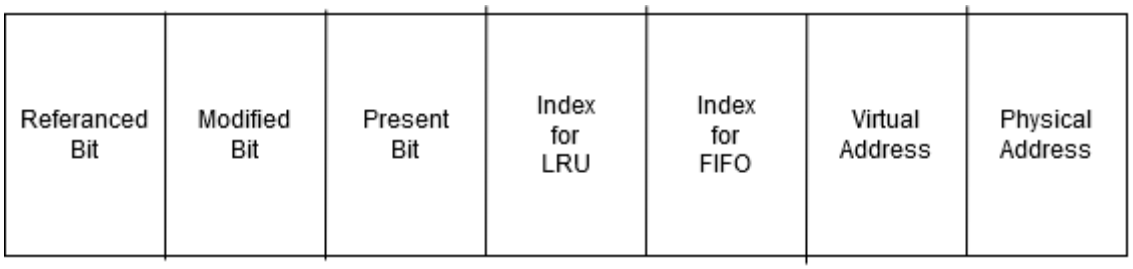
OPERATING SYSTEMS

FINAL PROJECT REPORT

Ahmet Yuşa Telli
151044092 | FINAL REPORT

PAGE TABLE

In Part 1, we design a page table structure. I use a C struct for page table. In this struct I have 3 bits (bool), 2 address spaces and 2 indexes (4 integers). There is a page table entry:



Present Bit: The entry is used or not.

Referenced: The entry is writing before. Did we access before?

Modified: The page has been modified or not. Did we write before?

Index for LRU: We take an index for LRU algorithm. We will explain in algorithms.

Index for FIFO: We take an index for FIFO algorithm. We will explain in algorithms.

Virtual Address: This address can store the virtual memory index.

Physical Address: This address can store the physical memory index.

We use page table as a bridge between virtual and physical memories. We fill virtual memory and when we are sorting it, we copy values to physical memory. We save the addresses in page table. We can easily find the location of the page in the virtual memory in the physical memory.

In page_table.cpp, we have some helper functions. We create empty page table and set bits to zero at the beginning. A function checks a free entry in page table. Another function checks the incoming virtual address is already existing and return its physical address. Other one is update an entry in page table, set bits 1, add virtual and physical addresses and update indexes for lru and fifo. Last one is clear reference bits for periodically. We also print page table in this file every counter access.

MEMORY:

We use a C struct for memories. The struct is “Page”. In Page struct we take an array and a valid bit.

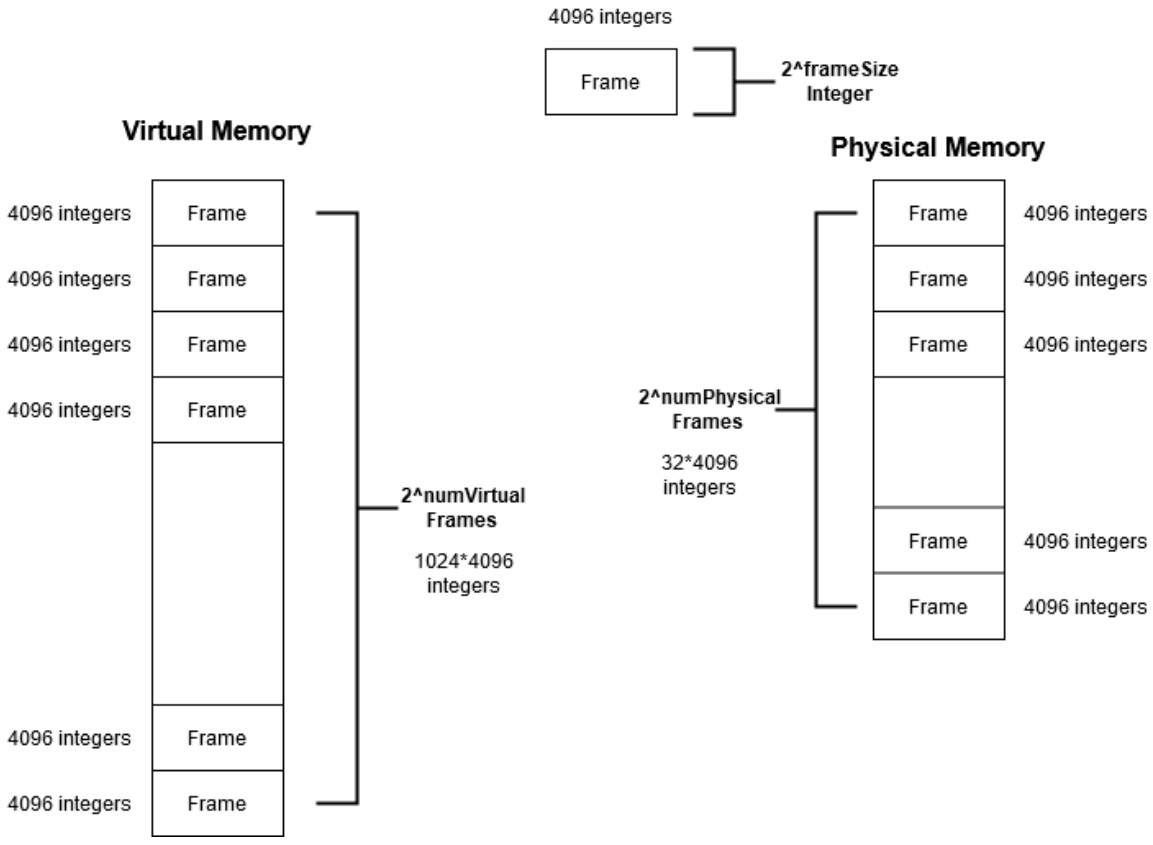
At the beginning, we take some spaces for physical and virtual memories using malloc.

