Counting Number of Lines in a File using Shell Script Function

A TERM PROJECT REPORT

Submitted in partial fulfilment for the 2nd year, 4th semester of

Bachelor of Technology in

Operating System CSD – 222

By

Avinal – 185067, Harsimranjeet – 185087, Pooja – 185101

Project Code

https://blitzar-to-supernova.github.io/Line-Count-Complexity-OS/

Under the Guidance of

Dr. Pardeep Singh

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY

Hamirpur, Himachal Pradesh

India - 177005

Project Statement

Using a Shell Script define a function `count_lines` that counts the number of lines in the file provided as argument. Inside the function the value of the argument is retrieved by accessing the variable $1. Discuss the complexity of the code and propose your amendments.

Our Approach

Since there is not much of file read and write functionality available in shell script, we had to use implement this function using alternate methods. Out version the function uses the grep utility available for Unix-like systems. grep stands for `Globally search for a Regular Expression and Print`. It is used to find the occurrences of a string of character or a regular expression in a text file. As described in manual grep is –

> `“grep searches for PATTERNS in each FILE. PATTERNS is one or more patterns separated by newline characters, and grep prints each line that matches a pattern. Typically, PATTERNS should be quoted when grep is used in a shell command.”`

Function description

We first define a function named `count_lines()` in shell script. The code is as below: -

```
#! /bin/bash
function count_lines() {
    find "$1" | while read FILE; do
        echo $(grep -c ^ <"$FILE") >> comparefile.txt
    done
}
count_lines $1
```

- `find “$1”` – This command searches for the file name passed as argument to the function and returns a FILE object.
- `|` - The pipe is a form of redirection. It is used to send the output of one command to another command. Here the output of find is passed to the while loop.
- `while read FILE` – read command acts as an input operator and this whole command runs until file end is not detected.
- `grep -c ^ <” $FILE”` – here file is passed to the grep command by indirection and the -c option is used to count the number of occurrences.
- At last the function call is written that is `count_lines $1`. Here $1 is the first argument passed to the shell script.

Running the script

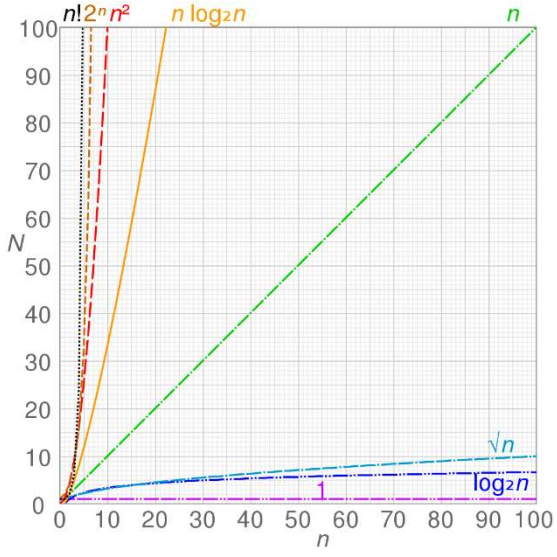
Shell Scripting is a kind of procedural programming. All the instructions are evaluated line-by-line. Now that our function is ready, we can save and test it. We may choose any file name to save with. In our case its test.sh. We may omit the '.sh' extension. There is no way to explicitly call the function from a shell script. So, we must call the function within the shell script itself. The command to run is as below.

```
#! /bin/bash
./test.sh testfile.txt
100
```

Here ./test.sh is the script call and testfile.txt is the argument denoted by $1.

Analysing the function's time complexity

Time complexity of a function or algorithm is the computational complexity that describes the amount of time taken to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken, and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.



Time complexities are denoted as Big O notation.

Most common time complexities are as below.

- Constant time – $O(1)$
- Linear time – $O(n)$
- Logarithmic time – $O(\log n)$
- Polynomial time – $O(n^a)$
- Exponential time – $O(2^n)$
- Factorial time – $O(n!)$
- $O(n \log n)$

We can hence run the function many times with different text file and record the time. This benchmark will help us determine the average time it takes to count single line. Then on basis of the time taken we can find the average time complexity of the function. We achieved this benchmarking result by writing a C++ program that runs the script 100 times with 1000 lines increment in each loop and records the time taken in microseconds. The C++ program is as below.

```cpp
#include <iostream>
#include <fstream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main(int argc, char const *argv[])
{
    string line = "random-1000-char-string";
    fstream timefile("timedur.txt");
    for (long i = 0; i < 100; i++)
    {
        fstream testfile("testfile.txt", ios::app);
        for (size_t j = 0; j < 1000; j++)
        {
            testfile << line << endl;
        }
        testfile.close();
        auto start = high_resolution_clock::now();
        system("./test.sh testfile.txt");
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop - start);
        timefile << (i + 1) * 1000 << " "
                 << duration.count() << std::endl;
    }
    timefile.close();
    return 0;
}
```

After this C++ program is run the results are saved in ‘timedur.txt’. This text file contains the number of lines and time taken to count them. A snapshot of the file is given below. Time is counted in microseconds for maximum accuracy.

```
#    time
1000 86119
2000 72407
3000 84871
4000 103455
5000 115358
```

Determining the time complexity from time duration data

The above program is run two times and the data is stored in two text files. Now we have data on how our functions performs for every 1000-line increment. The final number of lines was 1 million. We can now manually search for patterns in these data files to find the closest average time complexity for a function. Manually finding the complexity by comparing these 200 data points can be a very difficult task. And then we will not be able to determine the complexity function easily.

But to make things more interesting and accurate we used a bit of machine learning and statistics operations. Linear Regression is used to model linear relationship between many independent variables. It can then be used to fit a curve within the data set. Linear regression works by minimising the error between curve and data points.

