Faculty of Science and Engineering

Department of Electrical and Electronic Engineering

Report 1 – Code Documentation on Histogram Generation Program

ACADEMIC YEAR 2023/2024

Course Code: EEEE1027 UNMC

Student's Name : MARCUS YEW MIN JIE

Student ID : 20612926

Course : Please tick (√)

Electrical and Electronic Engineering (√)

Mechatronic Engineering ( )

TABLE OF CONTENTS

ABSTRACT

This report provides an insight on the histogram generation program made in C code. Several programming concepts such as file handling, calling by reference, use of arrays and implementation of conditional statements, nested loops are frequently used as the foundation of all the functions in the program. The program is capable of scanning file types specified as a '.hist' file and '.txt' file which contains numerical and alphabetical data respectively. The program is able to display the data in a horizontal and vertical histogram format. The explanation of the code will be discussed in later parts of the report and also reason behind the implementation of code used, and the design choices made.

INTRODUCTION

A histogram is a graph that shows frequency distributions. It offers a visual representation of a set of continuous or discrete data's underlying frequency distribution. A histogram's main objective is to display the data's shape, central tendency, and dispersion. The data for the histogram is divided into different bins or more commonly called intervals that represent different values. The frequency of the data will then be plotted accordingly on the bins that they correspond to. A histogram can be plotted horizontally or vertically depending on the requirements of the task. Histogram come in many shapes and sizes but there are two common shapes of a histogram. A normally distributed histogram is where all the data points tend to converge in the middle, producing a bell-shaped graph. This is also known as zero skew, as it has an evenly shaped graph. A skewed graph is where the data points are biased to one side of the histogram. This causes the graph to skew to one side over the other. Figure 1.1 shows the difference between these graphs. The distribution of frequency across the histogram can be seen clearly just by observing the shape of the graph. The code for the program that is constructed for this project will be replicating the format of a histogram and printing data in a histogram formatted horizontally and vertically.

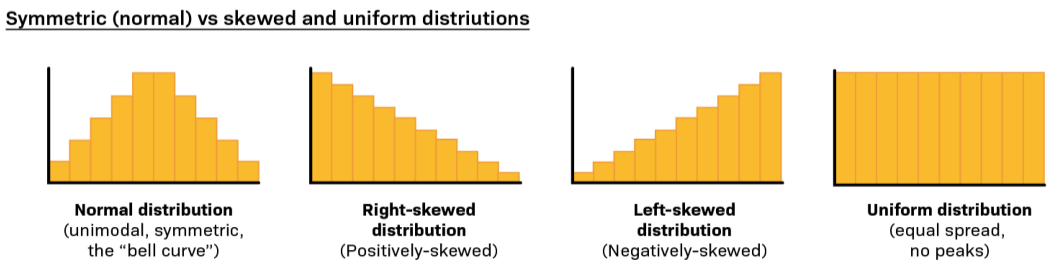
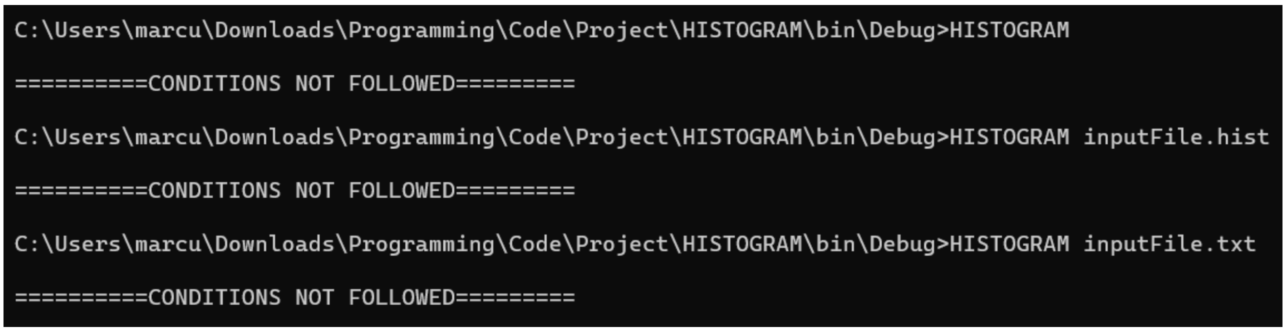


Figure 1.1: Different Shape of Histogram Graphs

1.0 MAIN.C FILE

The main.c file is set up so that it accepts two parameters, which are argument counter and argument vectors which are integers and strings respectively. This will allow the user to call commands from the command-line. Next, a few arrays and variables are initialized in the main file. The arrays consists of a bin array of floats, frequency array of floats and an integer array of the largest value all of size 100. An integer of N which is used to used as the highest bin index is also initialized to a value of 0. These variables are declared and initialized here to be able to call them by reference in later parts in the code as these parameters will often be passed to other functions.

There are six main functions that will be called in the main.c file which are readHistFile, printHist, plotHorizontalHist, findLargest, plotVerticalHist and readTextFile. Each of these functions play a role in printing the histogram which will be further explained in the section below. An ‘if’ statement is added where the argument counter has to be greater than two for it to run otherwise the program will return 0. This is so that the user would adhere to an expected input format where the two files have to be typed for the program to be ran. It is imperative that the histogram file name has to be entered first and the text file name second for the code to work. The choice behind setting the argument counter to be larger than 2 is to also avoid reading into unallocated memory and for better error handling. This provides an extra layer of security to ensure that argument vector 1 and argument vector 2 are truly present. Figure 1.1 shows a depiction of the output when the conditions are not fulfilled.



```
C:\Users\marcu\Downloads\Programming\Code\Project\HISTOGRAM\bin\Debug>HISTOGRAM

==========CONDITIONS NOT FOLLOWED=========

C:\Users\marcu\Downloads\Programming\Code\Project\HISTOGRAM\bin\Debug>HISTOGRAM inputFile.hist

==========CONDITIONS NOT FOLLOWED=========

C:\Users\marcu\Downloads\Programming\Code\Project\HISTOGRAM\bin\Debug>HISTOGRAM inputFile.txt

==========CONDITIONS NOT FOLLOWED=========
```

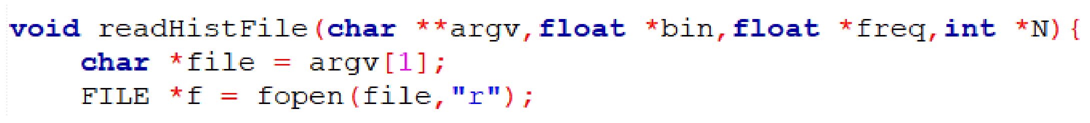
Figure 1.2: Output of Program When Conditions are Not Satisfied

2.0 HISTOGRAM.C FILE

The histogram.c file is where all the function prototypes are declared and defined. This file uses 4 libraries which are standard input output library, standard library, string library, math library and also a self-defined "histogram.h" library.

