OPERATING SYSTEM Assignment

CSE316

Submitted By-

Vishal Singh
Reg.No-11804269
Email - rkvishalsingh26@gmail.com
GitHub Link- https://github.com/rkvishal26/Operating-System

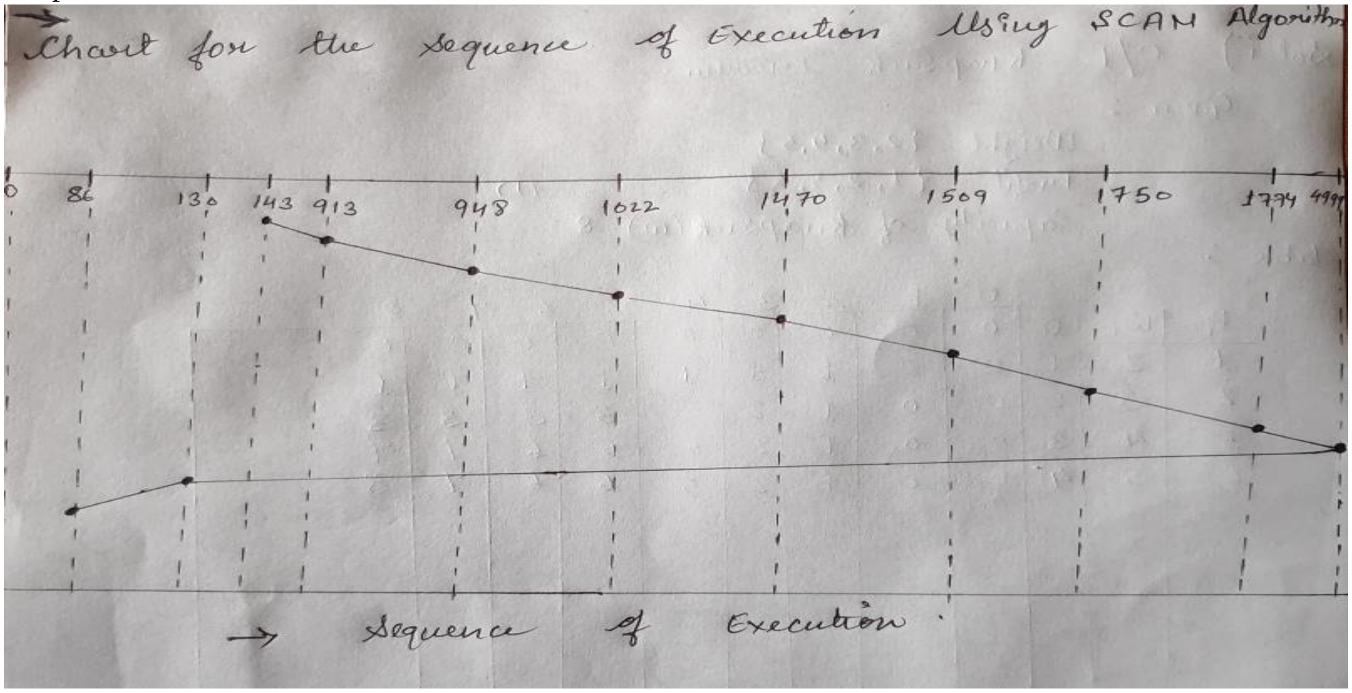
Submitted to-

Mr. Manpreet Singh
(Operating System)

Problem explanation

1. **Problem 1:**
   Since, It's given that the following problem is to be solved by SCAN algorithm.
   Therefore, according to SCAN algorithm we have to take 'head' value i.e; the current request value .Since, In the following problem it is specified that the head is moving towards the larger value, So the request that are towards the larger side are served first. After serving all the request of the larger side, it reverses its direction and serves the lower request.
   Note: If the head is moving towards the smaller value, then the smaller request are served first than the larger one.
   According to the problem,
   Head position =145, and it is moving towards the larger side, so it serves the request of the up to last limit i.e; 4999 (given) and then it reverses it direction to the smaller side to serve the smaller request.

Graph:



Formula Used-:

Total distance (in cylinders) = (|total limit -Head position| + |total limit- smallest request served|)

Total distance= (|4999-145| + |4999-86|)

