#include<bits/stdc++.h>

using **namespace** std;

#define INF **int**(1e9)

using u\_map=unordered\_map<**int**,**int**>;

**struct** Process{

**int** process\_num,arrival\_time,burst\_time;

};

**struct** Process\_idx{

Process pr;

**int** idx;

};

**double** calc\_avg(**int** n, u\_map**&** m){

**double** avg=0;

for(**auto** i:m)

avg+=i.second;

avg/=n;

return avg;

}

*// First Come First Serve Scheduling Algorithm*

**void** fcfs(**int** n\_process, vector<Process>**&** p, u\_map**&** ct, u\_map**&** tat, u\_map**&** wt, **double&** avg\_wt, **double&** avg\_tat){

sort(p.begin(), p.end(), [](Process a,Process b){

if(a.arrival\_time==b.arrival\_time)

return a.process\_num<b.process\_num;

return a.arrival\_time<b.arrival\_time;

});

**int** tim=0;

for(**auto** i:p){

**int** p\_num=i.process\_num;

**int** at=i.arrival\_time;

**int** bt=i.burst\_time;

tim+=bt;

ct[p\_num]=tim;

tat[p\_num]=tim-at;

wt[p\_num]=tat[p\_num]-bt;

}

avg\_wt=calc\_avg(n\_process,wt);

avg\_tat=calc\_avg(n\_process,tat);

}

*// Shortest Job First Scheduling Algorithm*

**void** sjf(**int** n\_process, vector<Process>**&** p, u\_map**&** ct, u\_map**&** tat, u\_map**&** wt, **double&** avg\_wt, **double&** avg\_tat){

sort(p.begin(), p.end(), [](Process a,Process b){

if(a.arrival\_time==b.arrival\_time)

return a.burst\_time!=b.burst\_time?a.burst\_time<b.burst\_time:a.process\_num<b.process\_num;

return a.arrival\_time<b.arrival\_time;

});

unordered\_map<**int**,**bool**> finished;

**int** process\_completed = 0;

**int** tim=0;

while(process\_completed<n\_process){

**int** idx=-1;

**int** min\_burst\_time=INF;

for(**int** i=0;i<n\_process;i++){

**int** p\_num=p[i].process\_num;

**int** bt=p[i].burst\_time;

**int** at=p[i].arrival\_time;

if(finished[p\_num]||at>tim)

continue;

if(bt<min\_burst\_time){

idx=i;

min\_burst\_time=bt;

}

else if(bt==min\_burst\_time&&at<p[idx].arrival\_time){

idx=i;

min\_burst\_time=bt;

}

}

if(idx!=-1){

**int** p\_num=p[idx].process\_num;

**int** bt=p[idx].burst\_time;

**int** at=p[idx].arrival\_time;

tim+=bt;

ct[p\_num]=tim;

tat[p\_num]=tim-at;

wt[p\_num]=tat[p\_num]-bt;

finished[p\_num]=true;

process\_completed++;

}

else

tim++;

}

avg\_wt=calc\_avg(n\_process,wt);

avg\_tat=calc\_avg(n\_process,tat);

}

*// Shortest Remaining Time First Scheduling Algorithm*

**void** srtf(**int** n\_process, vector<Process>**&** p, u\_map**&** ct, u\_map**&** tat, u\_map**&** wt, **double&** avg\_wt, **double&** avg\_tat){

sort(p.begin(), p.end(), [](Process a,Process b){

if(a.arrival\_time==b.arrival\_time)

return a.burst\_time!=b.burst\_time?a.burst\_time<b.burst\_time:a.process\_num<b.process\_num;

return a.arrival\_time<b.arrival\_time;

});

vector<Process>p\_temp=p;

unordered\_map<**int**,**bool**>finished;

**int** process\_completed=0;

**int** tim=0;

while(process\_completed<n\_process){

**int** idx=-1;

**int** min\_burst\_time=INF;

for(**int** i=0;i<n\_process;i++){

**int** p\_num=p\_temp[i].process\_num;

**int** bt=p\_temp[i].burst\_time;

**int** at=p\_temp[i].arrival\_time;

if(finished[p\_num]||at>tim)

continue;

if(bt<min\_burst\_time){

idx=i;

min\_burst\_time=bt;

}

else if(bt==min\_burst\_time&&at<p\_temp[idx].arrival\_time){

idx=i;

min\_burst\_time=bt;

