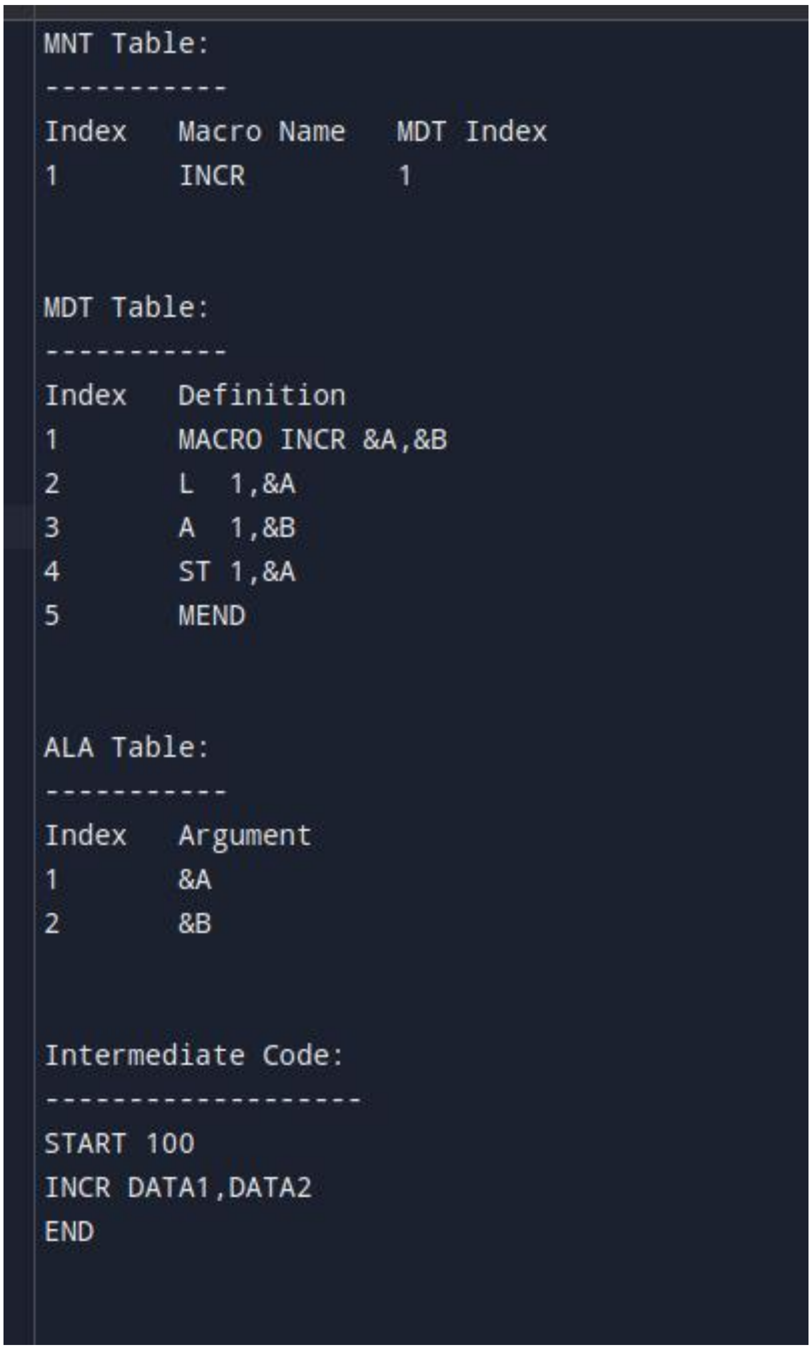
```
# ----- PASS 2: MACRO PROCESSOR -----
# Macro Name Table (MNT)
MNT = {
 "INCR": 1
}
# Macro Definition Table (MDT)
MDT = {
 1: "L 1,&A",
 2: "A 1,&B",
 3: "ST 1,&A",
 4: "MEND"
}
# Argument List Array (ALA)
ALA = ["&A", "&B"]
# Source code containing macro call
source_code = [
 "START",
 "INCR AREG, FIVE",
 "END"
]
print("PASS 2 OUTPUT:\n")
expanded_code = []
for line in source_code:
 parts = line.split()
 if parts[0] in MNT:
 # Macro found
 macro_name = parts[0]
 arguments = parts[1].split(",")

 # Replace formal args in ALA with actual args
 for i in range(MNT[macro_name], len(MDT) + 1):
```

```
 if MDT[i] == "MEND":
 break
 expanded_line = MDT[i]
 for idx, arg in enumerate(ALA):
 expanded_line = expanded_line.replace(arg, arguments[idx])
 expanded_code.append(expanded_line)
 else:
 expanded_code.append(line)
# ----- Final Output -----
for line in expanded_code:
 print(line)
# expected outcome
```