- **Physical Memory:**

  In physical memory, we have some pages and valid bits. If physical memory is empty, we take a free page index and we take a page from virtual.

- **Virtual Memory:**

  In virtual memory, we fill random numbers. Then we sort each page with four sorting algorithms. We use four threads and each thread will sort a quarter of virtual memory.



GET / SET:

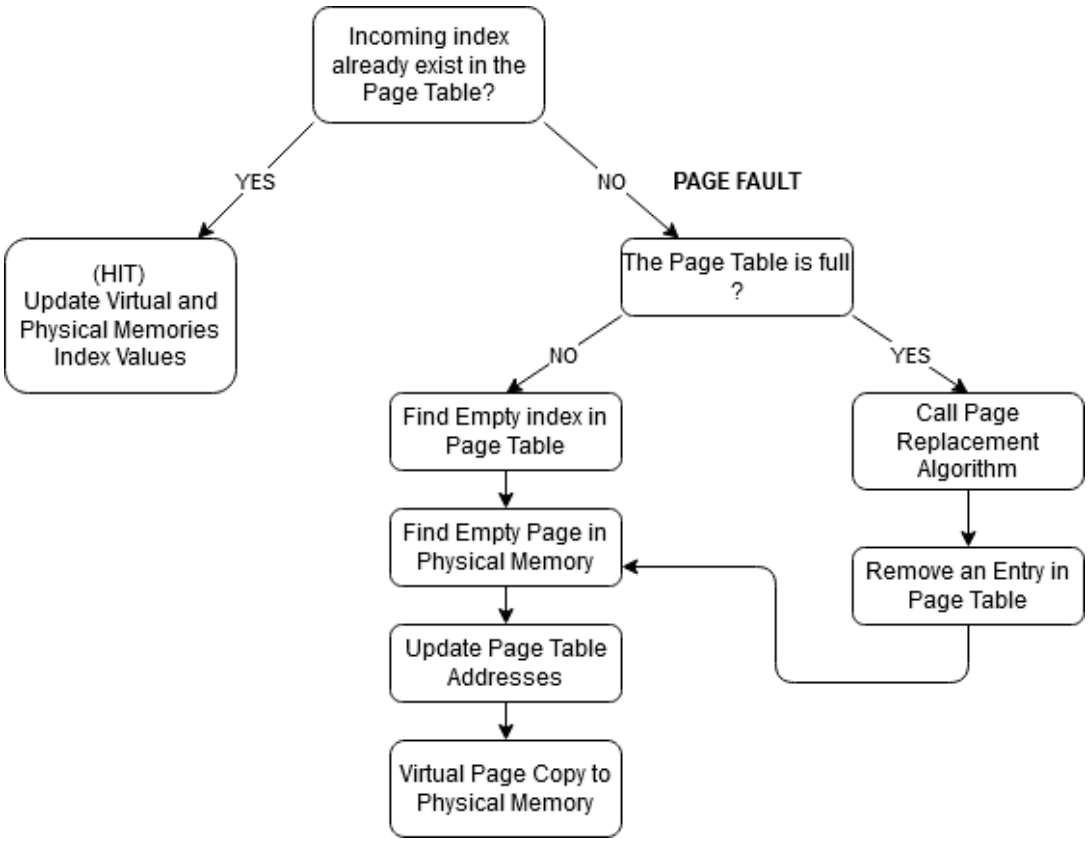
- **Set:**

At the beginning, we check the “local” and “global” index. If the coming thread wants to access other threads indexes, we can not let it in.

