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

Group Name	**25/25**				
Group Members					
Name 1	**Koh Shi Qi**		**ID 1**	**20811803**	
Name 2	**Fathima Sakinah Dil Fairaz**		**ID 2**	**20780335**	
Name 3	**Mariah Azmir Faizal**		**ID 3**	**20690412**	
Marks **Total (100)**	Program Functionalities (65)	Bonus (5)	Code Quality (10)	Presentation / Demo (10)	Report / Documentation (10)
Submission Date					

1. Introduction

This project implements the **Cyberbullying / Toxic Text Analyzer**, developed in **C**, to address the real-world cybersecurity challenge of detecting and quantifying harmful language in user-generated text datasets. The core objective was to demonstrate mastery of modular programming, dynamic memory management, and advanced algorithmic thinking.

The implementation includes several special and advanced features:

- **Algorithmic Depth:** Implementation and comparison of **three sorting algorithms** (Bubble Sort, Quick Sort, and Merge Sort). A unique feature allows users to **measure the execution time** and compare the performance of these algorithms.
- **Advanced Toxic Detection:** The analysis supports **multi-word toxic phrases** (bigrams/trigrams) and categorizes all toxic content into **Severity Levels** (Mild, Moderate, and Severe). The system uses a **multilingual lexicon** loaded dynamically from 12 language files.
- **Data Handling and Metrics:** Supports complex input formats, including analyzing **specific columns within CSV files**. Calculates advanced statistics such as the **Lexical Diversity Index** and **Average Sentence Length**.
- **Reporting Extensions:** Includes a dedicated menu for **Comparative Analysis** of metrics between two files (File A vs. File B) and exports detailed results into a **structured CSV report**. Visualization is provided via an **ASCII Bar Chart** for toxic word frequencies.

2. Key Design Choices

2.1 Data structures used

1. **Primary Arrays (Dynamic Allocation)**

```
1. char (*words)[50]
```

- 2D array sorting all extracted word from analysed text
- Fixed 50-char limit per word (sufficient for most languages)
- **Rationale:** Simple linear structure for fast O(1) random access during analysis

```
2. char (*uniqueWords)[50]
```

- Separate array for tracking unique words only
- Used for sorting operations and frequency analysis
- **Rationale:** Separation allows efficient unique word tracking without modifying main word list

2. **Fixed-Size Global Arrays**

```
1. stopwords[MAX_STOPWORDS][50]
```

- Common English words (210 entries)
- Linear search O(n) during tokenization
- **Rationale:** Simple lookup adequate for small stopword set

```
2. toxicWords[MAX_TOXIC][50]
```

- Single toxic terms (up to 50,000)
- Paired with `toxicFreq[MAX_TOXIC]` and `toxicSeverity[MAX_TOXIC]` arrays
- **Rationale:** Parallel arrays for frequency counting and severity classification

```
3. toxicPhrases[MAX_TOXIC_PHRASES][100]
```

- Multi-word phrases (up to 10,000)
- **Rationale:** Larger buffer (100 chars) needed for multi-word terms

3. **Structured Data**
   - `struct ToxicTerm` – Metadata container for each toxic word
     Rationale: Bundles related data for reporting and severity analysis
   - `struct AnalysisResult` – Complete analysis metrics for a file
     Rationale: Used for comparative analysis (File A vs File B)
   - `struct WordFreq` – Frequency tracking for sorting
     Rationale: Temporary structure for frequency-based sorting operations

Why Not Hash Tables or Linked Lists?

- **Arrays** preferred because:
  - Simpler C implementation without external libraries
  - Adequate performance for ~10K items with linear search
  - Predictable memory layout and allocation
- **Hash tables** would require custom implementation
- **Linked lists** would add pointer complexity and cache misses

2.2 File Handling Strategy

1. **Multi-File Format Support**
   A. **Text Files (.txt)**
      - Line-by-line reading with `fgets()`
      - Each line processed independently for words and sentences
      - **Rationale:** Simple streaming model

   B. **CSV Files (.csv)**
      - Header row detected and displayed to user
      - User selects specific columns for analysis
      - All selected columns concatenated with spaces
      - **Rationale:** Flexibility for dataset analysis (tweets, comments, etc.)
2. **Path Resolution Strategy**
   Search order:
   - Current directory → `./analysis\\` folder (Windows); **or**
   - Current directory → `./analysis/` folder (Unix)

   Rationale: Allows both local and organized file storage
   Implementation uses `strstr()` to detect if "analysis" already in path
3. **CSV Parsing**
   Custom parser `parseCSVLine()` handles:
   - Quoted fields (with embedded commas)
   - Field counting and extraction
   - Newline stripping

