CS132 Computer Organization and Architecture Coursework 2 Report

Marcel Afunyah

January 2021

Contents

Part 1: Sum of Digits Function

Briefing

This question was very challenging. I was not able to complete the question with a good solution. This is wholly due to the fact that I completely underestimated the question because of its length and its proportion of marks. I spent more time on part 2 which seemed more dense, and this was to my own demise.

At the point where I noticed the nature of the question, it was already too late to come up with any sensible solution. For the most part, I used up most of the time analysing the patterns of different sequences in a spreadsheet. However, I could not formulate anything of good value.

Therefore, I will outline some of my methods and some basic solutions that I came up with which will probably not meet the standards.

The problem I had with analysing patterns is that there were some changes which did not fit the trend and could not accounted for. As n gets larger, the trends change in unpredictable ways.

I found it very difficult to describe the patterns that I observed. After all the painstaking amount of effort I put into analysing thousands of numbers, I have nothing worthwhile to show. Without a meaningful conclusion, the numbers are useless, and the columns do not fit on the page.

Basic Approach

The most basic approach to find a solution for $f(n)$ was to continually add 9. From the example provided in the question for $n = 5$, it can be deduced that the constant addition of 9 will provide a sequence of terms with a sub-sequence which satisfies the conditions.

The sequence would need to be filtered such that only numbers whose digits add up to n and numbers with no zeros are chosen. Simple loops and conditionals can handle this.

The problem with this method is that it is very slow when $n > 10$.

For example, for n = 20, it takes quite a while to jump from 1211111111111111111 to 1111111111111111111 by just adding 9. This is just for n = 20, which is quite small.

Finding The First and Last Terms

For any integer n, there is a sequence of integers which satisfy the conditions. The sequence is such that only integers whose digits add up to n and contain no zero are chosen.

The first term is important, especially for large n, because it provides the starting point for finding subsequent terms.

By using the basic approach, I was able to obtain the sequence of integers
for $0 \leq n \geq 13$. By carefully analysing the patterns, I formulated how to obtain the first term of any sequence.

Let the first term be a.

If $n < 10$ then $a = n$. Else,

$$a = \text{“}(\sum n_d) mod 9\text{”} + \left\lfloor \frac{n}{9} \right\rfloor * \text{“9”}$$

The expressions in quotes are chars. What this means is:
(Sum of digits of n)mod9 +concat+ ‘9’ written floor(n/9) times.
For example, for n = 29,

- Sum of digits of n = 2 + 9 = 11
- 11 mod 9 = 2
- floor(n/9) = floor(29/9) = floor(3.22) = 3
- So, a = 2 +concat+ ‘9’ * 3
- a = 2999

The first term of the sequence for 29 is 2999.

The last term of the sequence is given by

$$l = \text{“1”} * n$$

For example, for n = 6, l = 111111

Calculating the first and last terms of a sequence gives an idea of how to handle the problem.

Attempt At A General Formula

From obtaining the solutions that the limited basic approach produced, I tried to find a general formula. I ended up with $\frac{11^n}{9}$. This always lingered the true value but it was never accurate.

Dividing all terms (of the actual sums) by 9 gives a common difference which converges, with a very low rate of change, at 11. The first term is 0.111111 or $\frac{1}{9}$.

This method is not sufficient since it does not provide the right answers. The problem is the presence of minuscule decimal numbers, so it was not possible to get exact values.

String Manipulation

This was the technique that I thought would have been a breakthrough in terms of range.

Instead of storing ints and long doubles, strings would be stored. So the integer 3989 would be represented as"3989" (as an array of chars).

I tried to deduce that the range would have been greatly increased since I could store, say, a 5000 digit integer as a char array (char myInt[5000]);

Addition would have been the only necessary operation to be carried out. So converting, a char at a time, to an int and performing addition would have been feasible.

The only problem was that finding the right pattern was difficult. It was very easy to miss just one number, in about 4000 numbers.