Then we check the coming index is already in page table. This is HIT. We update index value in virtual and physical memories. But the index is not in page table, this is Page Fault, we check page table.

If page table is full, we call page replacement algorithms and remove an entry from page table. If page table is not full, we find an empty page in physical memory and write virtual and physical memories indexes to page table. Then copy virtual memory values to physical memory.

This is “Set” diagram:



- **Get:**

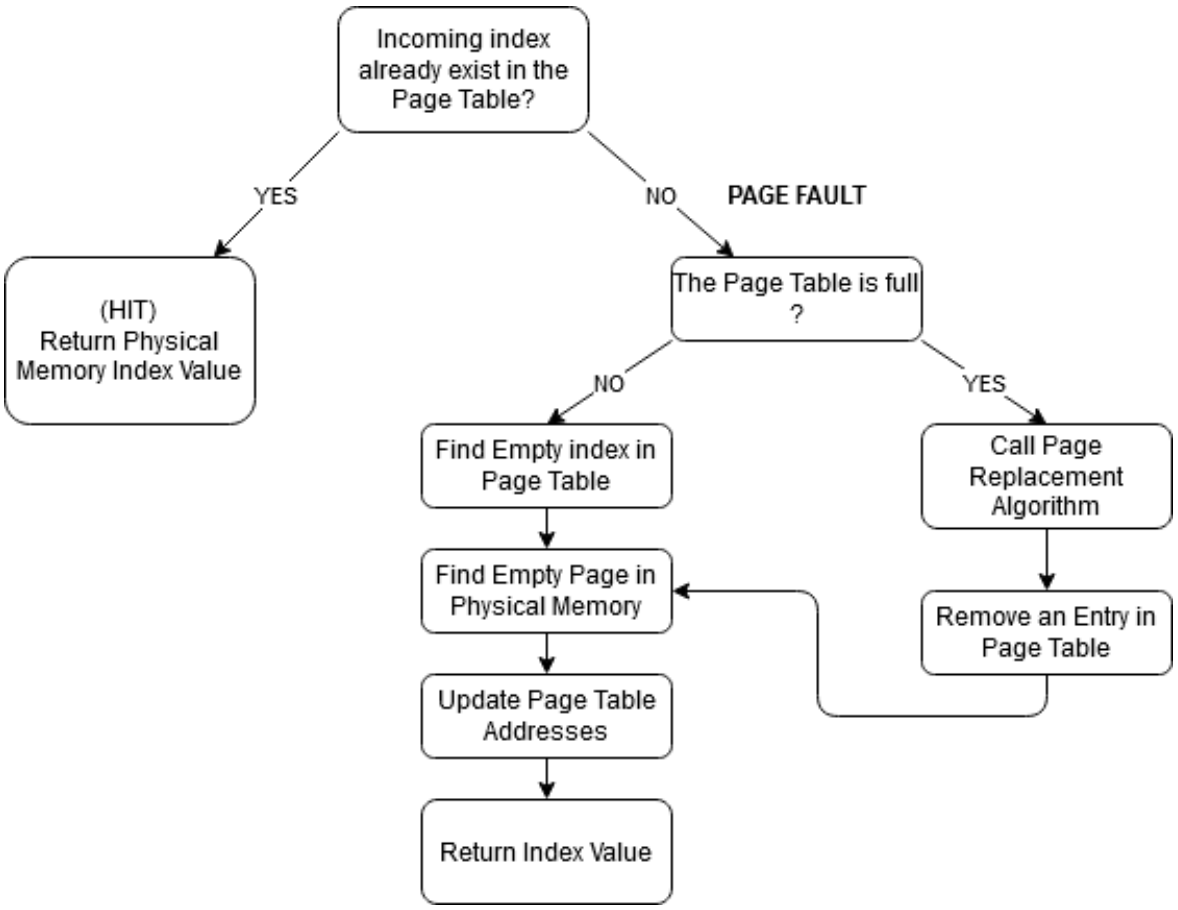
We do the same operations as the “Set” function. We find index values and return it.

