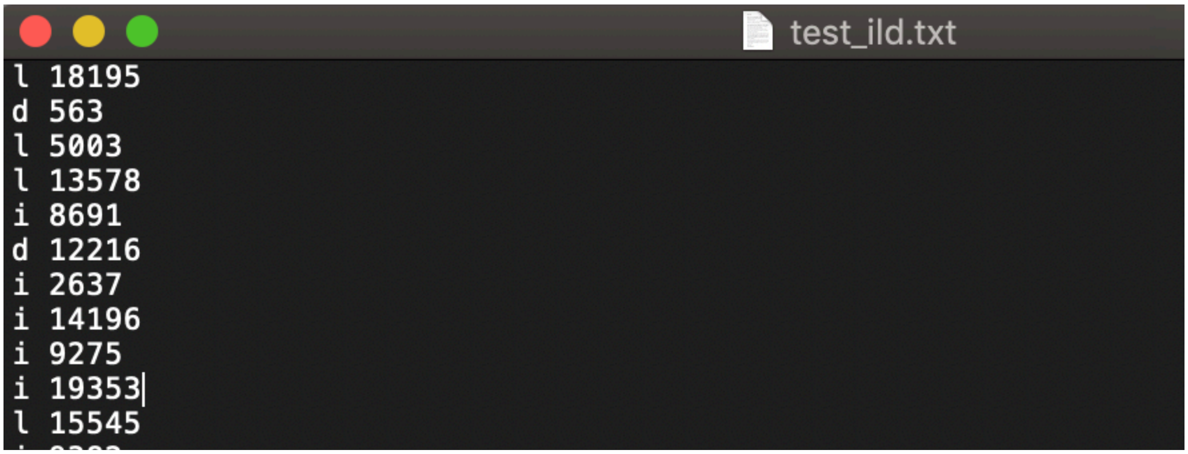
General steps for both the implementations:

1. **Input File:**
    - Is the Is the file which contains elements to insert, lookup or delete represented by shorthand i, l and d respectively as shown below:



test_ild.txt

```
l 18195
d 563
l 5003
l 13578
i 8691
d 12216
i 2637
i 14196
i 9275
i 19353
l 15545
```

    - This file is generated randomly by python script (randUnique.py), which generates n unique random numbers in the range [0,50000) with 60% probability of associating each of the n numbers with i and 20% each for l and d.
      (Usage: python randUnique.py <outputFileName> <n>)
2. **Output File:** The output file logs the following things:
    - Initial hash functions and any change in hash functions.
    - For all the insert operations it records the element, time in micro-seconds and load Factor.
    - This file can later on be used to plot graphs if need be.
    - At the end of the file the total time taken during that run will be printed in mili-seconds.

Linear Probing:

1. **How to invoke:** python **linearProb.py** <inputFileName> <outputFileName>

2. **Design decisions**:
    - **Size of tables (N):** Initially the size of table is 100 and it keeps on resizing to twice the size when more than 75% of the table is filled. If the elements contained in table are less than 25% then it resizes to half the size. The parameter N is configurable at the start.
    - **The Hash Function ($H_1(x)$):** The function is one degree polynomial mod by large prime number mod by size of table.
      $H_1(x)$: $((a_0 + a_1x) \bmod p_1) \bmod N$

Initial values of these constants are $a_0 = 7$, $a_1 = 3$, $p_1 = 393241$ and these are configurable.

- The time in micro-seconds is the time to insert the current element successfully in the hash table.
- The data is accessed in sequential manner but is randomly generated.

3. Pseudo code:

Initialize configurable parameters
Initialize user-based parameters (file names)
Initialize two hash tables based on value of N
Open file handlers and initialize files
Initialize global variables

Function to find the best fit line given two lists x and y.
[Func: *best_fit(x,y)*]

Function to plot scattered and log-log graph based on the pair (time for insert, load factor at that time).
[Func: *plotGraph(x,y)*]

Function to resizes the hash table to another size and rehash elements in the new table.
[Func: *resizeHashTable(size)*]

Function to calculate the expected index of element based on hash function and return the value.
[Func: *calculateIndex(number)*]

Function to insert element in hash Table by doing a linear search in case element is not found at expected location notes down the time and load factor in output file on successful insertion. [Func: *insertIntoHashTable(element)*]