}

}

if(idx!=-1){

p\_temp[idx].burst\_time--;

tim++;

if(p\_temp[idx].burst\_time==0){

**int** p\_num=p\_temp[idx].process\_num;

ct[p\_num]=tim;

tat[p\_num]=tim-p\_temp[idx].arrival\_time;

wt[p\_num]=tat[p\_num]-p[idx].burst\_time;

process\_completed++;

finished[p\_num]=true;

}

}

else

tim++;

}

avg\_wt=calc\_avg(n\_process,wt);

avg\_tat=calc\_avg(n\_process,tat);

}

*// Round Robin Scheduling Algorithm*

**void** rr(**int** n\_process, vector<Process>**&** p, **int** tq, u\_map**&** ct, u\_map**&** tat, u\_map**&** wt, **double&** avg\_wt, **double&** avg\_tat){

sort(p.begin(), p.end(), [](Process a,Process b){

if(a.arrival\_time==b.arrival\_time)

return a.process\_num<b.process\_num;

return a.arrival\_time<b.arrival\_time;

});

vector<Process>p\_temp=p;

unordered\_map<**int**,**bool**>added;

queue<Process\_idx>ready\_queue;

ready\_queue.push({p[0],0});

**int** p\_completed=0;

**int** temp=0;

**int** ctr=tq;

**int** tim=p[0].arrival\_time;

while(p\_completed<n\_process){

Process\_idx pr\_idx=ready\_queue.front();

Process pr=pr\_idx.pr;

**int** idx=pr\_idx.idx;

ready\_queue.pop();

added[pr.process\_num]=true;

while(ctr>0&&pr.burst\_time>0){

ctr--;

pr.burst\_time--;

tim++;

for(**int** i=temp+1;i<n\_process;i++,temp++){

if(p[i].arrival\_time>tim)

break;

ready\_queue.push({p[i],i});

}

}

p\_temp[idx].burst\_time=pr.burst\_time;

if(pr.burst\_time==0){

ct[pr.process\_num]=tim;

tat[pr.process\_num]=tim-pr.arrival\_time;

wt[pr.process\_num]=tat[pr.process\_num]-p[idx].burst\_time;

p\_completed++;

}

else

ready\_queue.push({p\_temp[idx],idx});

ctr=tq;

if(ready\_queue.empty()){

for(**int** i=temp+1;i<n\_process;i++,temp++){

ready\_queue.push({p[i],i});

tim=p[i].arrival\_time;

break;

}

}

}

avg\_wt=calc\_avg(n\_process,wt);

avg\_tat=calc\_avg(n\_process,tat);

}

**int** main(){

**int** n\_process;

cin>>n\_process;

vector<Process>p;

for(**int** i=0;i<n\_process;i++){

**int** process\_num,arrival\_time,burst\_time;

cin>>process\_num>>arrival\_time>>burst\_time;

p.push\_back({process\_num,arrival\_time,burst\_time});

}

**int** tq;

cin>>tq;

u\_map ct\_fcfs,tat\_fcfs,wt\_fcfs;

u\_map ct\_sjf,tat\_sjf,wt\_sjf;

u\_map ct\_srtf,tat\_srtf,wt\_srtf;

u\_map ct\_rr,tat\_rr,wt\_rr;

**double** avg\_wt\_fcfs,avg\_tat\_fcfs;

**double** avg\_wt\_sjf,avg\_tat\_sjf;

**double** avg\_wt\_srtf,avg\_tat\_srtf;

**double** avg\_wt\_rr,avg\_tat\_rr;

fcfs(n\_process,p,ct\_fcfs,tat\_fcfs,wt\_fcfs,avg\_wt\_fcfs,avg\_tat\_fcfs);

sjf(n\_process,p,ct\_sjf,tat\_sjf,wt\_sjf,avg\_wt\_sjf,avg\_tat\_sjf);

srtf(n\_process,p,ct\_srtf,tat\_srtf,wt\_srtf,avg\_wt\_srtf,avg\_tat\_srtf);

rr(n\_process,p,tq,ct\_rr,tat\_rr,wt\_rr,avg\_wt\_rr,avg\_tat\_rr);

for(**int** i=0;i<n\_process;i++){

**int** p\_num=p[i].process\_num;

cout<<p\_num<<" -> \t"<<tat\_fcfs[p\_num]<<"\t"<<wt\_fcfs[p\_num]<<"\t"<<tat\_sjf[p\_num]<<"\t"<<wt\_sjf[p\_num]<<"\t"<<tat\_srtf[p\_num]<<"\t"<<wt\_srtf[p\_num]<<"\t"<<tat\_rr[p\_num]<<"\t"<<wt\_rr[p\_num]<<"\n";

