print("Choose an Algorithm")

print("A. CPU Scheduling")

print("B. Memory Management (Fixed Partition)")

print("C. Disk Scheduling")

algo\_sched = input("Choose a Letter [A-C]: ")

if algo\_sched == "A":

print("CPU Scheduling")

print("Choose a type of CPU Scheduling")

print("1. FCFS")

print("2. SJF")

print("3. SRT")

print("4. PRIORITY")

print("5. RR")

print("6. HRRN")

cpu\_sched = int(input("Choose a number [1-6]: "))

if cpu\_sched == 1:

class Process:

def \_\_init\_\_(self, pid, arrival\_time, burst\_time):

self.pid = pid

self.arrival\_time = arrival\_time

self.burst\_time = burst\_time

self.completion\_time = 0

self.turnaround\_time = 0

self.waiting\_time = 0

self.response\_time = 0

def calculate\_metrics(self, start\_time):

self.response\_time = start\_time - self.arrival\_time if start\_time >= self.arrival\_time else 0

self.completion\_time = start\_time + self.burst\_time

self.turnaround\_time = self.completion\_time - self.arrival\_time

self.waiting\_time = self.turnaround\_time - self.burst\_time

def fcfs\_scheduling(processes):

print("\nFCFS CPU Scheduling\n")

processes.sort(key=lambda p: p.arrival\_time) # Sort by arrival time

time = 0

for process in processes:

if time < process.arrival\_time:

time = process.arrival\_time

process.calculate\_metrics(time)

time += process.burst\_time

# Print table headers with proper alignment

print(f"{'Process':<10}{'Arrival Time':<15}{'Burst Time':<15}{'Completion Time':<20}{'Turnaround Time':<20}{'Waiting Time':<15}{'Response Time':<15}")

for process in processes:

print(f"P{process.pid:<9}{process.arrival\_time:<15}{process.burst\_time:<15}{process.completion\_time:<20}{process.turnaround\_time:<20}{process.waiting\_time:<15}{process.response\_time:<15}")

avg\_turnaround\_time = sum(p.turnaround\_time for p in processes) / len(processes)

avg\_waiting\_time = sum(p.waiting\_time for p in processes) / len(processes)

avg\_response\_time = sum(p.response\_time for p in processes) / len(processes)

print(f"\nAverage Turnaround Time: {avg\_turnaround\_time:.2f}")

print(f"Average Waiting Time: {avg\_waiting\_time:.2f}")

print(f"Average Response Time: {avg\_response\_time:.2f}")

n = int(input("Enter the number of processes: "))

processes = []

for i in range(n):

arrival\_time = int(input(f"Enter arrival time for process P{i+1}: "))

burst\_time = int(input(f"Enter burst time for process P{i+1}: "))

processes.append(Process(i + 1, arrival\_time, burst\_time))

fcfs\_scheduling(processes)

elif cpu\_sched == 2:

# Function to find the waiting time for all processes

def findWaitingTime(processes, n, bt, at, wt):

wt[0] = 0 # Waiting time for the first process is always 0

for i in range(1, n):

wt[i] = bt[i-1] + wt[i-1] # Waiting time is the sum of burst times before the current process

# Function to find the turn around time for all processes

def findTurnAroundTime(processes, n, bt, wt, tat):

for i in range(n):

tat[i] = bt[i] + wt[i] # Turnaround time is burst time + waiting time

# Function to find the response time for all processes

def findResponseTime(wt, at, rt):

for i in range(len(wt)):

rt[i] = wt[i] - at[i] # Response time is the waiting time minus arrival time

# Function to calculate average time

def findAvgTime(processes, n, bt, at):

wt = [0] \* n # Initialize waiting time array

tat = [0] \* n # Initialize turnaround time array

rt = [0] \* n # Initialize response time array

findWaitingTime(processes, n, bt, at, wt) # Calculate waiting times

findTurnAroundTime(processes, n, bt, wt, tat) # Calculate turnaround times

findResponseTime(wt, at, rt) # Calculate response times

total\_wt = sum(wt)

total\_tat = sum(tat)

total\_rt = sum(rt)

# Format and print the table with proper alignment

print(f"{'Process':<10}{'Arrival Time':<15}{'Burst Time':<15}{'Waiting Time':<15}{'Turnaround Time':<20}{'Response Time':<15}")

for i in range(n):

print(f"{processes[i]:<10}{at[i]:<15}{bt[i]:<15}{wt[i]:<15}{tat[i]:<20}{rt[i]:<15}")

print(f"\nAverage waiting time: {total\_wt / n:.2f}")

print(f"Average turnaround time: {total\_tat / n:.2f}")

print(f"Average response time: {total\_rt / n:.2f}")

# Function to implement SJF scheduling

def sjfScheduling(processes, bt, at, n):

