Republic of the Philippines
Tarlac State University
COLLEGE OF COMPUTER STUDIES
Tarlac City, Tarlac
Tel. No. (045) 6068173

Step Algo: A Page-Replacement Algorithm Visualizer

Angelo V. Miranda

Instructor:

Jo Anne S. Cura

Table of Contents

I. Introduction

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in. Page replacement becomes necessary when a page fault occurs and no free page frames are in memory. in this article, we will discuss different types of page replacement algorithms.

1.1 Page Replacement Algorithms

Page replacement algorithms are techniques used in operating systems to manage memory efficiently when the physical memory is full. When a new page needs to be loaded into physical memory, and there is no free space, these algorithms determine which existing page to replace.

If no page frame is free, the virtual memory manager performs a page replacement operation to replace one of the pages existing in memory with the page whose reference caused the page fault. It is performed as follows: The virtual memory manager uses a page replacement algorithm to select one of the pages currently in memory for replacement, accesses the page table entry of the selected page to mark it as "not present" in memory, and initiates a page-out operation for it if the modified bit of its page table entry indicates that it is a dirty page.

1.1.1 First-In-First-Out (FIFO) Algorithm

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

1.1.2 Last Recently Used (LRU) Algorithm]

LRU replaces the page that has not been used for the longest period of time, based on the assumption that pages used recently will likely be used again soon.

1.1.3 Optimal Page Replacement

The Optimal algorithm replaces the page that will not be used for the longest time in the future. It provides the lowest possible page fault rate but requires future knowledge of page requests, making it impractical for real-time use.

1.2 Comparison of Time and Space Complexities of each Algorithms

Algorithm	Time Complexity	Space Complexity
First-In-First-Out (FIFO)	O(n) per operation (naïve), O(1) with hash for lookup + queue	O(n)
Last Recently Used (LRU)	O(n) naïve, O(1) optimized with HashMap + Doubly Linked List	O(n) frames + O(n) recency metadata
Optimal (OPT)	O(n) per replacement (scan future)	Needs full knowledge of future — only usable for benchmarking

1.3 Analysis

When simulating the FIFO, LRU, and Optimal page replacement algorithms, the most efficient one is identified based on the lowest number of page faults. If a single algorithm produces fewer page faults than the others, it is directly considered the most efficient for the given reference string and memory size. This straightforward case reflects a clear performance advantage and requires no further comparison.

