```
Function FCFS_Scheduling(n, at[], bt[]):
    // Initialize process IDs and tracking arrays
    Initialize proc_id[] with 1 to n
    Initialize ct[], tat[], wt[] with zeros
    Initialize ttat, twt to 0.0

    // Sort processes by arrival time
    Call sort(proc_id, at, bt, n)

    // Compute completion times
    c = 0
    For i from 0 to n-1:
        c = max(c, at[i]) + bt[i]
        ct[i] = c

    // Calculate turnaround and waiting times
    For i from 0 to n-1:
        tat[i] = ct[i] - at[i]
        wt[i] = tat[i] - bt[i]
        ttat += tat[i]
        twt += wt[i]

    // Calculate averages
    avg_tat = ttat / n
    avg_wt = twt / n

    // Output results
    Print "PID\tAT\tBT\tCT\tTAT\tWT"
    For i from 0 to n-1:
        Print proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i]

    Print "Average turnaround time:", avg_tat
    Print "Average waiting time:", avg_wt
```

```
Function SJF_Non_Preemptive_Scheduling(n, at[], bt[]):
    // Initialize process IDs and tracking arrays
    Initialize proc_id[] with 1 to n
    Initialize ct[], tat[], wt[], m[] with zeros
    Initialize avg_tat, ttat, avg_wt, twt to 0.0
    Initialize current_time c to 0
    count = 0

    While count < n:
        min = infinity
        mb = -1

        // Find the shortest job that is ready
        For i from 0 to n-1:
            If at[i] <= c and m[i] != 1:
                If bt[i] < min:
                    min = bt[i]
                    mb = i

        // If no job is ready, increment time
        If mb == -1:
            c += 1
            Continue

        m[mb] = 1
        count += 1
        c = max(c, at[mb]) + bt[mb]
        ct[mb] = c

    // Calculate turnaround and waiting times
    For i from 0 to n-1:
        tat[i] = ct[i] - at[i]
        wt[i] = tat[i] - bt[i]
        ttat += tat[i]
        twt += wt[i]

    // Calculate averages
    avg_tat = ttat / n
    avg_wt = twt / n

    // Output results
    Print "SJF Non-Preemptive scheduling:"
    Print "PID\tAT\tBT\tCT\tTAT\tWT"
    For i from 0 to n-1:
        Print proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i]

    Print "Average turnaround time:", avg_tat
    Print "Average waiting time:", avg_wt
```

```
Function Priority_Non_Preemptive_Scheduling(n, at[], bt[], p[]):
    // Initialize process IDs and tracking arrays
    Initialize proc_id[] with 1 to n
    Initialize ct[], tat[], wt[], rt[], m[] with zeros
    Initialize avg_tat, ttat, avg_wt, twt to 0.0
    Initialize c to 0

    // Input priorities, arrival, and burst times
    For i from 0 to n-1:
        proc_id[i] = i + 1
        m[i] = 0
        Input p[i], at[i], bt[i]
        rt[i] = -1

    // Sort processes by priority
    Sort proc_id[], p[], at[], bt[] based on priority

    count = 0
    priority = p[0]

    While count < n:
        x = -1
        // Find the highest priority process available
        For i from 0 to n-1:
            If at[i] <= c and p[i] >= priority and m[i] != 1:
                x = i
                priority = p[i]

        // Update response time
        If rt[x] == -1:
            rt[x] = c - at[x]

        // Update current time
        c = max(c, at[x]) + bt[x]
        count += 1
        ct[x] = c
        m[x] = 1

        // Adjust priority for the next process
        While x > 0 and m[x - 1] != 1:
            x -= 1
        priority = p[x]

    // Calculate turnaround and waiting times
    For i from 0 to n-1:
        tat[i] = ct[i] - at[i]
        wt[i] = tat[i] - bt[i]
```

```
    ttat += tat[i]
    twt += wt[i]

// Calculate averages
avg_tat = ttat / n
avg_wt = twt / n

// Output results
Print "Priority scheduling:"
Print "PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT"
For i from 0 to n-1:
    Print proc_id[i], p[i], at[i], bt[i], ct[i], tat[i], wt[i], rt[i]

Print "Average turnaround time:", avg_tat
Print "Average waiting time:", avg_wt
