CS 201 - Data Structures II:
Assignment #4

Muhammad Mobeen Movania (L1),
Syeda Saleha Raza (L2),
Faisal Alvi (L3, L4),
Abdullah Zafar (L5).

Due on April 26, 2024, 11.59pm

Student 1 Name, ID **Student 2 Name, ID**

Instructions

This assignment document consists of two problems.

- Problem 1 is a programming based question which requires implementation. It must be submitted by pushing all your code files to the Github repository. This problem is worth 40 points.
- Problem 2 is a theoretical question which requires analysis. It should be completed and submitted within this document as a pdf on Canvas. This problem is worth 20 points.

Problem 1

(40 points) [**Creating a Ranked Retrieval System**]
In this assignment you will be implementing an inverted index on a corpus of resumes taken from Resume Dataset - Kaggle and creating a story based on the data.

1. Understanding the Data Format

The data, available on the canvas assignment page, is divided into two folders called "text" and "html", containing text files and HTML files respectively. Each text file contains simplified/cleaned text [1] that corresponds to an HTML file of the same name containing the original resume text. You need to use the text files to create an inverted index and perform queries on it. The HTML files are given to help you explore the data better.

Required Tasks

None.

2. Create a Positional Index

Create a positional index of the following abstract format [2] using all the data from the "text" folder. Table 1 shows the abstract format of an entry and some of its values in the positional index.

Term, Total Frequency	[DocID, Frequency in Doc: [pos1, pos2, ...], weight=1 ...]
⋮	⋮
inspired, 16	[30, 1: [3036], 1 51, 1: [1452], 1 1440, 2: [2697, 5123], 1 ⋮]
⋮	⋮

Table 1: Entry for the term "inspired", which appears a total of 16 times in all documents combined. The Doc. ID corresponds to the number appended to the last underscore in the filename. A default weight of 1 is added to each document entry as a placeholder to be replaced with a weight when implementing ranked retrieval.

A positional index allows proximity and phrase search of variable length. You will use the positional index to create a search story that incorporates both proximity and phrase search. In order to create the required index, you need to consider:

1. *Text Extraction:* The entire data needs to be tokenized over whitespace. Each token will be treated as a term. The terms will probably need to be collated in some form before being added to the inverted index.

[1] The original text has been pre-processed to remove uppercase, single characters, punctuation, non-alphabetic characters, stop words, and then lemmatized.

[2] The implementation-level format may differ, as required, from the abstract format. For example, a hash table may use just the terms as keys, and all other attributes (including Total Frequency) as values. However, the index saved as JSON should be in its abstract format.

2. *Choice of Data Structure:* The two common choices are hash tables and tries. Deciding whether to use one or the other depends on the specific requirements of your application. First understand all the requirements of this assignment, and then do your own research before deciding on the appropriate data structure to use. You are free to use any data structure to implement the inverted index for the purposes of grading.

Required Tasks:

Create a class called `InvertedIndex` and write the functions `InvertedIndex, getSize` and `saveToJson` with the following specifications:

```
class InvertedIndex {
    public:
        InvertedIndex(const std::string& folderPath); //constructs an
            Inverted Index according to the given specifications.
        int getSize(); // returns the number of unique terms
        void saveToJson(const std::string& fileName); //saves entire Index
             as a JSON file called <fileName>.json
};
```

A JSON file, available on the canvas assignment page, has been provided for comparison.

3. Add Boolean and Proximity Search Functionality

Two types of boolean queries and two types of proximity queries should be supported by the positional index. These are:

1. t_1 AND t_2: postings that match both terms t_1 and t_2.
2. t_1 OR t_2: postings that match either terms t_1 or t_2.
3. t_1 BEFORE t_2: postings where term t_1 appears immediately before term t_2.
4. t_1 NEAR t_2 n: postings where the first letters of terms t_1 and t_2 appear within n (character) positions of each other. E.g. *effective* NEAR *skills* 26 matches the document containing "effective problem solving skills" but *effective* NEAR *skills* 25 does not.

