Amirkabir University of Technology
(Tehran Polytechnic)
Department of Industrial Engineering & Management Systems

Final Project

Simulation of a Manufacturing Workshop

By:
Pedram Peiro Asfia - 9825006
Mahdi Mohammadi - 9825041

Professor:
Dr. Abbas Ahmadi

Course:
Principles of Simulation

June 2022

Abstract
A *simulation* is the imitation of the operation of a real-world process or system over time it involves the generation of an artificial history of a system and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system. Potential changes to the system can first be simulated, to predict their impact on system performance. Simulation can also be used to study systems in the design stage before such systems are built.

In this project a manufacturing workshop is simulated which is a discrete system, thus the discrete-event simulation techniques are used for simulating it.

Keywords:

Discrete-event system simulation, Manufacturing workshop simulation

Contents

Problem Statement

A workshop repairs all kinds of small machines. The workshop consists of five workstations and the flow of orders inside the workshop is as shown below.

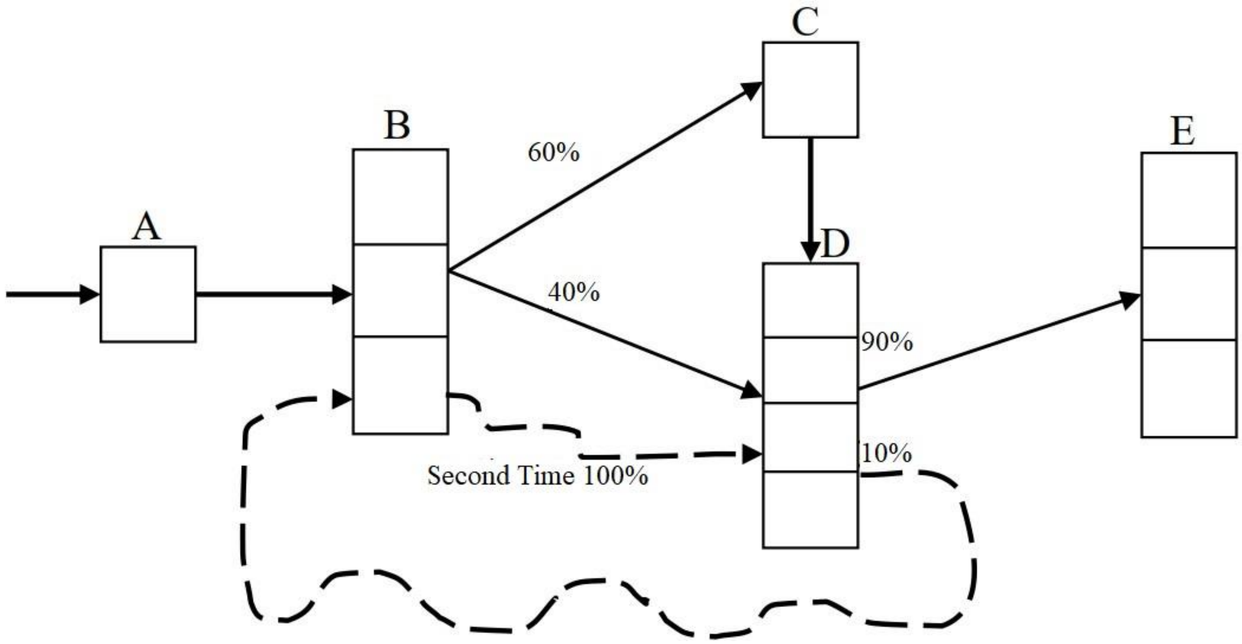


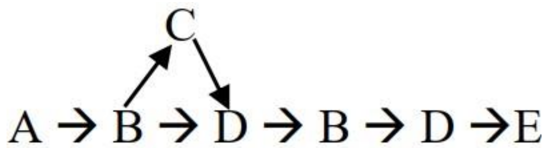
Figure 1-Problem

- Standard orders arrive at station A with the rate of 16 ± 12 minutes per order.
- Urgent orders arrive at the system with the rate of 5 ± 2 hours per order, these orders are placed on the conveyor belt along with all other orders, and cleaning and degreasing operations are performed on them, except at station C, they have a higher priority.
- The duration of order processing and repairs at the first arrival of each order to each station is as follows:

Table 1-Problem Data

Station	Number of machines or employees	Processing or repair times (minutes)	Description
A	1	2 ± 13	Receive order
B	3	20 ± 39	Replacement of parts
C	1	20	Degreasing
D	4	30 ± 47	Assembly of parts and adjustment
E	3	4 ± 42	Packing and shipping

- These periods are true for all orders that go through one of two sequences, $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ or $A \rightarrow B \rightarrow D \rightarrow E$. However, about 10% of orders leave from station D to station B to receive more service (which takes 27 ± 8 minutes). Then they are sent to D and finally to E. The path of these orders is as follows:



- The degreasing station C is closed every two hours, one hour after the opening of the station, for daily maintenance and repair work, which takes 2 ± 10 minutes. But these normal maintenance and repair work will not be done until the maintenance of the possible machines at station C is completed.

Variable Description

Variable	Description	Initial Value
serverA	Number of servers in workstation A	• 1 in Part A and C • 2 in Part B
serverB	Number of servers in workstation B	3
serverC	Number of servers in workstation C	1
serverD	Number of servers in workstation D	4
serverE	Number of servers in workstation E	3
arr_ID	The ID of the machine entered the system	0
MTOT	The total number of machines entered between 120 till 600	0
Status[i]	Status of Server i (0=idle, 1=busy)	0
Q[i]	Number of machines waiting in the i^{th} queue	0
Q_A[i]	List - machines waiting in the queue of server A; format: (ID, time, priority)	0
Q_B[i]	List - machines waiting in the queue of server B; format: (ID, time, priority)	0
Q_C[i]	List - machines waiting in the queue of server C; format: (ID, time, priority)	0
Q_D[i]	List - machines waiting in a queue of server D; format: (ID, time, priority)	0
Q_E[i]	List - machines waiting in the queue of server E; format: (ID, time, priority)	0
TotalQ[i]	Total number of machines waited in the i^{th} queue	0
SVR[i]	Service time of server i	0
Tnow	Simulation clock	0
T1	The start point of collecting statistics	120

T2	The endpoint of collecting statistics	600
TWT	Total waiting time of machines	0
NF	Number of failures	0
Max_WT	Maximum waiting Time	0
N[i]	Total number of served machines by server i	0
RespTime[i]	Response time (equals to departure time – arrival time)	0
FEL	List of Tuples – All future events are saved in this list with this format: • Code • Time • Priority (1 for yes, 0 for no) • Rework, whether the machine needs to rework or not (1 for rework, 0 for no need for rework) • The ID of the machine	[(0,0,1,0,1), (0,0,0,0,2), (3,180, -1,0,0)]
FEL_backup	List – All events (containing past and future) are saved in this list.	Empty
Demographic	Dictionary – containing the following info: • ID of the machine • Type of the order, whether it has a priority or not (1 for priority, 0 for non-priority) • Rework, whether the machine needs rework or not (1 for rework, 0 for no need for rework) • Arrival time, machine’s arrival time. the ID is used as a key and other info is used as a value in this dictionary.	Empty
Demographic_list	List – Saves all the demographics in each iteration	Empty
TWT_list	List – Saves TWT variable in each iteration	Empty
N_list	List - Saves N variable in each iteration	Empty

SVR_list	List - Saves SVR variable in each iteration	Empty
TotalQ_list	List - Saves TotalQ variable in each iteration	Empty
max_WT_list	List - Saves max_WT variable in each iteration	Empty
FEL_total	List - Saves FEL_backup variable in each iteration	Empty
MTOT_list	List - Saves MTOT variable in each iteration	Empty
rcounter	A counter for using random numbers	0

Modeling

- State variables: $(Q_A, Q_B, Q_C, Q_D, Q_E, Status(1), Status(2), \ldots, Status(12))$

  Q_i: Information about machines that waiting in the i^th station's queue

  $$Q_i[j] = (ID_{j_{th}\ part\ in\ the\ queue}, Arrival\ Time, priority\ or\ not\ , rework\ or\ not)$$

  $Status(i)$: Status server *i*

  $Status(i) = 0$ means that server *i* is idle.

  $Status(i) = 1$ means that server *i* is either busy or down (for server 4 in station C).

