Writeup Document for Assignment 1

Shirin Rokni

October 2, 2022

Plot 1

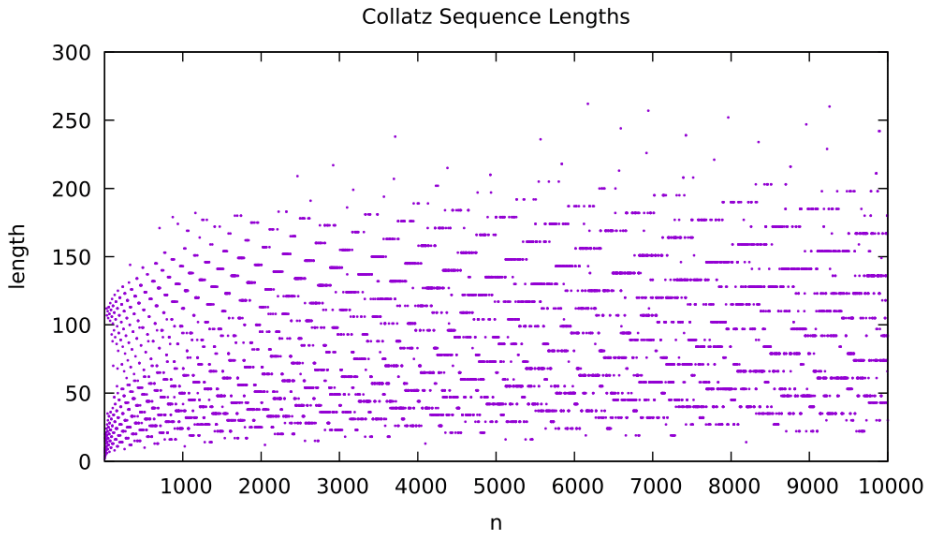

Figure 1: The length of Collatz sequences starting from n in [2,...,10000]

In graph 1, the x-axis **"n"** and the y-axis **“length”** are the main variables the bash script must pay attention to and plot. In order for the bash script plot to **n**, it must create an index that the ./collatz file can produce an output for. Thus, it uses a for-loop from 2 to 10,000. So, the x-axis **“n”** has been identified to be later compiled into another .dat file which follows the (x, y)-coordinate pattern that gnuplot plots with. However, in order for the bash script to run quickly and efficiently, it should call ./collatz several times within the loop, so there is a parameter coll.dat which catches all of the collatz sequence values. Now, the variable **“length”** must be identified. In this context, length refers to how long each collatz

sequence is, so the unix command that should be used is **“Word Count”** or **wc**. By using **wc -l** on the coll.dat parameter, the output will be the number of lines that are in the file, or the length. Another necessary command to find length is the unix command **“cat”** which concatenates files and prints on the standard output. So, in order for the word count command to work on the coll.dat file, it needs to be concatenated first which is why the command **“cat”** is so necessary. I was able to use **"cat"** on the file with the **pipelining** process. This allows the concatenation of the file to act as an input of the word count command. Now, both **“n”** and **“length”** arguments must be put into another file which can be plotted with gnuplot. This can be done through the unix command **“echo”** which can be used to display information that was passed through its argument. The for-loop utilizes **“echo”** with its arguments **“$i $length”** and appends it to the /tmp/graph1.dat file which will hold the final (x,y)-coordinates. After the for loop finishes, the /tmp/graph1.dat file will hold all of the coordinates that the gnuplot program will plot. Finally, the program calls the non-interactive mode for **gnuplot** and creates a file listing of commands that will instruct the final PDF to reflect the desired output. Such commands are set output, set xrange, set title, etc.

