SJF, FCFS And Round robin Implementation algorithm analysis

Name: Kshitij Kumar Sharma   Roll No: 1905514

----

1. SJF:

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
struct processDetails
{
 int processid;
 int arrival;
 int cpu;
};

int main(int argc, char* argv[])
{
 int i,n,a,b,c,j;
 n=(argc-1)/3;
 int tt[n],rt[n],ct[n];
 struct processDetails p[n],t;
 i=1;
 j=0;
 while(i<argc)
  {
   p[j].processid=atoi(argv[i]);
   i++;
   p[j].arrival=atoi(argv[i]);
   i++;
   p[j].cpu=atoi(argv[i]);
   i++;
   j++;
  }
 for(i=0;i<n;i++)
  {
   for(j=0;j<n-1;j++)
    {
     if(p[j].arrival>p[j+1].arrival)
      {
       t=p[j];
       p[j]=p[j+1];
       p[j+1]=t;
    }
   }
  }
for(i=0;i<n;i++)
 {
  for(j=0;j<n-1;j++)
   {
    if(p[j].arrival==p[j+1].arrival && p[j].cpu>p[j+1].cpu)
     {
      t=p[j];
      p[j]=p[j+1];
      p[j+1]=t;
    }
   }
  }
 c=0;
 int min,pos;
 for(i=1;i<n-1;i++)
 {
  c=c+p[i-1].cpu;
  min=p[i].cpu;
  pos=i;
  for(j=i+1;j<n;j++)
   {
    if(p[j].arrival < c && min > p[j].cpu)
      {
      min=p[j].cpu;
      pos=j;
      }
   }
  t=p[i];
  p[i]=p[pos];
  p[pos]=t;
 }
 int ts=0,id,ideal=0;
for(i=0;i<n;i++)
 {
  if(ts<p[i].arrival)
   {
     id=p[i].arrival-ts;
   }
   else
   {
    id=0;
   }
   ideal=ideal+id;
   ts=ts+id+p[i].cpu;
   ct[i]= ts;
   tt[i]=ct[i]-p[i].arrival;
   rt[i]=ct[i]-p[i].cpu-p[i].arrival;
  }
 /*
 printf("\n\n");
 printf("pid \tarrival \tcpu \t ct \t tt \t rt \n");
 for(i=0;i<n;i++)
 {
            printf("%d\t%d\t\t      %d      \t      %d      \t      %d      \t      %d\
n",p[i].processid,p[i].arrival,p[i].cpu,ct[i],tt[i],rt[i]);
 }
 */
 int utilization = (((ct[n-1]-ideal)*100)/ct[n-1]);
 float avg_tt=0,avg_rt=0,avg_ct=0;
 for(i=0;i<n;i++)
 {
        avg_tt=avg_tt+tt[i];
        avg_rt=avg_rt+rt[i];
        avg_ct=avg_ct+ct[i];
 }
 avg_tt=(avg_tt/n);
 avg_rt=(avg_rt/n);
 avg_ct=(avg_ct/n);
 // printf("Average TT = %f \n",avg_tt);
 // printf("Average WT = %f \n",avg_rt);
 FILE *f;
 f=fopen("sjf.txt","a");
 fprintf(f,"%f",avg_rt);
 fprintf(f," ");
 fprintf(f,"%f",avg_tt);
 fprintf(f," ");
 fprintf(f,"%f",avg_ct);
 fprintf(f,"\n");
 /*
 printf("CPU Ideal time = %d \n",ideal);
 printf("CPU Utilization = %d percent\n",utilization);
 */
}
```

2. FCFS:

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
struct processDetails
{
 int processid;
 int arrival;
 int cpu;
};

int main(int argc, char* argv[])
{
 int i,n,a,b,c,j;
 n=(argc-1)/3;
 int tt[n],rt[n],ct[n];
 struct processDetails p[n],t;
 i=1;
 j=0;
 while(i<argc)
  {
   p[j].processid=atoi(argv[i]);
   i++;
   p[j].arrival=atoi(argv[i]);
   i++;
   p[j].cpu=atoi(argv[i]);
   i++;
   j++;
  }
 for(i=0;i<n;i++)
  {
   for(j=0;j<n-1;j++)
    {
     if(p[j].arrival>p[j+1].arrival)
      {
       t=p[j];
       p[j]=p[j+1];
       p[j+1]=t;
      }
    }
   }
 int ts=p[0].arrival;
 int id,ideal=0;
 for(i=0;i<n;i++)
  {
   if(p[i].arrival> ts)
    {
      id=p[i].arrival-ts;
    }
   else
    {
     id=0;
    }
   ideal=ideal+id;
   ts=ts+p[i].cpu+id;
   ct[i]=ts;
   tt[i]=ct[i]-p[i].arrival;
   rt[i]=ct[i]-p[i].cpu-p[i].arrival;
  }
/*
 printf("\n\n");
 printf("pid \tarrival \tcpu \t ct \t tt \t wt \n");
 for(i=0;i<n;i++)
 {
		printf("%d\t%d\t\t	%d	\t	%d	\t	%d	\t	%d\
n",p[i].processid,p[i].arrival,p[i].cpu,ct[i],tt[i],rt[i]);
 }
 */
 float avg_tt=0,avg_rt=0,avg_ct=0;
 for(i=0;i<n;i++)
 {
	avg_tt=avg_tt+tt[i];
	avg_rt=avg_rt+rt[i];
	avg_ct=avg_ct+ct[i];
 }
 avg_tt=(avg_tt/n);
 avg_rt=(avg_rt/n);
 avg_ct=(avg_ct/n);
// printf("Average TT = %f \n",avg_tt);
// printf("Average WT = %f \n",avg_rt);
 FILE *f;
 f=fopen("fcfs.txt","a");
 fprintf(f,"%f",avg_rt);
 fprintf(f," ");
 fprintf(f,"%f",avg_tt);
 fprintf(f," ");
 fprintf(f,"%f",avg_ct);
 fprintf(f,"\n");
 /*
 printf("CPU Ideal time = %d \n",ideal);
 printf("CPU Utilization = %d percent\n",utilization);
 */
}
```