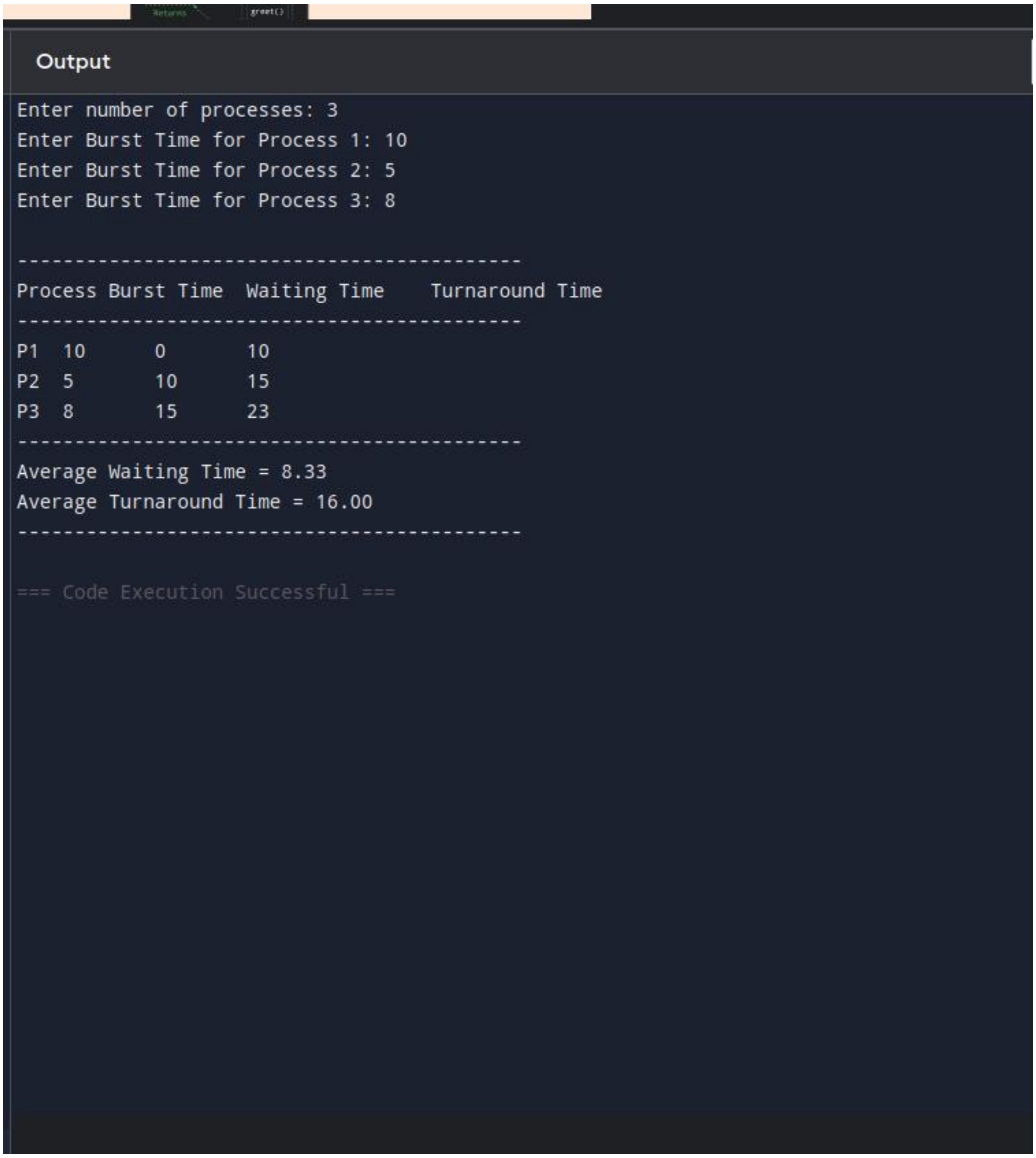
```
PASS 2 OUTPUT:
START
L 1,AREG
A 1,FIVE
ST 1,AREG
END
```



```
MNT Table:
-----------
Index   Macro Name   MDT Index
1       INCR         1


MDT Table:
-----------
Index   Definition
1       MACRO INCR &A,&B
2       L  1,&A
3       A  1,&B
4       ST 1,&A
5       MEND


ALA Table:
-----------
Index   Argument
1       &A
2       &B


Intermediate Code:
------------------
START 100
INCR DATA1,DATA2
END
```

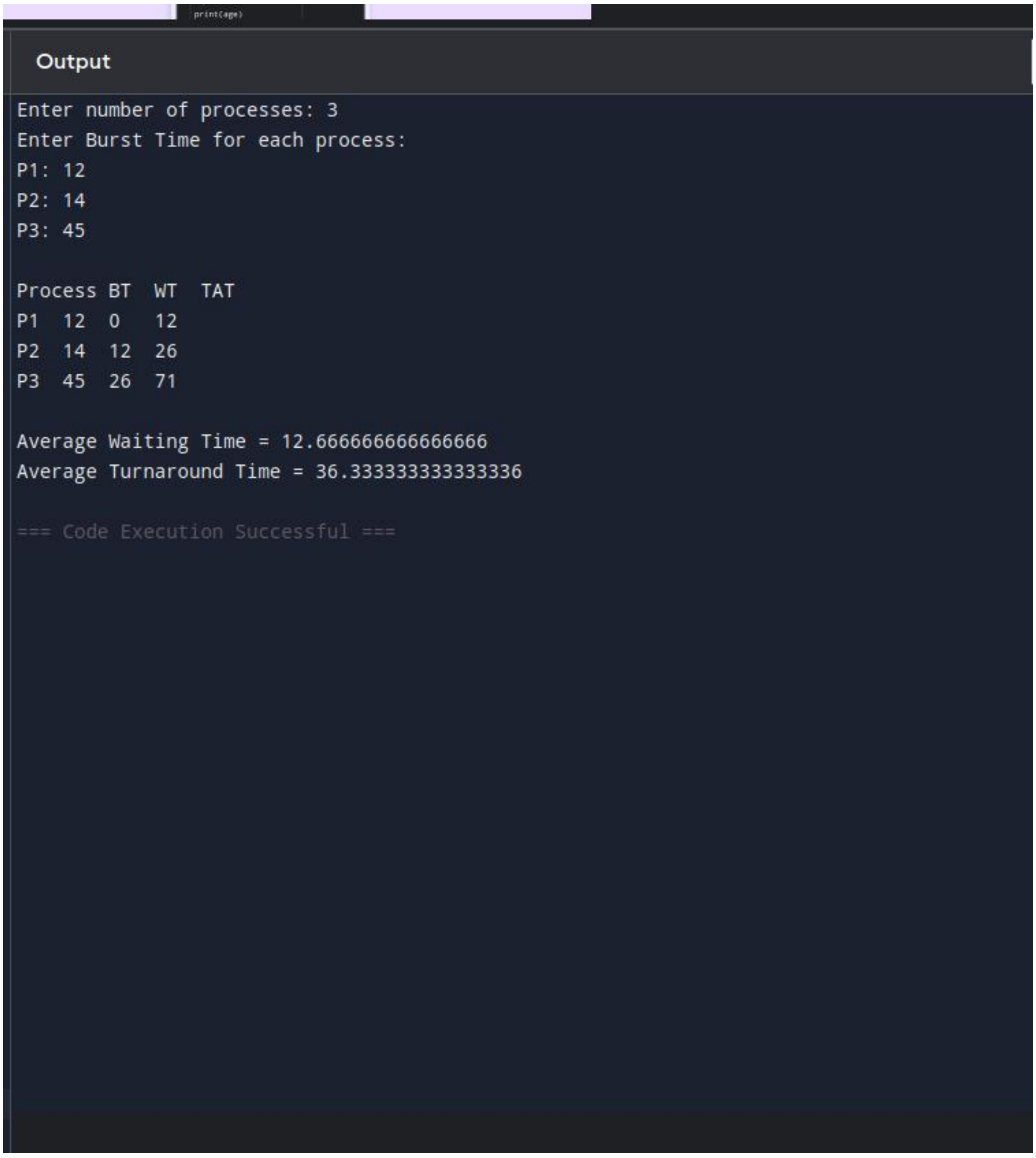
```
#FSFC

# FCFS (First Come First Serve) Scheduling Algorithm

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

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

 for i in range(1, n):

 wt[i] = bt[i - 1] + wt[i - 1] # Waiting time = sum of previous burst times

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

 for i in range(n):

 tat[i] = bt[i] + wt[i] # Turnaround time = Burst Time + Waiting Time

def findavgTime(processes, n, bt):

 wt = [0] * n # Waiting times

 tat = [0] * n # Turnaround times

 findWaitingTime(processes, n, bt, wt)

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

 print("\n--------------------------------------------")

 print("Process\tBurst Time\tWaiting Time\tTurnaround Time")

 print("--------------------------------------------")

 total_wt = 0

 total_tat = 0

 for i in range(n):

 total_wt += wt[i]

 total_tat += tat[i]

 print(f"P{processes[i]}\t{bt[i]}\t\t{wt[i]}\t\t{tat[i]}")

 avg_wt = total_wt / n

 avg_tat = total_tat / n

 print("--------------------------------------------")

 print(f"Average Waiting Time = {avg_wt:.2f}")

print(f"Average Turnaround Time = {avg_tat:.2f}")

 print("--------------------------------------------")

# ---------- MAIN ----------

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

