```c
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>


int main() {

    pid_t pid;


    // Create a child process

    pid = fork();


    if (pid < 0) {

        // Fork failed

        fprintf(stderr, "Fork Failed\n");

        return 1;

    } else if (pid == 0) {

        // Child process

        printf("This is the child process. My PID is %d\n", getpid());

        printf("Child's parent PID is %d\n", getppid());

    } else {

        // Parent process

        printf("This is the parent process. My PID is %d\n", getpid());

        printf("Parent's child PID is %d\n", pid);

    }


    return 0;

}


//practical3
```

```
//138

//WAP to find sum of n numbers using thread program.



import java.util.Scanner;


class SumCalculator extends Thread {
    int n;
    int sum;
    int esum;
    int osum;

    public SumCalculator(int n) {
        this.n = n;
        this.sum = 0;
        this.esum = 0;
        this.osum = 0;
    }

    public void run() {
        for (int i = 1; i <= n; i++) {
            sum += i;
            if (i % 2 == 0) {
                esum += i;
            } else {
                osum += i;
            }
        }
    }
    public int getSum() {
        return sum;
    }

    public int getEvenSum() {
        return esum;
    }

    public int getOddSum() {
        return osum;
    }
}

public class Even_odd {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter number of elements:");
        int n = scanner.nextInt();

        SumCalculator sumcal1 = new SumCalculator(n);
        SumCalculator sumcal2 = new SumCalculator(n);

        sumcal1.start();
        sumcal2.start();

        try {
            sumcal1.join();
            sumcal2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Sum of all numbers: " + sumcal1.getSum());
        System.out.println("Sum of even numbers: " + sumcal1.getEvenSum());
        System.out.println("Sum of odd numbers: " + sumcal2.getOddSum());

        scanner.close();
    }
}

//138
//practical4
// priority  non primitive cpu scheduling algorithm

#include <stdio.h>
#include <limits.h>

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

    int arrivaltime[n];
    int bursttime[n];
    int priority[n];
    int waitingTime[n];
    int turnaroundTime[n];

    printf("Enter arrival time, burst time, and priority for each process:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Process %d:\n", i + 1);
        printf("Arrival time: ");
        scanf("%d", &arrivaltime[i]);
        printf("Burst time: ");
        scanf("%d", &bursttime[i]);
        printf("Priority: ");
        scanf("%d", &priority[i]);
    }

    int CPU = 0;
    int NoP = n;
    int remainingTime[n];
    int completed[n];
    for (int i = 0; i < n; ++i)
    {
        remainingTime[i] = bursttime[i];
        completed[i] = 0;
    }

    while (NoP > 0 && CPU <= 1000)    {
        int selectedProcess = -1;
        int highestPriority = INT_MAX;
```

```
        for (int i = 0; i < n; i++)
        {
            if (!completed[i] && arrivaltime[i] <= CPU && priority[i] < highestPriority)
            {
                highestPriority = priority[i];
                selectedProcess = i;
            }
        }

        if (selectedProcess == -1)
        {
            CPU++;
            continue;
        }
        else
        {
            remainingTime[selectedProcess]--;

            if (remainingTime[selectedProcess] == 0)
            {
                NoP--;
                completed[selectedProcess] = 1;

                waitingTime[selectedProcess] = CPU - arrivaltime[selectedProcess] - bursttime[selectedProcess]
+ 1;
                if (waitingTime[selectedProcess] < 0)
                    waitingTime[selectedProcess] = 0;

                turnaroundTime[selectedProcess] = CPU - arrivaltime[selectedProcess] + 1;
                CPU++;
            }
            else
            {
                CPU++;
            }
        }
    }

printf("\nProcess_Number\tBurst_Time\tPriority\tArrival_Time\tWaiting_Time\tTurnaround_Time\n\n");
    for (int i = 0; i < n; i++)
    {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, bursttime[i], priority[i], arrivaltime[i], waitingTime[i], turnaroundTime[i]);
    }

    float AvgWT = 0;
    float AVGTaT = 0;
    for (int i = 0; i < n; i++)
    {
        AvgWT += waitingTime[i];
        AVGTaT += turnaroundTime[i];
    }

    printf("Average waiting time = %.2f\n", AvgWT / n);
    printf("Average turnaround time = %.2f\n", AVGTaT / n);
