
-    TRUE    //  1
- } Bool;
```

- FALSE = error
- TRUE = no error

Calculate Fibonacci Sequences:

- After the input is validated, in the main program I am converting each argument into an integer and storing it in an array.
- I pass this parsed array to `void printFibonacciSequences(int* triplets, int length);`
- This function does quite a few things.
- Firstly, it calculates each triplet's fibonacci sequence

Filter Duplicates:

- After a fibonacci sequence is calculated and we have an array of the values, we check each value to see if we have seen it before.
- Distinct values are held in an array called oversize, whose size is the sum of all Ns in each triplet. It is this size, because in the worst case, every digit in every sequence is unique so we store every digit from every sequence.
- To check if we have seen an item before, we loop through the fibonacci sequence, and for each item, loop through oversize to check if it exists there.
- If it does, it's a duplicate and ignored.
- If it does not, it is distinct and we add it to oversize

```
// print fibonacci sequence for each triplet
    for(i = 0; i < length; i+=3) {
        int F1 = triplets[i];
        int F2 = triplets[i+1];
        int n = triplets[i+2];
        int* fibSeq = printFibonacciSequence(n, F1, F2);

        int j;
        int k;

        // oversize holds the distinct values
```

```
        // so, one we compute a fibonacci sequence, we check if each value
    occurs already
        // in oversize, if it doesn't, that means it is distinct, and we can add
    it to oversize
        for(j = 0; j < n; j++) {
            Bool found = FALSE;

            for(k = 0; (k < counter) && !found; k++) {
                if(fibSeq[j] == oversize[k]) {
                    // value is not distinct
                    found = TRUE;
                }
            }

            if(!found) {
                // value is distinct
                oversize[counter++] = fibSeq[j];
            }
        }
        // free memory allocated for fibonacci sequence
        free(fibSeq);
    }
```

Print Sequences And Distinct Values:

- The sequences are printed as they are computed, with repeated calls to `int* printFibonacciSequence(int sequenceLength, int F1, int F2);`
- After computing and printing all sequences, we will have oversize, (see code above) filled with the distinct values, so, we just print them.

Extra Credit:

- I'll be brief with this one, everything here is the same as Part2, except at the end I am sorting the oversize array of distinct elements using qsort with a compare function that specifies to sort as integers.

Bash Script

Steps:

1. Validate User Input
2. Check last character
3. Loop through folder
4. Print found files

Validate User Input:

```
# Check that the program was given exactly two arguments
if [ $# -gt 2 ] || [ $# -lt 2 ] ; then
   echo "usage: ./script02.sh <path> <extension>"
   exit
fi
```

Here I'm checking if the number of arguments is 2.

Check Last Character / Loop Through Folder:

```

# last character of the folder path given
LastChar="${1: -1}"

# Script supports paths ending in / and those that don't
# If the path does not end in / then when looping through the directory, add /
if [ "$LastChar" != "/" ]
then
   # Path does not end in / , add / to loop statement
   for FILE in $"$1"/*."$2" ; do
       # parse filename to remove full path
       parsedFileName=$(basename -- $FILE)
```

```
        # if valid filename, add to array of files
        if [ $parsedFileName != "*.$2" ]
        then
            FILES+=( "$parsedFileName" )
        fi
    done

else
    # path does end in / then loop through it
    for FILE in $"$1"*."$2" ; do
        # parse filename to remove full path
        parsedFileName=$(basename -- $FILE)
        # if valid filename, add to array of files
        if [ $parsedFileName != "*.$2" ]
        then
            FILES+=( "$parsedFileName" )
        fi
    done
fi
```

- Here, LastChar is is the last character in the path string
- Because this program supports paths end in / and those that don't, there is a special case for that, in which if it doesn't have one we append it
- Here I am looping through a folder, and getting the files with the given extension
- Basename simplifies the file name and removes the full path

Print found files

- After doing this operation I loop through the array of files and print them according to the number of files.

```
-
-   echo "Lookng for files with extension .${2} in folder"
-   echo -e ' \t ' $1:
-
-   # holds length of files array
```

```
LEN=${#FILES[@]}

if [ $LEN -lt 1 ]
   # no files found
then
   echo "no file found."

elif [ $LEN -lt 2 ]
   # one file found
then
   echo "1 file found:"

else
   # multiple files found
   echo $LEN "files found:"
fi

# if 1 or multiple files found, loop through FILES array, and print each file
name
for FILE in ${FILES[@]} ; do
   echo -e ' \t ' $FILE
done
```

Test Performance:

- The only type of errors I found where If I used a relative path rather than an absolute path. So anything with ../ etc. give weird behaviour.
- In the code, you'll notice after checking the last character I enter one of two loops depending if the last character of the path is / or not. The loops are exactly the same except in one I append / to the path. This is pretty sloppy, I should have only one loop, ideally I would want to hold the path given in a variable, and edit that if I needed to. But

when trying to store a path in a variable, for example, if I called ./script02.sh /path/example txt

- If I tried to do Path=$1, the script would echo, something like “directory does not exist”, I couldn’t find a workaround, so that is the reason for the slight sloppiness there.