COMP1028 Programming and Algorithm
Session: Autumn 2025
Group Coursework (25%)

Group Name	**banana split**				
Group Members					
Name 1	Yee Grace Shuang		**ID 1**	20716333	
Name 2	Ooi You Sheng, Ciaran		**ID 2**	20708501	
Name 3	Min Pyae Phyo		**ID 3**	20710858	
Name 4	Sivanesan Jeevananthan		**ID 4**	20715651	
Name 5			**ID 5**		
Marks **Total (100)**	Program Functionalities (65)	Bonus (5)	Code Quality (10)	Presentation / Demo (10)	Report / Documentation (10)
Submission Date	**05/12/2025**				

1. Introduction

Explain concisely the scope of the project and what your implementation aims to achieve (can be in point form).

This project implements a Cyberbullying / Toxic and Benign Text Analyzer in C. It loads the data from the input files (either .txt or .csv format). It then can split the obtained sentences into lowercase words, removes all punctuations aside from hyphens and asterisks, discards common stopwords using stopword.txt, and then saves the clean filtered words into a .txt file (in the format of "x_filtered.txt) before analysing the statistics of the text. Toxic and benign phrase/word detection then can be done based on toxicwords.txt and benignwords.txt to get statistics such as the toxicity score. The statistics obtained can be either saved to an output file (in the format of "xAnalysisReport.csv") or used to sort by top toxic or frequent words using either bubble, quick, or merge sort. The program uses a menu-driven interface in the command prompt, and it can also display an ASCII chart of the top N words.

2. Key Design Choices

Describe in concise way the major design decisions in your implementation (can be in point form). Suggested subsections include (delete sections not applicable to you):

2.1 Data structures used (arrays, structs, hash tables, etc.)

FileData struct (from textinput.c) stores the raw lines. UniqueWords struct stores the frequency of all the unique words and AnalysisData struct (from tokenisation.c) stores the cleaned words (without stopwords or punctuation) and general statistics of the file. From worddetection.c, the FoundWord struct stores the frequency and the severity of the detected toxic or benign word, detectedWordData struct stores the toxic or benign statistics, and the WordListTerm struct stores the toxic or benign words from either toxicwords.txt or benignwords.txt. An enum called Category is used to determine between the three severities.

To reduce the time or cost needed, if the array size is needed to increase, the capacity is doubled each time instead of only increasing by one. Depending on whether the variables inside the struct need to be initialized as 0, calloc may be used to avoid having to manually do so. All the memory allocations are freed after finishing its usage to avoid memory leak. If a memory allocation fails, an error message will be printed before the entire program terminates to prevent any further issue.

2.2 File handling strategy

To handle multiple files, the user is prompted to enter the number of files they wish to process before entering the filenames. An error message will be printed if the file couldn't be found or read. Files which are required to check the file size are opened in read-only binary mode ("rb") for accuracy as the normal read-only mode ("r") will automatically merge things such as a carriage return line feed ("\r\n") into just a single character ("\n"). So using the normal read mode would give a less accurate size due to us using the pointer obtained from fseek() as the file size. Files larger than 10 MB are rejected to prevent the code from freezing

due to the amount of data. Empty files are also skipped.

The method to read the lines from the file is determined by the file extension. TXT files are saved line-by-line into the array using a loop. CSV input files require the row to be a header row, and the user can select the column they want based on the header row's text. Only the chosen column will have the data saved to the array. If the user states that the CSV file does not contain a header row, then the file will be skipped with an error message. Once all the lines of a file are read successfully, the data will be stored in the FileData struct before the program processes the next file. If a file was unsuccessfully read, the allocated memory for that file would be freed.

2.3 Tokenization approach (punctuation removal, case folding, splitting etc.)

To tokenise, the lines inside the FileData struct are used. If a file had an error while being processed previously, then the file will be skipped with an error message. To get the sentence count, the code looks for an ending punctuation (either period, question, or exclamation mark). Then the lines are split by looking for a space, tab, newline, or carriage return character.

Any character that isn't a letter, number, hyphen, or asterisk is discarded to get rid of special characters. Hyphens and asterisks are needed as they could be part of a toxic or benign word. The characters in the split line are then converted to lowercase. The words obtained from the tokenisation are also checked if they are a stopword. If no, it is then checked if the word is a unique word by comparing it against the array of previously found words. Based on the results, the integers keeping the total word count, unique word count, and character count are updated. The said integers will be used to calculate other statistics of the file, and the statistics are printed out.