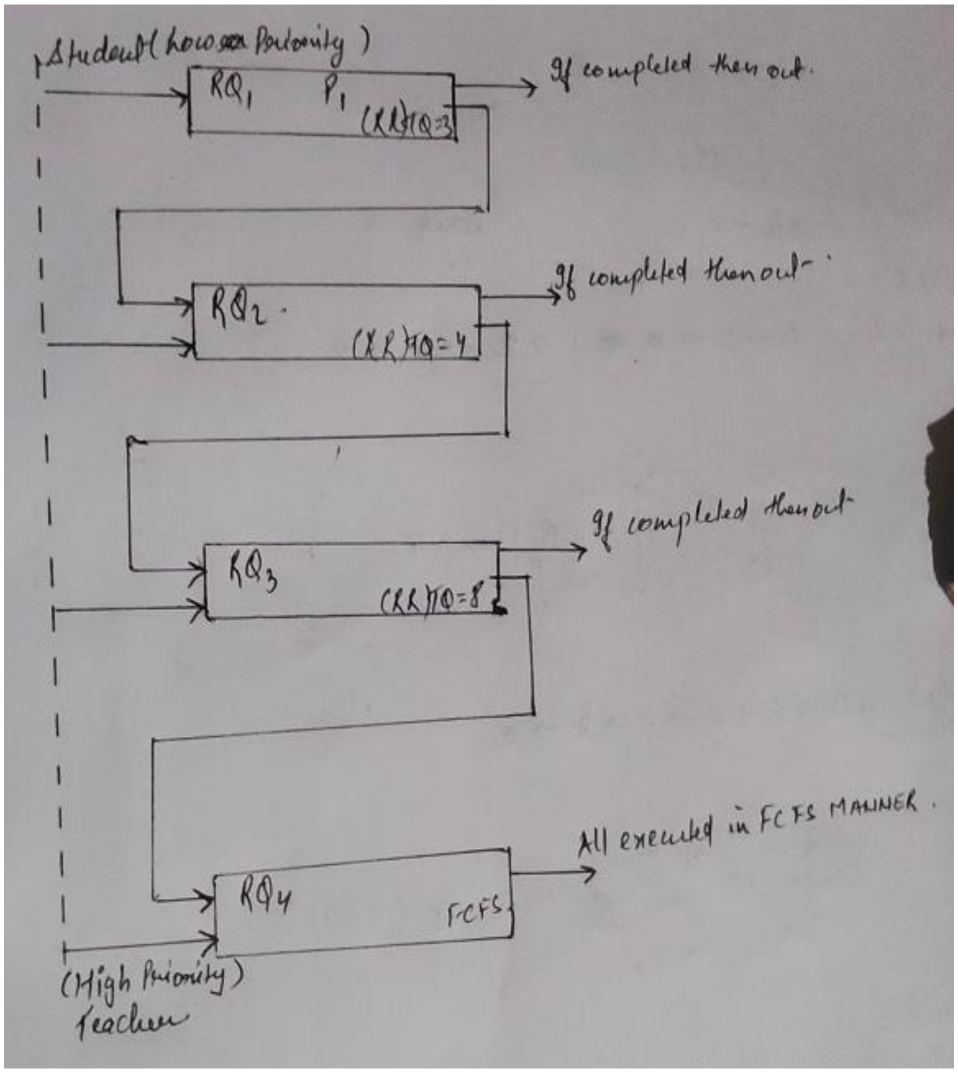
Total distance =9769

Problem 2:

According to the problem, if a teacher (high priority) is being served at a food mess and another teacher comes, he is also being served and so on. This lead to increase in the waiting time of the student to get food. Here, the food is referred to as the CPU and the teacher and student is referred to as the different processor waiting for their execution, where teacher is of high priority and student is of low priority processor.

So, in order to minimize the student (Low priority) waiting time the Multi-level Feedback Queue Scheduling is used which state that**,** It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue. Therefore, process with low priority can also execute in between the execution of high priority process. In this way the above problem is resolved.

Diagrammatic Illustration-



Algorithm used for different Problems

I) **SCAN Disk-Scheduling Algorithm-**

This is a type of disk-scheduling algorithm, in which the head starts from one ne and moves towards other end, serving all the request one by one and traverses whole of the request in a given sequence, the direction of the head is reversed and the process continues, as the head continuously scan back and forth to access the disk. This algorithm works as an elevator, so this algorithm is also known as "elevator algorithm". The request arriving behind will have to wait and the mid-range request are service are served first.

Steps involved in this algorithm-:

I) Let there is sequence of request stored in array and the 'head' is the position of disk head.

II) Let the problem represents whether the head is serving towards higher value or lower value.

III) In the direction in which the head is moving, serves all the request of that direction first.

IV) After reaching the other end, head reverses it's direction and move towards other end, serves all the request of that direction.

V) The same process repeats until all the request are served.

Advantages of SCAN disk-scheduling algorithm-:

i) High throughput.

ii) Low variance of response time.

iii) Average response time.

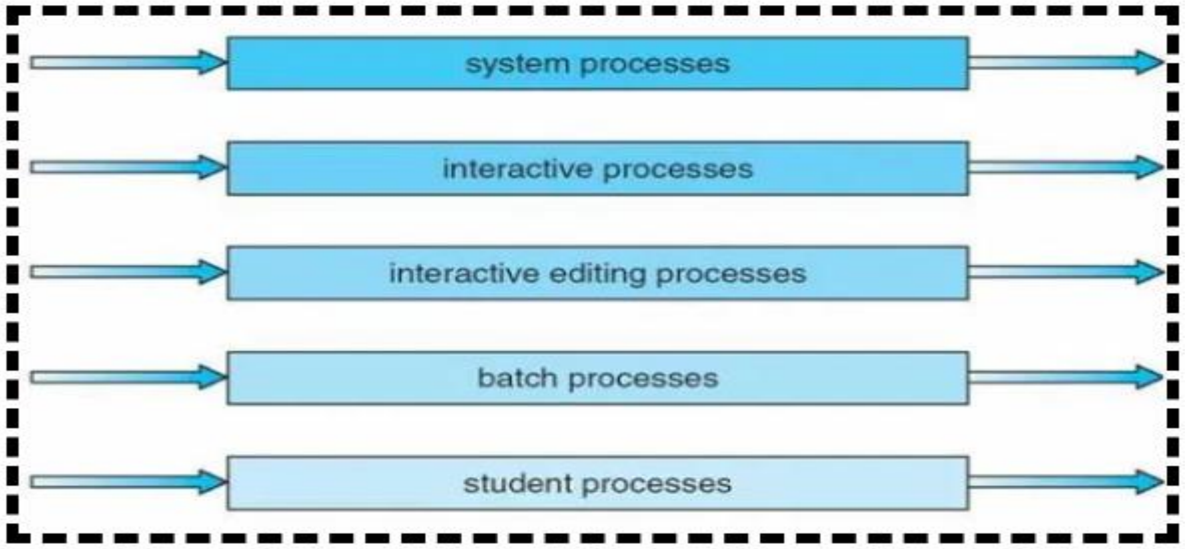
Disadvantage of SCAN disk-scheduling algorithm-:

i) Long waiting time.

ii) Multi level Feedback Queue Scheduling: It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue. So according the problem Multi-level feedback queue scheduling is used to minimize the waiting time. The algorithm allocates the jobs or processes to different queues based on their CPU execution time. If a process has a large burst-time, then it is automatically moved to a lower-priority queue. This technique helps to prevent starvation of lower priority processes too.

The algorithm prefers shorter jobs with low burst times and it prefers input/output bound processes. A process known as aging promotes lower priority jobs to a higher priority queue at regular intervals of time. An important thing to note is that there is a difference between multi-level feedback queue scheduling algorithm and multi-level queue scheduling algorithm. In the multi-level feedback queue scheduling algorithm, the processes are permanently assigned to a queue whereas, in a multilevel feedback scheduling algorithm, the processes can move between multiple queues according to their requirements. Since it uses both shortest job first (SJF) and first come first serve(FCFS). It reduces the waiting time of lower priority queue.