```
processes = [i + 1 for i in range(n)]
bt = []
for i in range(n):
 b = int(input(f"Enter Burst Time for Process {i + 1}: "))
 bt.append(b)
findavgTime(processes, n, bt)
```

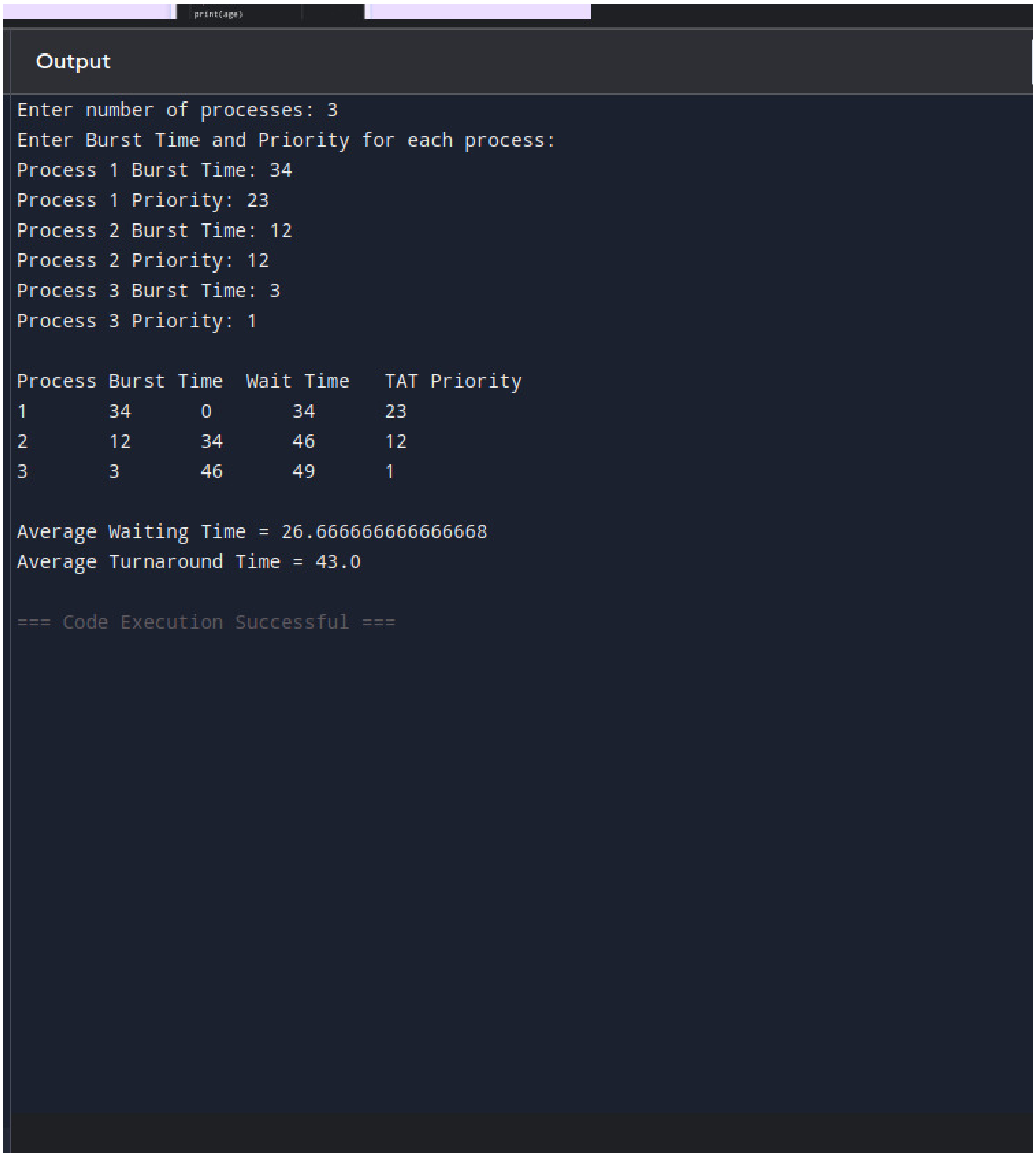


Output

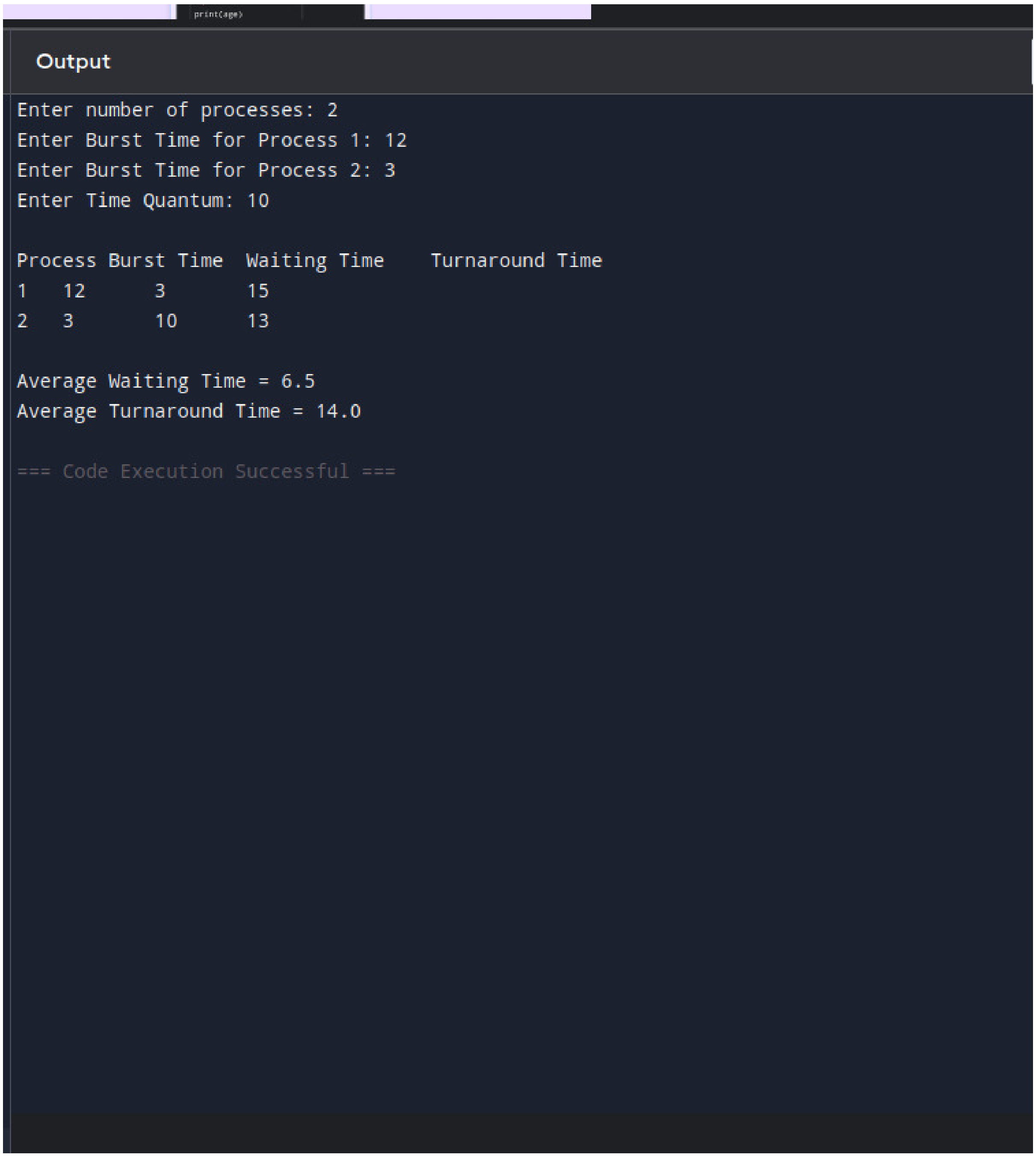
```
Enter number of processes: 3
Enter Burst Time for Process 1: 10
Enter Burst Time for Process 2: 5
Enter Burst Time for Process 3: 8