# Sort the burst times and process IDs based on burst times

sorted\_processes = sorted(zip(processes, bt, at), key=lambda x: x[1]) # Sort by burst time

sorted\_processes = list(zip(\*sorted\_processes)) # Unzip into separate lists

sorted\_processes[0], sorted\_processes[1], sorted\_processes[2] = list(sorted\_processes[0]), list(sorted\_processes[1]), list(sorted\_processes[2])

# Calculate average waiting time, turnaround time, and response time

findAvgTime(sorted\_processes[0], n, sorted\_processes[1], sorted\_processes[2])

# Driver code

n = int(input("Enter the number of processes: "))

processes = []

burst\_time = []

arrival\_time = []

# Get user input for each process

for i in range(n):

process\_id = int(input(f"Enter Process ID for process {i+1}: "))

processes.append(process\_id)

bt = int(input(f"Enter Burst Time for process {process\_id}: "))

burst\_time.append(bt)

at = int(input(f"Enter Arrival Time for process {process\_id}: "))

arrival\_time.append(at)

# Run the SJF Scheduling algorithm

sjfScheduling(processes, burst\_time, arrival\_time, n)

elif cpu\_sched == 3:

import heapq

# Function to find the waiting time for all processes

def findWaitingTime(processes, n, bt, at, wt):

wt[0] = 0 # Waiting time for the first process is always 0

for i in range(1, n):

wt[i] = bt[i-1] + wt[i-1] # Waiting time is the sum of burst times before the current process

# Function to find the turn around time for all processes

def findTurnAroundTime(processes, n, bt, wt, tat):

for i in range(n):

tat[i] = bt[i] + wt[i] # Turnaround time is burst time + waiting time

# Function to find the response time for all processes

def findResponseTime(wt, at, rt):

for i in range(len(wt)):

rt[i] = wt[i] - at[i] # Response time is the waiting time minus arrival time

# Function to calculate average time

def findAvgTime(processes, n, bt, at):

wt = [0] \* n # Initialize waiting time array

tat = [0] \* n # Initialize turnaround time array

rt = [0] \* n # Initialize response time array

findWaitingTime(processes, n, bt, at, wt) # Calculate waiting times

findTurnAroundTime(processes, n, bt, wt, tat) # Calculate turnaround times

findResponseTime(wt, at, rt) # Calculate response times

total\_wt = sum(wt)

total\_tat = sum(tat)

total\_rt = sum(rt)

# Format and print the table with proper alignment

print(f"{'Process':<10}{'Arrival Time':<15}{'Burst Time':<15}{'Waiting Time':<15}{'Turnaround Time':<20}{'Response Time':<15}")

for i in range(n):

print(f"{processes[i]:<10}{at[i]:<15}{bt[i]:<15}{wt[i]:<15}{tat[i]:<20}{rt[i]:<15}")

print(f"\nAverage waiting time: {total\_wt / n:.2f}")

print(f"Average turnaround time: {total\_tat / n:.2f}")

print(f"Average response time: {total\_rt / n:.2f}")

# Function to implement SRT scheduling

def srtScheduling(processes, bt, at, n):

# Create a list to store remaining burst times for each process

remaining\_bt = bt[:]

# Initialize variables

current\_time = 0

completed = 0

wt = [0] \* n

tat = [0] \* n

rt = [0] \* n

start\_time = [-1] \* n

# Create a priority queue (min-heap) to manage processes based on remaining burst time

pq = []

# Add processes to the priority queue (min-heap) based on their arrival time

while completed < n:

# Add processes that have arrived at the current time to the priority queue

for i in range(n):

if at[i] <= current\_time and remaining\_bt[i] > 0 and start\_time[i] == -1:

heapq.heappush(pq, (remaining\_bt[i], i))

start\_time[i] = current\_time # Mark the start time for the process

if pq:

# Pop the process with the smallest remaining burst time

bt\_remaining, idx = heapq.heappop(pq)

# Process execution (decrease remaining burst time)

remaining\_bt[idx] -= 1

current\_time += 1

# If the process is completed

if remaining\_bt[idx] == 0:

completed += 1

tat[idx] = current\_time - at[idx] # Turnaround time is current time - arrival time

wt[idx] = tat[idx] - bt[idx] # Waiting time is turnaround time - burst time

rt[idx] = start\_time[idx] - at[idx] # Response time is first start time - arrival time

# If the process has remaining burst time, push it back to the queue

if remaining\_bt[idx] > 0:

heapq.heappush(pq, (remaining\_bt[idx], idx))

else:

current\_time += 1 # If no process is ready to execute, increment time

# Calculate and print the results

findAvgTime(processes, n, bt, at)

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

# Get user input for number of processes

n = int(input("Enter the number of processes: "))

processes = []

burst\_time = []

arrival\_time = []

# Get user input for each process

for i in range(n):

process\_id = int(input(f"Enter Process ID for process {i+1}: "))

processes.append(process\_id)

bt = int(input(f"Enter Burst Time for process {process\_id}: "))

burst\_time.append(bt)

at = int(input(f"Enter Arrival Time for process {process\_id}: "))

arrival\_time.append(at)

