write a C program to simulate the CPU scheduling algorighn - round robin

Code::

```
#include <stdio.h>

struct Process {
    int pid;    // Process ID
    int bt;     // Burst Time
    int wt;     // Waiting Time
    int tat;    // Turnaround Time
};

void findWaitingTime(struct Process proc[], int n, int quantum) {
    int remaining_bt[n];
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = proc[i].bt;
        proc[i].wt = 0;
    }

    int time = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0) {
                done = 0;
                if (remaining_bt[i] > quantum) {
                    time += quantum;
                    remaining_bt[i] -= quantum;
                } else {
                    time += remaining_bt[i];
                    proc[i].wt = time - proc[i].bt;
                    remaining_bt[i] = 0;
                }
            }
        }
        if (done == 1)
            break;
    }
}

void findTurnaroundTime(struct Process proc[], int n) {
```

```
    for (int i = 0; i < n; i++) {
        proc[i].tat = proc[i].bt + proc[i].wt;
    }
}

void findAvgTime(struct Process proc[], int n) {
    int total_wt = 0, total_tat = 0;

    for (int i = 0; i < n; i++) {
        total_wt += proc[i].wt;
        total_tat += proc[i].tat;
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
}

void printResults(struct Process proc[], int n) {
    printf("\nPID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].bt, proc[i].wt, proc[i].tat);
    }
}

int main() {
    int n, quantum;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    struct Process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for Process %d: ", proc[i].pid);
        scanf("%d", &proc[i].bt);
    }

    printf("Enter time quantum: ");
    scanf("%d", &quantum);

    findWaitingTime(proc, n, quantum);
    findTurnaroundTime(proc, n);
    findAvgTime(proc, n);
```

```
    printResults(proc, n);

    return 0;
}
```

Input:
Enter number of processes: 4
Enter burst time for Process 1: 10
Enter burst time for Process 2: 5
Enter burst time for Process 3: 8
Enter burst time for Process 4: 6
Enter time quantum: 4

write a c program to inmplement longest job first algorithm

code:::

```
#include <stdio.h>

struct Process {
    int pid;    // Process ID
    int bt;     // Burst Time
    int wt;     // Waiting Time
    int tat;    // Turnaround Time
};

void findWaitingTime(struct Process proc[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[i].bt < proc[j].bt) {
                // Swap burst time
                temp = proc[i].bt;
                proc[i].bt = proc[j].bt;
                proc[j].bt = temp;

                // Swap Process IDs for maintaining the correct order
                temp = proc[i].pid;
                proc[i].pid = proc[j].pid;
                proc[j].pid = temp;
```

```
            }
        }
    }

    proc[0].wt = 0;
    for (int i = 1; i < n; i++) {
        proc[i].wt = proc[i - 1].bt + proc[i - 1].wt;
    }
}

void findTurnaroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].tat = proc[i].bt + proc[i].wt;
    }
}

void findAvgTime(struct Process proc[], int n) {
    int total_wt = 0, total_tat = 0;

    for (int i = 0; i < n; i++) {
        total_wt += proc[i].wt;
        total_tat += proc[i].tat;
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
}

void printResults(struct Process proc[], int n) {
    printf("\nPID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].bt, proc[i].wt, proc[i].tat);
    }
}

int main() {
    int n;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    struct Process proc[n];

    for (int i = 0; i < n; i++) {
```

```
        proc[i].pid = i + 1;
        printf("Enter burst time for Process %d: ", proc[i].pid);
        scanf("%d", &proc[i].bt);
    }

    // Implement Longest Job First scheduling
    findWaitingTime(proc, n);
    findTurnaroundTime(proc, n);
    findAvgTime(proc, n);
    printResults(proc, n);

    return 0;
}
```

input:

```
Enter number of processes: 4
Enter burst time for Process 1: 6
Enter burst time for Process 2: 8
Enter burst time for Process 3: 7
Enter burst time for Process 4: 3
```

Output::

```
Average waiting time = 10.50
Average turnaround time = 17.25

PID     Burst Time   Waiting Time   Turnaround Time
2       8            0              8
3       7            8              15
1       6            15             21
4       3            21             24
```

write a c program to simulate the following contiguous memory allocation :
worst fit
best fit
first fit