Diagrammatic view of Multi level Feedback Queue Scheduling**:**



Advantages

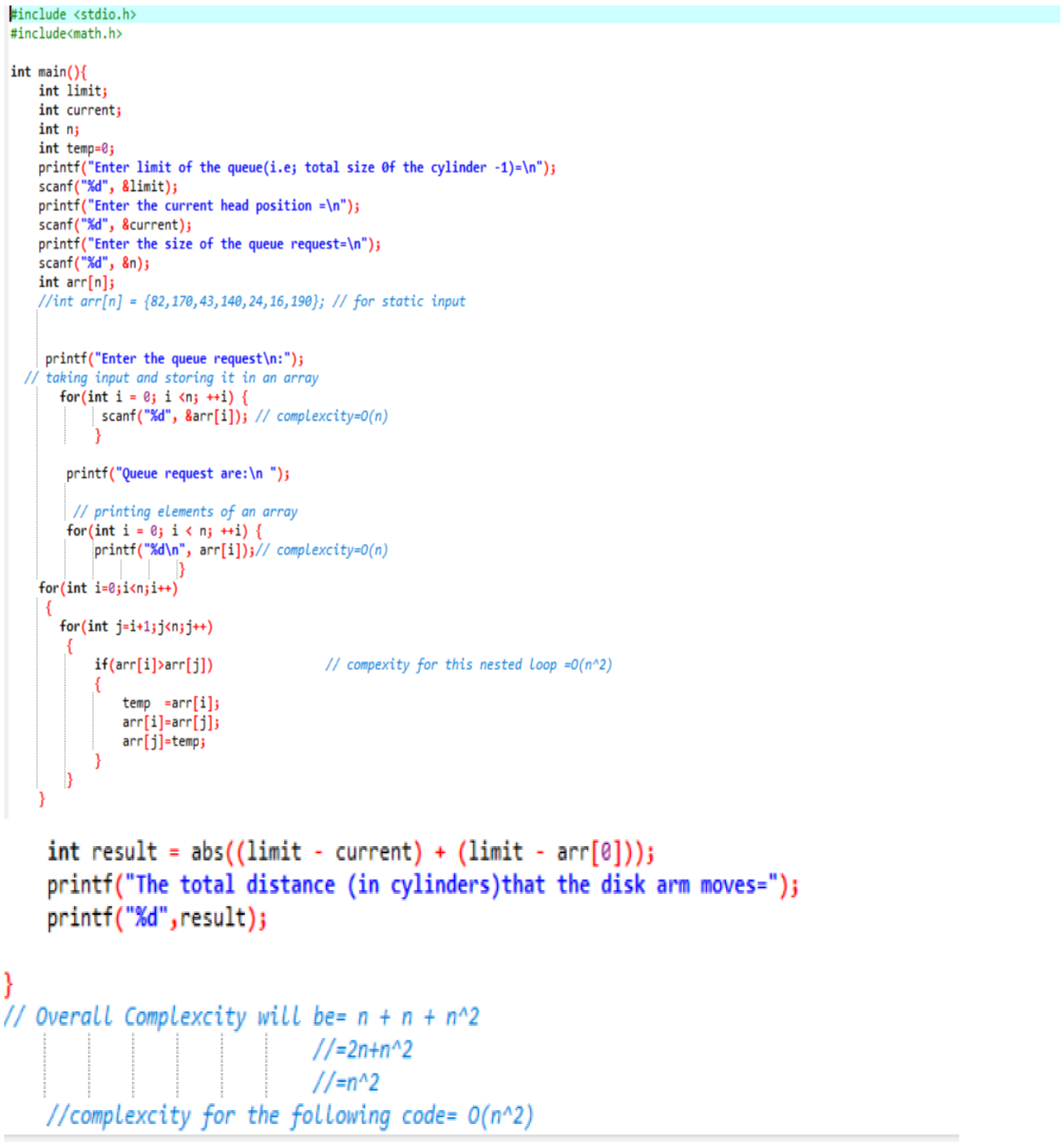
A high CPU time job that waits for too long in a lower priority queue can be moved to a higher priority queue at regular intervals of time. Hence the waiting time can be minimized.

Disadvantage

Moving the processes from one queue to another increases the CPU overhead.

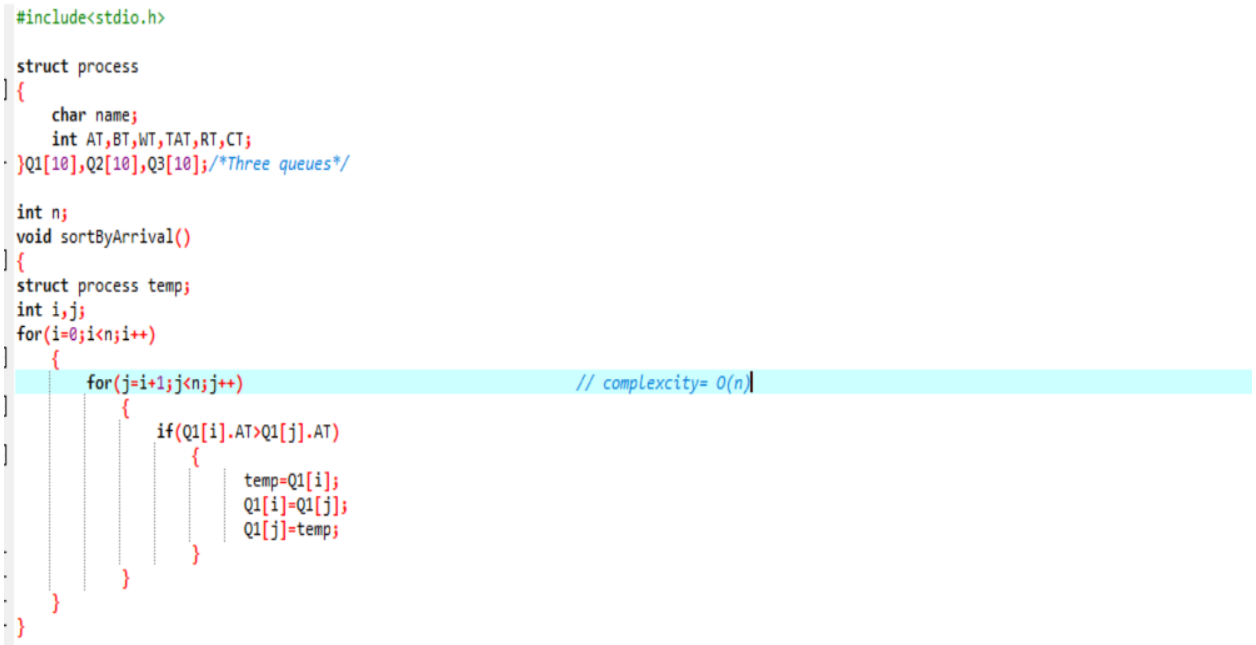
<u>Code Snippet with Complexity</u>

Problem1-:

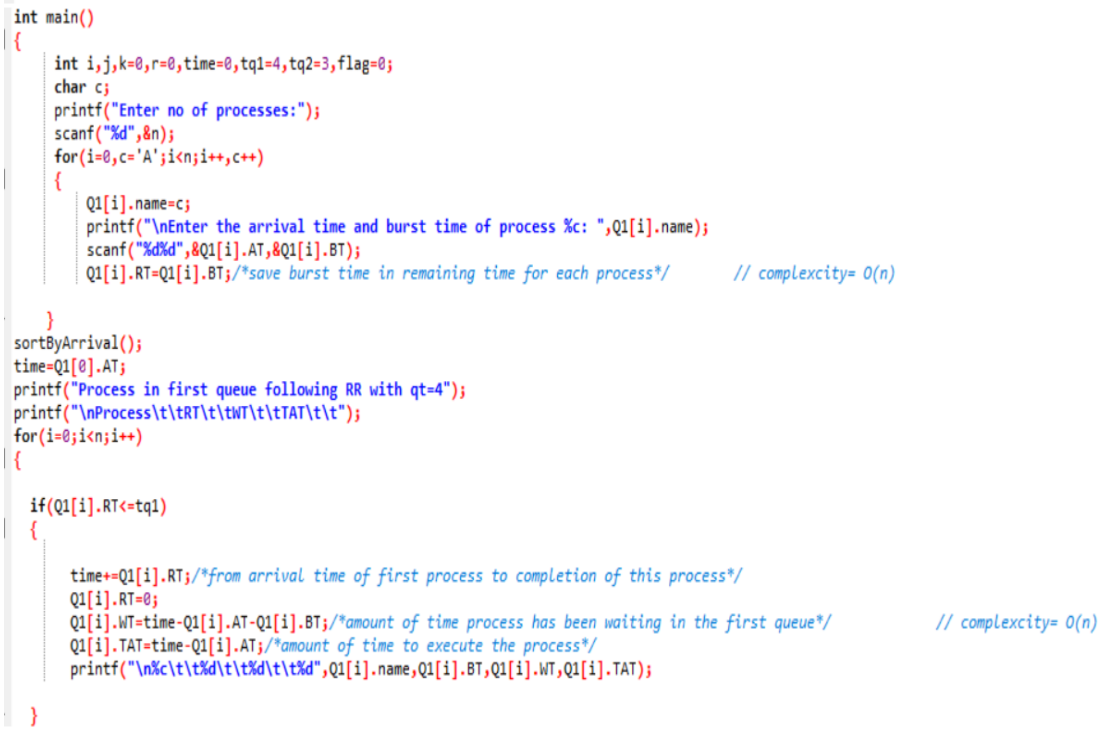


```
#include <stdio.h>
#include<math.h>

int main(){
    int limit;
    int current;
    int n;
    int temp=0;
    printf("Enter limit of the queue(i.e; total size 0f the cylinder -1)=\n");
    scanf("%d", &limit);
    printf("Enter the current head position =\n");
    scanf("%d", &current);
    printf("Enter the size of the queue request=\n");
    scanf("%d", &n);
    int arr[n];
    //int arr[n] = {82,170,43,140,24,16,190}; // for static input


     printf("Enter the queue request\n:");
  // taking input and storing it in an array
      for(int i = 0; i <n; ++i) {
            scanf("%d", &arr[i]); // complexcity=O(n)
         }

       printf("Queue request are:\n ");

        // printing elements of an array
       for(int i = 0; i < n; ++i) {
           printf("%d\n", arr[i]);// complexcity=O(n)
                       }
    for(int i=0;i<n;i++)
     {
      for(int j=i+1;j<n;j++)
       {
           if(arr[i]>arr[j])               // compexity for this nested loop =O(n^2)
           {
               temp  =arr[i];
               arr[i]=arr[j];
               arr[j]=temp;
           }
       }
    }

     int result = abs((limit - current) + (limit - arr[0]));
     printf("The total distance (in cylinders)that the disk arm moves=");
     printf("%d",result);

}
// Overall Complexcity will be= n + n + n^2
                                   //=2n+n^2
                                   //=n^2
     //complexcity for the following code= O(n^2)
```