   Rationale: No external CSV library available; custom implementation ensures compatibility
4. **Multi-Language Toxic Dictionary Loading**
   Platform-Specific Directory Scanning:
   The system supports multi-language loading by dynamically scanning the `toxic_words/` folder using platform-specific directory handling.
   - **Rationale:** Native APIs prevent directory listing issues
   - **Fallback:** If `toxic_words/` folder missing, tries single `toxicwords.txt` file
   - **CSV Format** in each file: `word,severity` (e.g., `badword,3`)
5. **Dictionary Caching Strategy**
   - All dictionaries loaded once at program startup into global arrays
     - ~7,776 total toxic terms from 12 language files

        - ~10,000 multi-word toxic phrases (subset of loaded terms)
        - **Max capacity:** 50,000 toxic words, 10,000 phrases (configurable in `data_types.h`)
    - Reused throughout program execution
    - User can reload dictionaries via Option 3
    - **Rationale:** O(1) access during analysis; minimal memory overhead for loaded dictionaries

2.3 Tokenization Approach

1. **Multi-Stage Normalization Pipeline**

   Each token undergoes:

   ```
   1. normalizeCase()      → Convert to lowercase
   2. removePunctuation() → Keep only alphanumeric + spaces
   3. removeNonASCII()    → Strip non-ASCII characters
   4. stopword filter     → Exclude common words (> 210 stopwords)
   5. length filter       → Reject single-character tokens
   ```

2. **Tokenization Strategy**
    - **Delimiter set:** `" \t\n,.:;!?\"'"`
    - Uses standard C `strtok()` function
    - **Rationale:** Comprehensive delimiter coverage for most text

3. **Special Handling for CSV**
    - **Tokenization:** Skips unique word tracking during initial tokenization pass (stores all words sequentially)
    - **Post-processing:** Builds unique word list after tokenization completes
    - **Optimization:** For files > 50K words, caps unique word extraction at 10K with progress reporting
    - **Rationale:** CSV files often very large (100K+ rows); deferred unique tracking allows faster initial processing with early exit for large datasets

4. **Sentence Detection**
    - Detected by characters: `.`, `!`, `?`
    - Counted globally during tokenization
    - Used for: Average sentence length, lexical diversity calculations
    - **Rationale:** Simple heuristic suitable for most text

5. **Sentence Detection**

   Why lowercase normalization?
    - Toxic words often capitalized inconsistently (e.g., "BAD", "Bad", "bad")
    - Case-insensitive matching ensures detection
    - Implementation: `tolower()` with `(unsigned char)` casting for safety

2.4 Stopword Handling

1. **Stopword Set Design**
    - **210 common English stopwords** loaded from `stopwords.txt`
    - Examples: "the", "a", "is", "are", "in", "on", "to", etc.
    - Format: One word per line
    - **Rationale:** Standard set covers most common non-content-bearing English words

2. **Stopword Filtering Strategy**
    - Applied during **tokenization phase** (early filtering)
    - Linear search: O(n) per word, acceptable for 210 stopwords
    - Checked before storing word in both arrays

3. **Rationale**

   A. **Why filter stopwords?**

- Remove statistical noise (common words add no value)
- Reduce unique word count for memory efficiency
- Focus analysis on meaningful content
- Industry standard in text analysis

B. **Why not use hash table?**

- Small set (210 words); linear search is O(210) which is negligible
- Array access cache-friendly
- Simple implementation in pure C

4. **Performance Impact**

- Stopword check: O(n) × word_count
- Overall acceptable because stopword checking trivial compared to file I/O

2.5 Toxic Word Detection Strategy

1. Hierarchical Detection Approach

Phase 1: Multi-Word Phrases (Bigrams & Trigrams)

```
// First, try to match 2-word phrases (bigrams)
"word1 word2" → Check against toxicPhrases[]
// Then, try to match 3-word phrases (trigrams)
"word1 word2 word3" → Check against toxicPhrases[]
```

Phase 2: Single Words

- If no phrase matched, check individual word against `toxicWords[]`

Phase 3: Severity Lookup

- Match found → lookup severity in `toxicSeverity[]` array
- Track frequency in `toxicFreq[]` array

2. Rationale for Phrase Detection

- **Why phrases?**
  Context matters: "go to hell" is different from "go" + "to" + "hell"
- **Why bigrams before trigrams?**
  Bigrams more common; early exit saves cycles
- **Why before single words?**
  Prevent double-counting (e.g., "hell" in phrase)

