**Documentation**

**Powerset.c**

*Mathematical Research – for minimizing ambiguity*

To be able to solve this exercise correctly there must be precise and clear understanding of the mathematical terms that it is referring to. According to the (Rosen, 2019) a set S is a collection of distinct objects which are unordered. Those objects can also be called elements of the set. Also, given a set S, the powerset of the set S is defined as a set of all subsets of the set S. So, the aim of this program is to print all the subsets of any arbitrary set S. To be able to print a powerset of any set using a programming language we need a relationship that expresses the size of any powerset in respect to the size of the set. From (Hammack, 2018), the relationship is the following |P(S)|=2|S| .

*Explanation of how the program is formed – Design Analysis*

Regarding the ***main function*** of the program. It is where the user is asked to input the whole subset. To make sure that enough space is allocated, the inputted set is stored in an array with space for 1000 characters for each element and 20 elements max. So realistically that is enough space for the specific exercise. Writing more than 1000 characters per element and more than 20 elements seems unlikely to happen. It is an exercise that solves a maths problem. And by researching different sources, and math books, it is very clear that the elements as well as their characters are never too many. The specific program is not indented to be used to solve an essay style question.

Also when testing the program, it’s just about before crashing when the cardinality of the Set is 20. Because it is about 1048576 subsets(makes the program unusable).

For the user input the program uses the

char\* fgets(char\* str, int n, FILE\* stream); function, because according to (Educative, 2022), “fgets is safe to use in comparison to gets since it checks for character array str bounds. gets keeps on reading characters from the users, until a newline character is encountered.”. This input function is executed repeatedly until the user inputs nothing. With the set stored in a 2D array of characters, it is now passed as a parameter in the printPowerSet() procedure to print calculate and print the powerset.

In the ***printPowerSet()*** function, firstly the size of the power set is calculated and stored in the powerset\_size variable(which according to the mathematics Size\_of\_Powerset =2Size\_of\_Set). The logic behind this code is that we want to print 2Size\_of\_Set number of combinations of the elements of the set. Which basically is the exact same logic as wanting to print all the binary representations of the numbers from 0 to a denary number. Lastly this function contains print statements to the right places so that the power set will be illustrated in a nice way.

Example:

Thus,

\*It is important to emphasize that the order of the elements in each subset, as well as the order of the subsets in the power set is completely irrelevant (as it was already mentioned in the research part of the documentation above) .

**Testing:**

It is noticeable that after a certain cardinality the power set becomes huge.

More specifically when the set’s cardinality is 20 the power set’s is 1mill. And the program produces critical performance drop. Sometimes it even shuts down the whole program execution because it feels up the CPU. To find where the problem is I added testing times, where each time the program is run it also outputs how much time was needed to calculate the power set. Apparently, the power set was successfully calculated very quickly each time. So, we can assume that the whole problem is caused when it comes to print on the screen that large amount of data, but this is not something that the specific program aims to solve. However, combining this program with another program that uses a buffer efficiently to print huge amounts of data it, then the cardinality of the input set could increase.

**Conclusion:**

Clock calculation reference:

https://www.techiedelight.com/find-execution-time-c-program/#:~:text=We%20can%20use%20the%20clock,total%20number%20of%20clock%20ticks.

**EditSmart.c**

*Research on command line editing file systems:*

From the first command line file text editor created in 1969 by Ken Thompson (Wikipedia, 2021), there are some key features that remain in the more modern command line file editors, but also there has been many new and differnt operations from editor to editor. For example in the (Fraser, 1980) the author discusses about a generalized editor that edits text, directories, binary core images, and certain operating system data with a single user interface.

This is an exercise with many different operations that can be executed, therefore I chose to divide the code into different functions-one function for each operation. Using top-down design makes our program modular and has many advantages. The main function does not get to crowed, the code is easier to debug, easier to modify and it’s more organised.

