# CSE 102 Programming Assignment 5

## DUE

December 3, 2024, 23:55

## Description

You are going to write a complete C program which implements the following functionality:

- Your program will read `n` input files:

```
language_1.txt
language_2.txt
language_3.txt
language_4.txt
language_5.txt
.
.
.
language_n_1.txt
language_n.txt
```

- `n` is between `1` and `100`.
- In addition to the `n` language files, there will be another file to read: `language_x.txt`.
- Each file contains text in a specific language. All files contain only english lowercase characters and whitespace. Text files will include the following characters:

```
'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n'
'o' 'p' 'q' 'r' 's' 't' 'u' 'v' 'w' 'x' 'y' 'z' ' '
```

- Your program will evaluate the dissimilarity scores of language pairs:

```
(language_x, language_1)
(language_x, language_2)
(language_x, language_3)
(language_x, language_4)
(language_x, language_5)
.
.
.
(language_x, language_n_1.txt)
(language_x, language_n.txt)
```

- First of all, calculate bi-gram frequencies for each language. A bi-gram is defined as follows: For a given sequence, each unique pairing of successive letters is a bi-gram. For example: for the word `" adana "` bi-grams are defined to be `" a", "ad", "da", "an", "na", "a "`. Beware: If there is a space before or after a character you will still be dealing with bi-grams. Each bi-gram has exactly two elements which are either characters or space. In order to calculate the frequency of a particular bi-gram(lets say bi-gram `"ad"`) you have to count all the bi-grams in a given text and for this bi-gram calculate the ratio `(# of "ad")/(total # of all bi-grams)`
- Given all the frequencies, dissimilarity score is calculated as follows:

$$dissimilarity(language_a, language_b) = \sum_i |f_a^i - f_b^i| \qquad (1)$$

- Here $f_a^i$ represents the frequency of $i^{th}$ bi-gram for the language $a$. If $c_a^i$ is the count of $i^{th}$ bi-gram in $language_a$, then;

$$f_a^i = c_a^i / (\sum_j c_a^j) \tag{2}$$

- After evaluating dissimilarities, your program will print all the dissimilarity values. Print:

```
dissimilarity(language_x, language_1)
dissimilarity(language_x, language_2)
dissimilarity(language_x, language_3)
dissimilarity(language_x, language_4)
dissimilarity(language_x, language_5)
.
.
.
dissimilarity(language_x, language_n_1.txt)
dissimilarity(language_x, language_n.txt)
```

- **You will print just the numbers(i.e. the dissimilarity scores). Don't print anything extra.**

## Remarks:

- **YOU CANNOT STORE ALL THE CONTENTS OF THE FILES IN THE MEMORY**
- **YOU CANNOT REPEATEDLY READ THE SAME FILE**
- **THE FILES CAN BE VERY LARGE**
- **Test your code with very large input files.**
- Efficiency of your implementation is important. Don't permanently store the content if you are not going to re-use it. Don't repeatedly calculate any output, if the input is the same.
  - Frequencies of bi-grams in language_x should be calculated only once. Store them in an array and re-use the calculated frequencies in order to calculate the dissimilarity scores.
  - Do not read the files multiple times.
- text files can include multiple concatenating whitespace. For example:

```
Here         we    are using a user defined    recursive
```

- Two adjacent whitespace do not create a bi-gram.
- Input files can be multi-line text files.
- There isn't any limit on the size of input files. Your program should work regardless of the size of the input.
- Be careful with the size of the array you allocate in program stack. Large arrays may not fit in program stack(stack size may be smaller on the test machine) and your program crashes.
- Make sure you can read input files with or without a tailing newline at the end. (If you are using a windows machine, newline is `CRLF`, on unix it is `LF`). You can alter this using advanced editors (i.e. Visual Studio Code). Test your code for every possible combination.
- Do not print anything other than the expected output.
- You cannot use anything which is not covered in class.
- Do not submit any of the files you used for testing.

## Hints:

- Bi-gram types do not depend on the input file. There are finite number of possibilities. Given all the lowercase english characters and a space, you can generate all the possible bi-grams.
- Do not try to store all the content of the file in the memory. Counting is possible without storing all of the text.
- You don't need to parse words.

## Turn in:

- Source code of a complete C program. Name of the file should be in this format: `<full_name>_PA5.c`.
- Example: `john_doe_PA5.c`. Please do not use any Turkish special characters.
- You don't need to use an IDE for this assignment. Your code will be compiled and run in a command window.
- Your code will be compiled and tested on a Linux machine(Ubuntu). GCC will be used.
- Make sure you don't get compile errors when you issue this command : `gcc <full_name>_PA5.c`.
- A script will be used in order to check the correctness of your results. So, be careful not to violate the expected output format.
- Provide comments unless you are not interested in partial credit. (If I cannot easily understand your design, you may loose points.)
- You may not get full credit if your implementation contradicts with the statements in this document.

## Late Submission

- Late submission is NOT accepted.

## Grading (Tentative)

- `Max Grade` : `100`.

All of the followings are possible deductions from `Max Grade`.

- No submission: `-100`.
- Compile errors: `-100`.
- Irrelevant code: `-100`.
- Major parts are missing: `-100`.
- Unnecessarily long code: `-30`.
- Using language elements and libraries which are not allowed: `-100`.
- Not caring about the structure and efficiency: `-30`. (avoid using hard-coded values).
- Significant number of compiler warnings: `-10`.
- Not commented enough: `-5`. (Comments are in English).
- Source code encoding is not `UTF-8` and characters are not properly displayed: `-5`. (You can use 'Visual Studio Code', 'Sublime Text', 'Atom' etc… Check the character encoding of your text editor and set it to `UTF-8` ).
- Fails at reading the file: `-30`.
- Fails during file write: `-30`.
- Similarity score is wrong: `-20`.
- Output format is wrong: `-30`. (Be careful with spacing)
- Infinite loop: `-90`.
- Prints anything extra: `-30`.
- Unwanted chars and spaces in the output: `-30`.
- Submission includes files other than the expected: `-10`.
- Submission does not follow the file naming convention: `-10`.
- Sharing or inheriting code: `-200`.
- IF YOU DON'T FOLLOW THE FILE NAMING CONVENTIONS YOU WILL GET `0`.

Note: Some of these items are not independent. So, you cannot expect isolation of many of them. For example, if you cannot read input file correctly, you will fail to produce the correct output file. Partial grading is not guaranteed.