Iterative Addition

Another technique that I tried using was "iterative adition". For example, take n = 29. The first few terms of the solution sequence would be 2999, 3899, 3989, 3998, 4799, 4889 ...
The first term can be obtained using the formula. A trend that I noticed was the addition cycle of 900,990 and 999.
Since floor(29/9) is 3, there will be 3 place values for the "9 multipliers".
However this is not also straightforward, as there are irregularities in the trend.

Another technique I attempted was a "recursion tree". Since the sequences of the first 9 numbers are easy to obtain, the terms of other sequences can be expressed as these base cases.
Too many deep branches would be created, decreasing efficiency exponentially after n = 9.

Coding

Most of my initial tests were done in python, because of how easy it is to work with strings, in addition to the time limit.

My C code will probably be meaningless to another person, as everything was still in the testing phase.

Conclusion

I fully admit that this question was poorly done, I undermined its significance and left it until it was too late. I would also say that another pitfall is that the analysis of all the data I used is very abstract and difficult to explain.

I have include C files and spreadsheets in the submission, but I do not expect that they should be checked. You may do so in your interest.

Part 2: Command-Line Editor

Briefing

In part 2 of the coursework, the objective was to implement a command-line editor which is capable of carrying out file operations and line operations in files.

The objective was simple and relatively straightforward. With this in mind, most of the documentation for this part will be code explanations and justification of my methods. However, there is also a function guide to help the reader understand certain aspects of some functions which may come across as slightly overwhelming.
There is also a good amount of comments in the code which give high-level explanations.

Line Management

One crucial aspect of the question and its solution is the understanding and outlining of what a line is. This might seem insignificant but I believe it is a fundamental part of tackling this problem.
The concern arises from deciding on what is defined as a line. I will be using two examples, Visual Studio Code and Microsoft Word.

In MS Word, a line is defined by the paper size. When an sentence reaches the boundary, a new line is created. For example, an A4 line is different from a letter line.
However, in VS code, a new line is defined by the presence of a newline character. The text area of the editor itself is controlled by the size of the window. There is no set standard such as A4 or letter size.
A single line in VS code can even be the same as 10 lines in MS Word. As long as there is no newline character, the boundary of a line has not been reached.

This characteristic of VS code can be noticed at the left of the editor, where the line numbers are displayed. A single line number can have sentences spanning many rows.

Therefore, a line in VS code is a set of rows with the boundary being the newline character. In MS Word, a line is a single row with the boundary being the page width.

In my implementation of the command-line editor, my approach to dealing with lines is more based on how VS code handles it. I prefer this method because there really is not any idea or concern over page sizes in the terminal. It would be simpler and more efficient to base line constraints on a newline character.

That being said, there is a variable in the code which influences how many characters are displayed in a single row in the terminal. It does not change line numbers though, the

definition remains the same. It is mostly for aesthetic purposes, as the editor does not gracefully shift rows downwards if the window size of the terminal were to be altered. My code does not handle the dynamic nature of window sizes, but it accomplishes the main task.

The bufferSize variable influences the amount of characters shown on a single row. fgets normally uses the bufferSize variable. It does not mean that it is specifies the amount of characters on a line. There are certain conditions such as breaking a long word across lines. For more information see the sendToDisplay function in the code or in the Functions section.

The Working Environment

This section informs the reader on how the program interacts with files and directories.

The program allows the user to navigate directories and carry out usual file operations provided. Directory navigation is included just to save the user from the inconvenience of copying the program files to other directories for use, for testing purposes.

I would also not recommend that the user uses the program to edit any files where line enumeration and whitespaces are important. For example, it might might not be a good idea to edit Python files in the editor.

Compiling the program: gcc -o filemanager filemanager.c

The Change Log

Changes are logged in a file whose name is given in the code. It contains a very random token so it should not somehow exist on the system prior to the first run of the program (hopefully).

The format of the log file is:
'file1' log:
+ Operation
+ Operation
- Operation

'file2' log:
+ Operation
- Operation
- Operation

The operation lines also contain a timestamp. A timestamp is useful because it gives more information about the files with respect to showing which sequences of command were per-

formed in the same timeframe and so on.
The number of lines after the operation was performed is also included.