# Run the SRT Scheduling algorithm

srtScheduling(processes, burst\_time, arrival\_time, n)

elif cpu\_sched == 4:

# Function to calculate Completion Time, Turnaround Time, Waiting Time, and Response Time

def priority\_scheduling(arrival\_time, burst\_time, priority):

n = len(arrival\_time)

# Initializing required lists

completion\_time = [0] \* n

turnaround\_time = [0] \* n

waiting\_time = [0] \* n

response\_time = [0] \* n

remaining\_burst\_time = burst\_time[:]

# Creating a list of processes with arrival, burst time, and priority

processes = []

for i in range(n):

processes.append([i, arrival\_time[i], burst\_time[i], priority[i]])

# Sorting processes based on arrival time

processes.sort(key=lambda x: x[1])

# Start the first process at the arrival time

current\_time = 0

completed\_processes = 0

while completed\_processes < n:

# Find the process with the highest priority that is ready to run

ready\_processes = [p for p in processes if p[1] <= current\_time and remaining\_burst\_time[p[0]] > 0]

if ready\_processes:

# Sort by priority and pick the process with the highest priority (lowest priority number)

ready\_processes.sort(key=lambda x: x[3])

process = ready\_processes[0]

index = process[0]

remaining\_burst\_time[index] -= 1

current\_time += 1

# Calculate response time if it's the first time the process is being executed

if remaining\_burst\_time[index] == burst\_time[index] - 1:

response\_time[index] = current\_time - arrival\_time[index]

# If the process finishes, update completion time and calculate turnaround and waiting time

if remaining\_burst\_time[index] == 0:

completion\_time[index] = current\_time

turnaround\_time[index] = completion\_time[index] - arrival\_time[index]

waiting\_time[index] = turnaround\_time[index] - burst\_time[index]

completed\_processes += 1

else:

current\_time += 1

return completion\_time, turnaround\_time, waiting\_time, response\_time

# Function to calculate average times

def calculate\_averages(turnaround\_time, waiting\_time, response\_time):

n = len(turnaround\_time)

avg\_turnaround\_time = sum(turnaround\_time) / n

avg\_waiting\_time = sum(waiting\_time) / n

avg\_response\_time = sum(response\_time) / n

return avg\_turnaround\_time, avg\_waiting\_time, avg\_response\_time

# User input for arrival time, burst time, and priority

n = int(input("Enter the number of processes: "))

arrival\_time = []

burst\_time = []

priority = []

# Collecting input for each process

for i in range(n):

arrival = int(input(f"Enter arrival time for process {i}: "))

burst = int(input(f"Enter burst time for process {i}: "))

pri = int(input(f"Enter priority for process {i} (lower number indicates higher priority): "))

arrival\_time.append(arrival)

burst\_time.append(burst)

priority.append(pri)

# Run the priority scheduling algorithm

completion\_time, turnaround\_time, waiting\_time, response\_time = priority\_scheduling(arrival\_time, burst\_time, priority)

# Calculate averages

avg\_turnaround\_time, avg\_waiting\_time, avg\_response\_time = calculate\_averages(turnaround\_time, waiting\_time, response\_time)

# Print the table with the results

print("\nProcess | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time | Response Time")

for i in range(n):

print(f"P{i} | {arrival\_time[i]} | {burst\_time[i]} | {completion\_time[i]} | {turnaround\_time[i]} | {waiting\_time[i]} | {response\_time[i]}")

# Print average times

print(f"\nAverage Turnaround Time: {avg\_turnaround\_time:.2f}")

print(f"Average Waiting Time: {avg\_waiting\_time:.2f}")

print(f"Average Response Time: {avg\_response\_time:.2f}")

elif cpu\_sched == 5:

from collections import deque

# Get number of processes

n = int(input("Enter the number of processes: "))

# Initialize an empty list for processes

processes = []

# Get arrival time and burst time for each process

for i in range(n):

pid = i + 1

at = int(input(f"Enter arrival time for Process {pid}: "))

bt = int(input(f"Enter burst time for Process {pid}: "))

processes.append([pid, at, bt])

# Get time quantum (time slice)

time\_quantum = int(input("Enter the time quantum: "))

# Initialize required lists

completion\_time = [-1] \* n

turnaround\_time = [0] \* n

waiting\_time = [0] \* n

response\_time = [-1] \* n

# Sort processes by arrival time

processes.sort(key=lambda x: x[1])

pid\_to\_index = {processes[i][0]: i for i in range(n)} # Mapping PID to original index

# Ready queue and remaining burst times

queue = deque()

remaining\_burst = {p[0]: p[2] for p in processes} # {PID: Remaining Burst Time}

current\_time = 0

index = 0 # To track arrival of new processes

# If first process arrives later, start from that time

if processes[0][1] > 0:

current\_time = processes[0][1]

