1)

SJF:(With preemption) / SRF:

```
// C++ program to implement Shortest Remaining Time First
// Shortest Remaining Time First (SRTF)

#include <bits/stdc++.h>
using namespace std;

struct Process {
	int pid; // Process ID
	int bt; // Burst Time
	int art; // Arrival Time
};

// Function to find the waiting time for all
// processes
void findWaitingTime(Process proc[], int n,int wt[])
{
	int rt[n];

	// Copy the burst time into rt[]
	for (int i = 0; i < n; i++)
		rt[i] = proc[i].bt;

	int complete = 0, t = 0, minm = INT_MAX;
	int shortest = 0, finish_time;
	bool check = false;

	// Process until all processes gets
	// completed
	while (complete != n) {

		// Find process with minimum
		// remaining time among the
		// processes that arrives till the
		// current time`
		for (int j = 0; j < n; j++) {
			if ((proc[j].art <= t) &&
			(rt[j] < minm) && rt[j] > 0) {
				minm = rt[j];
				shortest = j;
				check = true;
			}
		}

		if (check == false) {
			t++;
			continue;
		}
```

```
			// Reduce remaining time by one
			rt[shortest]--;

			// Update minimum
			minm = rt[shortest];
			if (minm == 0)
				minm = INT_MAX;

			// If a process gets completely
			// executed
			if (rt[shortest] == 0) {

				// Increment complete
				complete++;
				check = false;

				// Find finish time of current
				// process
				finish_time = t + 1;

				// Calculate waiting time
				wt[shortest] = finish_time -
							proc[shortest].bt -
							proc[shortest].art;

				if (wt[shortest] < 0)
					wt[shortest] = 0;
			}
			// Increment time
			t++;
		}
}

// Function to calculate turn around time
void findTurnAroundTime(Process proc[], int n,
							int wt[], int tat[])
{
	// calculating turnaround time by adding
	// bt[i] + wt[i]
	for (int i = 0; i < n; i++)
		tat[i] = proc[i].bt + wt[i];
}

// Function to calculate average time
void findavgTime(Process proc[], int n)
{
	int wt[n], tat[n], total_wt = 0,
					total_tat = 0;
```

```
	// Function to find waiting time of all
	// processes
	findWaitingTime(proc, n, wt);

	// Function to find turn around time for
	// all processes
	findTurnAroundTime(proc, n, wt, tat);

	// Display processes along with all
	// details
	cout << " P\t\t"
		<< "BT\t\t"
		<< "WT\t\t"
		<< "TAT\t\t\n";

	// Calculate total waiting time and
	// total turnaround time
	for (int i = 0; i < n; i++) {
		total_wt = total_wt + wt[i];
		total_tat = total_tat + tat[i];
		cout << " " << proc[i].pid << "\t\t"
			<< proc[i].bt << "\t\t " << wt[i]
			<< "\t\t " << tat[i] << endl;
	}

	cout << "\nAverage waiting time = "
		<< (float)total_wt / (float)n;
	cout << "\nAverage turn around time = "
		<< (float)total_tat / (float)n;
}

// Driver code
int main()
{
	Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 },
					{ 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
	int n = sizeof(proc) / sizeof(proc[0]);

	findavgTime(proc, n);
	return 0;
}
```

Priority (non-preemptive):

```
/*
 * C program to implement priority scheduling
 */
```

```c
#include <stdio.h>

//Function to swap two variables
void swap(int *a,int *b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
int main()
{
    int n;
    printf("Enter Number of Processes: ");
    scanf("%d",&n);

    // b is array for burst time, p for priority and index for process id
    int b[n],p[n],index[n];
    for(int i=0;i<n;i++)
    {
        printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
        scanf("%d %d",&b[i],&p[i]);
        index[i]=i+1;
    }
    for(int i=0;i<n;i++)
    {
        int a=p[i],m=i;

        //Finding out highest priority element and placing it at its desired position
        for(int j=i;j<n;j++)
        {
            if(p[j] > a)
            {
                a=p[j];
                m=j;
            }
        }

        //Swapping processes
        swap(&p[i], &p[m]);
        swap(&b[i], &b[m]);
        swap(&index[i],&index[m]);
    }

    // T stores the starting time of process
    int t=0;

    //Printing scheduled process
    printf("Order of process Execution is\n");
    for(int i=0;i<n;i++)
```

```
    {
        printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);
        t+=b[i];
    }
    printf("\n");
    printf("Process Id     Burst Time   Wait Time    TurnAround Time\n");
    int wait_time=0;
    for(int i=0;i<n;i++)
    {
        printf("P%d          %d          %d          %d\n",index[i],b[i],wait_time,wait_time + b[i]);
        wait_time += b[i];
    }
    return 0;
}
```