Query Format

Queries will be given as strings, and can be represented recursively. A query can be:

1. Standalone term: "<t>", where "<t>" represents the string representation of term t. E.g. "*inspired*".
2. AND: "(<t_1> `AND` <t_2>)". E.g. "(*inspired* `AND` *motivated*)".
3. OR: "(<t_1> `OR` <t_2>)". E.g. "(*inspired* `OR` *motivated*)".
4. BEFORE: "(<t_1> `BEFORE` <t_2>)". E.g. "(*computer* `BEFORE` *science*)"
5. NEAR: "<t_1> `NEAR` <t_2> <n>", where "<n>" is the string representation of the number n. E.g. "(*leadership* `NEAR` *teamwork* 20)"
6. COMPOUND AND: "<q_1> `AND` <q_2>", where q_1 and q_2 are queries.
   E.g."((*inspired* `OR` *motivated*) `AND` *graduate*)".
7. COMPOUND OR: "<q_1> `OR` <q_2>".
   E.g. "((*leadership* `NEAR` *teamwork* 20) `OR` (*leadership* `NEAR` *collaboration* 20))".

Required Tasks

Write a struct, `Posting`, of your own specification except that it must contain a `docId` attribute as follows:

```
struct Posting {
    std::string docId;
    //add more attributes as required
}
```

Write a function, `retrieve`, within the `InvertedIndex` class with the following specification:

```
class InvertedIndex {
    public:
        std::vector<Posting> retrieve (const std::string& query) //takes a
            query string and returns a vector of postings matching the
           query.
};
```

4. Add Ranked Retrieval Functionality

The positional index needs to support ranked retrieval, i.e. retrieved documents should be scored (weighted) and sorted in order of highest score to lowest score. The Term Frequency (TF) and Term Frequency-Inverse Document Frequency (TF-IDF) are two commonly used metrics to evaluate how important a term is, but they differ significantly in how they measure this importance. You are only required to implement the TF-IDF metric on your queries. You may decide to use a different metric, such as TF, in the next part of the assignment.

Required Tasks

Implement the TF-IDF metric within the `retrieve` function. You should add the tf-idf weight of each query term present in the document to compute the TF-IDF score for that document. Refer to this link for the definition of the TF-IDF score.

```
class InvertedIndex {
    public:
        std::vector<Posting> retrieve (const std::string& query) //updated
            : takes a query string and returns a vector of postings
            matching the query sorted by score, from highest score to
            lowest score.
};
```

Grading Criteria

The rubric for this homework is as follows:

- 15 points: Section 2 - Create a Positional Index
- 15 points: Section 3 - Add Boolean and Proximity Search Functionality
- 10 points: Section 4 - Add Ranked Retrieval Functionality

Penalties

- (-5) Code compiles with warnings.
- (-5) GitHub repository does not follow appropriate structure. All the code files should be in the code folder and output files in the output folder.

Compilation Guidelines: Before proceeding with submission, kindly verify that your code compiles utilizing the **C++17** standard.

Problem 2

(20 points) [**Analysis: Create a Search Story**]
Once you have implemented the required functionality of the positional index, you need to create your own search scenario. Think of the search scenario as a story that requires the following steps:

- *Explore the dataset:* Explore the dataset for categories or patterns using the search functionality you have implemented. Remember to search for compound phrases, not just standalone terms. HTML versions of resumes have been provided to help you explore the dataset further.
- *Identify an information need:* Once you have explored the dataset, identify a meaningful information need that you will try to accomplish using your inverted index. Remember that an "'information need' is the topic which the user desires to know more of, and is different from a query, which is what the user conveys to the computer in an attempt to communicate the information need"[3]. E.g. You may want resumes of computer science graduates who know certain frameworks.
- *Choose queries to communicate the information need:* Try out different queries, simple and compound, to retrieve documents to fulfill the intended information need. Distill your choice down to a few meaningful queries.
- *Choose the scoring metric:* Choose a scoring metric that will suit the data and information need. Each metric differs in how it measures the importance of terms; you may choose TF, TF-IDF, or come up with a new scoring metric.
- *Analyze Results:* Observe patterns in the retrieved documents. How effective is your search in terms of precision and recall? What shortcomings does your search have?

Required Tasks

1. Write a caption that succinctly captures your information need. The caption should be no more than a phrase or sentence.
2. Write 3 distinct queries that can be used in conjunction to retrieve documents. Each query, simple or compound, should be written in the format accepted by the retrieve function.
3. Justify, briefly, the scoring metric used, and define any new metrics if used.
4. Analyze, briefly, the effectiveness of your search, commenting on the precision, recall and any other factors relevant to your search.

Due date: April 26, 2024, 11.59pm

Acknowledgement

We thank Asad Tariq for all the help in designing this assessment.

[3] https://nlp.stanford.edu/IR-book/html/htmledition/an-example-information-retrieval-problem-1.html