- Events:

  Arrival Event: code 0

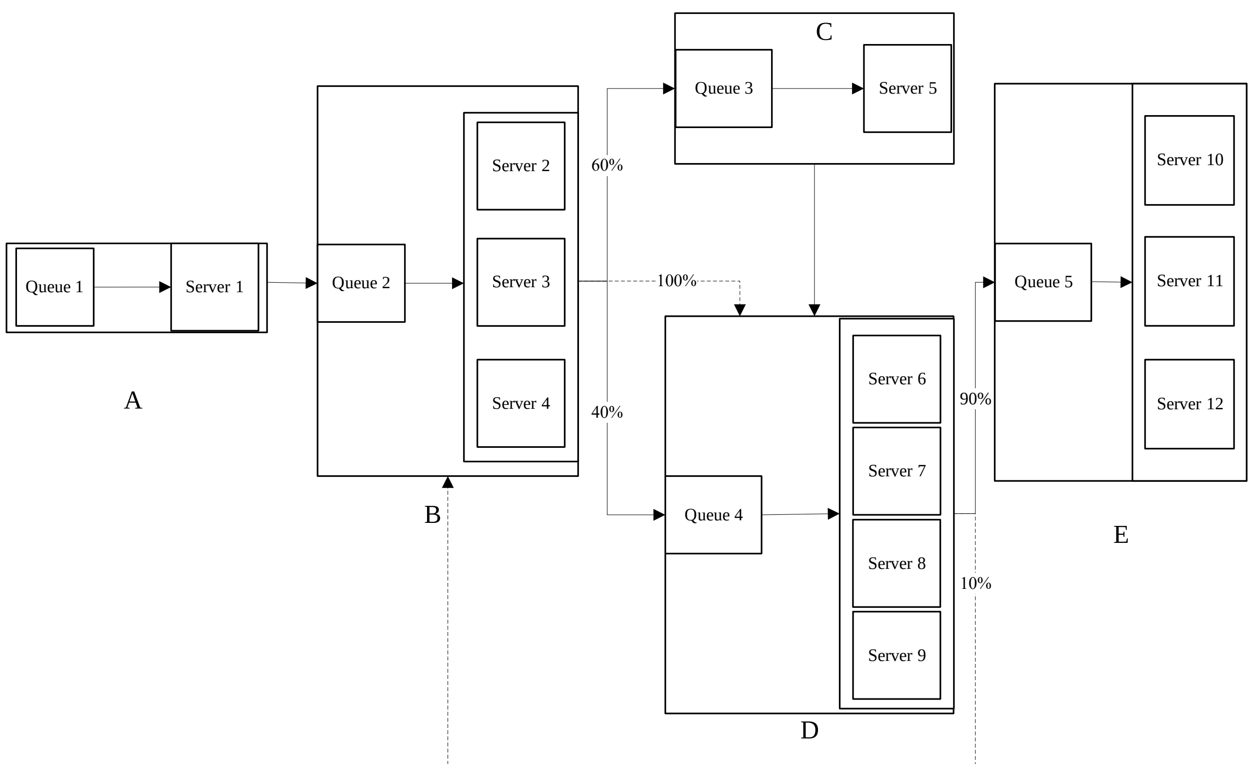
  Departure From A: Code 1

  Departure From B: Code 2,3,4

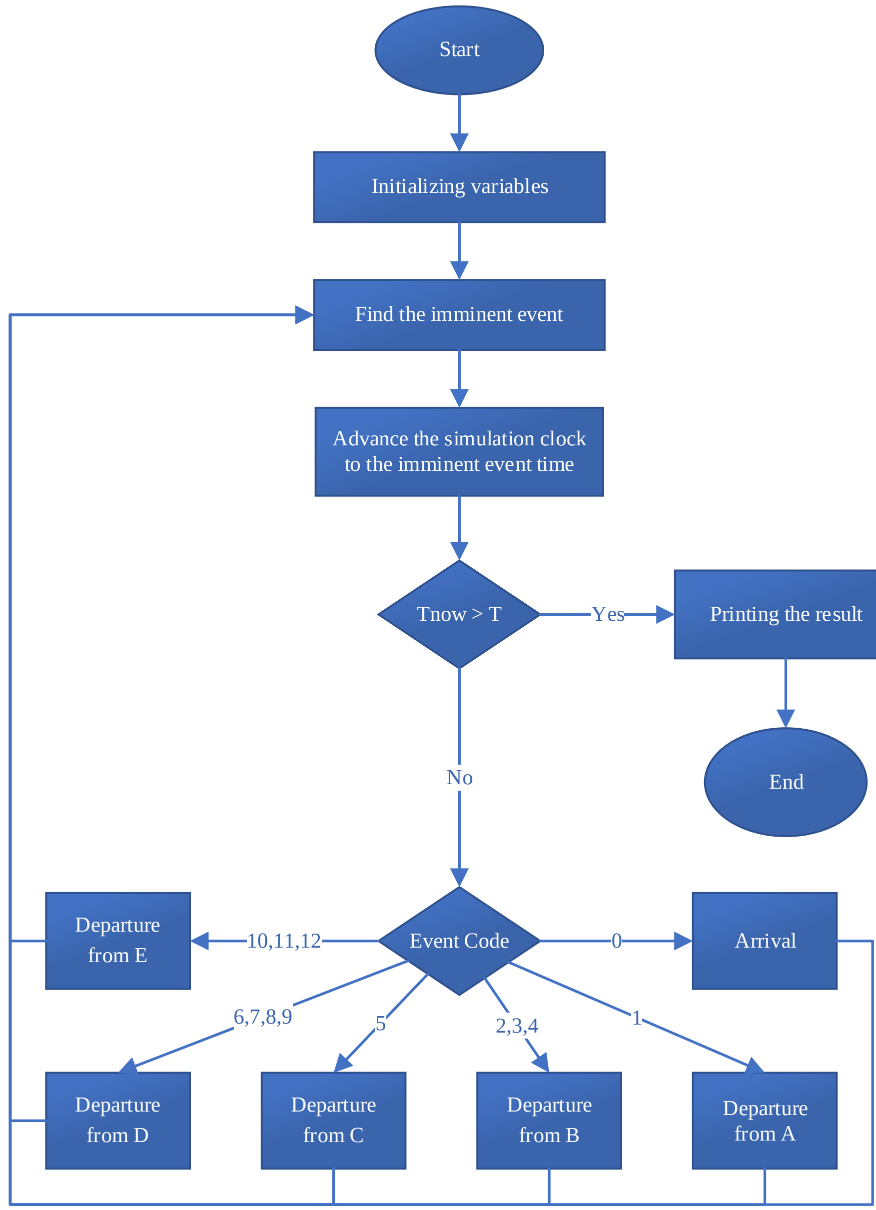
  Departure From C: Code 5

  Departure From D: Code 6,7,8,9

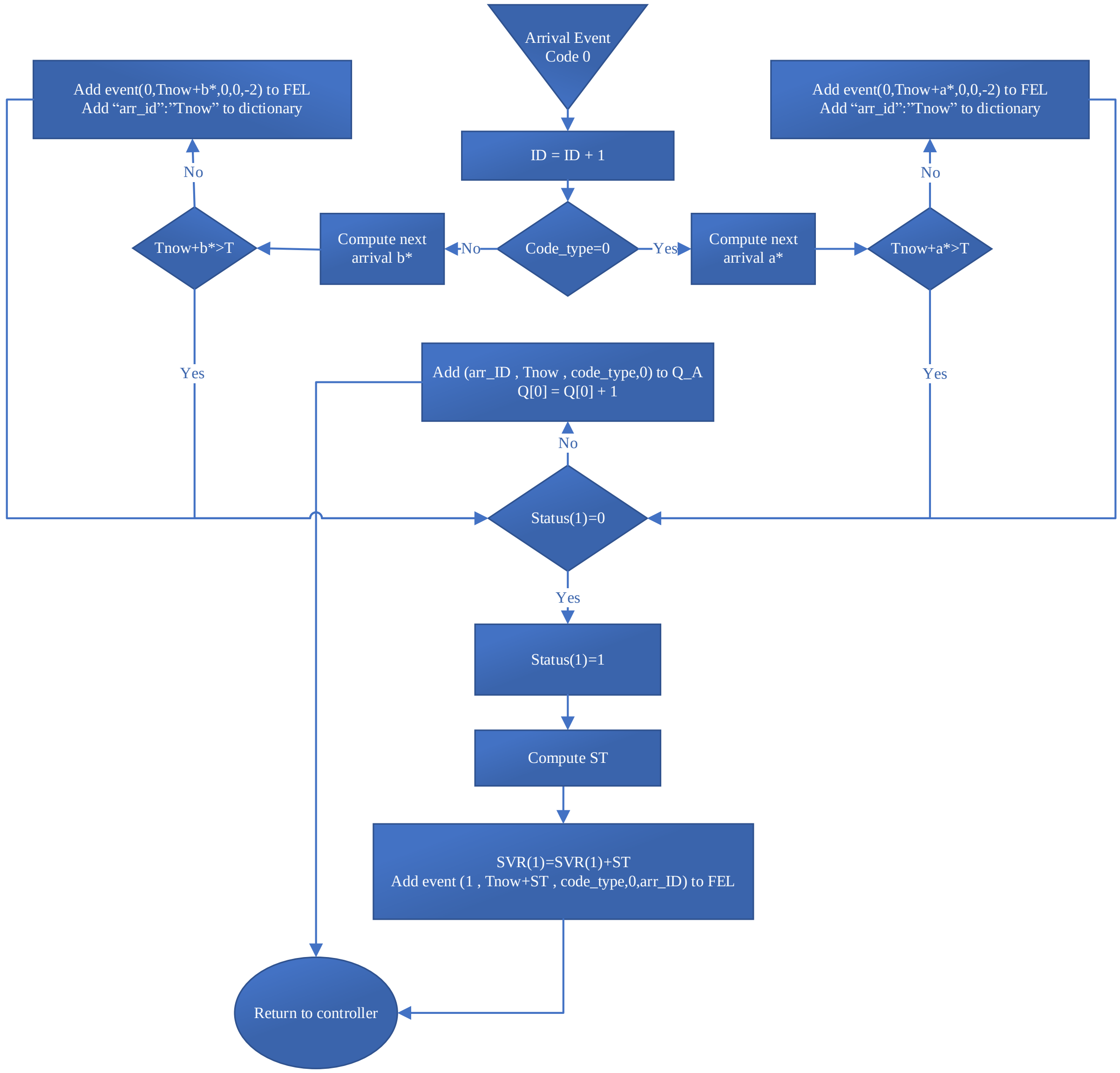
  Departure From E: Code 10,11,12



Controller


Start
Initializing variables
Find the imminent event
Advance the simulation clock to the imminent event time
Tnow > T
Yes
Printing the result
End
No
Event Code
10,11,12
0
6,7,8,9
5
2,3,4
1
Departure from E
Arrival
Departure from D
Departure from C
Departure from B
Departure from A

