EECS 4/5760 – Computer Security – Fall 2023

Programming Project 1 – Due Sunday, September 17 at 11:59 PM

We will have another programming project related to cryptography later in the semester, but this first one is designed to give you an easy introduction to analyzing frequency distributions in a file, and let you flex your coding muscles a bit.

You will be provided three files – Shakespeare.TXT, Shakespeare.ECB, and Shakespeare.CBC. The first is a plaintext file containing the concatenated text of all of Shakespeare's plays. The file is approximately 5 MB. I will also provide some shorter files with which you can test. The .ECB file is the TXT file encrypted using DES in ECB mode, and the .CBC file is the TXT file encrypted with one of the "better" modes I referred to in class (Cipher Block Chaining mode, to be precise, though that's not important for your code).

Your task will be to write a command-line program (in C/C++) to read a file (the lone command-line argument to **`main()`**), and perform a frequency analysis of the file. You must write your code in Visual Studio 2022 for Windows. If you don't have VS, you can get the (free) Community edition from the Microsoft website: https://visualstudio.microsoft.com/. YOUR CODE MAY NOT PROMPT THE USER FOR ANY INPUT; IT MUST TAKE THE FILENAME FROM THE COMMAND LINE AND RUN. IF YOU DON'T UNDERSTAND EXACTLY WHAT THIS MEANS, SEEK CLARIFICATION.

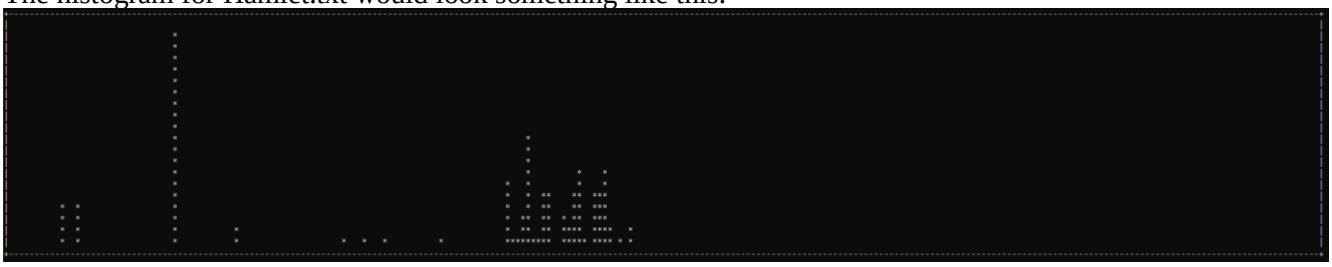
Your program is to output a 256-position vertical bar graph (using individual characters as "pixels") and the counts for how many times each possible byte value appears in the file, scaled such that the height of the graph corresponds to the largest value. Each "pixel" will be either a space or an asterisk.

For example, if you find that the character that appears most often is the space (and in the plaintext version, it WILL be), and there are 500,000 spaces in the file, then the height of your graph will represent 500,000, with smaller values scaled in-between (if some other character appears 250,000 times, its bar will be half as tall). Always "round up", such that even an occurrence of 1 in a 5 MB file will appear with a single *. Note that in a text file, the counts of ALL bytes whose values are > 127 will be zero, so don't be surprised by that.

Your output will be sent to the terminal (**`cout`**).

Your histogram will have a crude border, with vertical bars ('|') for the left and right borders, hyphens ('-') for the top and bottom borders, and plus signs ('+') in the corners.

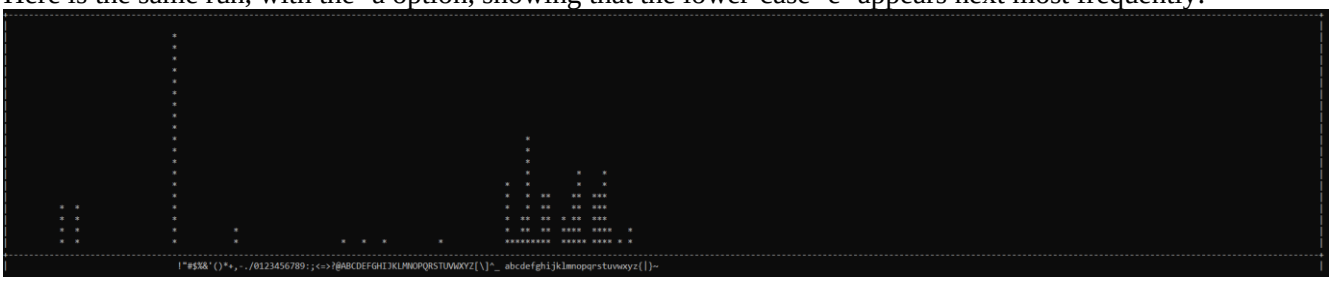
The histogram for Hamlet.txt would look something like this:



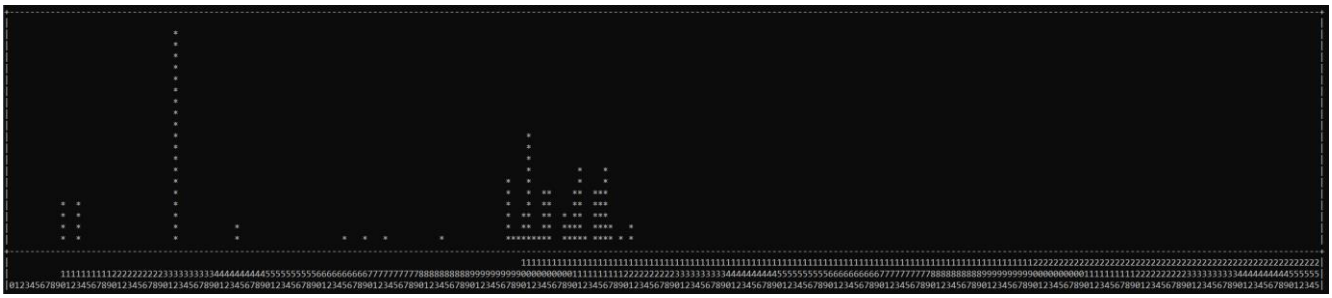
Your program will support five options:

`-a` shows ASCII characters (if printable) along the X-axis
`-n` shows 0-255 along the X-axis (in three rows)
`-rnn` sets the height of the histogram to **`nn`** rows (default is 20). Note: no space between **`r`** and the number
`-l` Display the Y-axis logarithmically (helps flatten huge spikes)

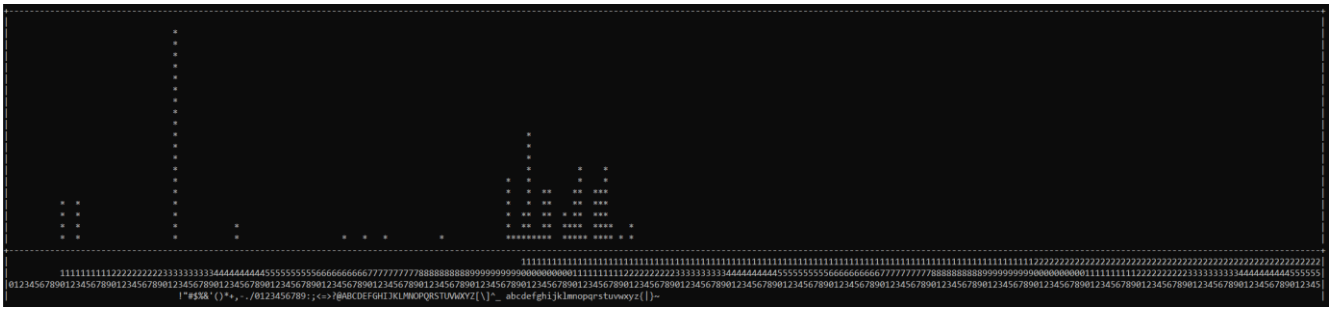
Here is the same run, with the -a option, showing that the lower-case ‘e’ appears next most frequently:



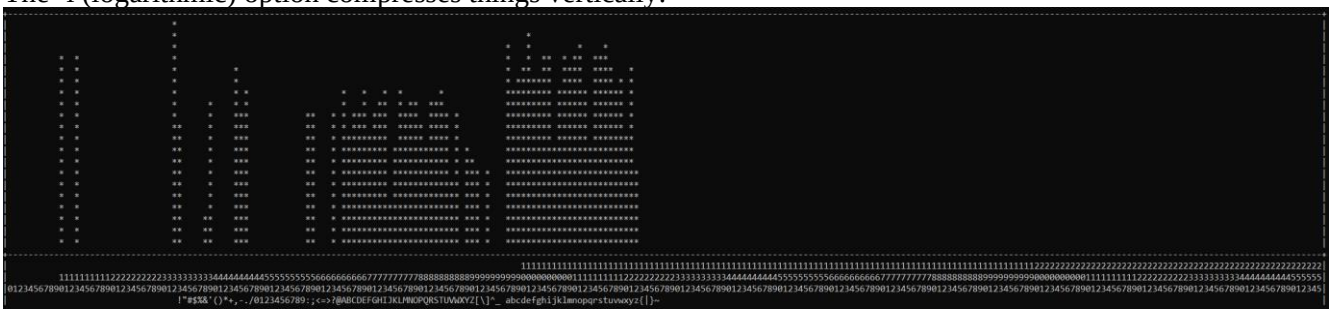
This example shows the use of the -n option:



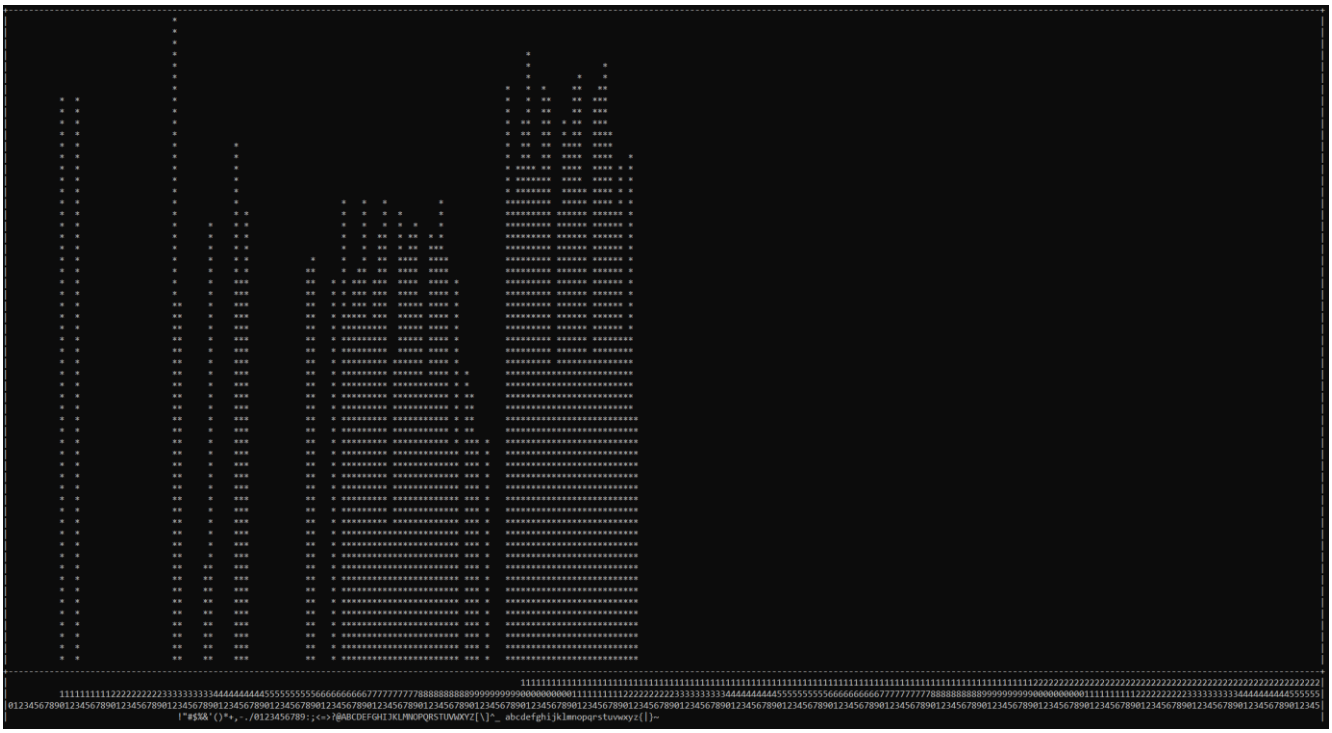
If *both* -a and -n are specified, add the -n output first, and the ASCII characters below that:



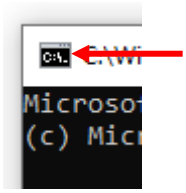
The -l (logarithmic) option compresses things vertically:



Finally, setting the number of rows up provides more resolution in the Y-axis (-r57):



You will likely want to change your command window to be at least 259 characters of width. To do so, open a command window, and click here:



Go to "Properties", "Layout", and set the BUFFER size to a width of 259 (or more), and a height of 300 (or more, but I recommend the maximum value of 9999 for the height – that lets you scroll back through a LOT of output). For the WINDOW size, make the width the same as the BUFFER size, and a height of whatever fits well on your screen (30 rows? 50? It depends on the font size and the resolution of your screen). On the "Font" tab, set the typeface to CONSOLAS, and whatever size is easy on your eyes (I use 20 on my laptop, and can juuuust get 50 rows to display at once).

After your graph, display a blank line as a separator, and then one line of text in this format:
`Min:NNNNNNNNN Max  (CCC):NNNNNNNNN AVG:XXXXXXXXX DEV:XXXXXXXXX`

Note: there are two spaces between "Max" and "(".

The Min and Max numbers will be the number of times the least- and most-frequently occurring characters appear. The `CCC` is the value of the character (000-255) that appears most often ("032" for space). The values for AVG and DEV will be the mean (the average of those 256 values) and the standard deviation, respectively. Use 9 spaces to display each, but let the scale of the number determine the number of decimal places. For example, if the average is 100, and you have 9 spaces to display it in, use 100.00000 (9 spaces total, 5 after the decimal place);

if the average is 10000, use the same 9 places, but move the decimal to the right: 10000.000. Do not display any values with commas.

As a double-check, the sum of all of your counts should match (*exactly*) the file's size, and the average * 256 should also be the same as the file size.

You may not have had occasion to calculate the standard deviation of a population before. Use this formula:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{255} count_i^2 - \left(\frac{\left(\sum_{i=0}^{255} count_i\right)^2}{N}\right)}{N}}$$

In this case, *N* is obviously 256. Once you have the 256 counts in an array, this formula makes it simple to do in a single pass.

I won't test your code on any file with more than 4 GB (and no more than 1 G of any single character), so you may use an array of **`unsigned int`**s to hold the individual counts. **Vectors, maps, bitmaps, collections, and all other C++ template library classes are not permitted; code this with a simple integer array.**

You are welcome to use C-style **`FILE*`** I/O, or a C++-style **`ifstream`** to read the file. Feel free to use http://www.cplusplus.com as a reference for syntax, what **`#include`** files you need, etc. Other than that, however, you should not use ANY other online reference – as students in a 4/5000-level EECS course, you should already have everything you need in order to do this. If your program can't find the file the user specifies, simply display "**`Input file not found`**", and exit with a return code of 1. Otherwise, process the file, display as the options dictate, and exit with a return code of zero.

Your goal in this project is to demonstrate the effect to which the encryption algorithm is able to "flatten out" the distribution and hide patterns. In plaintext, certain characters are expected to appear significantly more frequently than others (think "Wheel of Fortune" – 'L', 'N', 'R', 'S', 'T', and 'E', as discussed in class). If the encrypted version has no such patterns (or if they're SO exceptionally hard to spot that it's not feasible to look for them), it makes it harder to determine what plaintext the ciphertext might represent. Your goal is to document the extent to which the encryption has done so. Feel free to use the following websites for reference:
http://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html
https://en.wikipedia.org/wiki/Letter_frequency
http://www.ASCIIChart.com or http://www.asciitable.com

Submit a 7-zip archive of your workspace. Make sure your code has been tested on several files, particularly the ones I provide. It might be a good idea for you to make up some (tiny files) of your own, so you can check the results manually. *Make sure you submit x64 release code (not debug code). If you have both release and debug configurations, make sure you clean the debug one before submitting. I will grade only your x64 release code.*

The first requirement for code is that it works. Correctly. If your code doesn't produce the correct output for the test files I throw at it, it's not going to earn many points. If it won't compile at all, then it will receive a score of zero, no matter how many lines of code you have.