    return 0;
}

#include <iostream>
using namespace std;

int main()
{
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    int *arrivaltime = new int[n];
    int *bursttime = new int[n];
    int *priority = new int[n];
    int *waitingTime = new int[n];
    int *turnaroundTime = new int[n];

    cout << "Enter arrival time, burst time, and priority for each process:\n";
    for (int i = 0; i < n; i++)
    {
        cout << "Process " << i + 1 << ":\n";
        cout << "Arrival time: ";
        cin >> arrivaltime[i];
        cout << "Burst time: ";
        cin >> bursttime[i];
        cout << "Priority: ";
        cin >> priority[i];
    }
```

```
int CPU = 0;


int NoP = n;

int *PPt = new int[n];


int LAT = 0;

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

    if (arrivaltime[i] > LAT)

        LAT = arrivaltime[i];


int INT_MAX ;

int MIN_P = INT_MAX;

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

    if (priority[i] < MIN_P)

        MIN_P = priority[i];


int *ATt = new int[n];

int *P1 = new int[n];

int *P2 = new int[n];


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

{

    PPt[i] = priority[i];

    ATt[i] = arrivaltime[i];

    P1[i] = priority[i];

    P2[i] = priority[i];

}
```

```
while (NoP > 0 && CPU <= 1000)
{
    int ATi = 0;
    int j = -1;

    for (int i = 0; i < n; i++)
    {
        if ((ATt[i] <= CPU) && (ATt[i] != (LAT + 10)))
        {
            if (PPt[i] != (MIN_P - 1))
            {
                P2[i] = PPt[i];
                j = 1;

                if (P2[i] > P1[ATi])
                {
                    j = 1;
                    ATi = i;
                    P1[ATi] = PPt[i];
                    P2[ATi] = PPt[i];
                }
            }
        }
    }

    if (j == -1)
    {
        CPU++;
        continue;
    }
    else
    {
        waitingTime[ATi] = CPU - ATt[ATi];
        CPU += bursttime[ATi];
        turnaroundTime[ATi] = CPU - ATt[ATi];
        ATt[ATi] = LAT + 10;
        j = -1;
        PPt[ATi] = MIN_P - 1;
        NoP--;
    }
}

cout << "\nProcess_Number\tBurst_Time\tPriority\tArrival_Time\tWaiting_Time\tTurnaround_Time\n\n";
for (int i = 0; i < n; i++)
{
    cout << "P" << i + 1 << "\t\t" << bursttime[i] << "\t\t" << priority[i] << "\t\t" << arrivaltime[i] << "\t\t" << waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
}

float AvgWT = 0;
float AVGTaT = 0;
for (int i = 0; i < n; i++)
{
    AvgWT += waitingTime[i];
    AVGTaT += turnaroundTime[i];
}
cout << "Average waiting time = " << AvgWT / n << endl;
cout << "Average turnaround time = " << AVGTaT / n << endl;



delete[] arrivaltime;
delete[] bursttime;
delete[] priority;
delete[] waitingTime;
delete[] turnaroundTime;
delete[] PPt;
delete[] ATt;
delete[] P1;
delete[] P2;
return 0;
}
//fcfs non-preemptive

#include<stdio.h>
int main()
{
int n,at[10],bt[10],wt[10],tat[10],ct[10],sum,i,j,k;
float totaltat=0,totalwt=0;
printf("enter the total number of processes:");
scanf("%d",&n);
printf("\nEnter The Process Arrival Time & Burst Time\n");
for(i=0;i < n;i++)
{       printf("Enter Arrival time of process[%d]:",i+1);
        scanf("%d",&at[i]);
       printf("Enter Burst time of process[%d]:",i+1);
```

```
       scanf("%d",&bt[i]);
  }
 /Calculate completion time of processes/
 sum=at[0];
 for(j=0;j < n;j++)
 {
     sum=sum+bt[j];
     ct[j]=sum;
 }
 /*Calculate Turn Around time */
 for(k=0;k  < n;k++)
 {
      tat[k]=ct[k]-at[k];
      totaltat=totaltat+tat[k];
 }
   /*  Calculate Waiting time  */
 for(k=0;k  < n;k++)
 {
      wt[k]=tat[k]-bt[k];
  totalwt=totalwt+wt[k];
 }

  printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n\n\n");
 for(i=0;i  <n;i++)
 {
   printf("\nP%d\t %d\t %d\t %d\t %d\t %d\t\n",i+1,at[i],bt[i],ct[i],tat[i],wt[i]);
}
  printf("\nAverage TurnaroundTime:%f\n",totaltat/n);
  printf("\nAverage Waiting Time:%f",totalwt/n);
```

```
    return 0;
}
//SJF non preeemptive

#include <stdio.h>

typedef struct {
    int process_id;
    int burst_time;
    int waiting_time;
    int turnaround_time;
} Process;

void sortByBurstTime(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
```

```
scanf("%d", &n);

Process processes[n];
int total_waiting_time = 0, total_turnaround_time = 0;

printf("Enter the burst times of the processes:\n");
for (int i = 0; i < n; i++) {
    processes[i].process_id = i + 1;
    printf("Process %d Burst Time: ", i + 1);
    scanf("%d", &processes[i].burst_time);
}

// Sort processes by burst time