2.1 READ HISTOGRAM FILE FUNCTION

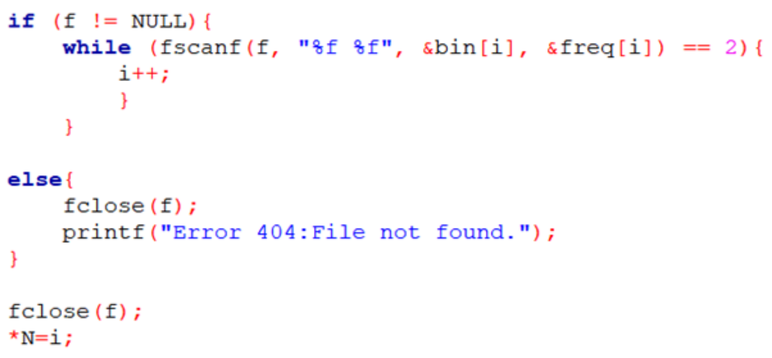
Before collecting the data required to print the histogram, it is required to access the file in interest to begin collecting data. "readHistFile" takes 4 input parameters, argument vector, the bin array, the frequency array and integer 'N'. The name of the file in interest will be passed from the command line which is argument vector 1 in this case. Figure 1.2 provides a code snippet of the code written to open the file.



```
void readHistFile(char **argv,float *bin,float *freq,int *N){
    char *file = argv[1];
    FILE *f = fopen(file,"r");
```

Figure 2.2: Code to Open File

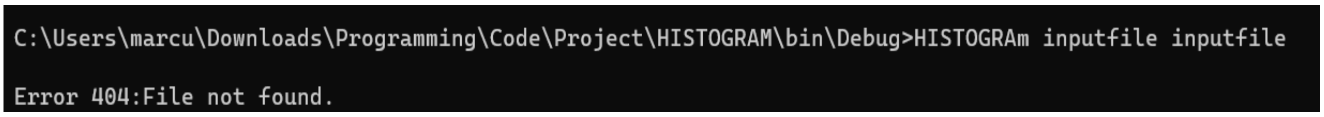
'char *file' initializes a pointer to a character called "file" and sets its value to the string that is the first command-line argument. Since the zeroth command-line argument is always the program name, it is not possible to declare file as the value of the zeroth command-line parameter. "FILE *f = fopen(file, "r")" assigns the outcome of opening the file given by "file" in read mode to the pointer to a 'FILE' structure named 'f'.



```
if (f != NULL){
    while (fscanf(f, "%f %f", &bin[i], &freq[i]) == 2){
        i++;
        }
    }

else{
    fclose(f);
    printf("Error 404:File not found.");
}

fclose(f);
*N=i;
```

Figure 2.3: Conditional statements for Opening File

Figure 1.3 shows an if else statement for two conditions; one where the file is valid and can be found (f !=NULL) and when the file cannot be found. If the file is found, the function will enter the if statement and enter a while loop that continuously scans the file for 2 floats and store them in bin array and frequency array respectively. This logic works because the histogram file is structured so that it has two columns, where the first value will be the bin and the second value encountered will be the frequency. With this knowledge in hand, it is expected that the two values will alternate between being a bin and a frequency value. Integer ‘i’ is initialized to a value of 0, and will increment every time 2 float values are detected. Integer ‘i’ corresponds to the total number of bins in this case and can also be said to be a counter for how many indexes are elements are in the file. Therefore, ‘i’ can be used to be the index number for both arrays. If the file is not found, the function will close the file that was opened and print an error message. Figure 2.4 shows output if file is not found.



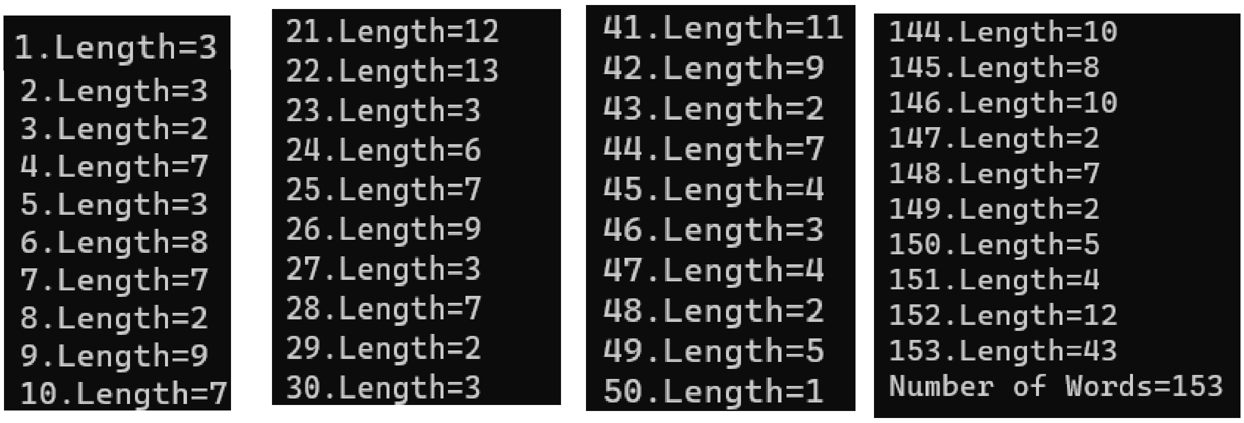
```
C:\Users\marcu\Downloads\Programming\Code\Project\HISTOGRAM\bin\Debug>HISTOGRAm inputfile inputfile

Error 404:File not found.
```

Figure 2.4: Output if File is Not Found

2.2 READ TEXT FILE FUNCTION

The “readTextFile” function objective is to be able to read text files just as how “readHistFile” are able to read hist files. The structure to open the files are the exact same as the “readHistFile” function except argument vector 1 is replaced with argument vector 2. The main objective of these two files are always to be able to read the files in their specified format but the secondary objective is to be able to populate the bin and frequency arrays, and pass the highest bin index out as a pointer to ‘N’ as these parameters are used in the general function to print the histogram horizontally and vertically.

A few variables were initialized and declared; among which were a integer ‘count’ with a value of 0 which acts as an index for length integer array and counts the total number of words. An array of characters called buffer which holds up to 100 letters is also declared. Similar to “readHistFile” function, it has an if-else statement, either it can read the file or it does not and prints an error message. If the file is valid, it will enter the while loop begin to scan the file with the ‘fscanf’ function. The ‘fscanf’ function will scan the word it encounters in the text file and store it in the buffer array that acts as a temporary storage. The length of the word stored can then be measured by using the ‘strlen’ function which stands for string length. The length of the word is added to the length integer array of index ‘count’ which starts at 0. After all this, the ‘count’ variable increments by 1 and this loop continues until it reaches the end of file. Since the text file is structured to be a paragraph, the same logic used in “readHistFile” function cannot be applied as a terminating condition for the loop. Instead, a special function called 'EOF' (end of file) is used where the 'fscanf' will scan up until the end of the file. Figure 2.6 depicts the contents of the length integer array and the value of count.



```
1.Length=3
2.Length=3
3.Length=2
4.Length=7
5.Length=3
6.Length=8
7.Length=7
8.Length=2
9.Length=9
10.Length=7
```

