CSN-232

PROJECT-2 REPORT

Team Members:

Aman Jaiswal	17114008
Aman Verma	17114009
Amit Vishwakarma	17114010
Anshuman Shakya	17114013
Abhishek Rathod	17114004
Deepika Choudhary	17114025
Theegala Lokesh Reddy	17114073
Karan lamba	17114043

LRU using Stacks :

Algorithm :

The pages are maintained in a stack and when a page is referenced, we search that page in the stack.If the page is found in the stack ,we remove the page from its original position and put it on the top of the stack.If it is not found, then we remove the bottom element of the stack and push the page which is referenced into the stack.In this way the most recently used pages are always near the top of the stack and least recently used are perhaps at the bottom.This was the algorithm behind implementing the least recently used replacement policy.

Implementation and Testing :

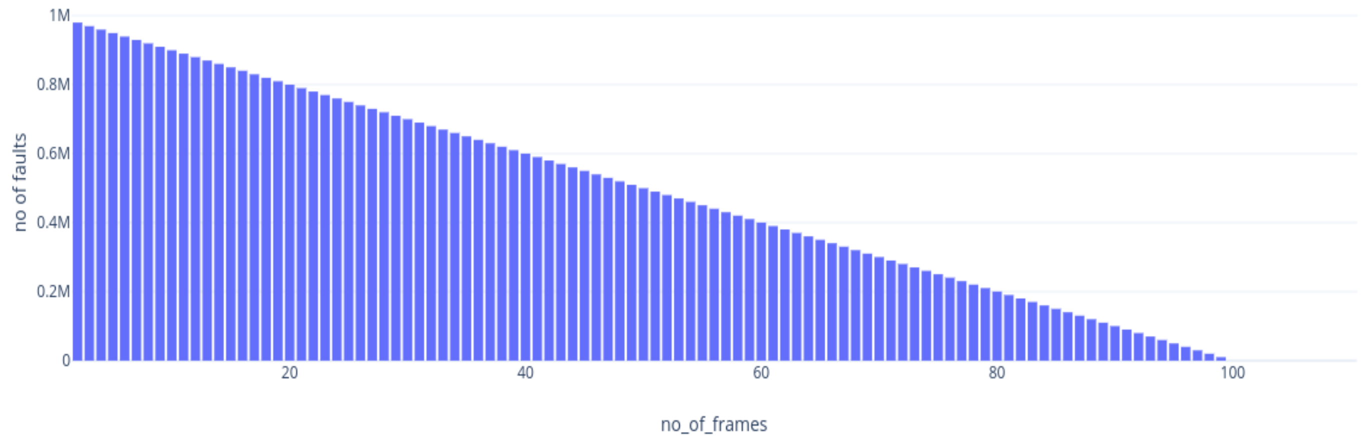
We are maintaining a stack s and coded down the arbitrary methods except the pop() function which is modified and feeds arguments like the element to be pushed and the stack size.The variable index is maintained each time we search the stack for the referenced page which signifies the index of the page referenced in the stack.Incase the page is not found ,it is -1.Ultimately it will update the stack according to the algorithm mentioned above.

A delay method is implemented each time a page fault is encountered as memory is referenced to get the required page.The fault variable gives the corresponding number of page faults as the output.

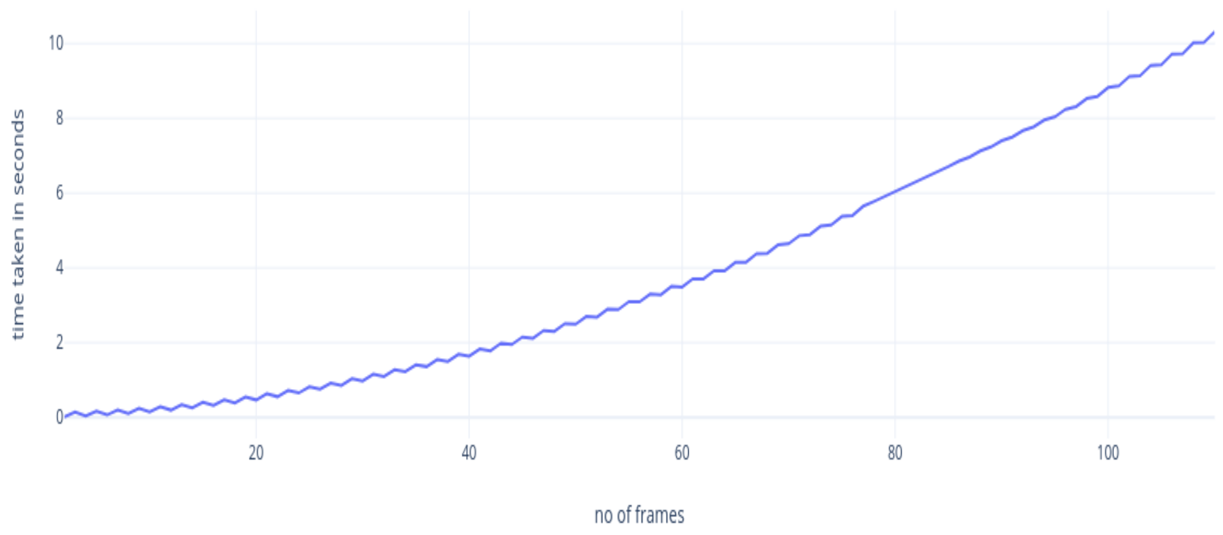
We have tested the algorithm with two test cases-

Test Case 1-

In this test case there are 10^6 inputs in the frame sequence with the maximum value of page number as 100, the size of page table varying from 2 to 100.



Above graph represent number of page faults vs no. of frames in the page table from 2 to 100 which is incremented by 1. When page table size is greater than 100 the no. of page faults is constant.It is decreasing as the frames increase there is more chance that the referenced page is already present in the stack.

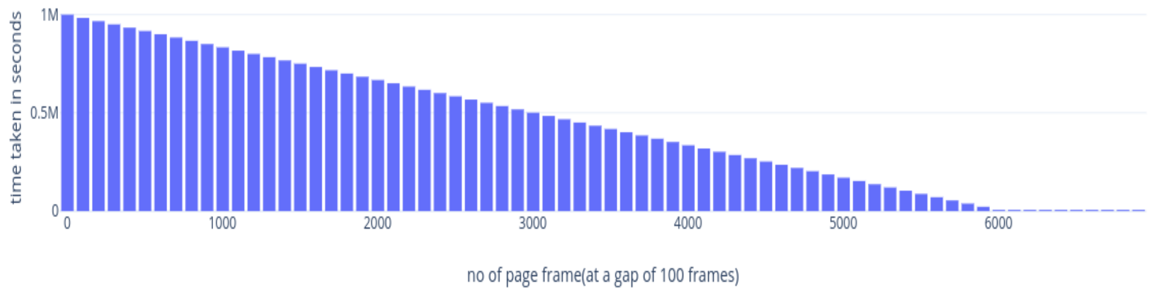


Above graph represent time taken in the execution of algorithm vs no. of frames in the page table.Time taken will definitely increase with the number of frames as anyway we are searching for the page in the stack and whatever be the result , the stack has to get modified.