Function to look up the element at the possible location and if not present does linear search till either -1 is encountered or we come back to the previous location from which we started. [Func: *lookUpInHashTable(element)*]

Function to delete an element which searches the element just like lookup and if found then puts -2 at its place and also rehashes when the occupancy of table is less than 25%.
[Func: *deleteFromHashTable (element)*]

Start reading the input file line by line

While(not EOF):
    Extract command and element

Switch(command):

Case i: insertIntoHashTable(element)
Case l: lookUpInHashTable(element)
Case d: deleteFromHashTable(element)

Read new line from input file

Output the index and corresponding element present at that location

Plot scattered and log-log graph and show

Close file handlers

4. Testing:

To test scaling, I tested the algorithm on test files with 1000, 9000, and 20,000 inserts. Each file was generated by randUnique.txt. I have also tested scaling on file with insert, lookup and delete operations which scales at-least till 30000.

The files considered for analysis are as follows:

1) **test_1000.txt:** File with 1000 insert operations. The graph of this is as follows:

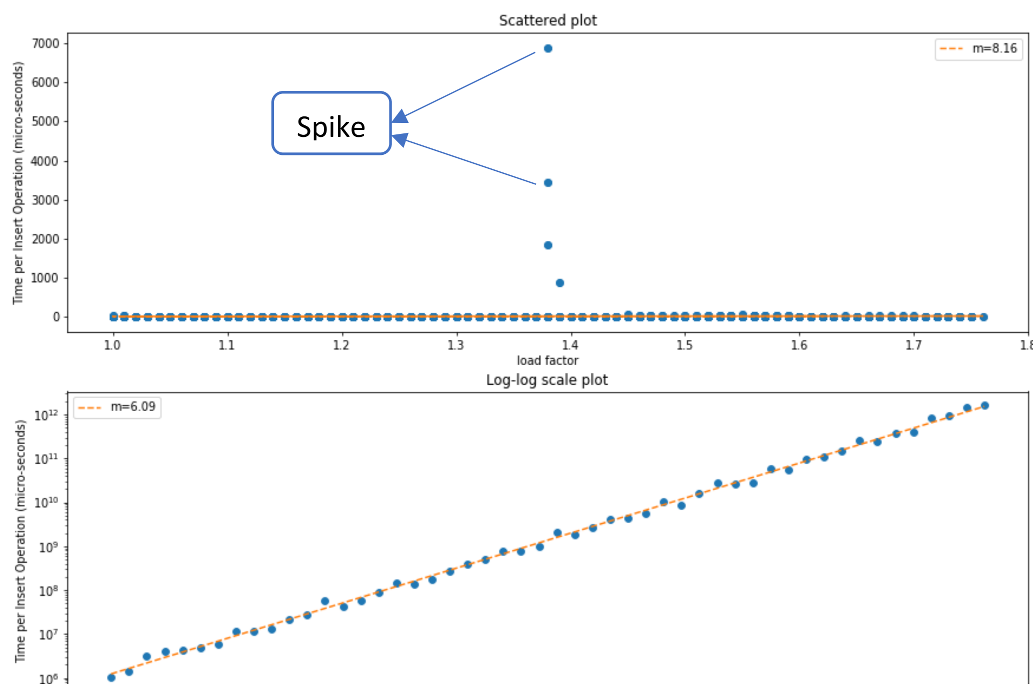


Figure 1: Test_1000

2) **test_9000.txt:** File with 9000 insert operations.