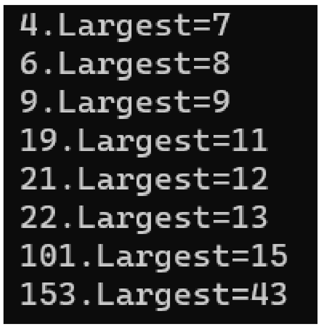
```
21.Length=12
22.Length=13
23.Length=3
24.Length=6
25.Length=7
26.Length=9
27.Length=3
28.Length=7
29.Length=2
30.Length=3
```

```
41.Length=11
42.Length=9
43.Length=2
44.Length=7
45.Length=4
46.Length=3
47.Length=4
48.Length=2
49.Length=5
50.Length=1
```

```
144.Length=10
145.Length=8
146.Length=10
147.Length=2
148.Length=7
149.Length=2
150.Length=5
151.Length=4
152.Length=12
153.Length=43
Number of Words=153
```

Figure 2.6: Depiction of the Contents of Length Array

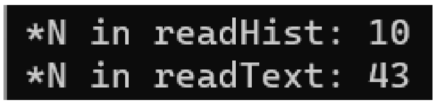
Now that the length of every word in the file is found, the longest word must be found in order to determine the highest bin index, since the length of word corresponds to the bin array and the number of occurrence of the same length will be added to the frequency array of that bin. To find the longest word, the first element in the length array is assumed to be the longest word. A for loop is then implemented to loop through all the elements in the length array up to the total words in the file, which is represented by the 'count' variable. Another if statement is implemented where the element encountered in the length array is larger than the previously determined largest element, it'll continuously update the largest value. A depiction of the output of this nested loop is shown in Figure 2.7 where it is determined by the nested loop that the longest word in the file is word 153, which is true.



```
4.Largest=7
6.Largest=8
9.Largest=9
19.Largest=11
21.Largest=12
22.Largest=13
101.Largest=15
153.Largest=43
```

Figure 2.7: Depiction of Output of the Nested Loop

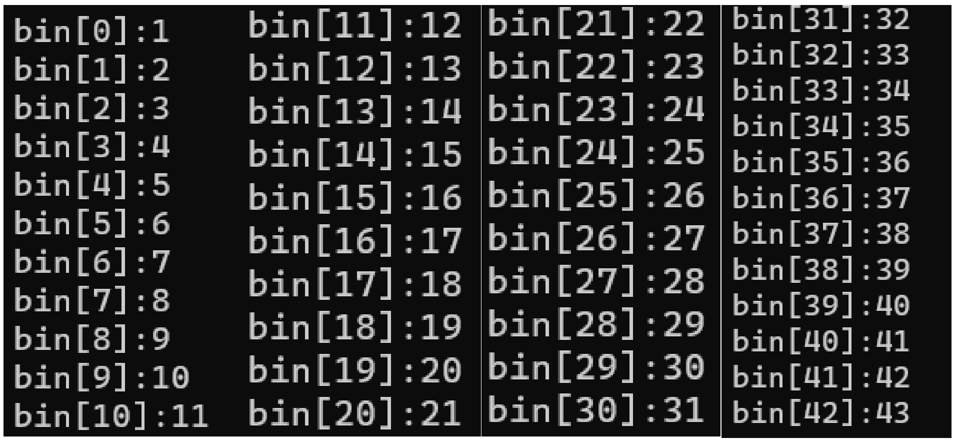
With the largest value found, the highest bin index can also be updated by passing largest to as a pointer to 'N' to call by reference in the main file. By using calling by reference, the previous '*N' can be modified remotely in another function since the memory address of N is passed in the function. Figure 2.8 depicts how the value of 'N' changes when it is called by reference.



```
*N in readHist: 10
*N in readText: 43
```

Figure 2.8: Depiction of value of N

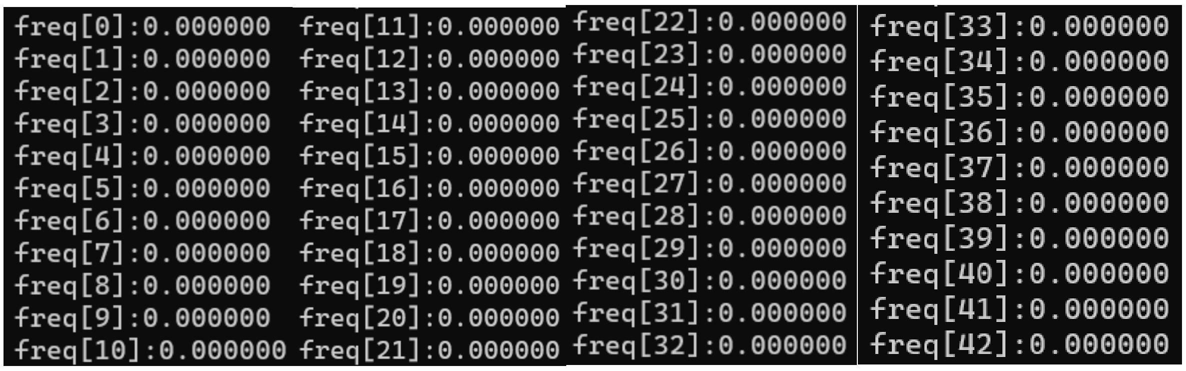
The next step is to determine the bin and frequency arrays. To fill the bin array, a simple for loop is used to cycle through all indices in the bin array up to the longest word length. It is worth noting that computers will determine the first element in the array as index 0. Therefore, it is important to take note that the value of the bin will always be 1 higher than its index as there are no word length that has 0 letters. This is to also avoid indexing errors and it is generally a good practice to start from the zeroth index. Figure 2.9 shows how the contents of the bin array are modified and printed.



```
bin[0]:1     bin[11]:12  bin[21]:22  bin[31]:32
bin[1]:2     bin[12]:13  bin[22]:23  bin[32]:33
bin[2]:3     bin[13]:14  bin[23]:24  bin[33]:34
bin[3]:4     bin[14]:15  bin[24]:25  bin[34]:35
bin[4]:5     bin[15]:16  bin[25]:26  bin[35]:36
bin[5]:6     bin[16]:17  bin[26]:27  bin[36]:37
bin[6]:7     bin[17]:18  bin[27]:28  bin[37]:38
bin[7]:8     bin[18]:19  bin[28]:29  bin[38]:39
bin[8]:9     bin[19]:20  bin[29]:30  bin[39]:40
bin[9]:10    bin[20]:21  bin[30]:31  bin[40]:41
bin[10]:11                           bin[41]:42
                                     bin[42]:43
```

