Assignment 1 : Report

**Stage 1: Input generation**

Input Size Restriction on Memory

In the approach which was tried, lists of different sizes were created. The script takes as input a number from the user and iterates a for loop n number of times, creating a bigger list each time. The length of the list created varies as per below. (Example shown)

If n=6

First Iteration : Length = 2^0 = 0

Second Iteraion : 2^1= 2

Third iteration : 2^2= 4

Fourth iteration : 2^3 = 8

Fifth Iteration : 2^4 = 16

Sixth Iteration : 2^5 = 32

Here, It can be seen that the size of the list increases exponentialy each time. The purpose here, is to determine the maximum size of the list that can be created until a memory error is encountered. This is attained using trial and error and was found that the maximum size of list possible was oof length 2^26 = 67108864

This was when the entire RAM gets used up and the process is killed automatically by the OS.

It was also ensured to delete the list after each iteration to free up the loop, and after implementing this update, we were able to go up by one power.

Ie 2^27= 134217728

Code Name : input\_restriction.py

Usage :**python input\_restriction.py -n <value of n to be tested>**

OR

**python input\_restriction.py --number <value of n to be tested>**

For Help : **python input\_restriction.py –help**

**Note:**

Here, you can see if the code executes successfully or if it gets killed automatically due to memory error.

On my PC, the killing was happening for values of n above 27.

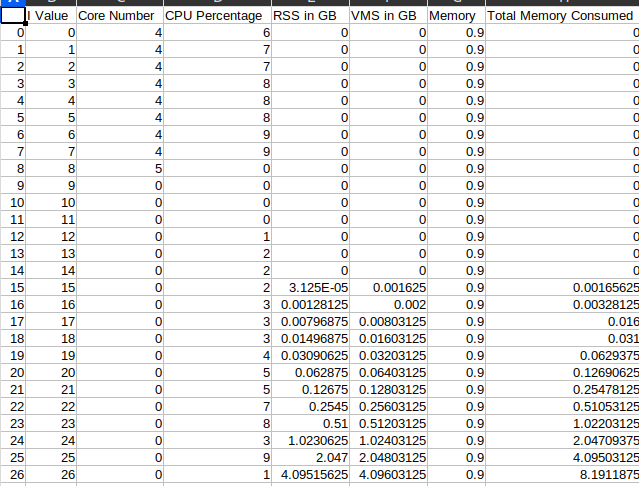
It also generates an excel file which shows the Memory and CPU usage as the arrays are being created

Inorder to ensure that the data logging parameters are not affecting the memory, a new py file called **input\_restrictions\_without\_logs.py** was created. On executing this the input size remains the same. ( 1.1GB np file is generated )

After this the results are saved into a numpy file. .

**Outputs generated with input\_restriction.py :** inputwlogs.np ( Size 1.1 GB)

inputProcess.csv ( Contains the variations of memory and CPU using input generation process )



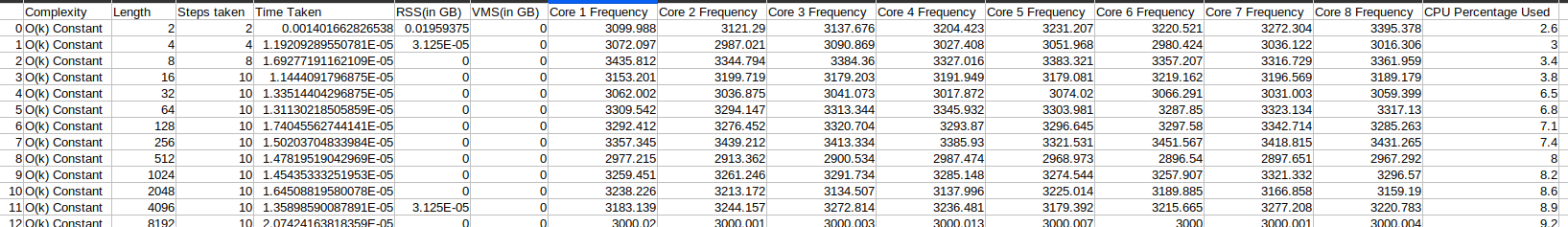
**Observations:**

* The process gets killed when 100% of RAM is used
* Total RAM in PC : 8 GB
* RAM consumed for other process : 4GB approx
* The CPU switches between cores during the process sometimes
* The CPU usage for the PID goes to a max to 10-20% and not more than that.

**Step 2: Array Processing**

The target here is to first open the file which contains the input and then process it under algorithms of different complexity and to monitor the variations. The function generates two outputs. An excel file which has the below details as columns

* The length of input array
* Number of steps which arer taken in each iteration
* The time taken for each iterations
* The RSS and VMS values in GB ( The Physical and Virtual memory utilisation)
* The frequency variations of all the 8 cores in CPU during each iteration
* The CPU utilization perentage during each iteration



A sample file would look like the below screenshot

**Process:**

The process takes as input the array from the previous stage, which has size 536.9 MB and has 33554432 entries inside. The function then takes different sizes of the array in a sequential order such that the size varies from 2, 4, 6 8 upto 33554432. It changes as 2^n when n is iterated in a loop from 1 to 26. Each iteration produces an array of bigger size and it is passed through the algorithm specified by the user and its diferent parameters are mentioned. The summary of the data is then generated as a CSV file and gets stored in the same folder as the code. All these functions gets executed upon calling **dsa.py.**

Usage :**python dsa.py -o <algorithm complexity> -t 10**

OR

**python dsa.py -option <algorithm complexity> --timeout 10**

For Help : **python dsa.py –help**

*Where algorithm complexity : ‘const’ for constant, ‘lin’ for linear, ‘quad’ for quadratic polynomial, ‘cube’ for cubic polynomial, ‘log’ for logarithmic, and ‘exp’ for exponential complexity*

*Timeout function is used to stop the process from getting killed. It has a default value of 240. Which means for each iteration, it will wait 3 minutes for the loop to be completed. If the iteration doesn’t gets get completed in 240 seconds, the code will exit, but all the data will be logged in the excel file.*

*This default value of 240, can be overwritten by using the -t or –timeout option by specifying the seconds for which the program should wait before timeout.*

Once the required files are created, call **plot.py** to generate the corresponding plots and to save them in the same folder as code.