Priority: (premption):

```
#include<stdio.h>
 // structure representing a structure
struct priority_scheduling {

  // name of the process
  char process_name;

  // time required for execution
  int burst_time;

  // waiting time of a process
  int waiting_time;

  // total time of execution
  int turn_around_time;

  // priority of the process
  int priority;
};

int main() {

  // total number of processes
  int number_of_process;

  // total waiting and turnaround time
  int total = 0;

  // temporary structure for swapping
  struct priority_scheduling temp_process;

  // ASCII numbers are used to represent the name of the process
  int ASCII_number = 65;
```

```
  return 0;
}
```

SJF (Without preemptive):

```
/*
 * C Program to Implement SJF Scheduling
 */

#include<stdio.h>
int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;
    }

    //sorting of burst times
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0;

    //finding the waiting time of all the processes
```

```
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
             //individual WT by adding BT of all previous completed processes
            wt[i]+=bt[j];

        //total waiting time
        total+=wt[i];
    }

    //average waiting time
    avg_wt=(float)total/n;

    printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        //turnaround time of individual processes
        tat[i]=bt[i]+wt[i];

        //total turnaround time
        totalT+=tat[i];
        printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
  //average turnaround time
    avg_tat=(float)totalT/n;
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f",avg_tat);
}
```

Round Robin:

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

void main()
{
    // initlialize the variable name
    int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
    float avg_wt, avg_tat;
    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP; // Assign the number of process to variable y

// Use for loop to enter the details of the process like Arrival time and the Burst Time
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
printf(" Arrival time is: \t");  // Accept arrival time
scanf("%d", &at[i]);
```

```
printf(" \nBurst time is: \t"); // Accept the Burst time
scanf("%d", &bt[i]);
temp[i] = bt[i]; // store the burst time in temp array
}
// Accept the Time qunat
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
// Display the process No, burst time, Turn Around Time and the waiting time
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0) // define the conditions
{
   sum = sum + temp[i];
   temp[i] = 0;
   count=1;
   }
   else if(temp[i] > 0)
   {
      temp[i] = temp[i] - quant;
      sum = sum + quant;
   }
   if(temp[i]==0 && count==1)
   {
      y--; //decrement the process no.
      printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
      wt = wt+sum-at[i]-bt[i];
      tat = tat+sum-at[i];
      count =0;
   }
   if(i==NOP-1)
   {
      i=0;
   }
   else if(at[i+1]<=sum)
   {
      i++;
   }
   else
   {
      i=0;
   }
}
// represents the average waiting time and Turn Around time
avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
```

```
}
```

3)

```c
// Banker's Algorithm
#include <stdio.h>
int main()
{
	// P0, P1, P2, P3, P4 are the Process names here

	int n, m, i, j, k;
	n = 5; // Number of processes
	m = 3; // Number of resources
	int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
						{ 2, 0, 0 }, // P1
						{ 3, 0, 2 }, // P2
						{ 2, 1, 1 }, // P3
						{ 0, 0, 2 } }; // P4

	int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
					{ 3, 2, 2 }, // P1
					{ 9, 0, 2 }, // P2
					{ 2, 2, 2 }, // P3
					{ 4, 3, 3 } }; // P4

	int avail[3] = { 3, 3, 2 }; // Available Resources

	int f[n], ans[n], ind = 0;
	for (k = 0; k < n; k++) {
		f[k] = 0;
	}
	int need[n][m];
	for (i = 0; i < n; i++) {
		for (j = 0; j < m; j++)
			need[i][j] = max[i][j] - alloc[i][j];
	}
	int y = 0;
	for (k = 0; k < 5; k++) {
		for (i = 0; i < n; i++) {
			if (f[i] == 0) {

				int flag = 0;
				for (j = 0; j < m; j++) {
					if (need[i][j] > avail[j]){
						flag = 1;
						break;
					}
				}
```