**General Specifications**

* Sting Input Stream

For this particular program I chose to use the fgets() over the scanf() to input data. Fgets() allows us to specify a specific buffer length ,whereas scanf( ) doesn’t. That means that by using fgets() we can avoid buffer overflow error that it could occur with the scanf(). (jflopezfernandez, 2019)

char \*fqets(char \*line, int maxline, FILE \*fp)

(Kernighan & Ritchie, 1988)

* Character Input Stream - Mode[2]
* Handling buffer

One of the hardest challenges in the design of this program was dealing with input data. This program is heavily based on what the user inputs. In other programming languages inputting data is straight forward. In C when inputting data, it is very critical to make sure that the input buffer is where we want it to be. In order to do that, whenever data is inputted, the buffer is immediately cleaned:

fgets(name, sizeof(name), stdin); //inputs file name

fseek(stdin,0,SEEK\_END); //clears buffer

* File name adjustment

In the process of inputting a file name it was noticed that each time there was an unwanted character in the end. This resulted in the creation of files with wrong file names. After testing different methods to address this problem, I found that if each time the file name is readjusted in a way that the second character from the end becomes the last character (‘\0’) the problem is solved.

name[strlen(name)-1]='\0'; //adjusts file name

//where strlen(name) is the size of the string

* Checking when a file is NULL. Instead of using the EXIT function, I handled it myself, because I want the program to continue running and not end immediately when something goes wrong.

More specific steps:

-Outputs Error Message

-Closes all open file streams

-No Recursion

* Log File: Every time one of the operations is executed, the program puts information about that operation in this file. It is the only global file pointer in the program since it needs to be accessibly by every function. Moreover, along with the description of each operation that takes place, it also adds date and time to each entry.
* When it comes to storing a line either because there is a need to read a line from a file or the user inputs a line, this program uses an array to do so. The array size is capped at 1025. It’s a large enough number and it should be sufficient in all the possible cases. Also we know that even if the input is larger there will be no error, due to the usage fgets( ). The number is specifically 1025, because 1024 is also the maximum number of characters that Microsoft Notepad allows (N, 2011), plus one because of the null character that C adds to the end

*Design chart of the program:*

**Input: 2**

**Input: 1**

**Input: 4**

**Input: 4**

**Input: 4**

**Input: 1**

**Input: 3**

**Input: 3**

**Input: 1**

**Input: 2**

**Input: 2**

**Input: 3**

**Input: 1**

**Input: 2**

**Main( ) :** Calling the print\_index() to print the menu of what the user has to input in order to access certain operations. Makes sure that if character entered by the user is not in the menu to ask user to input again until the character entered will be valid. It then uses switch statement to match the input character to the function that will insert the user to the corresponding operation. Instead of the whole program terminating once the function is over, there is an infinite while loop that will run the whole process again. It’s up to the user when he wants to terminate the program. That way if the user wants to access multiple operations , they don’t have to run the whole program again. It’s also how most applications work in the modern world. The program terminating after just one operation can be quite exhausting for the user.

**File\_op( ):** It is where the user is asked to choose a specific mode to be redirected to. It works very similarly with the main( ) function. Except the function is recursively called when it’s finished until the user decides to aborts by typing ‘x’ for mode. This function does not asks for a file name, since that depends on the kind of mode chosen. There might be one or two files needed for different sub-modes.

**Line\_op( ):** In this operation the first input from the user is the file name. Because it is about line manipulation, all of its sub-modes will be applied to only one file, so it makes sense to ask for the file name first and then move on. The rest of this function design is very similar to the File\_op( ), but with different context.

**General\_op( ):** After asking the user to choose one of the two modes, it executes those modes without referencing other functions. The log it just

**Createfile( )**: Request file name from the user to input. After checking that the file stream is opened successfully it also provides the user with the option to write lines to the file. In case the file failed to be created it asks for another filename to be inserted

**Deletefile( ):** Request from the user to input the name of file that wants to be deleted. Checks if the file was deleted successfully with the if(remove(name) == 0) statement and prints a feedback statement(a statement if the file was delete or a statement if it could not be delete ) .