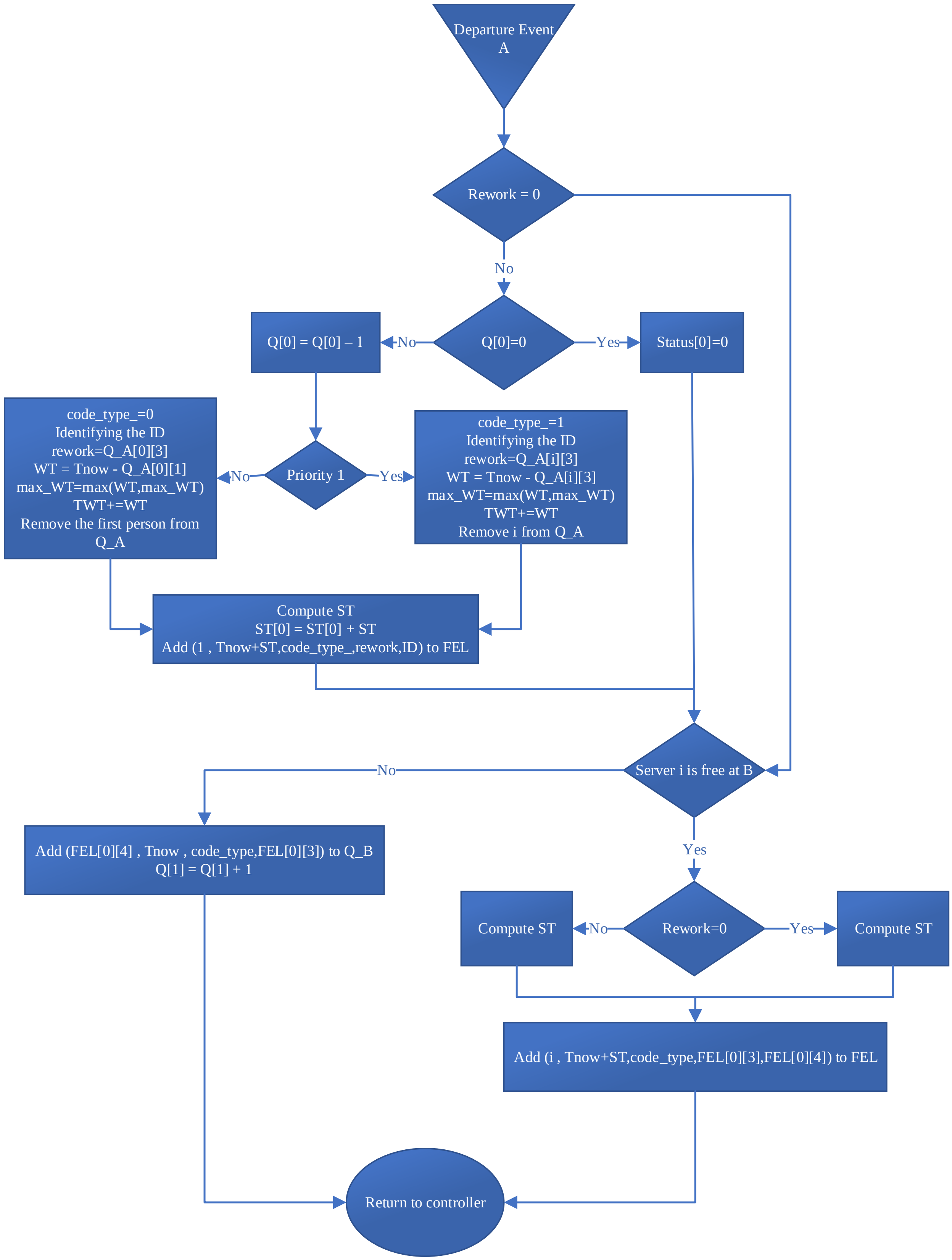
Arrival Event



Since we have two types of orders (standard and urgent) and the distribution of their inter-arrival time differs from each other, we have to compute their inter-arrival time independently (a* and b*). We first check the *code_type*; 0 means that the imminent event is a standard order and due to this, we compute the arrival time of the next standard order, 1 means the imminent event is an urgent order and due to this we compute the arrival time of the next urgent order.

If the server in A is busy, we add the machine to Q_A otherwise, we assign the machine to the server.

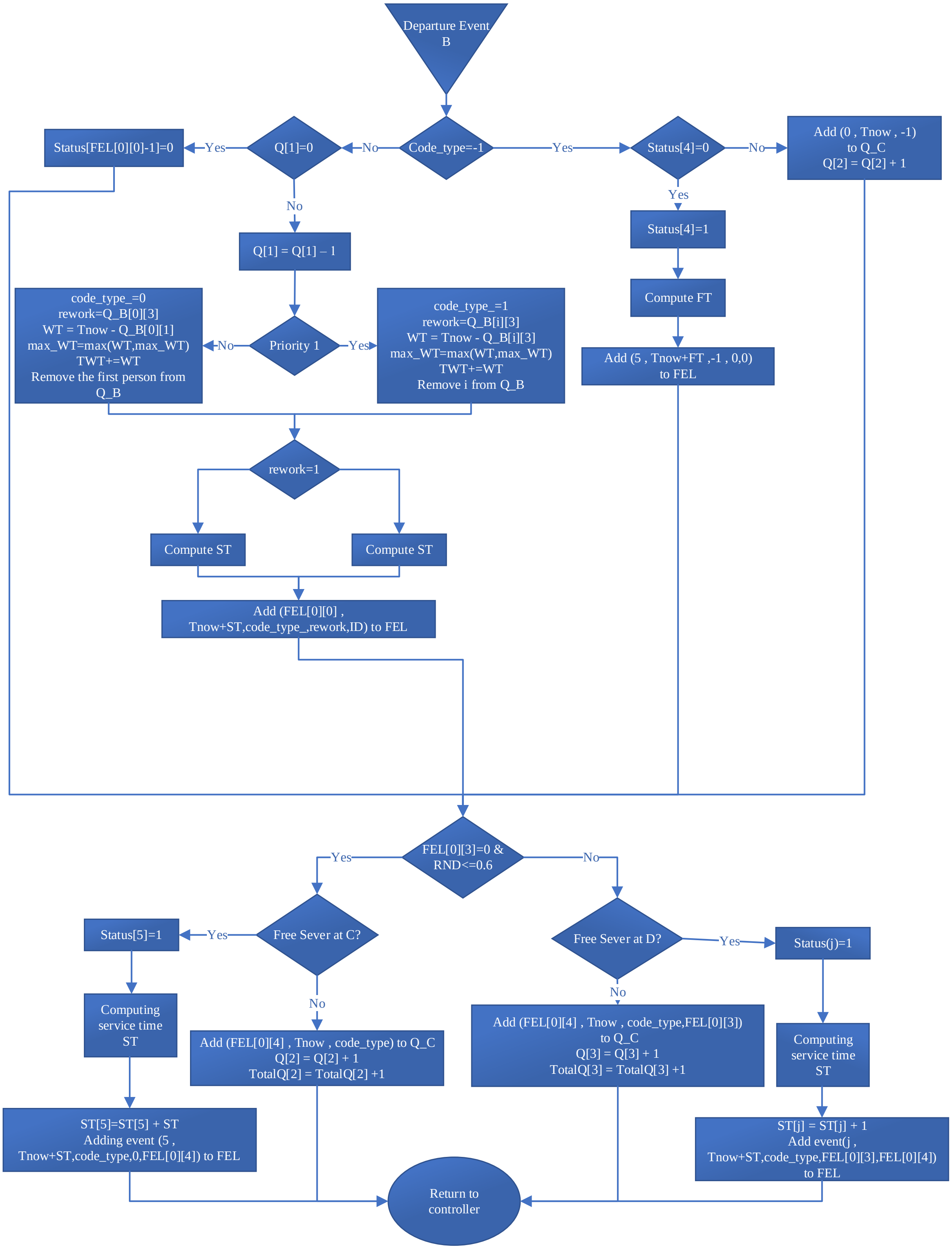
Departure From A



We have two types of *departure from A* event; first, for the machines that were sent to rework from station D for which we should not check Q_A for the next arrival, we just have to check the status of servers in station B and so on.

Second for the machines that have immediately departed from station A for which we should check the Q_A to assign the next machine (due to the priority) to station A's server and after that check the status of servers in station B.

