Set 1: CPU Scheduling Algorithms

```python
# a) FCFS
def fcfs(processes):
    processes.sort(key=lambda x: x['arrival_time'])
    completion_time = 0
    turnaround_time = 0

    for process in processes:
        completion_time += process['burst_time']
        turnaround_time += completion_time - process['arrival_time']

    average_turnaround_time = turnaround_time / len(processes)
    return average_turnaround_time

# b) SJF (Non-Preemptive)
def sjf_non_preemptive(processes):
    processes.sort(key=lambda x: (x['burst_time'], x['arrival_time']))
    completion_time = 0
    turnaround_time = 0

    for process in processes:
        completion_time += process['burst_time']
        turnaround_time += completion_time - process['arrival_time']

    average_turnaround_time = turnaround_time / len(processes)
    return average_turnaround_time

# b) SJF (Preemptive)
def sjf_preemptive(processes):
    processes.sort(key=lambda x: (x['arrival_time'], x['burst_time']))
    completion_time = 0
    turnaround_time = 0
    remaining_burst_time = {process['process_id']: process['burst_time'] for process in processes}

    while any(remaining_burst_time.values()):
        for process in processes:
            if remaining_burst_time[process['process_id']] > 0:
                completion_time += 1
                remaining_burst_time[process['process_id']] -= 1
                if remaining_burst_time[process['process_id']] == 0:
                    turnaround_time += completion_time - process['arrival_time']

    average_turnaround_time = turnaround_time / len(processes)
    return average_turnaround_time
```

```
# Sample Input for CPU Scheduling
cpu_scheduling_processes = [
    {"process_id": 1, "arrival_time": 0, "burst_time": 6},
    {"process_id": 2, "arrival_time": 1, "burst_time": 8},
    {"process_id": 3, "arrival_time": 2, "burst_time": 7},
    {"process_id": 4, "arrival_time": 3, "burst_time": 3},
]

# Sample Output for FCFS, SJF Non-Preemptive, SJF Preemptive
fcfs_result = fcfs(cpu_scheduling_processes)
sjf_non_preemptive_result = sjf_non_preemptive(cpu_scheduling_processes)
sjf_preemptive_result = sjf_preemptive(cpu_scheduling_processes)
print("FCFS Result:", fcfs_result)
print("SJF (Non-Preemptive) Result:", sjf_non_preemptive_result)
print("SJF (Preemptive) Result:", sjf_preemptive_result)
```

Output

FCFS Result: 11.75

SJF (Non-Preemptive) Result: 10.0

SJF (Preemptive) Result: 11.75

Set 2: Priority Scheduling Algorithms

```python
# a) Priority Scheduling (Non-Preemptive)
def priority_non_preemptive(processes):
    processes.sort(key=lambda x: (x['priority'], x['arrival_time']))
    completion_time = 0
    turnaround_time = 0

    for process in processes:
        completion_time += process['burst_time']
        turnaround_time += completion_time - process['arrival_time']

    average_turnaround_time = turnaround_time / len(processes)
    return average_turnaround_time

# b) Priority Scheduling (Preemptive)
def priority_preemptive(processes):
    processes.sort(key=lambda x: (x['arrival_time'], x['priority']))
    completion_time = 0
    turnaround_time = 0
    remaining_burst_time = {process['process_id']: process['burst_time'] for process in
processes}

    while any(remaining_burst_time.values()):
        for process in processes:
            if remaining_burst_time[process['process_id']] > 0:
                completion_time += 1
                remaining_burst_time[process['process_id']] -= 1
                if remaining_burst_time[process['process_id']] == 0:
                    turnaround_time += completion_time - process['arrival_time']

    average_turnaround_time = turnaround_time / len(processes)
    return average_turnaround_time

# Sample Input for Priority Scheduling
priority_processes = [
    {"process_id": 1, "arrival_time": 0, "priority": 3, "burst_time": 6},
    {"process_id": 2, "arrival_time": 1, "priority": 1, "burst_time": 8},
    {"process_id": 3, "arrival_time": 2, "priority": 2, "burst_time": 7},
    {"process_id": 4, "arrival_time": 3, "priority": 4, "burst_time": 3},
]

# Sample Output for Priority Scheduling (Non-Preemptive), Priority Scheduling (Preemptive)
priority_non_preemptive_result = priority_non_preemptive(priority_processes)
priority_preemptive_result = priority_preemptive(priority_processes)
print("Priority Scheduling (Non-Preemptive) Result:", priority_non_preemptive_result)
```

```
print("Priority Scheduling (Preemptive) Result:", priority_preemptive_result)
```

Output

Priority Scheduling (Non-Preemptive) Result: 9.75
Priority Scheduling (Preemptive) Result: 9.75