A log file will be created in any directory that the program interacts with, in terms for altering files.

The log file is updated by calling the updateLog function:
void updateLog(const char *filename, int fileDeleted, const char *logStr)

Each operation makes a call to the updateLog function after completion. A specific description of the operation performed is passed as the logStr.
In various functions, the logStr might be formatted to contain details such as line numbers. For example, deleteLineRange, logStr is formatted to contain the range of lines deleted.

The fileDeleted parameter is set to 0 unless the function is called from the deleteLine function. In that case, the file does not exist so there is no need to attempt calling countLines. The total number of lines is 0.

Function Guide

The function guide provides a detailed explanation of all the functions defined in the code. It explains the purpose of the function and gives a technical insight as to why it was implemented in that way. Note that these functions are not directly available to the user, they are interfaced via userInteraction().

fileStatus

int fileStatus(FILE *fp)

Checks the status of the file pointed to by fp.

Parameters:

- FILE *fp – The file pointer of the target file

Returns:

- 1 - No error found
- -1 - Error encountered

When an error is encountered, the error message is printed. It uses a simple if-statement to check whether the pointer is NULL.

slicePathname

char *slicePathname(const char *pathname, const char *toReturn)

Slices a pathname into a directory component and a file component. The specified component is returned.

Parameters:

- const char *pathname – The pathname of the file.
- const char *toReturn – The component of the pathname to return. Either dirname or fileName.

Returns: Returns either one of two strings

- char *dirNamePart – The directory component of the pathname.
- char *fileNamePart – The file component of the pathname.

This function is used in other functions when working with directories. The main reason for creating this function was when I decided to include directory navigation and to facilitate using the file operations in one directory from a different directory.

If the pathname contains '/', the directory component is obtained by copying n bytes from the pathname to a new string. Where n is the index at which the last '/' character occurs. The index is obtained by looping through the pathname and updating a counter variable.

The file component is obtained by copying the results of strtok(NULL, "/\n") into a variable and constantly overwriting it until the end of the string is reached. At this point the variable stores the file component.

fileExists

int fileExists(const char *filename)

Checks whether a file exists.

Parameters:

- const char *filename – The name of the target file.

Returns:

- 1 – If the file exists.
- 0 – If the file does not exist.

This function works by opening and reading the directory of the file. The directory is then searched to find an entry name which matches the target file name. Originally, this function worked by opening the file in read mode. If it was successful, it meant that the file exists.

I changed the approach because I tried to minimize the number of times a file is opened and close. Almost every other function would have a call to fileExists and would also open the target file. The reasoning behind the chosen approach is that there is less chance for an I/O

error if files are not being opened, closed and reopened as much, even though a directory is opened. The function can also check if a directory exists.

makedir

void makedir(const char *dirname)

Makes a new directory with permissions 700.

Parameters:

- const char *dirname – The name of the directory.

Makes use of the mkdir(...) function. The function does not overwrite a directory if it already exists.

changedir

void changedir(const char *dirname)

Changes the current directory to the specified directory.

Parameters:

- const char *dirname – The name of the directory.

Makes use of the chdir(...) function. The directory is only changed within the program. After the program ends, the process is killed and the user will be at the original directory from where the program was ran.

listDir

void listDir(const char *dirname)

Lists entries in a directory.

Parameters:

- const char *dirname – The name of the directory.

The function makes use of opendir() and readdir() to print out the entries in the directory.

lowerStr

char *lowerStr(char *s)

Converts a string to lowercase

Parameters:

- char *s – The string to be converted.

Returns:

- char *s – The lowercase string.

This function works by looping through all the characters in a string and applying toLower on them.

Lower case strings are needed when comparing user input.

digitCount

int digitCount(int number)

Counts the number of digits in an integer.

Parameters:

- int number – The integer whose digits are to be counted

Returns:

- int counter – The number of digits

This function works by converting the integer to a string using sprintf. The length of the string is obtained and returned.

A digit count is needed for aesthetic purposes: displaying line numbers with the correct number of columns based on the digit count of the total number of lines.

countLines

int countLines(const char *filename)

Counts the number of lines in a file

Parameters:

- const char *filename – The name of the file.

