READERS-WRITERS(WRITER PRIORITY):

The program includes necessary header files such as `<iostream>`, `<fstream>`, `<thread>`, `<mutex>`, `<semaphore.h>`, `<chrono>`, `<unistd.h>`, and `<queue>`. - `NUM_READERS`, `NUM_WRITERS`, `NUM_READS_PER_READER`, `NUM_WRITES_PER_WRITER`: These variables store the parameters read from the input file, representing the number of reader threads, writer threads, number of reads per reader, and number of writes per writer respectively. Reads the parameters from the input file and stores them in the global variables.

Function getSysTime():

- Returns the current system time in HH:MM format.

Semaphore Declarations:

- `reader_mutex`, `writer_mutex`, `resource_mutex`: Semaphores for ensuring mutual exclusion among readers, writers, and accessing the resource respectively.

- `readers`, `writers_waiting`: Count variables for tracking the number of active readers and waiting writers.

- `writer_queue`: Queue to store waiting writer threads.

Functions `reader(int id)` and `writer(int id)`:

- These functions represent the behavior of reader and writer threads respectively.

- Each reader/writer performs multiple reads/writes specified by `NUM_READS_PER_READER`/`NUM_WRITES_PER_WRITER`.

- They request access to the critical section (CS), execute their task (reading/writing), and then release access to CS.

Main Function:

- Reads input parameters, initializes semaphores, and opens output files.

- Creates arrays of reader and writer threads.

- Spawns threads for readers and writers.

- Joins all threads after they complete execution.

- Destroys semaphores.

- Calculates and writes average times for reader and writer accesses to CS in the output file.

Thread Execution:

- Readers and writers synchronize access to the critical section using semaphores:

- Readers increment the `readers` count when they enter and decrement it when they exit. They also acquire and release the `writer_mutex` to prevent writers from accessing the critical section.

- Writers request and release `writer_mutex` to control access to the critical section. If there are active readers, a writer is enqueued in `writer_queue` until all currently readers finish.

Output Files:

- The program writes the sequence of requests, entries, and exits of readers and writers into `RW-log.txt`.

- It also calculates and writes average times for reader and writer accesses to the critical section in `Average_time.txt`.

READERS-WRITERS(FAIR SOLUTION):

The program includes necessary header files such as `<iostream>`, `<fstream>`, `<thread>`, `<mutex>`, `<semaphore.h>`, `<chrono>`, `<unistd.h>`, and `<queue>`. `NUM_READERS`, `NUM_WRITERS`, `NUM_READS_PER_READER`, `NUM_WRITES_PER_WRITER`: These variables store the parameters read from the input file, representing the number of reader threads, writer threads, number of reads per reader, and number of writes per writer respectively.

Semaphore Declarations:

- `reader_mutex`, `writer_mutex`, `resource_mutex`: Semaphores for ensuring mutual exclusion among readers, writers, and accessing the resource respectively.

- `readers`, `writers_waiting`: Count variables for tracking the number of active readers and waiting writers.

- `reader_queue`, `writer_queue`: Queues to store waiting reader and writer threads respectively.

Thread Execution:

- Readers and writers synchronize access to the critical section using semaphores:

- Readers increment the `readers` count when they enter and decrement it when they exit. They also acquire and release the `writer_mutex` to prevent writers from accessing the critical section.

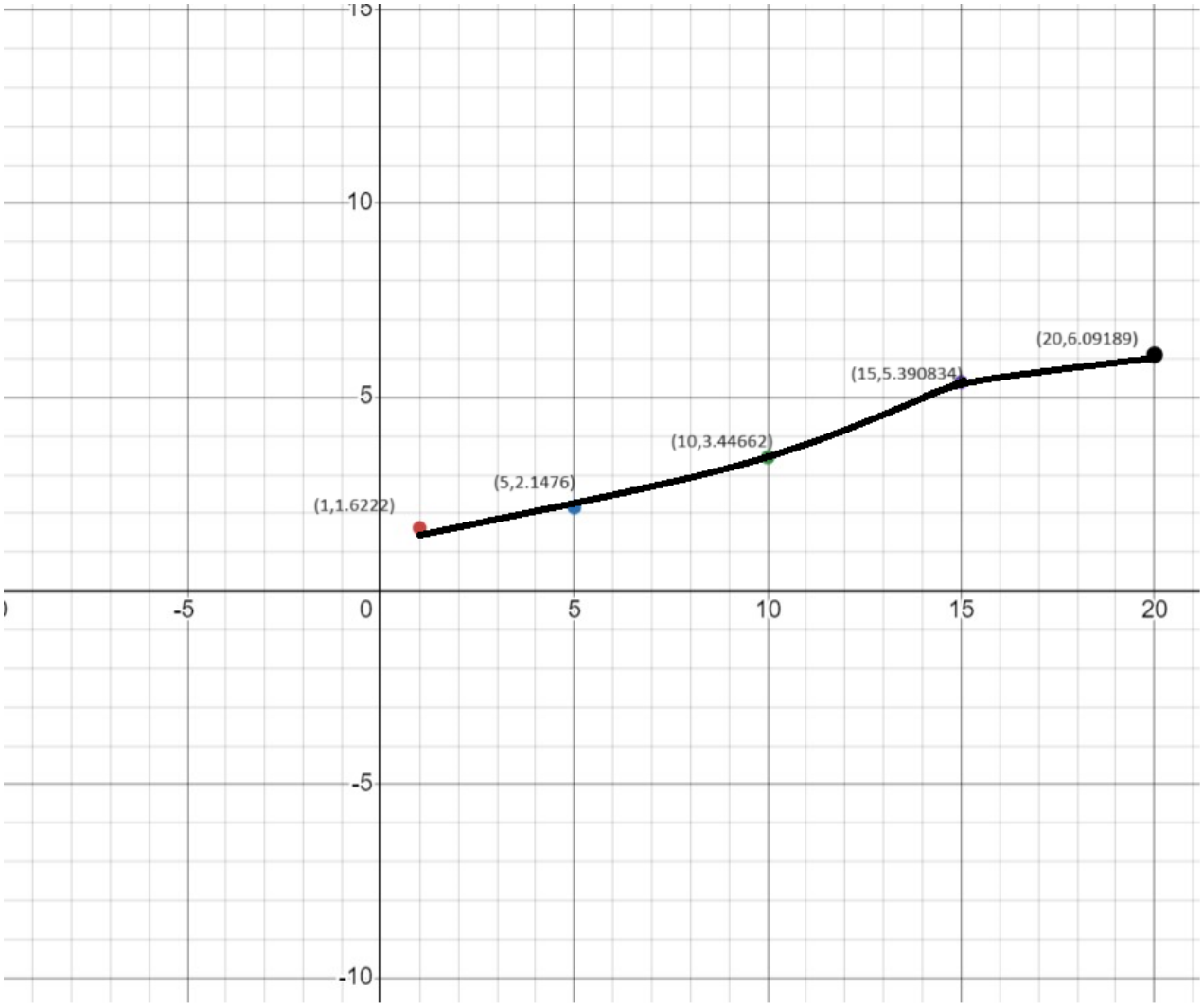
- Writers request and release `writer_mutex` to control access to the critical section. If there are active readers, a writer is enqueued in `writer_queue` until all readers finish.

