DEWAN VS INSTITUTE OF ENGINEERING & TECHNOLOGY

MEERUT(COLEGE CODE 311)

DEPARTMENT OF CSE/ AI

B.TECH IV Semester (CSE/AI)

OPERATING SYSTEM LAB MANUAL

(BCS-451)

DEWAN INSTITUTE OF ENGINEERING AND TECHNOLOGY MEERUT

DEPARTMENT OF COMPUTER SCIENCE &ENGINEERING / DEPARTMENT OF ARTIFICIAL INTELLIGENCE

OPERATING SYSTEM LAB (BCS-451)

LIST OF PROGRAM

1. Write a program to implement the bubble sorting algorithm.
2. Write a program to implement the insertion sorting algorithm.
3. Write a program to implement the selection sorting algorithm.
4. Write a program to implement the simulation of the First Come First Serve (FCFS) CPU scheduling algorithm.
5. Write a program to implement the simulation of the Shortest Job First (SJF) CPU scheduling algorithm.
6. Write a program to implement the Worst- Fit contiguous allocation technique.
7. Write a program to implement the Best- Fit contiguous allocation technique.
8. Write a program to implement the First- Fit contiguous allocation technique.
9. Write a program to implement the Priority Scheduling algorithm.
10. Write a program to implement the Round Robin Scheduling algorithm.

PROGRAM 1

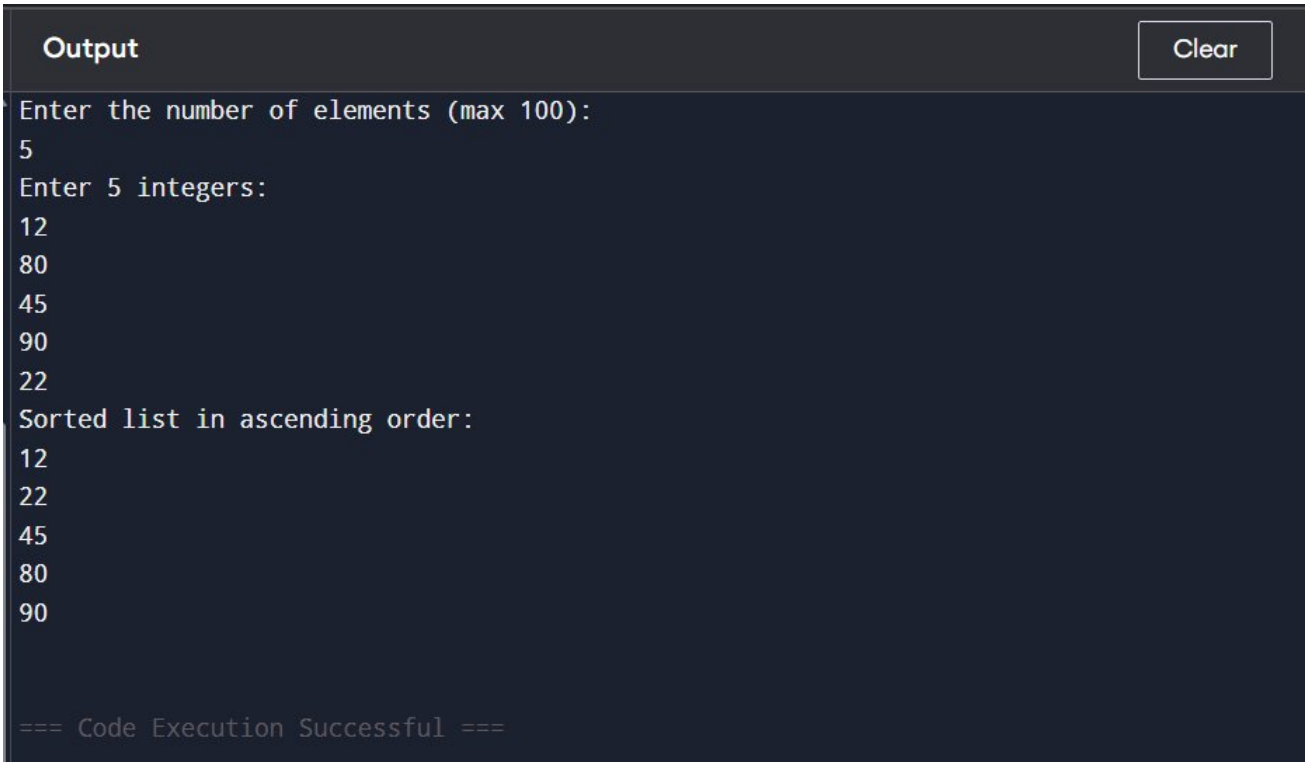
Objective: Write a program to implement the bubble sorting algorithm.

