Name:

|  |  |  |
| --- | --- | --- |
| Problem | Points | Score |
| Level 0 | 0 |  |
| Level 1 | 50 |  |
| Level 2 | 60 |  |
| Level 3 | 70 |  |
| Level 4 | 80 |  |
| Level 5 | 90 |  |
| Level 6 | 100 |  |

Notes:

1. For this exam you are allowed to open a terminal window on your computer, you are allowed to web surf with Google, but you cannot use online chat or other interactive services.

**Deliverables:**

1. *Source Code:* a compressed tar file named lastname\_firstname\_e04.tar.gz containing all your code (e.g., Makefile, header file, main driver program, library containing function implementations).

Send this as an attachment in an email. The subject line must read: “ECE 1111: Level #” where “#” indicates the level of completion of your code. Your code must satisfy the requirements of that level or you will receive the lowest score (e.g., 50). “Level #” essentially means you are applying for that grade (see the front page) because your code meets those requirements.

Again, please not that if you fail to meet the “Level #” requirements, you get the lowest score. Only apply for a level if you are 100% sure you meet the requirements for that level.

**Description:**

The directory */data/courses/ece\_1111/2018\_fall/exams/final* contains a set of files. Read these files into memory and count the number of lines that have three or more words in common. Your main program should support this interface:

**nedc\_999\_[1]: myprog.exe -n N f1.txt f2.txt ... f9999.txt**

**analyzing data...**

**match no. 001:**

**line: see Mary run fast**

**line: see John run fast**

**match no. 002:**

**...**

**total number of matching lines = 27**

Two lines are considered to match if they have N or more words in common, where N is specified by the “-n” option (e.g., “-n N”). The matching algorithm must be case-insensitive. The data above is just an example. You are to check all lines in all files against all other lines in the files. If there are 10 total lines in all the files, this means you will do 10 x 10 – 10 comparisons because you will ignore the case where a line is matched to itself.

You are allowed to load all the lines from all the files into memory, but you must do this by iterating over the files specified on the command line.

Note you must support the arguments “-n #”, where “#” is the number of words that match, and “-help”, which displays a help message. Either can appear anywhere on the line. The help message should describe your interface.

You must implement this program following the structure described below.

**Requirements:**

**Level 0 (0 pts):** Not follow these requirements (yes, I am serious!).

**Level 1 (50 pts):** Show up for final exam and submit no code or code that does not compile. Note that if you choose to do so, you can submit a blank email with the proper subject line and leave. Of course, 50 is a failing grade on this exam.

**Level 2 (60 pts):** Demonstrate that you parse the command line arguments properly. Display a help message when “-help” is specified anywhere on the command line. Demonstrate that you read the value of the “-n” option correctly by printing out its value. Iterate over the files specified from the command line and print the name of each file to stdout, one at a time. (Don’t just echo the command line, use a formatted print statement.)

**Level 3 (70 pts):** Demonstrate that you can open all the files and read the lines into memory. Print the lines of the file to stdout \*AFTER\* you have read them all into memory. You can use whatever data structure you want as long as it is sized to the number of total lines. For example, if you use an array of character strings, it must be sized to the total number of lines, not some arbitrarily large value. You should print out the total number of lines loaded as well as each line. You must strip any trailing linefeeds. Your code should work for an arbitrarily large number of files ­– I will test it on a different database that has 30,000 files in it.

**Level 4 (80 pts):** Loop over all lines and compare them to all other lines by implementing a function called “compare\_lines”. For this level, this function always returns true, so your program simply prints every line as though it matched every other line. If you read a total of 10 lines, you will print 10 x 10 - 10 = 90 matches, because you will ignore the case of a line matched against itself. The function compare\_lines must take two character strings as arguments, along with the number of words that need to match, and returns true or false depending on whether they match. For this level, stub this function out and have it always return true.

**Level 5 (90 pts):** Demonstrate your compare\_lines function calls a function parse\_words. The function parse\_words splits each line into an array of character strings containing the words. You can assume there are no more than 999 words per line. For this level, you must print the list of words returned from each parse, so we know you are able to accurately tokenize the line.

**Level 6 (100 pts):** Implement a function called compare\_words that takes two lists of words as input, the number of words on each list, and the number of words required to match. This function returns true or false depending on whether N words match. Accurately print out only the lines that have N or more matching words. I should be able to vary “-n N” and see the results change accordingly. For this level, your printout must match the format shown at the beginning of this document.