Departure From B

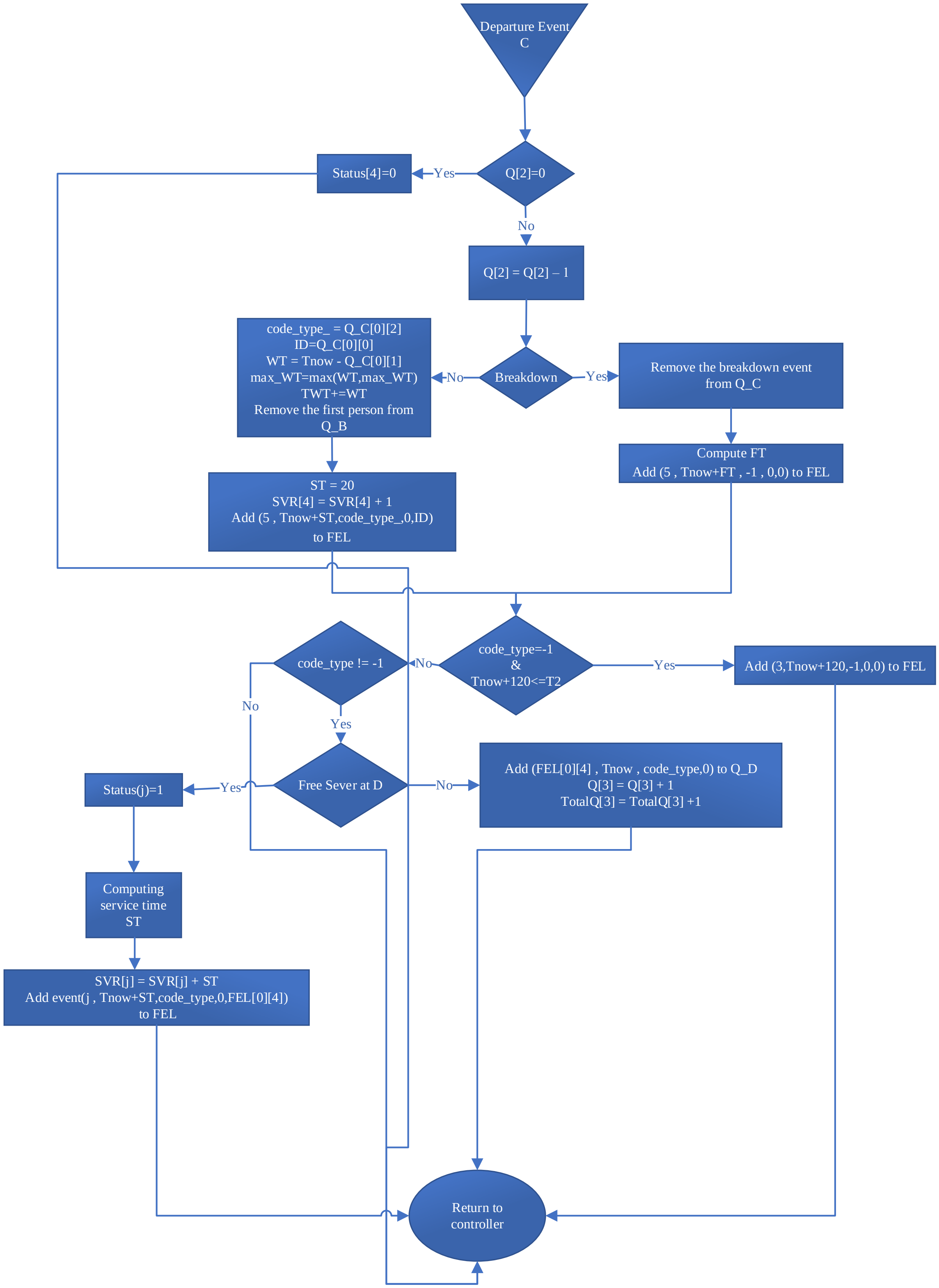


First, we check the code_type; code_type = -1 means it's breakdown time for station C's server (we modeled the breakdown event as a machine whose code_type=-1).

If $codetype = -1$, check the status of station C's server; if busy, we put the breakdown event at the top of Q_C and if it was idle, we change the status to 1.

If $codetype \neq -1$, we first check Q_B and assign the next machine to the server, after that, we create a random digit from which we decide to which station we should send the machine.

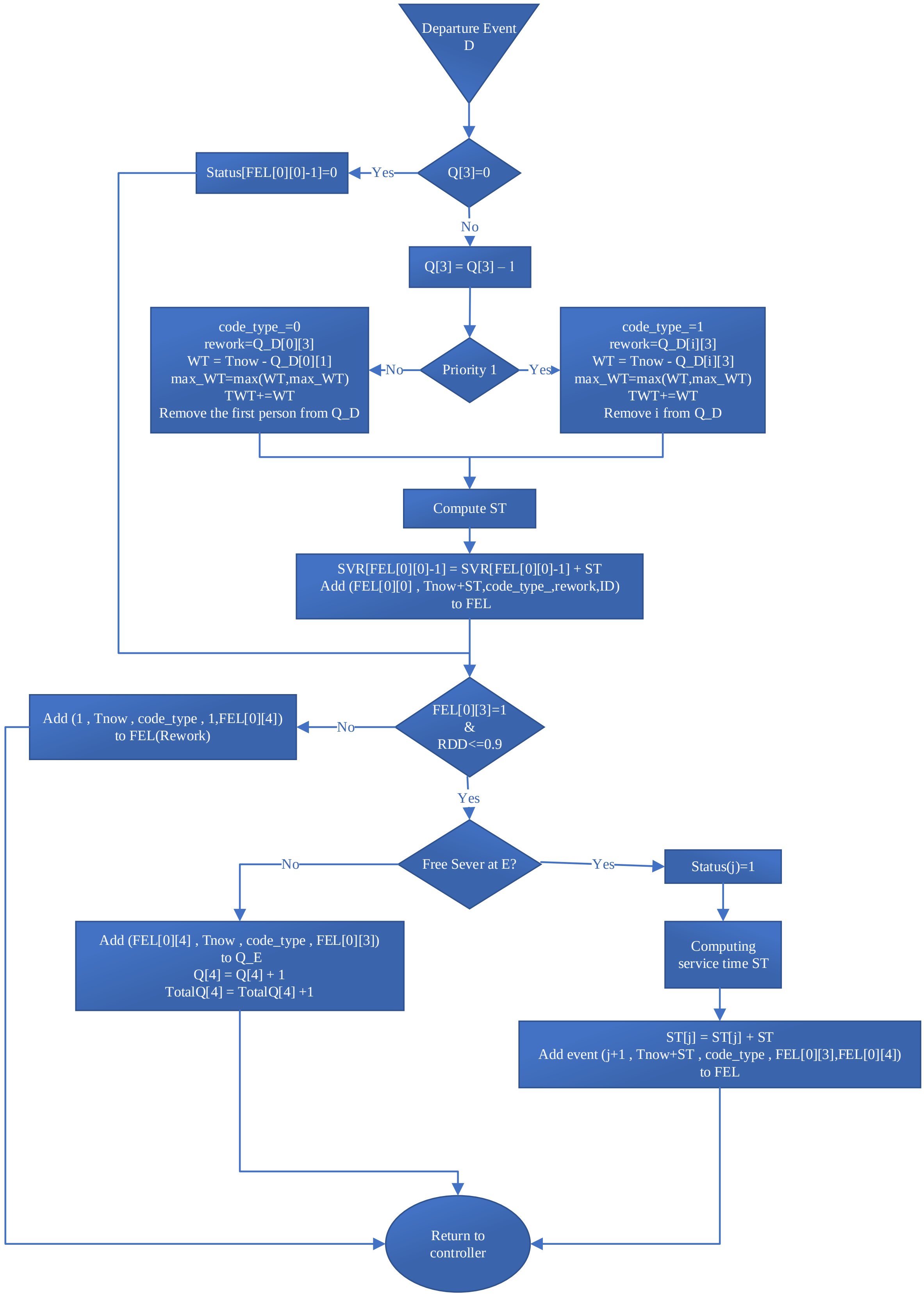
Departure From C



First check Q_C , if empty, we set status [4] =0, otherwise, we should decide whether the item at the top of the list is a breakdown event or a machine that was sent to C.

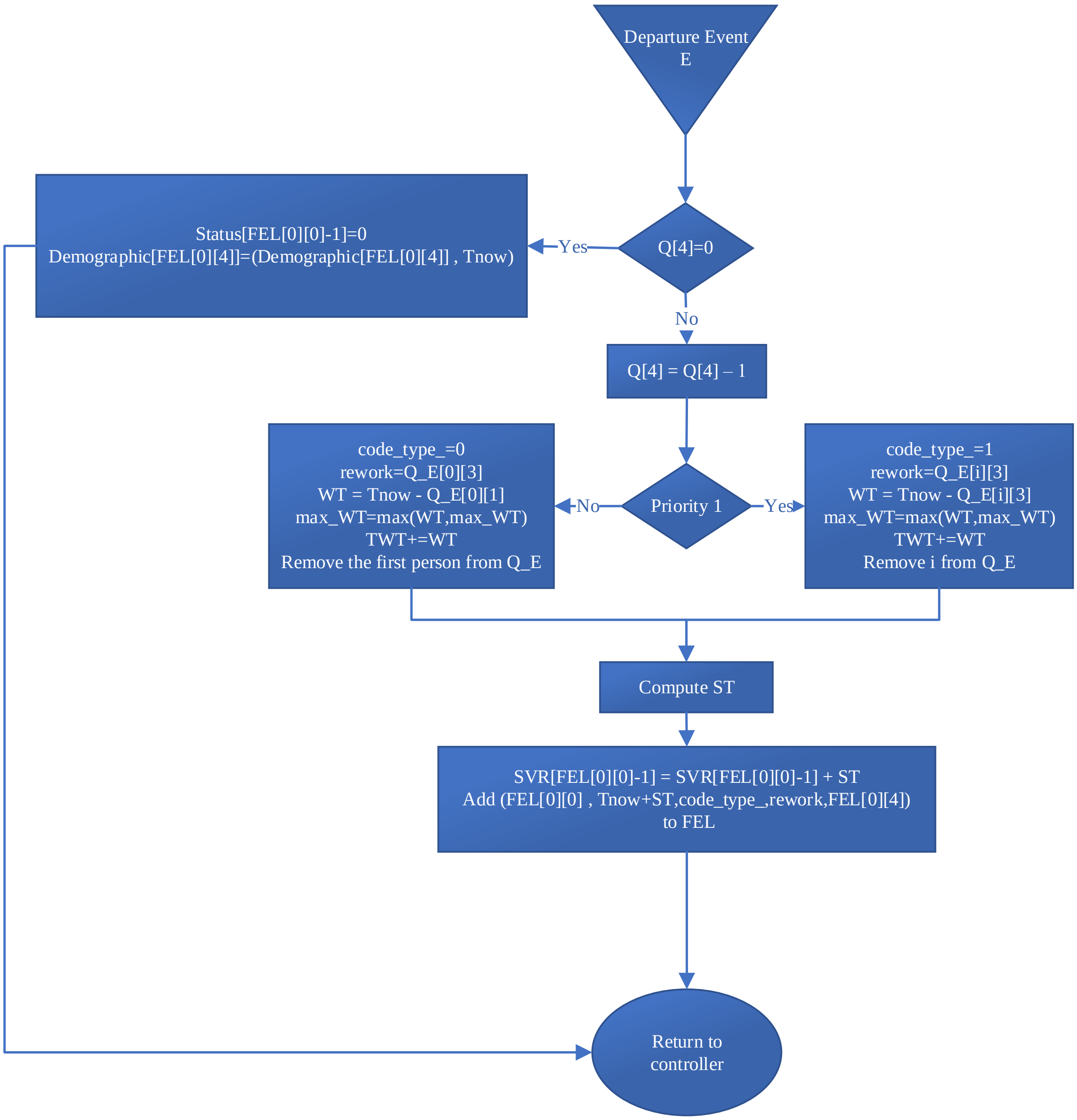
If it was the breakdown event, we compute the fixing time and append it to the FEL otherwise we check the status of station D’s servers and so on.