Figure 2.9: Contents of the Bin Array

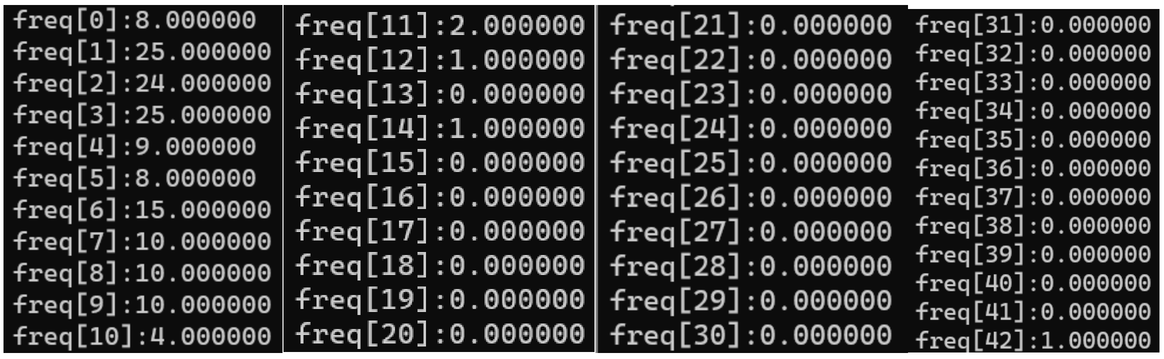
As mentioned before, the bin and frequency arrays are called by reference and used throughout almost every function in this project. Therefore, it is imperative to clear the arrays from their previously written data so that it starts recording data in an essentially clean slate of arrays. It is not necessary to clear the bin array as it values essentially do not change and only ever increase or decrease depending on the highest bin index. The frequency bin is cleared by using a for loop, cycling through all the array indices and equating them to a value of zero. This loop continues until it reaches the previously mentioned 'largest' variable which is the highest bin index. Figure 3.0 shows the contents of the frequency array after it is wiped.



```
freq[0]:0.000000  freq[11]:0.000000 freq[22]:0.000000 freq[33]:0.000000
freq[1]:0.000000  freq[12]:0.000000 freq[23]:0.000000 freq[34]:0.000000
freq[2]:0.000000  freq[13]:0.000000 freq[24]:0.000000 freq[35]:0.000000
freq[3]:0.000000  freq[14]:0.000000 freq[25]:0.000000 freq[36]:0.000000
freq[4]:0.000000  freq[15]:0.000000 freq[26]:0.000000 freq[37]:0.000000
freq[5]:0.000000  freq[16]:0.000000 freq[27]:0.000000 freq[38]:0.000000
freq[6]:0.000000  freq[17]:0.000000 freq[28]:0.000000 freq[39]:0.000000
freq[7]:0.000000  freq[18]:0.000000 freq[29]:0.000000 freq[40]:0.000000
freq[8]:0.000000  freq[19]:0.000000 freq[30]:0.000000 freq[41]:0.000000
freq[9]:0.000000  freq[20]:0.000000 freq[31]:0.000000 freq[42]:0.000000
freq[10]:0.000000 freq[21]:0.000000 freq[32]:0.000000
```

Figure 3.0: Contents of Frequency Bin After Cleaning

Now that the values of the frequency bin are all set to 0, the array is ready to be populated with data. The frequency array is supposed to collect data on the length of the word, and if the length of the word is the same, it is added the corresponding index of the frequency array. To achieve this, a for loop with an initial variable value, ‘k’, of zero is looped with a limit of variable ‘count’ with is the total number of words in the file. ‘k’ will increment by 1 each time the for loop runs. While the for loop is active, the frequency array will take each value of the length array at index ‘k’ as its index and increment its value by 1 if it encounters the same word length. The contents of the newly populated frequency array is shown in Figure 3.1 where its value increases depending on the number of times it encounters the word of the same length based on the index of ‘length’ array. The figure shows an example where the longest word is 43 letters long. The presence of the frequency arrays with the value of 0 proves that the frequency elements exist even if word of said length is not present.

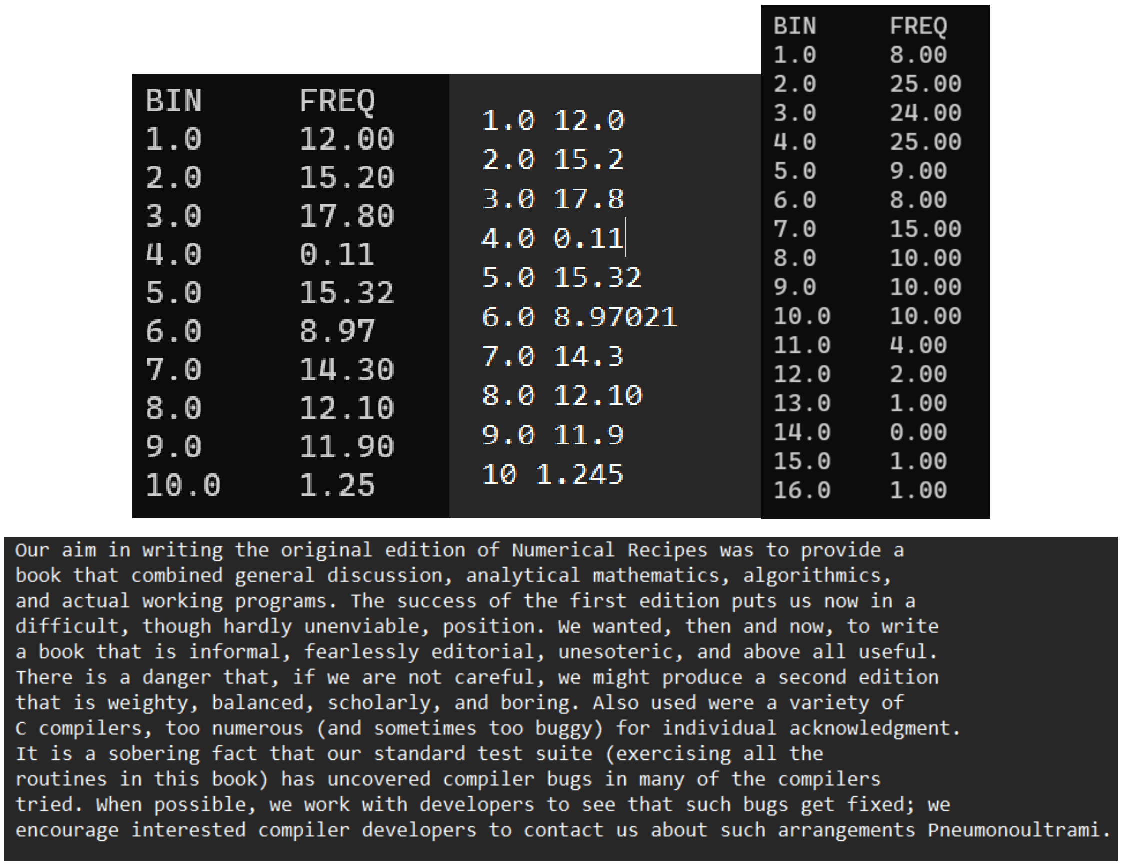


```
freq[0]:8.000000   freq[11]:2.000000 freq[21]:0.000000 freq[31]:0.000000
freq[1]:25.000000  freq[12]:1.000000 freq[22]:0.000000 freq[32]:0.000000
freq[2]:24.000000  freq[13]:0.000000 freq[23]:0.000000 freq[33]:0.000000
freq[3]:25.000000  freq[14]:1.000000 freq[24]:0.000000 freq[34]:0.000000
freq[4]:9.000000   freq[15]:0.000000 freq[25]:0.000000 freq[35]:0.000000
freq[5]:8.000000   freq[16]:0.000000 freq[26]:0.000000 freq[36]:0.000000
freq[6]:15.000000  freq[17]:0.000000 freq[27]:0.000000 freq[37]:0.000000
freq[7]:10.000000  freq[18]:0.000000 freq[28]:0.000000 freq[38]:0.000000
freq[8]:10.000000  freq[19]:0.000000 freq[29]:0.000000 freq[39]:0.000000
freq[9]:10.000000  freq[20]:0.000000 freq[30]:0.000000 freq[40]:0.000000
freq[10]:4.000000                                      freq[41]:0.000000
                                                       freq[42]:1.000000
```

