1. SJF:

```
#include<iostream>
#include <algorithm>
#include<iomanip>
#include<climits>
using namespace std;

struct process_struct
{
  int pid;
  int at;
  int bt;
  int ct,wt,tat,rt,start_time;
}ps[100];

int main()
{

   int n;
   bool is_completed[100]={false},is_first_process=true;
   int current_time = 0;
   int completed = 0;;
   cout<<"Enter total number of processes: ";
   cin>>n;
   int sum_tat=0,sum_wt=0,sum_rt=0,total_idle_time=0,prev=0,length_cycle;
   float cpu_utilization;
   int max_completion_time,min_arrival_time;

   cout << fixed << setprecision(2);

   for(int i=0;i<n;i++)
   {
      cout<<"\nEnter Process " <<i<< " Arrival Time: ";
      cin >> ps[i].at;
      ps[i].pid=i;
   }

   for(int i=0;i<n;i++)
   {
      cout<<"\nEnter Process " <<i<< " Burst Time: ";
      cin >> ps[i].bt;
   }


   while(completed!=n)
   {
      //find process with min. burst time in ready queue at current time
      int min_index = -1;
      int minimum = INT_MAX;
      for(int i = 0; i < n; i++) {
         if(ps[i].at <= current_time && is_completed[i] == false) {
            if(ps[i].bt < minimum) {
               minimum = ps[i].bt;
               min_index = i;
            }
            if(ps[i].bt== minimum) {
               if(ps[i].at < ps[min_index].at) {
                  minimum= ps[i].bt;
                  min_index = i;
               }
            }
         }
      }

      if(min_index==-1)
      {
```

```
            current_time++;
        }
        else
        {
        ps[min_index].start_time = current_time;
        ps[min_index].ct = ps[min_index].start_time + ps[min_index].bt;
        ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
        ps[min_index].wt = ps[min_index].tat - ps[min_index].bt;
        ps[min_index].rt = ps[min_index].wt;
        // ps[min_index].rt = ps[min_index].start_time - ps[min_index].at;

        sum_tat +=ps[min_index].tat;
        sum_wt += ps[min_index].wt;


        completed++;
        is_completed[min_index]=true;
        current_time = ps[min_index].ct;
        prev= current_time;
        is_first_process = false;
        }
      }


       //Output
      cout<<"\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n";
      for(int i=0;i<n;i++)
       cout<<i<<"\t\t"<<ps[i].at<<"\t"<<ps[i].bt<<"\t\t"<<ps[i].ct<<"\t"<<ps[i].tat<<"\t"<<ps[i].wt<<endl;
      cout<<endl;



      cout<<"\nAverage Turn Around time= "<< (float)sum_tat/n;
      cout<<"\nAverage Waiting Time= "<<(float)sum_wt/n;

      return 0;
    }
```

2. Priority Scheduling

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

struct process{
        int pid, arrival_time, burst_time, priority, start_time, completion_time, turnaround_time, waiting_time;
}p[100];

int main(){
        float avg_tat;
        float avg_waiting_time;
        float avg_response_time;
        int total_turnaround_time = 0;
        int total_waiting_time = 0;

        int is_completed[100];
        memset(is_completed, 0, sizeof(is_completed));

        int n;
        cout << "Enter the number of processes: ";
        cin >> n;
```

```
	for(int i = 0 ; i < n; i++){
		cout << "Enter arrival time of the process " << i << ":";
		cin >> p[i].arrival_time;
		cout << "Enter burst time of the process " << i << ":";
		cin >> p[i].burst_time;
		cout << "Enter priority of the process " << i << ":";
		cin >> p[i].priority;
		p[i].pid = i;
		cout << endl;
	}

	int current_time = 0;
	int completed = 0;


	while(completed != n){
		int idx = -1;
		int mx = -1;
		for(int i = 0; i < n; i++){
			if(p[i].arrival_time <= current_time && is_completed[i] == 0){
				if(p[i].priority>mx){
					mx = p[i].priority;
					idx = i;
				}
				if(p[i].priority == mx){
					if(p[i].arrival_time < p[idx].arrival_time){
						mx = p[i].priority;
						idx = i;
					}
				}
			}
		}

		if(idx != -1){
			p[idx].start_time = current_time;
			p[idx].completion_time = p[idx].start_time + p[idx].burst_time;
			p[idx].turnaround_time = p[idx].completion_time - p[idx].arrival_time;
			p[idx].waiting_time = p[idx].turnaround_time - p[idx].burst_time;

			total_turnaround_time+=p[idx].turnaround_time;
			total_waiting_time += p[idx].waiting_time;

			is_completed[idx] = 1;
			completed++;
			current_time = p[idx].completion_time;

		}
		else{
			current_time++;
		}
	}

	avg_tat = (float)total_turnaround_time/n;
	avg_waiting_time = (float) total_waiting_time/n;

	cout << endl;
	cout << "#P\t" << "AT\t" << "BT\t" << "Priority\t" << "TAT\t" << "WT\n" << endl;
	for(int i = 0 ; i < n; i++){
		cout << p[i].pid << "\t" << p[i].arrival_time << "\t" << p[i].burst_time <<"\t" << p[i].priority << "\t" <<
p[i].turnaround_time << "\t" << p[i].waiting_time << endl;
	}
```

```
        cout << endl;
        cout << "Avarage turnaround time: " << avg_tat << endl;
        cout << "Avarage waiting time: " << avg_waiting_time << endl;