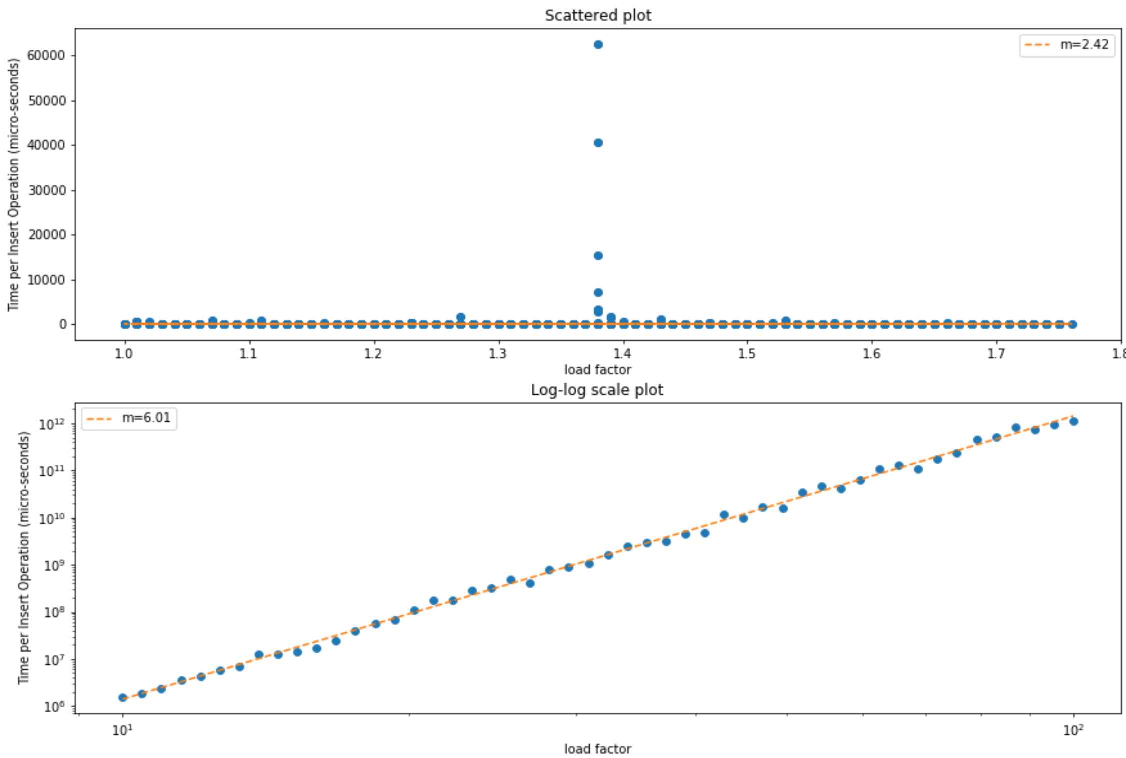


Figure 2: Test_9000

3) **test_20000.txt:** File with 20,000 insert operations.

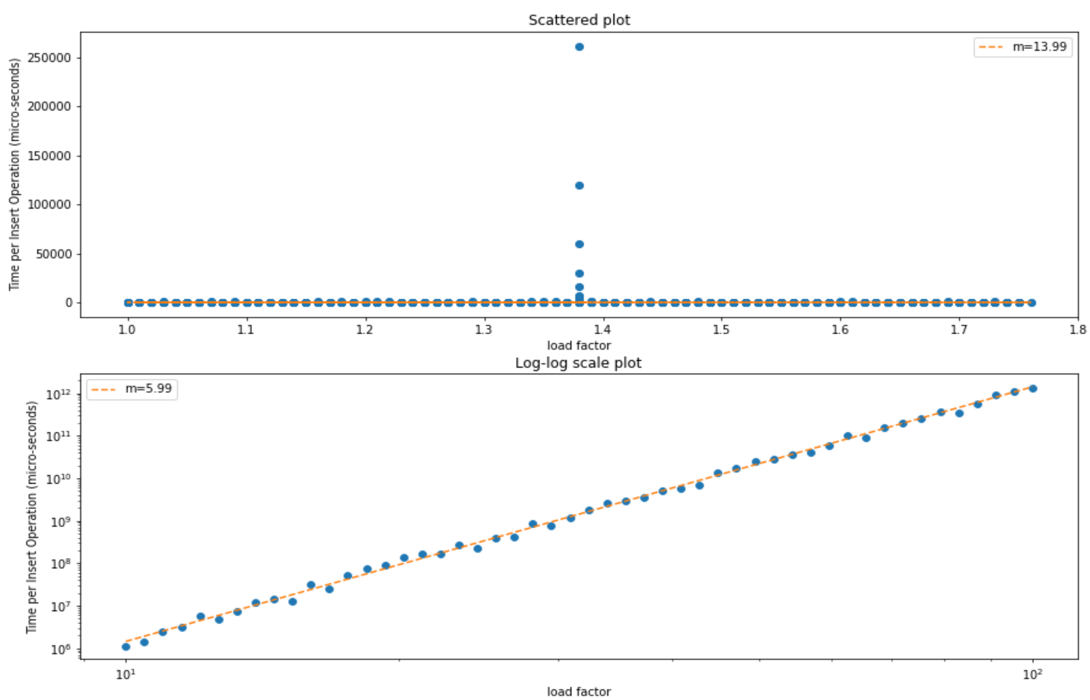


Figure 3: Test_20000

4) **test_ild.txt:** File with 30000 operations of which probably 60% are insert, 20% are delete and 25% are lookup.