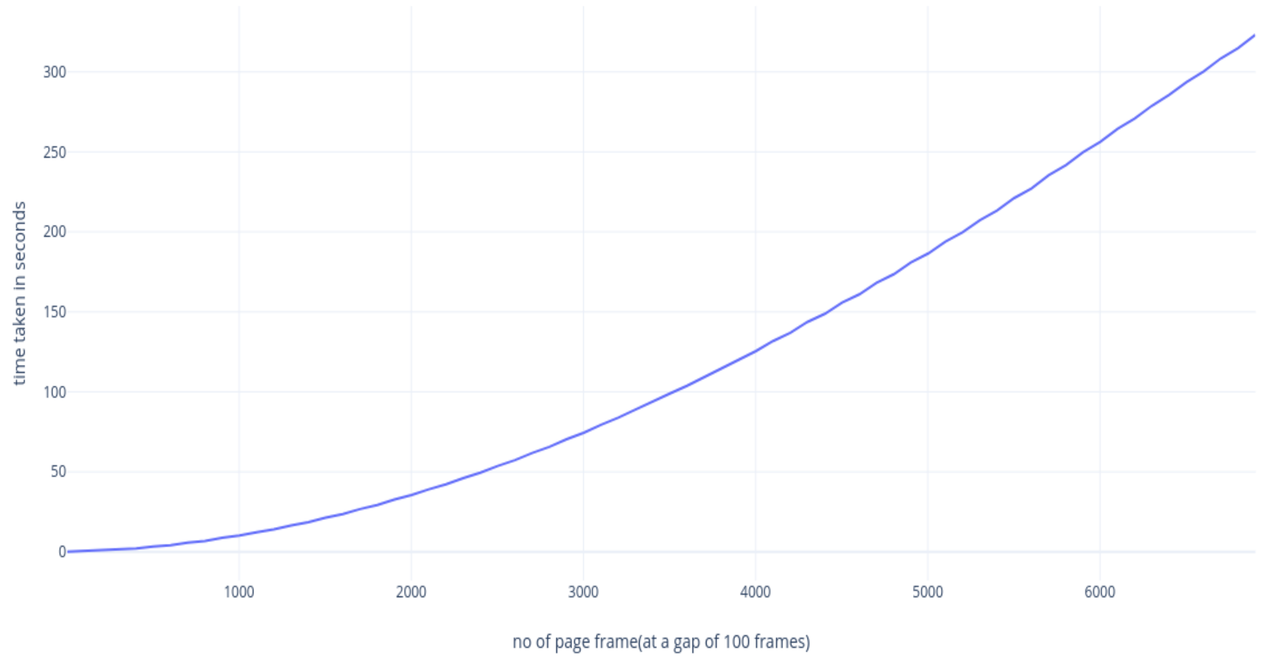
Test Case 2-

In this test case there are 10^6 inputs in the frame sequence with the maximum value of page number as 6000, the size of page table varying from 2 to 7500.

The reasons for nature of this test case are just the same as mentioned in the first test case.



Above graph represent number of page faults vs no. of frames in the page table from 2 to 7500 which is incremented by 100. When page table size is greater than 6000 the no. of page faults is constant.



Above graph represent time taken in the execution of algorithm vs no. of frames in the page table.

Time Complexity Analysis :

If T(n) is time complexity of our LRU_clock(),

f – length of input frame sequence.
n – number of pages in the list.
h – number of page hits.
p – page fault time.
c – some constant time.
Therefore, considering the worst-case analysis,

$$T(n) = f + h*n + (f-h)*(p+n) + c$$
$$= O(f*n + (f-h)*p)$$

Either page hit or miss, we have to search in the stack which requires linear time and if page miss is there, delay time (p) is added.

LRU - Counter Method

Algorithm Explanation-

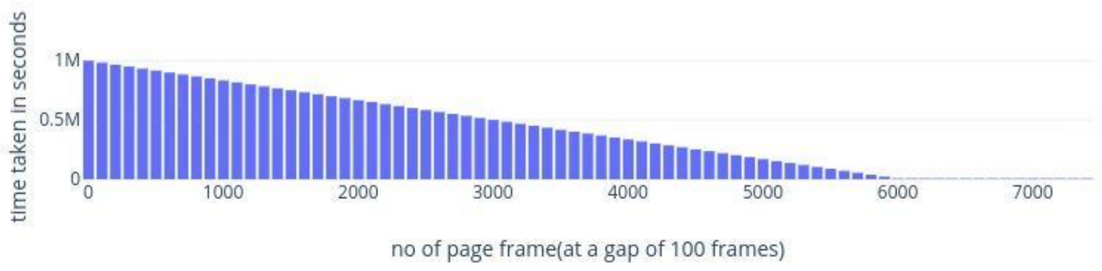
- Every page entry has a counter.
- Every time a page is referenced, increment a global counter and store it in the page counter.
- For page replacement, select the page with the lowest counter (search for it).

Code Explanation-

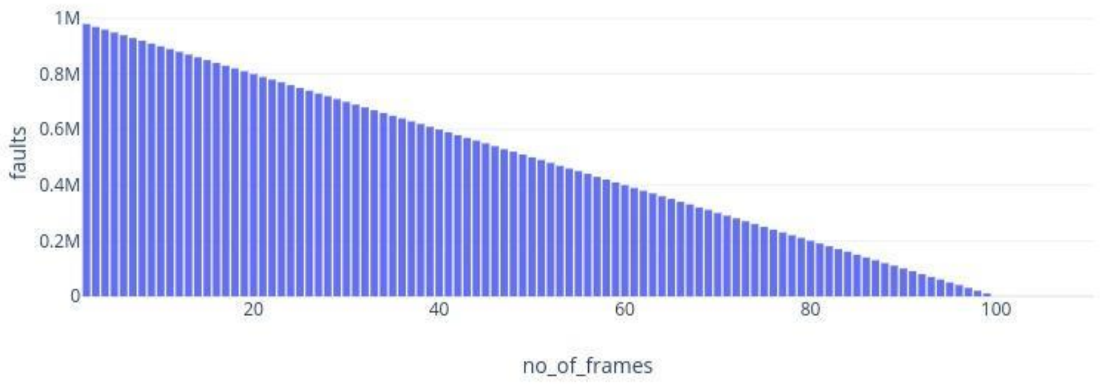
We have created two structs: page and cache which specify the pages to be inserted and the reference. The page contains the attributes like time and value to be stored. the cache struct contains a number of pages, no of faults, reference,time_stamp, start time. We have a clock which is used for time reference and time stamp. After the algorithm is implemented. The output print a different number of page frames and their corresponding page faults and their execution time.