}
```

3. Round Robin:

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

struct process{
        int pid, arrival_time, burst_time, priority, start_time, completion_time,
         turnaround_time, waiting_time;
}p[100];



bool comparatorAT(struct process a, struct process b){
        int x = a.arrival_time;
        int y = b.arrival_time;
        return x<y;
}

int main(){
        float avg_turnaround_time;
        float avg_waiting_time;
        float avg_response_time;
        int total_turnaround_time = 0;
        int total_waiting_time = 0;

        int bt_remaining[100];
        queue<int> q;
        int current_time  = 0;
        int completed = 0, tq;
        int visited[100] = {false};

        int n, index;
        cout << "Enter the number of processes: ";
        cin >> n;

        for(int i = 0 ; i < n; i++){
                cout << "Enter arr ival time of the process " << i << ":";
                cin >> p[i].arrival_time;
                cout << "Enter burst time of the process " << i << ":";
                cin >> p[i].burst_time;
                p[i].pid = i;
                cout << endl;
                bt_remaining[i] = p[i].burst_time;

        }
        cout << endl;
        cout << "Enter the time quanta: ";
        cin >> tq;

        sort(p, p+n, comparatorAT);

        q.push(0);
        visited[0] = true;

        while(completed != n){
```

```
                index = q.front();
                q.pop();
                if(bt_remaining[index] == p[index].burst_time){
                        p[index].start_time = max(current_time, p[index].arrival_time);
                        current_time = p[index].start_time;
                }
                if(bt_remaining[index]-tq > 0){
                        bt_remaining[index]-=tq;
                        current_time +=tq;
                }
                else{
                        current_time += bt_remaining[index];
                        bt_remaining[index] = 0;
                        completed++;

                        p[index].completion_time= current_time;
                        p[index].turnaround_time = p[index].completion_time - p[index].arrival_time;
                        p[index].waiting_time = p[index].turnaround_time - p[index].burst_time;

                        total_turnaround_time += p[index].turnaround_time;
                        total_waiting_time +=p[index].waiting_time;

                }
                for(int i = 1; i < n; i++){
                        if(bt_remaining[i] > 0 && p[i].arrival_time <= current_time && visited[i] == false){
                                q.push(i);
                                visited[i] = true;
                        }
                }

                if(bt_remaining[index] > 0)
                        q.push(index);

                if(q.empty()){
                        for(int i = 1; i < n; i++){
                                if(bt_remaining[i] > 0){
                                        q.push(i);
                                        visited[i] = true;
                                        break;
                                }
                        }
                }
        }

        cout << "Process NO\tAT\tBrust Time\tCT\tTAT\tWT" << endl;
        for(int i = 0; i < n; i++){
                cout << p[i].pid << "\t" << p[i].arrival_time << "\t" << p[i].burst_time << "\t" << p[i].completion_time << "\t" <<
p[i].turnaround_time << "\t" << p[i].waiting_time << endl;
        }

        cout << "Average turnaround time: " << (float)total_turnaround_time/n << endl;
        cout << "Average waiting time: " << (float)total_waiting_time/n << endl;
        return 0;

}
```

4. Banker's allocation:

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

struct file{
```

```
        int all[10];
        int max[10];
        int need[10];
        int flag;
}f[10];

int main(){
        int fl, i,j,k,p,n,r,g,cnt = 00, id, need;
        int avail[10], seq[10];
        cout << "Enter the number of process: ";
        cin >> n;
        cout << "Enter the number of resources: ";
        cin >> r;

        for(int i = 0; i < n;i++){
                cout << "Enter details for Process " << i << endl;
                cout << "Enter allocation: \t";
                for(int j = 0 ; j < r; j++){
                        cin >> f[i].all[j];
                }
                cout << "Enter Max \t";
                for(int j = 0; j < r; j++){
                        cin >> f[i].max[j];
                }
                f[i].flag = 0;
        }
        cout << "Enter available resources: " << endl;
        for(int i = 0; i < r; i++){
                cin >> avail[i];
        }

        //NEED Calculation
        for(int i = 0; i < n; i++){
                for(int j = 0 ; j < r; j++){
                        f[i].need[j] = f[i].max[j] - f[i].all[j];
                }
        }

        cnt = 0, fl = 0;
        while(cnt!=n){
                g = 0;
                for(int j = 0; j < n; j++){
                        if(f[j].flag == 0){
                                int b= 0 ;
                                for(int p = 0; p < r; p++){
                                        if(avail[p] >= f[j].need[p]){
                                                b+=1;
                                        }
                                        else
                                                break;
                                }
                                if(b==r){
                                        cout <<"\n" << j << " is visited";
                                        seq[fl++] = j;
                                        f[j].flag = 1;
                                        for(k = 0; k < r; k++){
                                                avail[k]+=f[j].all[k];
                                        }
                                        cnt+=1;
                                        g++;
                                }
```

```
                        }
                }
                if(g == 0){
                        cout << "Request is not granted and deadlock is present" <<endl;
                        cout << "SYSTEM IS UNSAFE" << endl;
                }
        }
        cout << endl;
        cout << "System is in SAFE STATE" << endl;
        cout << "Safe Sequence is: "<< endl;
        for(int i = 0 ; i < n; i++){
                cout << seq[i] << "\t";
        }
        cout << endl;

        printf("\nProcess\t\tAllocation\t\tMax\t\tNeed");
        cout << endl;
        for(int i = 0; i < n; i++){
                printf("%d\t", i);
                for(int j = 0 ; j < r; j++){
                        cout << " " << f[i].all[j];
                }
                cout << "\t" ;
                for(int j = 0 ; j < r; j++){
                        cout << " " << f[i].max[j];
                }

                cout << "\t";
                for(int j = 0 ; j < r; j++){
                        cout << " " << f[i].need[j];
                }
                cout << endl;
        }
}
```