First, we check the incoming index in page table. If it is already existing, HIT occurs. Then we return its value. If incoming page is a new page, we check the page table is full. If it is full, we have to make page replacement algorithm. If it is not full, we find a free entry index.

After that, we find physical memory free space and we connect each addresses in page table. Then, return index value.

In these two functions are critical area for four sorting thread. We must use mutex to work properly.

This is “Get” diagram:



ALGORITHMS:

Algorithms are in page_replacement.cpp file.

- ## LRU:

  For this algorithm, we store an index in page table. We write an index for each page table entry and we check this index. First entry is 0, second entry is 1 etc.

  When we have to do page replacement, we check this index and return the minimum index_for_lru. The important thing is, when HIT occurs, we update the index.

- ## FIFO:

  In this algorithm, we store an index in page table. We write an index for each page table entries.

  This algorithm is the same LRU. But we do not update the index when hit occurs.

- ## NRU:

  When we call the NRU algorithm, we check the referenced and modified bits. If there is a page that is not referenced and not modified, we remove this page. Then check referenced and not modified. If still, we did not find a page we check not reference and modified. Last time we check reference and modified. But we have found a page until the last check.

  The reference bit set 1, when we access the page. The modified bit set 1, when we chance the value. This means, when we call set function, we update modified bits and call get function we update reference bit. But there is an important thing, when page table access is equal to page table size, we set reference bits are zero. For NRU algorithm we need to reset reference bits.

- ## WSClock:

  In this algorithm, we take an index for page index. We start to check the page table at this index and check the reference bits. If reference bit is 1, we set is 0 then we check the next one. Until we find a page which has a reference bit is 0.

- ## Second Chance:

This algorithm is the same FIFO algorithm. We should find a page which has a minimum fifo index and reference bit is 0.

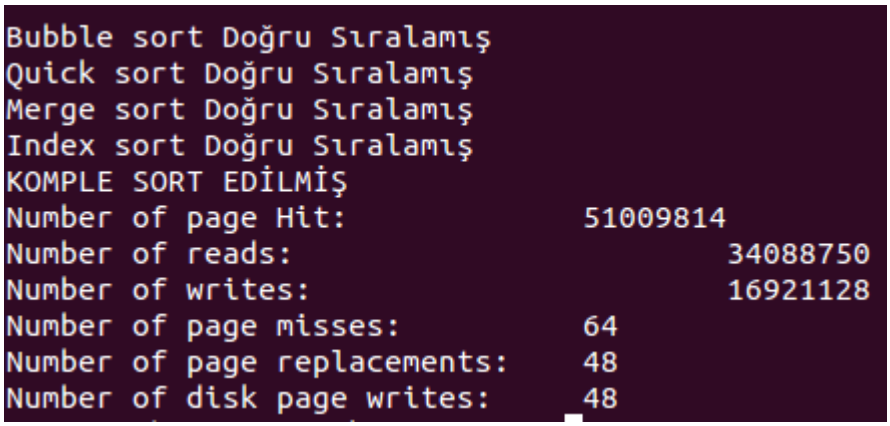
We check the fifo indexes and reference bits. We find the minimum fifo index page and check its reference bit.

PART 2:

In this part, we create virtual and physical memories and a page table. Page table structure explained before. We use dynamic allocation for these structures. We use page table and access so many times.

If "**`pageTablePrintInt`** " input is too small, it will be difficult to follow because the table will have a lot of writing. You should give this number is a million 1000000.

The result is like:



Number of reads: Number of read from virtual memory. This means, how many times we call get function.

Number of writes: Number of write to virtual memory. This means, how many times we call set function.

Number of page misses: How many times we have page fault.

Number of page replacements: Page table is full and there is a page fault. We have to remove an entry from page table. We call find_page_algorithm function and call a page replacement algorithm then find an entry index.

Number of disk page writes: If physical memory is full, we need an empty page. We select one page and write to disk. Then we reuse this page.

Number of disk page reads: reads from disk.

PART 3:

In this part 3, we calculate optimal parameters. Before start, we set the physical memory integer size is 16384 and virtual memory integer size is 131056 for calculate. (16K, 128K)

For run part 3 you should not enter numPhysical, numVirtual and frameSize.