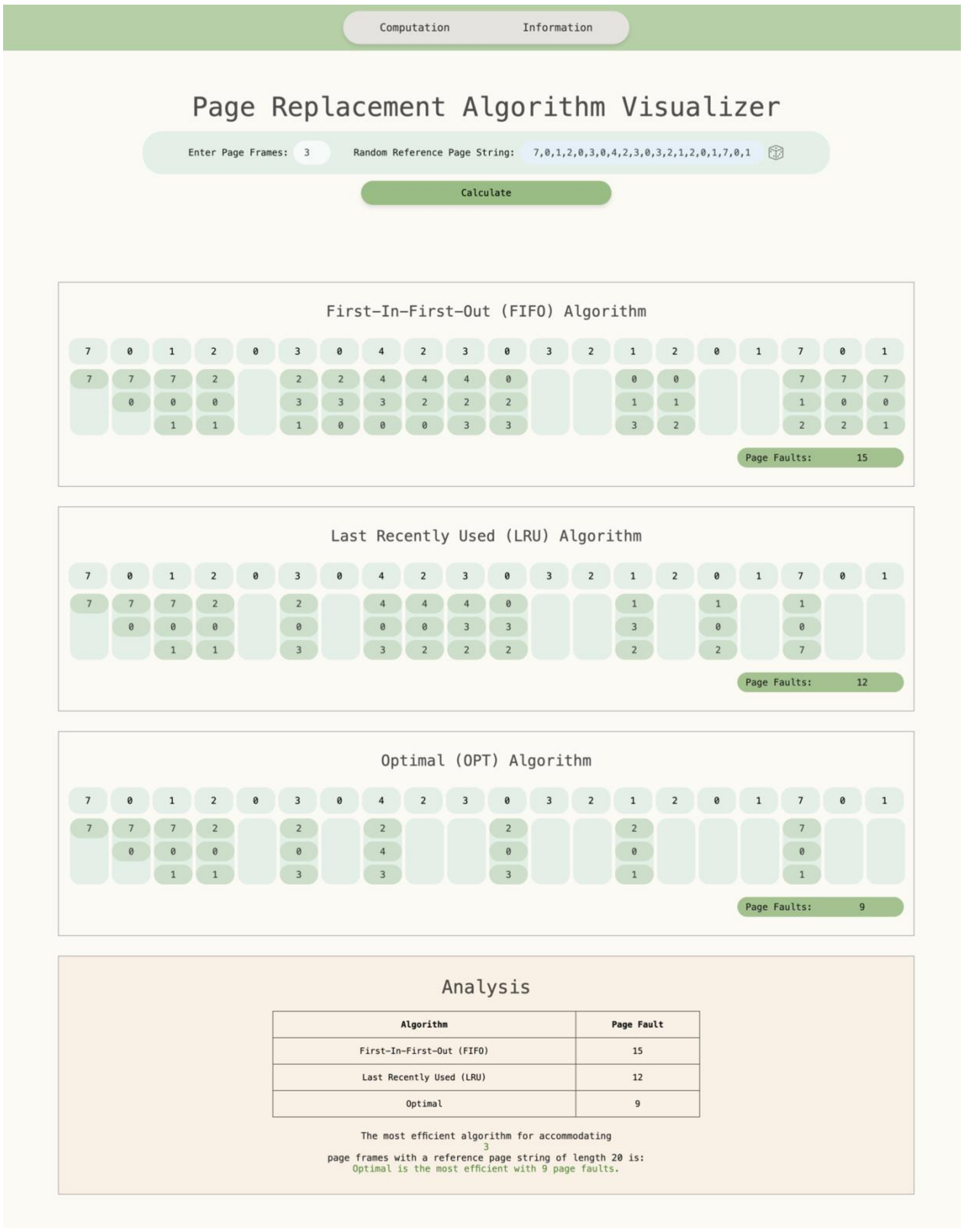
In scenarios where all three algorithms FIFO, LRU, and Optimal, have the same number of page faults, the system applies a tie-breaking preference. Among them, LRU is recommended due to its strong balance between practical efficiency and real-world usability. While FIFO is easy to implement, it may remove frequently used pages and suffers from Belady's anomaly. Optimal, though theoretically the best with minimal page faults, is impractical for real systems because it requires advance knowledge of future page references.

If two algorithms are tied for the lowest page fault count, the recommendation depends on which algorithms are involved. If LRU is among the tied algorithms, it is preferred because of its consistent and realistic performance. If FIFO is part of the tie but LRU is not, FIFO is chosen for its simplicity. However, if the tie includes Optimal but not LRU, then Optimal is acknowledged for its theoretical excellence, but still considered less practical due to its dependence on future information. These tie-handling rules ensure that algorithm selection is based not only on performance, but also on real-world feasibility.

II. Documentation

2.1 User Interface

2.1.1 Computation Tab



Page Replacement Algorithm Visualizer

Enter Page Frames: 3   Random Reference Page String: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