Observation-

Number of page faults vs Number of frames(input 1)

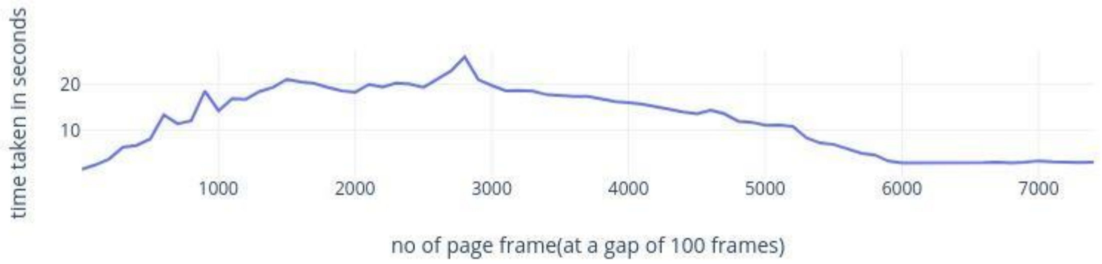


Number of page faults vs Number of frames(input 2)



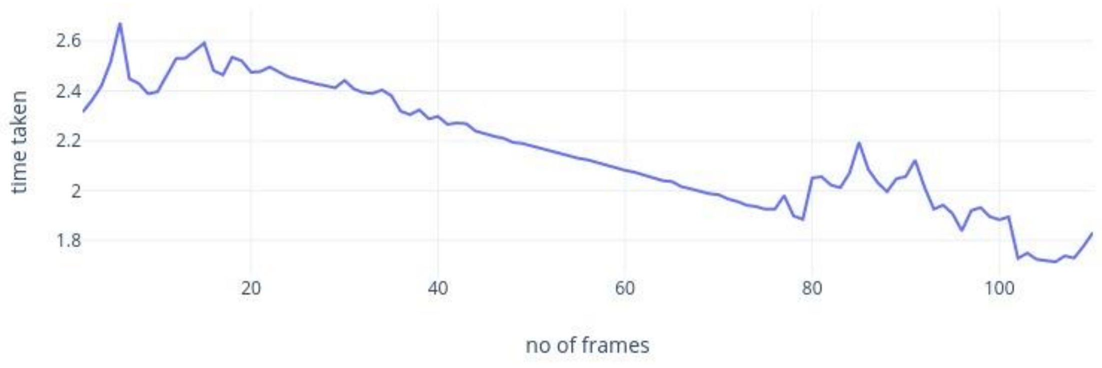
Time is taken in second vs Number of frames(input 1)

Starting it is increasing because search time of frames is increasing later page fault will be less and it will dominate.

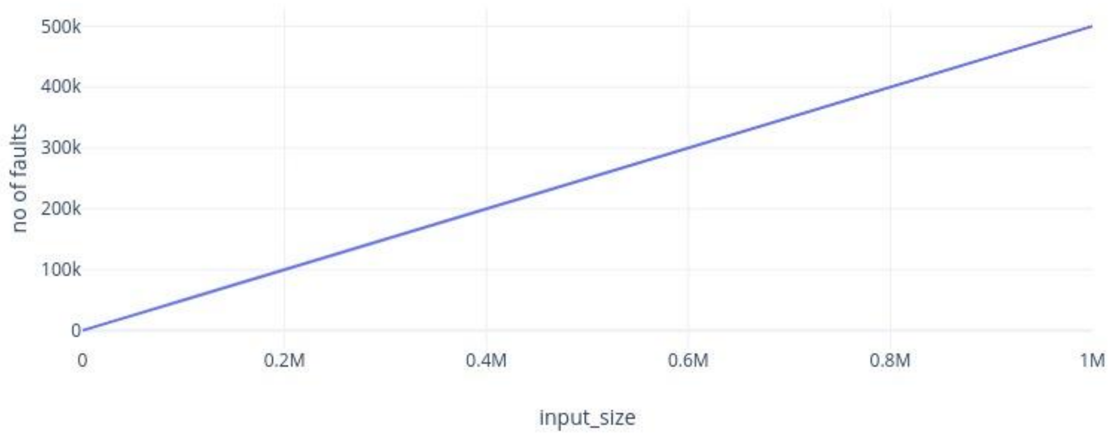


Time is taken in second vs Number of frames(input 2)

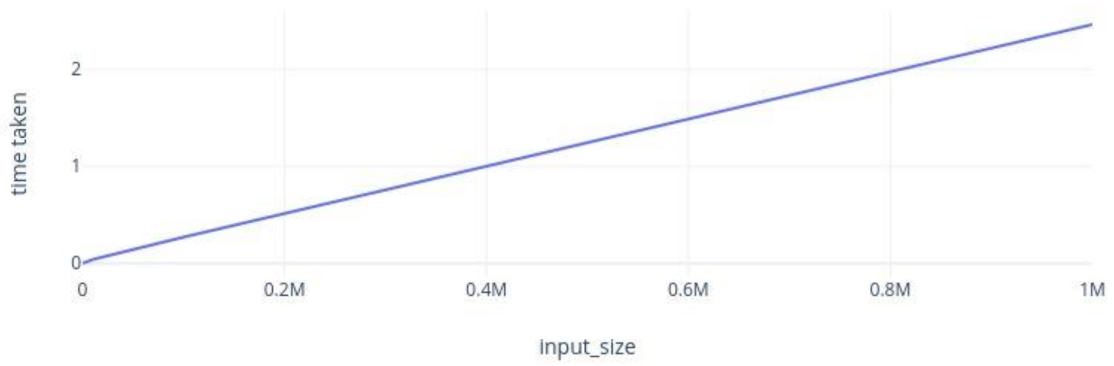
we know that the number of frames are increasing time taken will decrease.