Description: A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Program: Bubble Sort

```
#include <stdio.h>
int main() {
 int n, i, j, temp;
 // Declare an array with a maximum size (e.g., 100) to handle user input
 int array[100];
 printf("Enter the number of elements (max 100):\n");
 scanf("%d", &n); // Get the number of elements
 printf("Enter %d integers:\n", n);
 // Loop to get 'n' elements from the user
 for (i = 0; i < n; i++) {
scanf("%d", &array[i]);
}
// Bubble sort algorithm implementation
for (i = 0 ; i < n - 1; i++) {
for (j = 0 ; j < n - i - 1; j++) {
 // Compare adjacent elements and swap if they are in the wrong order
 if (array[j] > array[j+1]) {
   temp = array[j];
   array[j] = array[j+1];
   array[j+1] = temp;
  }
} }
```

```
 printf ("Sorted list in ascending order:\n");
 // Loop to print the sorted array
 for (i = 0; i < n; i++) {
   printf("%d\n", array[i]);
 }
 return 0;
}
```



PROGRAM 2

Objective: Write a program to implement the insertion sorting algorithm.

Description: A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Program: Insertion Sort

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
    int i, key, j;
    // Start the outer loop from the second element (index 1), assuming the first is sorted
    for (i = 1; i < n; i++) {
        key = arr[i]; // Store the current element to be inserted
        j = i - 1; // Start comparing with the element before the current one
        // Move elements of arr[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; // Shift the greater element to the right
            j = j - 1;
        }
        arr[j + 1] = key; // Insert the key in its correct position
    }
}
int main() {
    int n, i;
    int arr[100]; // Declare an array with a maximum size (e.g., 100)
    printf("Enter number of elements: ");
    scanf("%d", &n); // Get the number of elements from the user
    printf("Enter %d integers:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]); // Get the elements from the user
```

```
    }
    insertionSort(arr, n); // Call the insertion sort function
    printf("Sorted list in ascending order:\n");
    for (i = 0; i < n; i++) {
        printf("%d\n", arr[i]); // Print the sorted array
    }
    return 0;
}
```

PROGRAM 3

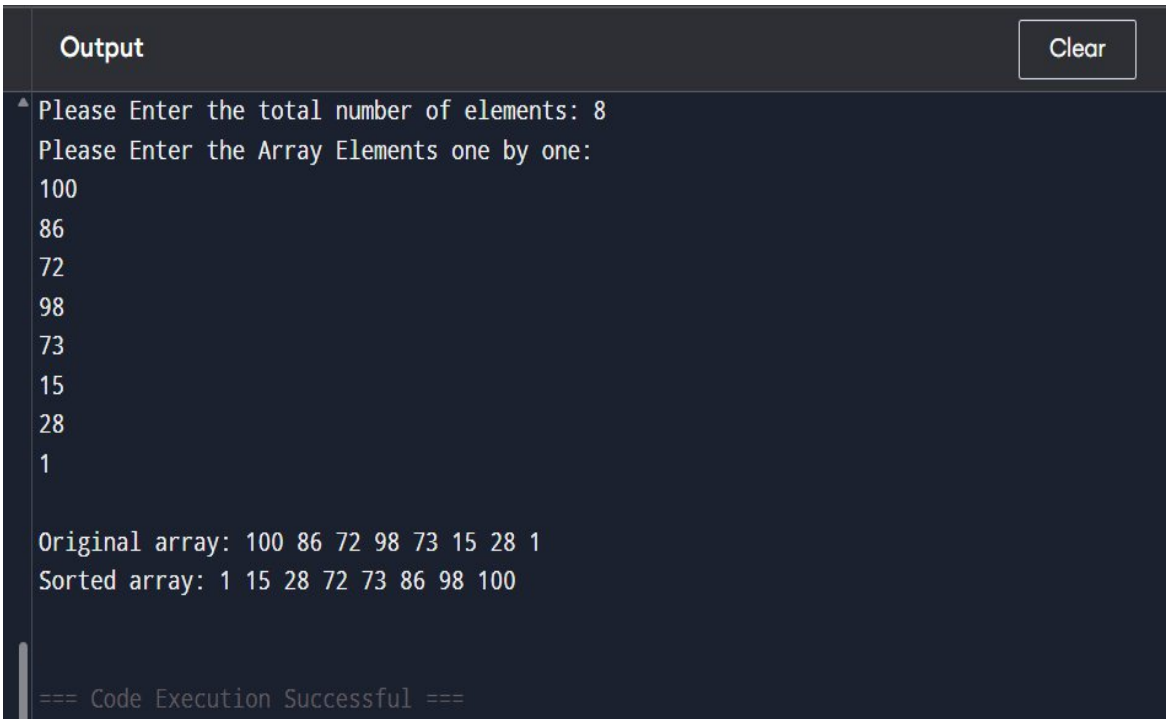
Objective: Write a program to implement the selection sorting algorithm.

Description: A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Program: Selection Sort