Usage :**python plot.py**

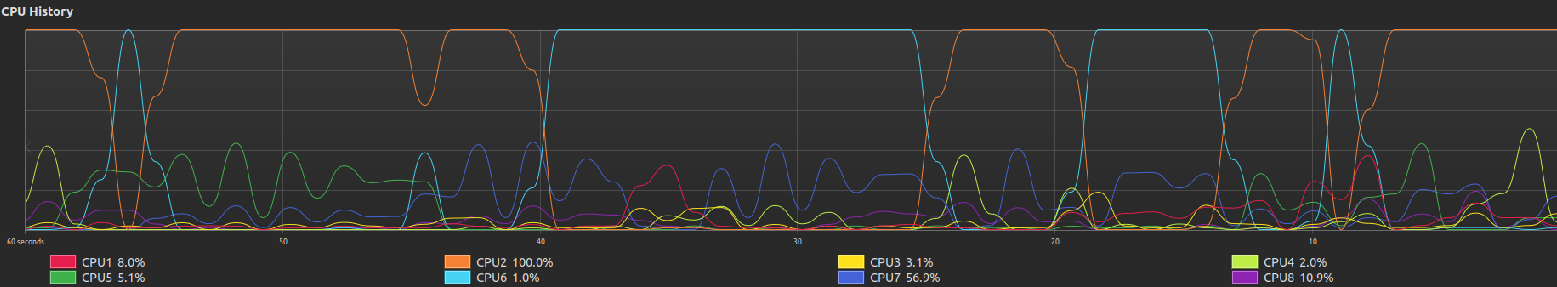
OR

**python plot.py**

**Plot.py** scans for all csv files within the folder, which are the files generated as a result of the dsa.py script. Plot.py scans all the csv files and converts a set of plots from each file and saves the png file of the plots in the same folder for each file.