Returns:

- int counter – The number of lines

This function works by looking for newline characters and incrementing the counter accordingly. fgets scans the lines and a newline character at the last index of the scan string indicates a new line.

createFile

char createFile(const char *filename)

Creates a new file.

Parameters:

- const char *filename – The name of the file to be created.

Returns:

- char create – 'y' for successful creation else 'n'

The function can also overwrite files, it prompts the user and asks for confirmation.

The function returns a char because it is used in the copy function when asking to overwrite the destination file.

Files are simply created by opening them in write mode.

deleteFile

void deleteFile(const char *filename)

Deletes a file.

Parameters:

- const char *filename – The name of the file to be deleted.

Makes use of the remove function.

This is the only function that calls updateLog with the fileDeleted argument as 1. This tells the updateLog function that the file has been deleted so it should not attempt to count its lines, since it does not exist. Instead the line count is stated to be 0 for the file entry in the log file.

sendToDisplay

int sendToDisplay(const char *filename, int startLine, int endLine)

Sends the content of a file to the display. The displayed content is in the range of lines from startLine to endLine (inclusive). Includes line numbers.

Parameters:

- const char *filename – The name of the file to be displayed.
- int startline – The start line number of the display range.
- int endline – The end line number of the display range.

Returns:

- 0 if an error is encountered .
- 1 if the content can be displayed.

This is the longest function, mostly due to the number of possible conditions when displaying text.

To obtain the given range, the function uses fgets to scan the file and find newline characters(last character of a string). A counter is incremented and it does this till each reaches the startLine. However, the startLine may be 0.
Similarly, the work of the function will be on all the lines up to endLine, which may also be the last line of the file.

The int 'state variables' used where newLine, multiLine and breakLong. These will either hold 0(false) or 1(true). They describe the state of a string scanned by fgets.

The main work of the function is to go through the file, line by line, and display it appropriately. Some of the if-statements are straightforward, and others are more convoluted. The code comments do well to explain them.

Some of the cases are:

- a line where the line spans multiple rows
- a multi-line where the last word crosses the bufferSize boundary of fgets, so the whole word needs to be moved to the next row
- a line whose first character is a blank space
- for the extreme users, a line where a single continuous word spans multiple rows

Many more of the cases are just different combinations of those mentioned above.

Another reason for the function's nature is the inclusion of line numbers. The need for appropriate formatting added to the complexity of the function.

It is worth taking a look at how a long word is moved from the end of a row to the next row. The word is read backwards, character by character, from the end of the line until a blankspace is read. These characters are stored as a separate sting and are replaced by blanks at the end of the original line.

The characters are (reversed and) printed on the next row. fgets scans the next line(row) from the file and it is printed with the reversed characters as a prefix. In effect, a word which is cut-off by bufferSize is printed on the next row in its entirety.

Another option would have been to use hyphens, but it is not as elegant.

An interesting point to be made is that junk characters were printed out when a single continuous word spanning multiple lines was printed. It was probably due to breaking strings into substrings and reversing characters multiple times. Somewhere along the process, an index might have been missed out.

Bear in mind that this only happened when testing a word which spanned about 200 rows. I corrected it by replacing the junk with blanks. Junk is identified by comparing the length of arrays, please refer to the code.
Only one or two junk characters were ever detected.

Apart from that, the majority of the function is just print statements with different formatting applied.

showLineRange

void showLineRange(const char *filename, int startLine, int endLine)

Displays the contents of a file in a line range (inclusive).

Parameters:

- const char *filename – The name of the file to be displayed.
- int startline – The start line number of the display range.
- int endline – The end line number of the display range.

This function just calls sendToDisplay with its parameters. The reason for making a new function instead of just using sendToDisplay was to simplify customising different outputs for different kinds of display functions.

showLine

void showLine(const char *filename, int lineNumber)

Displays a single line of a file.

Parameters:

- const char *filename – The name of the file to be displayed.
- int lineNumber – The line number to be displayed.

