fcfs

```
def fcfs_scheduling(burst_times):
    n = len(burst_times)
    waiting_time = [0] * n
    turnaround_time = [0] * n

    # Calculate waiting time
    for i in range(1, n):
        waiting_time[i] = waiting_time[i - 1] + burst_times[i - 1]

    # Calculate turnaround time
    for i in range(n):
        turnaround_time[i] = waiting_time[i] + burst_times[i]

    # Print process information
    print("Processes    Burst Time    Waiting Time    Turnaround Time")
    for i in range(n):
        print(f"P{i+1}          {burst_times[i]}              {waiting_time[i]}              {turnaround_time[i]}")

    # Calculate average waiting time and turnaround time
    avg_waiting_time = sum(waiting_time) / n
    avg_turnaround_time = sum(turnaround_time) / n

    print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
    print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")

# Example burst times
burst_times = [4, 3, 1, 2, 5]

fcfs_scheduling(burst_times)
```

producer consumer:

```
import random
import time

# Shared buffer and buffer size
buffer = []
BUFFER_SIZE = 5

def produce():
    return random.randint(1, 100)

def producer():
    item = produce()
    if len(buffer) < BUFFER_SIZE:
        buffer.append(item)
        print(f'Produced {item}')
    else:
        print('Buffer full, producer is waiting')
def consumer():
    if buffer:
        item = buffer.pop(0)
        print(f'Consumed {item}')
    else:
        print('Buffer empty, consumer is waiting')
# Simulate producer and consumer
for _ in range(10):
    producer()
    time.sleep(0.5)  # Simulate time delay for producing
    consumer()
    time.sleep(0.5)  # Simulate time delay for consuming
```

bankers algo :

```
def is_safe_state(available, allocation, max_need):
```

```
    num_processes = len(allocation)
    num_resources = len(available)

    # Calculate the need matrix
    need = [[max_need[i][j] - allocation[i][j] for j in range(num_resources)] for i in
range(num_processes)]

    # Initialize work and finish arrays
    work = available[:]
    finish = [False] * num_processes
    safe_sequence = []

    while len(safe_sequence) < num_processes:
        for i in range(num_processes):
            if not finish[i] and all(need[i][j] <= work[j] for j in range(num_resources)):
                work = [work[j] + allocation[i][j] for j in range(num_resources)]
                safe_sequence.append(i)
                finish[i] = True
                break
        else:
            return False, []

    return True, safe_sequence

# Example data
available = [3, 3, 2]
max_need = [
    [7, 5, 3],
    [3, 2, 2],
    [9, 0, 2],
    [2, 2, 2],
    [4, 3, 3]
]
```

```
allocation = [
    [0, 1, 0],
    [2, 0, 0],
    [3, 0, 2],
    [2, 1, 1],
    [0, 0, 2]
]

# Check if the system is in a safe state
is_safe, safe_sequence = is_safe_state(available, allocation, max_need)

if is_safe:
    print("The system is in a safe state.")
    print("Safe sequence:", ' -> '.join(f'P{p}' for p in safe_sequence))
else:
    print("The system is not in a safe state.")
```

first fit:

```
def first_fit(blocks, processes):
    allocation = [-1] * len(processes)

    for i in range(len(processes)):
```

```
        for j in range(len(blocks)):
            if blocks[j] >= processes[i]:
                allocation[i] = j
                blocks[j] -= processes[i]
                break

    return allocation

# Example data
blocks = [100, 500, 200, 300, 600]
processes = [212, 417, 112, 426]

# Run the First Fit algorithm
allocation = first_fit(blocks, processes)

# Print the results
print("Process No.  Process Size  Block No.")
for i in range(len(processes)):
    print(f"{i + 1}           {processes[i]}           {allocation[i] + 1 if allocation[i] != -1 else 'Not Allocated'}")
```

Merge sort:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
```

```
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)

def merge(left, right):
    sorted_arr = []
    while left and right:
        if left[0] < right[0]:
            sorted_arr.append(left.pop(0))
        else:
            sorted_arr.append(right.pop(0))
    sorted_arr.extend(left or right)
    return sorted_arr

# Example usage
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = merge_sort(arr)
print("Sorted array:", sorted_arr)
```

Fifo:

```
def fifo_page_replacement(pages, capacity):
    page_faults = 0
    queue = []

    for page in pages:
        if page not in queue:
```

```
            if len(queue) == capacity:
                queue.pop(0)
            queue.append(page)
            page_faults += 1
        # Print the current state of the queue after each page request
        print(f"Page Request: {page}, Current Queue: {queue}")

    return page_faults

# Example usage
pages = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
capacity = 4

print("\nRunning FIFO Page Replacement Algorithm:")
page_faults = fifo_page_replacement(pages, capacity)
print(f"\nTotal Page Faults: {page_faults}")
```