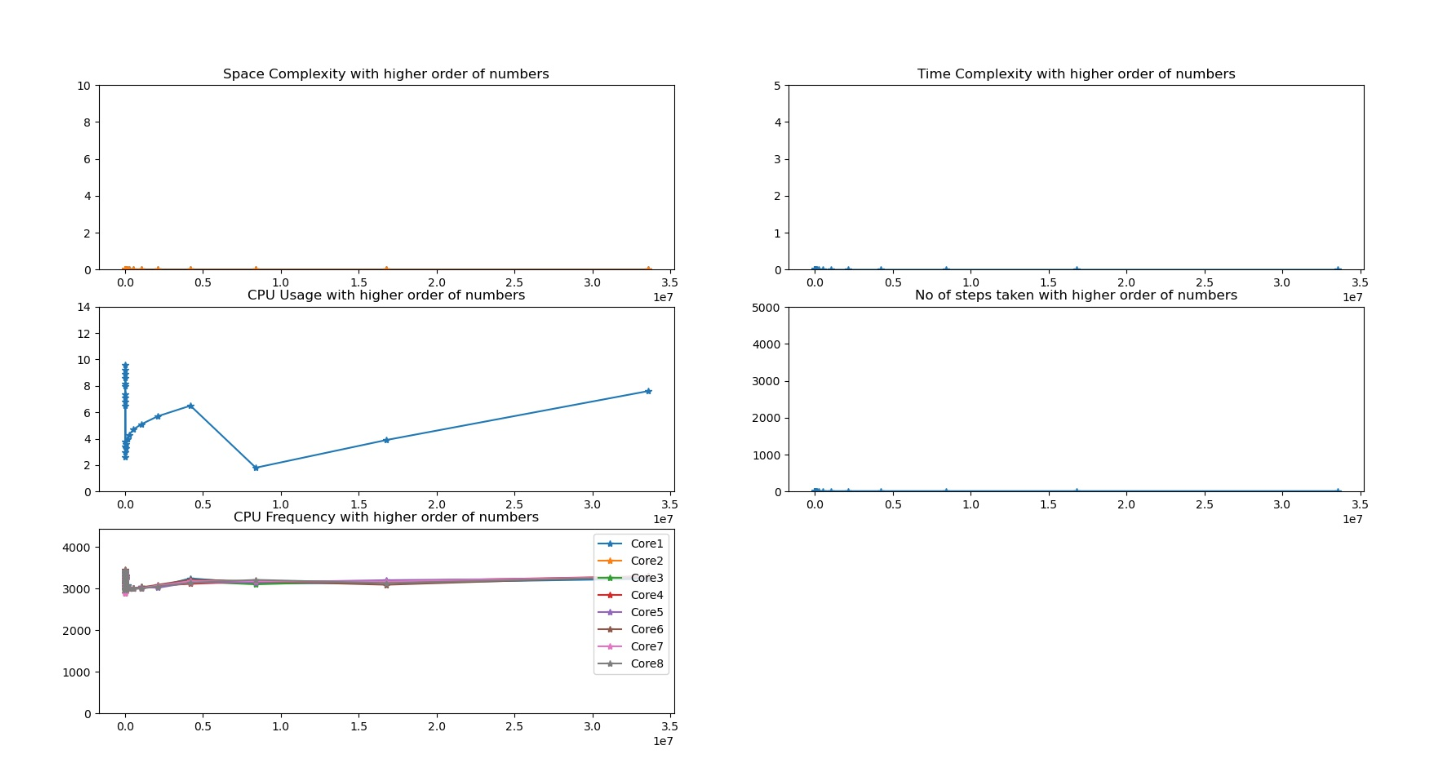
**Observations :**

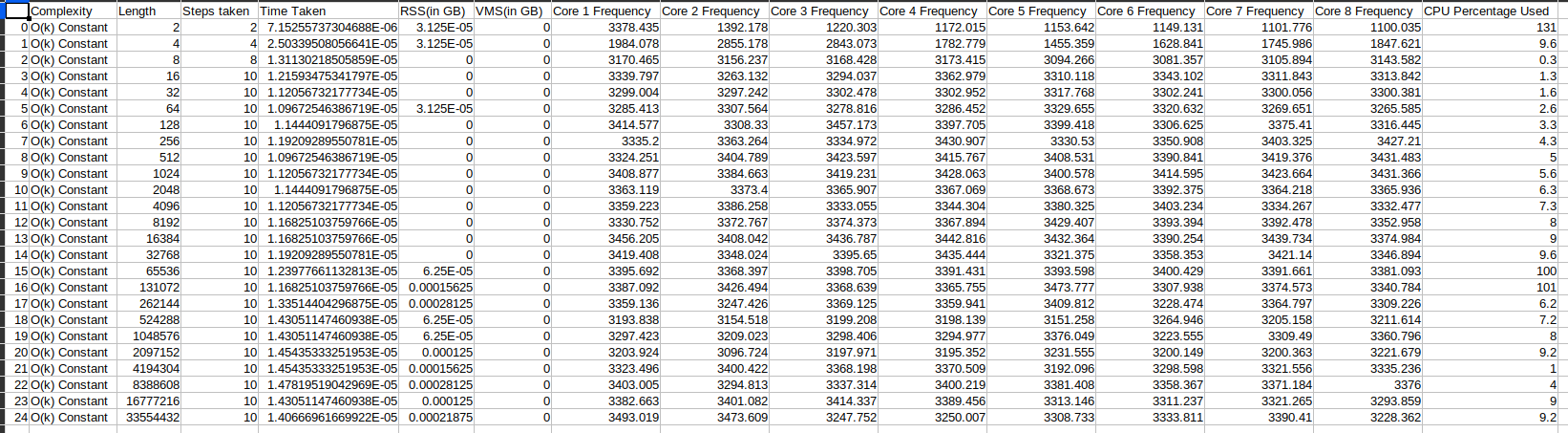
* The first observation was that the program was being automatically killed on trying to open the 1.1 GB Input file which was created earlier.
* For the process to work properly, the size had to be reduced from 2^27 to 2^26 . Hence reducing the size of the input file to load.
* The entire input file was loaded into the program at one using the np.load method and the process took around 2-3 seconds on an average to read the entire input file.
* The script gets killed by the OS for cubic and exponential complexity algorithms as the length of the array increases. Inorder to avoid this exception, the timeout option was used to reduce the timeout to 5 seconds for Cubic complexity.
* The exponential complexity gets timed out at the default 240 seconds.
* It is seen that when time complexity is low, the space complexity is usually high.
* The Core 1 of the CPU is usally processing the most when the codes were executed.
* It was seen that the cores keep switching when the usage gets too high on a core. ( Blue and Orange cores keep alternating when they become 100% )



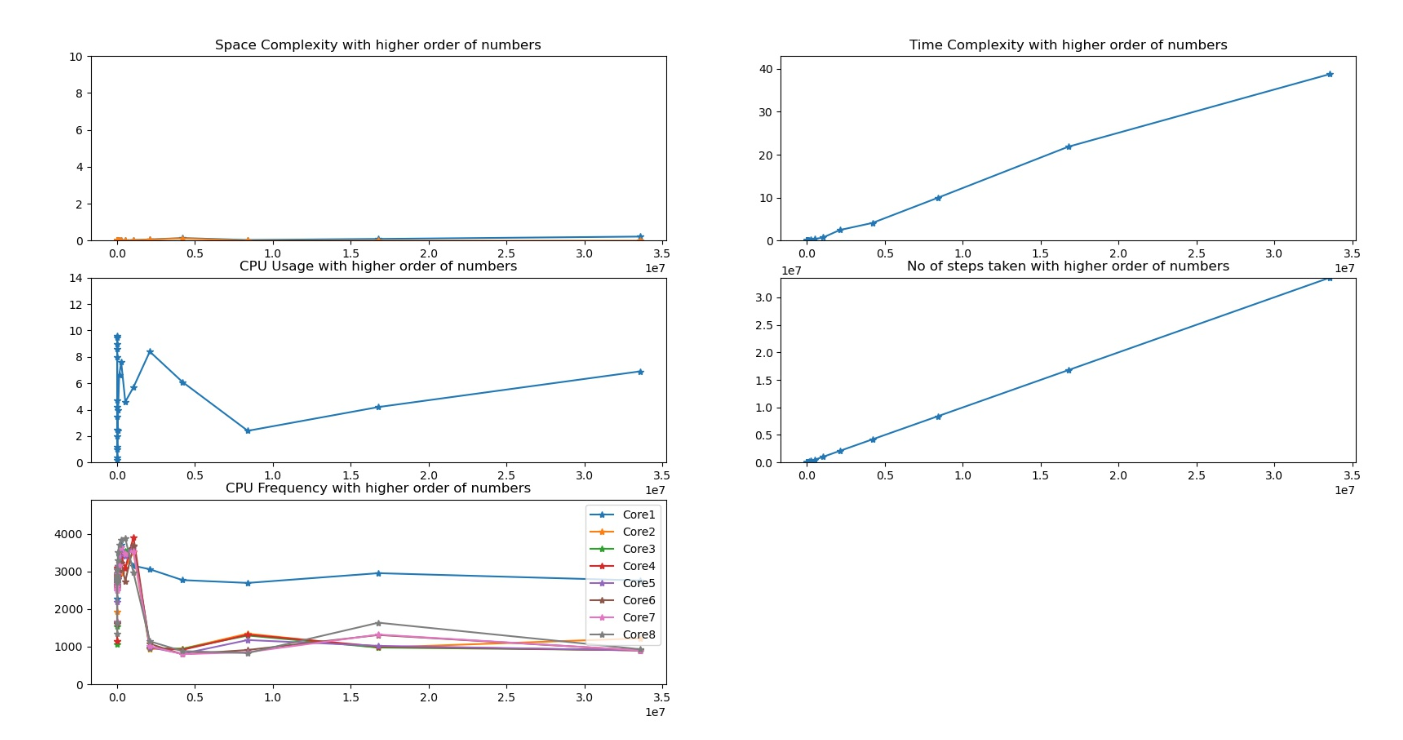
**Observations for different algorithms**