--------------------------------------------
Process Burst Time  Waiting Time    Turnaround Time
--------------------------------------------
P1   10      0       10
P2   5       10      15
P3   8       15      23
--------------------------------------------
Average Waiting Time = 8.33
Average Turnaround Time = 16.00
--------------------------------------------

=== Code Execution Successful ===
# SJF
def main():
 n = int(input("Enter number of processes: "))
 A = [[0 for _ in range(4)] for _ in range(n)]
 print("Enter Burst Time for each process:")
 for i in range(n):
 A[i][1] = int(input(f"P{i + 1}: "))
 A[i][0] = i + 1 # Process ID
 # Sorting by Burst Time
 A.sort(key=lambda x: x[1])
 total_wt = 0
 A[0][2] = 0 # Waiting Time for first process
 for i in range(1, n):
 A[i][2] = sum(A[j][1] for j in range(i))
 total_wt += A[i][2]
 avg_wt = total_wt / n
 total_tat = 0
 print("\nProcess\tBT\tWT\tTAT")
 for i in range(n):
 A[i][3] = A[i][1] + A[i][2]
 total_tat += A[i][3]
 print(f"P{A[i][0]}\t{A[i][1]}\t{A[i][2]}\t{A[i][3]}")
 print("\nAverage Waiting Time =", avg_wt)
 print("Average Turnaround Time =", total_tat / n)
if __name__ == "__main__":
 main()
```



```
Enter number of processes: 3
Enter Burst Time for each process:
P1: 12
P2: 14
P3: 45

Process BT  WT  TAT
P1  12  0   12
P2  14  12  26
P3  45  26  71

Average Waiting Time = 12.666666666666666
Average Turnaround Time = 36.333333333333336

=== Code Execution Successful ===
# PRIORITY
def main():
 n = int(input("Enter number of processes: "))
 p, pp, bt, w, t = [0]*n, [0]*n, [0]*n, [0]*n, [0]*n
 print("Enter Burst Time and Priority for each process:")
 for i in range(n):
 bt[i] = int(input(f"Process {i+1} Burst Time: "))
 pp[i] = int(input(f"Process {i+1} Priority: "))
 p[i] = i + 1
 # Sort by priority (Higher number = higher priority)
 for i in range(n - 1):
 for j in range(i + 1, n):
 if pp[i] < pp[j]:
 pp[i], pp[j] = pp[j], pp[i]
 bt[i], bt[j] = bt[j], bt[i]
 p[i], p[j] = p[j], p[i]
 w[0] = 0
 t[0] = bt[0]
 awt, atat = 0, t[0]
 for i in range(1, n):
 w[i] = t[i - 1]
 awt += w[i]
 t[i] = w[i] + bt[i]
 atat += t[i]
 print("\nProcess\tBurst Time\tWait Time\tTAT\tPriority")
for i in range(n):
 print(f"{p[i]}\t\t{bt[i]}\t\t{w[i]}\t\t{t[i]}\t\t{pp[i]}")
 print("\nAverage Waiting Time =", awt / n)
 print("Average Turnaround Time =", atat / n)
if __name__ == "__main__":
 main()
```



```
Enter number of processes: 3
Enter Burst Time and Priority for each process:
Process 1 Burst Time: 34
Process 1 Priority: 23
Process 2 Burst Time: 12
Process 2 Priority: 12
Process 3 Burst Time: 3
Process 3 Priority: 1

Process Burst Time  Wait Time   TAT Priority
1       34      0       34      23
2       12      34      46      12
3       3       46      49      1