sortByBurstTime(processes, n);

// First process has zero waiting time
processes[0].waiting_time = 0;

// Calculate waiting time for each process
for (int i = 1; i < n; i++) {
    processes[i].waiting_time = 0;
    for (int j = 0; j < i; j++) {
        processes[i].waiting_time += processes[j].burst_time;
    }
    total_waiting_time += processes[i].waiting_time;
}

// Calculate turnaround time for each process
for (int i = 0; i < n; i++) {
    processes[i].turnaround_time = processes[i].burst_time + processes[i].waiting_time;
    total_turnaround_time += processes[i].turnaround_time;
  }

  // Display results
  printf("\nProcess\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", processes[i].process_id, processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time);
  }

  printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / n);
  printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);

  return 0;
}
//RR cpu
#include <iostream>
using namespace std;
struct Process {
 int id;
 int arrivalTime;
 int burstTime;
 int completionTime;
 int turnaroundTime;
 int waitingTime;
};
void calculateTimes(Process processes[], int n, int quantum) {
 int remainingTime[n];
```

```
  for (int i = 0; i < n; i++) {
    remainingTime[i] = processes[i].burstTime;
  }
  int currentTime = 0;
  bool allDone = false;
  while (!allDone) {
    allDone = true;
    for (int i = 0; i < n; i++) {
      if (remainingTime[i] > 0) {
        allDone = false;
        if (remainingTime[i] > quantum) {
          currentTime = currentTime + quantum;
          remainingTime[i] = remainingTime[i] - quantum;
        } else {
          currentTime = currentTime + remainingTime[i];
          processes[i].completionTime = currentTime;
          remainingTime[i] = 0;
        }
      }
    }
  }
}
void calculateTurnaroundTime(Process processes[], int n) {
  for (int i = 0; i < n; i++)
    processes[i].turnaroundTime =
        processes[i].completionTime - processes[i].arrivalTime;
}

void claculateWaitingTime(Process processes[], int n) {
  for (int i = 0; i < n; i++)
    processes[i].waitingTime =
        processes[i].turnaroundTime - processes[i].burstTime;
}

void printTable(Process processes[], int n) {
  cout << "------------------------------------------------------------------"
          "----------------------\n";
  cout << "| Process | Arrival Time | Burst Time | Completion Time | "
          "Turnaround Time | Waiting Time |\n";
  cout << "------------------------------------------------------------------"
          "----------------------\n";
  for (int i = 0; i < n; i++) {
    cout << "|    " << processes[i].id << "    |      "
         << processes[i].arrivalTime << "      |     " << processes[i].burstTime
         << "     |        " << processes[i].completionTime
         << "        |        " << processes[i].turnaroundTime
         << "         |      " << processes[i].waitingTime << "      |\n";
  }
  cout << "------------------------------------------------------------------"
          "----------------------\n";
}
int main() {
  int n, quantum;
  cout << "Enter The Number of Process";
  cin >> n;
  cout << "Enter The Time Quantum";
  cin >> quantum;
  Process processes[n];
  cout << "Enter process details:\n";
  for (int i = 0; i < n; i++) {
    cout << "Process " << i + 1 << ":\n";
    processes[i].id = i + 1;
    cout << "   Arrival Time: ";
    cin >> processes[i].arrivalTime;
    cout << "   Burst Time: ";
    cin >> processes[i].burstTime;
  }

  calculateTimes(processes, n, quantum);
  calculateTurnaroundTime(processes, n);
  claculateWaitingTime(processes, n);

  cout << "\nRound Robin Scheduling Results:\n";
  printTable(processes, n);

  return 0;
}
// SRTF Cpu

#include <stdio.h>
#include <limits.h>

typedef struct {
    int process_id;
    int burst_time;
    int arrival_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int completed;
} Process;

int main() {
    int n, time = 0, completed = 0, shortest = 0, min_time = INT_MAX;
    float total_waiting_time = 0, total_turnaround_time = 0;

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

    Process processes[n];

    printf("Enter the arrival times and burst times of the processes:\n");
    for (int i = 0; i < n; i++) {
        processes[i].process_id = i + 1;
        printf("Process %d Arrival Time: ", i + 1);
        scanf("%d", &processes[i].arrival_time);
        printf("Process %d Burst Time: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].completed = 0;