```
                    if (flag == 0) {
                        ans[ind++] = i;
                        for (y = 0; y < m; y++)
                            avail[y] += alloc[i][y];
                        f[i] = 1;
                    }
                }
            }
        }

        int flag = 1;

        for(int i=0;i<n;i++)
        {
        if(f[i]==0)
        {
            flag=0;
            printf("The following system is not safe");
            break;
        }
        }

        if(flag==1)
        {
        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
            printf(" P%d ->", ans[i]);
        printf(" P%d", ans[n - 1]);
        }


        return (0);

        // This code is contributed by Deep Baldha (CandyZack)
}
```

5) LRU:

```
#include<stdio.h>
main()
{
int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];
printf("Enter no of pages:");
scanf("%d",&n);
printf("Enter the reference string:");
for(i=0;i<n;i++)
        scanf("%d",&p[i]);
printf("Enter no of frames:");
scanf("%d",&f);
```

 Earliest-deadline-first (EDF) : A variant of Priority scheduling where a deadline is given for each process and this deadline is treated as the priority of that process - the earlier the deadline, the higher the priority; the later the deadline, the lower the priority

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct process {
   int process_id;
   int arrival_time;
   int burst_time;
   int remaining_time;
   int priority;
   int waiting_time;
   int turnaround_time;
   int finish_time;
   int deadline;
};

bool compare_deadlines(process p1, process p2) {
```

```
    return p1.deadline < p2.deadline;
}

void edf_scheduling(vector<process>& processes) {
    int current_time = 0;
    int total_processes = processes.size();
    int completed_processes = 0;
    int total_wait_time = 0;
    int total_turnaround_time = 0;

    while (completed_processes < total_processes) {
        // Sort the processes based on their deadlines
        sort(processes.begin(), processes.end(), compare_deadlines);

        bool process_completed = false;
        for (int i = 0; i < total_processes; i++) {
            process& current_process = processes[i];

            // Check if process has arrived and has remaining burst time
            if (current_process.arrival_time <= current_time && current_process.remaining_time > 0) {
                // Execute the process for its remaining burst time
                current_time += current_process.remaining_time;

                // Set finish time for the process
                current_process.finish_time = current_time;
```

```
            // Set waiting time for the process
            current_process.waiting_time = current_process.finish_time - current_process.arrival_time - current_process.burst_time;

            // Set turnaround time for the process
            current_process.turnaround_time = current_process.finish_time - current_process.arrival_time;

            // Update total wait time and total turnaround time
            total_wait_time += current_process.waiting_time;
            total_turnaround_time += current_process.turnaround_time;

            // Mark the process as completed
            completed_processes++;

            process_completed = true;
          }
        }

        // If no process has completed, move to the next time unit
        if (!process_completed) {
          current_time++;
        }
      }
```

```
    // Calculate and print average wait time and average turnaround time
    float avg_wait_time = (float) total_wait_time / total_processes;
    float avg_turnaround_time = (float) total_turnaround_time / total_processes;
    cout << "Average Wait Time: " << avg_wait_time << endl;
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
}

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

    vector<process> processes(n);
    for (int i = 0; i < n; i++) {
        cout << "Enter arrival time for process " << i+1 << ": ";
        cin >> processes[i].arrival_time;
        cout << "Enter burst time for process " << i+1 << ": ";
        cin >> processes[i].burst_time;
        cout << "Enter deadline for process " << i+1 << ": ";
        cin >> processes[i].deadline;

        // Set remaining time and priority as deadline for EDF scheduling
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].priority = processes[i].deadline;
        processes[i].process_id = i+1;
```

```
    }

    // Schedule the processes using EDF algorithm
    edf_scheduling(processes);

    return 0;
}
```