Figure 3.1: Contents of New Frequency Array

2.3 PRINT HISTOGRAM FUNCTION

The bin and frequency array are now populated, but it is not explicitly shown because it has not been outputted yet. The “printHist” function will accept 3 parameters; the bin array, frequency array and the ‘N’ integer. The function consists of a simple for loop that will output the contents of each index of the array until it reaches the highest index for bin which is the pointer to ‘N’. The values of bin and frequency will also be formatted to display 1 significant figure and 2 significant figures respectively. The choice behind this decision is to be able to clearly read the data off of the histogram. Figure 2.5 shows an example of the content of the bin and frequency array and the contents of the histogram file for histogram files and text files.



```
BIN       FREQ
1.0       12.00
2.0       15.20
3.0       17.80
4.0       0.11
5.0       15.32
6.0       8.97
7.0       14.30
8.0       12.10
9.0       11.90
10.0      1.25
```

```
1.0 12.0
2.0 15.2
3.0 17.8
4.0 0.11
5.0 15.32
6.0 8.97021
7.0 14.3
8.0 12.10
9.0 11.9
10 1.245
```

```
BIN     FREQ
1.0     8.00
2.0     25.00
3.0     24.00
4.0     25.00
5.0     9.00
6.0     8.00
7.0     15.00
8.0     10.00
9.0     10.00
10.0    10.00
11.0    4.00
12.0    2.00
13.0    1.00
14.0    0.00
15.0    1.00
16.0    1.00
```

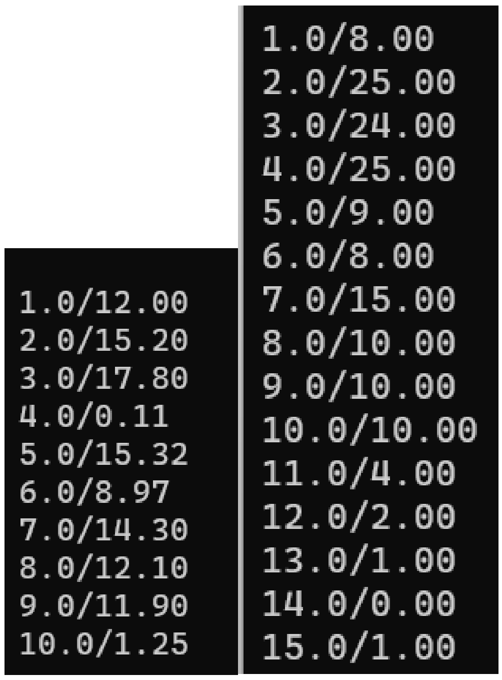
```
Our aim in writing the original edition of Numerical Recipes was to provide a
book that combined general discussion, analytical mathematics, algorithmics,
and actual working programs. The success of the first edition puts us now in a
difficult, though hardly unenviable, position. We wanted, then and now, to write
a book that is informal, fearlessly editorial, unesoteric, and above all useful.
There is a danger that, if we are not careful, we might produce a second edition
that is weighty, balanced, scholarly, and boring. Also used were a variety of
C compilers, too numerous (and sometimes too buggy) for individual acknowledgment.
It is a sobering fact that our standard test suite (exercising all the
routines in this book) has uncovered compiler bugs in many of the compilers
tried. When possible, we work with developers to see that such bugs get fixed; we
encourage interested compiler developers to contact us about such arrangements Pneumonoultrami.
```

Figure 2.5: Difference Between Output and File Data

2.4 PRINT HORIZONTAL HISTOGRAM FUNCTION

Now that the array is confirmed to have values corresponding to its index, it is possible to start printing in an actual histogram format. plotHorizontalHist accepts three parameters; the bin array, frequency array and the ‘N’ integer to print a histogram file horizontally. This function can be broken down to two parts, printing the values of the bin and frequency array, and printing the stars that correspond to the frequency. Similarly to printing the contents of the arrays from the section above, a for loop can be implemented to cycle through all the indexes of the array and print the values of them. The initial value of integer ‘i’ will be zero, which will stand for the index of the arrays, and will increment by 1 up until the highest bin index (*N).

The formatting is tweaked so that the values of bin and frequency are printed together separated by a slash and leaving space to print the stars to the side of them which can be depicted in Figure 3.2.

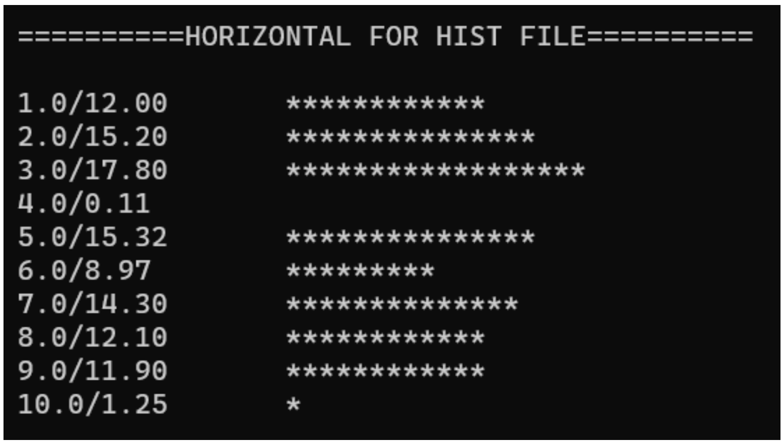


```
1.0/12.00
2.0/15.20
3.0/17.80
4.0/0.11
5.0/15.32
6.0/8.97
7.0/14.30
8.0/12.10
9.0/11.90
10.0/1.25
```

```
1.0/8.00
2.0/25.00
3.0/24.00
4.0/25.00
5.0/9.00
6.0/8.00
7.0/15.00
8.0/10.00
9.0/10.00
10.0/10.00
11.0/4.00
12.0/2.00
13.0/1.00
14.0/0.00
15.0/1.00
```