Calculate

First-In-First-Out (FIFO) Algorithm

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

Page Faults: 15

Last Recently Used (LRU) Algorithm

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

Page Faults: 12

Optimal (OPT) Algorithm

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

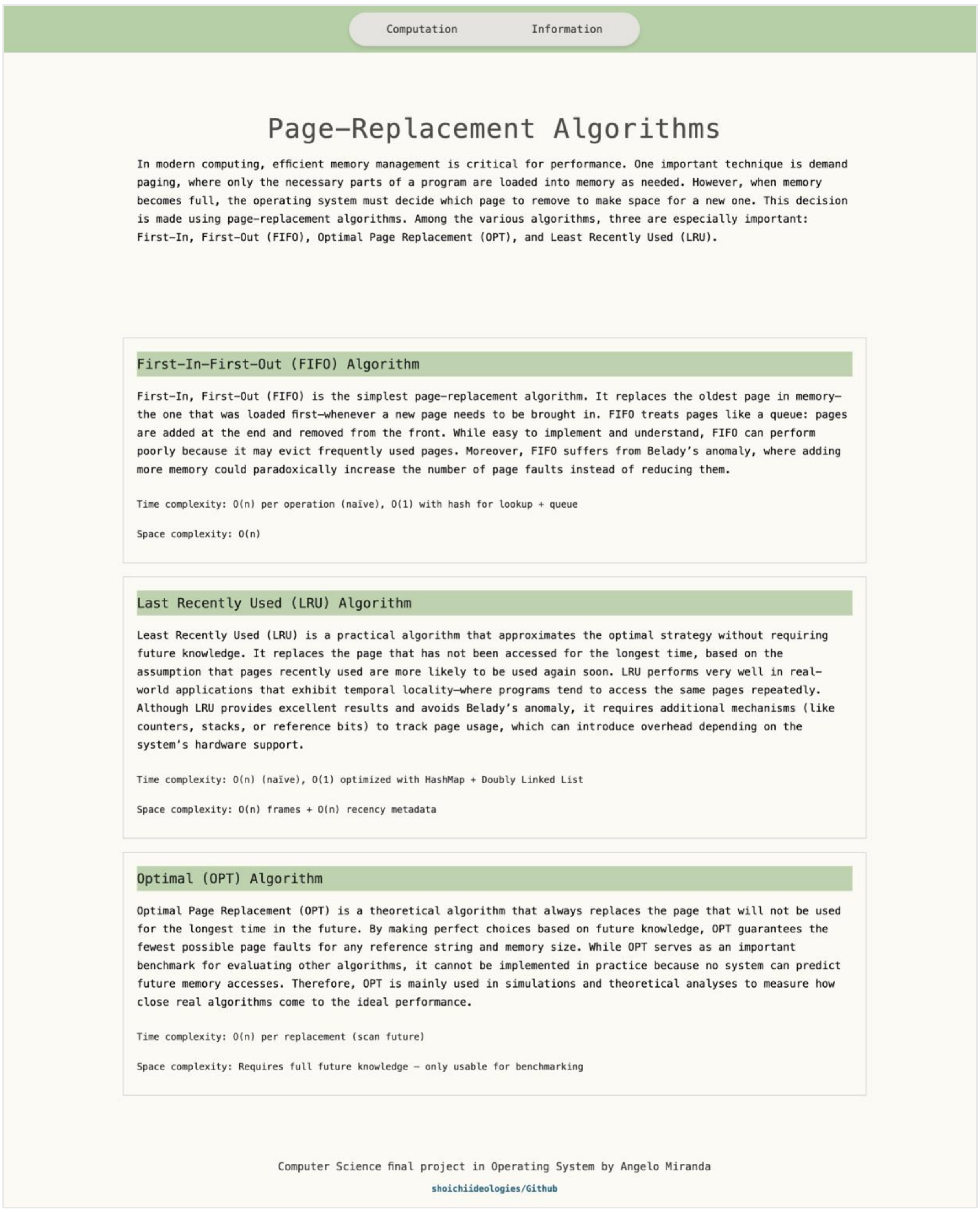
Page Faults: 9

Analysis

Algorithm	**Page Fault**
First-In-First-Out (FIFO)	15
Last Recently Used (LRU)	12
Optimal	9

The most efficient algorithm for accommodating
3
page frames with a reference page string of length 20 is:
Optimal is the most efficient with 9 page faults.

2.1.2 Information Tab



Page-Replacement Algorithms

In modern computing, efficient memory management is critical for performance. One important technique is demand paging, where only the necessary parts of a program are loaded into memory as needed. However, when memory becomes full, the operating system must decide which page to remove to make space for a new one. This decision is made using page-replacement algorithms. Among the various algorithms, three are especially important: First-In, First-Out (FIFO), Optimal Page Replacement (OPT), and Least Recently Used (LRU).

First-In-First-Out (FIFO) Algorithm

First-In, First-Out (FIFO) is the simplest page-replacement algorithm. It replaces the oldest page in memory—the one that was loaded first—whenever a new page needs to be brought in. FIFO treats pages like a queue: pages are added at the end and removed from the front. While easy to implement and understand, FIFO can perform poorly because it may evict frequently used pages. Moreover, FIFO suffers from Belady's anomaly, where adding more memory could paradoxically increase the number of page faults instead of reducing them.