# Enqueue first process

while index < n and processes[index][1] <= current\_time:

queue.append(processes[index][0])

index += 1

# Gantt Chart Data

gantt\_chart = []

time\_chart = [current\_time]

# Process queue

while queue:

pid = queue.popleft()

p\_index = pid\_to\_index[pid] # Get correct index after sorting

# If first time running, set response time

if response\_time[p\_index] == -1:

response\_time[p\_index] = current\_time - processes[p\_index][1]

# Execute process

execution\_time = min(time\_quantum, remaining\_burst[pid])

current\_time += execution\_time

remaining\_burst[pid] -= execution\_time

# Gantt Chart Updates

gantt\_chart.append(f"P{pid}")

time\_chart.append(current\_time)

# Check for newly arrived processes

while index < n and processes[index][1] <= current\_time:

queue.append(processes[index][0])

index += 1

# If process is not finished, re-add it to queue

if remaining\_burst[pid] > 0:

queue.append(pid)

else:

completion\_time[p\_index] = current\_time

turnaround\_time[p\_index] = completion\_time[p\_index] - processes[p\_index][1]

waiting\_time[p\_index] = turnaround\_time[p\_index] - processes[p\_index][2]

# Handle case when queue is empty and there are still processes arriving

if not queue and index < n:

current\_time = processes[index][1] # Jump forward in time to next process arrival

queue.append(processes[index][0]) # Enqueue the next process

index += 1

# Calculate averages

avg\_tat = sum(turnaround\_time) / n

avg\_wt = sum(waiting\_time) / n

avg\_rt = sum(response\_time) / n

# Display results

print("\nPID AT BT CT TAT WT RT")

for i in range(n):

print(f"{processes[i][0]:3} {processes[i][1]:2} {processes[i][2]:2} {completion\_time[i]:2} {turnaround\_time[i]:3} {waiting\_time[i]:2} {response\_time[i]:2}")

print(f"\nAverage Turnaround Time: {avg\_tat:.2f}")

print(f"Average Waiting Time: {avg\_wt:.2f}")

print(f"Average Response Time: {avg\_rt:.2f}")

# Display Gantt Chart

print("\nGantt Chart:")

print(" | ".join(gantt\_chart))

print(" ".join(str(t) for t in time\_chart))

elif cpu\_sched == 6:

# HRRN Scheduling Function

def hrrn\_scheduling(n, processes):

# Sort processes by arrival time

processes.sort(key=lambda x: x[1])

# Initialize variables

current\_time = 0

completed = 0

gantt\_chart = []

completion\_time = [-1] \* n

turnaround\_time = [0] \* n

waiting\_time = [0] \* n

response\_time = [-1] \* n

executed\_processes = []

initial\_rr\_values = {}

while completed < n:

# Find processes that have arrived but are not yet completed

available\_processes = [p for p in processes if p[1] <= current\_time and completion\_time[p[0] - 1] == -1]

if not available\_processes:

current\_time += 1 # Move time forward if no process is available

continue

# Calculate response ratios

response\_ratios = []

for p in available\_processes:

pid, at, bt = p

waiting\_time\_now = current\_time - at

response\_ratio = (waiting\_time\_now + bt) / bt # (WT + BT) / BT

response\_ratios.append((response\_ratio, p))

# Store initial response ratio before execution

if pid not in initial\_rr\_values:

initial\_rr\_values[pid] = response\_ratio

# Select process with highest response ratio

selected\_process = max(response\_ratios, key=lambda x: x[0])[1]

pid, at, bt = selected\_process

# Execute process

if response\_time[pid - 1] == -1: # First execution

response\_time[pid - 1] = current\_time - at

gantt\_chart.append(f"P{pid}")

executed\_processes.append((pid, current\_time))

current\_time += bt

completion\_time[pid - 1] = current\_time

turnaround\_time[pid - 1] = completion\_time[pid - 1] - at

waiting\_time[pid - 1] = turnaround\_time[pid - 1] - bt

completed += 1

# Calculate averages

avg\_tat = sum(turnaround\_time) / n

avg\_wt = sum(waiting\_time) / n

avg\_rt = sum(response\_time) / n

# Display results

print("\nPID AT BT CT TAT WT RT Initial RR")

for i in range(n):

initial\_rr = initial\_rr\_values[processes[i][0]] # Fetch stored initial RR

print(f"{processes[i][0]:3} {processes[i][1]:2} {processes[i][2]:2} {completion\_time[i]:2} {turnaround\_time[i]:3} {waiting\_time[i]:2} {response\_time[i]:2} {initial\_rr:.2f}")

print(f"\nAverage Turnaround Time: {avg\_tat:.2f}")

print(f"Average Waiting Time: {avg\_wt:.2f}")

print(f"Average Response Time: {avg\_rt:.2f}")

# Display Gantt Chart

print("\nGantt Chart:")

print(" | ".join(gantt\_chart))

print(" ".join(str(t[1]) for t in executed\_processes) + f" {current\_time}")