Plot 2

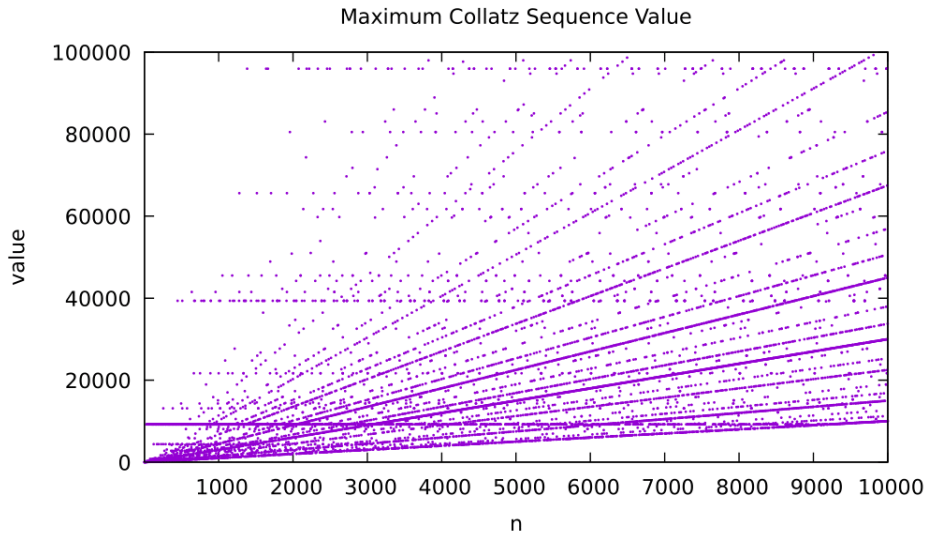

Figure 2: The maximum values for Collatz sequences starting from n in [2,...,10000]

In graph 2, the x-axis **“n”** and the y-axis **“value”** are the main variables the bash script must pay attention to and plot. Similar to graph1, graph2 finds **“n”** through the for-loop from 2 to 10,000. Now, the variable **“value”** must be identified. In this context, value refers to the highest value of each collatz sequence. Also similar to graph1, the coll.dat file needs to be concatenated before any command can be used on it, so the unix command **“cat”** is necessary. So, the unix commands that must be used are **“sort”** and **“tail”**. The unix command **“sort”** sorts lines of text files and the command **“tail”** outputs the last part of the file. When **“sort -n”** is run on the concatenated collatz parameter, the file is being numerically sorted. Then, when **“tail -1”** is run, it prints one line of the last part of the file. So in tandem, the two commands sort the file and then find the largest value. I was able to use **"cat"** on the file with the **pipelining** process. This allows the concatenation of the file to act as an input of the word count command. Now, both **“n”** and **“value”** arguments must be put into another file which can be plotted with gnuplot. This, similar to graph1, can be done through the unix command **“echo”**. The for-loop utilizes **“echo”** with its arguments **“$i $values”** and appends it to the /tmp/graph2.dat file which will hold the final (x,y)-coordinates. After the loop terminates, the /tmp/graph2.dat file should hold all of the coordinates that gnuplot will plot. Finally, the program calls the non-interactive mode for **gnuplot** and creates a file listing of commands that will instruct the final PDF to reflect the desired output. Such commands are the same ones being used for graph1.

Plot 3

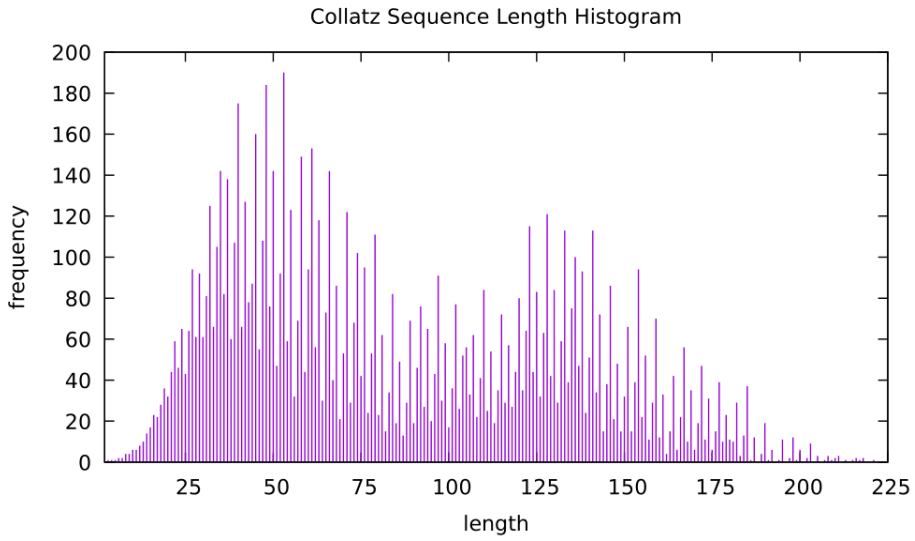

Figure 3: A histogram of Collatz sequence lengths starting from n in [2,...,10000]