2.4 Stopword handling

The stopwords.txt file is opened in read-only mode, with a warning message printed if the file couldn't be opened. The program will continue without filtering for stopwords if stopwords.txt could not be opened. If the file is successfully opened, the lines are read into an array. If the array runs out of capacity to store the lines, then its size will be doubled and the memory will be reallocated. If the reallocation fails, the program prints an error before ending itself to prevent any further issue.

To check if a word is a stopword, the program uses a loop to compare each element in the array containing all the stopwords against the word to be checked. The function will return true if yes, or false if no.

2.5 Toxic word detection strategy

The user is first prompted to choose whether they want to detect toxic or benign words. Based on their choice, either toxicwords.txt or benignwords.txt is opened in read-only mode. An error will be printed if the file couldn't be opened and the word detection will be terminated.

If successful, then the lines from the file will be read, with the empty lines being skipped. Each line is split by looking for a "|" character to determine which part contains the word or phrase, and which part contains its severity. The word or phrase is converted into lowercase and saved in an array, while the severity will be chosen from an enum. A flag is also set to true if the word or phrase contains a space character.

If a file the user uploaded to be detected has previously gotten an error, then the said file will be skipped. If not, each untokenized line of the file is first read and converted into lowercase. The line will be compared against the array of the toxic or benign words containing spaces. If the line contains the phrase, then the phrase is saved into the list of detected words, and the count of total or benign words is increased by one along with the respective severity frequency. If an individual word inside the phrase is also detected as toxic or benign, then the word's frequency is decremented by one to prevent double-counting as the phrase should only count as one frequency.

The tokenised words are then compared against the array of the toxic or benign words without space. If detected, the word is saved into the list of detected words and the respective counters are also incremented by one. Once all lines of a file are checked, the counters will be used to calculate the toxic or benign statistics and the statistics are printed out.

2.6 Sorting algorithm(s) used and justification

Merge sort (O(n log n)), quick sort(O(log(n) on average) bubble sort (O(n^2) on average) are implemented in sortingAlgos.h and used in our program for basic sorting. Mergesort is used most often in other functions due to its stability. The user can sort by alphabetical order, frequency, or toxicity score. If the user also wants to look at different severity ratings, they can sort via severity, severity with frequency or severity with alphabetical order. These three algorithms were chosen due to their ease of implementation as well as different use cases.

For example, as merge sort is a stable algorithm with a time complexity of O(n log n), it is recommended for larger data sets where performance is more critical, and where time complexities like bubblesort (O(n^2)) would struggle due to the exponential time complexity of n^2. However, merge sort's space complexity isn't great, with its worst case being O(n) due to having to duplicate the array/structs in order to sort them, leading to memory usage issues and the low chance of merge sort simply failing due to the program not having enough memory.

Bubblesort is a simple unstable algorithm, with its time complexity ranging from O(n) to O(n^2) from best to worst respectively. However, it is almost never going to achieve O(n) since that requires the array/struct to already be sorted in the first place, which means either the data is sorted already or the user performed a sorting function on the array of structs beforehand, making performance/usability in this case somewhat irrelevant as the array is already sorted. However, its space complexity is O(1), making it quite memory efficient. This means that bubblesort can be used to quickly check if the algorithm is sorted without needing any external checks/flags, which can be used in the future if the program is further developed.

Quicksort is another unstable algorithm that performs better with larger datasets. Its performance ranges from O(n log n) to O(n^2) from best to worst, making it basically on par with bubble sort in terms of performance. On average, its time complexity averages to O(n log n), making it almost identical to merge sort, with instability leading to worst edge cases. Although quicksort's time complexity is nothing to talk about, either being equal or worse compared to bubblesort or mergesort, its worst space complexity is O(log n), which is better than mergesort O(n) but worse than bubblesort O(1). This means that quicksort performs most optimally in larger, unsorted datasets where memory is an important factor and where performance is not considered to be too important.

In conclusion, mergesort is stable but has minor memory issues, bubblesort has major time complexity issues and good space complexity but struggles with larger datasets, and quicksort is in between, having decent performance and decent memory usage for larger datasets.

2.7 Menu-driven user interface design

The main menu in main.c presents options such as load files, tokenize and analyse files, toxic detection and benign word detection, update toxic or benign word list, save the analysis report, load the stop/toxic/benign word list from an analysis report, sort and display the top N words, compare the stats of two files, and exit. The options are chosen using the user's input and a switch case. An error will be printed if the input is invalid.

There are various flags to ensure the user has completed the analysis required for certain options, such as writing the file output into persistent storage or using sorting algorithms to sort the words, to prevent having an error for missing information. If the user chooses to exit, all the memory allocated will be freed to prevent memory leak before stopping the program.