Number of faults vs Input size



Time is taken in second vs input size



Time Complexity-

(number Input_Size)*O(1)+Log(frame_size)*(Number of Page fault)

Log(frame_size)-we are using map
O(1)-unordered map

LRU - Aging Register Method

Aging register method uses additional ordering information by recording the reference bits at regular intervals. It maintains an n-bit (generally 8-bit) aging register (reference counter) for each page frame. The reference counter on a page is shifted right (divided by 2) after every clock tick and if the page is referenced in the current clock tick the referenced bit at the left of that binary number is set. For instance, if a page has referenced bits 1,0,0,1,1,0 in the past 6 clock ticks, its referenced counter will look like this: 10000000, 01000000, 00100000, 10010000, 11001000, 01100100. Page references closer to the present time have more impact than page references long ago. This ensures that pages referenced more recently, though less frequently referenced, will have higher priority over pages more frequently referenced in the past. Thus, when a page needs to be swapped out, the page with the lowest counter will be chosen.

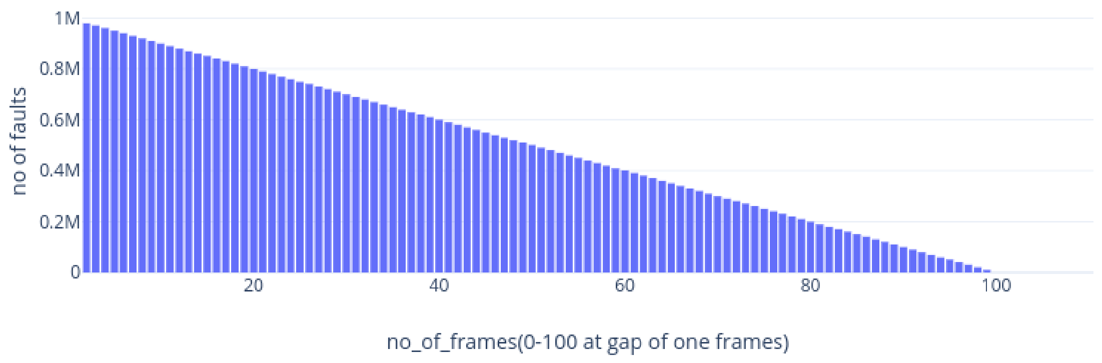
Implementation

In the implementation, we have used two structures.
One is a named *page* which contains variables-

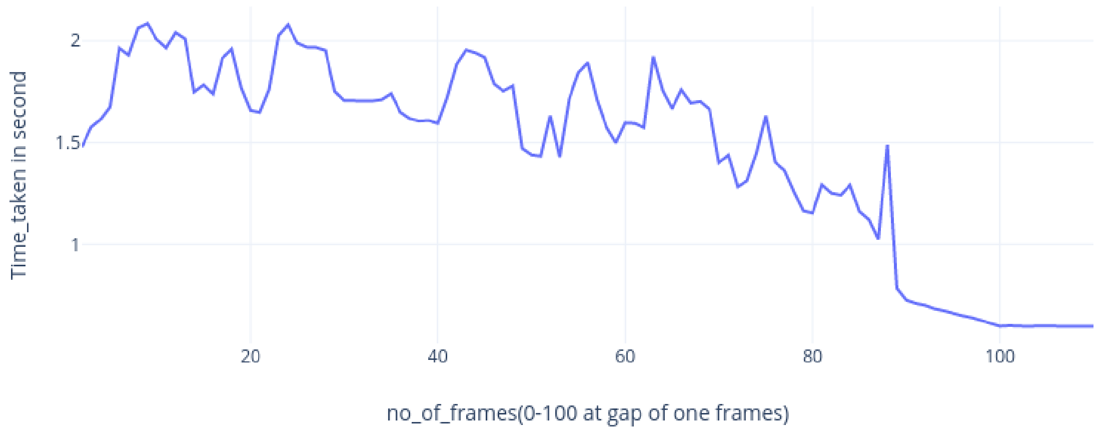
val- page number in the page table.
reg_counter- the value of reference counter for that page.
The second struct is named *cache* which represents the page table which contains variables-
no_of_pages- No. of pages in the page table.
vector<page> v- vector of the pages in the page table.
isfull- a boolean true if the page table is full.
no_of_faults- No. of the page of faults.
refresh_time- the time interval between every clock tick.
Afterwards, there is a function *run()* which will take a parameter as the no. of page frames in the page table. Then for each frame in the frame sequence the function is checking whether it is present in the page table or not, if it is already present it changes the leftmost bit of reg_counter to 1 otherwise if it is not present in the page table and the page table is not full we create another page in the page table but if it is full there is a page fault so we replace the page with least value of reg_counter and value of new reg_counter is initialized as 10000000. Next, we calculated the time spend after each frame in frame sequence and if it is greater than refresh time we shifted every reg_counter right by 1 bit as next clock tick began.
We have implemented the above algorithm varying the page table size from 2 to maximum value of page number in the frame sequence.
INPUT
The frame sequence.
OUTPUT
The execution time of the algorithm for each page table size and also the number of page faults.

Testing the algorithms

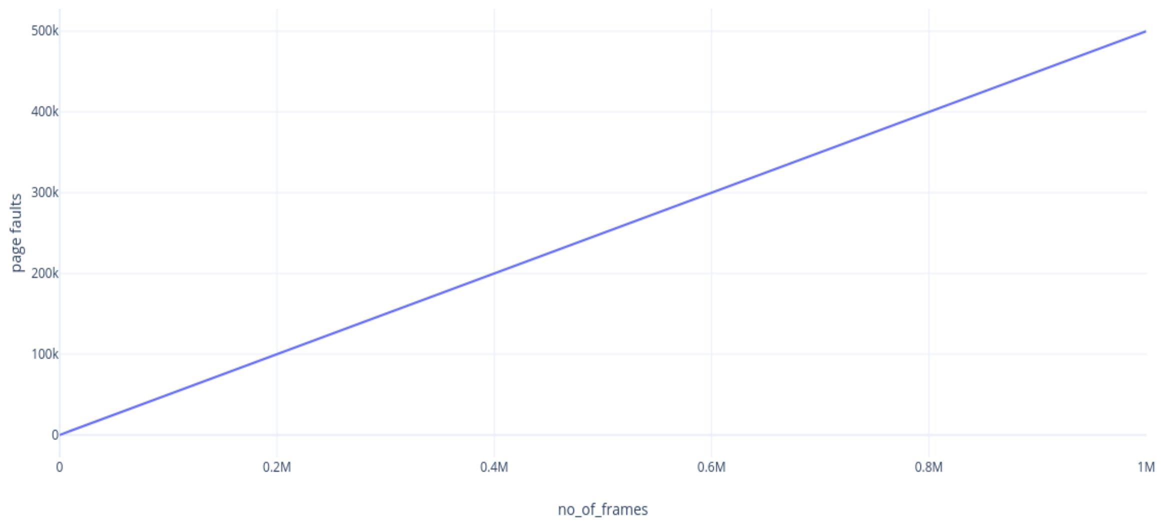
We have tested the algorithm with two test cases-
Test Case 1-
In this test case, there are 10^6 inputs in the frame sequence with the maximum value of page number as 100, the size of the page table varying from 2 to 100.