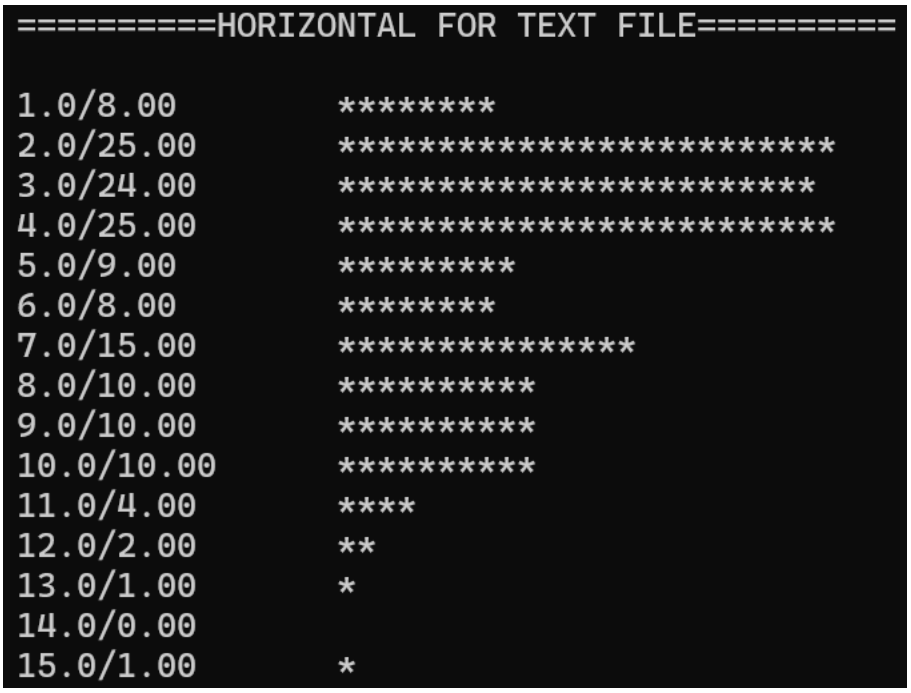
Figure 3.2: Bin and Frequency Arrays in Horizontal Histogram

The symbol used to represent each frequency is asterisk (*). In this project, each asterisk represents a frequency. The reason behind this decision is so that the granularity of the histogram can be more easily observed compared to a scaled frequency representation where the lowest limit that can be represented will always be one asterisk. A new integer variable, ‘stars’, is declared to be the values the frequency rounded up. To cycle through all the elements of the frequency array, it is placed within the previous for loop used to print the bin and frequency arrays. Another for loop is implemented into the existing for loop to print the number of stars corresponding to the frequency array. By using a nested loop, the condition of the second for loop still depends on the limits imposed on the first for loop. A new line character is printed after every line by implementing it within the first for loop but out of the second. This ensures that it will print after every bin array is cycled. Figure 3.3 depicts the complete horizontal histogram for the histogram file while Figure 3.4 shows the horizontal histogram for the text file.



```
==========HORIZONTAL FOR HIST FILE==========

1.0/12.00       ************
2.0/15.20       ***************
3.0/17.80       ******************
4.0/0.11
5.0/15.32       ***************
6.0/8.97        *********
7.0/14.30       **************
8.0/12.10       ************
9.0/11.90       ************
10.0/1.25       *
```

Figure 3.3: Horizontal Histogram for .hist File

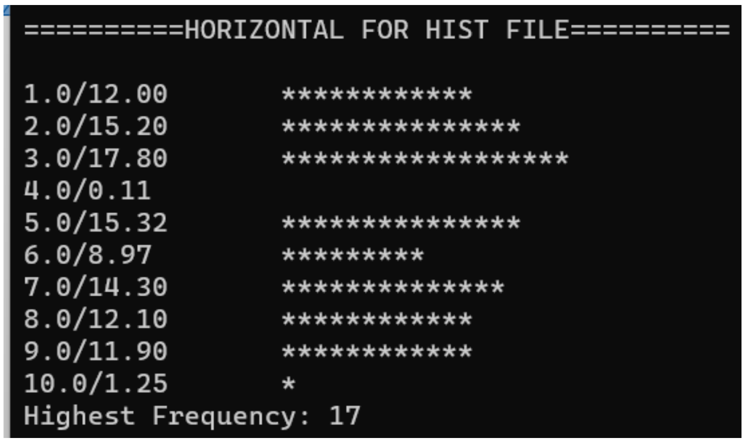


```
==========HORIZONTAL FOR TEXT FILE==========

1.0/8.00        ********
2.0/25.00       *************************
3.0/24.00       ************************
4.0/25.00       *************************
5.0/9.00        *********
6.0/8.00        ********
7.0/15.00       ***************
8.0/10.00       **********
9.0/10.00       **********
10.0/10.00      **********
11.0/4.00       ****
12.0/2.00       **
13.0/1.00       *
14.0/0.00
15.0/1.00       *
```

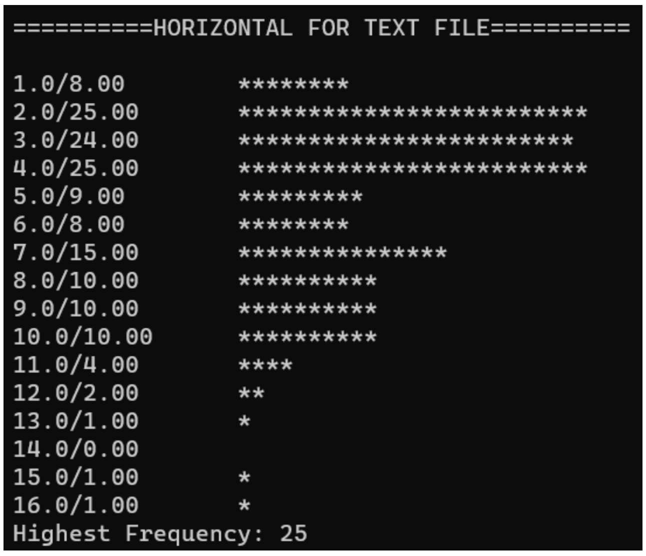
Figure 3.4: Horizontal Histogram for .txt File

2.5 FINDING HIGHEST FREQUENCY

This function is imperative in helping the program to print the histogram vertically. This is because the program will print the output line by line. To print a vertical histogram, it has to start printing from the frequency of highest value and decrement from there. The “findLargest” will help in accomplishing that goal. “findLargest” takes 3 input parameters, which are the frequency array, the pointer to integer ‘N’ and also the pointer to integer ‘largestValue’. ‘largestValue’ is also declared in the main file so that it can be passed out of the function and be called by reference later on in the project. This algorithm is identical the for loop used in ‘readTextFile’ to find the longest length of the word in the file. In this case, it is used to find the frequency with the highest value. Once the largest value is found, it is type casted as an integer and passed out as a pointer to ‘largestValue’. This is because largestValue will be used as a limit in a for loop in printing the vertical function; therefore it has to be an integer to be used as a conditional limit. Figure 3.5 shows the variable ‘largestValue’ corresponding to the highest spike in the horizontal histogram for histogram file and text file respectively.



```
==========HORIZONTAL FOR HIST FILE==========

1.0/12.00       ************
2.0/15.20       ***************
3.0/17.80       ******************
4.0/0.11
5.0/15.32       ***************
6.0/8.97        *********
7.0/14.30       **************
8.0/12.10       ************
9.0/11.90       ************
10.0/1.25       *
Highest Frequency: 17
```



```
==========HORIZONTAL FOR TEXT FILE==========

1.0/8.00        ********
2.0/25.00       *************************
3.0/24.00       ************************
4.0/25.00       *************************
5.0/9.00        *********
6.0/8.00        ********
7.0/15.00       ***************
8.0/10.00       **********
9.0/10.00       **********
10.0/10.00      **********
11.0/4.00       ****
12.0/2.00       **
13.0/1.00       *
14.0/0.00
15.0/1.00       *
16.0/1.00       *
Highest Frequency: 25
```