Progressive Round-Robin (PRR) : A variation of the RR scheduler where the timequantum is increased by 10% if a process does not complete within it's entire timequantum. When the time time-quantum increases by 100% and any process blocks before using its entire time quantum, the time quantum will be reduced to it's default value again.

```
#include <iostream>
#include <vector>

using namespace std;

struct process {
    int process_id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int priority;
    int waiting_time;
    int turnaround_time;
```

```
    int finish_time;
};

void prr_scheduling(vector<process>& processes, int time_quantum) {
    int current_time = 0;
    int total_processes = processes.size();
    int completed_processes = 0;
    int total_wait_time = 0;
    int total_turnaround_time = 0;
    float time_quantum_multiplier = 1.0;

    while (completed_processes < total_processes) {
        bool process_completed = false;
        for (int i = 0; i < total_processes; i++) {
            process& current_process = processes[i];

            // Check if process has arrived and has remaining burst time
            if (current_process.arrival_time <= current_time && current_process.remaining_time > 0) {
                int time_quantum_used = 0;

                // Check if process can complete within the time quantum
                if (current_process.remaining_time <= time_quantum*time_quantum_multiplier) {
                    time_quantum_used = current_process.remaining_time;
                }
                else {
```

```
    time_quantum_used = time_quantum*time_quantum_multiplier;
}

// Update remaining time for the current process
current_process.remaining_time -= time_quantum_used;
current_time += time_quantum_used;

// Update time quantum multiplier if process did not complete within the time quantum
if (time_quantum_used < time_quantum*time_quantum_multiplier) {
    time_quantum_multiplier += 0.1;
}

// Check if process has completed execution
if (current_process.remaining_time == 0) {
    // Set finish time for the process
    current_process.finish_time = current_time;

    // Set waiting time for the process
    current_process.waiting_time = current_process.finish_time - current_process.arrival_time -
current_process.burst_time;

    // Set turnaround time for the process
    current_process.turnaround_time = current_process.finish_time -
current_process.arrival_time;

    // Update total wait time and total turnaround time
```

```
                    total_wait_time += current_process.waiting_time;
                    total_turnaround_time += current_process.turnaround_time;

                    // Mark the process as completed
                    completed_processes++;

                    // Reset time quantum multiplier to 1.0 if process blocked before using its entire time quantum
                    if (time_quantum_used < time_quantum) {
                        time_quantum_multiplier = 1.0;
                    }

                    process_completed = true;
                }
            }
        }

        // If no process has completed, move to the next time unit
        if (!process_completed) {
            current_time++;
        }
    }

    // Calculate and print average wait time and average turnaround time
    float avg_wait_time = (float) total_wait_time / total_processes;
    float avg_turnaround_time = (float) total_turnaround_time / total_processes;
```

```
    cout << "Average Wait Time: " << avg_wait_time << endl;
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
}

int main() {
    int n;
    int time_quantum;
    cout << "Enter the number of processes: ";
    cin >> n;
    cout << "Enter time quantum: ";
    cin >> time_quantum;
    // Input process details
    vector<process> processes(n);
    for (int i = 0; i < n; i++) {
            processes[i].process_id = i + 1;
            cout << "Enter arrival time for process " << i+1 << ": ";
            cin >> processes[i].arrival_time;
            cout << "Enter burst time for process " << i+1 << ": ";
            cin >> processes[i].burst_time;
            processes[i].remaining_time = processes[i].burst_time;
}

// Call PRR scheduling function
prr_scheduling(processes, time_quantum);
```

```
return 0;

}
```