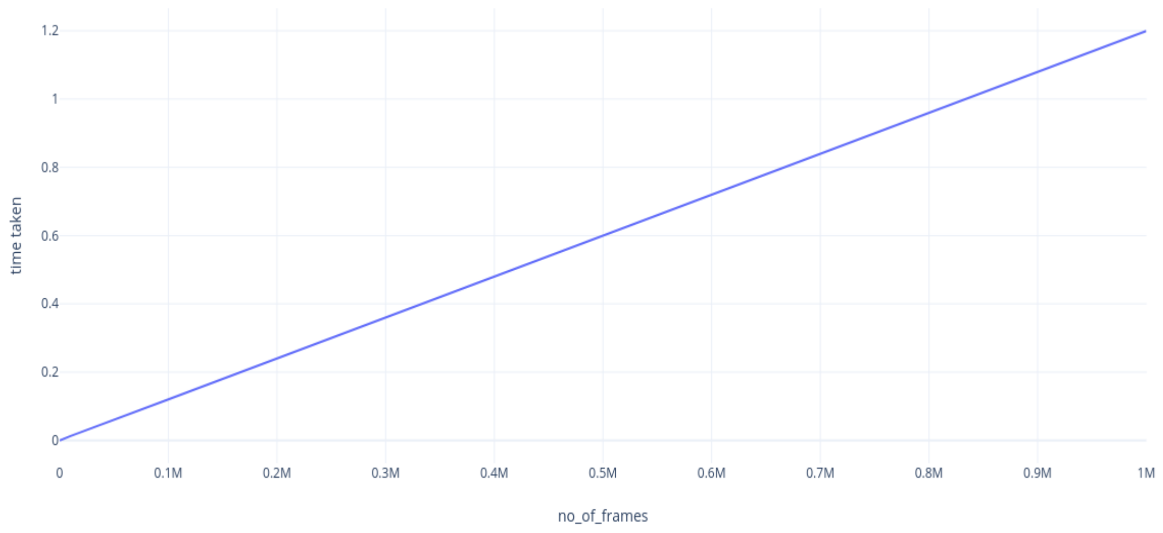
Above graph represent number of page faults vs no. of frames in the page table from 2 to 100 which is incremented by 1. As we can see from the graph when the no. of frames in the page table increases page faults decreases but when page table size is greater than 100 the no. of page faults is constant.



Above graph represent time taken in the execution of algorithm vs no. of frames in the page table. When the no. of frames in the page table increases, execution time decreases due to decrease in no. of page faults.



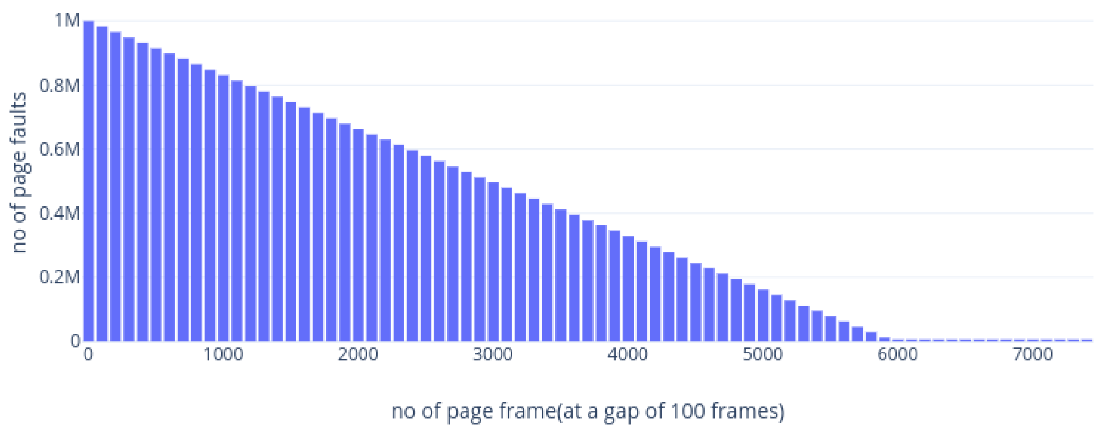
Above graph represent number of page faults vs no. of frames in the frame sequence with constant number of pages in the page table which is equal to 50. It is obvious that number of page faults increases with increase in the no. of frames in the frame sequence.



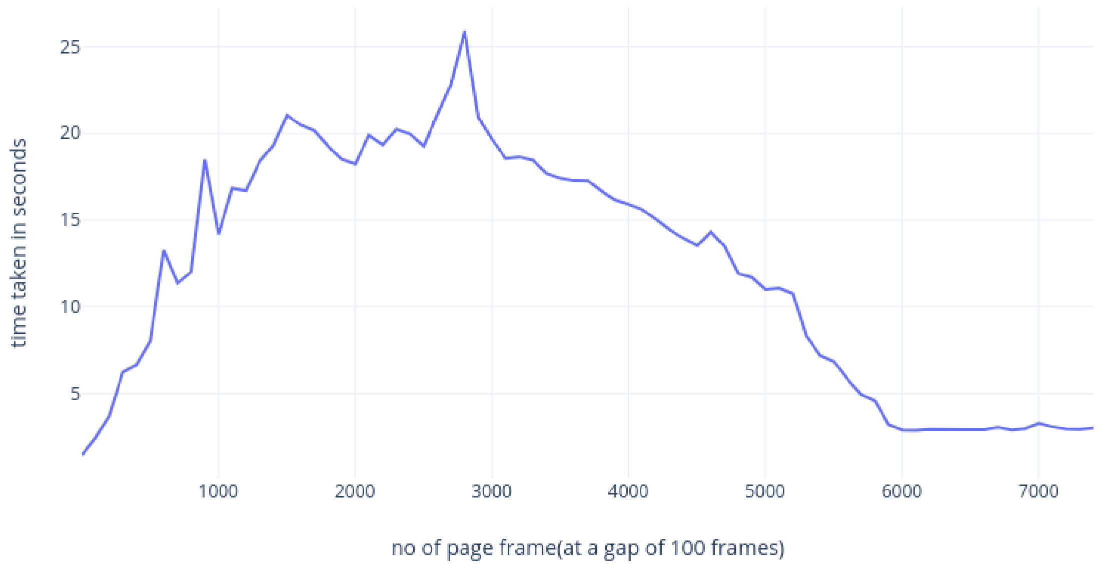
Above graph represent execution time vs no. of frames in the frame sequence with constant number of pages in the page table which is equal to 50. It is obvious that execution time increases with increase in the no. of frames in the frame sequence.