Figure 3.5: Highest Frequency Representation

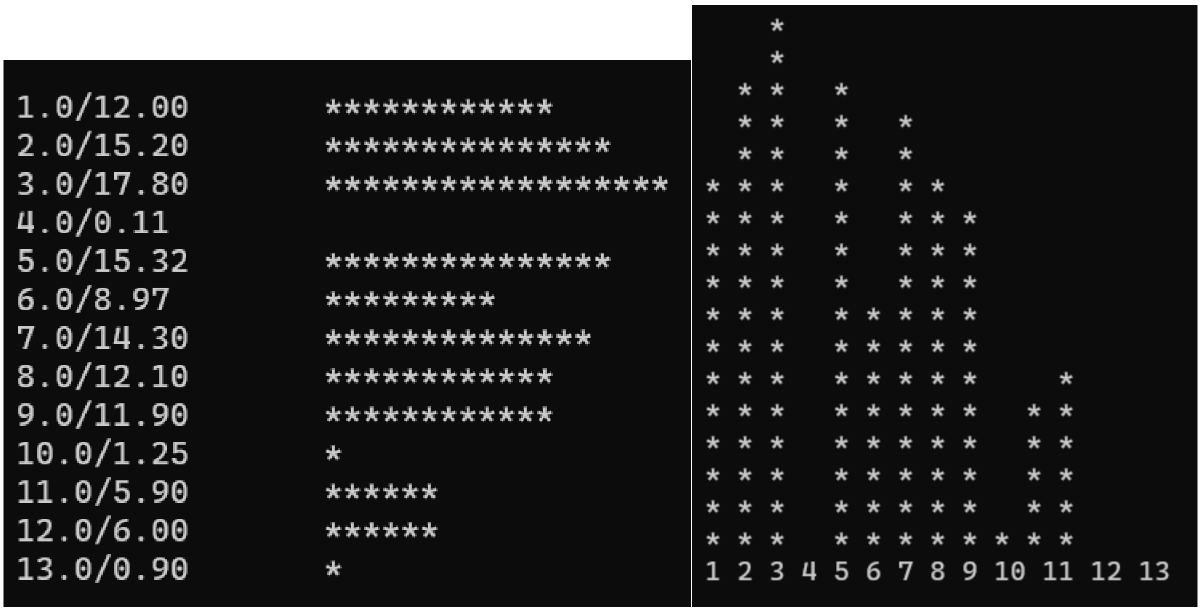
2.6 PRINT VERTICAL HISTOGRAM FUNCTION

With all the information needed to print the histogram vertically, the function definition can be constructed. The function “plotVerticalHist” accepts 4 input parameters, which are the bin array, the frequency array, pointer to integer ‘N’ and pointer to integer ‘largestValue’. The code to print a vertical histogram is shown in Figure 3.6.

```
for (int i = *largestValue; i > 0; i--) {
    for (int count = 0; count < *N; count++) {
        if ((int)freq[count] >= i) {
            printf("* ");
                                            }
        else {
            printf("  ");
                                        }
                                    }
    printf("\n");
            }
```

Figure 3.6: Code to Print a Vertical Histogram

This is a nested loop which includes 2 ‘for’ loops and an ‘if-else’ statement. The ‘for’ loop makes it so that the program will start printing from the highest frequency spike and decrements by 1 until it reaches 1. The ‘for’ loop within raises the condition that with an initial count of 0, it will keep looping as long as it is less than the highest bin index and increments by 1 every time the ‘for’ loop is carried out. The ‘if-else’ statement checks if each value if the frequency array is equal or bigger than the ‘largestValue’. If it is, it will print a asterisk to represent one frequency. If it is not, it will just print a blank space. Finally, a new line character is printed everytime ‘largestValue’ decrements, which indicates a new line. Figure 3.7 shows the output of this code and a comparison to the horizontal histogram.

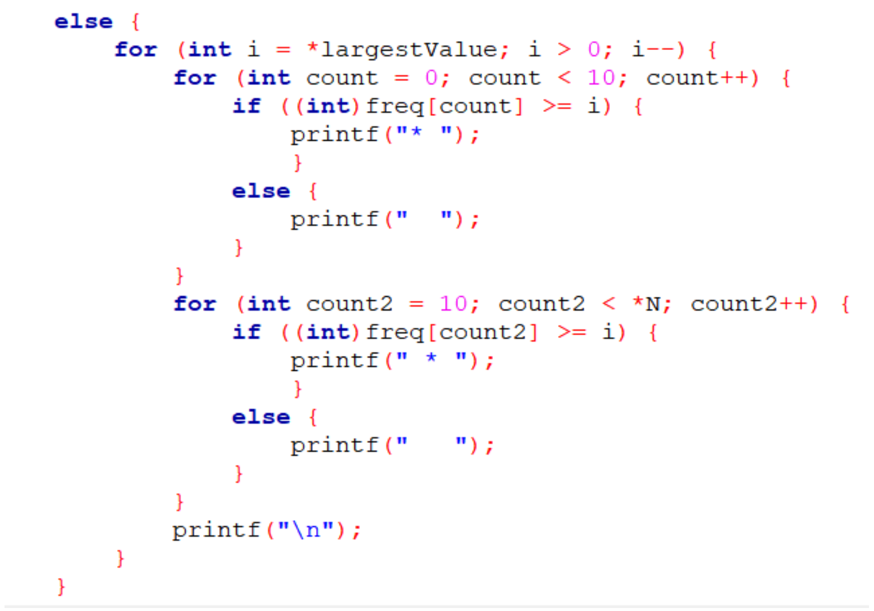

```
1.0/12.00       ************
2.0/15.20       ***************
3.0/17.80       ******************
4.0/0.11
5.0/15.32       ***************
6.0/8.97        *********
7.0/14.30       **************
8.0/12.10       ************
9.0/11.90       ************
10.0/1.25       *
11.0/5.90       ******
12.0/6.00       ******
13.0/0.90       *
```

```
    *
    *
  * *   *
  * *   *   *
  * *   *   *
* * *   *   * *
* * *   *   * * *
* * *   *   * * *
* * *   *   * * *
* * *   * * * * *
* * *   * * * * *
* * *   * * * * *     *
* * *   * * * * *   * *
* * *   * * * * *   * *
* * *   * * * * *   * *
* * *   * * * * *   * *
* * *   * * * * * * * *
1 2 3 4 5 6 7 8 9 10 11 12 13
```

Figure 3.7: Output of Vertical Histogram Compared to Horizontal Histogram

From the figure above, it can be seen that the vertical histogram has been successfully printed and the values also matches up to the horizontal histogram. Upon closer inspection, it can be seen that for plot points beyond the 10th bin array, the alignment is incorrect. To solve this problem, another block of code involving ‘if-else’ statements have been implemented.