# Get input from user

try:

n = int(input("Enter number of processes: "))

processes = []

print("Enter Arrival Time and Burst Time for each process:")

for i in range(n):

at, bt = map(int, input(f"Process {i+1}: ").split())

processes.append([i+1, at, bt])

# Run HRRN scheduling

hrrn\_scheduling(n, processes)

except Exception as e:

print("Error:", e)

elif algo\_sched == "B":

print("Memory Management (Fixed Partition)")

print("1. First Fit")

print("2. Best Fit")

print("3. Next Fit")

print("4. Worst Fit")

mem\_sched = int(input("Choose a number [1-4]: "))

if mem\_sched == 1:

def first\_fit\_memory\_management(partitions, processes):

allocation = [-1] \* len(processes) # Initialize allocation list (-1 means not allocated)

# Traverse all processes and allocate memory

for i in range(len(processes)):

for j in range(len(partitions)):

if partitions[j] >= processes[i]: # Check if partition is large enough for the process

allocation[i] = j # Allocate the process to the partition

partitions[j] -= processes[i] # Reduce the size of the partition

break # Move to the next process after allocation

# Print the result

print(f"\n{'Process':<10}{'Memory Required':<20}{'Partition Allocated':<20}")

for i in range(len(processes)):

if allocation[i] != -1:

print(f"P{i + 1:<9}{processes[i]:<20}{'Partition ' + str(allocation[i] + 1)}")

else:

print(f"P{i + 1:<9}{processes[i]:<20}{'Not Allocated'}")

if \_\_name\_\_ == "\_\_main\_\_":

# User inputs the partition sizes

num\_partitions = int(input("Enter the number of memory partitions: "))

partitions = []

for i in range(num\_partitions):

partition\_size = int(input(f"Enter size of partition {i + 1}: "))

partitions.append(partition\_size)

# User inputs the number of processes and their memory requirements

num\_processes = int(input("\nEnter the number of processes: "))

processes = []

for i in range(num\_processes):

process\_memory = int(input(f"Enter memory required by process P{i + 1}: "))

processes.append(process\_memory)

# Call the First Fit memory allocation function

first\_fit\_memory\_management(partitions, processes)

elif mem\_sched == 2:

class BestFitMemoryManagement:

def \_\_init\_\_(self, partition\_sizes):

self.partitions = partition\_sizes # List of partition sizes

self.partitions\_status = [None] \* len(partition\_sizes) # None means empty partition

def best\_fit(self, process\_size):

# Find the best fit partition for the given process

best\_fit\_index = -1

min\_diff = float('inf')

for i in range(len(self.partitions)):

if self.partitions\_status[i] is None and self.partitions[i] >= process\_size:

diff = self.partitions[i] - process\_size

if diff < min\_diff:

min\_diff = diff

best\_fit\_index = i

if best\_fit\_index != -1:

self.partitions\_status[best\_fit\_index] = process\_size

print(f"Process of size {process\_size} allocated to partition {best\_fit\_index} (Size {self.partitions[best\_fit\_index]})")

else:

print(f"Process of size {process\_size} could not be allocated.")

def deallocate(self, partition\_index):

# Deallocate the partition

if self.partitions\_status[partition\_index] is not None:

print(f"Partition {partition\_index} deallocated (Process size: {self.partitions\_status[partition\_index]})")

self.partitions\_status[partition\_index] = None

else:

print(f"Partition {partition\_index} is already empty.")

def display\_partitions(self):

print("\nCurrent Partition Status:")

for i in range(len(self.partitions)):

if self.partitions\_status[i] is None:

print(f"Partition {i}: Empty (Size {self.partitions[i]})")

else:

print(f"Partition {i}: Occupied (Process Size {self.partitions\_status[i]})")

# Input from the user

def get\_partition\_sizes():

partitions = []

num\_partitions = int(input("Enter the number of partitions: "))

for i in range(num\_partitions):

size = int(input(f"Enter the size of partition {i + 1}: "))

partitions.append(size)

return partitions

def get\_process\_sizes():

processes = []

num\_processes = int(input("Enter the number of processes: "))

for i in range(num\_processes):

size = int(input(f"Enter the size of process {i + 1}: "))

processes.append(size)

return processes

# Main function

def main():

# Get partition and process sizes from user

partition\_sizes = get\_partition\_sizes()

memory\_manager = BestFitMemoryManagement(partition\_sizes)

# Allocate processes

processes = get\_process\_sizes()

for process in processes:

memory\_manager.best\_fit(process)

memory\_manager.display\_partitions()