Test Case 2-

In this test case there are 10^6 inputs in the frame sequence with the maximum value of page number as 6000, the size of page table varying from 2 to 7500.



Above graph represent number of page faults vs no. of frames in the page table from 2 to 7500 which is incremented by 100. As we can see from the graph when the no. of frames in the page table increases page faults decreases but when page table size is greater than 6000 the no. of page faults is constant.



Above graph represent time taken in the execution of algorithm vs no. of frames in the page table. When the no. of frames in the page table increases, execution time first increases due to increase in size of page table is more prominent than decrease in no. of page faults but later the decrease in no. of page faults dominates the increase in page table size.

Complexity Analysis

If T(n) is time complexity of our LRU Aging algorithm,
f – length of input frame sequence.
n – number of pages in the page table.

m- number of page faults
p – page fault time.
c – some constant time.
Therefore, considering the worst-case analysis,
$T(n) = f*(n+m*(p+n))+c$
$= f*n+f*m*p+f*m*n+c$
$= O(f*m*n)$

Clock Implementation (Second Chance Algorithm) :(Approximate LRU)

Algorithm Explanation:

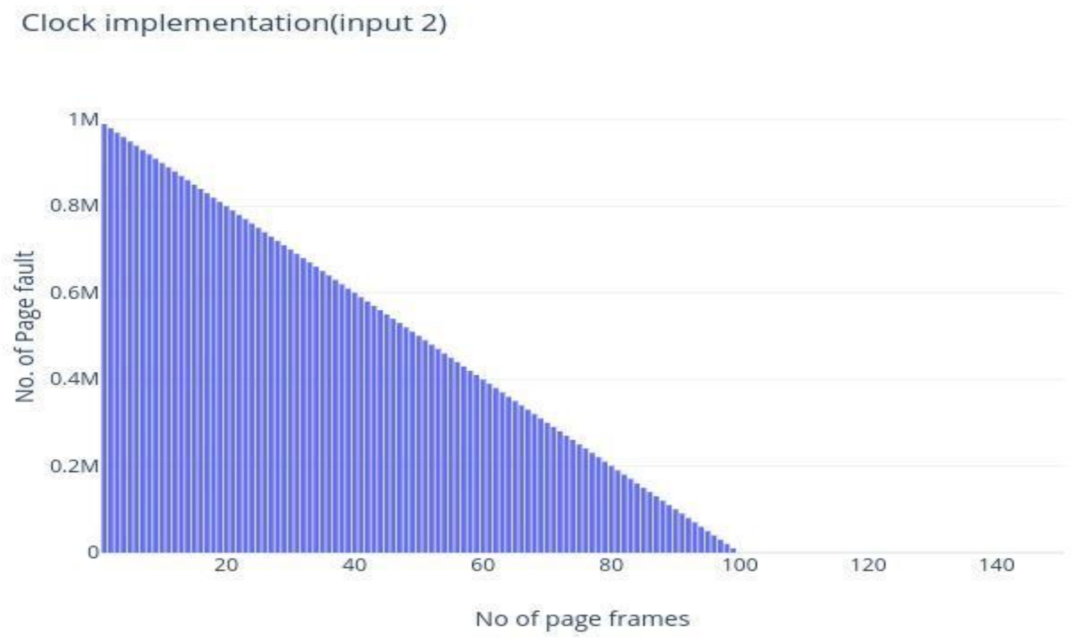
Clock replacement algorithm is a more efficient version of the FIFO replacement algorithm because pages don't have to be constantly pushed to the back of the list. The clock algorithm keeps a circular list of pages in the memory, with the replacement pointer pointing to the last examined page frame in the list. When a new page is required, the list is searched, if the page is found then we update the used bit of that page and then read the next input, else if there is a page fault and there are no empty frames on the circular queue, then the used bit is inspected at the replacement pointer's location. If the used bit is 0, that means that the second chance of the page is over, then it needs to be replaced. The new page is put in place of the page the replacement pointer points to. Otherwise, the used bit is cleared, then the clock hand is incremented until the used bit of some page is found as zero.

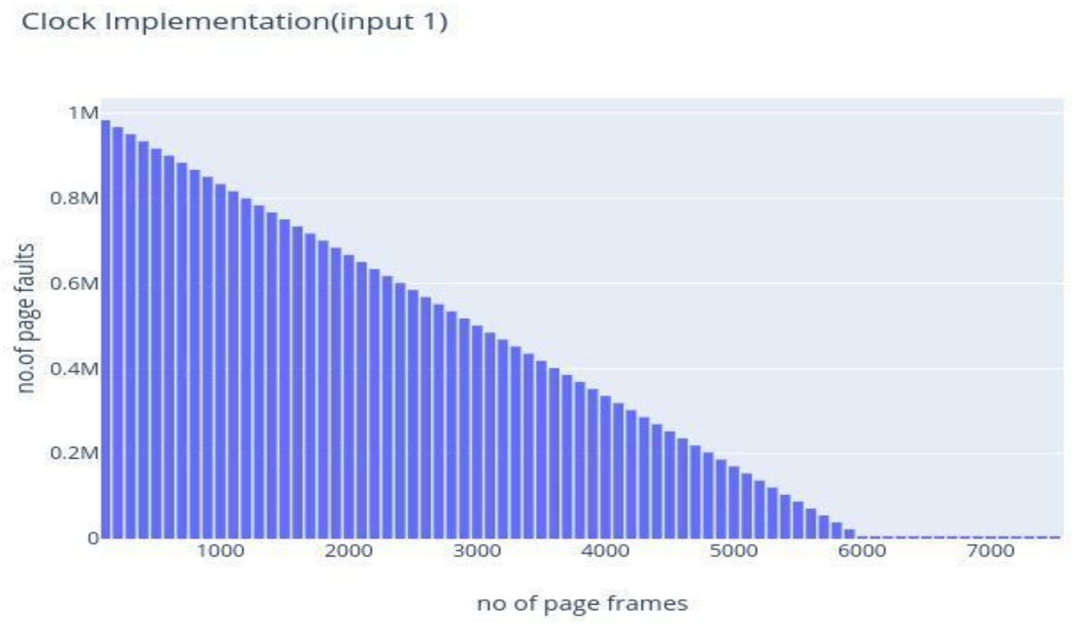
Code Explanation:

We have created two structs: page and virtual_mem which specify the pages to be inserted and the circular list. The page contains attributes like used bit and value to be stored. The virtual_mem struct contains the vector which is our circular list and other attributes. Our clock function is LRU_clock(). It takes in the number of page frames as a parameter. Then it takes in the input frame sequence, which we have taken as 10^6. Our vector clock_mem is the circular list and insert_page is a new page which will be inserted into the clock_mem during a page fault. Every time there is a page fault, we call the page_fault_delay() function. The page_fault_delay() will produce some delay. The replacement_pointer will point to the last examined page frame in the clock_mem. The reference_pointer is a kind of iterator which will iterate through clock_mem to search the required page. We have used two variables start and end to get the time taken for each different value of page frame. At last, we print them into the output file.