Code::

```
#include <stdio.h>

#define MAX_BLOCKS 20
#define MAX_PROCESSES 10

void firstFit(int blocks[], int blockCount, int processes[], int processCount) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < processCount; i++) {
        allocation[i] = -1;  // Initially no process is allocated
        for (int j = 0; j < blockCount; j++) {
            if (blocks[j] >= processes[i]) {
                allocation[i] = j; // Allocate block j
                blocks[j] -= processes[i]; // Reduce block size
                break;
            }
        }
    }

    // Print the allocation result
    printf("\nFirst Fit Allocation:\n");
    for (int i = 0; i < processCount; i++) {
        if (allocation[i] != -1)
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        else
            printf("Process %d not allocated\n", i + 1);
    }
}

void bestFit(int blocks[], int blockCount, int processes[], int processCount) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < processCount; i++) {
        allocation[i] = -1;
        int bestIdx = -1;
        for (int j = 0; j < blockCount; j++) {
            if (blocks[j] >= processes[i]) {
                if (bestIdx == -1 || blocks[bestIdx] > blocks[j]) {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
```

```
            blocks[bestIdx] -= processes[i];
        }
    }

    // Print the allocation result
    printf("\nBest Fit Allocation:\n");
    for (int i = 0; i < processCount; i++) {
        if (allocation[i] != -1)
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        else
            printf("Process %d not allocated\n", i + 1);
    }
}

void worstFit(int blocks[], int blockCount, int processes[], int processCount) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < processCount; i++) {
        allocation[i] = -1;
        int worstIdx = -1;
        for (int j = 0; j < blockCount; j++) {
            if (blocks[j] >= processes[i]) {
                if (worstIdx == -1 || blocks[worstIdx] < blocks[j]) {
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blocks[worstIdx] -= processes[i];
        }
    }

    // Print the allocation result
    printf("\nWorst Fit Allocation:\n");
    for (int i = 0; i < processCount; i++) {
        if (allocation[i] != -1)
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        else
            printf("Process %d not allocated\n", i + 1);
    }
}

int main() {
    int blocks[MAX_BLOCKS], processes[MAX_PROCESSES];
```

```
int blockCount, processCount;

// Get the number of blocks and processes
printf("Enter number of memory blocks: ");
scanf("%d", &blockCount);
printf("Enter number of processes: ");
scanf("%d", &processCount);

// Get the size of each memory block
printf("Enter the size of each memory block:\n");
for (int i = 0; i < blockCount; i++) {
    printf("Block %d: ", i + 1);
    scanf("%d", &blocks[i]);
}

// Get the size of each process
printf("Enter the size of each process:\n");
for (int i = 0; i < processCount; i++) {
    printf("Process %d: ", i + 1);
    scanf("%d", &processes[i]);
}

// Make copies of the block array for each algorithm
int blocks1[MAX_BLOCKS], blocks2[MAX_BLOCKS], blocks3[MAX_BLOCKS];
for (int i = 0; i < blockCount; i++) {
    blocks1[i] = blocks[i];
    blocks2[i] = blocks[i];
    blocks3[i] = blocks[i];
}

// Call allocation functions for First Fit, Best Fit, and Worst Fit
firstFit(blocks1, blockCount, processes, processCount);
for (int i = 0; i < blockCount; i++) {
    blocks1[i] = blocks[i];
}

bestFit(blocks2, blockCount, processes, processCount);
for (int i = 0; i < blockCount; i++) {
    blocks2[i] = blocks[i];
}

worstFit(blocks3, blockCount, processes, processCount);

return 0;
```

}

Input::

Enter number of memory blocks: 5
Enter number of processes: 3
Enter the size of each memory block:
Block 1: 10
Block 2: 20
Block 3: 30
Block 4: 40
Block 5: 50
Enter the size of each process:
Process 1: 12
Process 2: 18
Process 3: 30

Output:
First Fit Allocation:
Process 1 allocated to Block 2
Process 2 allocated to Block 3
Process 3 allocated to Block 5

Best Fit Allocation:
Process 1 allocated to Block 2
Process 2 allocated to Block 3
Process 3 allocated to Block 5

Worst Fit Allocation:
Process 1 allocated to Block 5
Process 2 allocated to Block 4
Process 3 allocated to Block 3

write a c program to simulate the cpu scheduling algrorithm first come first serve

```
#include <stdio.h>

struct Process {
    int pid;    // Process ID
    int bt;     // Burst Time
    int wt;     // Waiting Time
    int tat;    // Turnaround Time
};

void findWaitingTime(struct Process proc[], int n) {
    proc[0].wt = 0; // The first process doesn't wait
    for (int i = 1; i < n; i++) {
        proc[i].wt = proc[i - 1].bt + proc[i - 1].wt;
    }
}

void findTurnaroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].tat = proc[i].bt + proc[i].wt;
    }
}

void findAvgTime(struct Process proc[], int n) {
    int total_wt = 0, total_tat = 0;

    for (int i = 0; i < n; i++) {
        total_wt += proc[i].wt;
        total_tat += proc[i].tat;
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
}

void printResults(struct Process proc[], int n) {
    printf("\nPID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].bt, proc[i].wt, proc[i].tat);
    }
}

int main() {
    int n;
```

```
    printf("Enter number of processes: ");
    scanf("%d", &n);

    struct Process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for Process %d: ", proc[i].pid);
        scanf("%d", &proc[i].bt);
    }

    // Implement First Come First Serve scheduling
    findWaitingTime(proc, n);
    findTurnaroundTime(proc, n);
    findAvgTime(proc, n);
    printResults(proc, n);

    return 0;
}
```

Input :
Enter number of processes: 4
Enter burst time for Process 1: 6
Enter burst time for Process 2: 8
Enter burst time for Process 3: 7
Enter burst time for Process 4: 3