Constant Complexity Algorithm

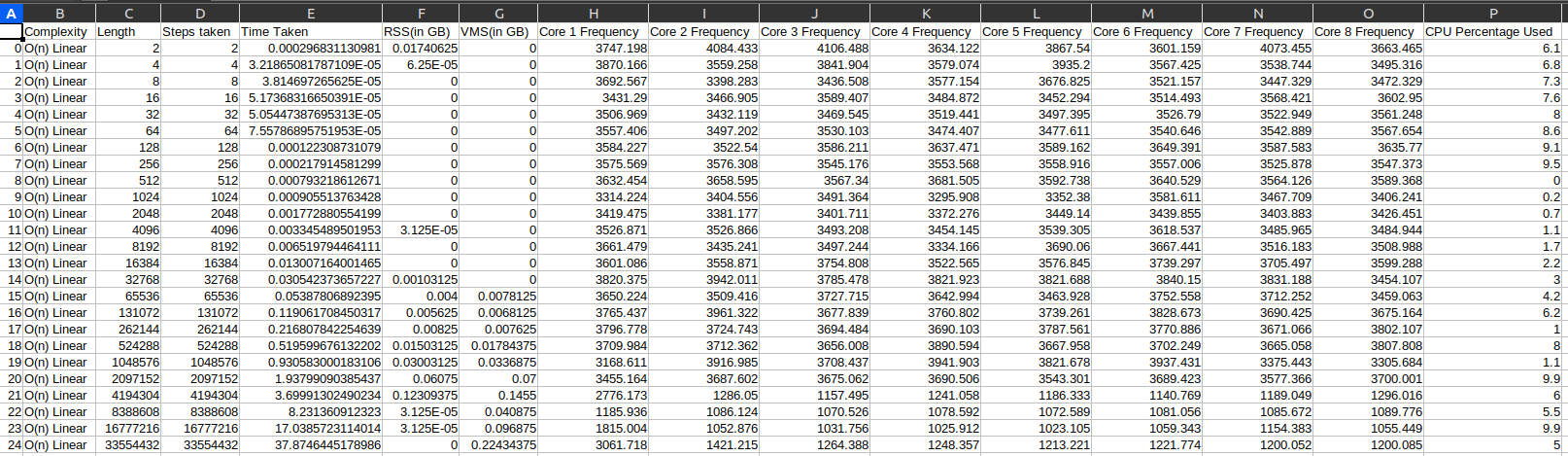
* For the algorithm with constant complexity, the number of steps always remain the same. (10 steps)
* The time taken is also almost the same ( Around 1.9073486328125E-05)
* The CPU frequency also remains a constant without much fluctuations. ( Around 3Ghz )
* The Space complexity,or the memory needed also remains low. The maximum memory used being 0.00021875 GB
* The CPU Utilization remains below 7 percent.
* It remains as a stable process and the entire array gets executed with ease irrespective of the size.