3. Case-Insensitive Matching

- Uses `stringEqualsIgnoreCase()` helper function
- Normalizes both dictionary term and matched word to lowercase
- **Rationale:** Handles user-generated content variations

4. Frequency Counting

```
// First, try to match 2-word phrases (bigrams)
"word1 word2" → Check against toxicPhrases[]
```

- Used for: Toxicity ratio, bar charts, frequency reports

5. Severity Levels

- **1 (Mild):** Vulgar/crude language
- **2 (Moderate):** Insults/derogatory terms
- **3 (Severe):** Hate speech/slurs
- **Rationale:** Categorize toxicity for detailed analysis

6. Performance Considerations

- **Time Complexity:** O(n × m) where n = words, m = toxic terms (~7.8K loaded, ~50K capacity max)
- **For CSV files:** Progress reporting every 10K words to indicate responsiveness

- **Optimization:** Phrase detection skipped if no bigram/trigram found

2.6 Sorting Algorithms

1. **Three Implementations**
   - A. **Bubble Sort ($O(n^2)$)**
     - **Time:** $O(n^2)$ worst/average case
     - **Space:** O(1) - in-place
     - **Stability:** Stable (preserves equal elements' order)
     - **Use Case:** Teaching reference; acceptable for < 1K items
     - **Justification:** Simple, easy to understand, demonstrates fundamentals
   - B. **Quick Sort (O(n log n) avg, $O(n^2)$ worst)**
     - **Time:** O(n log n) average, $O(n^2)$ worst (pivot always min/max)
     - **Space:** O(log n) recursion stack
     - **Stability:** Unstable (doesn't preserve relative order)
     - **Use Case:** Fast average performance; ideal for typical datasets
     - **Justification:** Most commonly used in practice; excellent typical-case performance
   - C. **Merge Sort (O(n log n) guaranteed)**
     - **Time:** O(n log n) best/average/worst (guaranteed)
     - **Space:** O(n) - temporary array needed
     - **Stability:** Stable (equal elements maintain order)
     - **Use Case:** Guaranteed performance for large datasets
     - **Justification:** Consistent O(n log n) even for worst-case inputs
2. **Sorting Criterion Options**

   User can sort by:
   - **Alphabetical (A-Z)** - Alphabetic string comparison
   - **Frequency** - Most to least common words found
   - **Toxicity** - Highest to lowest toxic word counts

   Each criterion works with all three algorithms.
3. **Performance Comparison Feature**
   - **Option 7** runs all three algorithms on same dataset
   - Measures execution time with `clock()`
   - Verifies all produce identical results

2.7 Menu-Driven User Interface Design

1. **Architecture and Structure**
   - A. **Core Mechanism**

     The UI operates on an infinite loop in `main()`, using a switch statement to dispatch user selections to the appropriate functions
   - B. **Menu Grouping**

     The main menu is structured hierarchically into four logical sections for easy navigation:
     - **File Operations** (e.g., Load, Add/Manage Toxic Words, Reload Dictionaries)
     - **Analysis & Reports** (e.g., Bar Charts, Word Frequencies, Severity Breakdown)
     - **Advanced** (e.g., Compare two files, Export CSV Report)
     - **Exit** (Option 0)
   - C. **Sub-Menu Pattern**

     Sequential sub-menus are used, such as the three-step selection process for sorting (Criterion Algorithm Display) to guide the user
   - D. **Aesthetics:**

     ANSI color codes (from `colors.h`) are utilized to enhance readability, providing distinct colors for headers, menu options, and prompts

2. **Input Validation and Robustness**
   - A. **Buffer Overflow Prevention**
     Input collection (for menu choices or filenames) uses the robust C function `fgets()` instead of `scanf()` to prevent buffer overflow issues
   - B. **Input Validation Helpers**
     Dedicated helper functions are implemented:
     - `getValidMenuChoice()`: Validates that input matches a whitelist of allowed characters (e.g., '0' to '9', 'A', 'a').
     - `getValidIntInput()`: Validates that numerical input falls within a specified range (min/max).