    }

    while (completed != n) {
        shortest = -1;
        min_time = INT_MAX;
        // Find the process with the shortest remaining time at the current time
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= time && !processes[i].completed && processes[i].remaining_time < min_time) {
                min_time = processes[i].remaining_time;
                shortest = i;
            }
        }

        // If no process is found, increment the time
        if (shortest == -1) {
            time++;
            continue;
        }

        // Execute the process with the shortest remaining time
        processes[shortest].remaining_time--;
        time++;

        // If the process is completed
        if (processes[shortest].remaining_time == 0) {
            processes[shortest].completed = 1;
            completed++;

            processes[shortest].turnaround_time = time - processes[shortest].arrival_time;
            processes[shortest].waiting_time = processes[shortest].turnaround_time - processes[shortest].burst_time;
            total_waiting_time += processes[shortest].waiting_time;
            total_turnaround_time += processes[shortest].turnaround_time;
        }
    }

    // Display results
    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_id, processes[i].arrival_time,
processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time);
    }

    printf("\nAverage Waiting Time: %.2f\n", total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);

    return 0;
}

// Bankers algorithm

#include <iostream>
using namespace std;

const int P = 5;

const int R = 3;

void calculateNeed(int need[P][R], int maxm[P][R],
                  int allot[P][R])
```

```
{
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)


            need[i][j] = maxm[i][j] - allot[i][j];
}

bool isSafe(int processes[], int avail[], int maxm[][R],
            int allot[][R])
{
    int need[P][R];

    calculateNeed(need, maxm, allot);

    bool finish[P] = {0};

    int safeSeq[P];

    int work[R];
    for (int i = 0; i < R; i++)
        work[i] = avail[i];



    int count = 0;
    while (count < P)
    {

        bool found = false;
        for (int p = 0; p < P; p++)
        {
            if (finish[p] == 0)
            {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R)
                {
                    for (int k = 0; k < R; k++)
                        work[k] += allot[p][k];

                    safeSeq[count++] = p;

                    finish[p] = 1;

                    found = true;
                }
            }
        }

        if (found == false)
        {
            cout << "System is not in safe state";
            return false;
        }
    }
```

```
  cout << "System is in safe state.\nSafe"
      " sequence is: ";
  for (int i = 0; i < P; i++)
    cout << safeSeq[i] << " ";

  return true;
}

int main()
{
  int processes[] = {0, 1, 2, 3, 4};

  int avail[] = {3, 3, 2};

  int maxm[][R] = {{7, 5, 3},
           {3, 2, 2},
           {9, 0, 2},
           {2, 2, 2},
           {4, 3, 3}};

  int allot[][R] = {{0, 1, 0},
            {2, 0, 0},
            {3, 0, 2},
            {2, 1, 1},
            {0, 0, 2}};

  isSafe(processes, avail, maxm, allot);

  return 0;
```

```c
}

// first, best , worst fit
#include <stdio.h>
#include <string.h> // For memcpy

#define N_BLOCKS 5 // Number of memory blocks
#define N_PROCESSES 4 // Number of processes

// Function to display the allocation
void displayAllocation(int processes[], int allocation[], int n) {
    printf("Process No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1); // Displayed as 1-indexed
        else
            printf("Not Allocated\n");
    }
}

void firstFit(int blockSize[], int processes[]) {
    int allocation[N_PROCESSES];
    memset(allocation, -1, sizeof(allocation)); // Initialize all allocations to -1

    for (int i = 0; i < N_PROCESSES; i++) {
        for (int j = 0; j < N_BLOCKS; j++) {
            if (blockSize[j] >= processes[i]) {
                allocation[i] = j; // Allocate block j to process i
                blockSize[j] -= processes[i]; // Reduce available memory in block j
                break; // Move to the next process
            }
        }
    }

    displayAllocation(processes, allocation, N_PROCESSES);
}

void bestFit(int blockSize[], int processes[]) {
    int allocation[N_PROCESSES];
    memset(allocation, -1, sizeof(allocation)); // Initialize all allocations to -1