Remaining code remains the same.

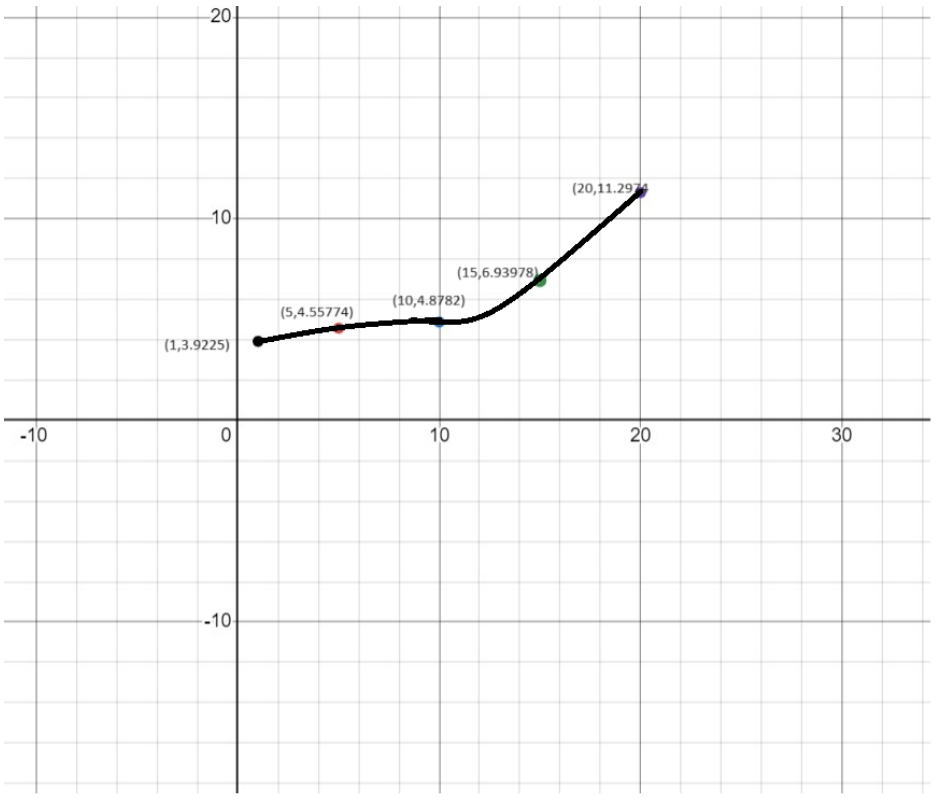
READERS – WRITERS(WRITER PRIORITY):

1)

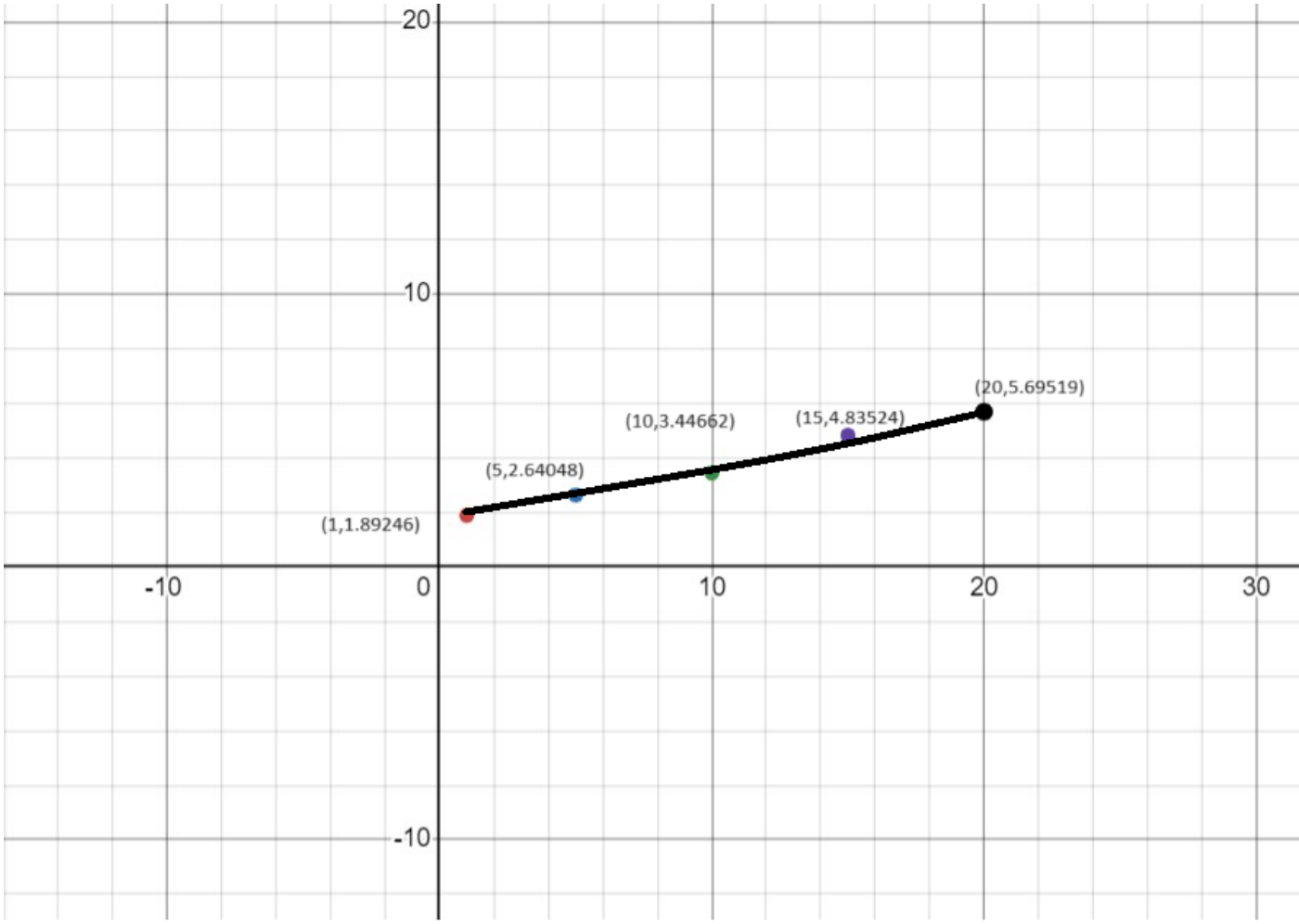
READERS:



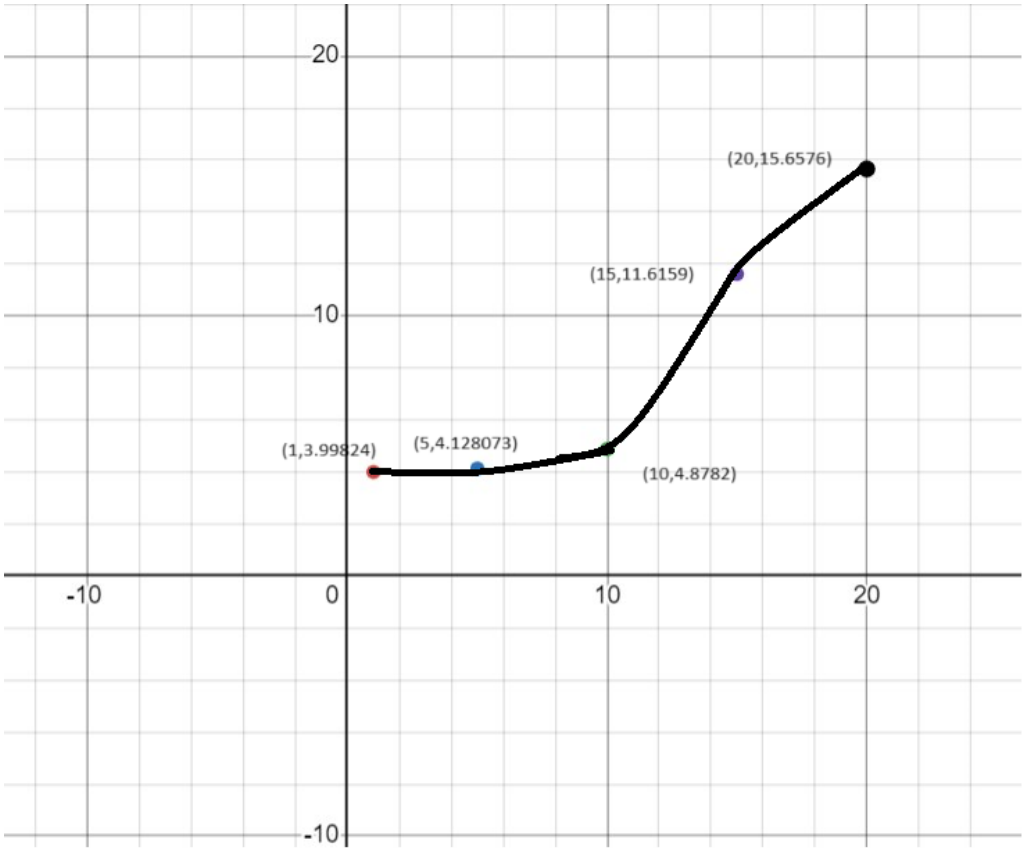
WRIETRS:


20
10
-10
-10
0
10
20
30
(1,3.9225)
(5,4.55774)
(10,4.8782)
(15,6.93978)
(20,11.2974)

2)READERS:

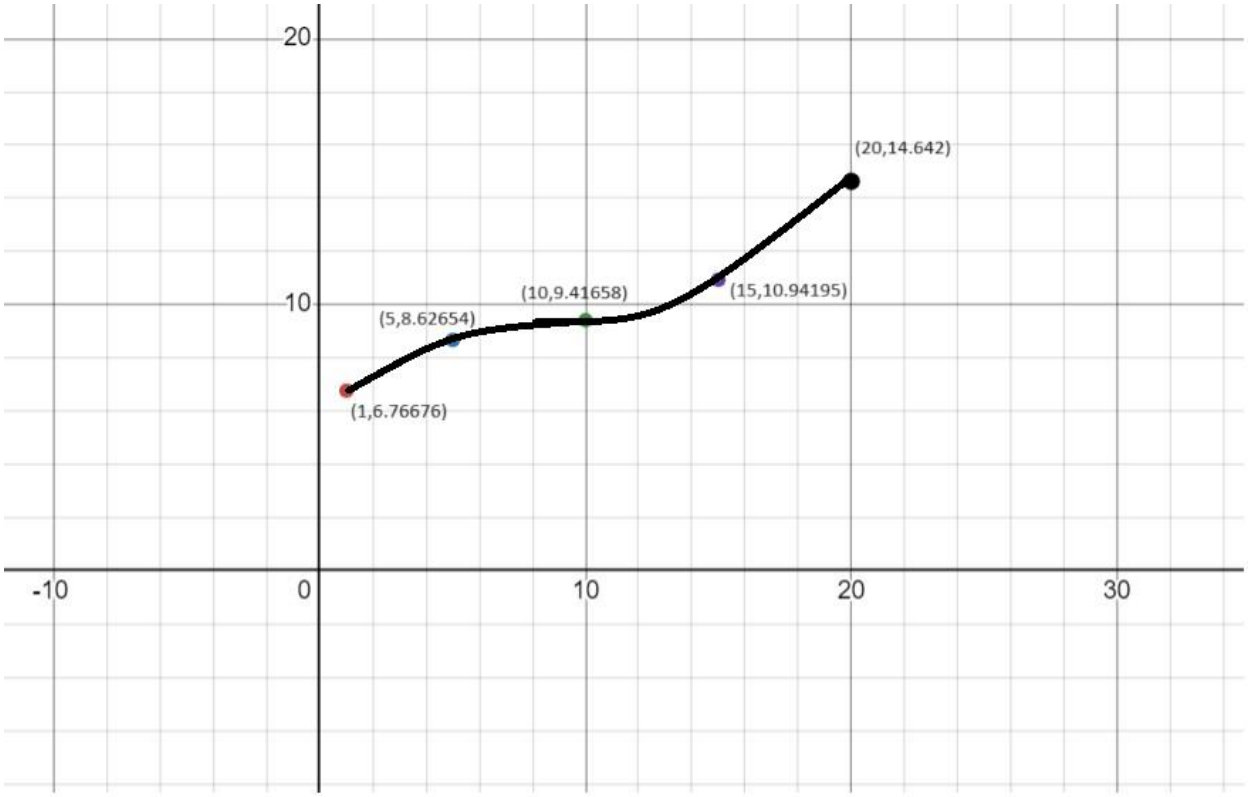


WRITERS:

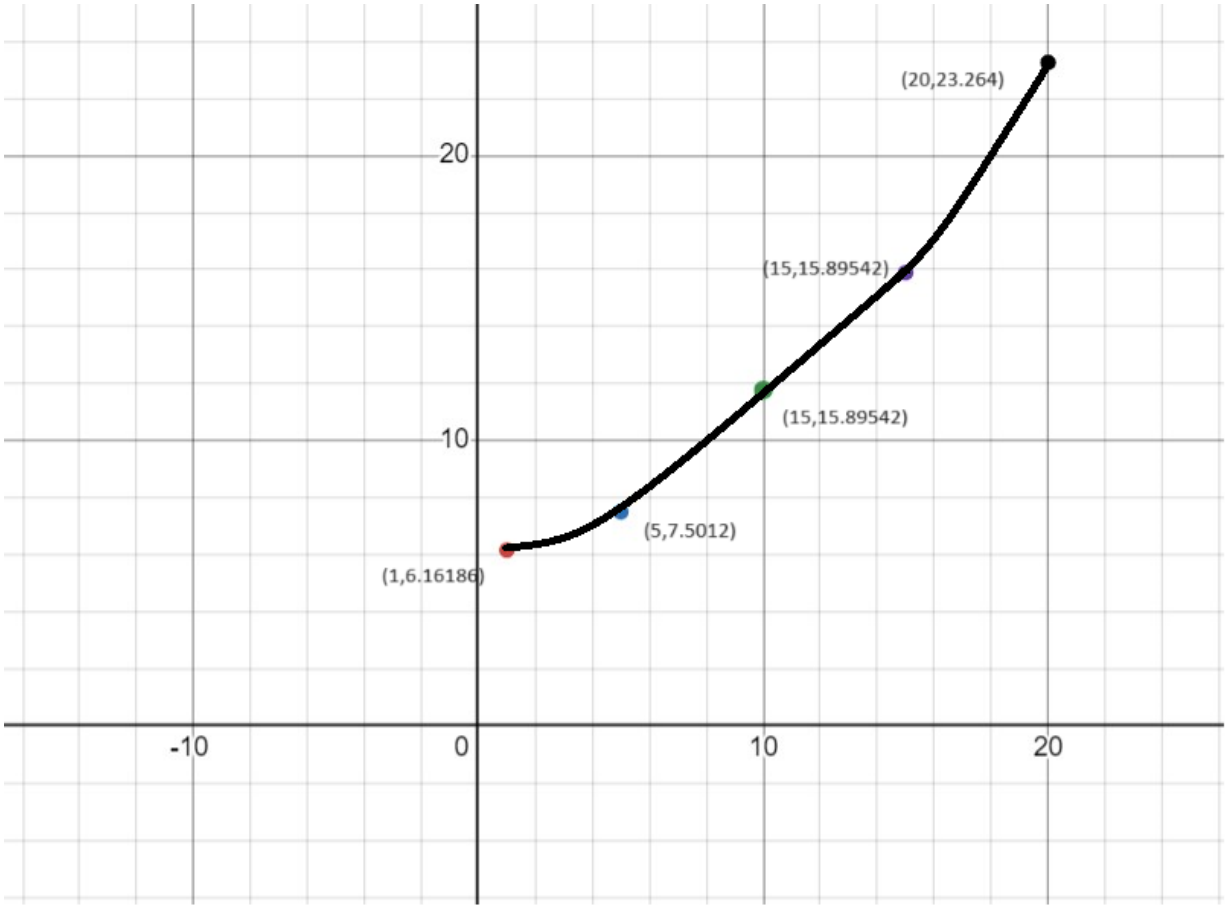

20
10
-10
-10
0
10
20
(20,15.6576)
(15,11.6159)
(1,3.99824)
(5,4.128073)
(10,4.8782)