     These helpers include error messages and retry logic to ensure stability
3. **User Feedback**
   - A. **Error Reporting:**
     File loading includes comprehensive error messages that detail the multiple locations checked for the file (current directory, `./analysis/`, and `./analysis\`).
   - B. **Progress Indicators:**
     Status messages and word count updates are provided during long-running operations (like large file processing or toxicity analysis) to confirm the program is running.
4. **Menu Design Principles Applied**
   1. **Grouping** - Related options grouped (File Ops, Analysis, Advanced)
   2. **Naming** - Clear, action-oriented option descriptions
   3. **Ordering** - Logical workflow (Load → Analyze → Report → Compare)
   4. **Confirmation** - No destructive operations without preview
   5. **Help** - Inline error messages with solutions
   6. **Accessibility** - Single-character input, numbered options
   7. **Feedback** - Every operation confirms completion

3. Challenges Faced and Lessons Learned

3.1 Challenges Encountered

1. **Efficiency Bottleneck (O(n^2) Operations):**
   - **Unique Word Extraction:** Identifying unique words from the input list required comparing every new word against the existing unique list, resulting in an expensive time complexity.
   - **Observed Impact:** This created noticeable delays (2–5 seconds) when processing large datasets (> 50,000 words), despite implementing optimizations like capping unique words at 10,000.
   - **Toxic Lookups:** Checking every word against the **~7,800 loaded toxic terms** (from 12 files) relies on linear search, making toxicity detection the most critical performance bottleneck, operating at O(n × m) complexity.
2. **Multilingual and Encoding Limitations (UTF-8):**
   - The C language's reliance on ASCII caused problems with our Multi-Language Support. While we successfully loaded dictionary terms for 12 languages (including Arabic and Chinese), our text processing pipeline was forced to strip non-ASCII (UTF-8) characters early. The consequence is that the non-Latin portions of the dictionaries are unusable, meaning the analyzer only reliably handles ASCII-like text.

3. **Memory and Robustness Issues:**
   - The use of large, dynamically allocated arrays for storing up to 1,000,000 words meant the program requires approximately 58–60 MB of memory at startup. If memory allocation fails—especially during recursive operations like Merge Sort which requires a temporary array—the program has limited ability to inform the user or recover gracefully, often leading to a silent failure.
4. **String Handling and Buffer Limitations:**
   - To maintain simplicity and prevent buffer overflows, fixed limits were necessary. Words were capped at 50 characters, and CSV fields were capped at 256 characters. This design choice risks silently truncating exceptionally long words or text fields, leading to data loss or skewed analysis. Furthermore, our custom CSV parser does not handle complex RFC 4180 edge cases, such as escaped quotes or embedded newlines.

3.2 Lessons Learned and Critical Reflection

1. **The Priority of Data Structures over Algorithms:**
   - The most important lesson learned was that **algorithmic efficiency is only as strong as the underlying data structure**. Implementing three advanced sorting algorithms (Quick Sort and Merge Sort are O(n log n)) showed mastery of sorting, but the slowness observed during analysis highlighted that dictionary lookups (the search problem) were the true bottleneck. For a production system, a hash table would be essential for O(1) average lookup time, proving that the **trade-off between implementation simplicity (arrays) and production efficiency (hash tables)** was a major design decision.
2. **C Development Complexity:**
   - The project underscored the complexity of low-level programming. Managing dynamic memory allocation and subsequent freeing `(free(words))` requires constant vigilance. The challenges faced with cross-platform directory scanning (Windows vs. Unix APIs) and string encoding issues emphasize the difficulty of writing portable and robust C code.
3. **Algorithmic Justification (Quick Sort Pivot):**
   - We observed that our Quick Sort implementation, which uses a last-element pivot, suffers poor performance O(n^2) if the data is already nearly sorted. This provides strong practical justification for implementing a more sophisticated pivot selection strategy, like the median-of-three, to ensure consistent performance, fulfilling the requirement for a critical reflection on algorithm choices.
4. **Design for Scalability:**
   - Since the design requires storing **all words in memory** to enable features like frequency counting and sorting, we learned that alternative, streaming-based approaches would be necessary for files exceeding the 1M word limit, demonstrating an awareness of scalability constraints.

Appendices

Appendix A: ReadMe

GitHub Link & Video Link

GitHub Repository: COMP1028_25-25_Toxic-Text-Analyzer
Video Presentation Link: COMP1028_25-25_Video