# Deallocate a partition if needed

deallocate\_choice = input("Do you want to deallocate any partition? (yes/no): ")

if deallocate\_choice.lower() == 'yes':

partition\_index = int(input("Enter the partition index to deallocate: "))

memory\_manager.deallocate(partition\_index)

memory\_manager.display\_partitions()

if \_\_name\_\_ == "\_\_main\_\_":

main()

elif mem\_sched == 3:

def next\_fit\_fixed\_partition(partition\_sizes, job\_sizes):

allocation = [-1] \* len(job\_sizes) # Stores partition index for each job (-1 means not allocated)

last\_allocated = 0 # Start from last allocated partition

for i in range(len(job\_sizes)): # Iterate through each job

allocated = False

for j in range(len(partition\_sizes)):

index = (last\_allocated + j) % len(partition\_sizes) # Circular search (Next Fit)

if partition\_sizes[index] >= job\_sizes[i]: # Check if job fits in the partition

allocation[i] = index # Allocate job to partition

partition\_sizes[index] -= job\_sizes[i] # Reduce available space in partition

last\_allocated = index # Update last allocated partition

allocated = True

break # Move to next job

return allocation

# Get user input for total memory and OS size

total\_memory = int(input("Enter total memory size: "))

os\_size = int(input("Enter OS size: "))

# Calculate available memory

available\_memory = total\_memory - os\_size

print(f"Available memory for partitions: {available\_memory}\n")

# Get user input for memory partitions

num\_partitions = int(input("Enter number of memory partitions: "))

partition\_sizes = list(map(int, input(f"Enter {num\_partitions} partition sizes separated by space: ").split()))

# Validate if partition sizes exceed available memory

if sum(partition\_sizes) > available\_memory:

print("\nError: Total partition size exceeds available memory!")

exit()

# Get user input for jobs

num\_jobs = int(input("\nEnter number of jobs: "))

job\_sizes = list(map(int, input(f"Enter {num\_jobs} job sizes separated by space: ").split()))

# Perform Next Fit Fixed Partition Allocation

allocation = next\_fit\_fixed\_partition(partition\_sizes.copy(), job\_sizes)

# Display results in a table format

print("\nJob No. Job Size Partition No.")

for i in range(num\_jobs):

if allocation[i] != -1:

print(f"{i+1:7} {job\_sizes[i]:8} {allocation[i]+1:10}") # Partition numbers start from 1

else:

print(f"{i+1:7} {job\_sizes[i]:8} Not Allocated")

# Text-based memory allocation chart

print("\nMemory Allocation Chart:")

for i in range(num\_jobs):

if allocation[i] != -1:

print(f"Job {i+1} (Size {job\_sizes[i]}) -> Partition {allocation[i]+1}")

else:

print(f"Job {i+1} (Size {job\_sizes[i]}) -> Not Allocated")

elif mem\_sched == 4:

class WorstFitMemoryManagement:

def \_\_init\_\_(self, partition\_sizes):

self.partitions = partition\_sizes # List of partition sizes

self.partitions\_status = [None] \* len(partition\_sizes) # None means empty partition

def worst\_fit(self, process\_size):

# Find the worst fit partition for the given process

worst\_fit\_index = -1

max\_diff = -1

for i in range(len(self.partitions)):

if self.partitions\_status[i] is None and self.partitions[i] >= process\_size:

diff = self.partitions[i] - process\_size

if diff > max\_diff:

max\_diff = diff

worst\_fit\_index = i

if worst\_fit\_index != -1:

self.partitions\_status[worst\_fit\_index] = process\_size

print(f"Process of size {process\_size} allocated to partition {worst\_fit\_index} (Size {self.partitions[worst\_fit\_index]})")

else:

print(f"Process of size {process\_size} could not be allocated.")

def deallocate(self, partition\_index):

# Deallocate the partition

if self.partitions\_status[partition\_index] is not None:

print(f"Partition {partition\_index} deallocated (Process size: {self.partitions\_status[partition\_index]})")

self.partitions\_status[partition\_index] = None

else:

print(f"Partition {partition\_index} is already empty.")

def display\_partitions(self):

print("\nCurrent Partition Status:")

for i in range(len(self.partitions)):

if self.partitions\_status[i] is None:

print(f"Partition {i}: Empty (Size {self.partitions[i]})")

else:

print(f"Partition {i}: Occupied (Process Size {self.partitions\_status[i]})")

# Input from the user

def get\_partition\_sizes():

partitions = []

num\_partitions = int(input("Enter the number of partitions: "))

for i in range(num\_partitions):

size = int(input(f"Enter the size of partition {i + 1}: "))

partitions.append(size)

return partitions

def get\_process\_sizes():

processes = []

num\_processes = int(input("Enter the number of processes: "))

for i in range(num\_processes):

size = int(input(f"Enter the size of process {i + 1}: "))

processes.append(size)

return processes

# Main function

def main():

# Get partition and process sizes from user

partition\_sizes = get\_partition\_sizes()

memory\_manager = WorstFitMemoryManagement(partition\_sizes)

# Allocate processes

processes = get\_process\_sizes()

for process in processes:

memory\_manager.worst\_fit(process)

memory\_manager.display\_partitions()

# Deallocate a partition if needed

deallocate\_choice = input("Do you want to deallocate any partition? (yes/no): ")

if deallocate\_choice.lower() == 'yes':

partition\_index = int(input("Enter the partition index to deallocate: "))

memory\_manager.deallocate(partition\_index)

memory\_manager.display\_partitions()

if \_\_name\_\_ == "\_\_main\_\_":

main()

elif algo\_sched == "C":

print("Disk Scheduling")

print("1. FCFS")

print("2. SSTF")

print("3. SCAN")

print("4. C-SCAN")

print("5. LOOK")

print("6. C-LOOK")

disk\_sched = int(input("Choose a number[1-6]"))

if disk\_sched == 1:

import matplotlib.pyplot as plt

def fcfs(requests, head):

"""

First-Come, First-Served (FCFS) disk scheduling algorithm.