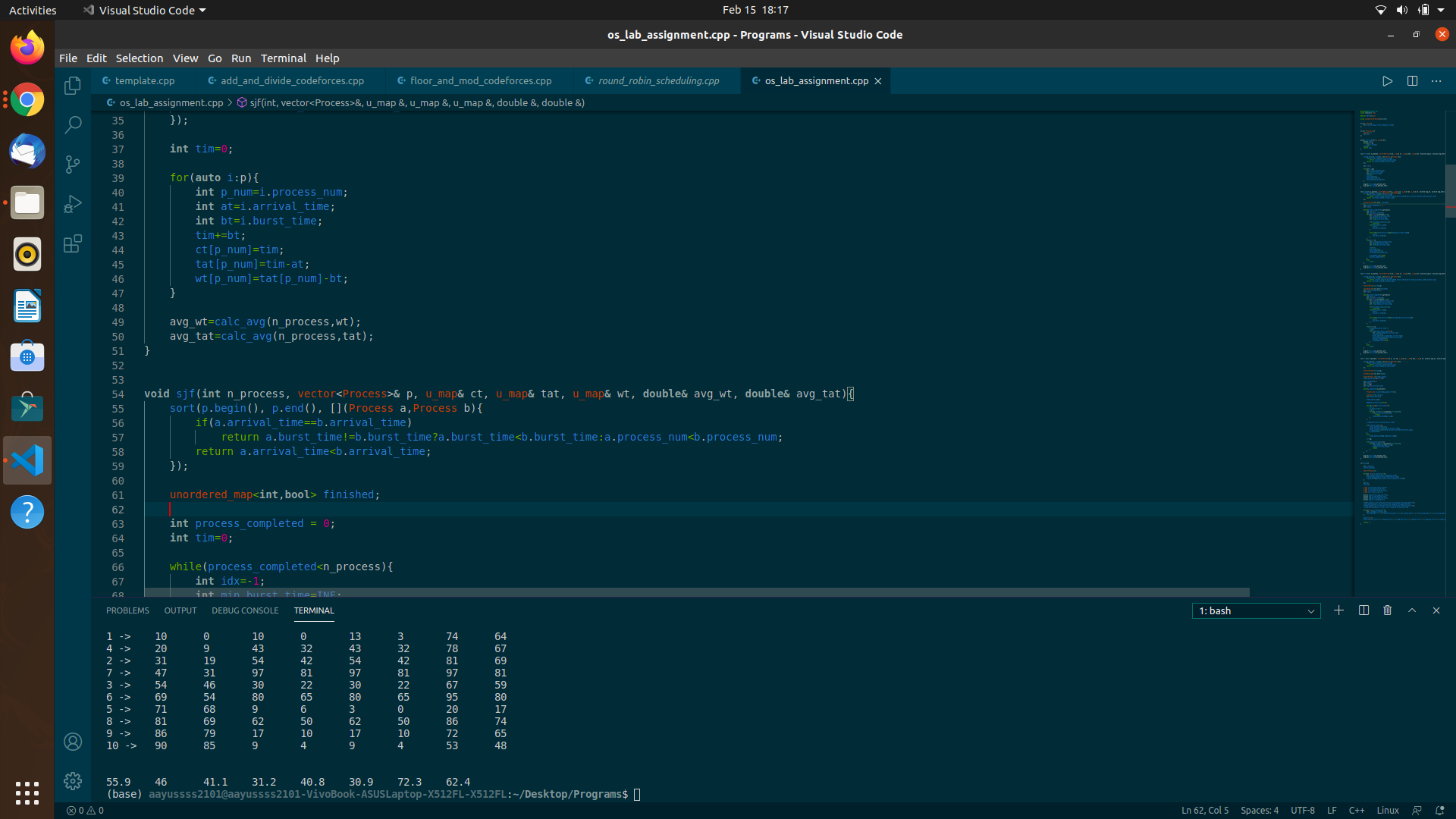
}

cout<<"\n\n";

cout<<avg\_tat\_fcfs<<"\t"<<avg\_wt\_fcfs<<"\t"<<avg\_tat\_sjf<<"\t"<<avg\_wt\_sjf<<"\t"<<avg\_tat\_srtf<<"\t"<<avg\_wt\_srtf<<"\t"<<avg\_tat\_rr<<"\t"<<avg\_wt\_rr<<"\n";

return 0;

}



In my opinion SRTF is the best algorithm for this problem as it has the least average turnaround time and average waiting time.