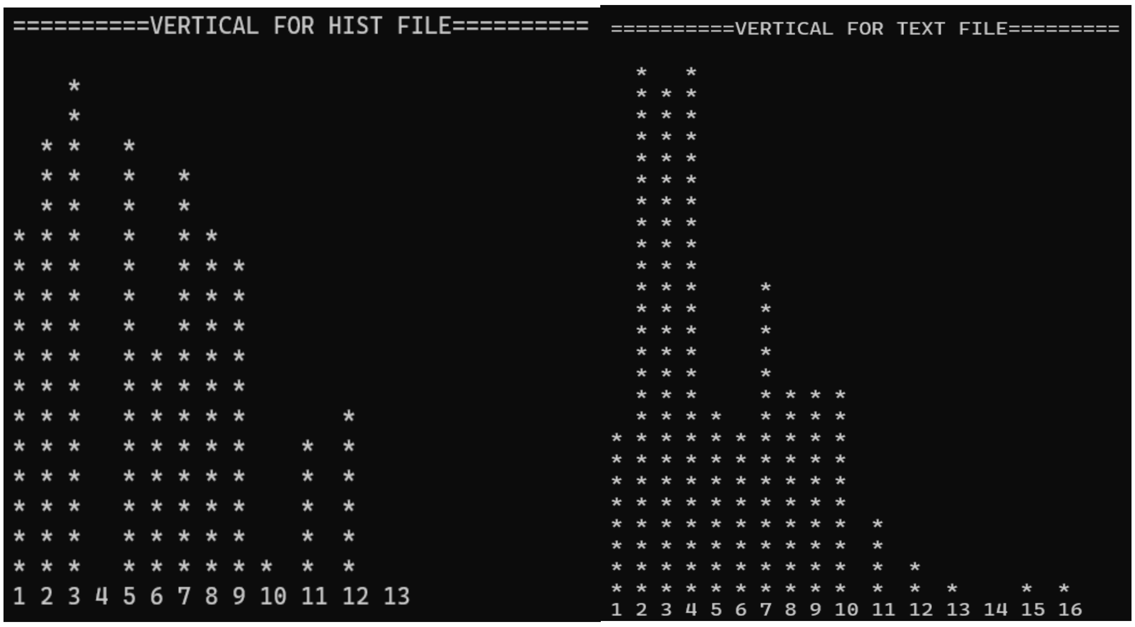
An if statement is added in front of the block of code to print the vertical histogram where the condition is if the highest bin index is lesser than 10, it will just print with normal indentations. The ‘else’ statement has the condition where if the highest bin index is higher than 10, it will print the data with normal indentations for values corresponding lesser than the 10th bin array. For values after the 10th bin array, a secondary ‘for’ loop is implemented where the initial variable ‘count2’ is set to 10 and will keep looping up until the highest bin index but with an extra indentation to complement the spacing for the double digit bin arrays. Figure 3.8 shows a code snippet of the condition where it has more than 10 bin arrays.



```c
else {
    for (int i = *largestValue; i > 0; i--) {
        for (int count = 0; count < 10; count++) {
            if ((int)freq[count] >= i) {
                printf("* ");
                }
            else {
                printf("  ");
            }
        }
        for (int count2 = 10; count2 < *N; count2++) {
            if ((int)freq[count2] >= i) {
                printf(" * ");
                }
            else {
                printf("   ");
            }
        }
        printf("\n");
    }
}
```

Figure 3.8: Code to Print Values Beyond 10th Bin Arrays

The last part of the code is to print the bin array corresponding to the frequency spikes at the bottom of the histogram. A simple 'for' loop is implemented to print all the values of the bin array. Figure 3.9 shows the output of the corrected histogram.



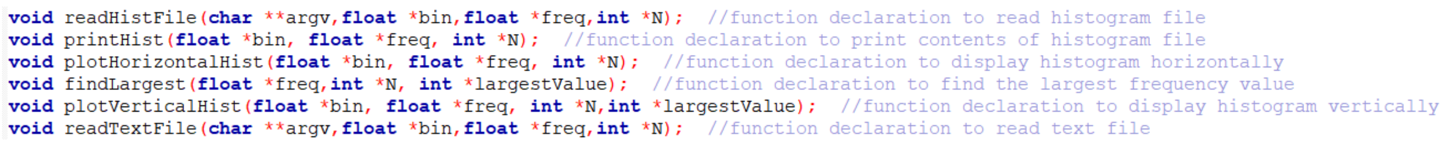
```
==========VERTICAL FOR HIST FILE==========

    *
    *
  * *   *
  * *   *   *
  * *   *   *
* * *   *   * *
* * *   *   * * *
* * *   *   * * *
* * *   *   * * *
* * *   * * * * *
* * *   * * * * *
* * *   * * * * *       *
* * *   * * * * *    *  *
* * *   * * * * *    *  *
* * *   * * * * *    *  *
* * *   * * * * *    *  *
* * *   * * * * * *  *  *
1 2 3 4 5 6 7 8 9 10 11 12 13
```

```
==========VERTICAL FOR TEXT FILE=========

  *   *
  * * *
  * * *
  * * *
  * * *
  * * *
  * * *
  * * *
  * * *
  * * *
  * * *     *
  * * *     *
  * * *     *
  * * *     *
  * * *     *
  * * *     * * * *
  * * * *   * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *  *
* * * * * * * * * *  *
* * * * * * * * * *  *  *
* * * * * * * * * *  *  *  *     *  *
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

Figure 3.9: Output of the Vertical Histogram

3.0 HISTOGRAM.H FILE

The “histogram.h” consists of the 6 function prototypes that are declared. The choice to include them in a separate header file is to reduce the run time of the program so that it runs smoother. It also provides better modularity as it is separated with the implementation details. It is also easier to make comments to provide documentation on what each of the function does without needing to look through the implementation details. Figure 4.0 shows the “histogram.h” file with all the function prototype and comments with a brief summary of what each of them do.



```
void readHistFile(char **argv,float *bin,float *freq,int *N);  //function declaration to read histogram file
void printHist(float *bin, float *freq, int *N);  //function declaration to print contents of histogram file
void plotHorizontalHist(float *bin, float *freq, int *N);  //function declaration to display histogram horizontally
void findLargest(float *freq,int *N, int *largestValue);  //function declaration to find the largest frequency value
void plotVerticalHist(float *bin, float *freq, int *N,int *largestValue);  //function declaration to display histogram vertically
void readTextFile(char **argv,float *bin,float *freq,int *N);  //function declaration to read text file
```

Figure 4.0: Function Prototypes

CONCLUSION

In conclusion, the program constructed is able to successfully read from ‘.hist’ files and ‘.txt’ files and output the data processed in a horizontal and vertical histogram format. ‘.hist’ files consists of numerical data and ‘.txt’ file will consist of paragraphs of word. The program will read into the files by inputting the file names as arguments in the command-line. The distribution of the frequency can be observed with ease once the data is processed by the program such as the occurrence of each word length in the ‘.txt’ file. Functions are implemented separately to improve the code readability and maintainability. Future improvements for the program may include the use of dynamically allocating memory compared to statically allocating memory to arrays. The use of dynamically allocated memory provides more flexibility for the program towards frequencies and bins with extreme values. Overall, the program fulfils its purpose by effectively processing information and showing data distributions for alphabetical and numerical data.