In graph3, the x-axis **“length”** and y-axis **“frequency”** are the main variables the bash script must pay attention to and plot. Since graph1 also has a variable **“length”**, the script utilizes the already existing information and builds upon it. In order for changes to be made to the length variable, it must be a file with various data points inside of it, so inside of the main for-loop, each length value will be appended to a file named /tmp/length.dat. This is done through the unix command **“echo”** which has been used for the previous graphs. After the for-loop terminates, all of the lengths of each collatz sequence has been recorded, and so the script can now find the frequency. In this context, **“frequency”** refers to how many times a collatz sequence of a certain length has been repeated. Thus, the unix commands **“sort”** and **“uniq”** will be used. Similar to graph2, **“sort -n”** is used to numerically sort the /tmp/length.dat file. The reason for this is that **uniq** requires a sorted list in order to perform properly. The unix command **“uniq”** reports or omits repeated lines, and for this context the script reports the repeated lines. So when the script runs **“uniq -c”** on the sorted length file, it will output two columns; the first being the **frequency** and the second being the **length** value. I was able to use **"uniq"** on the file with the **pipelining** process. This allows the sorted file to act as an input of the uniq command. These pairs of numbers are then appended to the /tmp/graph3.dat file. Finally, the program calls the non-interactive mode for **gnuplot**

and creates a file listing of commands that will instruct the final PDF to reflect the desired output. However, unlike the previous two graphs, graph3 requires its values to be swapped before it is plotted. This is because of the nature of **uniq -c** which outputs the repetition value before the value itself, which is the opposite of what our graph should plot. So, in addition to the commands that are being used the previous graphs, graph3 plots its points with the second column acting as the x-value and the first column acting as the y-value to achieve the desired **(length, frequency)** coordinate pairing.

Plot 4

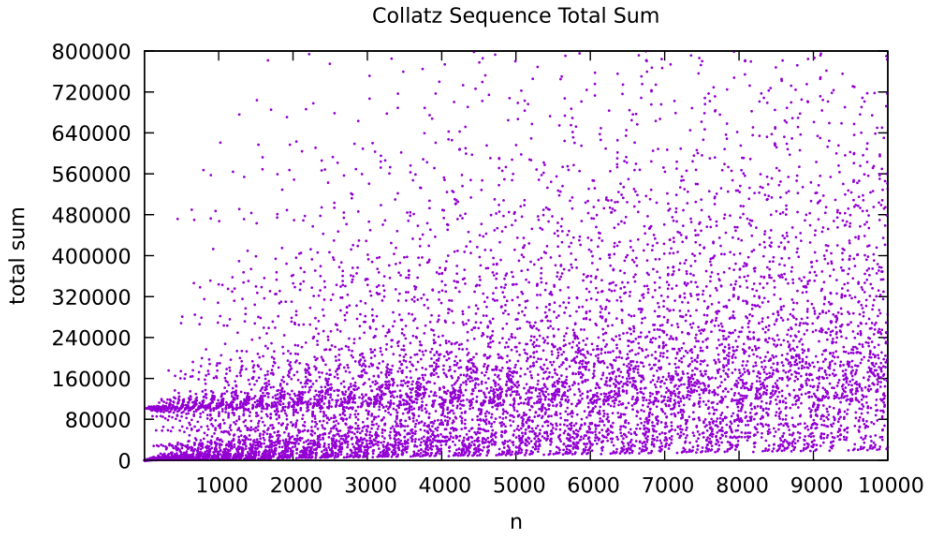

Figure 4: The total sum of each Collatz sequence starting from n in [2,...,10000]

In graph4, the x-axis **"n"** and the y-axis **"total sum"** are the main variables the bash script must pay attention to and plot. Unlike the previous three graphs, the process of obtaining the coordinates does not use any unix commands; Instead, it uses a nested for loop. In the context of the collatz sequence, **"total sum"** refers to the sum of each collatz sequence. Since the information on each collatz sequence exists within the coll.dat parameter, the script only needs to iterate through it and add all of its values into a variable **"total"** in order to obtain the y-coordinate data. Now, both **"n"** and **"total sum"** arguments must be put into another file which can be plotted with gnuplot. This, similar to graphs 1 and 2 can be done through the unix command **"echo"**. The for-loop utilizes **"echo"** with its arguments **"$i $total"** and

appends them to the /tmp/graph4.dat file. After the loop terminates, the /tmp/graph4.dat file should hold all of the coordinates that gnuplot will plot. Finally, the program calls the non-interactive mode for **gnuplot** and creates a file listing of commands that will instruct the final PDF to reflect the desired output. Such commands are the same ones being used for graph1 and 2.