Generated Data

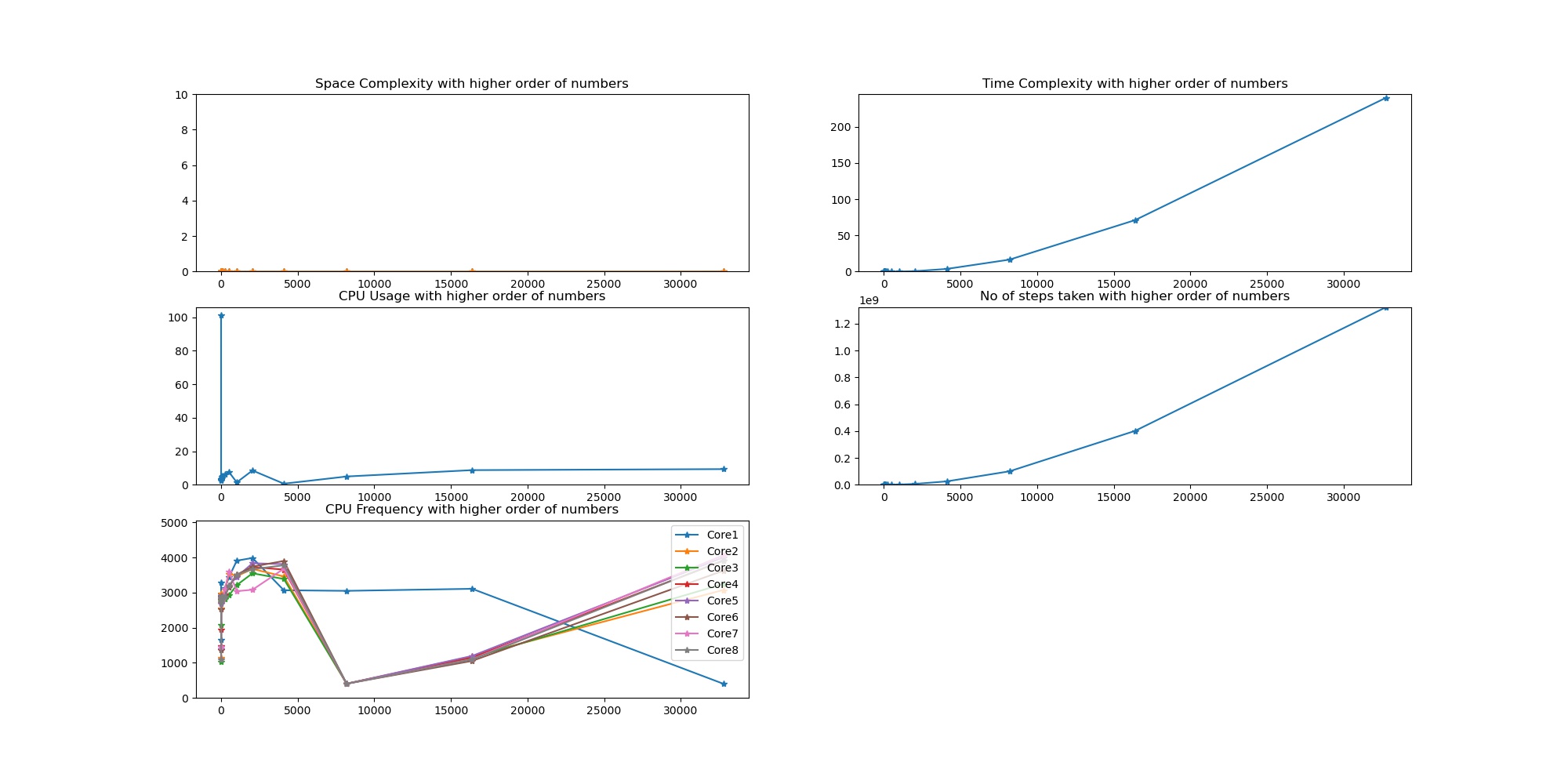
Linear Complexity Algorithm



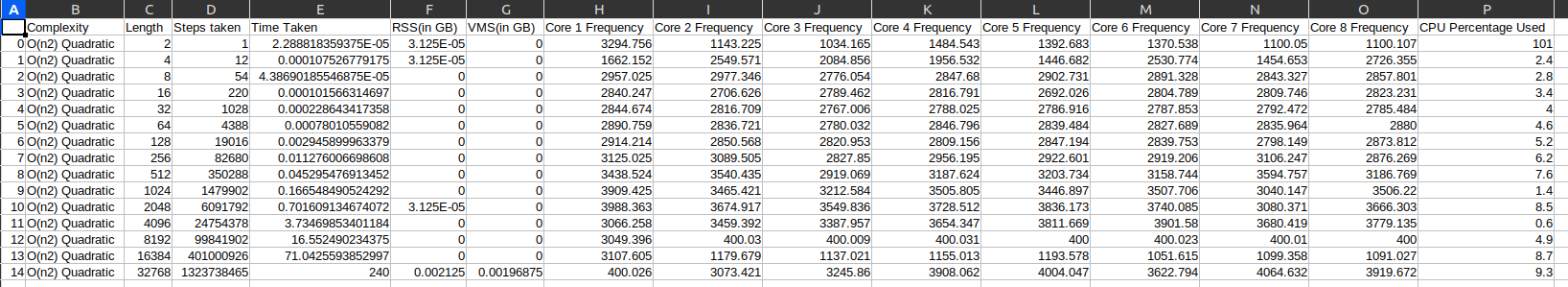
* In this approach the number of steps keeps increasing as the size keeps increasing in a linear way as in graph 4
* The Time complexity also increases in a linear way, with maximum time being close to 40 seconds for the largest size
* The memory utilization keeps a very low value consuming close to 0.2GB
* All the array sizes runs for this algorithm without any memory constraints.
* The maximum CPU Usage is less than 6%