```
#include <stdio.h>
// Function to perform selection sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp;
    // One by one move the minimum element to the beginning of the unsorted array
    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in the remaining unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Swap the found minimum element with the first element of the unsorted part
        // Only swap if the minimum element is not already at the correct position
        if (min_idx != i) {
            temp = arr[i];
            arr[i] = arr[min_idx];
            arr[min_idx] = temp;
        }
    }
}
// Function to print an array
void printArray(int arr[], int size) {
```

```
    int i;
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
// Main function to get user input and run the sort
int main() {
    int arr[100], n, i;
    printf("Please Enter the total number of elements: ");
    // Use the official documentation for scanf to understand input handling
    scanf("%d", &n);
    printf("Please Enter the Array Elements one by one:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("\nOriginal array: ");
    printArray(arr, n);
    // Call the selection sort function
    selectionSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
```


Output
Clear
Please Enter the total number of elements: 8
Please Enter the Array Elements one by one:
100
86
72
98
73
15
28
1
Original array: 100 86 72 98 73 15 28 1
Sorted array: 1 15 28 72 73 86 98 100
=== Code Execution Successful ===

PROGRAM 4

Objective: Write a program to implement the simulation of the First Come First Serve (FCFS) CPU scheduling algorithm.

Description: This algorithm is **non-preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time. In contrast, preemptive scheduling is based on priority: a scheduler may preempt a low-priority running process at any time when a high-priority process enters the ready state.

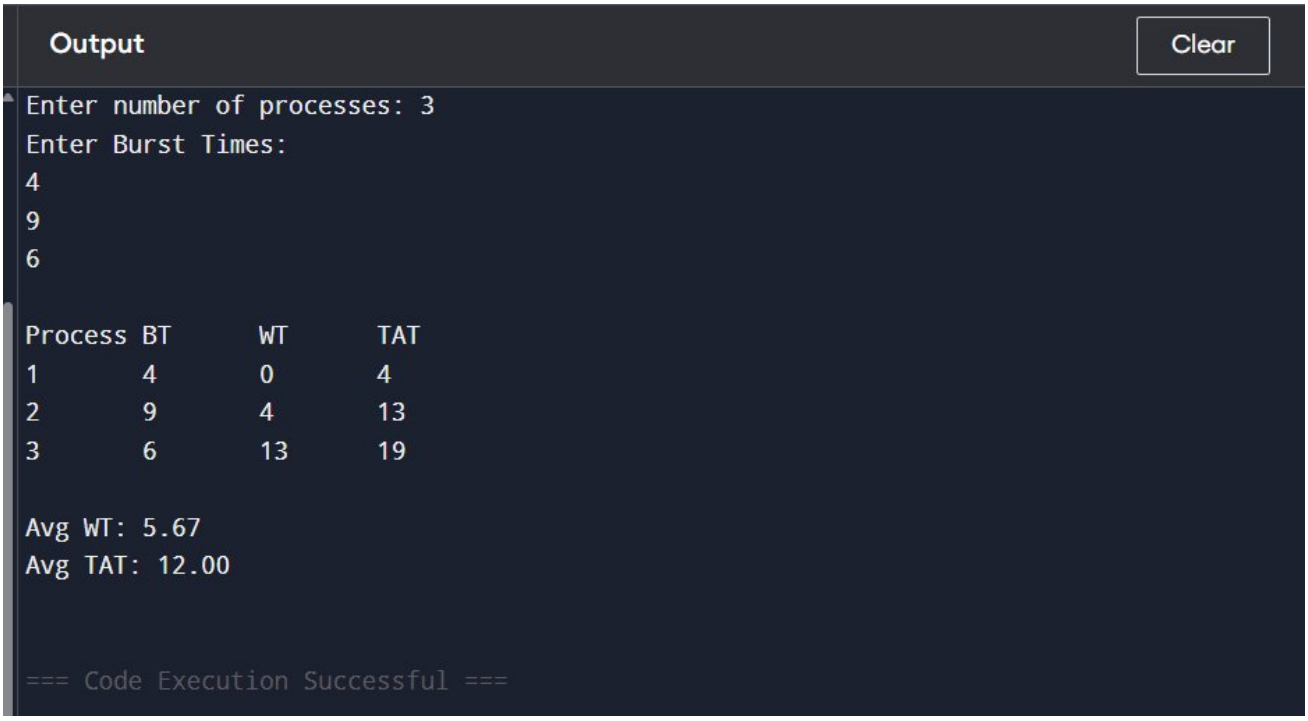
First Come First Serve (FCFS)

- Jobs are executed on a first-come, first-served basis.
- It is a non-preemptive, preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on a FIFO queue.
- Poor performance as the average wait time is high.

Program: First Come First Serve (FCFS)