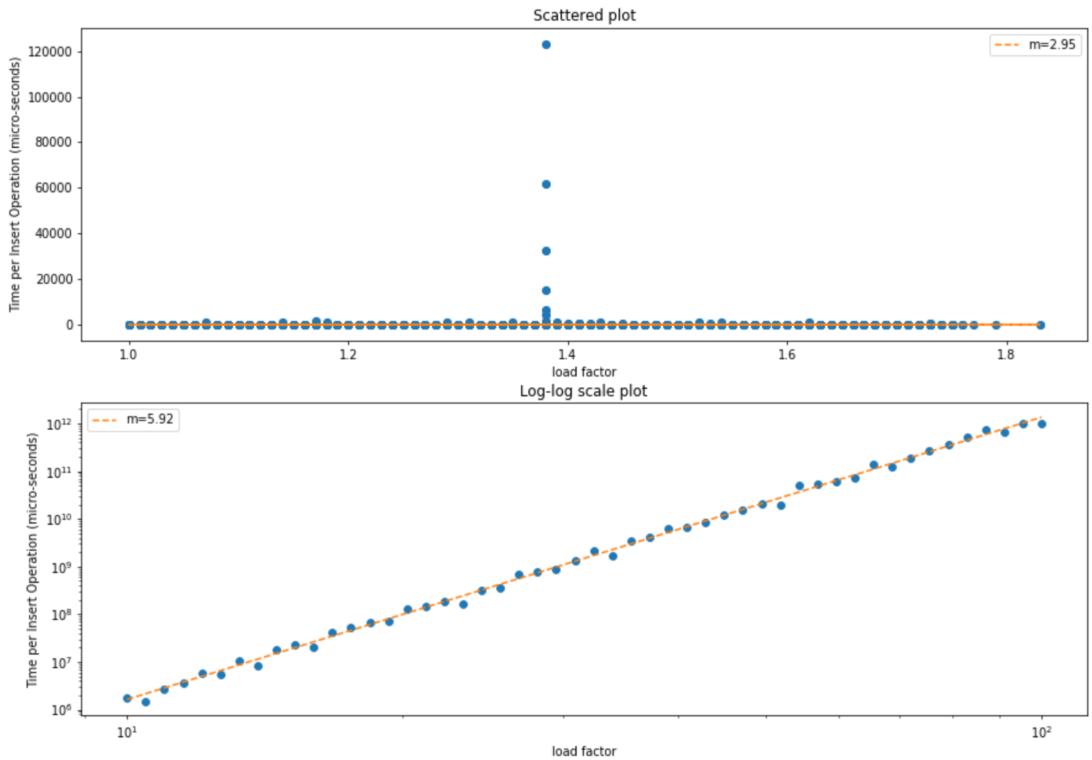


Figure 4: test_ild_30000

5. Observations/Conclusions:

1) The scattered plot of Figure 1 having 1000 insert operations has greater slope than for Figure 2 having 2000 insert operations. We know from the output file that for 1000 inserts we have resized the table 4 time last of which was done at 601^{th} insert to increase the size to 1600, while for 9000 inserts we have resized the table 7 time last of which was done at 4801^{th} insert to increase the size of table to 12800. We observe this increase in time per operation(m) is less for table with 9000 insertions due to lesser probability of collisions as relatively more space i.e. 3800 is available in it than 600 for 1000 inserts at the end of insertions.
2) The slope of log-log plots of all the Figure 1, Figure 2, Figure 3 and Figure 4 revolves around 6, as we have a much tighter bound with log-log(1+load Factor(x), time in milli-seconds(y)) plot when we assume a $y = x^6$ relation.
3) In the hash function ($(a_0 + a_1x) \bmod p_1$) mod N value of p_1 highly affects the time taken per insert operation. Based on the observation that if the value $a_0 + a_1x > p_1$ then there are more collisions and hence we move to linear search which takes a lot

of time. On the other hand if value of $a_0 + a_1x \ll p_1$ we get evenly spread data and frequency of collisions decrease.