Average Waiting Time = 26.666666666666668
Average Turnaround Time = 43.0

=== Code Execution Successful ===
# RR
def findWaitingTime(processes, n, bt, wt, quantum):
 rem_bt = bt.copy()
 t = 0 # Current time
 while True:
 done = True
 for i in range(n):
 if rem_bt[i] > 0:
 done = False
 if rem_bt[i] > quantum:
 t += quantum
 rem_bt[i] -= quantum
 else:
 t += rem_bt[i]
 wt[i] = t - bt[i]
 rem_bt[i] = 0
 if done:
 break
def findTurnAroundTime(processes, n, bt, wt, tat):
 for i in range(n):
 tat[i] = bt[i] + wt[i]
def findavgTime(processes, n, bt, quantum):
 wt = [0] * n
 tat = [0] * n
 findWaitingTime(processes, n, bt, wt, quantum)
 findTurnAroundTime(processes, n, bt, wt, tat)
 print("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time")
total_wt, total_tat = 0, 0
 for i in range(n):
 total_wt += wt[i]
 total_tat += tat[i]
 print(f"{processes[i]}\t{bt[i]}\t\t{wt[i]}\t\t{tat[i]}")
 print("\nAverage Waiting Time =", total_wt / n)
 print("Average Turnaround Time =", total_tat / n)
# Main
n = int(input("Enter number of processes: "))
processes = [i + 1 for i in range(n)]
bt = [int(input(f"Enter Burst Time for Process {i + 1}: ")) for i in range(n)]
quantum = int(input("Enter Time Quantum: "))
findavgTime(processes, n, bt, quantum)
```



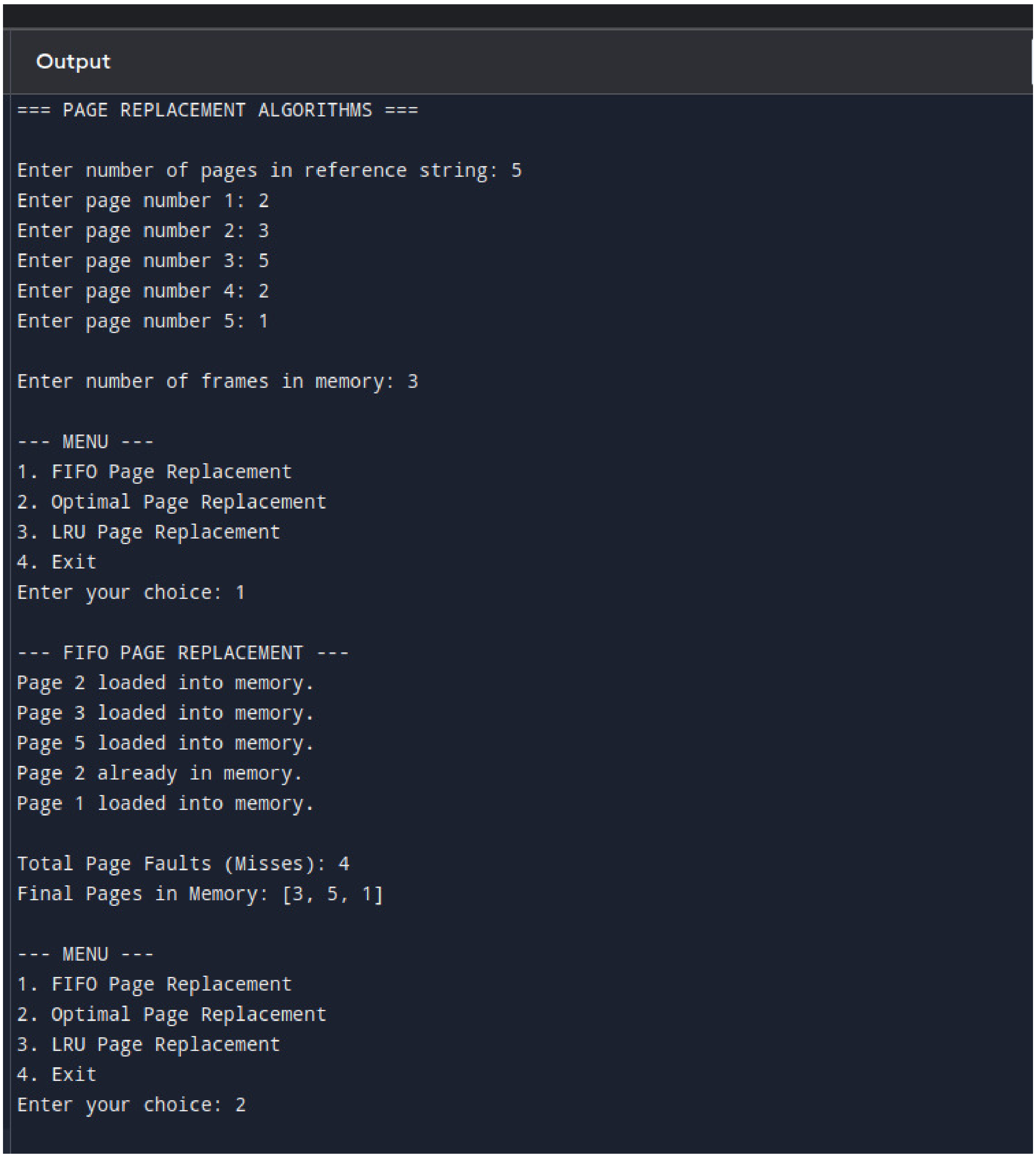
```
Enter number of processes: 2
Enter Burst Time for Process 1: 12
Enter Burst Time for Process 2: 3
Enter Time Quantum: 10

Process Burst Time  Waiting Time    Turnaround Time
1   12      3       15
2   3       10      13