```
#include <stdio.h>
// Function to implement FCFS scheduling
void FCFS(int bt[], int n) {
int wt[n], tat[n], total_wt = 0, total_tat = 0;
// First process has 0 waiting time
wt[0] = 0;
// Calculate waiting time for remaining processes
for (int i = 1; i < n; i++) {
      wt[i] = wt[i - 1] + bt[i - 1];
      total_wt += wt[i];
   }
// Calculate turnaround time and print results
   printf("\nProcess\tBT\t\tWT\t\tTAT\n");
   for (int i = 0; i < n; i++) {
```

```
        tat[i] = wt[i] + bt[i];
        total_tat += tat[i];
        printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
    }
printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", (float)total_wt/n, (float)total_tat/n);
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int bt[n];
    printf("Enter Burst Times:\n");
    for (int i = 0; i < n; i++) scanf("%d", &bt[i]);
    FCFS(bt, n);
    return 0;
}
```



PROGRAM 5

Objective: Write a program to implement the simulation of the Shortest Job First (SJF) CPU scheduling algorithm.

Description: This algorithm is **non-preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time. In contrast, preemptive scheduling is based on priority: a scheduler may preempt a low-priority running process at any time when a high-priority process enters the ready state.

Shortest Job First (SJF)

- This is a non-preemptive, preemptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where the required CPU time is known in advance.
- Impossible to implement in interactive systems where the required CPU time is not known.
- The processors should know in advance how much time the process will take.

Program: Shortest Job First (SJF)