Additionally, we also implemented a bar chart to display certain sorting results using the # character, which scales according to the highest frequency of a word in order to avoid the bar chart looking too short or too long. This is done by making the bar chart length be based on the percentage of the highest frequency, so all bar size is relative to that, with checks to ensure that anything that is reduced to 0 due to integer division, but actually has frequency is printed as a single # for the bar chart.

If the user chose to save the analysed data into an output file, the data will be saved in a CSV file, with the first row being a header row. To be able to write the data across a span of columns instead of only using the first column, the longest list needing to be written is found by comparing all the list sizes so the code knows when to stop writing to the file. Then, the data is written row-by-row and uses commas to indicate when to move to a new column. If any error is encountered, then the output file is deleted to prevent the user from getting incomplete data. And at the end, all the memory allocated is freed to prevent memory leak.

To reload the stop/toxic/benign word list file using the one saved by an output file, the program manually looks for the comma indicating a new column as strtok cannot be used since it ignores empty cells. Based on the column number, the cell content is written to the appropriate file.

3. Challenges Faced and Lessons Learned

Discuss difficulties encountered during development in a concise manner. Examples:

- String handling complexities
- Memory management
- Handling large text files
- Efficiency considerations
- etc.

Reflect on what your group learned about C programming, algorithms, text processing, and working collaboratively.

During the whole process we faced numerous roadblocks while creating the text-analysis system in C. One was dealing with strings and memory manually as C doesn't automatically free unneeded memory allocation nor have a dedicated string variable like C++. We dealt with this by using malloc, calloc, realloc, and free when necessary. For example, to prevent memory leak, we would free the allocated memory if the program will no longer use it. We used calloc instead of malloc if we needed to initialize all the variables to NULL or 0. Even though calloc is said to be slower than malloc, as we didn't have to manually create a loop to set the variables to 0, it helps the program be more efficient.

As arrays need to be initialized with a size, it becomes resource-heavy if we have to consistently increase the size by 1 every time the array is full. So instead, we double the array size each time it is full. This makes it easier to manage memory, as the lines to resize the arrays are executed less, thus making it more efficient. Since strings require a size when being initialized, we also check and see if the input is over the size limit in some scenarios, such as when they are entering the filepath. If it is, we print an error message informing the user to enter a shorter text to avoid the input from being cut-off due to being too big.

Another issue was to write a word detection function capable of detecting toxic or benign phrases and only counting the frequency as one. At first, phrases with another toxic or benign word inside it would count as multiple frequencies. For example, "son of a bitch" would be counted as one frequency, and the word "bitch" inside it would also count as another frequency.

To avoid this, once we detected a toxic or benign phrase, we would tokenise that phrase into individual words. Then, we compare the individual words with the benign or toxic dictionary. If a match was found, then we decrease the word's frequency by 1 so later when the code does the detection for single words that don't contain spaces, the deduction would cancel out the increment that would have made the frequency inaccurate.

We also had troubles when trying to load the stop/toxic/benign dictionaries from the ones saved in an output CSV file, as strtok would ignore empty cells. This is an issue since our CSV file contains empty cells for both readability and due to some data being shorter than others, thus needing less rows to be written to. To solve this, we manually get and keep track of the current column by using strchr to look for the commas indicating the start of a new column. Using the pointer obtained from strchr, we can use it to make the code only read the cell inside the specific column. With this, we can accurately keep track of the current column and be able to know which part of the row we need to write into the corresponding

dictionary file.

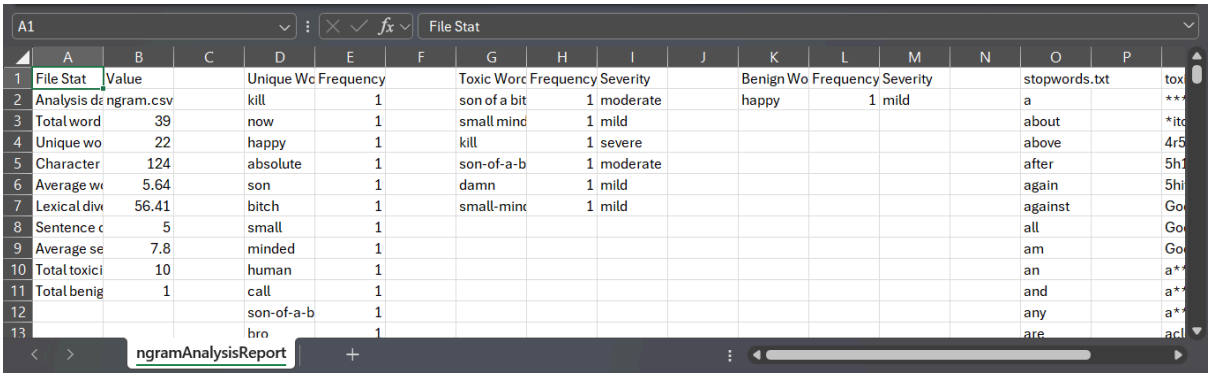
Lastly, we found it difficult to collaborate as there were some different people working on separate stages of the assignment. We solved this by clearly dividing up the work between each member, sharing changes regularly, and testing the codes for errors before putting everything together. For stages which require the information from previous stages, such as the sorting algorithm requiring the list of words and its severity, we fixed it by temporarily using placeholder data structures. Then, when the required stage is finished, we can just replace the placeholder with the actual data structure.