This function just calls sendToDisplay with its parameters. The reason for making a new function instead of just using sendToDisplay was to simplify customising different outputs for different kinds of display functions.

The startLine and endLine are both equal to the lineNumber of lines in the file, obtained by calling countLines.

showFile

void showFile(const char *filename)

Parameters:

- const char *filename – The name of the file to be displayed.

This function just calls sendToDisplay with its parameters. The reason for making a new function instead of just using sendToDisplay was to simplify customising different outputs for different kinds of display functions.

The startLine is 0 and the endLine is the total number of lines in the file, obtained by calling countLines.

showLineCount

void showLineCount(const char *filename)

Displays the number of lines in a file.

Parameters:

- const char *filename – The name of the target file.

This function makes use of the counLines function.

showSingleLog

void showSingleLog(const char *filename)

Displays the change log of a single file.

Parameters:

- const char *filename – The name of the target file.

This function works by searching and retrieving the appropriate line numbers in the log file and then passing them as parameters to showLineRange.

The format of a log file is:
'file1' log:
+ Operation
+ Operation
- Operation

'file2' log:
+ Operation
- Operation
- Operation

sprintf is used to format the given filename into the appropriate format, that is the "'filename' log:".

To find the entry, fgets scans each line and they are compared to our formatted filename string. When a match is found, fgets continues to scan the lines until a sequence of '\n\n' is found.
Such a sequence is only found between two different file entries. Both the endLine and startLine have been retrieved and are passed to showLineRange.

Note that fgets uses logBufferSize, which is larger, instead of the usual bufferSize. This is to deal with very long file names for more 'adventurous' users.

showFullLog

void showFullLog()

Displays the full change log of all files in the current directory.

Makes a call to showFile with logFile as the parameter.

Note that fgets uses logBufferSize, which is larger, instead of the usual bufferSize. This is to deal with very long file names for more 'adventurous' users.

copyFile

void copyFile(const char *origFile, const char *newFile)

Copies content of one file to another file.

Parameters:

- const char *origFile – The name of the source file.
- const char *newFile – The name of the destination file.

This function uses fgets to scan each line in origFile and then writes them in newFile.

If newFile already exists, the user is prompted and asked to confirm an overwrite. This is because a call to createFile is made.

appendLine

void appendLine(const char *filename, const char *appendstr)

Adds a line to the end of a file.

Parameters:

- const char *filename – The name of the target file.
- const char *appendstr – The string to be appended to the file.

This function opens a file in a+ mode, for reading and appending. The last character of a file is read. If this character is a newline, appendstr is written on that line.
Else if the last character is not a newline, a newline is written and appendstr is written on the new line.

The function also asks to create the file if it does not already exist.

lineDeletion

int lineDeletion(const char *filename, int startLine, int endLine)

Deletes a range of lines (inclusive) from a file.

Parameters:

- const char *filename – The name of the target file.
- int startline – The start line number in the deletion range.
- int endline – The end line number in the deletion range.

Returns:

- 0 Unable to delete.
- 1 Successful deletion.

A temporary file is created using tempfile(). The target file is opened in read mode.

The function scans lines using fgets and increments the counter when an newline character is identified (as the last character of fgets return string). Each line is written to the temp file. This is done until startLine is reached.

From startLine till endLine, the process is repeated but none of the lines are written to the temp file.
From endLine to the last line, the lines are written to the temp file.

The temp file now contains all the lines from the original file except from the lines specified for deletion.

The target file is closed and reopened in write mode, clearing all its content. Each line from the temp file is then written to the target file. The deletion is complete.

deleteLineRange

void deleteLineRange(const char *filename, int startLine, int endLine)

Deletes a range of lines (inclusive) from a file.

Parameters:

- const char *filename – The name of the target file.

- int startline – The start line number in the deletion range.
- int endline – The end line number in the deletion range.

Makes a call to lineDeletion with its parameters. The reason for making a new function instead of just using lineDeletion is for customising outputs of different delete types.

truncateFile

void truncateFile(const char *filename, int lineNumber)

Deletes all lines from the given line number and downwards.

