Round Robin

```
#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)= %.2f",cpu_utilization*100);
  return 0;
}
```

2. priority (Non-Preemptive)

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

struct process_struct
{
 int at;
 int bt;
 int priority;
```

```
  int ct,wt,tat,rt,start_time;
}ps[100];

int findmax(int a, int b)
{
   return a>b?a:b;
}

int findmin(int a, int b)
{
   return a<b?a:b;
}

int main()
{

   int n;
   bool is_completed[100]={false},is_first_process=true;
   int current_time = 0;
   int completed = 0;
   int total_idle_time=0,prev=0,length_cycle;
   float cpu_utilization;
   int max_completion_time,min_arrival_time;
   //printf("Enter total number of processes");
   scanf("%d",&n);
   float sum_tat=0,sum_wt=0,sum_rt=0;

   int i;

   //printf("\nEnter Process Number\n");
   // for(i=0;i<n;i++)
```

```
// {
//   scanf("%f",&ps[i].process_num);
// }
printf("\nEnter Process Arrival Time\n");
for(i=0;i<n;i++)
{
  scanf("%d",&ps[i].at);
}
printf("\nEnter Process Burst Time\n");
for(i=0;i<n;i++)
  scanf("%d",&ps[i].bt);
printf("\nEnter Priority\n");
for(i=0;i<n;i++)
  scanf("%d",&ps[i].priority);

while(completed!=n)
{
   //find process with min. burst time in ready queue at current time
   int max_index = -1;
   int maximum = -1;
   for(int i = 0; i < n; i++) {
      if(ps[i].at <= current_time && is_completed[i] == 0) {
         if(ps[i].priority > maximum) {
            maximum = ps[i].priority;
            max_index = i;
         }
         if(ps[i].priority== maximum) {
            if(ps[i].at < ps[max_index].at) {
               maximum= ps[i].priority;
               max_index = i;
            }
```

```
        }
      }
    }


 if(max_index==-1)
 {
    current_time++;
 }
 else
 {
   ps[max_index].start_time = current_time;
   ps[max_index].ct = ps[max_index].start_time + ps[max_index].bt;
   ps[max_index].tat = ps[max_index].ct - ps[max_index].at;
   ps[max_index].wt= ps[max_index].tat - ps[max_index].bt;
   ps[max_index].rt = ps[max_index].start_time - ps[max_index].at;
   total_idle_time += (is_first_process==true) ? 0 : (ps[max_index].start_time -  prev);

   sum_tat +=ps[max_index].tat;
   sum_wt += ps[max_index].wt;
   sum_rt += ps[max_index].rt;
   completed++;
   is_completed[max_index]=true;
   current_time = ps[max_index].ct;
   prev= current_time;
   is_first_process = false;


 }
}
//Calculate Length of Process completion cycle
```

```
    max_completion_time = INT_MIN;
    min_arrival_time = INT_MAX;
    for(int i=0;i<n;i++)
    {
        max_completion_time = findmax(max_completion_time,ps[i].ct);
        min_arrival_time = findmin(min_arrival_time,ps[i].at);
    }
    length_cycle = max_completion_time - min_arrival_time;
    cpu_utilization = (float)(length_cycle - total_idle_time)/ length_cycle;

    //Start times
    for(int i=0;i<n;i++)
    {
        printf("%d ",ps[i].start_time);
    }
    printf("\n");
    //completion times
    for(int i=0;i<n;i++)
    {
        printf("%d ",ps[i].ct);
    }
    printf("\n%.2f",sum_tat/n);
    printf("\n%.2f",sum_wt/n);
    printf("\n%.2f",sum_rt/n);
    printf("\n%.2f",n/(float)length_cycle);
    printf("\n%.2f",cpu_utilization*100);
    return 0;
}
```

3. Priority scheduling (Preemptive)

```
#include<stdio.h>
#include<stdbool.h>
struct process_struct
{

  int at;
  int bt;
  int priority;
  int ct,wt,tat,rt,start_time;
}ps[100];

int main()
{

    int n;
    bool is_completed[100]={false};
    int bt_remaining[100];
    int current_time = 0;
    int completed = 0;;
    //printf("Enter total number of processes");
    scanf("%d",&n);
    float sum_tat=0,sum_wt=0,sum_rt=0;

    int i;

    //printf("\nEnter Process Number\n");
    // for(i=0;i<n;i++)
    // {
    //   scanf("%f",&ps[i].process_num);
    // }
```

```
//printf("\nEnter Process Arrival Time\n");
for(i=0;i<n;i++)
{
 scanf("%d",&ps[i].at);
}
//printf("\nEnter Process Burst Time\n");
for(i=0;i<n;i++)
 scanf("%d",&ps[i].bt);
//printf("\nEnter Priority\n");
for(i=0;i<n;i++)
 scanf("%d",&ps[i].priority);

while(completed!=n)
{
   //find process with min. burst time in ready queue at current time
   int max_index = -1;
   int maximum = -1;
   for(int i = 0; i < n; i++) {
      if(ps[i].at <= current_time && is_completed[i] == 0) {
         if(ps[i].priority > maximum) {
            maximum = ps[i].priority;
            max_index = i;
         }
         if(ps[i].priority== maximum) {
            if(ps[i].at < ps[max_index].at) {
               maximum= ps[i].priority;
               max_index = i;
            }
         }
      }
   }
```

```
// printf("max Index=%d ",max_index);
 if(max_index==-1)
 {
    current_time++;
 }
 else
 {
   if(bt_remaining[max_index]==ps[max_index].bt)
      ps[max_index].start_time = current_time;

   bt_remaining[max_index]--;
   current_time++;

   if(bt_remaining[max_index]==0)
   {
       ps[max_index].start_time = current_time;
       ps[max_index].ct = ps[max_index].start_time + ps[max_index].bt;
       ps[max_index].tat = ps[max_index].ct - ps[max_index].at;
       ps[max_index].wt= ps[max_index].tat - ps[max_index].bt;
       ps[max_index].rt = ps[max_index].start_time - ps[max_index].at;


       sum_tat +=ps[max_index].tat;
       sum_wt += ps[max_index].wt;
       sum_rt += ps[max_index].rt;
       completed++;
       is_completed[max_index]=true;

       printf("Max=%d ", ps[max_index].ct);
   }
```

```
    }
  }
    for(int i=0;i<n;i++)
    {
       printf("%.2d ",ps[i].ct);
    }

    printf("\n%.2f",sum_tat/n);
    printf("\n%.2f",sum_wt/n);
    printf("\n%.2f",sum_rt/n);
    return 0;
}
```