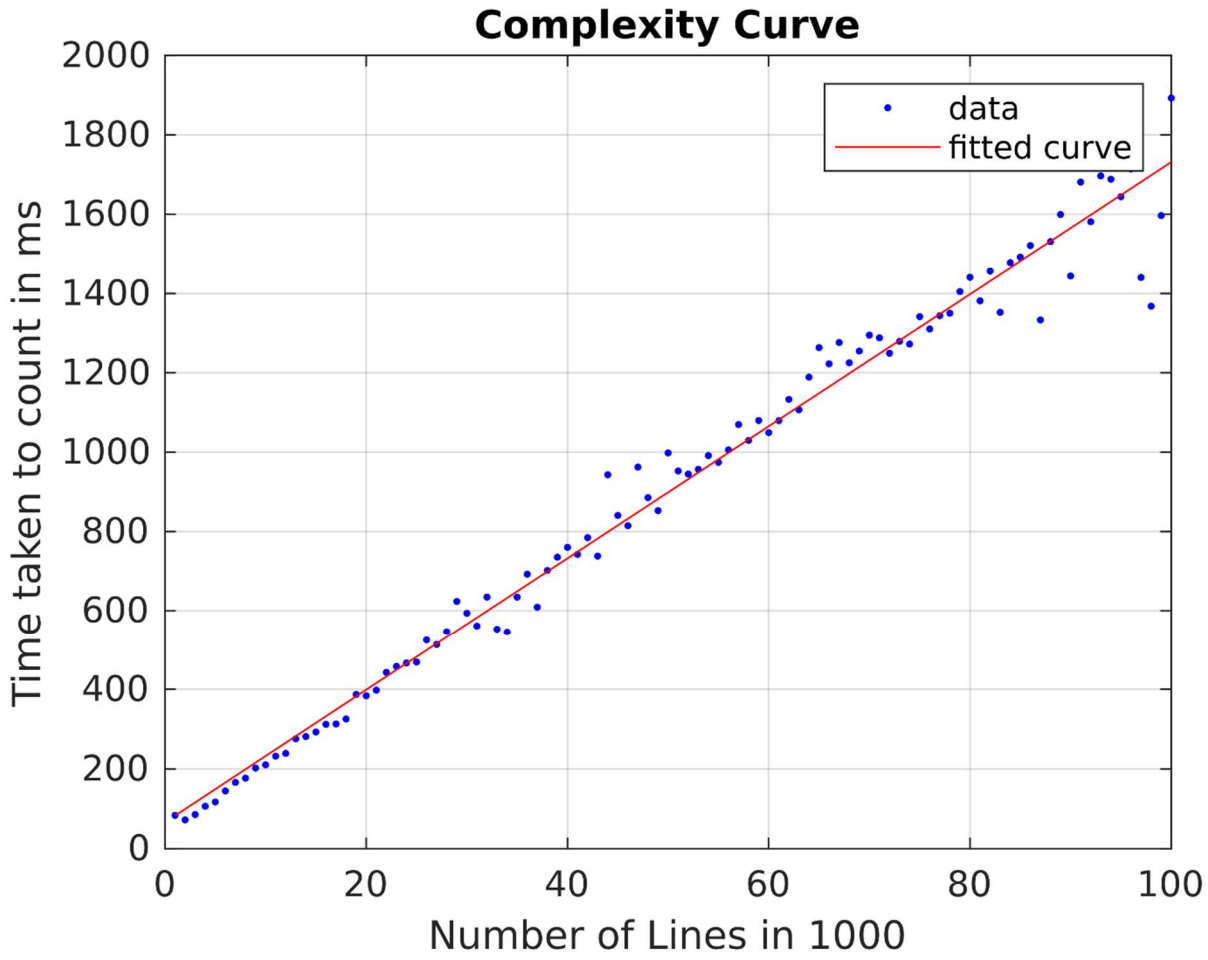
The reason we choose linear regression is because we can observe that our data obtain follow slightly linear approach. To accomplish this task, we created a MATLAB script. This script takes the two data files combines them and form two vectors, number of lines and time duration. Then it uses a fitting model provided by MATLAB (based on linear regression) to find a fitting function. The MATLAB script is given below.

```
function complexity_curve()
    data0 = readmatrix("timedur.txt");
    data1 = readmatrix("timedur1.txt");
    data = (data0 + data1) / 2000;
    fitdata = fit(data(:,1),data(:,2),'poly1');
    f = plot(fitdata, data(:,1), data(:,2));
    xlabel("Number of Lines in 1000");
    ylabel("Time taken to count in ms");
    title("Complexity Curve");
    grid("on");
    exportgraphics(f, "timecurve.png","Resolution",600);
end
```

Command Description-

- `readmatrix()` – reads the data file and returns a matrix
- `fit()` – finds the fitting curve, ‘poly1’ denotes linear curve
- `plot()` – plots the data and the fitted curve
- `exportgraphics()` – exports the graph form as an image file

Now everything at hand, we can plot the curve and analyse its properties. The plotted curve is as below.



The blue points are the data points and the red line in the fitted curve. We can see that the curve nearly fits all the data points. We can write the time and number of lines relationship by following equation.

$$T = aN + c$$

Where T is time, N is number of lines, a and c are two constants. We can write the above equation as below.

$$O(aN + c)$$

Since in time complexity calculation we ignore the constants then the final equation becomes $O(n)$.

Amendments

$O(n)$ is a linear time complexity and is one of the best time complexities only after $O(\log n)$ and $O(1)$. Constant time complexity is not possible in this case because our input is increasing, and it will take some time to analyse the input. Logarithmic time may be possible but for this case average linear time complexity is out best bet. Since there is no way to count line without going through each line, there may a constant time improvement, but it will mostly be close to linear time.

Thank you