Time complexity: O(n) per operation (naïve), O(1) with hash for lookup + queue

Space complexity: O(n)

Last Recently Used (LRU) Algorithm

Least Recently Used (LRU) is a practical algorithm that approximates the optimal strategy without requiring future knowledge. It replaces the page that has not been accessed for the longest time, based on the assumption that pages recently used are more likely to be used again soon. LRU performs very well in real-world applications that exhibit temporal locality—where programs tend to access the same pages repeatedly. Although LRU provides excellent results and avoids Belady's anomaly, it requires additional mechanisms (like counters, stacks, or reference bits) to track page usage, which can introduce overhead depending on the system's hardware support.

Time complexity: O(n) (naïve), O(1) optimized with HashMap + Doubly Linked List

Space complexity: O(n) frames + O(n) recency metadata

Optimal (OPT) Algorithm

Optimal Page Replacement (OPT) is a theoretical algorithm that always replaces the page that will not be used for the longest time in the future. By making perfect choices based on future knowledge, OPT guarantees the fewest possible page faults for any reference string and memory size. While OPT serves as an important benchmark for evaluating other algorithms, it cannot be implemented in practice because no system can predict future memory accesses. Therefore, OPT is mainly used in simulations and theoretical analyses to measure how close real algorithms come to the ideal performance.

Time complexity: O(n) per replacement (scan future)

Space complexity: Requires full future knowledge — only usable for benchmarking

Computer Science final project in Operating System by Angelo Miranda
shoichiideologies/Github

2.2 Source Code

2.1.3 Renderer.js

2.2.1.1 Analysis

```
function determineBestAlgorithm() {
        const results = {
                FIFO: fifoPageFaults,
                LRU: lruPageFaults,
                Optimal: optimalPageFaults,
        };

        const minFaults = Math.min(...Object.values(results));

        // Find all algorithms that have the minimum fault count
        const bestAlgorithms = Object.entries(results)
                .filter(([_, faults]) => faults === minFaults)
                .map(([algo]) => algo);

        const outputElement = document.querySelector('#efficient-algo');

        if (bestAlgorithms.length === 1) {
                outputElement.textContent = `${bestAlgorithms[0]} is the most efficient with ${minFaults} page
faults.`;
        } else if (bestAlgorithms.length === 3) {
                // All tied — compare by complexity
                outputElement.textContent
   = `All algorithms have ${minFaults} page faults. Based on time and space complexity, `
   + 'LRU is preferred because it performs well in practice and does not require future knowledge like Optimal.';
        } else {
                // Tie between 2 algorithms
                const list = bestAlgorithms.join(' and ');
                let recommended = '';

                if (bestAlgorithms.includes('LRU')) {
                        recommended = 'LRU is preferred due to better practical efficiency.';
                } else if (bestAlgorithms.includes('FIFO')) {
                        recommended = 'FIFO is preferred for its simplicity.';
                } else {
                        recommended = 'Optimal is theoretically best, but not practical in real-world systems.';
                }

                outputElement.textContent
   = `${list} are tied with ${minFaults} page faults. ${recommended}`;
        }
}
```