4) The time per insert directly varies with the load factor and hence as soon as load factor reaches around 0.7-0.8 the performance of insert operation degrades drastically. Hence, we maintain the fact that at any point in time load factor <= 0.75 and load Factor >= 0.25 except at the very beginning.

5) The extra time spent on resizing and rehashing can be observed by the spikes in the Figure 1 at around load factor of 0.4. This extra time spent compensates for the reduced time observed for the following inserts.

6) The hash map implementation scales for any number of insert operations based on the amount of memory available.

Cuckoo Hashing:

1. **How to invoke:** python **CHashing.py** <inputFileName> <outputFileName>

2. **Design decisions**:
   - **Size of tables (N):** The size of each hash table is pre-determined to be 10,000 but can be configured in the file by changing the value of N.
   - **The Hash Functions ($H_1(x)$ and $H_2(x)$):** The functions are second degree polynomials mod by large prime number mod by size of table.

     $H_1(x)$: $((a_0 + a_1x + a_2x^2) \bmod p_1) \bmod N$

     $H_2(x)$: $((b_0 + b_1x + b_2x^2) \bmod p_2) \bmod N$

     Initial values of these constants are $a_0 = 101$, $a_1 = 3$, $a_2 = 5$, $b_0 = 2$, $b_1 = 7$, $b_2 = 11$, $p_1 = 1299821$, $p_2 = 982381$ and these are configurable.

     For change in hash functions: We choose a random value from list *smallPrimeNumbers* for variables a_0, a_1, a_2, b_1, b_2 and b_3 and we choose the next available value from the list *primeNumbers* for p_1 and p_2.
   - **Detecting a loop:** A loop is detected if we try to insert an element x in the first hash table 4th time.
   - **To break loop:** we try to rehash the elements present in both the tables by changing the hash functions.
   - The time in micro-seconds is the time to insert the current element successfully in the hash table.
   - The data is accessed in sequential manner but is randomly generated.

3. **Pseudo code:**

   Initialize configurable parameters
   Initialize user-based parameters (file names)

Initialize two hash tables based on value of N
Open file handlers and initialize files
Initialize global variables

Function to find the best fit line given two lists x and y.
[Func: *best_fit(x,y)*]

Function to plot scattered and log-log graph based on the pair (time for insert, load factor at that time).
[Func: *plotGraph(x,y)*]

Function to change hash functions.
[Func: *changeHash()*]

Function to calculate the index of element in both the hash tables.
[Func: *calculateIndex(number)*]

Function to insert element in appropriate hash table and if loop is detected returns false else returns true. [Func: *insertIntoHashTable(element)*]

Function to resize hash tables to twice the size and rehash elements (currently not used)
[Func: *resizeAndRehash (element)*]

Function to rehash elements in hash table using a temporary table.
[Func: *rehash (s, element)*]

Function to lookup the element in both the hash tables at appropriate position.
[Func: *lookUpInHashTable(element)*]

Function to delete an element from either of the hash tables using the appropriate position. [Func: *deleteFromHashTable (element)*]
Start reading the input file line by line

While(not EOF):
    Extract command and element

    Switch(command):

    Case i: Try to insert the element and if loop is detected then change hash function and rehash while rehashing not successful.
        On successful insertion increase number of current elements and recompute load factor
        Also, note time taken to insert and current load factor in output file and in lists loadX and timeY.

Case l: Lookup the element in two possible positions and print in which table it was found else print not found.

Case d: Delete and element by putting -1 at its place in either of the two hash tables where it is found else print not found.

Read new line from input file

Output the index of table 1 and table 2 with elements present at those locations if they both are not empty.

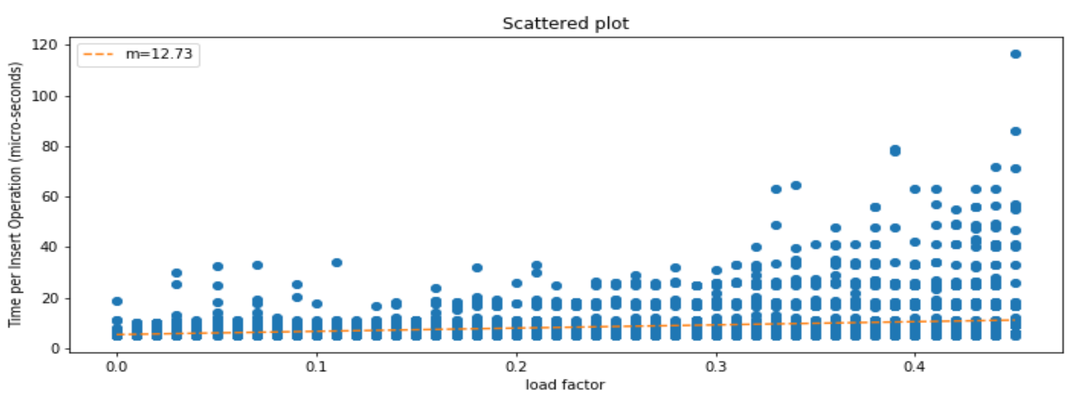
Plot scattered and log-log graph and show