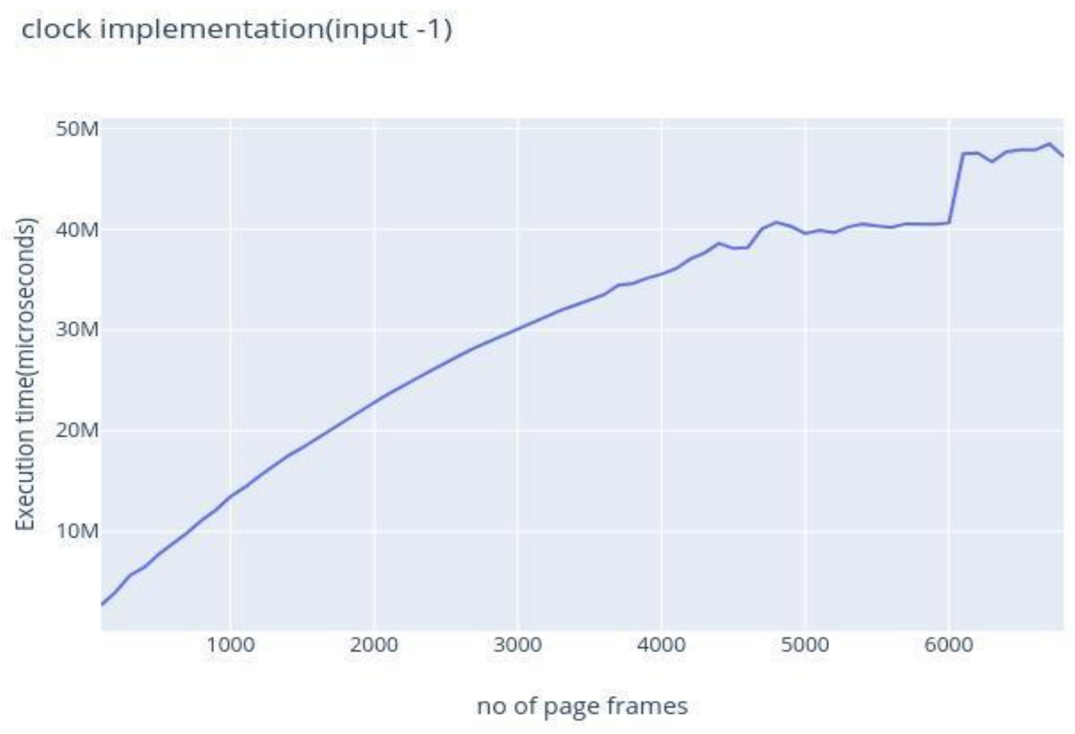
Observation :

We have taken two input files. Each file contains an input frame sequence of length 10^6. The first file contains a maximum value of 6000 and the second file contains a maximum value of 100. In each of the output files, we are printing a different number of page frames with their page faults and execution times, which counts till 7500-page frames in the first output file and 110-page frames in the second output file.