The input like: *./sortArrays LRU local 10000 diskFileName.dat*

- **Part 3.1:**

We change the frame size and need to find optimal page size for each sorting algorithm.

For this calculation, we have a loop for frame size. The frame size starting from 16 up to 1024 in multiples of 2. Why we select 16 and 1024, explained in readme file. Then we count the page replacement for each sorting algorithm. We calculate number of pages in memories. For example: frameSize = 16. And Physical Memory can hold 16K integers. Then 16384 / 16 = 1024 pages in physical memory. Virtual memory can hold 128k integers. 131056 / 16 = 8192 pages in virtual memory.

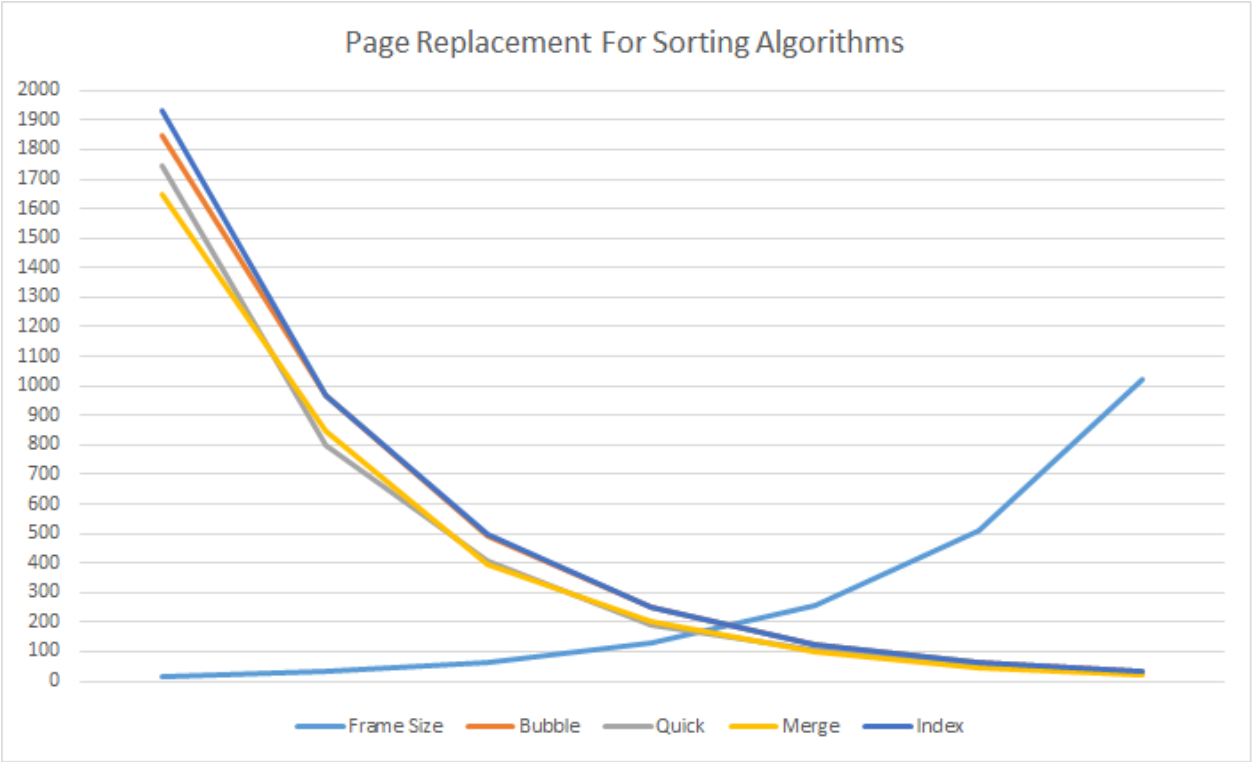
Here is my output:

```
Frame Size : 16
Page Table Size: 1024
Bubble Sort Page Replacement: 1847
Qucik Sort Page Replacement : 1743
Merge Sort Page Replacement : 1646
Index Sort Page Replacement : 1932

Frame Size : 32
Page Table Size: 512
Bubble Sort Page Replacement: 967
Qucik Sort Page Replacement : 801
Merge Sort Page Replacement : 846
Index Sort Page Replacement : 970

Frame Size : 64
Page Table Size: 256
Bubble Sort Page Replacement: 491
Qucik Sort Page Replacement : 405
Merge Sort Page Replacement : 396
Index Sort Page Replacement : 500

Frame Size : 128
Page Table Size: 128
Bubble Sort Page Replacement: 249
Qucik Sort Page Replacement : 193
Merge Sort Page Replacement : 204
Index Sort Page Replacement : 250
```

```
Frame Size : 256
Page Table Size: 64
Bubble Sort Page Replacement: 125
Qucik Sort Page Replacement : 105
Merge Sort Page Replacement : 94
Index Sort Page Replacement : 124

Frame Size : 512
Page Table Size: 32
Bubble Sort Page Replacement: 63
Qucik Sort Page Replacement : 54
Merge Sort Page Replacement : 44
Index Sort Page Replacement : 63

Frame Size : 1024
Page Table Size: 16
Bubble Sort Page Replacement: 31
Qucik Sort Page Replacement : 26
Merge Sort Page Replacement : 24
Index Sort Page Replacement : 31

Bubble Best Frame Size: 1024 Page Replacement: 31
Quick Best Frame Size: 1024 Page Replacement: 26
Merge Best Frame Size: 1024 Page Replacement: 24
Index Best Frame Size: 1024 Page Replacement: 31
```