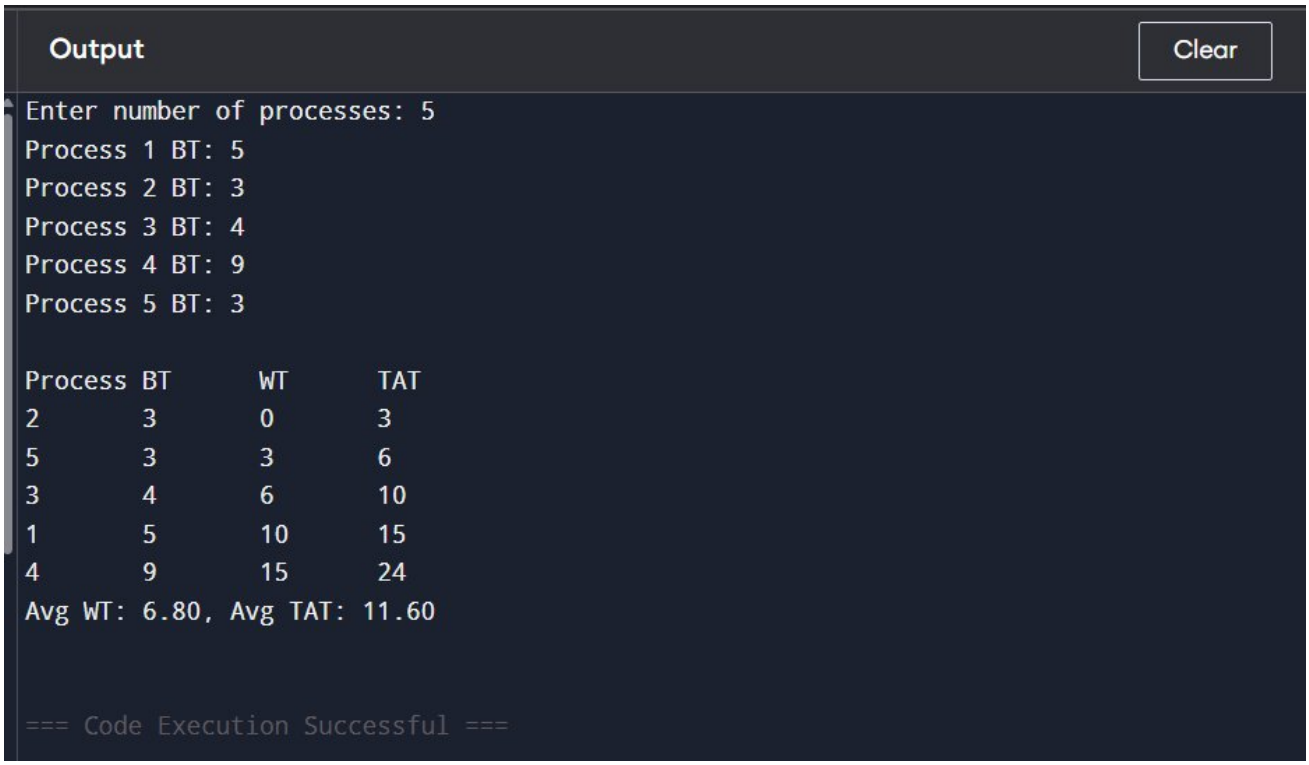
```
#include <stdio.h>
struct Process {
   int id, bt, wt, tat;
};

// Sorts processes by burst time and calculates times
void SJF(struct Process p[], int n) {
   int total_wt = 0, total_tat = 0;
   // Simple bubble sort for demonstration
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (p[i].bt > p[j].bt) {
```

```
            struct Process temp = p[i];
            p[i] = p[j];
            p[j] = temp;
        }
    }
  }
  p[0].wt = 0;
  for (int i = 1; i < n; i++) {
    p[i].wt = p[i - 1].wt + p[i - 1].bt;
    total_wt += p[i].wt;
  }
  for (int i = 0; i < n; i++) {
    p[i].tat = p[i].wt + p[i].bt;
    total_tat += p[i].tat;
  }
  printf("\nProcess\tBT\t\tWT\t\tTAT\n");
  for (int i = 0; i < n; i++)
    printf("%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].bt, p[i].wt, p[i].tat);
  printf("Avg WT: %.2f, Avg TAT: %.2f\n", (float)total_wt/n, (float)total_tat/n);
}

int main() {
  int n;
  printf("Enter number of processes: ");
  scanf("%d", &n);
  struct Process p[n];
  for (int i = 0; i < n; i++) {
    p[i].id = i + 1;
    printf("Process %d BT: ", p[i].id);
    scanf("%d", &p[i].bt);
  }
```

```
    SJF(p, n);
    return 0;
}
```



PROGRAM 6

Objective: Write a program to implement the Worst- Fit contiguous allocation technique.

Description: Worst fit is a contiguous memory allocation strategy that assigns a process to the largest available memory partition (hole). It searches the entire memory to find this space, aiming to leave behind smaller, more usable fragments. It is often slow and inefficient, resulting in higher internal fragmentation.

Program: Worst- Fit Contiguous Allocation Technique.