Outpt:
Average waiting time = 6.00
Average turnaround time = 13.50

PID	Burst Time	Waiting Time	Turnaround Time
1	6	0	6
2	8	6	14
3	7	14	21
4	3	21	24

Write a c program for bankers algorithm for deadlock avoidance

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

// Function to calculate if the system is in a safe state
bool isSafe(int processes[], int avail[], int max[][MAX_RESOURCES], int
allot[][MAX_RESOURCES], int n, int m) {
    int need[n][m];
    bool finish[n];
    int safeSeq[n];
    int work[m];
    int count = 0;

    // Calculate the need matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            need[i][j] = max[i][j] - allot[i][j];
        }
    }

    // Initialize finish and work arrays
    for (int i = 0; i < n; i++) {
        finish[i] = false;
    }
    for (int i = 0; i < m; i++) {
        work[i] = avail[i];
    }

    // Find a safe sequence
    while (count < n) {
        bool found = false;

        for (int p = 0; p < n; p++) {
            if (!finish[p]) {
                int j;
                for (j = 0; j < m; j++) {
                    if (need[p][j] > work[j]) {
                        break;
                    }
                }
                if (j == m) { // All resources can be allocated to process p
                    for (int k = 0; k < m; k++) {
```

```
                    work[k] += allot[p][k];  // Simulate the allocation of resources
                }
                safeSeq[count++] = p;
                finish[p] = true;
                found = true;
            }
        }
    }

    if (!found) {
        printf("System is in an unsafe state!\n");
        return false;
    }
  }

  // If the system is in a safe state
  printf("System is in a safe state. Safe Sequence: ");
  for (int i = 0; i < n; i++) {
    printf("P%d ", safeSeq[i]);
  }
  printf("\n");
  return true;
}

int main() {
  int processes[MAX_PROCESSES] = {0, 1, 2, 3, 4};
  int avail[MAX_RESOURCES] = {3, 3, 2};  // Available resources

  // Maximum resources required by each process
  int max[MAX_PROCESSES][MAX_RESOURCES] = {
    {7, 5, 3},  // Process 0
    {3, 2, 2},  // Process 1
    {9, 0, 2},  // Process 2
    {2, 2, 2},  // Process 3
    {4, 3, 3}   // Process 4
  };

  // Resources allocated to each process
  int allot[MAX_PROCESSES][MAX_RESOURCES] = {
    {0, 1, 0},  // Process 0
    {2, 0, 0},  // Process 1
    {3, 0, 2},  // Process 2
    {2, 1, 1},  // Process 3
    {0, 0, 2}   // Process 4
```

```
    };

    // Call the Banker's Algorithm to check if the system is in a safe state
    isSafe(processes, avail, max, allot, 5, 3);

    return 0;
}
```

input

Available Resources: [3, 3, 2]
Maximum Resources for processes:
P0: [7, 5, 3]
P1: [3, 2, 2]
P2: [9, 0, 2]
P3: [2, 2, 2]
P4: [4, 3, 3]

Allocated Resources for processes:
P0: [0, 1, 0]
P1: [2, 0, 0]
P2: [3, 0, 2]
P3: [2, 1, 1]
P4: [0, 0, 2]

Output:

System is in a safe state. Safe Sequence: P1 P3 P4 P0 P2

Write a c program shortest job first

```
#include <stdio.h>

struct Process {
    int pid;    // Process ID
    int bt;     // Burst Time
    int wt;     // Waiting Time
    int tat;    // Turnaround Time
};

void findWaitingTime(struct Process proc[], int n) {
    proc[0].wt = 0; // The first process doesn't wait
    for (int i = 1; i < n; i++) {
        proc[i].wt = proc[i - 1].bt + proc[i - 1].wt;
    }
}

void findTurnaroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].tat = proc[i].bt + proc[i].wt;
    }
}

void findAvgTime(struct Process proc[], int n) {
    int total_wt = 0, total_tat = 0;

    for (int i = 0; i < n; i++) {
        total_wt += proc[i].wt;
        total_tat += proc[i].tat;
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
}

void printResults(struct Process proc[], int n) {
    printf("\nPID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].bt, proc[i].wt, proc[i].tat);
    }
}

// Function to sort the processes by burst time (for SJF)
void sortByBurstTime(struct Process proc[], int n) {
    struct Process temp;
```

```
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[i].bt > proc[j].bt) {
                temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }
}

int main() {
    int n;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    struct Process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for Process %d: ", proc[i].pid);
        scanf("%d", &proc[i].bt);
    }

    // Sort processes by burst time to implement Shortest Job First
    sortByBurstTime(proc, n);

    // Calculate Waiting Time, Turnaround Time, and Average Times
    findWaitingTime(proc, n);
    findTurnaroundTime(proc, n);
    findAvgTime(proc, n);

    // Print the results
    printResults(proc, n);

    return 0;
}
```

```
Enter number of processes: 4
Enter burst time for Process 1: 6
```

Enter burst time for Process 2: 8
Enter burst time for Process 3: 7
Enter burst time for Process 4: 3

Average waiting time = 6.00
Average turnaround time = 13.50

PID	Burst Time	Waiting Time	Turnaround Time
4	3	0	3
1	6	3	9
3	7	9	16
2	8	16	24