Parameters:

- const char *filename – The name of the target file.
- int lineNumber – The number of the line to begin deletion.

Makes a call to lineDeletion with its parameters (endLine being the total line number). The reason for making a new function instead of just using lineDeletion is for customising outputs of different delete types.

deleteLine

void deleteLine(const char *filename, int lineNumber)

Deletes a single line from a file.

Parameters:

- const char *filename – The name of the target file.
- int lineNumber – The number of the line to be deleted.

Makes a call to lineDeletion with its parameters (endLine, startLine and lineNumber being equal). The reason for making a new function instead of just using lineDeletion is for customising outputs of different delete types.

clearFile

void clearFile(const char *filename)

Clears the content of a file.

Parameters:

- const char *filename – The name of the target file.

Makes a call to lineDeletion with its parameters (startLine is 0 and endLine is countLines(filename)). The reason for making a new function instead of just using lineDeletion is for customising outputs of different delete types.

insertLine

void insertLine(const char *filename, int lineNumber, const char *insertStr)

Inserts a line at the given line number.

Parameters:

- const char *filename – The name of the target file.
- int lineNumber – The line number to insert at.
- const char *insertStr – The string to be inserted.

A temporary file is created and the target file is opened in r+ mode (updating).

This function works by scanning (using fgets) each line and writing them into the temp file until the target line number is reached.
A counter keeps track of the line number by incrementing when a newline character is identified at the last index of the fgets return string.

When the startLine is reached, the insertStr is written into the temporary file.
The remaining lines in the target file are read and written into the temporary file. At this point, the temporary file includes all the content of the target file and the insertStr as well.

The target file is reopened in write mode, clearing its content. Each line from the temporary file is then written into the target file.

updateLog

void updateLog(const char *filename, int fileDeleted, const char *logStr)

Updates the log of files in a directory. Keeps track of any changes made to the files.

Parameters:

- const char *filename – The name of the file to be updated in the log.
- int fileDeleted – either a 0 or 1. 0 denotes that the file to be be logged exists. 1 denotes that the file has been deleted (after deleteFile(..))
- const char *logStr – The string of text which describes the change which occured in the file.

The format of the log file entries is:
'file1' log:
+ Operation
+ Operation
- Operation

'file2' log:
+ Operation
- Operation
- Operation

This function is able to create new entries for files and add operations to current entries. Altering a file in a directory from the program will cause a log file to be created there.

sprintf is used to format the given filename into the appropriate format, that is the "'filename' log:".

To find the entry, fgets scans each line and they are compared to our formatted filename string. If a match is found, fgets continues to scan the lines until a sequence of '\n\n' is found.
Such a sequence is only found between two different file entries.
This means that the most recent recorded operation for the file is on the previous line. The insertLine method can be called with this line number as a parameter and the insertStr as the logStr (describes the operation).

If a file entry does not exist, it is created in the format and the operation is also added.

A good amount of formatting is applied in this function, for aesthetic purposes. A timestamp is also included by using time() and localtime().

At the beginning of the function, the target file name is manipulated to obtain the directory and file name components (if applicable) using splicePathname.

This is done so log files can be created in any directory as well as displaying correct paths within the file itself.
For example, running the program in /home/dir1/dir2 and creating a file with file name ./dir3/dir4/dir5/newFile.txt will cause a log file to be created in dir5.
The file log entry would be for newFile.txt and not ./dir3/dir4/dir5/newFile.txt.

userInteraction

int userInteraction()

Provides the interface for the user to interact with the functions. It makes use of a while loop which runs until the user explicitly ends the program.

fgets obtains user input from stdin.

A string of input is analysed word by word and is matched to specific functions by using if-statements. To obtain each word in the input, strtok is used with blankspaces and newlines as separators.

When appropriate, user input is converted to lower case for matching the input to command names.

For functions such as insertLine and appendLine, the delimeter for the final strtok is just a newline, not a blank space. This allows the use to enter a whole string of text with spaces.

Any numerical arguments are first received as strings and then converted to long using strtol.