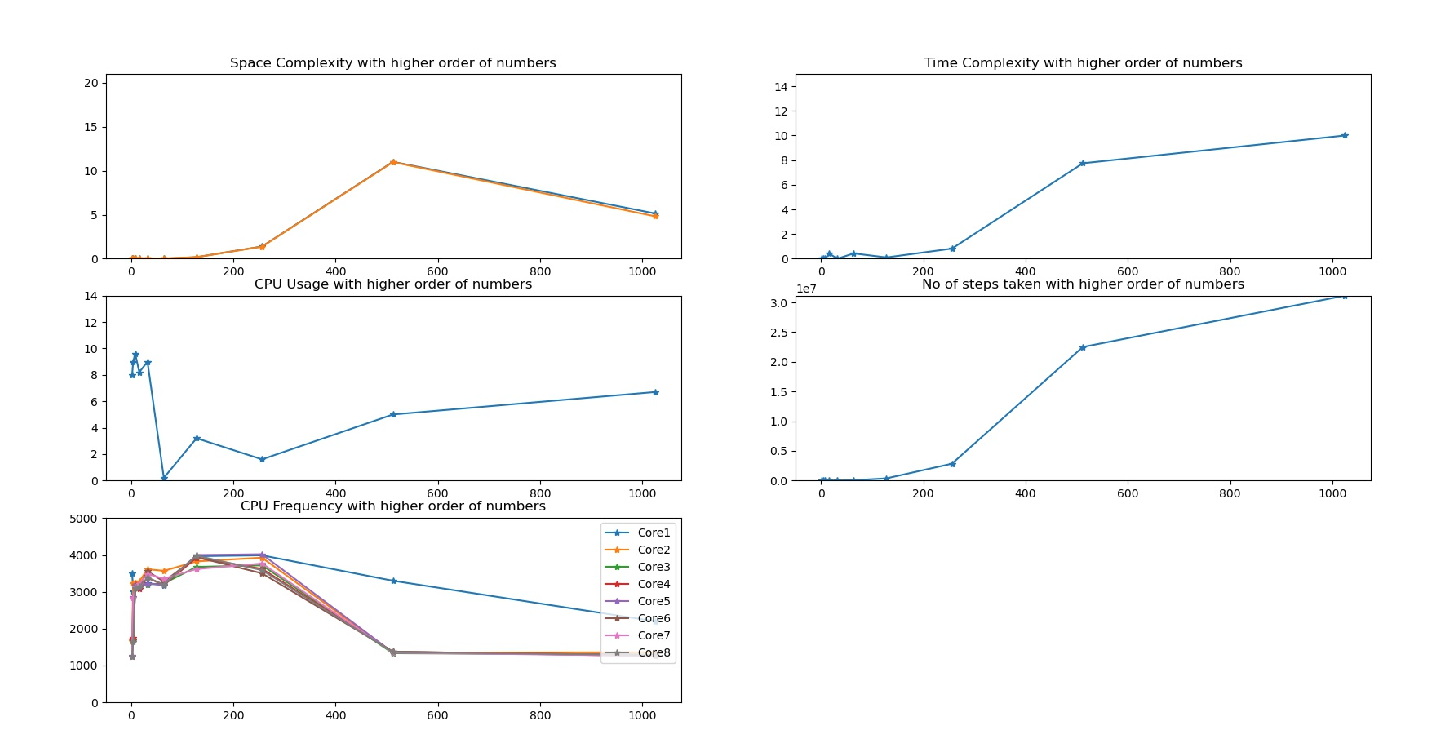
Polynomial Complexity Algorithm ( Quadratic )



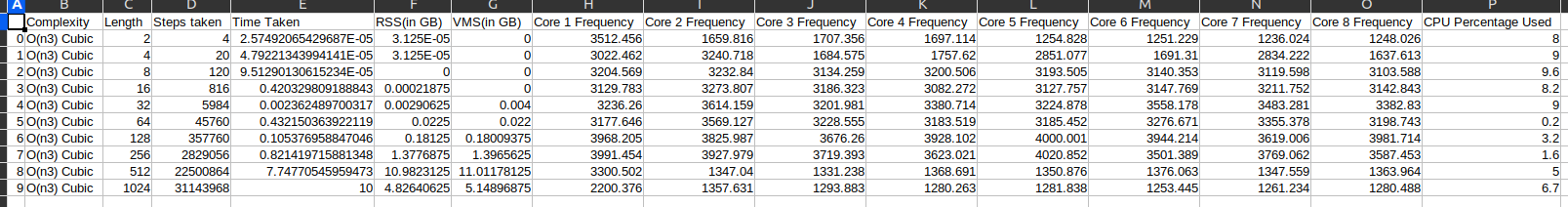
* This follows a complexity of nsquare.
* It can be seen that the time complexity increases sharply as the length of input increases.It goes to a peak of 240seconds, and fails to complete the proces. This happened when the length of input was 32768.
* The number of steps can also be seen to be increasing sharply, which seems to be correlated with the time taken.
* The process consumes excessive memory at input length of 32768, and gets killed due to memory error.
* It consumes very little memory throughout the entire process, even though the time complexity is high.
* Hence it has very low space complexity.
* The Maximum CPU usage remains around 9%.