```

```
Function Priority_Preemptive_Scheduling(n, at[], bt[], p[]):
    // Initialize process IDs and tracking arrays
    Initialize proc_id[] with 1 to n
    Initialize ct[], tat[], wt[], rt[], b[] with zeros
    Initialize m[] to track completed processes
    Initialize avg_tat, ttat, avg_wt, twt to 0.0
    Initialize c to 0

    // Input priorities, arrival, and burst times
    For i from 0 to n-1:
        proc_id[i] = i + 1
        m[i] = 0
        Input p[i], at[i], bt[i]
        b[i] = bt[i]
        rt[i] = -1

    // Sort processes by priority
    Sort proc_id[], p[], at[], bt[], b[] based on priority

    count = 0
    priority = p[0]

    While count < n:
        x = -1
        // Find the highest priority process available
        For i from 0 to n-1:
            If at[i] <= c and p[i] >= priority and b[i] > 0:
                x = i
                priority = p[i]

        // Update response time and burst time
        If rt[x] == -1:
            rt[x] = c - at[x]
        b[x] -= 1
        c += 1

        // Check for process completion
        If b[x] == 0:
            count += 1
            ct[x] = c
            m[x] = 1

    // Calculate turnaround and waiting times
    For i from 0 to n-1:
        tat[i] = ct[i] - at[i]
        wt[i] = tat[i] - bt[i]
        ttat += tat[i]
        twt += wt[i]
```

```
// Calculate averages
avg_tat = ttat / n
avg_wt = twt / n

// Output results
Print "Priority scheduling (Preemptive):"
Print "PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT"
For i from 0 to n-1:
   Print proc_id[i], p[i], at[i], bt[i], ct[i], tat[i], wt[i], rt[i]

Print "Average turnaround time:", avg_tat
Print "Average waiting time:", avg_wt
```

```
Function RoundRobin_Scheduling(n, at[], bt[], t):
    // Initialize process IDs and time tracking
    Initialize proc_id[] with 1 to n
    Initialize ct[], tat[], wt[], rt[], b[] with zeros
    Initialize m[] to track completed processes
    Initialize avg_tat, ttat, avg_wt, twt to 0.0
    Initialize c to 0

    // Input arrival and burst times
    For i from 0 to n-1:
        Input at[i], bt[i]
        b[i] = bt[i]
        m[i] = 0
        rt[i] = -1

    // Sort processes by arrival time
    Sort proc_id[], at[], bt[], b[] based on at

    // Initialize queue
    Initialize queue q[100] and set front f, rear r to 0
    q[0] = proc_id[0]
    count = 0

    While f >= 0:
        p = q[f++]
        i = 0
        While p != proc_id[i]:
            i++

        // Process execution
        If b[i] >= t:
            rt[i] = rt[i] == -1 ? c : rt[i]
            b[i] -= t
            c += t
        Else:
            rt[i] = rt[i] == -1 ? c : rt[i]
            c += b[i]
            b[i] = 0

        m[0] = 1

        // Add new processes to the queue
        For j from 0 to n-1:
            If at[j] <= c and proc_id[j] != p and m[j] != 1:
                q[++r] = proc_id[j]
                m[j] = 1

        // Check for completion
```

```
    If b[i] == 0:
        ct[i] = c
    Else:
        q[++r] = proc_id[i]

    If f > r:
        f = -1

// Calculate turnaround and response times
For i from 0 to n-1:
    tat[i] = ct[i] - at[i]
    rt[i] = rt[i] - at[i]
    wt[i] = tat[i] - bt[i]
    ttat += tat[i]
    twt += wt[i]

// Compute averages
avg_tat = ttat / n
avg_wt = twt / n

// Output results
Print "RRS scheduling:"
Print "PID\tAT\tBT\tCT\tTAT\tWT\tRT"
For i from 0 to n-1:
    Print proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i], rt[i]

Print "Average turnaround time:", avg_tat
Print "Average waiting time:", avg_wt
```

```
Function RateMonotonic_Scheduling(n, b[], pt[]):
   // Initialize process IDs and remaining burst times
   Initialize proc[] with 1 to n
   Initialize rem[] for remaining burst times
   Initialize l for LCM of periods
   Initialize sum for feasibility test

   // Input burst times and periods
   For i from 0 to n-1:
      Input b[i]          // Burst time
      rem[i] = b[i]       // Set remaining time
      Input pt[i]         // Period

   // Sort processes based on period
   Call sort(proc, b, pt, n)

   // Calculate LCM of periods
   l = lcmul(pt, n)

   // Display process details
   Print "Rate Monotone Scheduling:"
   Print "PID\tBurst\tPeriod"
   For i from 0 to n-1:
      Print proc[i], b[i], pt[i]

   // Feasibility test
   sum = 0.0
   For i from 0 to n-1:
      sum += b[i] / pt[i]

   rhs = n * (2^(1/n) - 1)
   Print "sum <= rhs =>", (sum <= rhs)
   If sum > rhs:
      Exit  // Not schedulable

   Print "Scheduling occurs for", l, "ms"

   // Rate Monotonic Scheduling
   time = 0
   prev = 0

   While time < l:
      f = 0
      For i from 0 to n-1:
         If time % pt[i] == 0:
            rem[i] = b[i]  // Reset remaining time

         If rem[i] > 0:
            If prev != proc[i]:
```

```
        Print time, "ms onwards: Process", proc[i], "running"
        prev = proc[i]

      rem[i]--  // Decrease remaining time
      f = 1
      Break

If not f:
   time++  // Idle time
```

```
Function EDF_Scheduling(n, b[], d[], pt[]):
    // Initialize process IDs and remaining burst times
    Initialize proc[] with 1 to n
    Initialize rem[] to store remaining burst times
    Initialize nextDeadlines[] to store next deadlines

    // Input burst times, deadlines, and periods
    For i from 0 to n-1:
        Input b[i]          // Burst time
        rem[i] = b[i]       // Set remaining time
        Input d[i]          // Deadline
        Input pt[i]         // Period

    // Sort processes based on deadlines
    Call sort(proc, d, b, pt, n)

    // Calculate LCM of periods
    l = lcmul(pt, n)

    // Display process details
    Print "Earliest Deadline Scheduling:"
    Print "PID\tBurst\tDeadline\tPeriod"
    For i from 0 to n-1:
        Print proc[i], b[i], d[i], pt[i]

    Print "Scheduling occurs for", l, "ms"

    // Initialize deadlines and remaining times
    For i from 0 to n-1:
        nextDeadlines[i] = d[i]
        rem[i] = b[i]

    time = 0
    prev = 0

    While time < l:
        // Update deadlines and remaining times at each period
        For i from 0 to n-1:
            If time % pt[i] == 0 and time != 0:
                nextDeadlines[i] = time + d[i]  // Update deadline
                rem[i] = b[i]                   // Reset remaining time

        // Find the task with the earliest deadline
        minDeadline = l + 1
        taskToExecute = -1
        For i from 0 to n-1:
            If rem[i] > 0 and nextDeadlines[i] < minDeadline:
```

```
        minDeadline = nextDeadlines[i]
        taskToExecute = i

// Execute the task with the earliest deadline
If taskToExecute != -1:
    Print time, "ms: Task", proc[taskToExecute], "is running."
    rem[taskToExecute]--  // Decrease remaining time
Else:
    Print time, "ms: CPU is idle."

time++  // Move to the next time unit
```

```
PRODUCER CONSUMER:
Initialize:
   mutex = 1
   full = 0
   empty = 5
   x = 0

Function wait():
   mutex = mutex - 1

Function signal():
   mutex = mutex + 1

Function producer():
   wait()
   full = full + 1
   empty = empty - 1
   x = x + 1
   Print "Producer has produced: Item", x
   signal()

Function consumer():
   wait()
   full = full - 1
   empty = empty + 1
   Print "Consumer has consumed: Item", x
   x = x - 1
   signal()

Function main():
   Print "Enter 1. Producer 2. Consumer 3. Exit"
   While true:
      Print "Enter your choice:"
      Input ch
      Switch ch:
         Case 1:
          If mutex == 1 and empty != 0:
             Call producer()
          Else:
             Print "Buffer is full!"
       Case 2:
          If mutex == 1 and full != 0:
             Call consumer()
          Else:
             Print "Buffer is empty!"
       Case 3:
          Exit
       Default:
          Print "Invalid choice!"