For this results, optimum frame size is maximum frame size.

Page Replacements Graph:



- **Part 3.2:**

We need to find best replacement algorithm for each sort algorithm. We take frame size is 512.

We call each page replacement algorithm for two times. And we calculate average number of page replacement count.

We have two nested loops. One for chance algorithms and other one is for find average of page replacement.

For example, we start with LRU algorithm. We find page replacement numbers for each sort methods. Sum all bubble sort page replacements numbers and sum all quick sort page replacements numbers etc. Then we calculate average these numbers.

We compare these averages and find the best algorithm for LRU.

Then we start other algorithms.

The output is:

```
Bubble sort Average 63 for LRU
Bubble sort Average 63 for WSClock
Quick sort Average 49 for LRU
Merge sort Average 45 for WSClock
Merge sort Average 45 for SC
Index sort Average 63 for LRU
Index sort Average 63 for WSClock
```

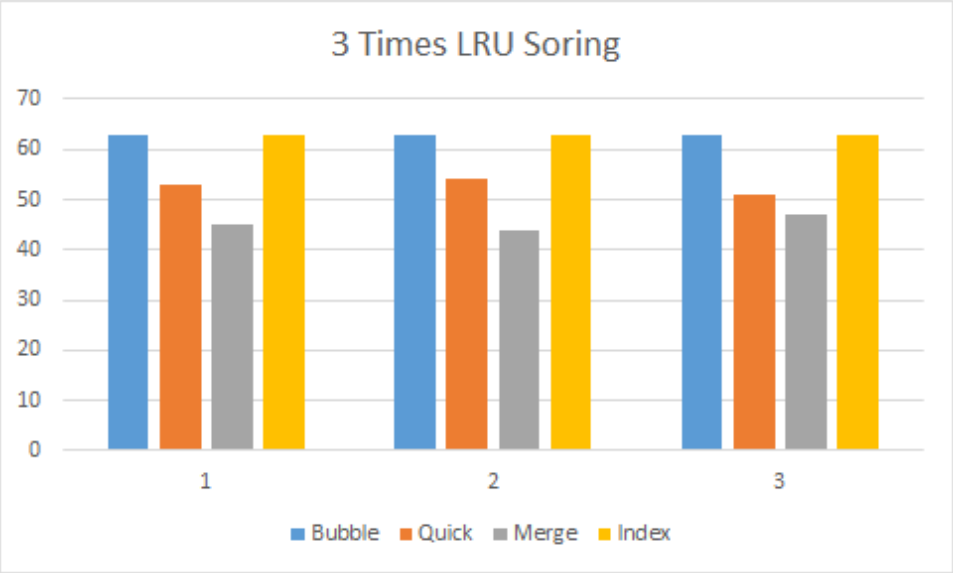
For these result the,

Best page replacement algorithm is LRU and WSClock for bubble sort.

Best page replacement algorithm is LRU for quick sort.

Best page replacement algorithm is WSClock and SC for merge sort.

Best page replacement algorithm is LRU and WSClock for index sort.



Thank you

Ahmet Yuşa Telli

151044092