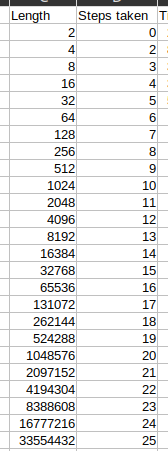


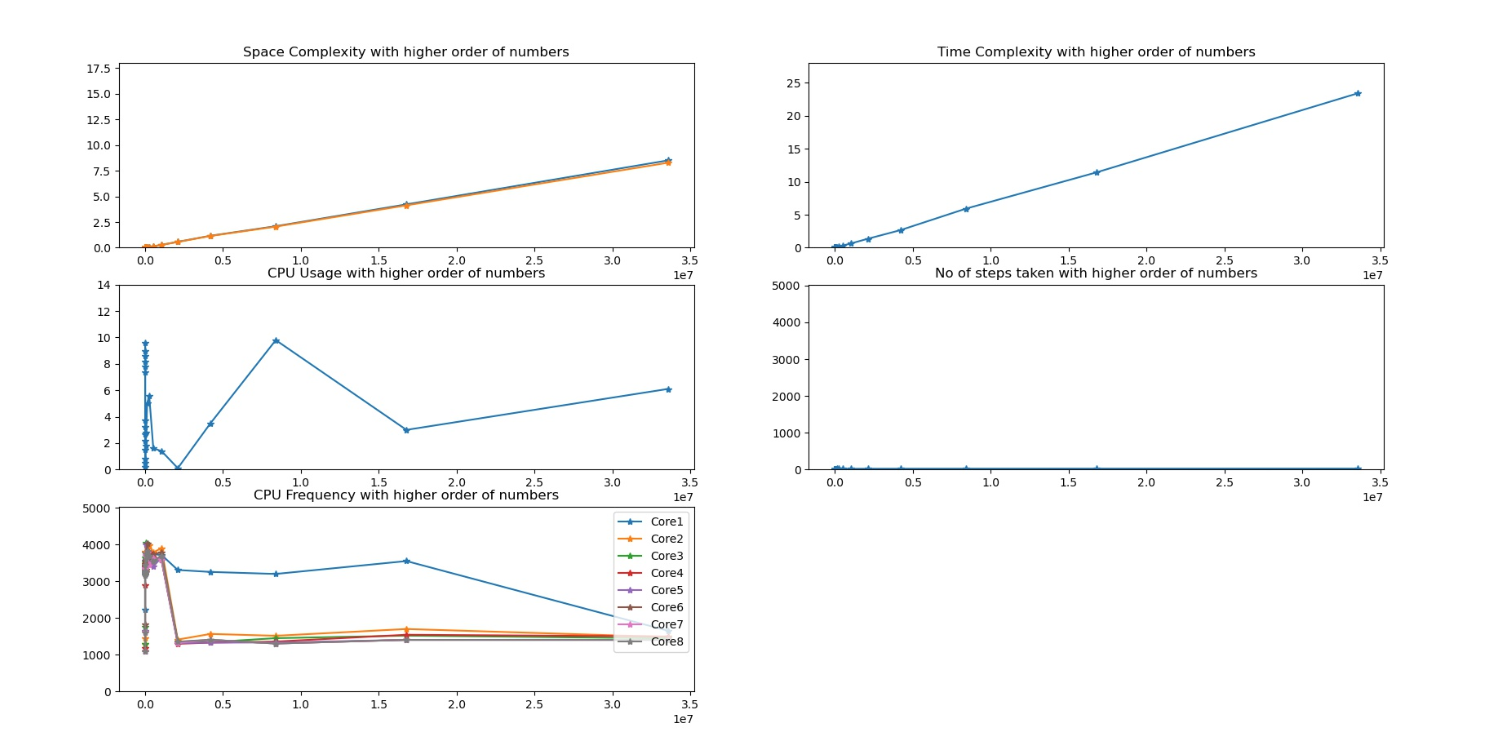
Polynomial Complexity ( Cubic )



* The cubic algorithm gets times out at an earlier stage. (When the input length is 1024 )
* It uses almost 4GB of Physical Memory and 5 GB of Virtual memory, which causes it to be killed by the system.
* At first the program was killed by a default timeout of 240 seconds. It got killed even on reducing timeout to 10 seconds.
* However, after reducing the timeout to 5seconds, we were able to stop the process from getting killed.
* Hence the last value which causes a dip in all the graphs is an incomplete process and is giving a wrong picture
* It can be seen that the space and time complexities are increasing sharply than a Quadratic process and also gets terminated much earlier.
* It can also be seen that the number of steps taken increases sharply in each iteration.
* Here, again the CPU usage remains at less than 10%.



Logarithm Complexity Algorithm



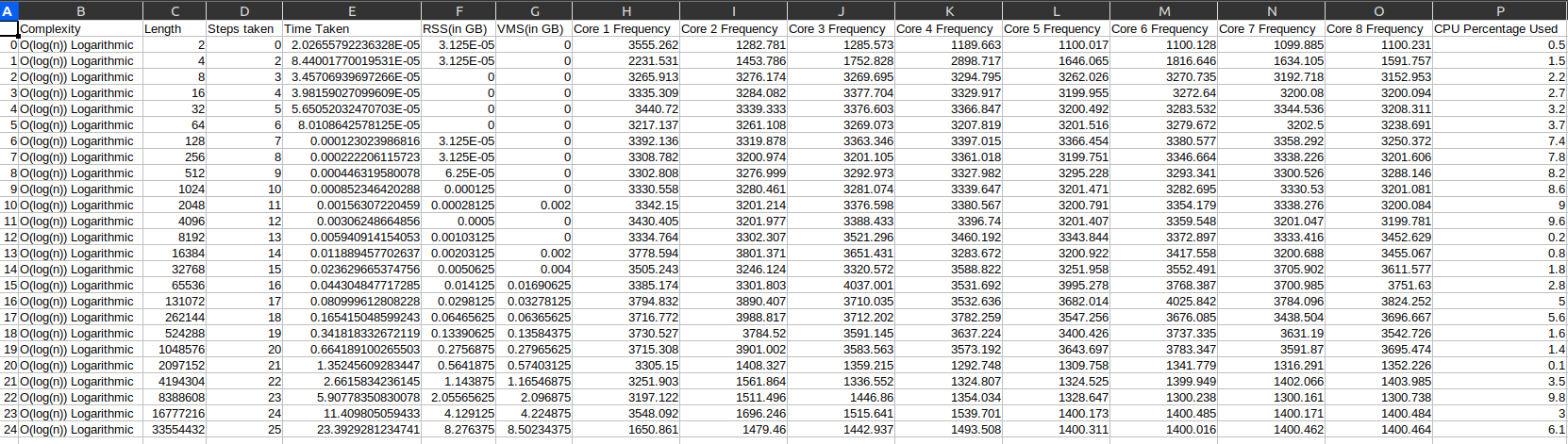
* In Logarithm complexity algorithm, the number of steps increase in a very slow fashion.