2.2.1.2 Simulate Algorithms

```
function simulateAlgorithms(referenceString, frameCount) {
        const simulations = document.querySelectorAll('.simulation');

        fifoQueue = [];
        fifoPageFaults = 0;
        lruPageFaults = 0;
        optimalPageFaults = 0;

        for (const simulation of simulations) {
                const algoId = simulation.id;
                const pageContainer = simulation.querySelector('.page-container');
                const pageDivs = pageContainer.querySelectorAll('.page');

                for (const pageDiv of pageDivs) {
                        pageDiv.innerHTML = '';
                }

                const frames = [];

                for (const [index, pageNumber] of referenceString.entries()) {
                        let pageFault = false;

                        switch (algoId) {
                        case 'fifo': {
                                pageFault = simulateFIFO(frames, frameCount, pageNumber);

                                break;
                        }

                        case 'lru': {
                                pageFault = simulateLRU(frames, frameCount, pageNumber,
referenceString.slice(0, index));

                                break;
                        }

                        case 'optimal': {
                                pageFault = simulateOptimal(frames, frameCount, pageNumber,
referenceString.slice(index + 1));

                                break;
                        }
                        // No default
                        }

                        const pageDiv = pageDivs[index];

                        if (pageFault) {
                                for (const framePage of frames) {
                                        const pageResult = document.createElement('div');
                                        pageResult.className = 'page-results';
                                        pageResult.textContent = framePage;
                                        pageResult.style.width = '100%';
                                        pageResult.style.height = '30px';
                                        pageResult.style.borderRadius = 'inherit';
                                        pageDiv.append(pageResult);
                                        }
                                }
                                // Otherwise, leave blank for page hits
                        }

                        // After loop, update page fault result
                        switch (algoId) {
                        case 'fifo': {
                                document.querySelector('#fifo-page-faults-result').textContent = fifoPageFaults;
                                document.querySelector('#fifo-page-fault').textContent = fifoPageFaults;

                                break;
                        }

                        case 'lru': {
                                document.querySelector('#lru-page-faults-result').textContent = lruPageFaults;
                                document.querySelector('#lru-page-fault').textContent = lruPageFaults;

                                break;
                        }

                        case 'optimal': {
                                document.querySelector('#optimal-page-faults-result').textContent = optimalPageFaults;
                                document.querySelector('#optimal-page-fault').textContent = optimalPageFaults;

                                break;
                        }
                        // No default
                        }
                }

                // Compare the page faults to find the best algorithm
                determineBestAlgorithm();
}
```

2.2.1.2.1 FIFO

```
let lruPageFaults = 0; // Counter for LRU faults (declare global like fifoPageFaults)

function simulateLRU(frames, frameCount, currentPage, pastPages) {
        let changed = false;

        if (!frames.includes(currentPage)) {
                changed = true; // It is a page fault
                lruPageFaults++;

                if (frames.length < frameCount) {
                        frames.push(currentPage);
                } else {
                        // Find the least recently used page
                        let lruIndex = -1;
                        let lruPage = null;
```

```
                    for (const page of frames) {
                        const lastUsed = pastPages.lastIndexOf(page);
                        if (lruIndex === -1 || lastUsed < lruIndex) {
                            lruIndex = lastUsed;
                            lruPage = page;
                        }
                    }

                    const replaceIndex = frames.indexOf(lruPage);
                    frames[replaceIndex] = currentPage; // Replace LRU page in same position
                }
        }

        return changed; // Return true if page fault occurred
}

let optimalPageFaults = 0; // Counter for Optimal faults (global)
```

2.2.1.2.2 LRU

```
function simulateOptimal(frames, frameCount, currentPage, futurePages) {
        let changed = false;

        if (!frames.includes(currentPage)) {
                changed = true;
                optimalPageFaults++;

                if (frames.length < frameCount) {
                    frames.push(currentPage);
                } else {
                    let farthestIndex = -1;
                    let pageToReplace = null;

                    for (const page of frames) {
                        const nextUse = futurePages.indexOf(page);

                        if (nextUse === -1) {
                            // This page is never used again → best to replace
                            farthestIndex = Infinity;
                            pageToReplace = page;
                        } else if (nextUse > farthestIndex) {
                            // Find page used farthest in future
                            farthestIndex = nextUse;
                            pageToReplace = page;
                        }
                    }

                    const replaceIndex = frames.indexOf(pageToReplace);
                    frames[replaceIndex] = currentPage; // Replace at the same slot
                }
        }
```

```
    return changed; // Return whether page fault occurred
}
```

2.2.1.2.3 OPT

```
let fifoQueue = []; // FIFO queue for arrival order
let fifoPageFaults = 0; // Counter for page faults

function simulateFIFO(frames, frameCount, currentPage) {
    let changed = false;

    if (!frames.includes(currentPage)) {
        changed = true; // It is a page fault
        fifoPageFaults++;

        if (frames.length < frameCount) {
            frames.push(currentPage);
            fifoQueue.push(currentPage);
        } else {
            const oldestPage = fifoQueue.shift(); // Remove oldest
            const oldestIndex = frames.indexOf(oldestPage); // Find where it is
            frames[oldestIndex] = currentPage; // Replace at same spot
            fifoQueue.push(currentPage);
        }
    }

    return changed; // Return whether page fault occurred
}
```