Departure From D



First, we check the queue and assign the next machine to the sever after that we create a random digit to decide whether to send the machine for a rework (to station B) or to send it to station E.

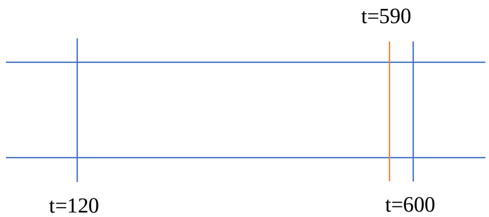
Departure From E



We just have to assign the next machine to the server.

Simplifying Assumptions

- The first standard and urgent orders enter the system at T=0.
- Tasks are assigned to servers by the numerical order of the servers in a station.
- All machines that enter the system will be served.
- The first breakdown of station C occurs at T=180 and repeats every two hours without considering the service time of the station.
- If the server at station C is busy at the breakdown time, we wait till it becomes idle and then break it down.
- After T=600 we do not accept any entrance.
- We compute the statistics for the time between T=120 to T=600, consider this example:



If a new arrival occurs at T=590 and the machine enters a queue, its waiting time is at most 10 minutes even if it is being served at T = 610.

Code Description

First, we discuss some of the important functions and algorithms used in this project, then we'll check out the main code for simulation.

Random Number Generation

To move forward in our simulation project, we need to generate some random numbers and perform some statistical tests on them. As was mentioned in the project, the method of generating these random numbers should be the linear congruential method aka LCM. LCM has 4 parameters, a, the multiplier, X0, the seed, c, the increment, and finally, m, the modulus. To produce a sequence of integers, X1, X2, … between 0 and m-1, we use the following recursive relationship:

$$X_{i+1} = (aX_i + c)\ mod\ m,\ \ i = 0,1,2, \dots$$

The important point is to choose the parameters to achieve the maximal period. For $m = 2^b$, and $c \neq 0$, the longest possible period is $P = 2^b$, which is achieved whenever c is relatively prime to m and $a = 1 + 4k$, where k is an integer. So, we choose the following parameters:

- a=5
- m=2^{21}
- X0=1
- c=3

In this case, $P = 2^{21} = 2097152$. We can see the first 5 numbers and the functions below:

```python
def RNG(a,m,X0,c):
    return (a*X0+c)%m

flag = True
#The first number of the sequnce
randnum = [1]
while(flag):
    r = RNG(5,2**21,randnum[-1],3)
    if(r != 1): #Check whether we entered the Loop or not
        randnum.append(r)
    else:
        flag = False
randnum = np.array(randnum)/(2**21) #Scaling random numbers
randnum = list(randnum)
print(f"{len(randnum)} random numbers have been generated")
print(randnum[:5])
```

executed in 1.06s, finished 3 hours ago

```
2097152 random numbers have been generated
[4.76837158203125e-07, 3.814697265625e-06, 2.0503997802734375e-05, 0.00010395050048828125, 0.0005211830139160156]
```

All of the random numbers are saved in a list called “randnum”.

We know that the sequence of our random numbers should have 2 important characteristics:

1. Uniformity
2. Independency

Uniformity Test Function

We use the Kolmogorov-Smirnov test for checking the uniformity of the given random numbers. The test is as follows:

$$\begin{cases} H_0: R_i \sim U(0,1) \\ H_1: ow \end{cases}$$

First of all, we should sort the given random numbers, then separate [0,1] interval into N (length of the random numbers) equal intervals (or N+1 cut points). Then we should find the distance of the i^{th} random number (in the sorted given list) from the starting cut point and the ending cut point of the i^{th} interval. These distances are saved in lists namely D_plus for “ending cut point – random number”, and D_minus for “random number – starting cut point”.

The test statistic of the KS test is the maximum distance in D_minus and D_plus. The critical value for $\alpha = 5\%$ and $n \geq 35$ is $\frac{1.36}{\sqrt{N}}$. If the test statistic is greater than the critical value, then we should reject the null hypothesis under a significant level of 5%, otherwise, there is not enough evidence to reject the null hypothesis, which means that random numbers are uniformly distributed.

```
def uniformity_test(sample_random):
    N=len(sample_random)
    sorted_rnd = sorted(sample_random)
    D_plus=[]
    D_minus=[]
    for i in range(N):
        D_plus.append((i+1)/N-sorted_rnd[i])
        D_minus.append(sorted_rnd[i]-i/N)
    D=max(max(D_plus) , max(D_minus))
    critical_value = 1.36/N**0.5
    print('D is {}\ncritical value is {}'.format(D,critical_value))
    if D>critical_value:
        print('Reject Null Hypothesis')
    else: print('Not Enough Evidence to Reject The Null Hypothesis')
```

Independency Test Function

We use the Autocorrelation test for checking the independency of the given random numbers. The test is as follows:

$$\begin{cases} H_0: \rho_{im} = 0 \\ H_1: \rho_{im} \neq 0 \end{cases}$$

This statistical test, tests the autocorrelation between every m number (aka lag), starting with the i^{th} number. Also, there is a parameter M, which is the largest integer such that:

$$i + (M+1)m \leq N$$

In which N is the length of the given list. To test the autocorrelation between all of the numbers in a given list of random numbers, the parameters should be:

- i=1
- m=1
- M=N-2

Test statistics are:

$$T = \frac{\hat{\rho}_{im}}{\hat{\sigma}_{\hat{\rho}_{im}}}$$

$$\hat{\rho}_{im} = \frac{1}{M+1}\left[\sum_{k=0}^{M} R_{(i+km)} R_{i+(k+1)m}\right] - 0.25$$

$$\hat{\sigma}_{\hat{\rho}_{im}} = \frac{\sqrt{13M+7}}{12(M+1)}$$

We use the p-value as a measurement to reject the null hypothesis or not. The test I performed was under a 5% of significant level.

```python
def independency_test(sample_random):
    N=len(sample_random)
    M=N-2

    rho_hat=np.sum([(sample_random[i]*sample_random[i+1]) for i in range(N-1)])/(M+1)-0.25
    sigma_hat=((13*M+7)**0.5)/(12*(M+1))
    Z=rho_hat/sigma_hat
    print('rho_hat = {}\nsigma_hat = {}\nZ = {}'.format(rho_hat,sigma_hat,Z))

    pval=st.norm.sf(abs(Z))*2
    print('p-value is {}'.format(pval))
    if pval<=0.05:
        print('Reject Null Hypothesis')
    else: print('Not Enough Evidence to Reject The Null Hypothesis')
```

Note that all of the 2097152 numbers are not going to be used, so we only sample 1000000 and use them in our simulation process, and the random number 0 should be deleted due to computational problems it would cause if selected.

```
sample_random = random.sample(list(randnum),1000000)
if 0 in sample_random: sample_random.remove(0)
print(sample_random[:5])
```

executed in 1.09s, finished 4 hours ago

```
[0.2520432472229004, 0.8717637062072754, 0.7911367416381836, 0.06396341323852539, 0.1673884391784668]
```

Let's perform statistical tests on "sample_random" random numbers and see the results:

```
print('Results for Uniformity Test:')
uniformity_test(sample_random)
print('\n\nResults for Independency Test: ')
independency_test(sample_random)
```

executed in 2.17s, finished 4 hours ago

```
Results for Uniformity Test:
D is 0.0005979267272949174
critical value is 0.00136
Not Enough Evidence to Reject The Null Hypothesis


Results for Independency Test:
rho_hat = -0.0002535209399909255
sigma_hat = 0.00030046268718244505
Z = -0.8437684637925911
p-value is 0.3987988012345528
Not Enough Evidence to Reject The Null Hypothesis
```

Normal Random Variate Generator Function

Also, we need to generate normal variates too in our project, because the service time of some servers follows the normal distribution. Based on direct transformation, to achieve a standard normal random variate, we need 2 random numbers and we should do the calculations below:

$$Z = (-2 \ln R_1)^{0.5} \cos(2\pi R_2)$$

So let $X \sim N(\mu, \sigma^2)$:

$$Z = \frac{X - \mu}{\sigma} \rightarrow X = \mu + \sigma Z$$

```python
def normal_generator(r1,r2,mu,sigma):
    Z=((-2*np.log(r1))**0.5)*np.cos(2*math.pi*r2)
    X=mu+sigma*Z
    return X
```

Priority Checking Algorithm

In the project, we have two types of orders, standard orders, and urgent orders. Urgent orders are those orders that should be given service as soon as possible, which means, if there is a queue for a server, and based on FIFO[1], the first and second machines waiting in the queue are standard orders, but the third one is urgent, the next machine that should be given service is the third machine.

To achieve this and determine which order is urgent and which one is not, a code was defined and used in FEL as its third argument.