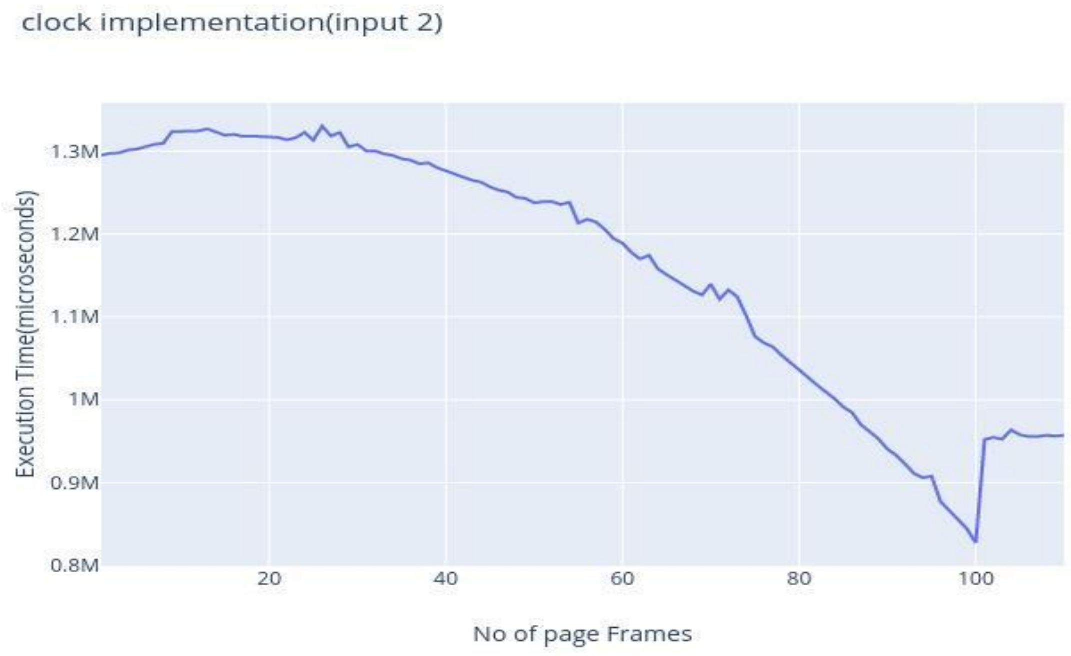




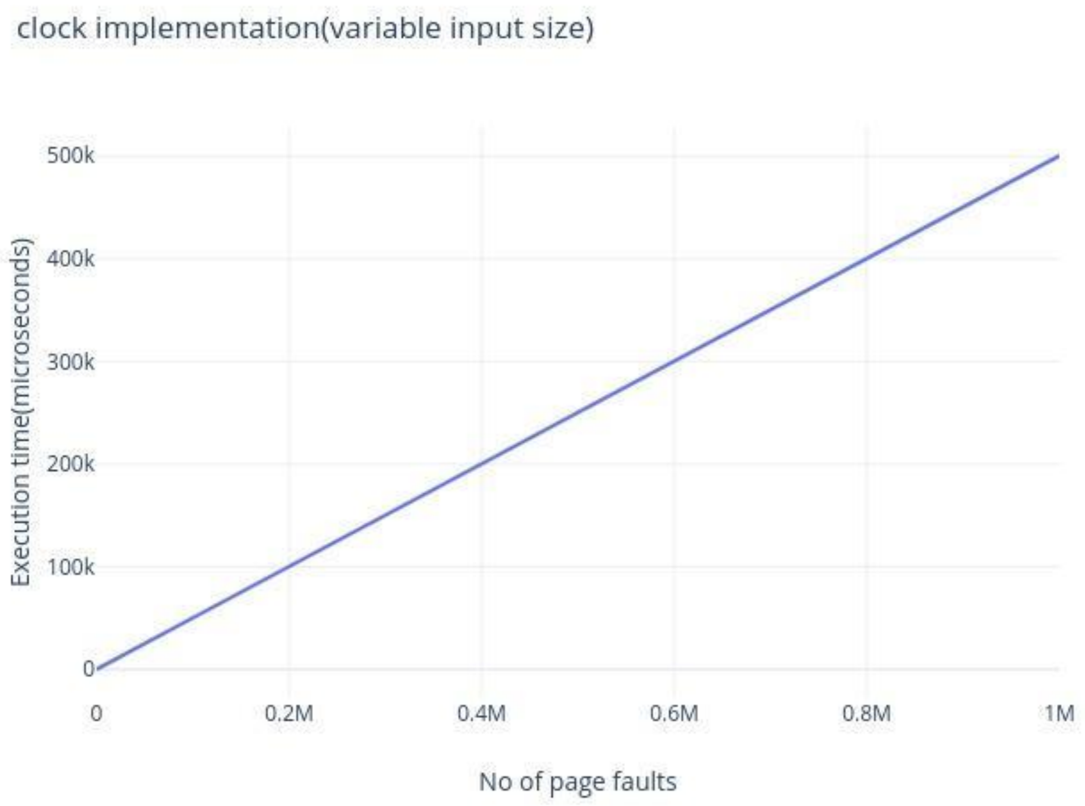
We can see that as the number of page frames increases, the number of page faults decreases.

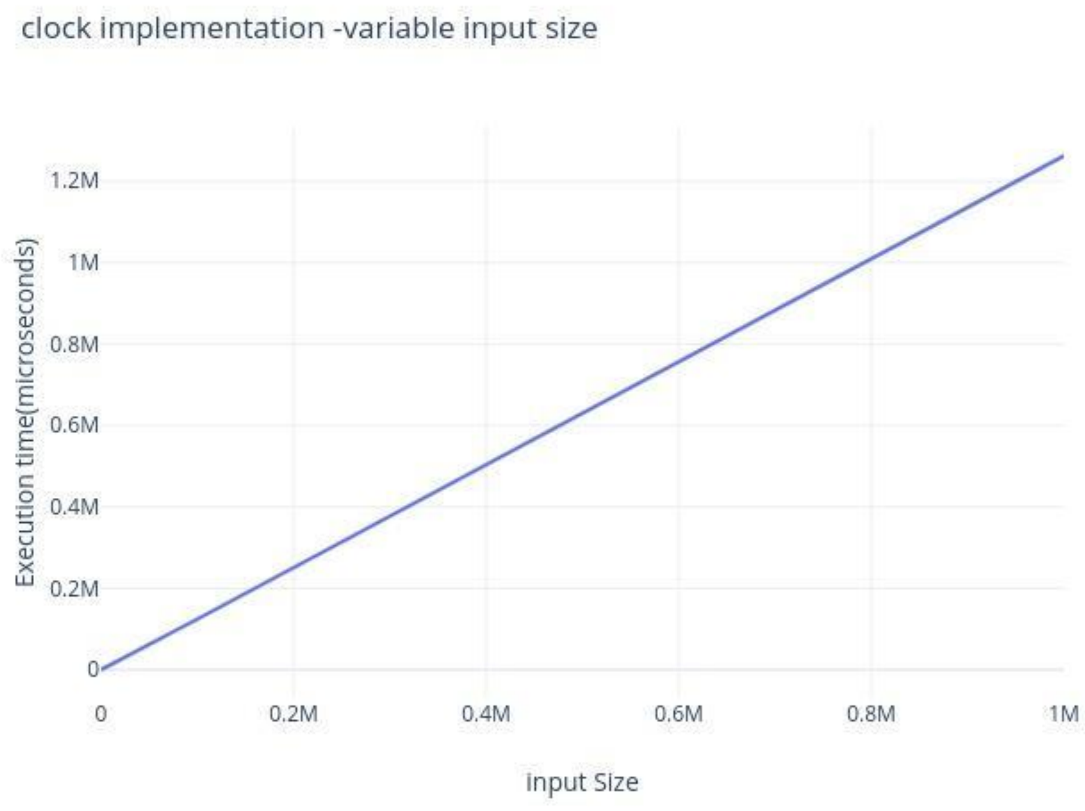


Here, as the input size of the frame sequence is very large, the time required to search the circular list is much more than the decrement in the time taking place due to the reduction in page faults. So, as the number of page frames is increasing the execution time is increasing. After the number of page entries exceeds the number of maximum value in the input frame sequence which is 6000 here, the execution time becomes constant.



Here, we can understand that as the number of frames are increasing, the number of page faults will reduce, due to which the execution time will reduce. After the number of page entries exceeds the number of maximum value in the input frame sequence which is 100 here, the execution time becomes constant.


clock implementation(variable input size)
Execution time(microseconds)
500k
400k
300k
200k
100k
0
0
0.2M
0.4M
0.6M
0.8M
1M
No of page faults



Here, we have fixed the size of the page frames as 50 and the size od the frame sequence is variable. So, as the input size increases, the number of page faults would increase, which will lead to the increment in execution time.

Complexity Analysis:

If T(n) is the time complexity of our LRU_clock(),
f – the length of the input frame sequence.
n – the number of pages in the list.
h – the number of page hits.
p – page fault time.

c – some constant time.
Therefore, considering the worst-case analysis,
$T(n) = f + h*n + (f-h)*p + c$
$= O(h*n + (f-h)*p)$
So, here we can see that as when the page frames as less, the hit rate will be less and hence, the page faults will be more and hence the execution time will be more at the start. As the number of page frames increases, the hit rate will increase and thus the number of page faults will decrease.