```
#include <stdio.h>
#define MAX_BLOCKS 10
#define MAX_PROCESSES 10
/**
 * @brief Implements the Worst Fit memory allocation algorithm.
 *
 * @param blockSize Array of memory block sizes (this array is modified).
 * @param m Number of memory blocks.
 * @param processSize Array of process sizes.
 * @param n Number of processes.
 */
void worstFit(int blockSize[], int m, int processSize[], int n) {
    // Stores the block ID of the block allocated to a process (index of blockSize array)
    int allocation[n];

    // Initially, no block is assigned to any process, so initialize with -1
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }
    // Iterate through all processes
    for (int i = 0; i < n; i++) {
        // Find the index of the largest suitable block for the current process
```

```
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx]) {
                    worstIdx = j;
                }
            }
        }
        // If a suitable block is found, allocate the process to it
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            // Reduce the available memory in the allocated block
            blockSize[worstIdx] -= processSize[i];
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d \t\t\t %d \t\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            // Display block number (using 1-based indexing for readability)
            printf("%d\n", allocation[i] + 1);
        } else {
            printf("Not Allocated\n");
        }
    }
}
// Driver code
int main() {
    // Example memory blocks (in KB)
    int blockSize[] = {100, 500, 200, 300, 600};
    // Example processes (in KB)
```

```
    int processSize[] = {212, 417, 112, 426};
    // Calculate the number of blocks and processes
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    printf("--- Worst-Fit Memory Allocation Technique ---\n");
    worstFit(blockSize, m, processSize, n);
    return 0;
}
```

PROGRAM 7

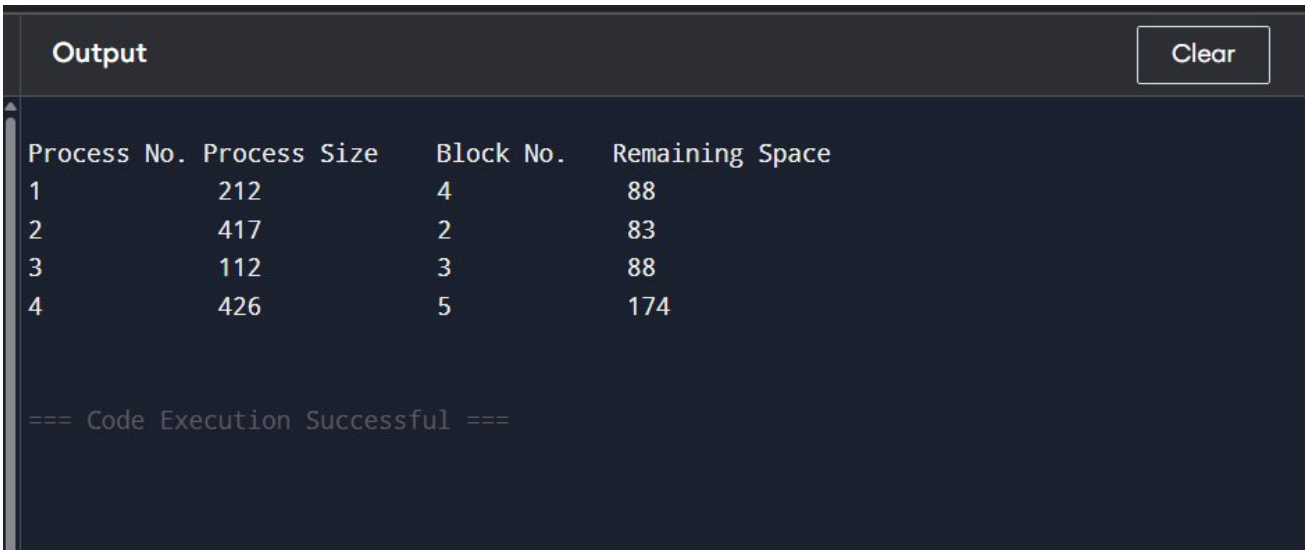
Objective: Write a program to implement the Best- Fit contiguous allocation technique.

Description: Best fit is a memory management algorithm that allocates the smallest available free partition (hole) capable of holding a process, minimizing wasted space and reducing internal fragmentation. It scans all memory blocks to find the closest fit, which can be computationally intensive but results in high memory utilization.

Program: Best- Fit Contiguous Allocation Technique.