3) WORST:

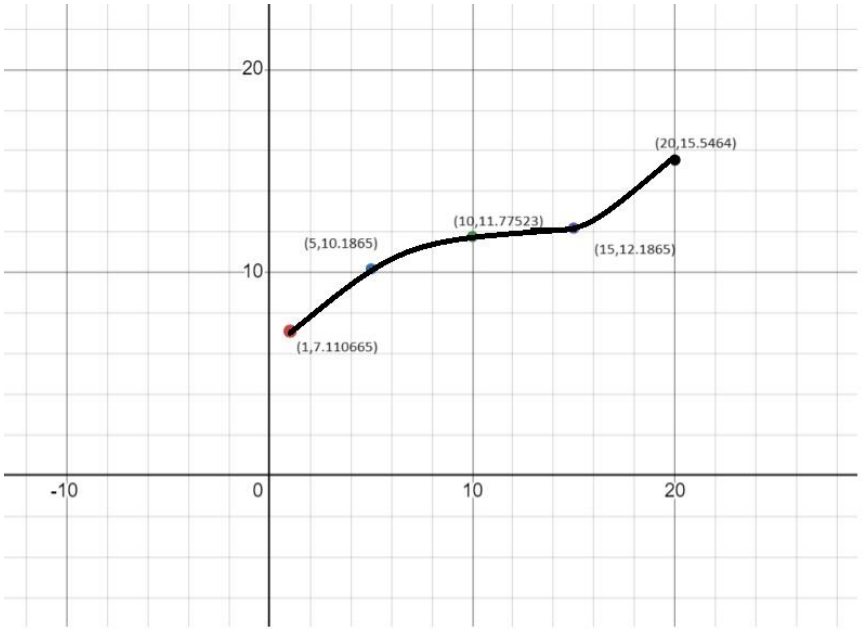
READERS:


20
10
-10
0
10
20
30
(20,14.642)
(10,9.41658)
(15,10.94195)
(5,8.62654)
(1,6.76676)

WRITERS:



4)READERS:


20
10
-10
0
10
20
(20,15.5464)
(10,11.77523)
(5,10.1865)
(15,12.1865)
(1,7.110665)

WRITERS:

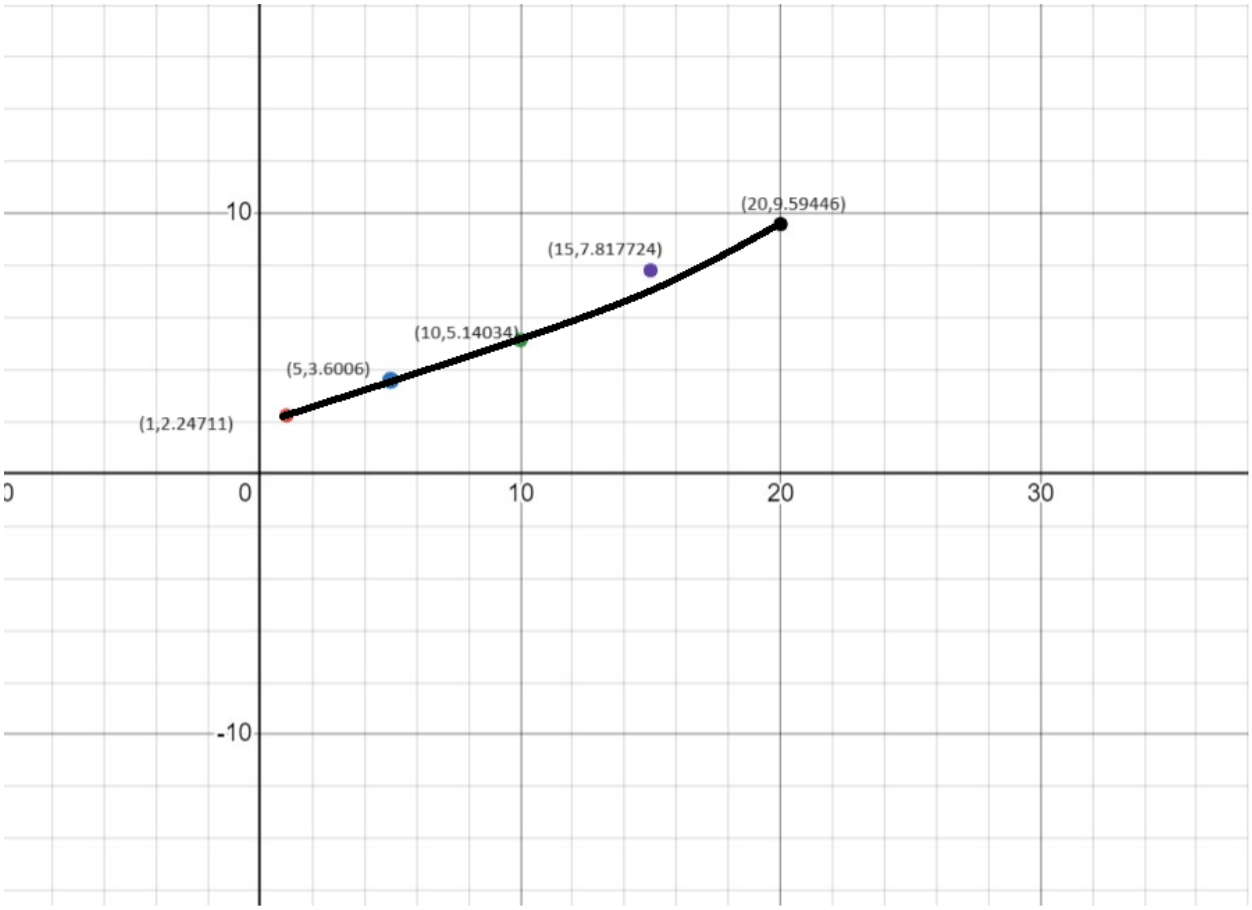
READERS – WRITERS(FAIR):