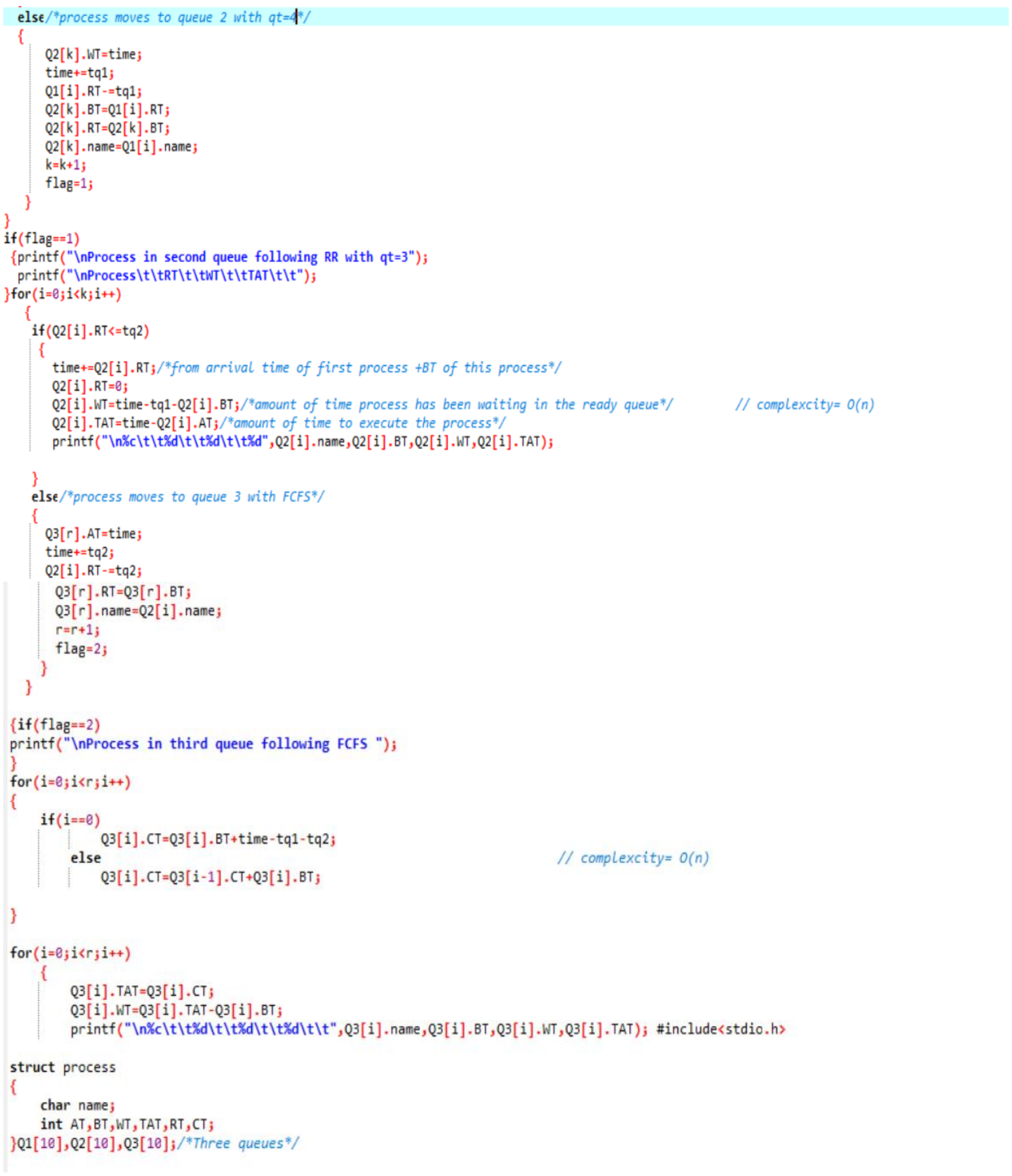
Problem 2-:



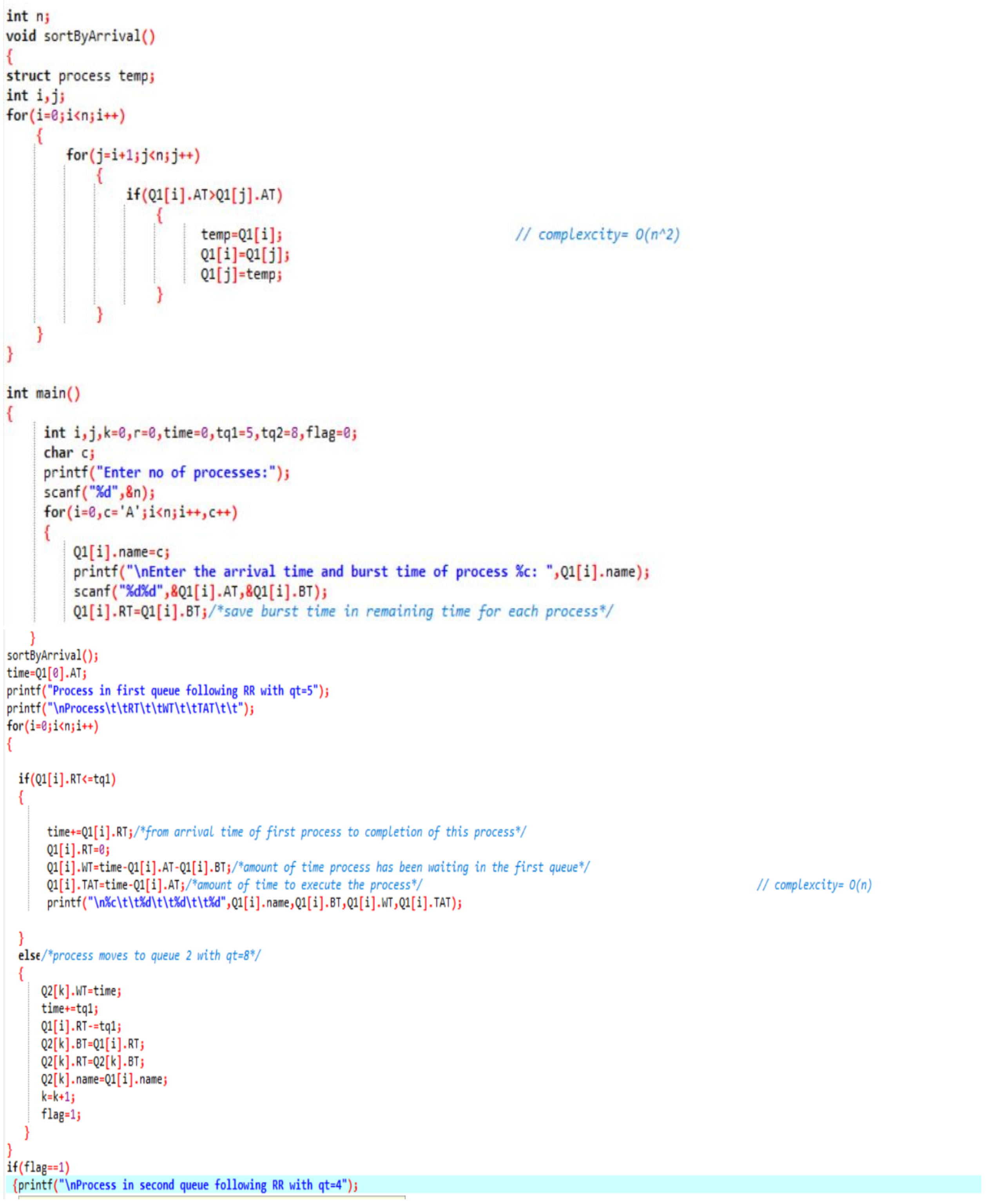
```
#include<stdio.h>

struct process
{
     char name;
     int AT,BT,WT,TAT,RT,CT;
}Q1[10],Q2[10],Q3[10];/*Three queues*/

int n;
void sortByArrival()
{
struct process temp;
int i,j;
for(i=0;i<n;i++)
     {
          for(j=i+1;j<n;j++)                                    // complexcity= O(n)
               {
                    if(Q1[i].AT>Q1[j].AT)
                         {
                              temp=Q1[i];
                              Q1[i]=Q1[j];
                              Q1[j]=temp;
                         }
               }
     }
}
```