Your code must be acceptably documented. That includes header comments, and enough block and line comments scattered throughout your code for someone other than the author to follow it.

As with all assignments in this course, you may not use code from any source other than the references I have provided. Code from websites, classmates, books, or any other source other than your own creation will be considered an academic integrity violation, which will result in a grade of F for the semester.

If / when you run into problems on this project, resist the urge to check with your classmates on how to get past it. If you've checked the allowed resources and are still stuck, let ME know – don't confuse your peers' role in this course with your instructor's role!

For those of you taking the course for graduate credit:

Not only are *single*-letter frequencies important in terms of patterns, *digram* and *trigram* frequencies are important, too. Digrams are pairs of characters that appear sequentially (see https://en.wikipedia.org/wiki/Bigram). In English text, "**th**", "**e** ", "**ng**" (for most any present participle) and "**.** " are some of the most frequently occurring character *pairs*. Similarly, a trigram is any three characters that appear sequentially.

You will collect and report statistics on FOUR lists (so there will be four lines of statistics output at the end):

1. A list of the *single*-letter frequencies to display in a graph (as above),

2. A list of digrams (ANY two sequential characters, so the text "**they**" would produce the digrams "**th**", "**he**", and "**ey**"). Because there are 256 possibilities for the first character, and 256 for the second, there are 256*256, or 65,536 (64K) possible digrams to count. You are not to graph these, but you ARE to compute the max, min, average, and standard deviations of these counts as well, and display them in a SECOND summary line. Rather than "**MAX (CCC)**", display "**MAX (XXYY)**", where XX and YY are the HEXADECIMAL values of the two characters in the most frequently occurring diagram (for example, if the most frequently occurring diagram is "**th**", you would display "**MAX (7468)**" in that part of the line. Follow the format closely – the averages and standard deviations must line up vertically. Note, also, that the maximum number of possible combinations for the digrams is 65536, rather than 256. If more than one diagram corresponds to the minimal or maximal count, use any of them (but the minimal count can't be zero, unless you run it on a one-byte file!).

3. A list of *tri*grams (ANY *three* sequential characters, so an input of "**Everyone**" would produce six trigrams: "**Eve**", "**ver**", "**ery**", "**ryo**", "**yon**", and "**one**". Again, don't graph these; just output the max, min, average, and standard deviation. The maximum number of possible combinations for the trigrams is 256 * 256 * 256, or 16,777,216. If more than one trigram corresponds to the minimal or maximal count, use any of them (but the minimal count can't be zero!).

4. A list of "octagrams". Let's assume for the moment that we're analyzing cyphertext that came from plaintext encrypted with DES. The block size for DES is 64 bits (8 bytes). Therefore, the first 8 bytes of the cyphertext represent the encryption of the first 8 bytes of the plaintext, the next 8 bytes of ciphertext represent the encryption of the next 8 bytes of the plaintext, etc. The question is, how many times does EACH 8-byte block appear? Again, you will not graph these, but you DO need to count how many times each 8-byte block appears, and compute the max, min, average, and standard deviation of those counts. Note also that this is a little different from a single letter / diagram / trigram analysis, in that there's no overlap – these are *sequential* blocks of 8 characters. If a file contains 800 bytes, then there are 100 blocks to consider (which means $n = 100$ when computing the average and standard deviations). This one is complicated by the fact that there will be 2^{64} (18.4 quintillion) *possible* 8-byte patterns to count – you won't be able to put the counts into an array (as you can for the first three lists); you'll need something else. DO NOT use a linked list; your code must run the Shakespeare files in 10 seconds or less.

NO DATA STRUCTURING LIBRARY CODE MAY BE USED – WRITE YOUR OWN IF YOU NEED SOMETHING BEYOND AN ARRAY.