Set 3: Round Robin Scheduling Algorithm

```python
# 3. Round Robin Scheduling Algorithm
def round_robin(processes, time_quantum):
    remaining_burst_time = {process['process_id']: process['burst_time'] for process in processes}
    completion_time = 0
    turnaround_time = 0

    while any(remaining_burst_time.values()):
        for process in processes:
            if remaining_burst_time[process['process_id']] > 0:
                if remaining_burst_time[process['process_id']] <= time_quantum:
                    completion_time += remaining_burst_time[process['process_id']]
                    turnaround_time += completion_time
                    remaining_burst_time[process['process_id']] = 0
                else:
                    completion_time += time_quantum
                    remaining_burst_time[process['process_id']] -= time_quantum

    average_turnaround_time = turnaround_time / len(processes)
    return average_turnaround_time

# Sample Input for Round Robin Scheduling
rr_processes = [
    {"process_id": 1, "burst_time": 6},
    {"process_id": 2, "burst_time": 8},
    {"process_id": 3, "burst_time": 7},
    {"process_id": 4, "burst_time": 3},
]

rr_time_quantum = 3

# Sample Output for Round Robin Scheduling
rr_result = round_robin(rr_processes, rr_time_quantum)
print("Round Robin Scheduling Result:", rr_result)
```

Output

Round Robin Scheduling Result: 12.0

Set 4: Contiguous Memory Allocation Techniques

```python
# a) First Fit
def first_fit(memory_blocks, processes):
    allocation = [-1] * len(processes)

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

    return allocation

# b) Worst Fit
def worst

_fit(memory_blocks, processes):
    allocation = [-1] * len(processes)

    for i in range(len(processes)):
        index = -1
        for j in range(len(memory_blocks)):
            if memory_blocks[j] >= processes[i]:
                if index == -1 or memory_blocks[j] > memory_blocks[index]:
                    index = j

        if index != -1:
            allocation[i] = index
            memory_blocks[index] -= processes[i]

    return allocation

# c) Best Fit
def best_fit(memory_blocks, processes):
    allocation = [-1] * len(processes)

    for i in range(len(processes)):
        index = -1
        for j in range(len(memory_blocks)):
            if memory_blocks[j] >= processes[i]:
                if index == -1 or memory_blocks[j] < memory_blocks[index]:
                    index = j

        if index != -1:
```

```
            allocation[i] = index
            memory_blocks[index] -= processes[i]

    return allocation

# Sample Input for Contiguous Memory Allocation
memory_blocks = [100, 200, 300, 400, 500]
memory_processes = [212, 417, 112, 426]

# Sample Output for First Fit, Worst Fit, Best Fit
first_fit_result = first_fit(memory_blocks, memory_processes)
worst_fit_result = worst_fit(memory_blocks, memory_processes)
best_fit_result = best_fit(memory_blocks, memory_processes)
print("First Fit Result:", first_fit_result)
print("Worst Fit Result:", worst_fit_result)
print("Best Fit Result:", best_fit_result)
```

Output

First Fit Result: [1, 2, 3, -1]
Worst Fit Result: [3, 2, 0, -1]
Best Fit Result: [3, 2, 0, -1]

Set 5: Page Replacement Policies

```python
# a) FIFO
def fifo(page_frames, page_references):
    page_queue = []
    page_faults = 0

    for page in page_references:
        if page not in page_queue:
            if len(page_queue) < page_frames:
                page_queue.append(page)
            else:
                page_queue.pop(0)
                page_queue.append(page)
            page_faults += 1

    return page_faults

# b) LRU
def lru(page_frames, page_references):
    page_queue = []
    page_faults = 0

    for page in page_references:
        if page not in page_queue:
            if len(page_queue) < page_frames:
                page_queue.append(page)
            else:
                page_queue.pop(0)
                page_queue.append(page)
            page_faults += 1
        else:
            page_queue.remove(page)
            page_queue.append(page)

    return page_faults

# c) Optimal Page Replacement Algorithm
def optimal_page_replacement(page_frames, page_references):
    page_queue = []
    page_faults = 0

    for page in page_references:
        if page not in page_queue:
            if len(page_queue) < page_frames:
                page_queue.append(page)
```

```
        else:
            index = max((i for i, x in enumerate(page_queue) if x in
page_references[page_references.index(page):]), default=-1)
            if index != -1:
                page_queue[index] = page
        page_faults += 1

    return page_faults

# Sample Input for Page Replacement Policies
page_frames = 3
page_reference_sequence = [2, 3, 1, 4, 2, 1, 3, 4]

# Sample Output for FIFO, LRU, Optimal Page Replacement Algorithm
fifo_result = fifo(page_frames, page_reference_sequence)
lru_result = lru(page_frames, page_reference_sequence)
optimal_result = optimal_page_replacement(page_frames, page_reference_sequence)
print("FIFO Result:", fifo_result)
print("LRU Result:", lru_result)
print("Optimal Page Replacement Result:", optimal_result)
```

Output

FIFO Result: 6

LRU Result: 5

Optimal Page Replacement Result: 4

-By Tushar