    for (int i = 0; i < N_PROCESSES; i++) {
        int bestIdx = -1;
        for (int j = 0; j < N_BLOCKS; j++) {
            if (blockSize[j] >= processes[i]) {
                if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {
                    bestIdx = j;
                }
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx; // Allocate best fit block to process i
            blockSize[bestIdx] -= processes[i]; // Reduce available memory in the best fit block
        }
    }
    displayAllocation(processes, allocation, N_PROCESSES);
}

void worstFit(int blockSize[], int processes[]) {
    int allocation[N_PROCESSES];
    memset(allocation, -1, sizeof(allocation)); // Initialize all allocations to -1

    for (int i = 0; i < N_PROCESSES; i++) {
        int worstIdx = -1;
        for (int j = 0; j < N_BLOCKS; j++) {
            if (blockSize[j] >= processes[i]) {
                if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {
                    worstIdx = j;
                }
            }
        }

        if (worstIdx != -1) {
            allocation[i] = worstIdx; // Allocate worst fit block to process i
            blockSize[worstIdx] -= processes[i]; // Reduce available memory in the worst fit block
        }
    }

    displayAllocation(processes, allocation, N_PROCESSES);
}

int main() {
    int blockSize[N_BLOCKS] = {100, 500, 200, 300, 600};
    int processes[N_PROCESSES] = {212, 417, 112, 426};
    int blockCopy[N_BLOCKS];

    printf("First Fit Allocation:\n");
    memcpy(blockCopy, blockSize, sizeof(blockSize)); // Reset block sizes
    firstFit(blockCopy, processes);
    printf("\n");

    printf("Best Fit Allocation:\n");
    memcpy(blockCopy, blockSize, sizeof(blockSize)); // Reset block sizes
    bestFit(blockCopy, processes);
    printf("\n");

    printf("Worst Fit Allocation:\n");
    memcpy(blockCopy, blockSize, sizeof(blockSize)); // Reset block sizes
    worstFit(blockCopy, processes);
    printf("\n");

 return 0;
}

// LRU page replacement

#include <stdio.h>
#include <stdlib.h>

#define MAX_CAPACITY 100

void LRU(int capacity, int page_count) {
    int pages[MAX_CAPACITY];
    int frame[MAX_CAPACITY];
    int indexes[MAX_CAPACITY];
    int pageFaults = 0;
    int frameSize = 0;

    printf("Enter the page sequence:\n");
    for (int i = 0; i < page_count; i++) {
        scanf("%d", &pages[i]);
    }

    for (int i = 0; i < page_count; i++) {
        int j;
        for (j = 0; j < frameSize; j++) {
            if (frame[j] == pages[i]) {
                for (int k = j; k < frameSize - 1; k++) {
                    indexes[k] = indexes[k + 1];
                }
                indexes[frameSize - 1] = pages[i];
                break;
            }
        }
        if (j == frameSize) {
            if (frameSize < capacity) {
                frame[frameSize] = pages[i];
                indexes[frameSize] = pages[i];
                frameSize++;
            } else {
                int k;
                for (k = 0; k < capacity; k++) {
                    if (frame[k] == indexes[0])
                        break;
                }
                for (int l = 0; l < capacity - 1; l++) {
                    indexes[l] = indexes[l + 1];
                }
                indexes[capacity - 1] = pages[i];
                frame[k] = pages[i];
            }
            pageFaults++;
        }
    }
    printf("Number of Page Faults (LRU): %d\n", pageFaults);
}

int main() {
    int capacity, page_count;

    printf("Enter the capacity of the frame: ");
    scanf("%d", &capacity);
    printf("Enter the number of pages: ");
    scanf("%d", &page_count);

    LRU(capacity, page_count);

    return 0;
}

//Optimal page replacement
```

```
#include <stdio.h>

int findFarthest(int frame[], int future[], int current, int frame_size, int ref_size) {
    int max_distance = -1;
    int farthest_index = -1;

    for (int i = 0; i < frame_size; i++) {
        int j;
        for (j = current + 1; j < ref_size; j++) {
            if (frame[i] == future[j]) {
                if (j > max_distance) {
                    max_distance = j;
                    farthest_index = i;
                }
                break;
            }
        }

        // If the page is not going to be used again
        if (j == ref_size) {
            return i;
        }
    }

    // If all pages in frame will be used in the future, return the one farthest away
