PROGRAM IMPLEMENTATION:

TAS:

Shared Variables:

- C: Represents the current row being processed.
- lock: A boolean variable used to indicate whether a thread has acquired the lock.

Test-and-Set Algorithm:

-The test_and_set function is a custom implementation of the test-and-set algorithm. It atomically reads the value of `lock`, sets it to `true`, and returns the previous value of `lock`.

Thread Execution:

Each thread is responsible for processing a range of rows from the matrix.

The threads continuously loop to acquire the lock using `test_and_set`.

Once a thread acquires the lock, it reads the current value of `C` and calculates the start and end indices for processing rows.

The thread updates `C` to the next row to be processed and releases the lock by setting `lock` to `false`.

The thread then performs matrix multiplication for the assigned row range.

Initialization and Termination:

The `matrixinit` function reads the matrix size and other parameters from a file and initializes the input matrix (`matrixA`) with values.

The main function initializes the matrices, creates threads, and starts the matrix multiplication process.

After all threads have completed processing, the main function calculates the CPU time used and writes the result matrix to an output file.

Resource Deallocation:

Once the matrix multiplication is completed, the program deallocates the memory allocated for the input and result matrices.

CAS:

Compare-and-Swap Algorithm:

The compare_and_swap function is a custom implementation of the compare-and-swap algorithm. It atomically compares the value of `lock` with an expected value (`0`) and updates it to a new value (`1`) if the comparison succeeds. It returns the original value of `lock`.

Thread Execution:

Each thread is responsible for processing a range of rows from the matrix.

Threads continuously loop to acquire the lock using `compare_and_swap`.

Once a thread acquires the lock, it reads the current value of `C` and calculates the start and end indices for processing rows.

The thread updates `C` to the next row to be processed and releases the lock by setting `lock` back to `0`.

The thread then performs matrix multiplication for the assigned row range.

Rest of the program is completely the same.

Bounded CAS:

Bounded Compare-and-Swap Algorithm:

The main loop of each thread is modified to implement the Bounded Compare-and-Swap algorithm.

Each thread enters the critical section by attempting to acquire the lock using Bounded Compare-and-Swap.

The thread sets its `waiting` flag to `true` and then tries to acquire the lock by setting `lock` to `1` if the lock is currently unowned.

If the lock is already acquired by another thread, the thread sets its `waiting` flag and spins in a loop until the lock is released.

Once the thread successfully acquires the lock, it proceeds to execute the critical section, processing a range of rows from the matrix.

After processing, the thread releases the lock by setting `lock` to `0` and updates the `C` variable to indicate the next row to be processed.

The thread also checks if any other threads are waiting for the lock and signals them to proceed if their turn has arrived.
Rest of the program is the same.

Atomic:

Atomic Fetch and Add Operation:

Each thread uses the fetch_add operation on `C` to atomically increment and fetch the current value of `C`.

This operation ensures that each thread gets a unique range of rows to process without race conditions.

Thread Execution:

Each thread continuously loops to fetch a range of rows to process until all rows have been processed.