Average Waiting Time = 6.5
Average Turnaround Time = 14.0

=== Code Execution Successful ===
# FIFO | OPTIMAL | LRU (with user input)
# ======================================
from collections import deque, OrderedDict
# ---------------- FIFO ----------------
def fifo_page_replacement(pages, capacity):
 page_queue = deque(maxlen=capacity)
 page_faults = 0
 print("\n--- FIFO PAGE REPLACEMENT ---")
 for page in pages:
 if page not in page_queue:
 print(f"Page {page} loaded into memory.")
 page_queue.append(page)
 page_faults += 1
 else:
 print(f"Page {page} already in memory.")
 print(f"\nTotal Page Faults (Misses): {page_faults}")
 print(f"Final Pages in Memory: {list(page_queue)}")
# ---------------- OPTIMAL ----------------
def optimal_page_replacement(pages, capacity):
 page_faults = 0
 page_frames = [-1] * capacity
 print("\n--- OPTIMAL PAGE REPLACEMENT ---")
 for i in range(len(pages)):
 if pages[i] not in page_frames:
 if -1 in page_frames:
 index = page_frames.index(-1)
 page_frames[index] = pages[i]
 else:
 future_occurrences = {page: float('inf') for page in page_frames}
 for j in range(i + 1, len(pages)):
 if pages[j] in future_occurrences and future_occurrences[pages[j]] ==
float('inf'):
future_occurrences[pages[j]] = j
 page_to_replace = max(future_occurrences, key=future_occurrences.get)
 index = page_frames.index(page_to_replace)
 page_frames[index] = pages[i]
 print(f"Page {pages[i]} loaded into memory.")
 page_faults += 1
 else:
 print(f"Page {pages[i]} already in memory.")
 print(f"\nTotal Page Faults: {page_faults}")
 print(f"Final Pages in Memory: {page_frames}")
# ---------------- LRU ----------------
class LRUCache:
 def __init__(self, capacity):
 self.cache = OrderedDict()
 self.capacity = capacity
 def refer(self, page):
 if page in self.cache:
 self.cache.move_to_end(page)
 else:
 if len(self.cache) >= self.capacity:
 self.cache.popitem(last=False)
 self.cache[page] = None
def lru_page_replacement(pages, capacity):
 lru_cache = LRUCache(capacity)
 page_faults = 0
 print("\n--- LRU PAGE REPLACEMENT ---")
 for page in pages:
 if page not in lru_cache.cache:
 print(f"Page {page} loaded into memory.")
 page_faults += 1
 else:
 print(f"Page {page} already in memory.")
 lru_cache.refer(page)
 print(f"\nTotal Page Faults: {page_faults}")
 print(f"Final Pages in Memory: {list(lru_cache.cache.keys())}")
# ---------------- MAIN PROGRAM ----------------
if __name__ == "__main__":
 print("=== PAGE REPLACEMENT ALGORITHMS ===")
# Take input for page reference string
 n = int(input("\nEnter number of pages in reference string: "))
 pages = []
 for i in range(n):
 page = int(input(f"Enter page number {i+1}: "))
 pages.append(page)
 # Take input for memory capacity
 capacity = int(input("\nEnter number of frames in memory: "))
 # Menu-driven choice
 while True:
 print("\n--- MENU ---")
 print("1. FIFO Page Replacement")
 print("2. Optimal Page Replacement")
 print("3. LRU Page Replacement")
 print("4. Exit")
 choice = int(input("Enter your choice: "))
 if choice == 1:
 fifo_page_replacement(pages, capacity)
 elif choice == 2:
 optimal_page_replacement(pages, capacity)
 elif choice == 3:
 lru_page_replacement(pages, capacity)
 elif choice == 4:
```

print("\nExiting... Thank you!")

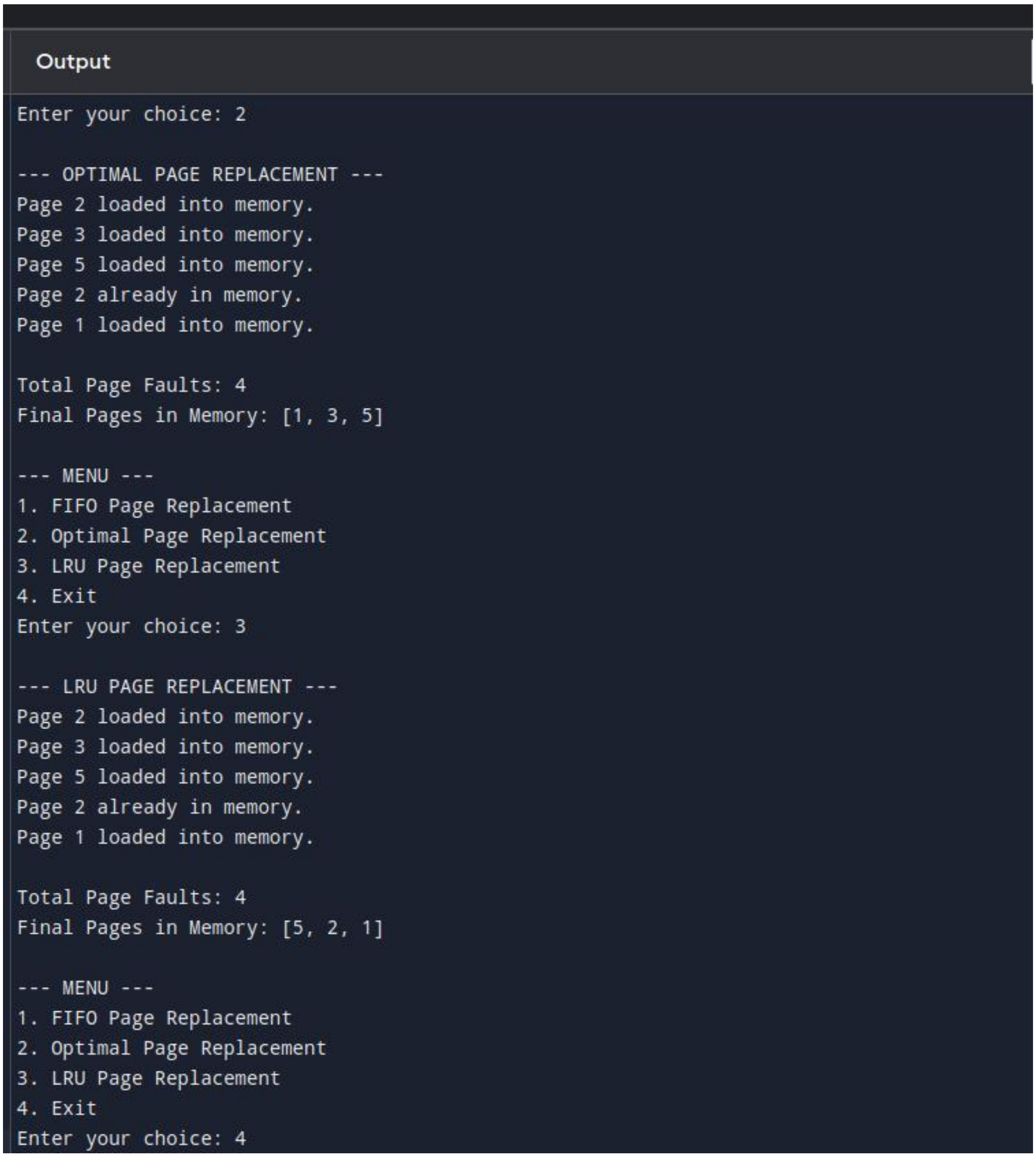
break

else:

print("Invalid choice! Please try again.")



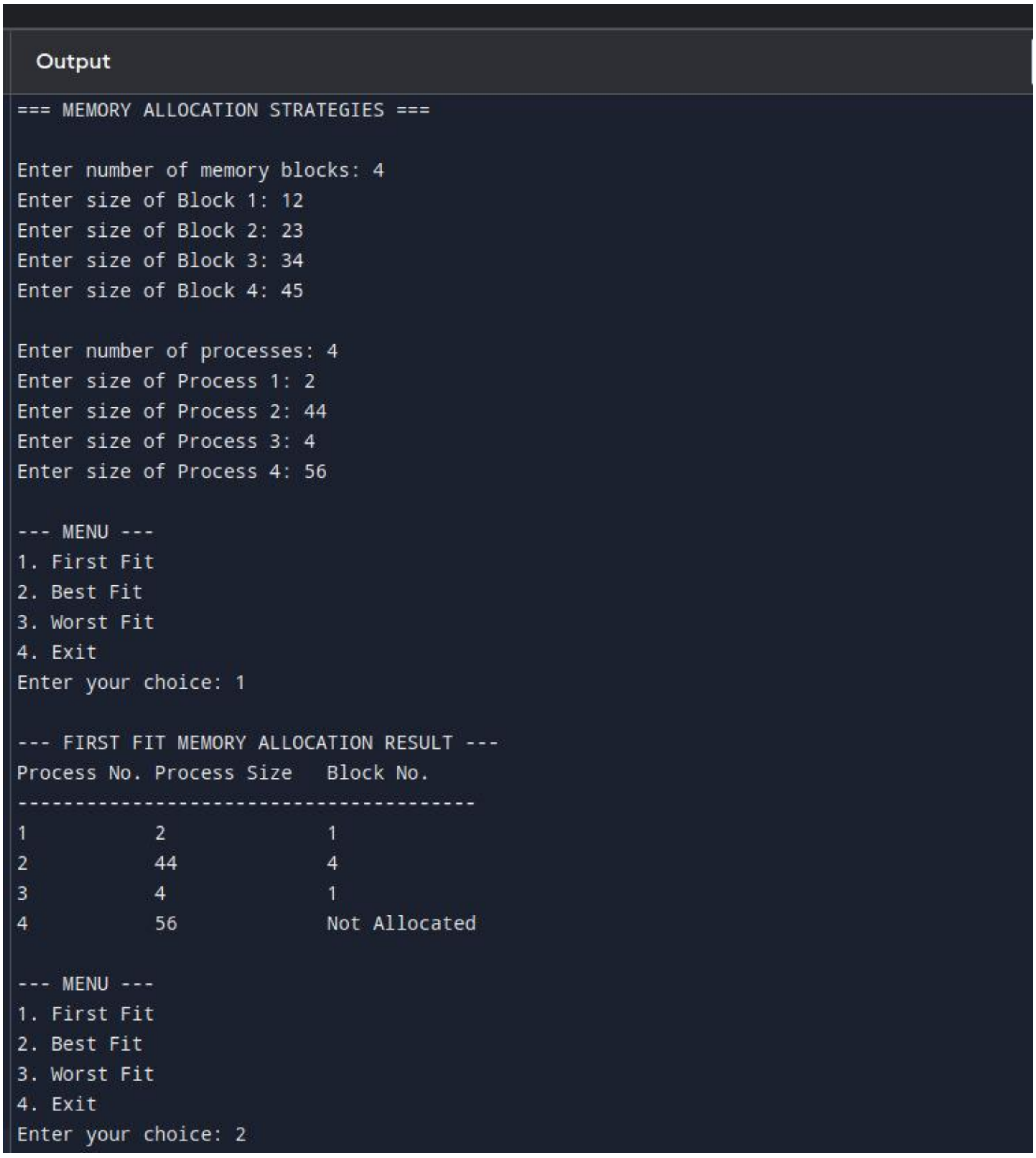
```
=== PAGE REPLACEMENT ALGORITHMS ===