Parameters:

requests (list): List of disk requests.

head (int): Initial position of the disk head.

Returns:

tuple: A tuple containing the seek sequence and total seek count.

"""

seek\_sequence = []

seek\_count = 0

current\_position = head

for request in requests:

seek\_sequence.append(request)

seek\_count += abs(current\_position - request) # Calculate seek time

current\_position = request # Move head to the requested position

return seek\_sequence, seek\_count

# User input

requests = list(map(int, input("Enter the disk requests (comma-separated): ").split(',')))

head = int(input("Enter the initial head position: "))

# Compute FCFS disk scheduling

sequence, count = fcfs(requests, head)

# Display results

print(f"Seek Sequence: {sequence}")

print(f"Total Seek Count: {count}")

# Plotting the seek sequence

positions = [head] + sequence # Include the initial head position in the plot

plt.plot(range(len(positions)), positions, marker='o', color='b', linestyle='-')

plt.title('Disk Scheduling - FCFS Algorithm')

plt.xlabel('Sequence Step')

plt.ylabel('Disk Position')

plt.grid(True)

plt.show()

elif disk\_sched == 2:

import matplotlib.pyplot as plt

def sstf(requests, head):

"""

Shortest Seek Time First (SSTF) disk scheduling algorithm.

Parameters:

requests (list): List of disk requests.

head (int): Initial position of the disk head.

Returns:

tuple: A tuple containing the seek sequence and total seek count.

"""

seek\_sequence = []

seek\_count = 0

current\_position = head

requests = requests.copy() # Copy to avoid modifying the original list

while requests:

# Find the closest request to the current head position

closest\_request = min(requests, key=lambda x: abs(current\_position - x))

seek\_sequence.append(closest\_request)

seek\_count += abs(current\_position - closest\_request) # Calculate seek time

current\_position = closest\_request # Move head to the closest request

requests.remove(closest\_request) # Remove the processed request

return seek\_sequence, seek\_count

requests = list(map(int, input("Enter the disk requests (comma-separated): ").split(',')))

head = int(input("Enter the initial head position: "))

# Compute FCFS disk scheduling

sequence, count = sstf(requests, head)

# Display results

print(f"Seek Sequence: {sequence}")

print(f"Total Seek Count: {count}")

# Plotting the seek sequence

positions = [head] + sequence # Include the initial head position in the plot

plt.plot(range(len(positions)), positions, marker='o', color='b', linestyle='-')

plt.title('Disk Scheduling - SSTF Algorithm')

plt.xlabel('Sequence Step')

plt.ylabel('Disk Position')

plt.grid(True)

plt.show()

elif disk\_sched == 3:

import matplotlib.pyplot as plt

def scan(requests, head, direction):

"""

SCAN disk scheduling algorithm.

Parameters:

requests (list): List of disk requests.

head (int): Initial position of the disk head.

direction (str): Direction of head movement ('left' or 'right').

Returns:

tuple: A tuple containing the seek sequence and total seek count.

"""

seek\_sequence = []

seek\_count = 0

current\_position = head

requests = sorted(requests) # Sort requests for processing

# Determine the order of processing based on the direction

if direction == 'left':

requests = [r for r in requests if r < head][::-1] + [0] + [r for r in requests if r >= head]

else:

requests = [r for r in requests if r >= head] + [199] + [r for r in requests if r < head][::-1]

for request in requests:

seek\_sequence.append(request)

seek\_count += abs(current\_position - request) # Calculate seek time

current\_position = request # Move head to the requested position

return seek\_sequence, seek\_count

requests = list(map(int, input("Enter the disk requests (comma-separated): ").split(',')))

head = int(input("Enter the initial head position: "))

direction = input("Enter direction (left/right): ").strip().lower()

# Compute FCFS disk scheduling

sequence, count = scan(requests, head, direction)

# Display results

print(f"Seek Sequence: {sequence}")

print(f"Total Seek Count: {count}")

# Plotting the seek sequence

positions = [head] + sequence # Include the initial head position in the plot

plt.plot(range(len(positions)), positions, marker='o', color='b', linestyle='-')

plt.title('Disk Scheduling - SCAN Algorithm')

plt.xlabel('Sequence Step')

plt.ylabel('Disk Position')

plt.grid(True)

plt.show()

elif disk\_sched == 4:

import matplotlib.pyplot as plt

def cscan(requests, head, direction):

"""

Circular SCAN (C-SCAN) disk scheduling algorithm.