This project taught our group how to write cleaner, more structured C programs, particularly with the use of pointers and memory allocation, including the splitting of code across multiple source files by using header (“.h”) files. Also, the basic algorithms such as sorting were better understood. We learnt how to debug by using printf to check which parts of the code aren't working, and tested by trying out multiple combinations of input files. In text processing, we learned how to change strings to lowercase, remove punctuation, and handle input files, which is often messy or inconsistent, without crashing. Finally, by working collaboratively, we learned how to divide tasks, run the code from different C files, and communicate efficiently to code a complete program together.

Appendices (optional)

Include any additional materials, screenshots, tables, etc. (if any)

Below is an example of the saved output file, containing the data about ngram.csv.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	File Stat	Value		Unique Wc	Frequency		Toxic Worc	Frequency	Severity		Benign Wo	Frequency	Severity		stopwords.txt	tox
2	Analysis da	ngram.csv		kill	1		son of a bit	1	moderate		happy	1	mild		a	***
3	Total word	39		now	1		small mind	1	mild						about	*it
4	Unique wo	22		happy	1		kill	1	severe						above	4r5
5	Character	124		absolute	1		son-of-a-b	1	moderate						after	5h1
6	Average w	5.64		son	1		damn	1	mild						again	5hi
7	Lexical div	56.41		bitch	1		small-minc	1	mild						against	Go
8	Sentence c	5		small	1										all	Go
9	Average se	7.8		minded	1										am	Go
10	Total toxici	10		human	1										an	a**
11	Total benig	1		call	1										and	a**
12				son-of-a-b	1										any	a**
13				bro	1										are	acl

ngramAnalysisReport

Appendix A: ReadMe

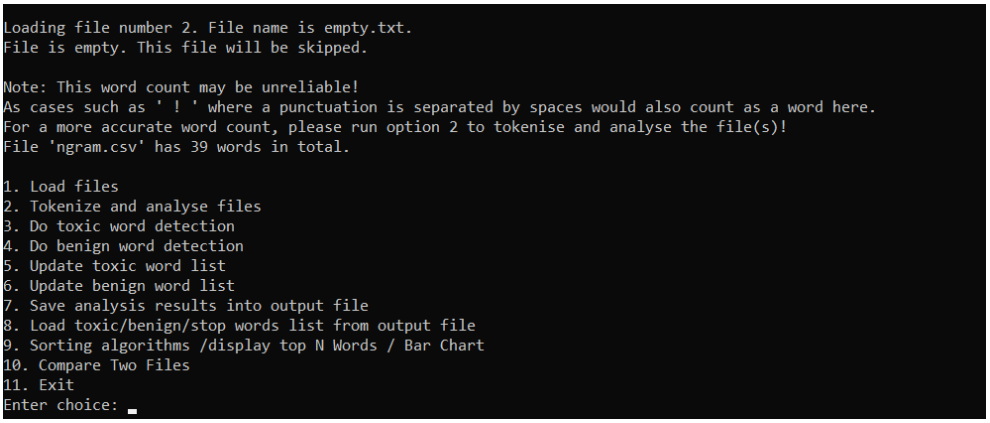
Provide instructions for compiling and running your program. Example:

- How to compile using gcc / Visual Studio Code / Visual Studio Community or any tool you used
- Required input files (e.g. stopwords.txt, toxicwords.txt etc.)
- Example command-line usage (screenshots)
- Notes on dependencies, e.g. special header files needed to be included (if any)

To compile using minGW in the command prompt, open the command prompt in the same folder as all the source files and enter “gcc main.c textinput.c tokenisation.c worddetection.c sortingAlgos.c output.c -o main -std=c99”. You can then run the program with “.\main.exe” or just clicking on the EXE file in the file browser.