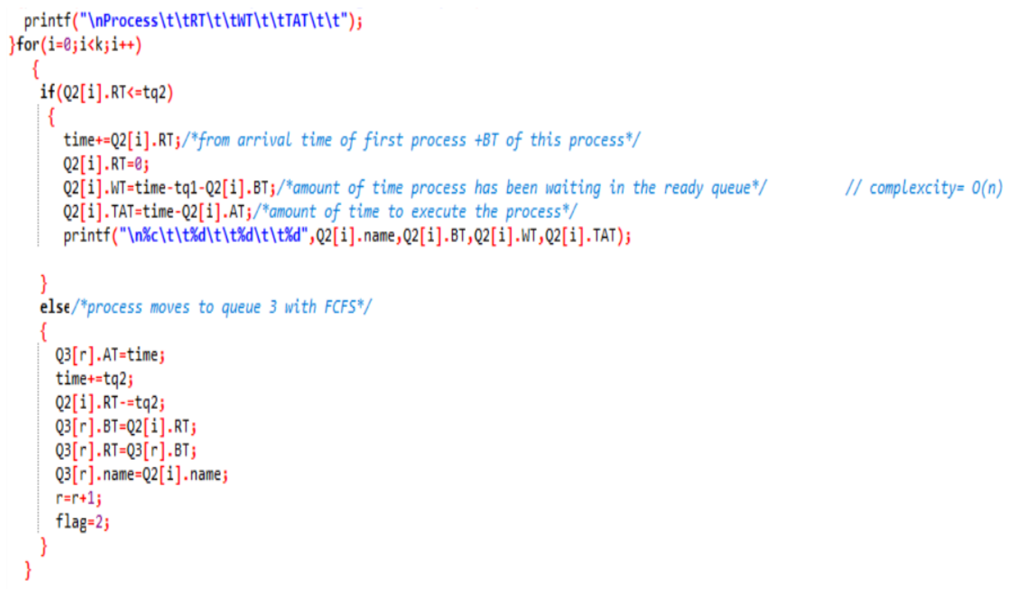
```
int main()
{
     int i,j,k=0,r=0,time=0,tq1=4,tq2=3,flag=0;
     char c;
     printf("Enter no of processes:");
     scanf("%d",&n);
     for(i=0,c='A';i<n;i++,c++)
     {
          Q1[i].name=c;
          printf("\nEnter the arrival time and burst time of process %c: ",Q1[i].name);
          scanf("%d%d",&Q1[i].AT,&Q1[i].BT);
          Q1[i].RT=Q1[i].BT;/*save burst time in remaining time for each process*/        // complexcity= O(n)

     }
sortByArrival();
time=Q1[0].AT;
printf("Process in first queue following RR with qt=4");
printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
for(i=0;i<n;i++)
{

  if(Q1[i].RT<=tq1)
  {

     time+=Q1[i].RT;/*from arrival time of first process to completion of this process*/
     Q1[i].RT=0;
     Q1[i].WT=time-Q1[i].AT-Q1[i].BT;/*amount of time process has been waiting in the first queue*/          // complexcity= O(n)
     Q1[i].TAT=time-Q1[i].AT;/*amount of time to execute the process*/
     printf("\n%c\t\t%d\t\t%d\t\t%d",Q1[i].name,Q1[i].BT,Q1[i].WT,Q1[i].TAT);

  }
```