1)

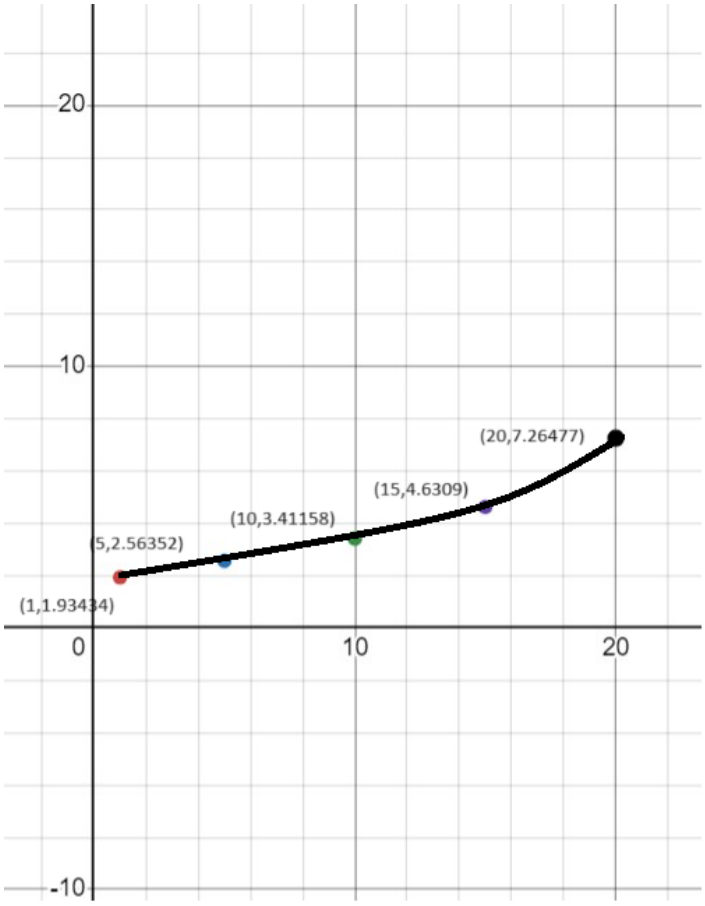
READERS:


10
-10
0
10
20
30
-10
(1,1.45295)
(5,2.6407)
(10,3.18922)
(15,4.50724)
(20,5.0369442)

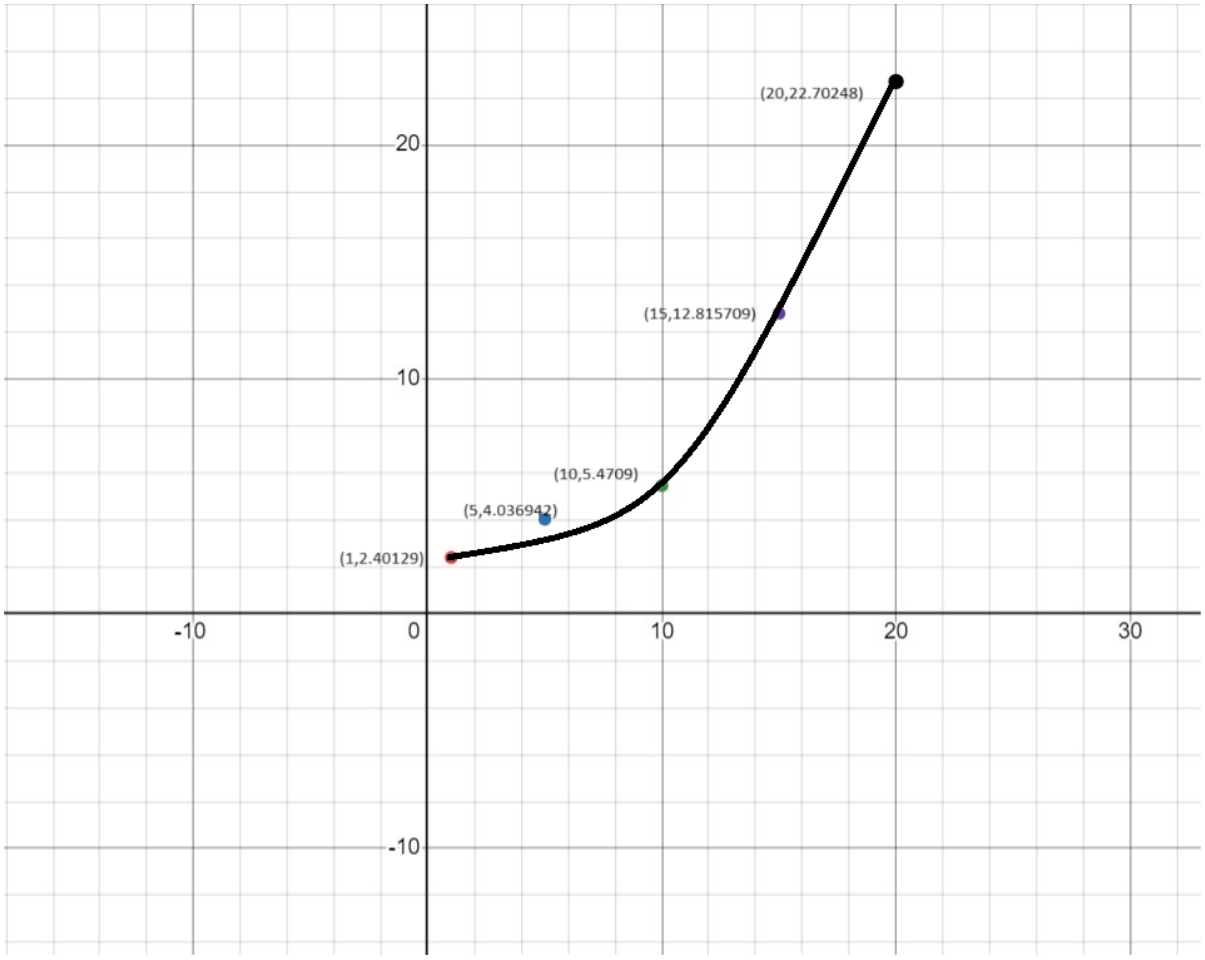
WRIERS:



2)READERS:

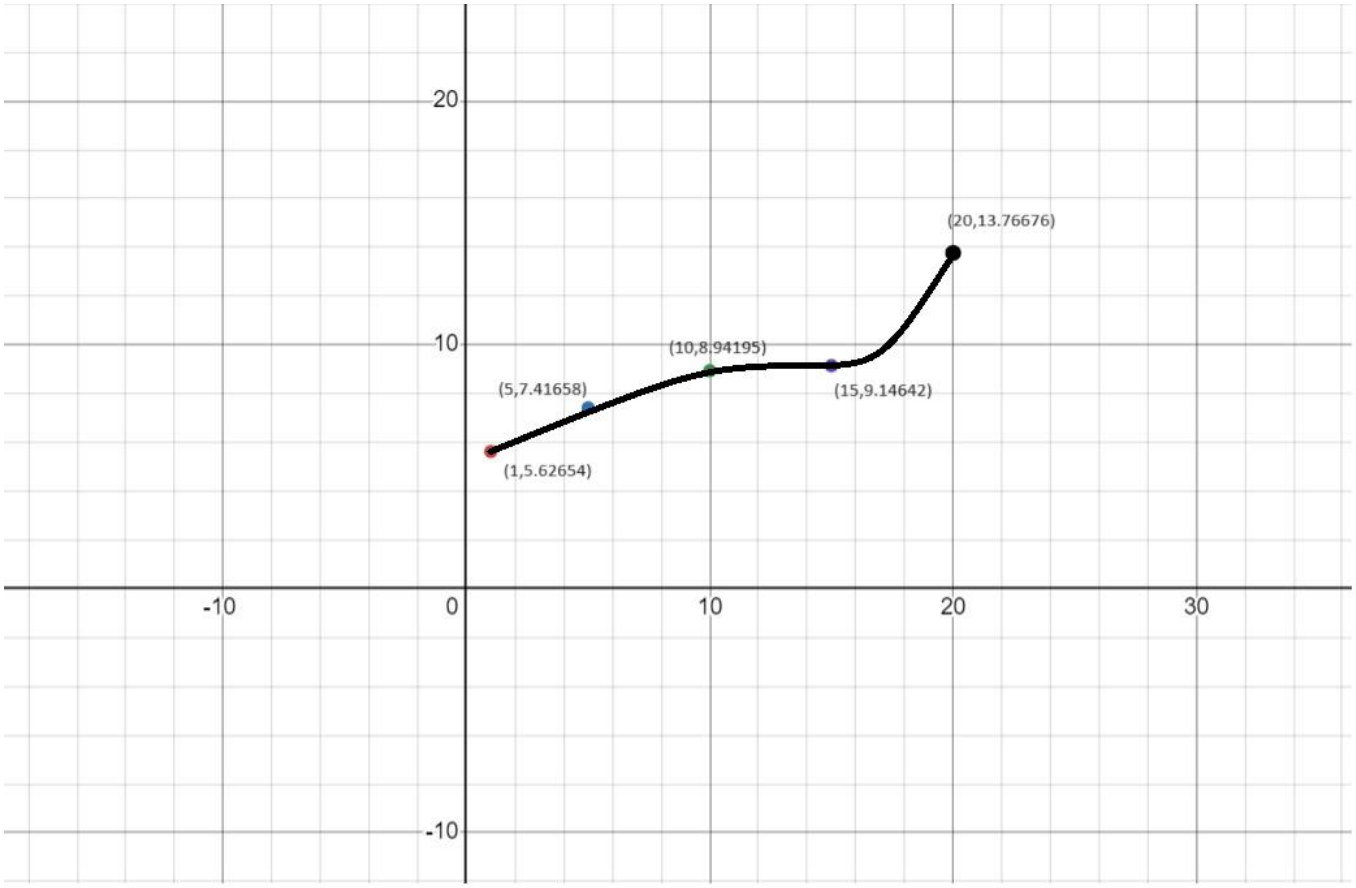


WRIETRS:

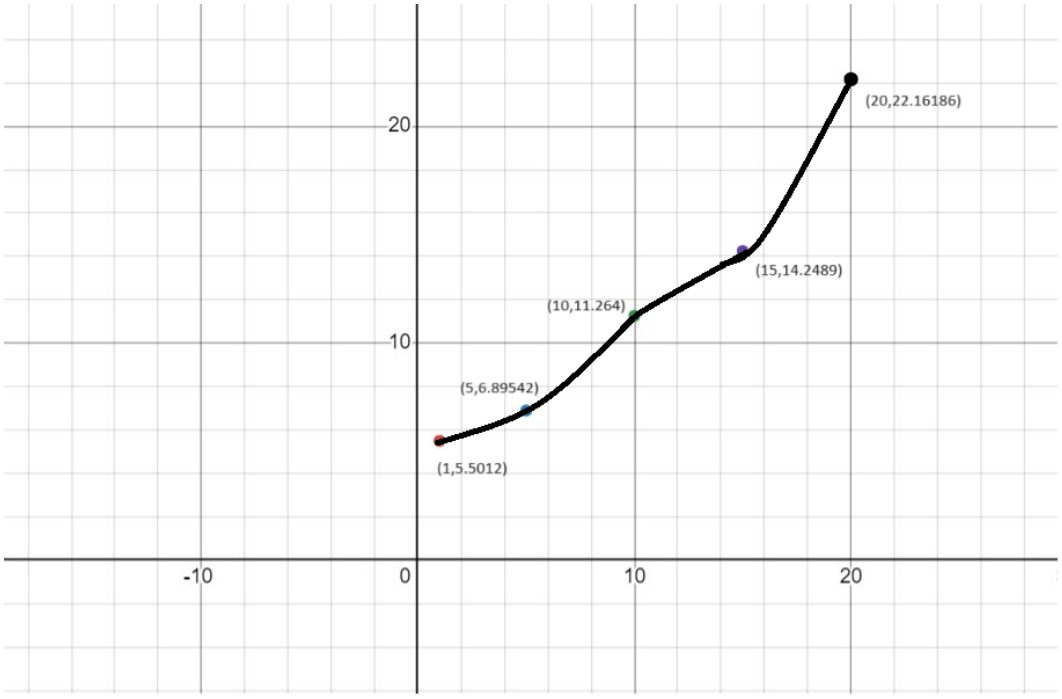


3)WORST:

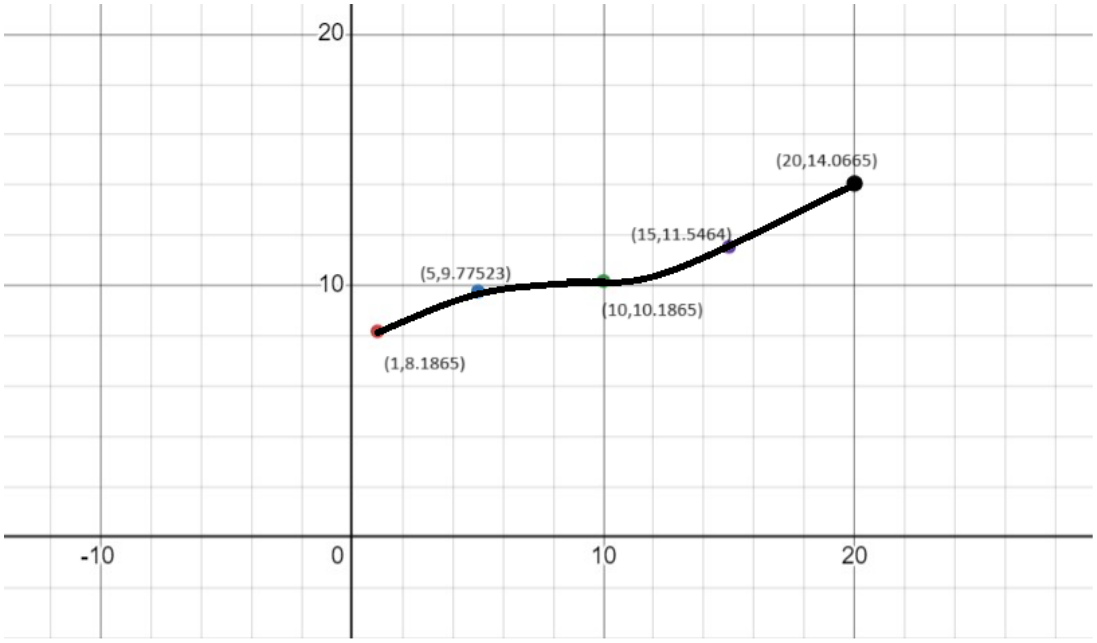
READERS:


20
10
-10
0
10
20
30
(20,13.76676)
(10,8.94195)
(5,7.41658)
(15,9.14642)
(1,5.62654)

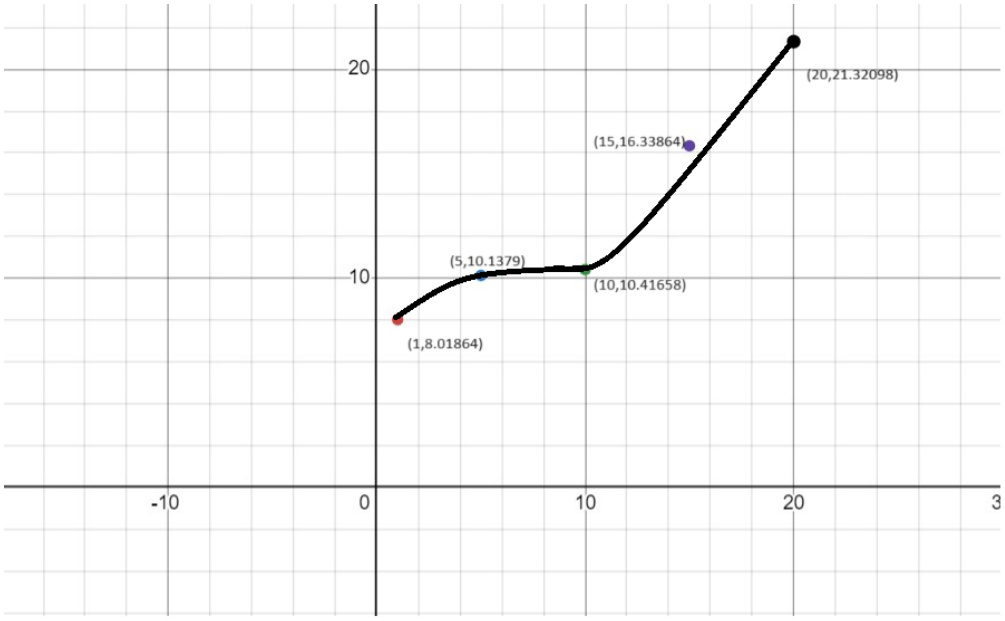
WRITERS:



4)READERS:



WRITERS:



ANALYSIS:

1)As per the graphs the time taken is always increasing.

2)The increment in time is greater in increment of writers than increment of readers irrespective of the algorithm or scenario(worst or average).

3)The fair solution takes less time for readers as compared to writer priority solution.

4) The fair solution takes more time for writers as compared to writer priority solution.

5)The graphs of average and worst cases seem to go in similar manner for both algorithms in case of reader and writer increment both.