```
#include <stdio.h>
#define MAX_BLOCKS 10
#define MAX_PROCESSES 10
// Function to implement Best-Fit
void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[MAX_PROCESSES];
    int tempBlockSize[MAX_BLOCKS];
    // Initialize all allocations to -1 (not allocated)
    for (int i = 0; i < n; i++) allocation[i] = -1;
    for (int i = 0; i < m; i++) tempBlockSize[i] = blockSize[i];
    // Pick each process and find the best fit block
    for (int i = 0; i < n; i++) {
        int bestIdx = -1;
        for (int j = 0; j < m; j++) {
            if (tempBlockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || tempBlockSize[j] < tempBlockSize[bestIdx]) {
                    bestIdx = j;
                }
            }
        }
        // Allocate block if a suitable one is found
        if (bestIdx != -1) {
```

```
            allocation[i] = bestIdx;
            tempBlockSize[bestIdx] -= processSize[i];
        }
    }
    // Display the results
    printf("\nProcess No.\tProcess Size\tBlock No.\tRemaining Space\n");
    for (int i = 0; i < n; i++) {
        printf("%d \t\t\t %d \t\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) printf("%d \t\t\t %d\n", allocation[i] + 1, tempBlockSize[allocation[i]]);
        else printf("Not Allocated\n");
    }
}
int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    bestFit(blockSize, 5, processSize, 4);
    return 0;
}
```



PROGRAM 8

Objective: Write a program to implement the First- Fit contiguous allocation technique.

Description: First Fit is a contiguous memory allocation strategy where the operating system searches for the first available memory block (or "hole") that is large enough to accommodate a process. Unlike other algorithms that scan all blocks to find an optimal match, First Fit stops its search as soon as it finds the first sufficient space.

Program: First- Fit Contiguous Allocation Technique.

```
#include <stdio.h>
// Function to allocate memory to blocks as per First Fit algorithm
void firstFit(int blockSize[], int m, int processSize[], int n)
{
   // Stores block id of the block allocated to a process
   int allocation[n];
   int i, j;
   // Initially no block is assigned to any process
   for (i = 0; i < n; i++) {
      allocation[i] = -1;
   }
   // pick each process and find suitable blocks according to its size
   for (i = 0; i < n; i++) // n -> number of processes
   {
      for (j = 0; j < m; j++) // m -> number of blocks
      {
         if (blockSize[j] >= processSize[i])
         {
            // allocating block j to the ith process
            allocation[i] = j;
            // Reduce available memory in this block
            blockSize[j] -= processSize[i];
```

```
            break; // Go to the next process in the queue
        }
    }
  }
  printf("\nProcess No.\tProcess Size\tBlock No.\n");
  for (i = 0; i < n; i++)
  {
    printf(" %i\t\t\t", i + 1);
    printf("%i\t\t\t\t", processSize[i]);
    if (allocation[i] != -1)
      printf("%i", allocation[i] + 1);
    else
      printf("Not Allocated");
    printf("\n");
  }
}
// Driver code
int main()
{
  // Example memory blocks and process sizes
  int blockSize[] = {100, 500, 200, 300, 600};
  int processSize[] = {212, 417, 112, 426};
  int m = sizeof(blockSize) / sizeof(blockSize[0]); // Number of blocks
  int n = sizeof(processSize) / sizeof(processSize[0]); // Number of processes
  firstFit(blockSize, m, processSize, n);
  return 0;
}
```

Output

Clear

```
Process No. Process Size    Block No.
 1          212             2
 2          417             5
 3          112             2
 4          426             Not Allocated


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

PROGRAM 9

Objective: Write a program to implement the Priority Scheduling algorithm.

Description: Priority scheduling is an operating system algorithm that assigns a priority value (numerical) to each process, executing the highest-priority tasks first. It supports both preemptive (interrupts lower-priority tasks) and non-preemptive (runs to completion) modes to manage CPU time efficiently, often using FCFS for equal-priority processes.