3. Round Robin:

<u>Implementation</u>:

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

struct process
 {
  int process_id;
  int arrival,cpu;
 };

 struct status
 {
  int pid;
  int timeleft;
  int starttime;
  int endtime;
 };

int max_endtime(struct status a[] , int n)
 {
  int i,m=a[0].endtime;
  for(i=0;i<n;i++)
   {
    if(m<a[i].endtime)
     m=a[i].endtime;
   }
 }

int is_not_complete(struct status a[],int n)
 {
   int p=a[0].timeleft,i;
   for(i=0;i<n;i++)
    p=p || a[i].timeleft;
   return p;
 }
```

```
void main(int argc, char* argv[])
 {
  int n,i,j,q,time=0;
  n=(argc-1)/3;
  i=1;j=0;
  struct process p[n],t;
  struct status s[n];
  while(i<argc)
   {
     p[j].process_id= atoi(argv[i]);
     i++;
     p[j].arrival= atoi(argv[i]);
     i++;
     p[j].cpu= atoi(argv[i]);
     i++;
     j++;
   }
  for(i=0;i<n;i++)
   {
    for(j=0;j<n-1;j++)
     {
      if(p[j].arrival > p[j+1].arrival )
       {
        t=p[j];
        p[j]=p[j+1];
        p[j+1]=t;
       }
     }
   }
  for(i=0;i<n;i++)
   {
    s[i].pid = p[i].process_id;
    s[i].timeleft=p[i].cpu;
    s[i].starttime=-1;
    s[i].endtime=-1;
   }
  //printf("Enter the Quantum time : ");
  //scanf("%d",&q);
  q=2;
  int it=0,ideal=0;
  while(is_not_complete(s,n))
   {
     it=0;
     for(i=0;i<n;i++)
```