Enter number of pages in reference string: 5
Enter page number 1: 2
Enter page number 2: 3
Enter page number 3: 5
Enter page number 4: 2
Enter page number 5: 1

Enter number of frames in memory: 3

--- MENU ---
1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. Exit
Enter your choice: 1

--- FIFO PAGE REPLACEMENT ---
Page 2 loaded into memory.
Page 3 loaded into memory.
Page 5 loaded into memory.
Page 2 already in memory.
Page 1 loaded into memory.

Total Page Faults (Misses): 4
Final Pages in Memory: [3, 5, 1]

--- MENU ---
1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. Exit
Enter your choice: 2
```


Output

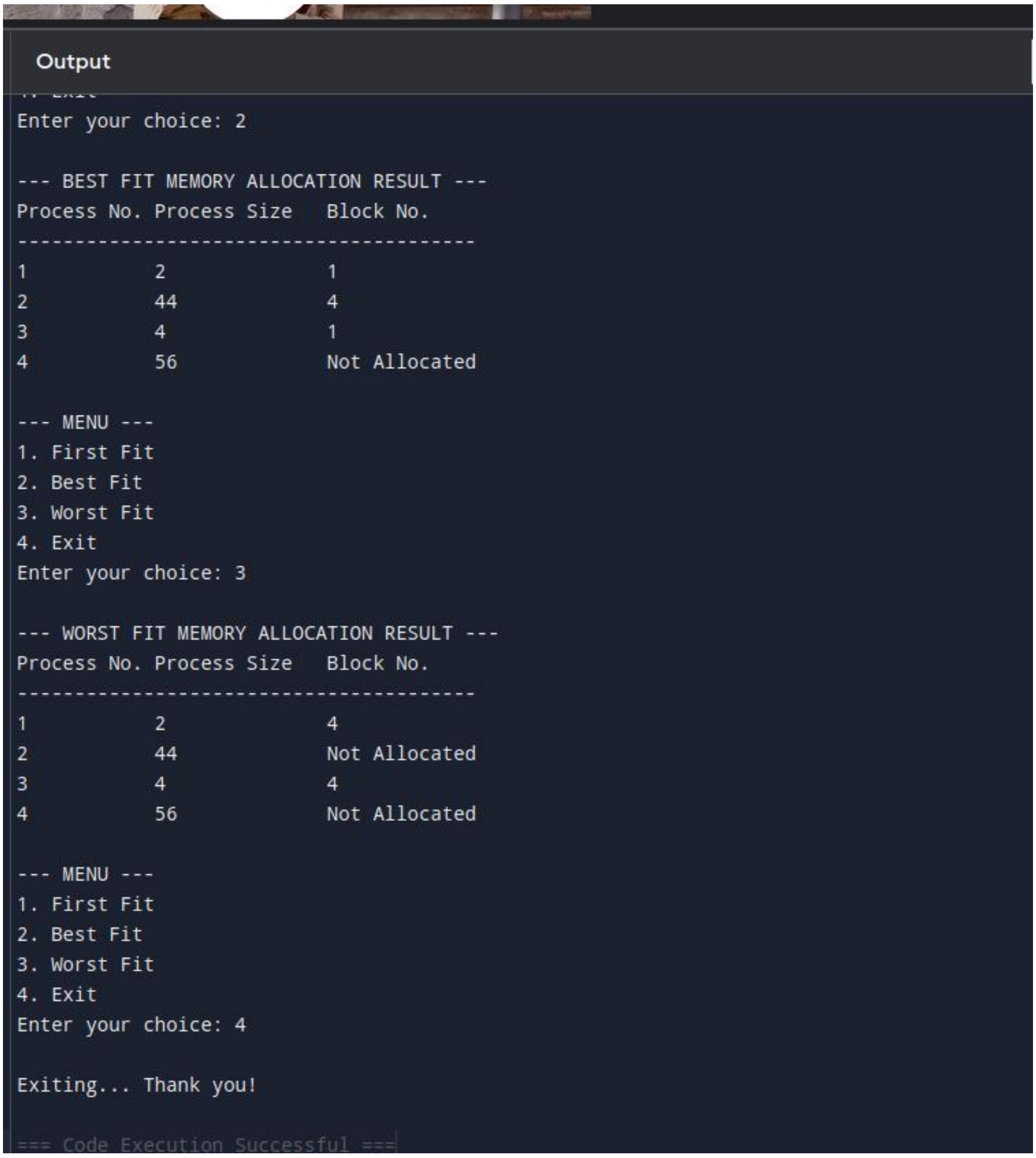
```
Enter your choice: 2