    return (farthest_index == -1) ? 0 : farthest_index;
}

int main() {
```

```
int ref_size, frame_size;

printf("Enter the number of reference pages: ");
scanf("%d", &ref_size);

int reference[ref_size];
printf("Enter the reference string:\n");
for (int i = 0; i < ref_size; i++) {
    scanf("%d", &reference[i]);
}

printf("Enter the number of frames: ");
scanf("%d", &frame_size);

int frame[frame_size];
int page_faults = 0;

// Initialize frame array with -1 (indicating empty)
for (int i = 0; i < frame_size; i++) {
    frame[i] = -1;
}

printf("\nReference String\tFrames\n");

for (int i = 0; i < ref_size; i++) {
    int page = reference[i];
    int found = 0;

    // Check if the page is already in the frame
```

```c
        for (int j = 0; j < frame_size; j++) {
            if (frame[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Find the frame to be replaced
            int replace_index = (i < frame_size) ? i : findFarthest(frame, reference, i, frame_size, ref_size);
            frame[replace_index] = page;
            page_faults++;
        }

        // Print current state of frames
        printf("%d\t\t\t", page);
        for (int j = 0; j < frame_size; j++) {
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

    printf("\nTotal Page Faults: %d\n", page_faults);

    return 0;
```

```
}

//FIFO page replacement


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

int main() {
   int ref_size, frame_size;

   // Input number of reference pages
   printf("Enter the number of reference pages: ");
   scanf("%d", &ref_size);

   int reference[ref_size];
   printf("Enter the reference string:\n");
   for (int i = 0; i < ref_size; i++) {
      scanf("%d", &reference[i]);
   }

   // Input number of frames
   printf("Enter the number of frames: ");
   scanf("%d", &frame_size);

   int frame[frame_size];
   int page_faults = 0;
   int index = 0; // Points to the oldest page in the frame
```

```
// Initialize frame array with -1 (indicating empty)
for (int i = 0; i < frame_size; i++) {
    frame[i] = -1;
}

printf("\nReference String\tFrames\n");

for (int i = 0; i < ref_size; i++) {
    int page = reference[i];
    bool found = false;

    // Check if the page is already in the frame
    for (int j = 0; j < frame_size; j++) {
        if (frame[j] == page) {
            found = true;
            break;
        }
    }

    if (!found) {
        // Replace the oldest page with the new page
        frame[index] = page;
        index = (index + 1) % frame_size;
        page_faults++;
    }

    // Print current state of frames
    printf("%d\t\t\t", page);
    for (int j = 0; j < frame_size; j++) {
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

    printf("\nTotal Page Faults: %d\n", page_faults);

    return 0;
}
```

```
//FCFS Disk Scheduling
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n;
    printf("Enter the number of disk movement: ");
    scanf("%d",&n);
    int initial_head;
    printf("enter initial head movement");
    scanf("%d",&initial_head);
    int disk[n];
    for(int i=0;i<n;i++){
        scanf("%d",&disk[i]);
    }
    int sum=0;
    for(int i=0;i<n;i++){
        sum+=abs(disk[i]-initial_head);
        initial_head=disk[i];
    }
    printf("total disk movement is %d",sum);


}
//SSTF Disk Scheduling
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n;
    printf("Enter the number of disk movements: ");
    scanf("%d", &n);

    int initial_head;
    printf("Enter initial head position: ");
    scanf("%d", &initial_head);

    int disk[n];
    printf("Enter the disk positions:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &disk[i]);
    }

    int processed_count = 0;
    int sum = 0;
    int current_head = initial_head;

    while (processed_count < n) {
        int min_distance = _INT_MAX_;
        int nearest_index = -1;

        for (int i = 0; i < n; i++) {
            if (disk[i] != -1) {  // Check if request is not processed
                int distance = abs(disk[i] - current_head);
                if (distance < min_distance) {
                    min_distance = distance;
                    nearest_index = i;
                }
            }
        }

        if (nearest_index != -1) {
            sum += min_distance;
            current_head = disk[nearest_index];
            disk[nearest_index] = -1;  // Mark request as processed
            processed_count++;
        }
    }

    printf("Total disk movement is %d\n", sum);

    return 0;
}
```