Image attached to right of graphs

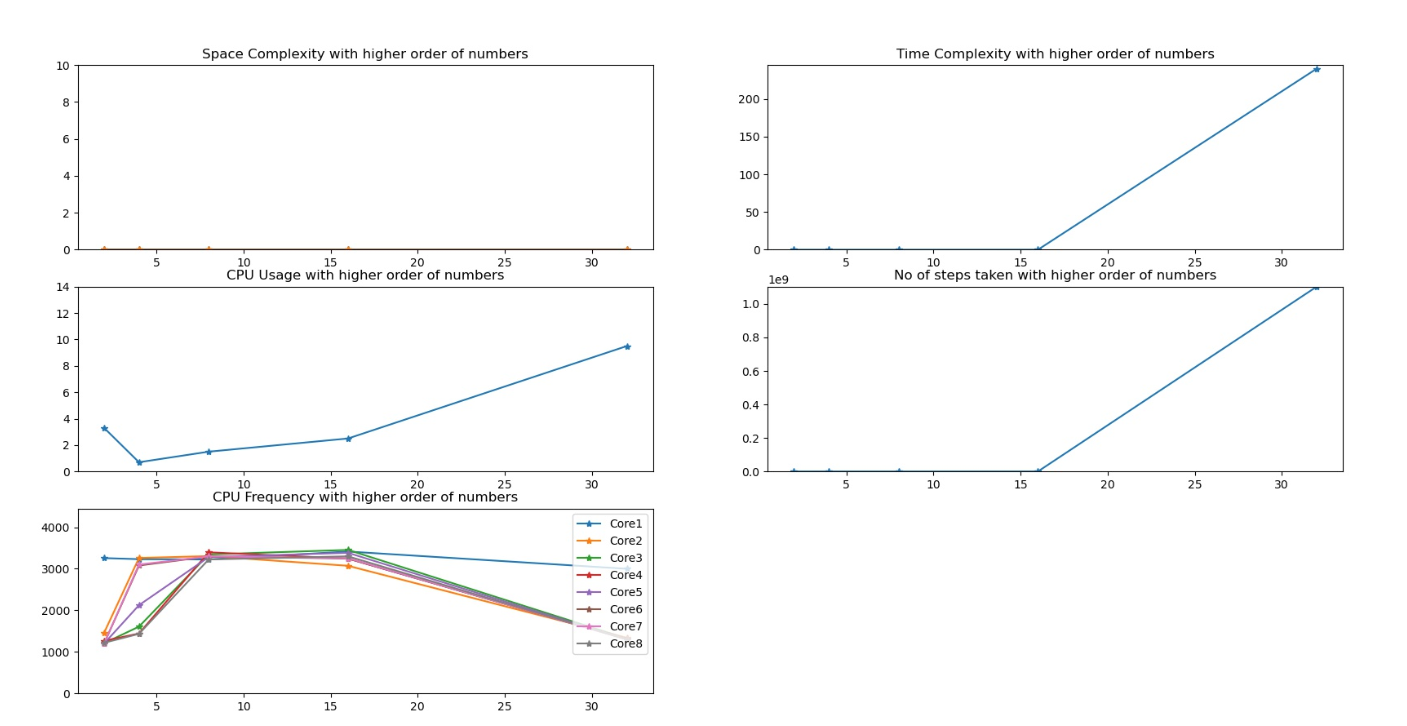
* Hence, the time taken for the process is very less. Although, it follows a linear growth, it can be seen that the maximum time taken for

an input of size 33554432 would be 23 seconds.

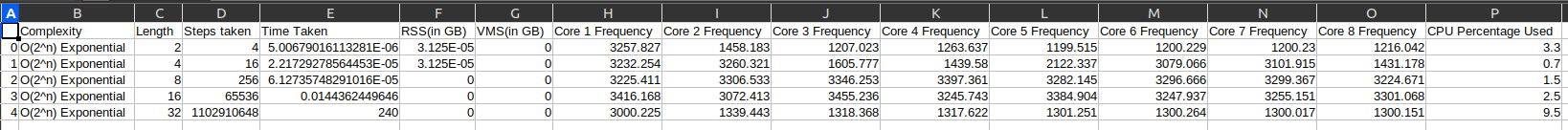
* However, it can be seen to be using a higher amount of Physical and Virtual Memory.
* The process gets completed at ease without any timeouts and all inputs are processed.



Exponential Complexity Algorithm



* In exponential complexity algorithm, the steps increases as a power of 2. The same can be seen in the fourth graph as it starts rising steeply beyond a point.
* A similar pattern can also be observed with time complexity with execution times exceeding 240 seconds. The process was not even completed at 240 seconds and had times out.
* The timeout happened when the input size was 32,hence making it the algorithm which timeout the fastest.
* Even with high time complexity, it can be seen that the space complexity of this algorithm is very low. Hereby proving that space and time complexity are inversely related.



Script Automation

As it can be time consuming to generate the file and to do the process, a bash file has been configured to do the testing and to generate the required output files.

Usage:

* Place the files : input\_restriction.py

dsa.py

dsaplot.py

automate.sh , all in the same folder

* Open Terminal in the folder
* Type “chmod +x automate.sh”
* Type “bash automate.sh”
* The file will not start executing all the scripts in backend and a few files will be generated after some time
  + inputwlogs.npy
  + O(k)constant.csv
  + O(2^n)exponential.csv
  + O(n^2)quadratic.csv
  + O(n^3)cubic.csv
  + O(logn)logarithmic.csv
  + O(n)linear.csv
  + O(k)constant.png
  + O(2^n)exponential.png
  + O(n^2)quadratic.png
  + O(n^3)cubic.png
  + O(logn)logarithmic.png
  + O(n)linear.png