To check this algorithm, let's go through an example:

As was earlier mentioned, this algorithm is used whenever there is a queue. So, consider the machines waiting in server A's queue. We have the list of all of these machines in the Q_A list that contains this information respectively:

- The ID of the machine
- Time of the entrance to the queue
- Urgent or not

First, we define a variable ID and set it equal to -1, then we iterate over all of the machines in Q_A to check whether there is a machine with priority or not, to achieve this, we check the 3rd argument of tuples in Q_A, and if it was equal to 1, it means that the machine has priority.

code_type_ variable is related to the priority problem, so it should be 1 if there was an urgent machine, and the ID variable has to be changed into the first argument of our spoken machine.

To reduce bias, statistics are collected only after 120mins, and finally, we should remove the current machine from Q_A (because it has been given service). But what if there was no urgent machine in the queue? In this case, the first machine should be served. This machine has code_type_ of 0 (because it is a standard machine) and then after calculating the statistics, we remove it from the queue

It doesn't matter whether our order is standard or not, service time should be calculated anyway and finally (departure code for server A, future departure time, code_type_ of the chosen machine, its rework status, the ID of the machine) is added to FEL.

Note that "service time" should be a positive number but because of standard deviation, our generated service time may become negative. To solve this problem, we use the absolute value of the generated number.

[1] First In First Out

```
ID=-1
for x in Q_A:  #priority chekcing
    if x[2]==1:
        code_type_=1
        rework=x[3]
        ID=x[0]
        if Tnow>=T1:
            WT= min(Tnow,T2)-max(x[1],T1)
            if WT>0:
                max_WT=max(WT,max_WT)
                TWT+=WT
        Q_A.remove(x)
        break
if ID==-1:    #ordinary chcecking
    code_type_=0
    rework=Q_A[0][3]
    ID=Q_A[0][0]
    if Tnow>=T1:
        WT = min(Tnow,T2) - max(Q_A[0][1],T1)
        if WT>0:
            max_WT=max(WT,max_WT)
            TWT+=WT
    Q_A.pop(0)

ST = abs(normal_generator(sample_random[rcounter] , sample_random[rcounter+1],2,13))
rcounter+=2
if (Tnow>=T1) and (Tnow<=T2): SVR[0]+=min(T2-Tnow,ST)
FEL.append((1 , Tnow+ST,code_type_,rework,ID))         # Future departure event from server 1
```

Arrival Event

The arrival event occurs whenever the code of the event is equal to zero. There are 2 things that matter, first of all, the code type of the current machine that is entering the system (saved in the code_type variable). Second, we should give our machine an ID for further calculations. So arr_ID is increased by one and the FEL of the current machine should change, why? Because the 5th argument is 0 when entering the system (that refers to the ID of the machine). We replace it with arr_ID. Also for calculating the response time of each machine, we created a dictionary namely "Demographic", in which keys are the ID of the machine, and values are (arrival time, departure time). In this section, we only add the arrival time of the machine.

Next, we calculate the interarrival time based on what the code_type is and add it to FEL.

```
# arrival event
if FEL[0][0]==0:
    code_type=FEL[0][2]
    arr_ID+=1

    FEL[0]=(FEL[0][0] , FEL[0][1],FEL[0][2],FEL[0][3],arr_ID)
    Demographic[arr_ID]=Tnow
    if (Tnow>=T1) and (Tnow<=T2): MTOT+=1

    if code_type==0: #ordinary

        #interarrival for ordinary order
        InterArrival=normal_generator(sample_random[rcounter],sample_random[rcounter+1], 16,12)
        rcounter+=2
        if (Tnow+InterArrival <=T2):
            #add arrival event of non-priority order
            FEL.append((0 , Tnow+InterArrival,0,0,-2))

    elif code_type==1:
         #interarrival for priority order
        InterArrival=normal_generator(sample_random[rcounter],sample_random[rcounter+1] , 5,2)
        rcounter+=2
        if (Tnow+InterArrival <=T2):
            #add arrival event of priority order
            FEL.append((0 , Tnow+InterArrival,1,0,-2))
```

Now is the time to see whether the server of workstation A is free or not. If free, it is occupied now with our machine and we should update its utilization and the number of machines it has served (under the condition that Tnow is between 120 and 600). If busy, the ID of the machine, time of the entrance, and whether it needs to rework or not (which is zero for now) is added to the queue of workstation A.

```
if Status[0]==0:

    Status[0]=1
    ST = abs(normal_generator(sample_random[rcounter],sample_random[rcounter+1] , 2,13))
    rcounter+=2

    if (Tnow>=T1) and (Tnow<=T2):
        SVR[0]+=min(T2-Tnow,ST)
        N[0]+=1
    FEL.append((1 , Tnow+ST , code_type,0,FEL[0][4]))

else:
    Q_A.append((FEL[0][4] , Tnow , code_type,0))    # (ID, Time, priority or not , rework or not)
    Q[0]+=1
    if (Tnow>=T1) and (Tnow<=T2): TotalQ[0]+=1
```

Departure Event from Workstation A

First, let's look at the workstation D process quickly. 10% of its machine needs rework and they have to be sent back into workstation B. But the key question is "what parts of the code relate to workstation B?". There are two parts relating to workstation B in our simulation code:

1. Arrival event to workstation B under "Departure Event from workstation A"
2. Determining the status of workstation B when a machine just left it.

After running "Priority Checking Algorithm", it's time for the arrival event to workstation B (for the current machine in the server). After searching for an idle server, we assign the current machine to that server and change its status to busy, and calculate its service time. Of course, it must be noted whether the current machine's rework is 0 or 1, because its service time follows a normal distribution with different parameters (no rework follows $N(20{,}39^2)$, with rework follows $N(27{,}8^2)$). Finally, FEL should be updated with the departure event of the workstation B server.

If no server is idle, the machine will be queued and the statistics will be updated.

```
# arrival event to B (under departure event from A) (for current machine)
for i in range(1,4):
    if Status[i]==0:
        Status[i]=1
        if (Tnow>=T1) and (Tnow<=T2): N[i]+=1
        if FEL[0][3]==1:
            ST=abs(normal_generator(sample_random[rcounter] , sample_random[rcounter+1],27,8))
            rcounter+=2

        else:
            ST=abs(normal_generator(sample_random[rcounter] , sample_random[rcounter+1],20,39))
            rcounter+=2

        if (Tnow>=T1) and (Tnow<=T2): SVR[i]+=min(T2-Tnow,ST)

        FEL.append((i+1 , Tnow+ST,code_type,FEL[0][3],FEL[0][4]))       # Future departure event from ser
        break

    elif i==3:
        Q_B.append((FEL[0][4] , Tnow , code_type,FEL[0][3]))
        Q[1]+=1
        if (Tnow>=T1) and (Tnow<=T2): TotalQ[1]+=1
```

Departure Event from Workstation B

Departure from server B happens whenever the code is equal to 2 or 3 or 4.

Let's take a closer look at FEL's initial value: [(0,0,1,0,1), (0,0,0,0,2), (3,180, -1,0,0)]. The third value is a tuple in which the time is 180. In the problem, it is mentioned that machine C breaks down every two hours after 180 minutes, but first, it waits for the car that is receiving service to finish its service, and then the repair of this machine begins. The initial value in the FEL is because this is the first time the server is going to fail.

Now the question is, why do we discuss this matter when we are departing from workstation B? Because part of the entry into workstation C and the issues of forming the C's queue occurs in the subset of "Departure from Workstation B", we consider the code of "breakdown of server C", the same as the code for departing from server B (i.e. code 3).

However, there should be a way so to differentiate breakdown events from departure events. Therefore, a special code_type was considered for the breakdown of the machine, which is equal to -1.

```
# departure from server B
elif FEL[0][0] in [2,3,4]:
    code_type=FEL[0][2]                                    # for the machine that is leaving server B NOW

    if code_type==-1:                                      # breakdown of machine C event
        if (Tnow>=T1) and (Tnow<=T2): NF+=1
        if Status[4]==0:
            Status[4]=1
            FT=normal_generator(sample_random[rcounter] , sample_random[rcounter+1],10,2)
            rcounter+=2
            FEL.append((5 , Tnow+FT ,-1 , 0,0))
        else:
            Q_C.append((0 , Tnow , -1))
            Q[2]+=1
```

Then, if there was a queue for the workstation B, we run the "Priority Checking Algorithm". Of course, we have to pay attention to the issue of whether the machine needs rework or not because the service time would be different. And then we update the FEL.

It's time for arrival to server C or D (under departure event from server B). Because only 60% of the machines go to server C, a random number is needed to be generated and checked. If its value is less than 0.6, our current machine will be served on this server. However, it should be checked that the machine doesn't need rework (because if it does, it should go to server D, not C).

After this, we repeat the task of checking for an idle server and assign the machine to the server or queue the machine instead.

```
#arrival event to C (under departure event from B)
if (FEL[0][3]==0) and (sample_random[rcounter]<=0.6):    # (send the machine to C if rework==0 and rnd
    rcounter+=1
    if Status[4]==0:
        Status[4]=1
        ST=20
        if (Tnow>=T1) and (Tnow<=T2):
            N[4]+=1
            SVR[4]+=min(T2-Tnow,ST)
        FEL.append((5 , Tnow+ST,code_type,0,FEL[0][4]))

    else:
        Q_C.append((FEL[0][4] , Tnow , code_type))
        Q[2]+=1
        if (Tnow>=T1) and (Tnow<=T2): TotalQ[2]+=1
```

The same thing goes with arrival to server D:

```
#arrival event to D (under departure event from B)
else:
    rcounter+=1
    for i in range(5,9):
        if Status[i]==0:
            Status[i]=1
            if (Tnow>=T1) and (Tnow<=T2): N[i]+=1
            ST=abs(normal_generator(sample_random[rcounter] , sample_random[rcounter+1],30,47))
            rcounter+=2
            if (Tnow>=T1) and (Tnow<=T2): SVR[i]+=min(T2-Tnow,ST)
            FEL.append((i+1 , Tnow+ST,code_type,FEL[0][3],FEL[0][4]))
            break

        elif i==8:
            Q_D.append((FEL[0][4] , Tnow , code_type,FEL[0][3]))
            Q[3]+=1
            if (Tnow>=T1) and (Tnow<=T2): TotalQ[3]+=1
```

Departure Event from Workstation C

Departure from server C happens whenever the code is equal to 5.