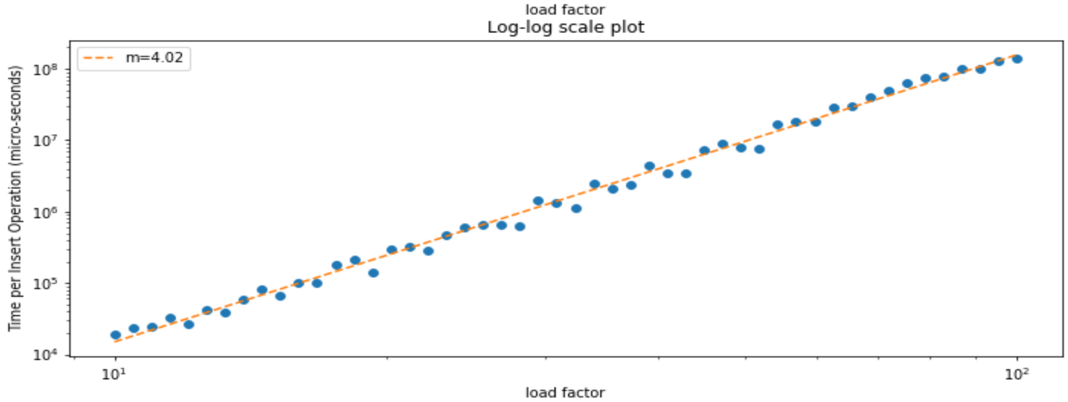
Close file handlers

4. **Testing:**

I did test on a lot of files generated by the randUnique.py file, to check how well did the hash map implementation scale. Unfortunately, for N = 10,000 the maximum inserts it is able to perform is approximately 11,400. And for files that have a mixture of insert, lookup and delete operations it scales for approximately 15000 operations.
For testing there are 2 files which I have attached and considered for analysis as shown below:

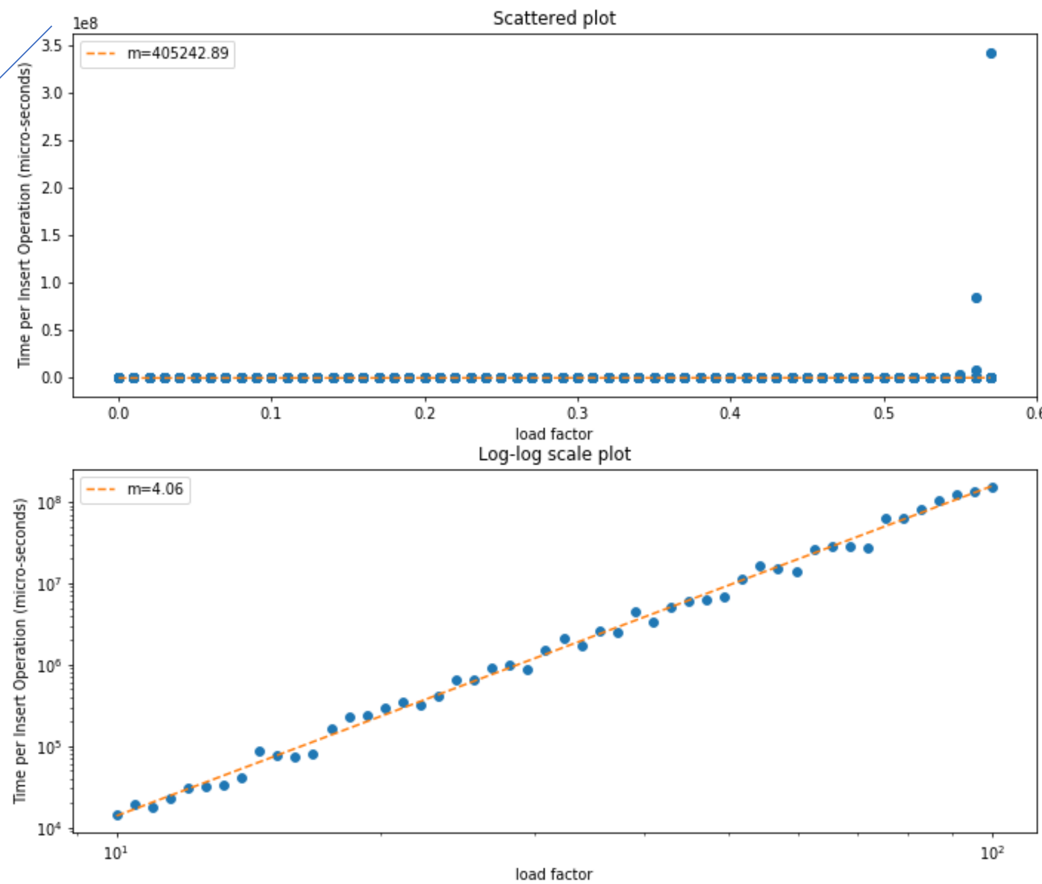
1) **Test_9000.txt:** This file contains **9000** insert commands for which the graph of time vs factor is as shown below:





2) **Test_11400.txt:** This file contains **11400** insert commands for which the graph of time vs load factor is as shown below:

Increase in worst case time by factor of 10^6 as compared to previous plot



5. Observations/Conclusions:

1) The hash function with higher degree like quadratic is much better than linear, as I had tried liner functions such as ($(a_0 + a_1x) \bmod p_1$) mod N which took more time to rehash than the quadratic function that I have implemented now.
2) The insertion time is directly proportional to load factor power 4, this is because a greater number of elements need to be shuffled while inserting a single element if the tables are relatively fuller. The spikes observed in figures 1,2 and 3 are due to the loops encountered while hashing in a new element due to which the whole table had to be rehashed.
3) Increasing or decreasing the primes p_1 and p_2 above a certain threshold can lead to more time to hash for a set of elements. Some primes just work more efficiently than others for a given set of randomly generated data.
4) For scaling the hash maps till 0.7-0.8 load factor more improved hash functions need to be seen. Third degree polynomials work worse than second degree polynomial for given set of test inputs and hence have not been used.

5) While trying to insert more than 11500 elements it takes about 20 mins to try out all the hash functions using the list of *primeNumbers* and exits. For about 15000 element insertions the terminal stops responding.

Linear Probing vs Cuckoo hashing:

Basis	**Linear Probing**	**Cuckoo Hashing**
Insertion of repeated values	Same keys can be inserted n times with worst-case time of O(n-1)	Same valued keys cannot be more than 2 in our case as by the definition these keys have a fixed location in both the hash tables and the 3rd key can never fit in either of these locations.
Total time taken for 9000 insertions	307.61 milli-seconds, is more than cuckoo hashing as resizing operation is performed 4 times and collisions are much more frequent.	158.3 milli-seconds, is less than linear probing because of N being 10,000 as well as stronger hash functions which distribute the keys evenly.
Slope of graphs and insights	The slope of 6.01 is observed for 9000 inserts on log-log graph. As for successive inserts the linear search keeps on increasing for each collision even if the size of the table increases.	Slope of 4.02 is observed for 9000 inserts on log-log graph. As the probability of collisions increase and loops become longer to detect but is relatively lesser than linear probing as liner search is much more expensive.
Comparison of lookup operation	The lookup operations can take O(n) time due to linear search when load factor approaches 1.	The lookup operations are very fast as only two locations are to be checked which is O(1)
Comparison of delete operation	The delete operations being similar to lookup searches for the element linearly and hence has O(n) time.	The delete operations being similar to lookup searches for the element in two locations and hence has O(1) time.
Average time of insert per load factor	The average time of insert operation is more for load factor 0.75 and 0.25 as that is when the forced resizing happens .	The average time per insert operation increases as load factor approaches 0.55 at which point almost all locations of table1 are filled and hence any new insertion will have a long sequence of displacing elements or a loop being detected in which case all the elements need to be rehashed.