```
      {
       if(p[i].arrival <= time && s[i].timeleft!=0)
        {
         it++;
         if(s[i].starttime==-1)
           s[i].starttime=time;

          if(s[i].timeleft<q)
           {
            time=time+s[i].timeleft;
            s[i].timeleft=0;
            s[i].endtime=time;
           }
          else
           {
            time=time+q;
            s[i].timeleft=s[i].timeleft-q;
            s[i].endtime=time;
           }
         }
      }
     if(it==0)
      {
       ideal=ideal+1;
       time++;
      }
   }
 float avg_tt=0,avg_wt=0,avg_ct=0;
 //printf("\n\n");
 //printf("pid \tarrival \tcpu \t ct \t tt \t wt \n");
 for(i=0;i<n;i++)
 {
        //       printf("%d\t%d\t\t       %d       \t       %d       \t       %d       \t       %d\
n",p[i].process_id,p[i].arrival,p[i].cpu,s[i].endtime,s[i].endtime-
p[i].arrival,s[i].endtime-p[i].arrival-p[i].cpu);
          avg_wt=avg_wt+(s[i].endtime-p[i].arrival-p[i].cpu);
          avg_tt=avg_tt+(s[i].endtime);
          avg_ct=avg_ct+(s[i].endtime-p[i].arrival);
 }
 avg_wt=avg_wt/n;
 int et=max_endtime(s,n);
 int utilization = (((et-ideal)*100)/et);
 FILE *f;
 f=fopen("round-robin.txt","a");
 fprintf(f,"%f",avg_wt);
 fprintf(f," ");
 fprintf(f,"%f",avg_tt);
 fprintf(f," ");
 fprintf(f,"%f",avg_ct);
 fprintf(f,"\n");
 /*
 printf("CPU Ideal time = %d \n",ideal);
 printf("CPU Utilization = %d percent\n",utilization);
 */
}
```

Analysis

1. **Generation of test cases**: done by the following code named as generate.c which generates random 2D array of size [50][12] and stores it into a file named array.txt.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

void main()
{
        int a[50][12],i,j;
        FILE *f;
        for(i=0;i<50;i++)
        {
                for(j=0;j<12;j++)
                {
                a[i][j]= (rand()% (50-1+1));
                }
        }
        f=fopen("array.txt","a");
        for(i=0;i<50;i++)
        {
                for(j=0;j<12;j++)
                {
                        fprintf(f,"%d",a[i][j]);
                        fprintf(f,"%c",',');
                }
                fprintf(f,"\n");
        }
        fclose(f);
}
```

Note: I have executed this code several times to generate total 1000 test cases.

2. **Automating execution of the three algorithms:** for that I have written a shell script which takes care of it.

```
#!/bin/bash
gcc sjf.c
cat array.txt | while read line; do
  ./a.out $line
done
gcc fcfs.c
cat array.txt | while read line; do
 ./a.out $line
done
gcc round_robin.c
cat array.txt | while read line; do
 ./a.out $line
done
```