An important feature of server C is that, it doesn’t care if our order is urgent or not, it serves them in order. But we know that if there is a breakdown event in its queue, after serving its current machine, the next event that will happen to this server is its breakdown, not the arrival of another machine. So, we have to run the “Priority Checking Algorithm” here as well (however we are searching for code_type=-1, not 1).

```
    if Q[2]==0:                                            # Q[2] is queue of server C
        Status[4]=0                                        # server 5 (from C) is idle now

    else:
        Q[2]-=1
        ID=-2
        for x in Q_C:  #breakdown checking
            if x[2]==-1:
                ID=x[0]
                Q_C.remove(x)
                FT = normal_generator(sample_random[rcounter] , sample_random[rcounter+1],10,2)
                rcounter+=2
                FEL.append((5 , Tnow+FT , -1 , 0,0))
                break
        if ID==-2:
            code_type_=Q_C[0][2]
            ID=Q_C[0][0]
            if (Tnow>=T1):
                WT = min(Tnow,T2) - max(Q_C[0][1],T1)
                if WT>0:
                    max_WT=max(WT,max_WT)
                    TWT+=WT
            Q_C.pop(0)
            ST = 20
            if (Tnow>=T1) and (Tnow<=T2): SVR[4]+=min(T2-Tnow,ST); N[4]+=1
            FEL.append((5 , Tnow+ST,code_type_,0,ID))     # departure event from server i=5
```

Here comes the tricky point, to add the future breakdown event, we have to pay attention to 2 important things:

1. The next breakdown event will happen after 120 minutes, so the event will happen at least at Tnow+120.
2. When should we add this event? We have to put a condition, something like "if there is at least a busy server in the system except server C, this server needs to be fixed after 2 hours, but if all of the servers were idle, there is no need for another breakdown, because all of the machines have already departed from the system."

The conditions are as below:

```
# adding next event of breakdown
if (code_type==-1) and (sum([Status[i] for i in range(12) if i!=4])>0) :
    FEL.append((3,Tnow+120,-1,0,0))
```

If the code_type of the event wasn't -1, it means that we have a machine that is departing from server C right now and is arriving at server D, and we do the repetitive task of searching for an idle server ….

```
#arrival event to D (under departure event from C) (for the current machine leaving C)
elif code_type!=-1:
    for i in range(5,9):
        if Status[i]==0:
            Status[i]=1
            ST=abs(normal_generator(sample_random[rcounter] , sample_random[rcounter+1],30,47))
            rcounter+=2
            if (Tnow>=T1) and (Tnow<=T2):
                N[i]+=1
                SVR[i]+=ST
            FEL.append((i+1 , Tnow+ST,code_type,0,FEL[0][4]))
            break

        elif i==8:
            Q_D.append((FEL[0][4] , Tnow , code_type,0))
            Q[3]+=1
            if (Tnow>=T1) and (Tnow<=T2): TotalQ[3]+=1
```

Departure Event from Workstation D

Departure from server D happens whenever the code is equal to 6 or 7 or 8 or 9.

We run the "Priority Checking Algorithm" and search for an idle server in this part too. Because it has already been explained, we skip this part. Although we have to note that only 90% of the machines will go straight to server E and 10% of them needs to rework. And if the machine has experienced a rework, it will need no further rework and will go straight to server E as well.

To send a machine into rework, it needs to go through the rework process from server B. Arrival event to server B is under the departure event from server A, so the code for this event to be added into FEL should be 1:

```
#Rewrok to B (under departure event from D)
else:
    rcounter+=1
    FEL.append((1 , Tnow , code_type , 1,FEL[0][4]))
```

Departure Event from Workstation E

In this part, we have to update the departure time for the current machine and update our demographic list according to our current machine's ID. Then we run the "Priority Checking algorithm" …:

```
# departure from server E
elif FEL[0][0] in [10,11,12]:
    Demographic[FEL[0][4]]=(Demographic[FEL[0][4]] , Tnow)
    code_type=FEL[0][2]
    if Q[4]==0:                                     # Q[3] is queue of server D
        Status[FEL[0][0]-1]=0                       # server i=10/11/12 (from E) is idle now

    else:
        Q[4]-=1
        if (Tnow>=T1) and (Tnow<=T2): N[FEL[0][0]-1]+=1
        ID=-1
        for x in Q_E:  #priority chekcing
            if x[2]==1:
                code_type_=1
                rework=x[3]
                ID=x[0]
                if (Tnow>=T1):
                    WT = min(Tnow,T2) - max(x[1],T1)
                    if WT>0:
                        max_WT=max(WT,max_WT)
                        TWT+=WT
                Q_E.remove(x)
                break
        if ID==-1:    #ordinary chcecking
            code_type_=0
            rework=Q_E[0][3]
            ID=Q_E[0][0]
            if (Tnow>=T1):
                WT = min(Tnow,T2) - max(Q_E[0][1],T1)
                if WT>0:
                    max_WT=max(WT,max_WT)
                    TWT+=WT
            Q_E.pop(0)
```

Lastly, we update the FEL by adding the next departure event from workstation E.

```
# Future departure event from server i=10/11/12 (E)
FEL.append((FEL[0][0] , Tnow+ST,code_type_,rework,ID))
```

Collecting Statistics

1. Response Time assuming that arrival event to the system>=120 and departure event from the system<=600 for a specific machine

We have 10 different demographics for our 10 times simulation. For calculating response time, we have to iterate through every machine in "demographics" for each iteration and check its arrival time to the system and departure time from the system. If the arrival is greater than 120 and departure is less than 600, (departure-arrival) is considered as the statistic. Finally, we calculate the average of all of these (departure-arrival)s and report it as the statistics.

```
ResponseTimeA1 = [tuple((x[1]-x[0]) for x in Demographic.values() if x[0]>=120 and x[1]<=600)
                  for Demographic in Demographic_list_A]
mean_RTA1 = [np.mean(x) for x in ResponseTimeA1]
```

2. Response Time assuming that arrival event to the system>=120 for a specific machine

We do the same thing here, with the exception that we do not consider the departure time.

```
ResponseTimeA2 = [tuple((x[1]-x[0]) for x in Demographic.values() if x[0]>=120) for Demographic in Demographic_list_A]
mean_RTA2 = [np.mean(x) for x in ResponseTimeA2]
```

3. Response Time at different time intervals

In the problem statement, it is said to calculate response time at different time intervals. To do that, we calculated response time based on arrival at different time intervals. Meaning that we calculate the average response time for those machines that entered the system between two hours in a row (i.e. between 120 and 180).

```
df_RT_A = pd.DataFrame({'From':np.zeros(8) , 'To':np.zeros(8) , 'Mean Response Time':np.zeros(8)})
for lower_hour,upper_hour,i in zip(range(120,600,60) , range(180,660,60),range(0,8)):
    rt = [tuple((x[1]-x[0]) for x in Demographic.values() if x[0]>=lower_hour and x[0]<=upper_hour)
          for Demographic in Demographic_list_A]
    df_RT_A.loc[i]=[lower_hour , upper_hour , np.mean([np.mean(x) for x in rt])]
df_RT_A
```

4. Average waiting time for each machine

we only need to calculate the average total waiting time for those machines that were given service between 120 and 600.

```
print('Mean waiting time of machines waiting in queues is:',
      np.mean(np.array(TWT_list_A)) , '(calculated waiting time only for 120<=T<=600)')
print('95% confidence interval for average waiting time is:' ,
      st.t.interval(alpha=0.95 , df=len(TWT_list_A)-1 , loc=np.mean(TWT_list_A) , scale=st.sem(TWT_list_A)))
```

5. Average service time for each server in workstations

We should calculate (summation of the utilization of all the servers in a station)/(number of the machines they served).

```
mean_ST_A = np.mean(np.array([x[0] for x in SVR_list_A])/np.array([x[0] for x in N_list_A]))
mean_ST_B = np.mean(np.array([(x[1]+x[2]+x[3]) for x in SVR_list_A])/
                    np.array([(x[1]+x[2]+x[3]) for x in N_list_A]))

mean_ST_C = np.mean(np.array([x[4] for x in SVR_list_A])/
                    np.array([x[4] for x in N_list_A]))

mean_ST_D = np.mean(np.array([(x[5]+x[6]+x[7]+x[8]) for x in SVR_list_A])/
                    np.array([(x[5]+x[6]+x[7]+x[8]) for x in N_list_A]))

mean_ST_E = np.mean(np.array([(x[9]+x[10]+x[11]) for x in SVR_list_A])/
                    np.array([(x[9]+x[10]+x[11]) for x in N_list_A]))

mean_ST_total_value=[mean_ST_A,mean_ST_B,mean_ST_C,mean_ST_D,mean_ST_E]
df_ST_A = pd.DataFrame({'Average Service Time': mean_ST_total_value} , index=['A','B','C','D','E'])
df_ST_A
```