Program: Priority Scheduling Algorithm.

```
#include <stdio.h>
struct Process {
   int id, bt, wt, tat, priority;
};
// Sorts processes by priority (lower value = higher priority)
void sortByPriority(struct Process p[], int n) {
   struct Process temp;
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (p[i].priority > p[j].priority) {
            temp = p[i]; p[i] = p[j]; p[j] = temp;
         }
      }
   }
}
// Calculates waiting and turnaround times
void calculateTimes(struct Process p[], int n) {
   p[0].wt = 0; // First process has 0 waiting time
   p[0].tat = p[0].bt;
   for (int i = 1; i < n; i++) {
      p[i].wt = p[i - 1].wt + p[i - 1].bt;
      p[i].tat = p[i].bt + p[i].wt;
```

```
    }
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P%d - Burst Time & Priority: ", p[i].id);
        scanf("%d %d", &p[i].bt, &p[i].priority);
    }
    sortByPriority(p, n); //
    calculateTimes(p, n); //
    printf("\nID\tBT\tPriority\tWT\tTAT\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t\t%d\t\t%d\t%d\n", p[i].id, p[i].bt, p[i].priority, p[i].wt, p[i].tat);
    return 0; }
```

Output

Clear

```
Enter number of processes: 6
P1 - Burst Time & Priority: 47 3
P2 - Burst Time & Priority: 12 5
P3 - Burst Time & Priority: 38 6
P4 - Burst Time & Priority: 26 1
P5 - Burst Time & Priority: 1 4
P6 - Burst Time & Priority: 53 2

ID  BT  Priority    WT  TAT
4   26      1       0   26
6   53      2       26  79
1   47      3       79  126
5   1       4       126 127
2   12      5       127 139
3   38      6       139 177


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

PROGRAM 10

Objective: Write a program to implement the Round Robin Scheduling algorithm.

Description: Round Robin (RR) scheduling is a preemptive CPU algorithm that assigns a fixed time unit, known as a time quantum (typically 10-100 ms), to each process in the ready queue in a circular, First-In-First-Out (FIFO) manner. It ensures fairness, prevents starvation, and is ideal for time-sharing systems.

Program: Round Robin Scheduling algorithm.

```
#include <stdio.h>
// Function to find the waiting time and turnaround time for all processes
void findavgTime(int processes[], int n, int burst_time[], int quantum) {
   int waiting_time[n], turnaround_time[n], total_wt = 0, total_tat = 0;
   // Store remaining burst time for each process
   int remaining_burst_time[n];
   for (int i = 0; i < n; i++) {
      remaining_burst_time[i] = burst_time[i];
   }
   int current_time = 0; // Current time
   // Keep traversing processes until all of them are completed
   while (1) {
      int done = 1; // Flag to check if all processes are done
      // Traverse all processes one by one repeatedly
      for (int i = 0; i < n; i++) {
         if (remaining_burst_time[i] > 0) {
            done = 0; // A process is still pending
            if (remaining_burst_time[i] > quantum) {
               // Process runs for a time quantum
               current_time += quantum;
               remaining_burst_time[i] -= quantum;
            } else {
```

```
            // Process runs for the remaining burst time (last cycle)
            current_time += remaining_burst_time[i];
            waiting_time[i] = current_time - burst_time[i];
            remaining_burst_time[i] = 0; // Mark process as completed
        }
      }
    }
    // If all processes are done, break the loop
    if (done == 1)
      break;
  }
  // Calculate turnaround time for all processes
  for (int i = 0; i < n; i++) {
    turnaround_time[i] = burst_time[i] + waiting_time[i];
    total_wt += waiting_time[i];
    total_tat += turnaround_time[i];
  }
  // Display process details and average times
  printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
  printf("-------------------------------------------------\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t\t%d\t\t\t%d\t\t\t%d\n", processes[i], burst_time[i], waiting_time[i],
turnaround_time[i]);
  }
  printf("\nAverage Waiting Time: %.2f\n", (float)total_wt / (float)n);
  printf("Average Turnaround Time: %.2f\n", (float)total_tat / (float)n);
}
// Main function
int main() {
  // Process IDs
  int processes[] = {1, 2, 3};
```

```
    int n = sizeof processes / sizeof processes[0];
    // Burst time of all processes
    int burst_time[] = {10, 5, 8};

    // Time quantum
    int quantum = 2;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```

Output

Clear

```
Process Burst Time  Waiting Time    Turnaround Time
-------------------------------------------------
1           10          13          23
2           5           10          15
3           8           13          21

Average Waiting Time: 12.00
Average Turnaround Time: 19.67


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