3. **Plotting the graph:** this I have done using a python code

```
import matplotlib.pyplot as plt
import numpy as np

fcfs_ct=[]
fcfs_tt=[]
fcfs_wt=[]
sjf_ct=[]
sjf_tt=[]
sjf_wt=[]
rr_ct=[]
rr_tt=[]
rr_wt=[]

with open("fcfs.txt","r") as fp:
	lines=fp.readlines()
	for line in lines:
		word=line.split()
		fcfs_wt.append(float(word[0]))
		fcfs_tt.append(float(word[1]))
		fcfs_ct.append(float(word[2]))

with open("sjf.txt","r") as fp:
	lines=fp.readlines()
	for line in lines:
		word=line.split()
		sjf_wt.append(float(word[0]))
```

```
                sjf_tt.append(float(word[1]))
                sjf_ct.append(float(word[2]))

with open("round-robin.txt","r") as fp:
        lines=fp.readlines()
        for line in lines:
                word=line.split()
                rr_wt.append(float(word[0]))
                rr_tt.append(float(word[1]))
                rr_ct.append(float(word[2]))

fcfs_ct=np.array(fcfs_ct)
fcfs_tt=np.array(fcfs_tt)
fcfs_wt=np.array(fcfs_wt)

sjf_ct=np.array(sjf_ct)
sjf_tt=np.array(sjf_tt)
sjf_wt=np.array(sjf_wt)

rr_ct=np.array(rr_ct)
rr_tt=np.array(rr_tt)
rr_wt=np.array(rr_wt)

x=[]
for i in range(1,1001):
        x.append(i)
x=np.array(x)

#print(len(rr_ct))
#print(len(x))

plt.scatter(x,fcfs_ct,color='r')
plt.scatter(x,sjf_ct,color='g')
plt.scatter(x,rr_ct,color='b')
plt.legend(['FCFS','SJF','RR'])
plt.xlabel("Test cases")
plt.ylabel("Completion time")
plt.show()

plt.scatter(x,fcfs_tt,color='r')
plt.scatter(x,sjf_tt,color='g')
plt.scatter(x,rr_tt,color='b')
plt.legend(['FCFS','SJF','RR'])
plt.xlabel("Test cases")
plt.ylabel("Turn around time")
plt.show()

plt.scatter(x,fcfs_wt,color='r')
plt.scatter(x,sjf_wt,color='g')
plt.scatter(x,rr_wt,color='b')
plt.legend(['FCFS','SJF','RR'])
plt.xlabel("Test cases")
plt.ylabel("Waiting time")
plt.show()
```