```

DINING PHILOSOPHER:

```
Function calculateNeed(P, R, need, max, allot):
    For each process i:
        For each resource j:
            need[i][j] = max[i][j] - allot[i][j]

Function isSafe(P, R, processes[], avail[], max, allot):
    Call calculateNeed(P, R, need, max, allot)
    Initialize finish[P] to false
    Initialize work = avail

    While count < P:
        found = false
        For each process p:
            If need[p] <= work:
                Update work with allot[p]
                safeSeq[count] = p
                finish[p] = true
                found = true

        If not found:
            Print "Not in safe state"
            Return false

    Print "Safe state:", safeSeq
    Return true

Function main():
    Input P, R
    For each process i:
        Input allot[i] and max[i]
    Input avail

    Call isSafe(P, R, processes, avail, max, allot)

    Print allocation details

Return 0
```

BANKERS ALGORITHM:

```
Function calculateNeed(P, R, need, max, allot):
    For each process i:
        For each resource j:
            need[i][j] = max[i][j] - allot[i][j]

Function isSafe(P, R, processes[], avail[], max, allot):
    Call calculateNeed(P, R, need, max, allot)
    Initialize finish[P] to false
    Initialize work = avail
    count = 0

    While count < P:
        found = false
        For each process p:
            If not finished[p]:
                If need[p] <= work:
                    Update work with allot[p]
                    safeSeq[count] = p
                    finish[p] = true
                    found = true
                    count++

        If not found:
            Print "Not in safe state"
            Return false

    Print "Safe state with sequence:", safeSeq
    Return true

Function main():
    Input P, R
    Initialize processes, avail, max, allot
    For each process i:
        Input allot[i] and max[i]
    Input avail

    Call isSafe(P, R, processes, avail, max, allot)

    Print allocation details

Return 0
```

DEADLOCK DETECTION:

```
Function firstFit(nb, nf, b[], f[])
   Initialize frag[MAX], bf[MAX], ff[MAX] to 0
   For each file f[i]:
      For each block b[j]:
         If block is free:
            Calculate temp = b[j] - f[i]
            If temp >= 0:
               Allocate block and record frag
               Break
   Print results

Function bestFit(nb, nf, b[], f[])
   Initialize frag[MAX], bf[MAX], ff[MAX] to 0
   For each file f[i]:
      lowest = infinity
      For each block b[j]:
         If block is free:
            Calculate temp
            If temp < lowest:
               Allocate block and update lowest
      Record frag
   Print results

Function worstFit(nb, nf, b[], f[])
   Initialize frag[MAX], bf[MAX], ff[MAX] to 0
   For each file f[i]:
      highest = 0
      For each block b[j]:
         If block is free:
            Calculate temp
            If temp > highest:
               Allocate block and update highest
      Record frag
   Print results

Function main()
   Input nb, nf, sizes of blocks and files
   Copy sizes to b1, b2, b3
   Call firstFit, bestFit, worstFit
Return 0
```

CONTIGIOUS MEMORY ALLOCATION:

```
Function firstFit(nb, nf, b[], f[])
   Declare frag[MAX], bf[MAX] initialized to 0, ff[MAX] initialized to 0
   For i from 1 to nf:
      For j from 1 to nb:
         If bf[j] != 1:
            temp = b[j] - f[i]
            If temp >= 0:
               ff[i] = j
               frag[i] = temp
               bf[j] = 1
               Break
   Print allocation results

Function bestFit(nb, nf, b[], f[])
   Declare frag[MAX], bf[MAX] initialized to 0, ff[MAX] initialized to 0
   For i from 1 to nf:
      lowest = 10000
      For j from 1 to nb:
         If bf[j] != 1:
            temp = b[j] - f[i]
            If temp >= 0 and lowest > temp:
               ff[i] = j
               lowest = temp
      frag[i] = lowest
      bf[ff[i]] = 1
   Print allocation results

Function worstFit(nb, nf, b[], f[])
   Declare frag[MAX], bf[MAX] initialized to 0, ff[MAX] initialized to 0
   For i from 1 to nf:
      highest = 0
      For j from 1 to nb:
         If bf[j] != 1:
            temp = b[j] - f[i]
            If temp >= 0 and highest < temp:
               ff[i] = j
               highest = temp
      frag[i] = highest
      bf[ff[i]] = 1
   Print allocation results

Function main()
   Declare b[MAX], f[MAX], nb, nf
   Input nb, nf, sizes of blocks and files
   Copy block sizes to b1, b2, b3
   Call firstFit, bestFit, worstFit
Return 0
```

PAGE REPLACEMENT ALGORITHM:

```
FUNCTION isPagePresent(frames, n, page)
   FOR each frame in frames
      IF frame == page THEN RETURN 1
   RETURN 0

FUNCTION printFrames(frames, n)
   PRINT each frame or "-" if empty

FUNCTION fifoPageReplacement(pages, numPages, numFrames)
   INITIALIZE frames to -1
   SET front = 0, pageFaults = 0
   FOR each page in pages
      IF page not in frames THEN
         frames[front] = page
         front = (front + 1) MOD numFrames
         INCREMENT pageFaults
      PRINT frames
   PRINT total page faults

FUNCTION findOptimalReplacementIndex(pages, numPages, frames, numFrames,
currentIndex)
   FIND frame to replace using future use
   RETURN index

FUNCTION optPageReplacement(pages, numPages, numFrames)
   INITIALIZE frames to -1
   FOR each page in pages
      IF page not in frames THEN
         REPLACE using optimal strategy
      PRINT frames
   PRINT total page faults

FUNCTION lruPageReplacement(pages, numPages, numFrames)
   INITIALIZE frames and timestamps to -1
   FOR each page in pages
      IF page not in frames THEN
         REPLACE using LRU strategy
      UPDATE timestamp
      PRINT frames
   PRINT total page faults

MAIN
   INPUT numFrames, numPages, pages
   CALL fifoPageReplacement, optPageReplacement, lruPageReplacement
```