**Copyfile( ):** It asks the user to input the name of the file that they want to copy. In case the file cannot be opened to read, it askes again until finally a file is opened successfully. After that with the exact same process it askes and opens a file to write. When both of the file pointers are set to the beginning of the two files, with a while loop it reads one character at a time from the first file and puts that character in the second file, until the character that is read is EOF which means that the end of the file has been reached. Finally the function prints a statement to the command line to confirm to the user that operation is carried out successfully and it closes the files streams. For more flexibility the user is able to abort the operation at any time.

**Show\_file( ):** Opens the file that the user has inputed to read it( using “ r ” mode). Reads one by one character and outputs it on the screen until the end of file is reached.

**Append\_line( ):** It works the same way as writing a line which is done in the Createfile( ) function. The only difference is the the file is opened using “a” mode(append).

**Insert\_line( )**: To insert a line in a file the user is firstly asked to enter number of line, as well as context that wants to put in that line. Then the whole file is being copied to a temporary file that the program creates which works like a buffer(This file’s name also starts with “.” as it’s consider the invisible file). When it reaches the number of line that the user has entered, it puts the user’s input content to that line and then it continuous copying the original file to the buffer file until the EOF is reached. Then in the exact same way it copies the the buffer file to the original file(rewriting the contents original file with the contents of the buffer file). The process of coping data from one file to another is the same as in the ***copyfile()*** function(see above). Generally this operation could be done by using an array instead of a file as a buffer. The drawback of using an array in this case is that the file may be exiting the array size sometimes and thus other problems are created (matsp, 2009). Therefore, using a file seemed a better option. Finally there is also a special case that had to be taken into consideration when implementing this operation. More specifically, when the number of line inserted is not an existing line in the file, it adds next line instead of EOF in the buffer file until the number of line we want is reached and the new line content is placed.

**Delete\_line( ):** Follows very similar logic as the ***Insert\_line()***. But, instead of writing a new line in the buffer file, it just doesn’t put the line that we want to delete in in the buffer file and it continues by adding each other line one line above. And then it overwrites the original file with the buffer.

**Show\_line( ):** Counts the lines of the file until the number of the line that the user has requested to view is found. The first line of the file is line number 1 and every time a ‘\n’ character is read, it adds one to the number of the line. When the line is found it reads and prints every character of that line and stops reading the file further. The isfound is an integer variable that if the number of the line wasn’t found in the file it remains 0, and prints a statement to inform the user, otherwise it will become 1 and not show the message.

**Count\_lines( ):** It’s a function that takes any file pointer and returns the number of lines of a file. The pointer is restored to its previous position once is used in the function.

**Search( ):** From various sources it is very noticeable that every file editor has a way of searching for specific data inside. So this function provides the user of the program the ability to write a string and to be shown the lines that the string is found in the file. This function extents further to a second operation. After seen the lines that the input string is included, the user has

The choice to replace that string with another one in a specified line. However if the string is included more than one times in a line, all the strings that are the same as the input string in that line, will be replaced too.

# Bibliography

Educative, I., 2022. *https://www.educative.io/.* [Online]   
Available at: https://www.educative.io/edpresso/how-to-use-the-fgets-function-in-c  
[Accessed 12 January 2022].

Hammack, R., 2018. *Book of Proof.* 3 ed. Richmond: Hammack, Richard.

jflopezfernandez, 2019. *GeeksforGeeks.* [Online]   
Available at: https://www.geeksforgeeks.org/why-to-use-fgets-over-scanf-in-c/  
[Accessed 5 January 2022].

Kernighan, B. W. & Ritchie, D. M., 1988. *The C Programming Language.* 2 ed. Murray Hill(New Jersey): AT&T Bell Laboratories.

matsp, 2009. *Cprogramming.com.* [Online]   
Available at: https://cboard.cprogramming.com/c-programming/112317-how-insert-new-line-file.html  
[Accessed 24 January 2022].

Rosen, K. H., 2019. *Discrete Mathematics and Its Applications.* 8th Edition ed. New York: McGraw-Hill Education.