The required input files are stopwords.txt, toxicwords.txt, and benignwords.txt. To test the program, you can also use normal.csv, normaltxt.txt, ngram.csv, ngramtxt.txt. To test larger files, you can use max.csv or maxtxt.txt, which are 7 to 8MB. To test for errors, you can use empty.csv, empty, or empty.txt to test empty files. An example of loading an empty file and a normal file is attached below.

```
C:\Users\jim\Downloads\uploadThis2>gcc main.c textinput.c tokenisation.c worddetection.c sortingAlgos.c output.c -o main -std=c99

C:\Users\jim\Downloads\uploadThis2>.\main

1. Load files
2. Tokenize and analyse files
3. Do toxic word detection
4. Do benign word detection
5. Update toxic word list
6. Update benign word list
7. Save analysis results into output file
8. Load toxic/benign/stop words list from output file
9. Sorting algorithms /display top N Words / Bar Chart
10. Compare Two Files
11. Exit
Enter choice: 1

Enter how many files to analyse (up to 5 files can be analysed at a time): 2

Enter path for file 1: ngram.csv

Enter path for file 2: empty.txt

Loading file number 1. File name is ngram.csv.
List of columns found in order: text column

Is this row a header row? Enter 1 for yes, 0 for no: 1
Please enter the column number to analyze: 1
Successfully loaded 10 lines from 'ngram.csv'!

Loading file number 2. File name is empty.txt.
File is empty. This file will be skipped.
```



GitHub Link & Video Link

GitHub Repository: https://github.com/Mint3Ds/Toxic-Text-Analyzer

(Note: Please also zip all project files and submit a copy to Moodle as well)

Video Presentation Link:

https://drive.google.com/file/d/1UFaQeDarJn9KjolUQ83LStZUahBXy01U/view?usp=sharing

Appendix B: Marking Scheme Summary (Optional)

Criteria	Description / Notes
Text Input & File Processing	Load and read a .txt file line by line. Count total words. Handle basic file errors. Support multiple input files. Handle large files by limiting maximum size. Support CSV input. Optimize reading for very large datasets by doubling a full array's size to be more efficient.
Tokenization & Word Analysis	Tokenize words (normalize case, remove punctuation). Count unique words (excluding stopwords). Ignore stopwords using stopwords.txt. Provide character-level stats (avg. word length). Save filtered word list (excluding stopwords). Calculate lexical diversity (calculated with (unique/total count)). Add sentence-level stats (avg. sentence length).
Toxic Word Detection	Detect toxic words using toxicwords.txt. Detect benign words using benignwords.txt. Count frequency and % toxic words. Allow dictionary updates (add new terms). Print frequency of each toxic word. Categorize toxic words by severity. Detect multi-word toxic phrases (bigrams/trigrams).
Sorting & Reporting	Implemented three sorting algorithms: Mergesort, quicksort, bubblesort Sorted for alphabetical order, frequency, severity, severity

	with frequency, severity with alphabetical order and toxicity score Allowed users to compare the results of sorting algorithms in terminal (should be identical unless sorting for something else) Showed results in ascending or descending order Display top N words with bar chart Display top N most toxic words with bar chart Compared the analysis results of two files Added some extra statistics like wordPercent, which tells how much of a file is benign or toxic
Persistent Storage	Reload stopword/toxic/benign dictionaries dynamically. Save results in structured tabular format (CSV).
User Interface	Provide a menu-driven interface. Compare two files (benign vs toxic analysis). Add ASCII bar chart for top toxic words.
Error Handling & Robustness	Handle invalid input filenames. Prevent crashes for empty files. Handle unusual characters and numbers robustly. Provide clear error messages. Handle corrupted/large files gracefully. Add user-friendly recovery instructions.
Code Quality & Modularity	Use functions for modularity (e.g., loadFiles()). Contains commenting. Advanced modularization (separating into multiple .c and .h files). Each sorting function is separated for easier readability Use of data structures.
Bonus Features	Detect toxic or benign bigrams/trigrams/phrases. Compare two text files (e.g., benign vs toxic) and print the comparison.
Presentation / Demo	Notes / achievements summary (to be filled by students).
Report / Documentation	Write short implementation report. Include design choices. Link reflection to efficiency theory. Include justification with references to advanced CS concepts such as big O notation (time and space complexity for sorting algorithms)

Note: This may help to ease the marker on having an overview of your entire project in a glance.