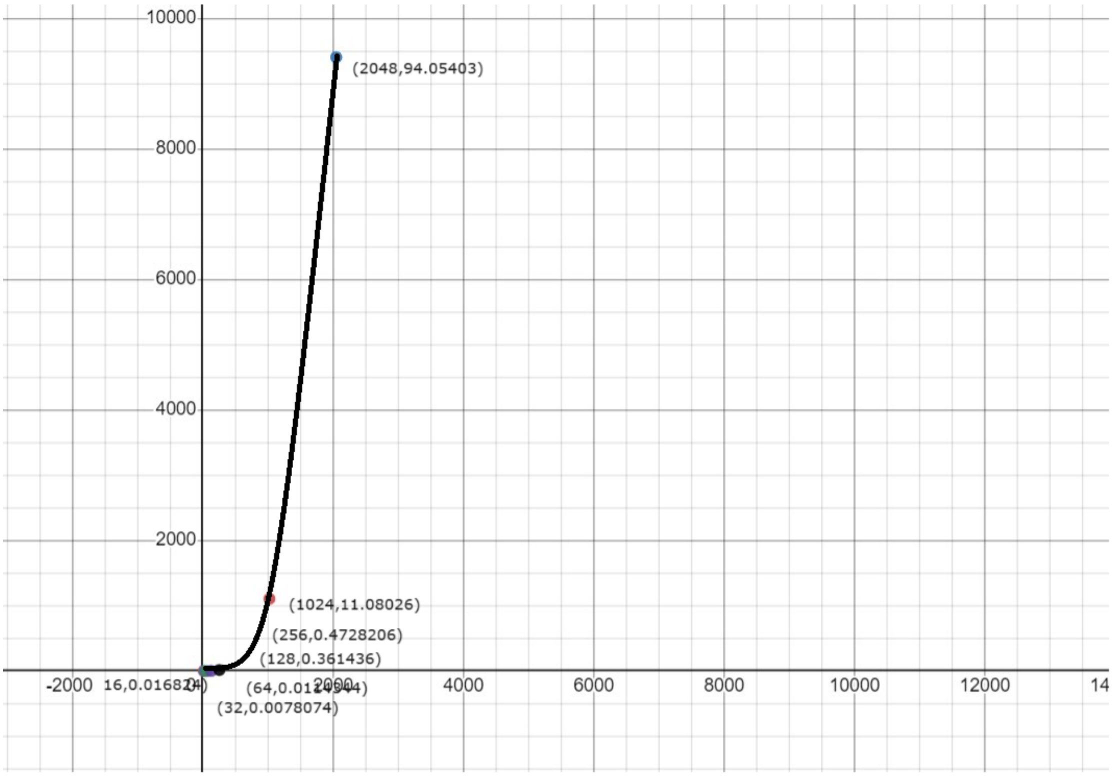
The thread atomically fetches the starting row index to process from `C`.

It calculates the ending row index as the minimum of the starting row plus the row increment and the total number of rows.

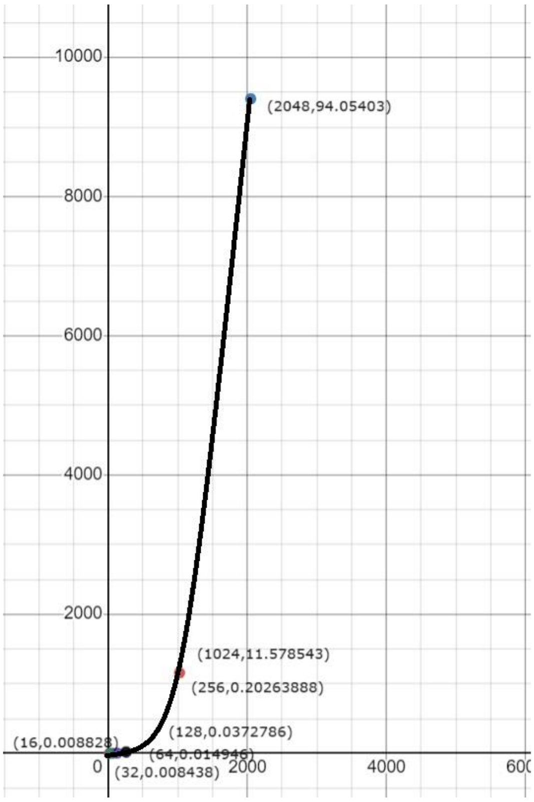
The thread then performs matrix multiplication for the assigned row range.
Rest of the program is completely the same.