--- OPTIMAL PAGE REPLACEMENT ---
Page 2 loaded into memory.
Page 3 loaded into memory.
Page 5 loaded into memory.
Page 2 already in memory.
Page 1 loaded into memory.

Total Page Faults: 4
Final Pages in Memory: [1, 3, 5]

--- MENU ---
1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. Exit
Enter your choice: 3

--- LRU PAGE REPLACEMENT ---
Page 2 loaded into memory.
Page 3 loaded into memory.
Page 5 loaded into memory.
Page 2 already in memory.
Page 1 loaded into memory.

Total Page Faults: 4
Final Pages in Memory: [5, 2, 1]

--- MENU ---
1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. Exit
Enter your choice: 4
```

```
# -------- 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
 print("\n--- FIRST FIT MEMORY ALLOCATION RESULT ---")
 print(f"{'Process No.':<12}{'Process Size':<15}{'Block No.'}")
 print("-" * 40)
 for i in range(len(processes)):
 if allocation[i] != -1:
 print(f"{i+1:<12}{processes[i]:<15}{allocation[i]+1}")
 else:
 print(f"{i+1:<12}{processes[i]:<15}Not Allocated")
# -------- BEST FIT --------
def best_fit(blocks, processes):
 allocation = [-1] * len(processes)
 for i in range(len(processes)):
 best_idx = -1
 for j in range(len(blocks)):
 if blocks[j] >= processes[i]:
 if best_idx == -1 or blocks[j] < blocks[best_idx]:
 best_idx = j
 if best_idx != -1:
 allocation[i] = best_idx
 blocks[best_idx] -= processes[i]
 print("\n--- BEST FIT MEMORY ALLOCATION RESULT ---")
 print(f"{'Process No.':<12}{'Process Size':<15}{'Block No.'}")
 print("-" * 40)
 for i in range(len(processes)):
 if allocation[i] != -1:
print(f"{i+1:<12}{processes[i]:<15}{allocation[i]+1}")
 else:
 print(f"{i+1:<12}{processes[i]:<15}Not Allocated")
# -------- WORST FIT --------
def worst_fit(blocks, processes):
 allocation = [-1] * len(processes)
 for i in range(len(processes)):
 worst_idx = -1
 for j in range(len(blocks)):
 if blocks[j] >= processes[i]:
 if worst_idx == -1 or blocks[j] > blocks[worst_idx]:
 worst_idx = j
 if worst_idx != -1:
 allocation[i] = worst_idx
 blocks[worst_idx] -= processes[i]
 print("\n--- WORST FIT MEMORY ALLOCATION RESULT ---")
 print(f"{'Process No.':<12}{'Process Size':<15}{'Block No.'}")
 print("-" * 40)
 for i in range(len(processes)):
 if allocation[i] != -1:
 print(f"{i+1:<12}{processes[i]:<15}{allocation[i]+1}")
 else:
 print(f"{i+1:<12}{processes[i]:<15}Not Allocated")
# ------------------- MAIN PROGRAM -------------------
print("=== MEMORY ALLOCATION STRATEGIES ===\n")
# Take memory block details
m = int(input("Enter number of memory blocks: "))
blocks = []
```

```
for i in range(m):
 size = int(input(f"Enter size of Block {i+1}: "))
 blocks.append(size)
# Take process details
n = int(input("\nEnter number of processes: "))
processes = []
for i in range(n):
 size = int(input(f"Enter size of Process {i+1}: "))
 processes.append(size)
# Menu-driven choice
while True:
 print("\n--- MENU ---")
 print("1. First Fit")
print("2. Best Fit")
 print("3. Worst Fit")
 print("4. Exit")
 choice = int(input("Enter your choice: "))
 if choice == 1:
 first_fit(blocks.copy(), processes)
 elif choice == 2:
 best_fit(blocks.copy(), processes)
 elif choice == 3:
 worst_fit(blocks.copy(), processes)
 elif choice == 4:
 print("\nExiting... Thank you!")
 break
 else:
 print("Invalid choice! Please try again.")
```



```
=== MEMORY ALLOCATION STRATEGIES ===

Enter number of memory blocks: 4
Enter size of Block 1: 12
Enter size of Block 2: 23
Enter size of Block 3: 34
Enter size of Block 4: 45

Enter number of processes: 4
Enter size of Process 1: 2
Enter size of Process 2: 44
Enter size of Process 3: 4
Enter size of Process 4: 56

--- MENU ---
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 1

--- FIRST FIT MEMORY ALLOCATION RESULT ---
Process No. Process Size   Block No.
----------------------------------------
1           2              1
2           44             4
3           4              1
4           56             Not Allocated

--- MENU ---
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 2
```


Output

```
Enter your choice: 2

--- BEST FIT MEMORY ALLOCATION RESULT ---
Process No. Process Size   Block No.
----------------------------------------
1           2              1
2           44             4
3           4              1
4           56             Not Allocated

--- MENU ---
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 3

--- WORST FIT MEMORY ALLOCATION RESULT ---
Process No. Process Size   Block No.
----------------------------------------
1           2              4
2           44             Not Allocated
3           4              4
4           56             Not Allocated

--- MENU ---
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 4

Exiting... Thank you!

=== Code Execution Successful ===
```