Parameters:

requests (list): List of disk requests.

head (int): Initial position of the disk head.

direction (str): Direction of head movement ('left' or 'right').

Returns:

tuple: A tuple containing the seek sequence and total seek count.

"""

seek\_sequence = []

seek\_count = 0

current\_position = head

requests = sorted(requests) # Sort requests for processing

# Determine the order of processing based on the direction

if direction == 'left':

requests = [r for r in requests if r < head][::-1] + [0] + [r for r in requests if r >= head]

else:

requests = [r for r in requests if r >= head] + [199] + [r for r in requests if r < head]

for request in requests:

seek\_sequence.append(request)

seek\_count += abs(current\_position - request) # Calculate seek time

current\_position = request # Move head to the requested position

return seek\_sequence, seek\_count

requests = list(map(int, input("Enter the disk requests (comma-separated): ").split(',')))

head = int(input("Enter the initial head position: "))

direction = input("Enter direction (left/right): ").strip().lower()

# Compute FCFS disk scheduling

sequence, count = cscan(requests, head, direction)

# Display results

print(f"Seek Sequence: {sequence}")

print(f"Total Seek Count: {count}")

# Plotting the seek sequence

positions = [head] + sequence # Include the initial head position in the plot

plt.plot(range(len(positions)), positions, marker='o', color='b', linestyle='-')

plt.title('Disk Scheduling - CSCAN Algorithm')

plt.xlabel('Sequence Step')

plt.ylabel('Disk Position')

plt.grid(True)

plt.show()

elif disk\_sched == 5:

import matplotlib.pyplot as plt

def look(requests, head, direction):

"""

LOOK disk scheduling algorithm.

Parameters:

requests (list): List of disk requests.

head (int): Initial position of the disk head.

direction (str): Direction of head movement ('left' or 'right').

Returns:

tuple: A tuple containing the seek sequence and total seek count.

"""

seek\_sequence = []

seek\_count = 0

current\_position = head

requests = sorted(requests) # Sort requests for processing

# Determine the order of processing based on the direction

if direction == 'left':

requests = [r for r in requests if r < head][::-1] + [r for r in requests if r >= head]

else:

requests = [r for r in requests if r >= head] + [r for r in requests if r < head][::-1]

for request in requests:

seek\_sequence.append(request)

seek\_count += abs(current\_position - request) # Calculate seek time

current\_position = request # Move head to the requested position

return seek\_sequence, seek\_count

requests = list(map(int, input("Enter the disk requests (comma-separated): ").split(',')))

head = int(input("Enter the initial head position: "))

direction = input("Enter direction (left/right): ").strip().lower()

# Compute FCFS disk scheduling

sequence, count = look(requests, head, direction)

# Display results

print(f"Seek Sequence: {sequence}")

print(f"Total Seek Count: {count}")

# Plotting the seek sequence

positions = [head] + sequence # Include the initial head position in the plot

plt.plot(range(len(positions)), positions, marker='o', color='b', linestyle='-')

plt.title('Disk Scheduling - LOOK Algorithm')

plt.xlabel('Sequence Step')

plt.ylabel('Disk Position')

plt.grid(True)

plt.show()

elif disk\_sched == 6:

import matplotlib.pyplot as plt

def clook(requests, head, direction):

"""

C-LOOK disk scheduling algorithm.

Parameters:

requests (list): List of disk requests.

head (int): Initial position of the disk head.

direction (str): Direction of head movement ('left' or 'right').

Returns:

tuple: A tuple containing the seek sequence and total seek count.

"""

seek\_sequence = []

seek\_count = 0

current\_position = head

requests = sorted(requests) # Sort requests for processing

# Determine the order of processing based on the direction

if direction == 'left':

requests = [r for r in requests if r < head][::-1] + [r for r in requests if r >= head]

else:

requests = [r for r in requests if r >= head] + [r for r in requests if r < head]

for request in requests:

seek\_sequence.append(request)

seek\_count += abs(current\_position - request) # Calculate seek time

current\_position = request # Move head to the requested position

return seek\_sequence, seek\_count

requests = list(map(int, input("Enter the disk requests (comma-separated): ").split(',')))

head = int(input("Enter the initial head position: "))

direction = input("Enter direction (left/right): ").strip().lower()

# Compute FCFS disk scheduling

sequence, count = clook(requests, head, direction)

# Display results

print(f"Seek Sequence: {sequence}")

print(f"Total Seek Count: {count}")

# Plotting the seek sequence

positions = [head] + sequence # Include the initial head position in the plot

plt.plot(range(len(positions)), positions, marker='o', color='b', linestyle='-')

plt.title('Disk Scheduling - CLOOK Algorithm')

plt.xlabel('Sequence Step')

plt.ylabel('Disk Position')

plt.grid(True)

plt.show()