Resource-Allocation graph:

```
#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <cstring>


using namespace std;


const int MAXN = 100;

const int MAXM = 100;


int n, m;  // n = number of processes, m = number of resources


int available[MAXM];  // available resources

int max_need[MAXN][MAXM];  // max need of each process

int allocated_resources[MAXN][MAXM];  // allocated resources of each process

int need[MAXN][MAXM];  // need of each process


vector<int> graph[MAXN];  // resource allocation graph

int in_degree[MAXN];  // in-degree of each node in the graph
```

```
void initialize() {
    memset(in_degree, 0, sizeof(in_degree));
    cout << "Enter the number of processes: ";
    cin >> n;
    cout << "Enter the number of resources: ";
    cin >> m;
    cout << "Enter the available resources: ";
    for (int j = 0; j < m; j++) {
        cin >> available[j];
    }
    cout << "Enter the max need of each process: ";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> max_need[i][j];
        }
    }
    cout << "Enter the allocated resources of each process: ";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> allocated_resources[i][j];
            need[i][j] = max_need[i][j] - allocated_resources[i][j];
        }
    }
}
```

```
void buildGraph() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (need[i][j] <= available[j]) {
                // Process i can request resource j
                graph[i].push_back(j + n);
                in_degree[j + n]++;
            }
        }
    }
}

bool isSafe() {
    queue<int> q;
    vector<int> safe_sequence;
    int work[MAXM];
    memcpy(work, available, sizeof(available));

    // Add all nodes with in-degree 0 to the queue
    for (int i = 0; i < n + m; i++) {
        if (in_degree[i] == 0) {
            q.push(i);
        }
    }
```

```
// Perform a topological sort of the resource allocation graph
while (!q.empty()) {
    int u = q.front();
    q.pop();
    if (u < n) {
        // Process u can be allocated resources
        safe_sequence.push_back(u);
        for (int j = 0; j < m; j++) {
            work[j] += allocated_resources[u][j];
        }
    } else {
        // Resource u-n can be allocated to a process
        int v = u - n;
        for (int i = 0; i < graph[v].size(); i++) {
            int w = graph[v][i];
            in_degree[w]--;
            if (in_degree[w] == 0) {
                q.push(w);
            }
        }
    }
}

if (safe_sequence.size() == n) {
```

```
        cout << "System is in safe state" << endl;

        cout << "Safe sequence is: ";

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

            cout << "P" << safe_sequence[i] << " ";

        }

        cout << endl;

        return true;

        }

    else {

        cout << "System is not in safe state" << endl;

        return false;

        }

    }


int main() {

    initialize();

    buildGraph();

    isSafe();

    return 0;

}
```

Round Robin scheduling algorithm (with variable time quantum) :

```
#include <iostream>

#include <vector>
```

```
using namespace std;

struct Process {
    int process_id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int wait_time;
    int turnaround_time;
};

void calculate_round_robin(vector<Process>& processes, vector<int>& time_quantum) {
    int n = processes.size();
    int current_time = 0;
    int total_wait_time = 0;
    int total_turnaround_time = 0;
    int completed = 0;

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                int quantum = min(time_quantum[i], processes[i].remaining_time);
                current_time += quantum;
                processes[i].remaining_time -= quantum;
```

```
            if (processes[i].remaining_time == 0) {
                completed++;
                processes[i].turnaround_time = current_time - processes[i].arrival_time;
                processes[i].wait_time = processes[i].turnaround_time - processes[i].burst_time;
            }
        }
    }
}

for (auto& process : processes) {
    total_wait_time += process.wait_time;
    total_turnaround_time += process.turnaround_time;
}

double avg_wait_time = (double)total_wait_time / n;
double avg_turnaround_time = (double)total_turnaround_time / n;

cout << "Average Waiting Time (AWT): " << avg_wait_time << endl;
cout << "Average Turnaround Time (ATT): " << avg_turnaround_time << endl;
}

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

```
    cin >> n;

    vector<Process> processes(n);
    vector<int> time_quantum(n);

    cout << "Enter the burst time, arrival time, and time quantum for each process:" << endl;
    for (int i = 0; i < n; i++) {
        processes[i].process_id = i + 1;
        cout << "Process " << processes[i].process_id << endl;
        cout << "Burst time: ";
        cin >> processes[i].burst_time;
        cout << "Arrival time: ";
        cin >> processes[i].arrival_time;
        cout << "Time quantum: ";
        cin >> time_quantum[i];
        processes[i].remaining_time = processes[i].burst_time;
        cout << endl;
    }

    calculate_round_robin(processes, time_quantum);

    return 0;
}
```