Booth’s algorithm

```python
from binary_subtraction import binary_adder, twos_complement

def booth(M, Q):
    count = max(len(M), len(Q))
    M = M.zfill(count)
    Q = Q.zfill(count)

    M_comp = twos_complement(M)

    Q_1 = "0"
    # initialize accumulator with zero
    A = "0" * count

    for i in range(count):
        deciding_bits = Q[-1] + Q_1
        if deciding_bits == "10":
            A = binary_adder(A, M_comp)
        elif deciding_bits == "01":
            A = binary_adder(A, M)
        # Arithmentic Shift Right (ASR A, Q, Q_-1)
        Q_1 = Q[-1]
        Q = A[-1] + Q[:-1]
        A = A[0] + A[:-1]

    return A + Q

if __name__ == "__main__":
    M = input("Enter multiplicand: ")
    Q = input("Enter multiplier: ")
    product = booth(M, Q)
    print(f"Product = {product}")
```

```
PS C:\Programming\COA> python booth_algorithm.py
Enter multiplicand: 00000011
Enter multiplier: 11111011
Product = 1111111111110001
PS C:\Programming\COA> python booth_algorithm.py
Enter multiplicand: 11111101
Enter multiplier: 11111011
Product = 0000000000001111
PS C:\Programming\COA> python booth_algorithm.py
Enter multiplicand: 00000011
Enter multiplier: 00000010
Product = 0000000000000110
```

This is a Python program that uses Booth's algorithm to multiply two signed binary numbers. It accepts two binary strings as arguments — the multiplier (`Q`) and the multiplicand (`M`) — and carries out a series of bitwise and arithmetic operations to compute their product. The algorithm utilizes an accumulator (A), a one-bit register (Q_1), and the two's complement of the multiplicand (M_comp) to determine whether it needs to

add, subtract, or nothing in accordance with the previous bit of the multiplier and the content of Q_1. In each step, it executes a conditional add and an arithmetic right shift of the running totals of A, Q, and Q_1. After repeated applications of the same steps for as many steps as there are in the bit length of inputs, the binary product is arrived at by concatenating `A` and `Q`. The implementation uses two helper functions binary_adder` and `twos_complement` — to perform binary addition and two's complement.

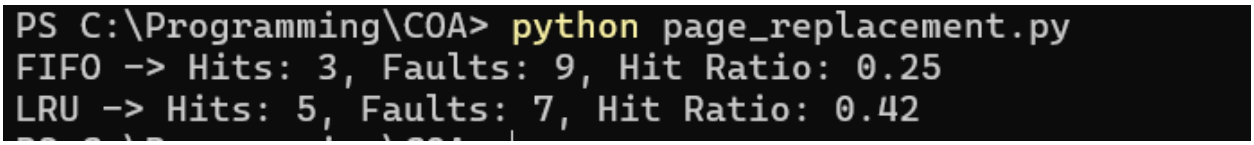
Page replacement

```python
def fifo(pages, capacity):
    queue = []
    faults = 0
    for page in pages:
        if page not in queue:
            if len(queue) == capacity:
                queue.pop(0)
            queue.append(page)
            faults += 1
    hits = len(pages) - faults
    ratio = hits / len(pages)
    return hits, faults, ratio

def lru(pages, capacity):
    memory = []
    faults = 0
    for page in pages:
        if page not in memory:
            if len(memory) == capacity:
                memory.pop(0)
            memory.append(page)
            faults += 1
        else:
            memory.remove(page)
            memory.append(page)
    hits = len(pages) - faults
    ratio = hits / len(pages)
    return hits, faults, ratio

def lfu(pages, capacity):
    faults = 0
    memory = []
    freq = {}
    for page in pages:
        if page not in memory:
            if len(memory) == capacity:
                # Find LFU page
                min_freq = min(freq.values())
                lfu_pages = [p for p in memory if freq[p] == min_freq]
                to_remove = lfu_pages[0]
                memory.remove(to_remove)
                freq.pop(to_remove)
            memory.append(page)
            freq[page] = 1
            faults += 1
        else:
            freq[page] += 1
    hits = len(pages) - faults
    ratio = hits / len(pages)
    return hits, faults, ratio

def print_result(name, result):
    hits, faults, ratio = result
    print(f"{name} -> Hits: {hits}, Faults: {faults}, Hit Ratio:
{ratio:.2f}")
if __name__ == "__main__":
    pages = [2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2]
    capacity = 3
    print_result("FIFO", fifo(pages, capacity))
    print_result("LRU", lru(pages, capacity))
```



This Python program models and compares three page replacement algorithms — FIFO (First-In-First-Out), LRU (Least Recently Used), and LFU (Least Frequently Used) — on a specified sequence of page requests and a fixed memory size. Each algorithm is within its own function that maintains the count of page hits (already in memory) and page faults (not in memory) during the simulation. The `fifo` function replaces the oldest page when memory is full, the `lru` function replaces the least recently used page, and the `lfu` function replaces the least frequently used page when space is needed. All three functions return the hits, faults, and hit ratio after the page sequence has been processed. The `print_result` function formats and prints these results. The primary block runs all three algorithms on a sample page list and prints each of their performance characteristics.