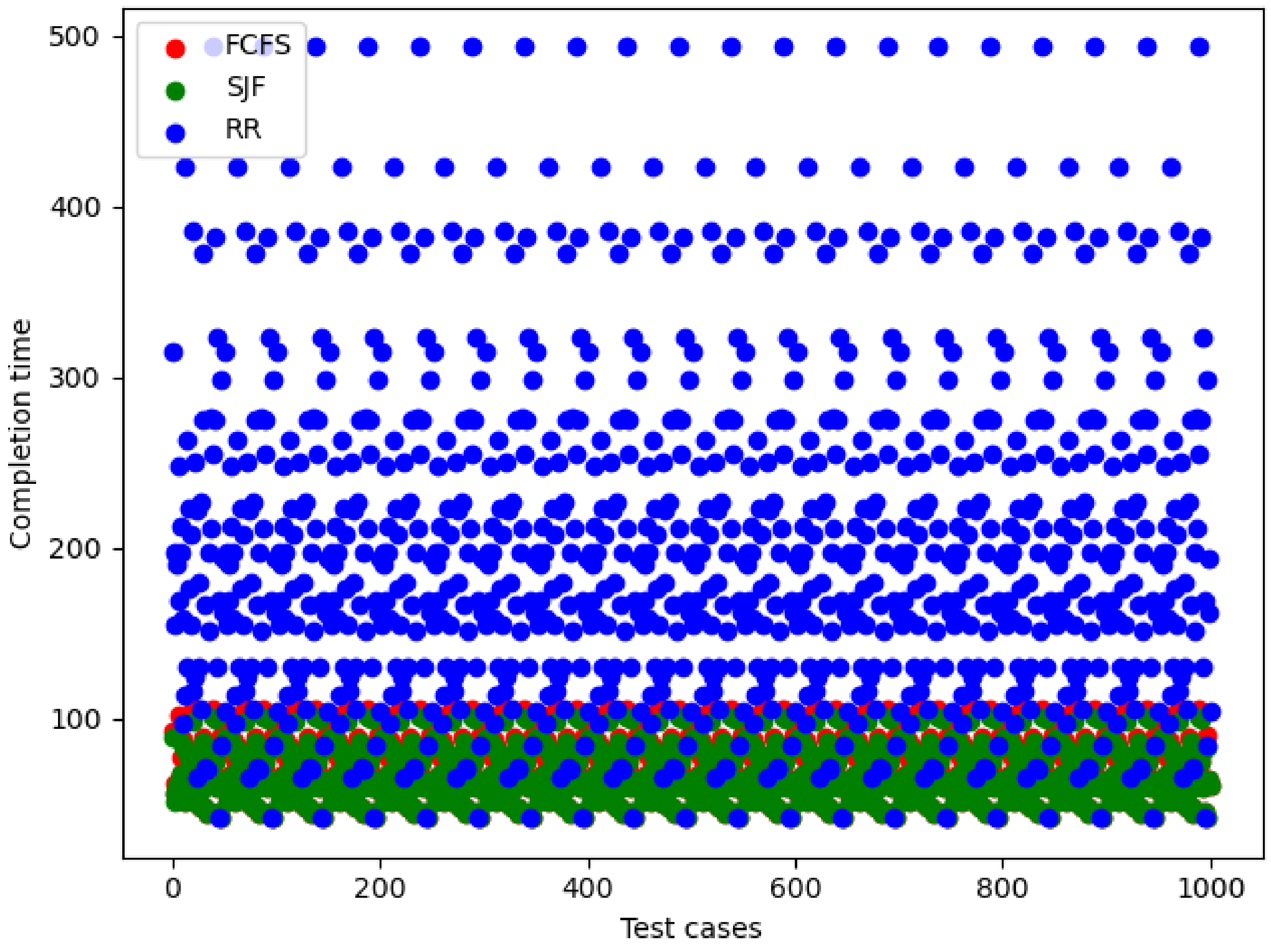

Figure 1: Completion time

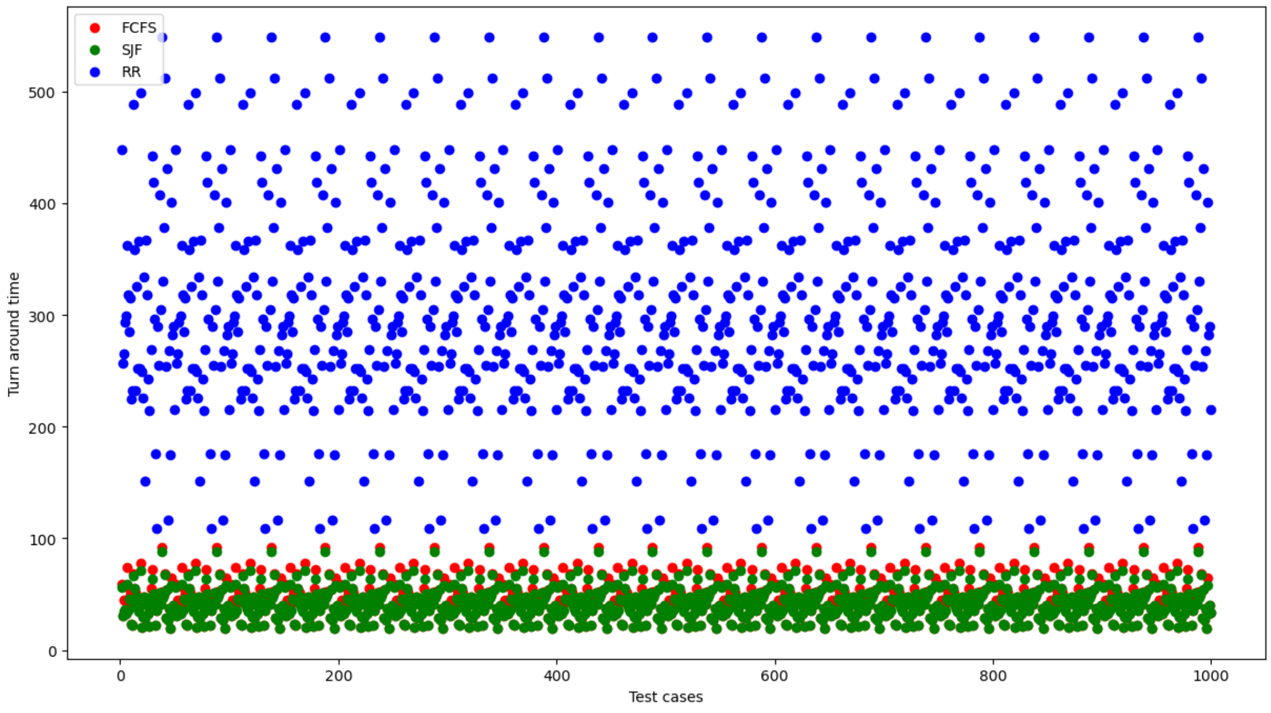

Figure 2: Turn around time

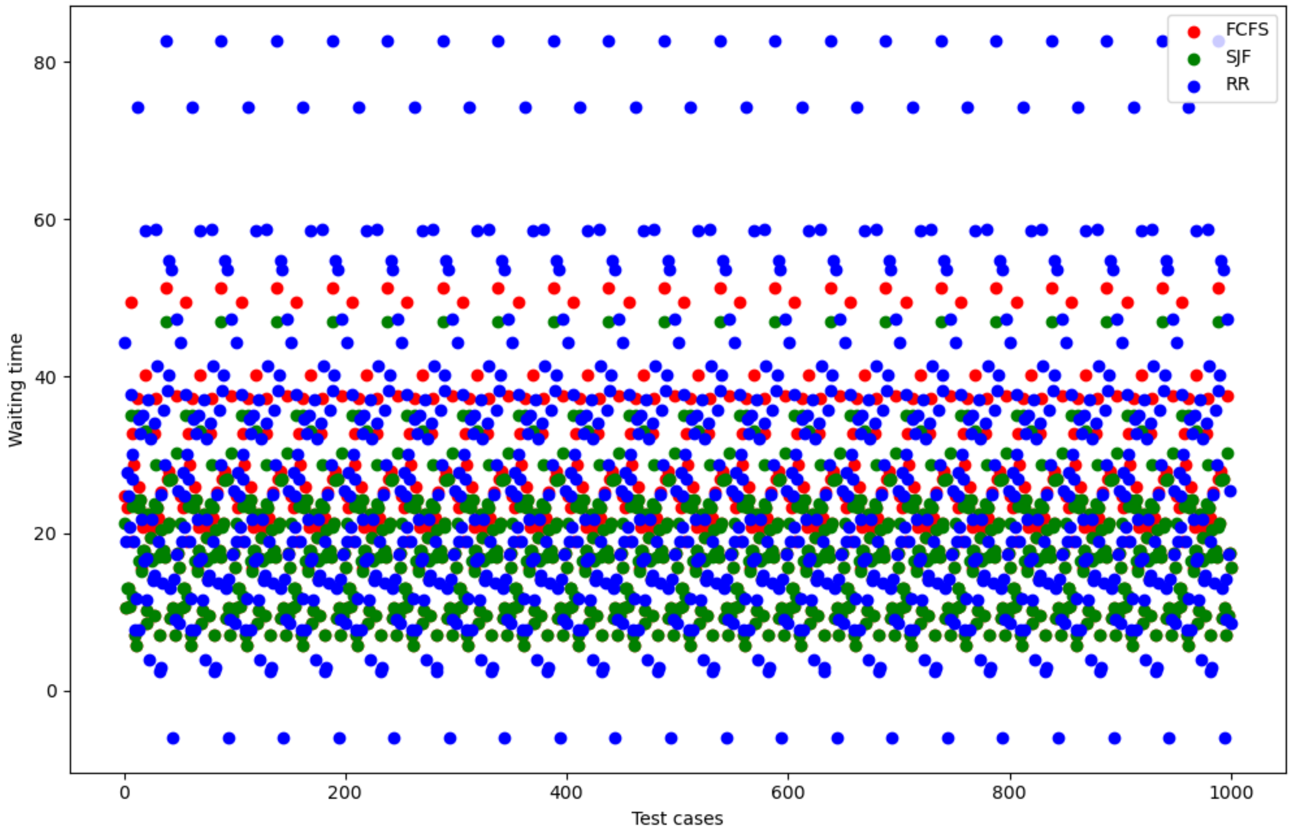

Figure 3: Waiting time

From figure 1,2 and 3 it clearly depicts that for differnt attributes different algorithm is better like if we want less wait time then RR seem to be better but has varied time so SJF can also be preffered whereas if we take turn around time as our constraint the SJF and FCFS are preferable choice and at last if we take completion time as our constraint then FCFS and SJF are the best option.