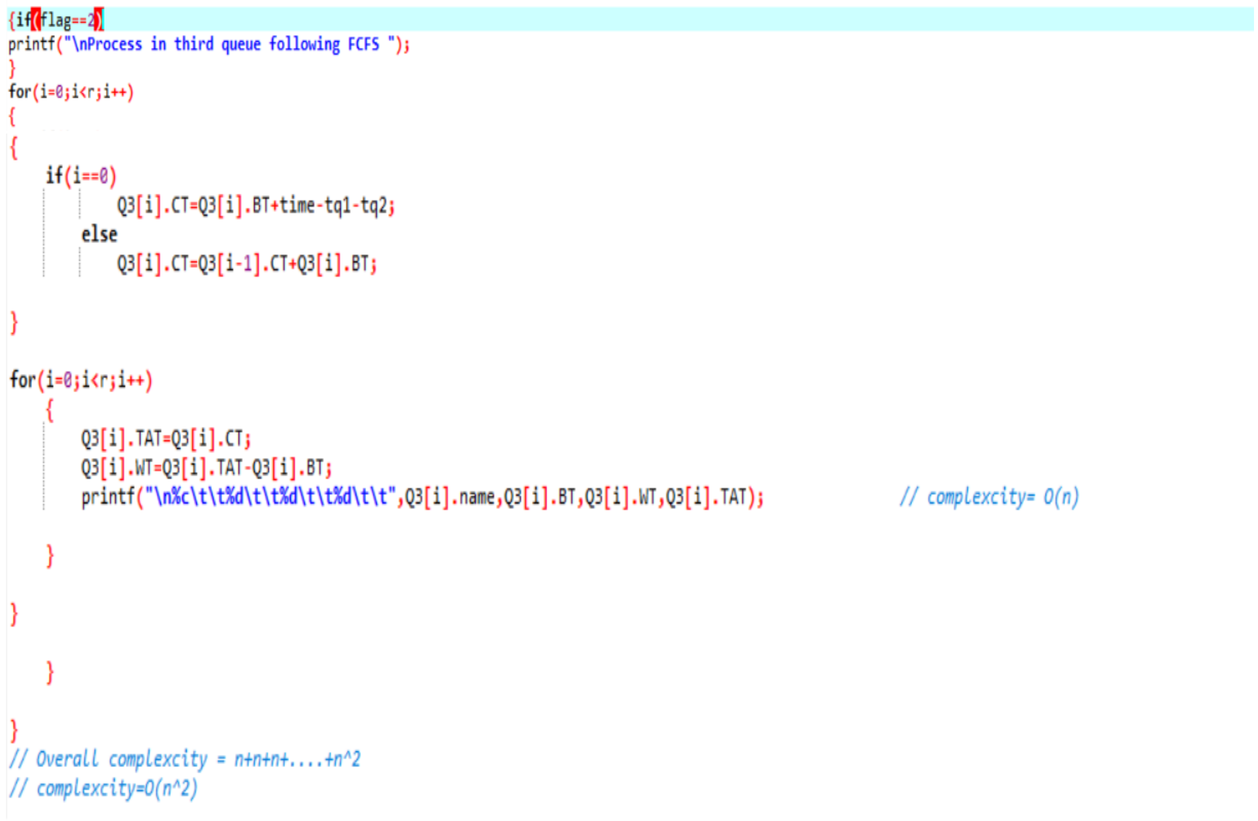
```
  else/*process moves to queue 2 with qt=4*/
  {
      Q2[k].WT=time;
      time+=tq1;
      Q1[i].RT-=tq1;
      Q2[k].BT=Q1[i].RT;
      Q2[k].RT=Q2[k].BT;
      Q2[k].name=Q1[i].name;
      k=k+1;
      flag=1;
   }
}
if(flag==1)
 {printf("\nProcess in second queue following RR with qt=3");
  printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
}for(i=0;i<k;i++)
   {
    if(Q2[i].RT<=tq2)
     {
       time+=Q2[i].RT;/*from arrival time of first process +BT of this process*/
       Q2[i].RT=0;
       Q2[i].WT=time-tq1-Q2[i].BT;/*amount of time process has been waiting in the ready queue*/         // complexcity= O(n)
       Q2[i].TAT=time-Q2[i].AT;/*amount of time to execute the process*/
       printf("\n%c\t\t%d\t\t%d\t\t%d",Q2[i].name,Q2[i].BT,Q2[i].WT,Q2[i].TAT);

    }
    else/*process moves to queue 3 with FCFS*/
    {
      Q3[r].AT=time;
      time+=tq2;
      Q2[i].RT-=tq2;
       Q3[r].RT=Q3[r].BT;
       Q3[r].name=Q2[i].name;
       r=r+1;
       flag=2;
     }
  }

 {if(flag==2)
 printf("\nProcess in third queue following FCFS ");
 }
 for(i=0;i<r;i++)
 {
     if(i==0)
             Q3[i].CT=Q3[i].BT+time-tq1-tq2;
         else                                                   // complexcity= O(n)
             Q3[i].CT=Q3[i-1].CT+Q3[i].BT;

 }

 for(i=0;i<r;i++)
     {
         Q3[i].TAT=Q3[i].CT;
         Q3[i].WT=Q3[i].TAT-Q3[i].BT;
         printf("\n%c\t\t%d\t\t%d\t\t%d\t\t",Q3[i].name,Q3[i].BT,Q3[i].WT,Q3[i].TAT); #include<stdio.h>

 struct process
 {
     char name;
     int AT,BT,WT,TAT,RT,CT;
 }Q1[10],Q2[10],Q3[10];/*Three queues*/
```



```c
int n;
void sortByArrival()
{
struct process temp;
int i,j;
for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
            {
                if(Q1[i].AT>Q1[j].AT)
                    {
                        temp=Q1[i];                                    // complexcity= O(n^2)
                        Q1[i]=Q1[j];
                        Q1[j]=temp;
                    }
            }
    }
}

int main()
{
    int i,j,k=0,r=0,time=0,tq1=5,tq2=8,flag=0;
    char c;
    printf("Enter no of processes:");
    scanf("%d",&n);
    for(i=0,c='A';i<n;i++,c++)
    {
        Q1[i].name=c;
        printf("\nEnter the arrival time and burst time of process %c: ",Q1[i].name);
        scanf("%d%d",&Q1[i].AT,&Q1[i].BT);
        Q1[i].RT=Q1[i].BT;/*save burst time in remaining time for each process*/
    }
sortByArrival();
time=Q1[0].AT;
printf("Process in first queue following RR with qt=5");
printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
for(i=0;i<n;i++)
{

  if(Q1[i].RT<=tq1)
  {

      time+=Q1[i].RT;/*from arrival time of first process to completion of this process*/
      Q1[i].RT=0;
      Q1[i].WT=time-Q1[i].AT-Q1[i].BT;/*amount of time process has been waiting in the first queue*/
      Q1[i].TAT=time-Q1[i].AT;/*amount of time to execute the process*/                        // complexcity= O(n)
      printf("\n%c\t\t%d\t\t%d\t\t%d",Q1[i].name,Q1[i].BT,Q1[i].WT,Q1[i].TAT);

  }
  else/*process moves to queue 2 with qt=8*/
  {
     Q2[k].WT=time;
     time+=tq1;
     Q1[i].RT-=tq1;
     Q2[k].BT=Q1[i].RT;
     Q2[k].RT=Q2[k].BT;
     Q2[k].name=Q1[i].name;
     k=k+1;
     flag=1;
  }
}
if(flag==1)
 {printf("\nProcess in second queue following RR with qt=4");
```