Time vs. Size, N:

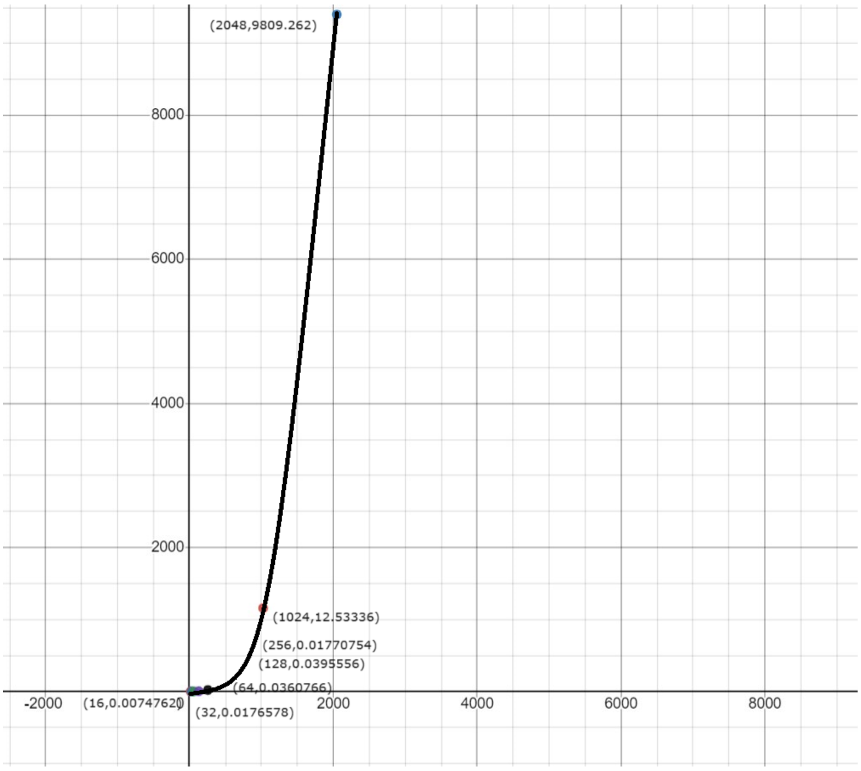
TAS:



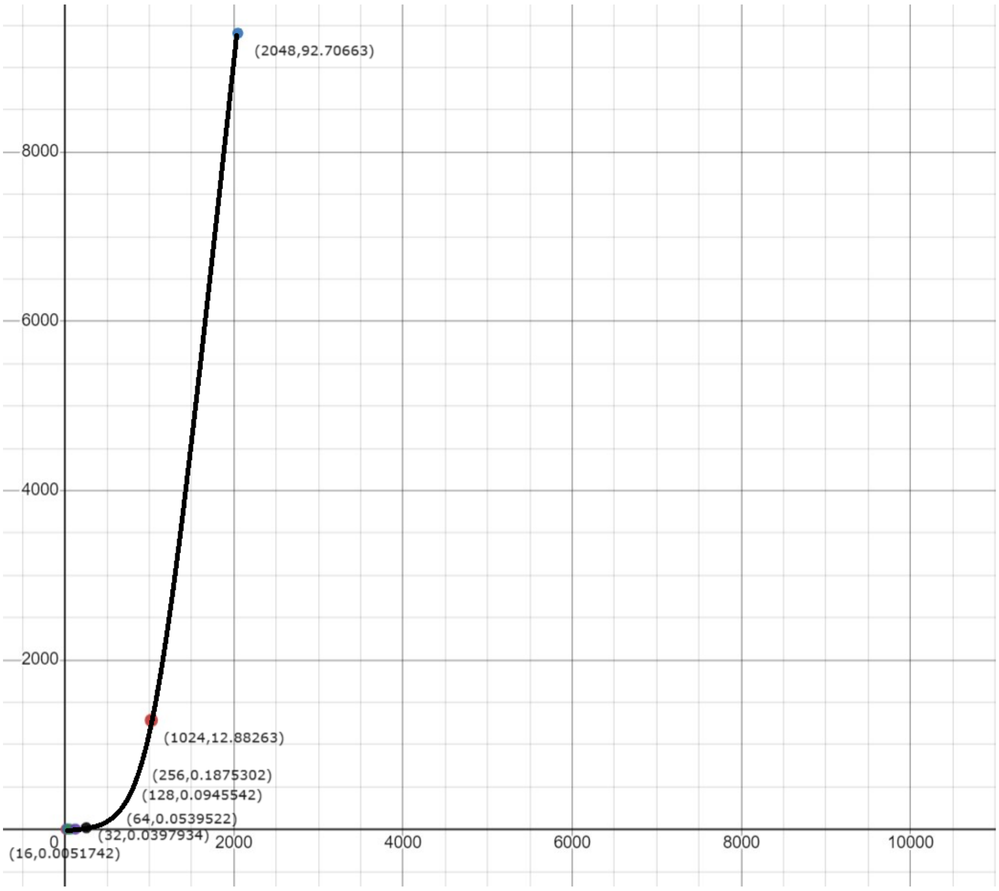
CAS:



Bounded CAS:

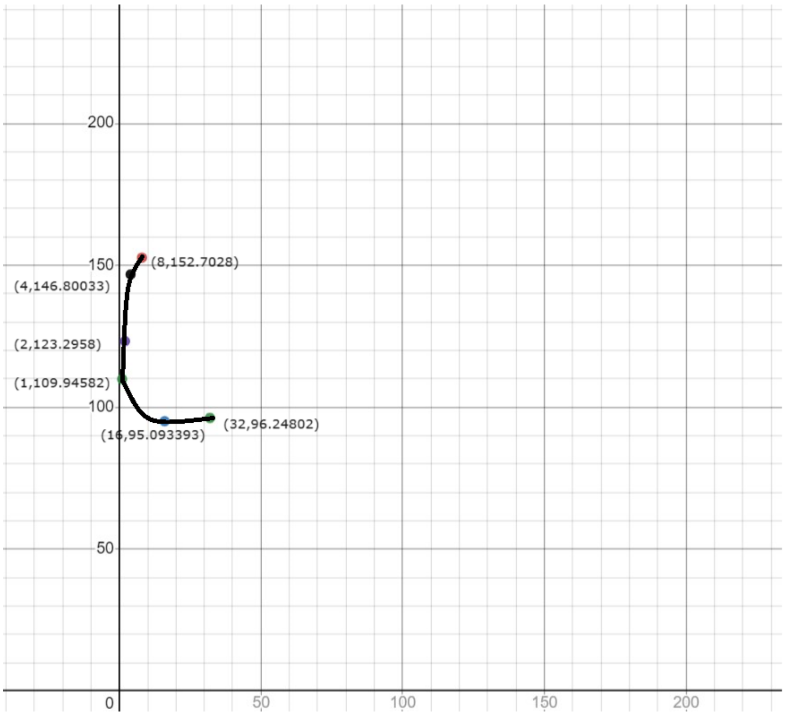


Atomic:

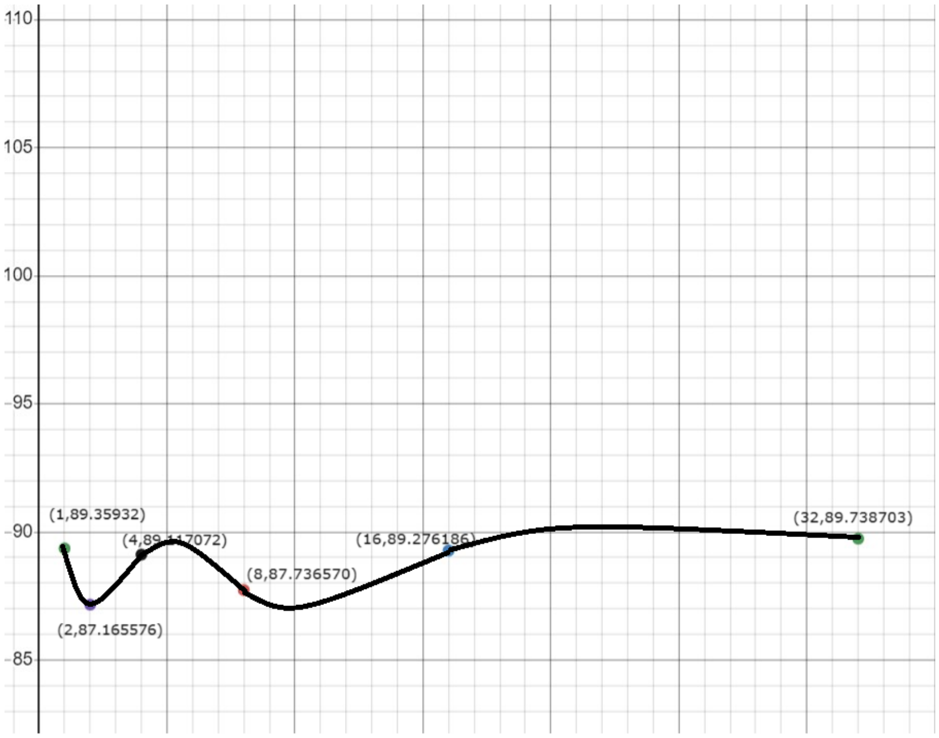


Time vs. rowInc, row Increment:

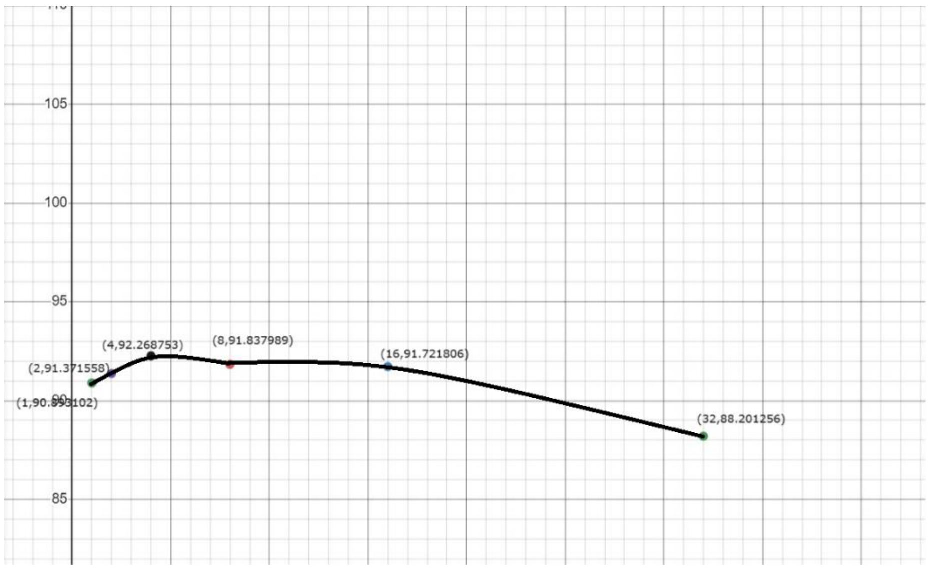
TAS:



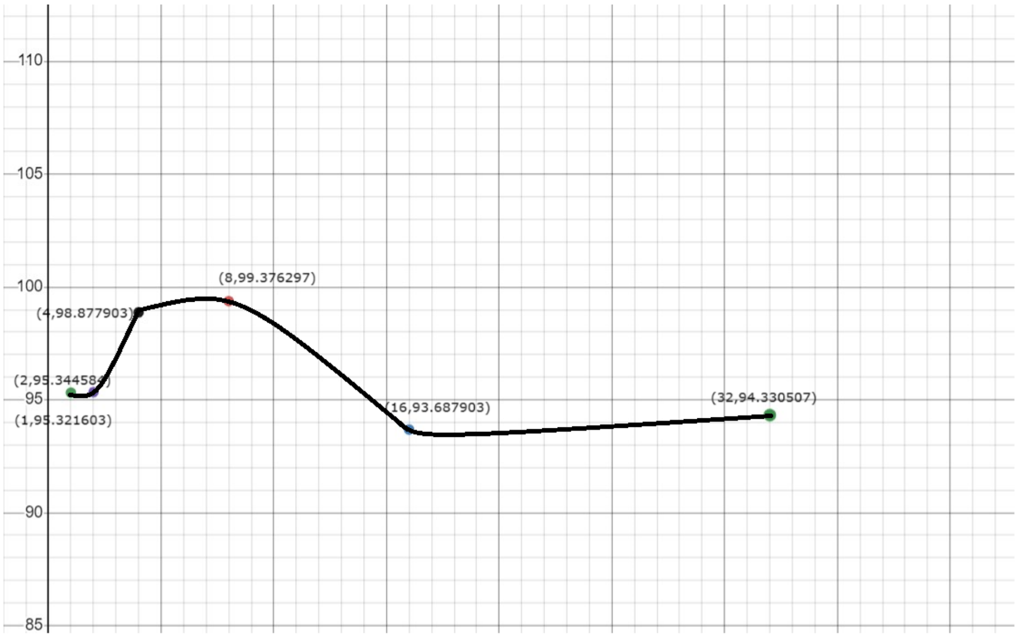
CAS:



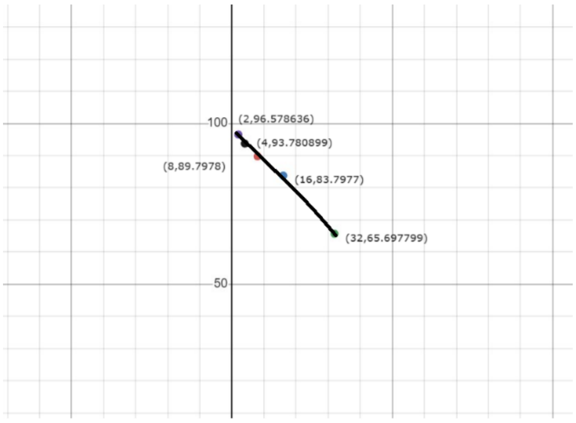
Bounded CAS:

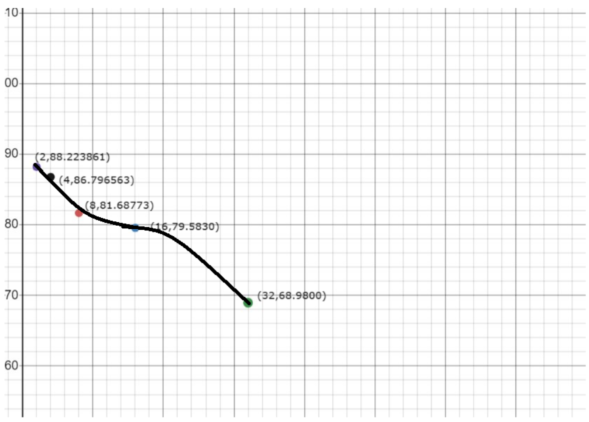


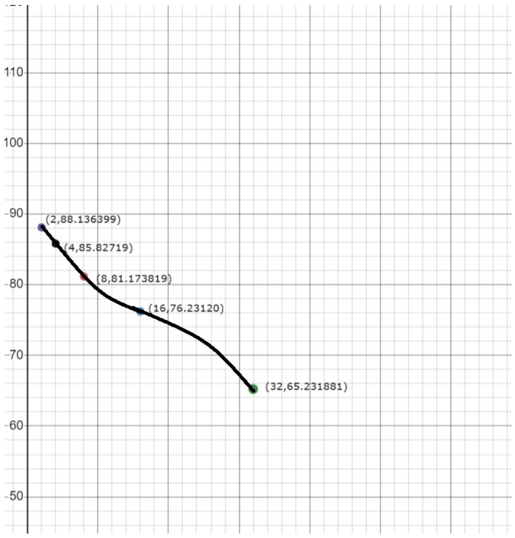
Atomic:

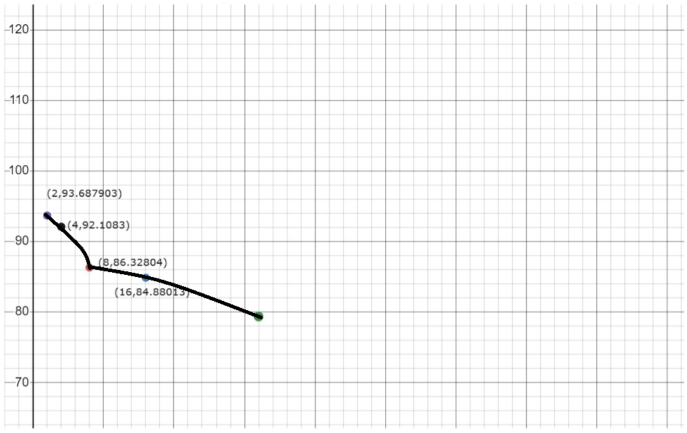


Time vs. Number of threads, K:

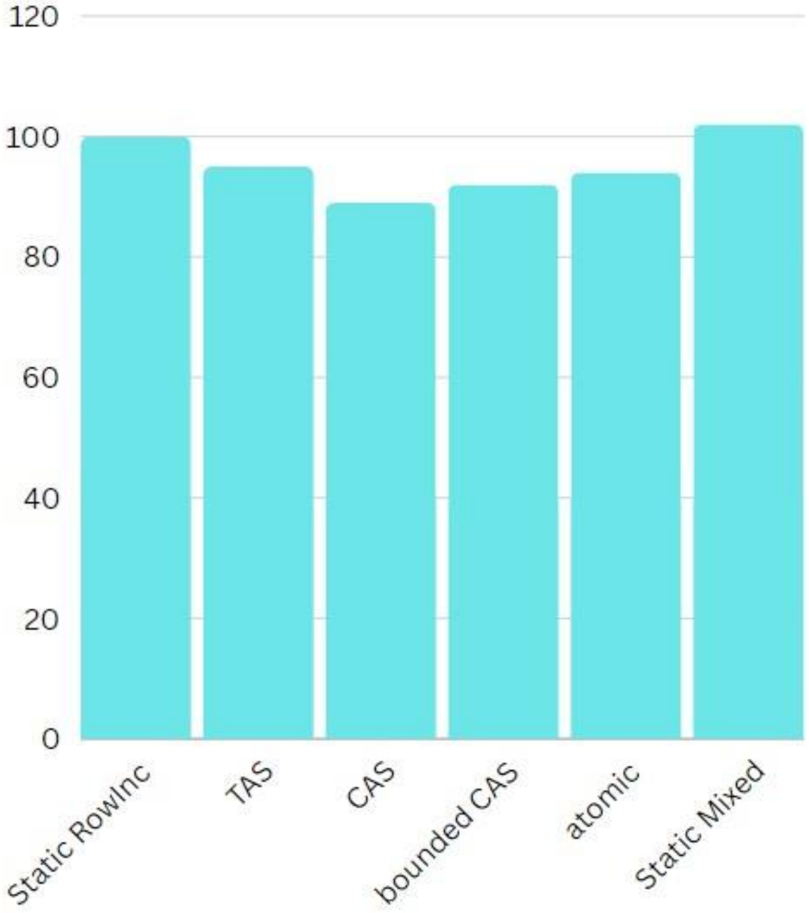








Time vs. Algorithms:
Bar Graph:



GRAPH ANALYSIS:

1.The time Vs N graphs for all Mutual exclusion algorithms is similar and almost have the same time although as observed most pointS in the graph in comparison show that time taken by TAS< CAS < Bounded CAS < Atomic.

2.In time Vs rowInc graphs in case of TAS the graph decreases then increases by a small value.

3. In time Vs rowInc graphs in case of CAS the graph has no staedy values it goes up and down but the values seem to be very near.

4. In time Vs rowInc graphs in case of Bounded CAS in the graph there is a little increment then decrement in the graph but the values of time are near.

5. In time Vs rowInc graphs in case of Atomic in the graph there is large increment ,then large increment then a little decrement as compared to the previous increments and decrements. Although the values are very near.

6 . In time Vs rowInc graphs between the algorithms the order seems to be TAS>CAS<Bounded CAS <Atomic approximately.

7.In time Vs K the time decrement graph seems to be similar.

8. In time Vs K the time taken by TAS>CAS>Bounded CAS <Atom is observed.