**Round Robbin CPU Scheduling:**

```
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>  //for bool datatype
#include <stdlib.h> //for qsort
struct process_struct
{
    int pid;
    int at;
    int bt;
    int ct,wt,tat,rt,start_time;
    int bt_remaining;
} ps[100];
int findmax(int a, int b)
{
    return a>b?a:b;
}
int comparatorAT(const void * a, const void *b)
{
    int x =((struct process_struct *)a) -> at;
    int y =((struct process_struct *)b) -> at;
    if(x<y)
        return -1;  // No sorting
    else if( x>=y) // = is for stable sort
        return 1;    // Sort
}
int comparatorPID(const void * a, const void *b)
{
    int x =((struct process_struct *)a) -> pid;
    int y =((struct process_struct *)b) -> pid;
    if(x<y)
        return -1;  // No sorting
    else if( x>=y)
        return 1;    // Sort
}
int main()
{
    int n,index;
    int cpu_utilization;
//queue<int> q;
    bool visited[100]= {false},is_first_process=true;
    int current_time = 0,max_completion_time;
    int completed = 0,tq, total_idle_time=0,length_cycle;
    printf("Enter total number of processes: ");
    scanf("%d",&n);
    int queue[100],front=-1,rear=-1;
    float sum_tat=0,sum_wt=0,sum_rt=0;
    for(int i=0; i<n; i++)
    {
        printf("\nEnter Process %d Arrival Time: ",i);
        scanf("%d",&ps[i].at);
        ps[i].pid=i;
    }

    for(int i=0; i<n; i++)
    {
        printf("\nEnter Process %d Burst Time: ",i);
        scanf("%d",&ps[i].bt);
        ps[i].bt_remaining= ps[i].bt;
    }

    printf("\nEnter time quanta: ");
    scanf("%d",&tq);

    //sort structure on the basis of Arrival time in increasing order
    qsort((void *)ps,n, sizeof(struct process_struct),comparatorAT);
    // q.push(0);
    front=rear=0;
    queue[rear]=0;
    visited[0] = true;

    while(completed != n)
    {
        index = queue[front];
        //q.pop();
        front++;

        if(ps[index].bt_remaining == ps[index].bt)
        {
            ps[index].start_time = findmax(current_time,ps[index].at);
            total_idle_time += (is_first_process == true) ? 0 : ps[index].start_time -
current_time;
            current_time =  ps[index].start_time;
            is_first_process = false;
        }

        if(ps[index].bt_remaining-tq > 0)
        {
            ps[index].bt_remaining -= tq;
            current_time += tq;
        }
        else
        {
            current_time += ps[index].bt_remaining;
            ps[index].bt_remaining = 0;
            completed++;
            ps[index].ct = current_time;
            ps[index].tat = ps[index].ct - ps[index].at;
            ps[index].wt = ps[index].tat - ps[index].bt;
            ps[index].rt = ps[index].start_time - ps[index].at;
            sum_tat += ps[index].tat;
            sum_wt += ps[index].wt;
            sum_rt += ps[index].rt;
        }
//check which new Processes needs to be pushed to Ready Queue from Input list
        for(int i = 1; i < n; i++)
        {
            if(ps[i].bt_remaining > 0 && ps[i].at <= current_time && visited[i] == false)
            {
               //q.push(i);
               queue[++rear]=i;
               visited[i] = true;
            }
        }
        //check if Process on CPU needs to be pushed to Ready Queue
        if( ps[index].bt_remaining> 0)
        {   //q.push(index);
            queue[++rear]=index;
        }
        //if queue is empty, just add one process from list, whose remaining burst time > 0
        if(front>rear)
        {
            for(int i = 1; i < n; i++)
            {
                if(ps[i].bt_remaining > 0)
                {
                    queue[rear++]=i;
                    visited[i] = true;
                    break;
                }
            }
        }
    } //end of while

    //Calculate Length of Process completion cycle
    max_completion_time = INT_MIN;

    for(int i=0; i<n; i++)
    {
        max_completion_time = findmax(max_completion_time,ps[i].ct);
    }
    length_cycle = max_completion_time - ps[0].at;  //ps[0].start_time;
```

```
        cpu_utilization = (float)(length_cycle - total_idle_time)/ length_cycle;

//sort so that process ID in output comes in Original order (just for interactivity- Not needed otherwise)
        qsort((void *)ps,n, sizeof(struct process_struct),comparatorPID);
//Output
        printf("\nProcess No.\tAT\tCPU Burst Time\tStart Time\tCT\tTAT\tWT\tRT\n");
        for(int i=0; i<n; i++)
        {

        printf("%d\t\t%d\t%d\t\t%d\t\t%d\t%d\t%d\t%d\n",i,ps[i].at,ps[i].bt,ps[i].start_time,ps[i].ct,ps
                [i].tat,ps[i].wt,ps[i].rt);
        }
        printf("\n");
        printf("\nAverage Turn Around time= %.2f",(float)sum_tat/n);
        printf("\nAverage Waiting Time= %.2f",(float)sum_wt/n);
        printf("\nAverage Response Time= %.2f",(float)sum_rt/n);
        printf("\nThroughput= %.2f",n/(float)length_cycle);
        printf("\nCPU Utilization(Percentage)= %.2lf",cpu_utilization*100);
        return 0;
}
```

**Bankers’s Algorithm:**

```
#include<stdio.h>
struct Process{
  int A , B , C; // resources
};


int main()
{
  int n , A , B , C;
  printf("Enter the Number of Process - ");
  scanf("%d" ,&n);

  struct Process allocated[n];
  struct Process require[n];
  struct Process Need[n];
  int isExecute[n];
  int ans[n];

  for(int i = 0 ; i < n ; i++)
    isExecute[i] = 0;
  int al_a= 0;
  int al_b = 0;
  int al_c= 0 ;
  for(int i = 0 ; i < n ; i++)
```

```
{
   printf("Enter the Allocated and Required Resources of P%d - " , i);
   scanf("%d%d%d" , &allocated[i].A, &allocated[i].B, &allocated[i].C);
   scanf("%d%d%d" , &require[i].A , &require[i].B , &require[i].C);

   al_a += allocated[i].A;
   al_b += allocated[i].B;
   al_c += allocated[i].C;

   Need[i].A = require[i].A - allocated[i].A;
   Need[i].B = require[i].B - allocated[i].B;
   Need[i].C = require[i].C - allocated[i].C;
}

printf("Enter the Aavilable Resources = ");
scanf("%d%d%d" , &A,&B,&C);
A -= al_a;
B -= al_b;
C -= al_c;

int idx = 0;
for(int i = 0 ; i < n ; i++)
{
   if(isExecute[i] == 0)
   {
      for(int j = 0 ; j < n ; j++)
      {
         if(isExecute[j] == 0)
         {
            if(Need[j].A <= A && Need[j].B <= B && Need[j].C <= C)
            {
               A += allocated[j].A;
               B += allocated[j].B;
               C += allocated[j].C;

               ans[idx] = j;
               idx++;
               isExecute[j] = 1;
            }
         }
      }
   }
}


for(int i = 0 ; i < n ; i++)
{
   if(isExecute[i] == 0)
```

```
        {
            printf("No Safe Sequence is Possible");
            return 0;
        }
    }

    printf("The Safe Sequence is - ");
    for(int i = 0 ; i < n-1 ; i++)
    {
        printf("P%d ->" , ans[i]);
    }
    printf("P%d" , ans[n-1]);
    return 0;
}
```