Appendix B: Marking Scheme Summary

Criteria	Description / Notes
Text Input & File Processing	Supports loading and parsing **CSV files** with **column selection**. Configured to handle **very large files** (up to 1,000,000 words). Robust file path checking in 3 locations (`./`, `./analysis/`, `./analysis\`)
Tokenization & Word Analysis	Calculates **Lexical Diversity Index** and **Average Sentence Length** (by counting '.!?' characters). Implements multi-stage cleaning (lowercase, punctuation / non-ASCII removal, stopword filtering)
Toxic Word Detection	Detects **multi-word toxic phrases** (bigrams/trigrams) hierarchically. Terms are categorized and reported by **Severity 1, 2, and 3**. Supports **dictionary expansion** via a menu option (`addToxicWordsMenu()`) across 12 languages
Sorting & Reporting	Implemented **three sorting algorithms**: **Bubble Sort**, **Quick Sort**, and **Merge Sort**. Features an algorithm comparison menu (Option 7) that **measures and reports execution time** for all three sorts.
Persistent Storage	Provides **structured output** by exporting metrics (ratios, diversity, counts) to a dedicated **CSV report** (analysis_report.csv). Also saves standard text report.
User Interface	Implemented a **Comparative Analysis Menu** (Option 8) for side-by-side comparison of two files. Displays an **ASCII Bar Chart** visualization of toxic word frequencies. Uses ANSI color codes for enhanced UI.
Error Handling & Robustness	Robust input validation (`fgets()`, `getValidIntInput()`) to prevent crashes and ensure stability. Gracefully handles missing files with clear recovery instructions. Checks for **memory allocation failure** on startup.
Code Quality & Modularity	**Advanced modularization** using separate `.c` and `.h` files (e.g., `sorting.c`, `file_handler.c`, `reporting.c`). Extensive use of **structures** (`AnalysisResult`, `WordFreq`) for clean data management.
Bonus Features	Algorithmic Performance Benchmarking (timing sort speed); Comparative Analysis (File A vs. File B); Toxic Phrase Detection; CSV Column Selection; ASCII Bar Chart.
Presentation / Demo	Prepared a 6-minute video script focused on demonstrating all features and providing critical justification for algorithmic choices (e.g., Quick Sort vs. Merge Sort).
Report / Documentation	Report includes **critical reflection** on limitations, specifically addressing the **unique word extraction bottleneck** and justifying the trade-off of using array structures over hash tables for educational purposes.

Appendix C: Data Sources and Citations

The functionality of the Toxic Text Analyzer relies on extensive multilingual lexicons and external datasets utilized for testing, comparative analysis, and building the toxic word dictionaries.

Dataset Description	**URL Source**
Jigsaw Toxic Comment Classification Challenge (English text data for general toxicity analysis, used for benchmarking).	https://huggingface.co/datasets/thesofakillers/jigsaw-toxic-comment-classification-challenge/blob/main/train.csv
Cyberbullying Classification Dataset (General corpus of categorized tweets, used primarily for CSV processing tests).	https://www.kaggle.com/datasets/andrewmvd/cyberbullying-classification?resource=download
Hate Malay Dataset (Malay language dataset for testing multilingual support).	https://github.com/MaityKrishanu/Hate_Malay/blob/master/HateMalay%20Dataset.csv
ToxiCN Dataset (Chinese language data, used for testing non-Latin script dictionary effectiveness).	https://huggingface.co/datasets/JunyuLu/ToxiCN/blob/main/ToxiCN_1.0.csv
L-HSAB Dataset (First Arabic Levantine Hate Speech Dataset, used for testing Arabic language analysis).	https://github.com/Hala-Mulki/L-HSAB-First-Arabic-Levantine-HateSpeech-Dataset/blob/master/Dataset/L-HSAB#L3
FTR Dataset (Dataset used for general text classification and report validation).	https://github.com/NataliaVanetik/FTR-dataset/blob/main/FTR_new_labels.csv
Refined N-gram Dictionary (Used to build the core multi-word phrase dictionary for detection, a key Distinction feature).	https://github.com/t-davidson/hate-speech-and-offensive-language/blob/master/lexicons/refined_ngram_dict.csv
General Hate Speech and Offensive Language Lexicons (Reference repository for toxic word lists and dictionary sources).	https://github.com/t-davidson/hate-speech-and-offensive-language/tree/master/lexicons
Labeled Data for Hate Speech Classification (General English/Social Media toxic word list source, including severity categories).	https://github.com/t-davidson/hate-speech-and-offensive-language/blob/master/data/labeled_data.csv
Hinglish Profanity List (Used to supplement and verify the Hindi dictionary for toxic content).	https://github.com/neerajvashistha/online-hate-speech-recog/blob/master/data/hi/Hinglish-Offensive-Text-Classification/Hinglish_Profanity_List.csv
NLLB-200 Toxic Words List (TWL)	The core multilingual toxic word list (over 7,700 terms across 12 languages) used in the project is attributed to the NLLB-200 TWL project by Facebook Research, as documented in the README.