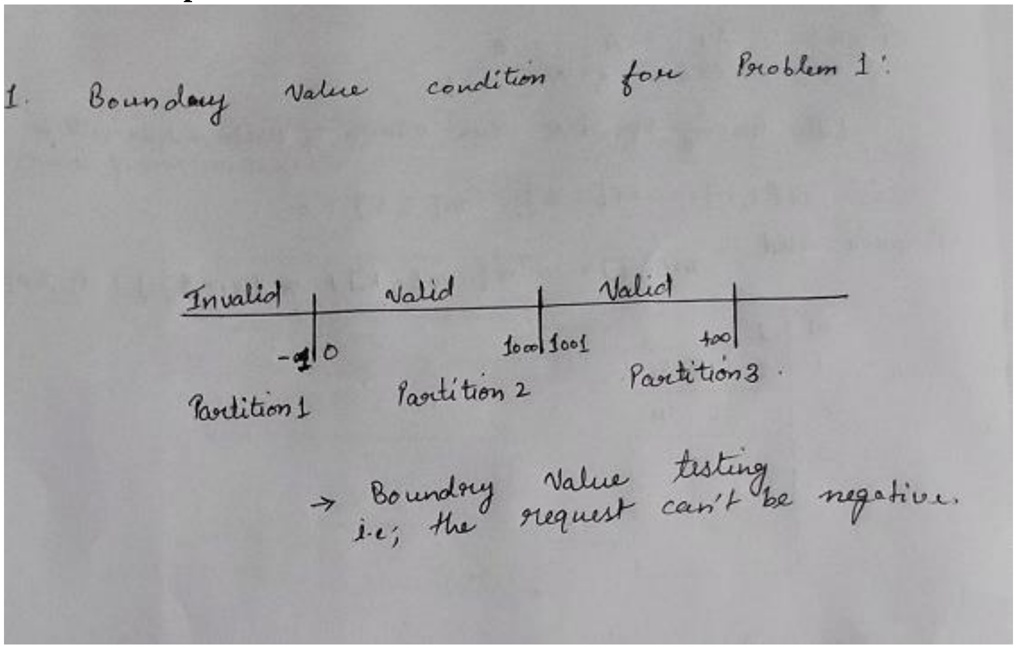
```
  printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
}for(i=0;i<k;i++)
   {
    if(Q2[i].RT<=tq2)
     {
       time+=Q2[i].RT;/*from arrival time of first process +BT of this process*/
       Q2[i].RT=0;
       Q2[i].WT=time-tq1-Q2[i].BT;/*amount of time process has been waiting in the ready queue*/          // complexcity= O(n)
       Q2[i].TAT=time-Q2[i].AT;/*amount of time to execute the process*/
       printf("\n%c\t\t%d\t\t%d\t\t%d",Q2[i].name,Q2[i].BT,Q2[i].WT,Q2[i].TAT);

    }
    else/*process moves to queue 3 with FCFS*/
    {
      Q3[r].AT=time;
      time+=tq2;
      Q2[i].RT-=tq2;
      Q3[r].BT=Q2[i].RT;
      Q3[r].RT=Q3[r].BT;
      Q3[r].name=Q2[i].name;
      r=r+1;
      flag=2;
    }
  }
```



```
{if(flag==2)
printf("\nProcess in third queue following FCFS ");
}
for(i=0;i<r;i++)
{
{
     if(i==0)
             Q3[i].CT=Q3[i].BT+time-tq1-tq2;
         else
             Q3[i].CT=Q3[i-1].CT+Q3[i].BT;

}

for(i=0;i<r;i++)
     {
         Q3[i].TAT=Q3[i].CT;
         Q3[i].WT=Q3[i].TAT-Q3[i].BT;
         printf("\n%c\t\t%d\t\t%d\t\t%d\t\t",Q3[i].name,Q3[i].BT,Q3[i].WT,Q3[i].TAT);                // complexcity= O(n)

     }

}

     }

}
// Overall complexcity = n+n+n+....+n^2
// complexcity=O(n^2)
```

Boundary Conditions

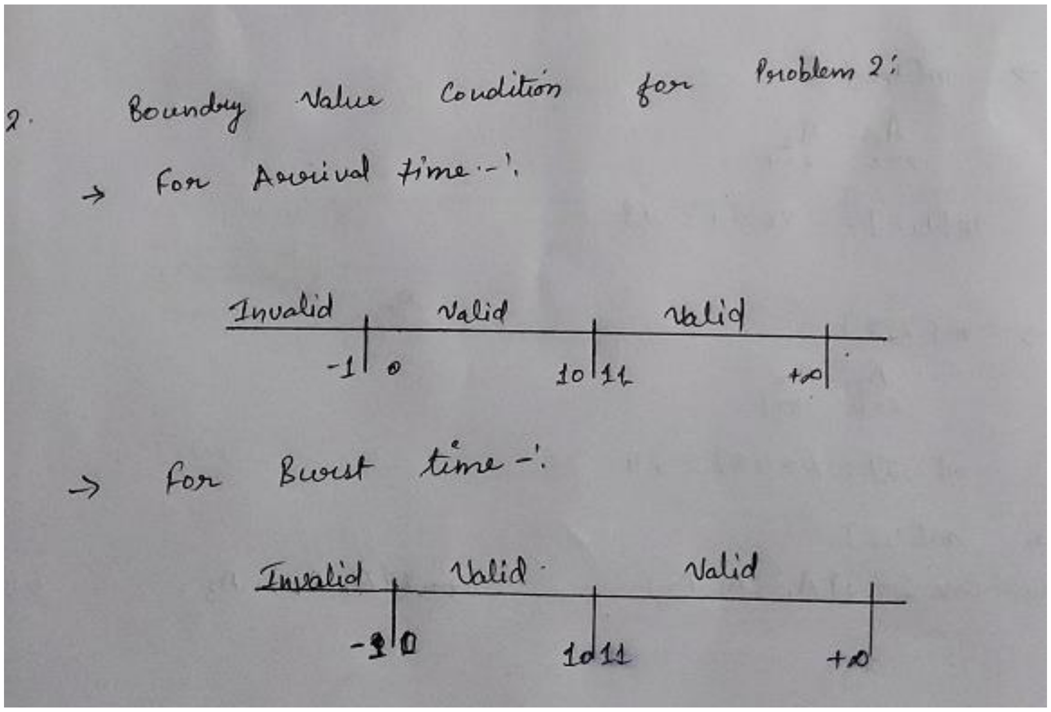
Boundary Condition for problem 1:



1. Boundary Value condition for Problem 1:

→ Boundry Value testing i.e; the request can't be negative.

Boundary Condition for problem 2:



2. Boundry Value Condition for Problem 2:

→ For Arrival time:-

→ For Burst time:-

Test Cases

1. **Test case for problem 1:**

Test Case Id	**Test case description**	**Test Data**	**Result**	**Pass/Fail**
1.	Check response for the cylinder size.	4999	No error.	Pass.
2.	Check response for header position.	143	No error.	Pass.
3.	Check response for request size.	9	No error.	Pass.
4.	Check response for request size.	-8	Error.	Failed.
5.	Check response for request served.	130	No error.	Pass.

2. **Test Cases for problem 2:**

Test Case Id	**Test case description**	**Test Data**	**Result**	**Pass/Fail**
1.	Check response for total number of process.	3	No error.	Pass.
2.	Check response for total number of process.	-1	Error.	Failed
3.	Check response for Arrival Time.	2	No error.	Pass.
4.	Check response for Burst Time.	4	No error.	Pass.
4.	Check response for Priority.	2	No error.	Pass.