6. Average working time for each server in workstations

Equals to the summation of utilization for the whole station and per server:

```
SVR_A = np.mean([x[0] for x in SVR_list_A])
SVR_B = np.mean([(x[1]+x[2]+x[3]) for x in SVR_list_A])
SVR_C = np.mean([x[4] for x in SVR_list_A])
SVR_D = np.mean([(x[5]+x[6]+x[7]+x[8]) for x in SVR_list_A])
SVR_E = np.mean([(x[9]+x[10]+x[11]) for x in SVR_list_A])

mean_SVR_total=[SVR_A,SVR_B,SVR_C,SVR_D,SVR_E]
mean_SVR_server = [SVR_A,SVR_B/3,SVR_C,SVR_D/4,SVR_E/3]
df_SVR_A = pd.DataFrame({'Average working time (for the whole station)': mean_SVR_total ,
                         'Average working time (per server in the station)':mean_SVR_server} ,
                        index=['A','B','C','D','E'])
df_SVR_A
```

7. Average total queue length for each workstation

Average of the queue's length for each workstation in each iteration of the simulation.

```
Queue_A=np.mean([x[0] for x in TotalQ_list_A])
Queue_B=np.mean([x[1] for x in TotalQ_list_A])
Queue_C=np.mean([x[2] for x in TotalQ_list_A])
Queue_D=np.mean([x[3] for x in TotalQ_list_A])
Queue_E=np.mean([x[4] for x in TotalQ_list_A])
Q_total = [Queue_A,Queue_B,Queue_C,Queue_D,Queue_E]

df_Q_A = pd.DataFrame({'Average total queue length': Q_total}, index=['A','B','C','D','E'])
df_Q_A
```

8. Average of maximum waiting time for each iteration in 10 times simulation:

```
print('Average of maximum waiting time for 10 iterations is',np.mean(max_WT_list_A))
```

Results

In this section, we review the results and code outputs. Note that the analysis of the results is also given in this section after the output of each code if needed.

It should also be noted that we first analyze the results of the current simulation system itself in part A, and after that, we compare the results of the current system, the system that has an additional server, and the system whose server C is replaced.

Part A

Response Time

For each statistic, there is a point estimation and confidence interval that was asked to be calculated.

```
Average Response time for the machine that entered the system after 120min and left the system before 600min is 417.0 2
95% confidence interval for average response time is: (404.3545783431939, 429.19245294866323)
```

Figure 4-Average Response time period 120-600 - Part A

```
mean Response time for the machine that entered the system after 120min is 1168.6527419776605
95% confidence interval for average response time is: (1101.4544013702362, 1235.851082585085)
```

Figure 5-Average response time for arrivals after 120 - Part A

The reason behind why we calculated response time under 2 different conditions, is that, for period 120-600, the response time of those machines that need rework aren't taken into account very well. We can trace those machines in FEL easily just by looking at its 4^{th} argument and taking their ID and searching them in the Demographic list.

For example, for the first iteration in simulation, machines that need rework are as follows:

```
rework_needed = set([x[4] for x in FEL_total_A[0] if x[3]==1])
print(rework_needed)
```

executed in 12ms, finished a few seconds ago

```
{133, 134, 11, 13, 147, 152, 26, 157, 33, 37, 44, 51, 54, 56, 69, 73, 76, 83, 85, 94, 103, 109, 110, 124}
```

Figure 6-Machines with rework

Now if we look at their demographic, we can see the arrival and departure times:

```python
keys = sorted(list(rework_needed))
for key in keys:
    print(f'ID={key}:',Demographic_list_A[0].get(key))
```

executed in 13ms, finished 6 hours ago

```
ID=11: (35.77570415900999, 2380.441639962473)
ID=13: (42.092374299254104, 483.9173208743896)
ID=26: (87.93531945677316, 648.879594842458)
ID=33: (125.39032761246773, 601.5911498639205)
ID=37: (145.76726937265525, 758.515189408474)
ID=44: (176.66903097641426, 788.428002281865)
ID=51: (197.87855703384204, 2299.6306122087217)
ID=54: (210.08669573150078, 2322.573886271145)
ID=56: (217.17284936523967, 2362.0617345290093)
ID=69: (265.53514424942, 1039.7967593539624)
ID=73: (277.82581042273574, 1188.8604184474027)
ID=76: (290.82096328349866, 2347.724660584383)
ID=83: (312.24349689908644, 1040.566079819679)
ID=85: (317.8992091700404, 1380.2801957208264)
ID=94: (355.6108401417561, 1572.7157292365155)
ID=103: (387.2540827459234, 1636.42220904463)
ID=109: (421.84184777941937, 2471.8285255930464)
ID=110: (424.3892759749671, 1701.763493226777)
ID=124: (481.9489744952636, 1746.581970284453)
ID=133: (515.2861083046332, 1702.9102270039016)
ID=134: (516.3863609641576, 1816.6386599964865)
ID=147: (557.8438356264904, 2450.4265306934967)
ID=152: (579.5675474260278, 1801.2967653115022)
ID=157: (597.2178253735007, 2428.1610604715815)
```

Figure 7-Demographics of rework-needed-machines

Now with the condition of T_arrival>=120 and T_departure<=600, only the response time of the machine with ID=38 is taken into account which is not a very good imitation of how the system works.

We can see the response time of the machines which entered the system in different time intervals as below:

	From	To	Mean Response Time
0	120.0	180.0	883.122812
1	180.0	240.0	1091.189042
2	240.0	300.0	1070.128864
3	300.0	360.0	1085.326311
4	360.0	420.0	1200.475807
5	420.0	480.0	1280.703528
6	480.0	540.0	1349.557847
7	540.0	600.0	1417.245218

Figure 8-Average response time in different time intervals – Part A

	From	To	95%-CI Response Time
0	120.0	180.0	[810.69, 955.55]
1	180.0	240.0	[963.58, 1218.8]
2	240.0	300.0	[988.89, 1151.36]
3	300.0	360.0	[1042.7, 1127.95]
4	360.0	420.0	[1113.41, 1287.54]
5	420.0	480.0	[1209.52, 1351.88]
6	480.0	540.0	[1265.34, 1433.77]
7	540.0	600.0	[1348.87, 1485.62]

Figure 9-Confidence interval for response time in different time intervals - Part A

Average Waiting Time

```
Mean waiting time of machines waiting in queues is: 37662.41203700386 (calculated waiting time only for 120<=T<=600)
95% confidence interval for average waiting time is: (35533.29878399458, 39791.52529001314)
```

Figure 10-Average waiting time in queues - Part A

Average Service Time

One of the questions that may arise is why the average service time, for example on server A, is 9.87, if it follows a normal distribution with a mean of two. Since this normal random variable has a standard deviation of 13, therefore it can take negative values with a high probability ($P(X < 0) = P\left(\frac{X-2}{\sqrt{13}} < \frac{0-2}{\sqrt{13}}\right) = 28.95\%$). But service time can't be negative, so we use the absolute value of the random value generated by this random variable and the average changes due to this.

	Average Service Time
A	9.875886
B	35.079490
C	19.189698
D	42.541796
E	31.326775

Figure 12-Average service time for each server in each workstation – Part A

	95%-Confidence Interval
A	[9.19, 10.56]
B	[32.0, 38.16]
C	[19.03, 19.35]
D	[38.45, 46.64]
E	[27.66, 35.0]

Figure 11-Confidence interval for Average service time – Part A

Average working Time

It is important to know how much each station and servers in those stations work, so we can determine which station is the busiest and where is the bottleneck for further analysis and improvement in the system.

	Average working time (for the whole station)	Average working time (per server in the station)
A	476.862826	476.862826
B	1327.089665	442.363222
C	380.134782	380.134782
D	1453.954481	363.488620
E	963.733170	321.244390

Figure 13-Average working time for each station and server - Part A

	95%-Confidence Interval for working time (for the whole station)	95%-Confidence Interval for working time (per server in the station)
A	[473.52, 480.2]	[473.52, 480.2]
B	[1266.4, 1387.78]	[422.13, 462.59]
C	[364.25, 396.02]	[364.25, 396.02]
D	[1337.95, 1569.96]	[334.49, 392.49]
E	[856.07, 1071.4]	[285.36, 357.13]

Figure 14-Confidence interval for working time - Part A

So, the busiest station is station A which has only one server.

Average total queue length

Another metric to identify the bottleneck is the total queue length of each station:

	Average total queue length
A	137.4
B	33.2
C	22.8
D	12.2
E	11.4

Figure 15-Average total queue length - Part A

	95%-confidence interval for total queue length
A	[134.08, 140.72]
B	[30.49, 35.91]
C	[18.88, 26.72]
D	[10.78, 13.62]
E	[8.68, 14.12]

Figure 16-Confidence interval for queue length – Part A

We can see that even with this measurement, server A is the busiest.

Maximum Waiting Time

```
Average of maximum waiting time for 10 iterations is 480.0
```

Figure 17-Maximum waiting time - Part A

Part B: Adding Another Server to The System

If we want to add another server to the system, we have to assign this server to a station that is practically a bottleneck, so that the machines receive service sooner and stay in the queue less. However, we have to state that, the waiting time or response time may not change, because the statistics are only gathered between 120 and 600 and this improvement in the system may not be clear in that interval.

Anyway, as it was mentioned earlier, the bottleneck in the system is workstation A. so by adding a server to this station, we have now 2 servers in this station.

Part C: Replacing Server C With a Newer Version

Service time of server C in part C changes from 20 to 15, so we would expect a smoother system with less response time and less waiting time, but still, it may not change because of the period we are collecting statistics.

Comparison Between Part A, B & C:

Response Time

	Average response time A	Average response time B	Difference A & B	Average response time C	Difference A & C
0	416.773516	364.547284	52.226232	380.106971	36.666545

Figure 18-Average response time for period 120-600 - All Parts

	Average response time A	Average response time B	Difference A & B	Average response time C	Difference A & C
0	1168.652742	1150.861465	17.791277	990.772377	177.880365

Figure 19-Average response time for after 120 – All Parts

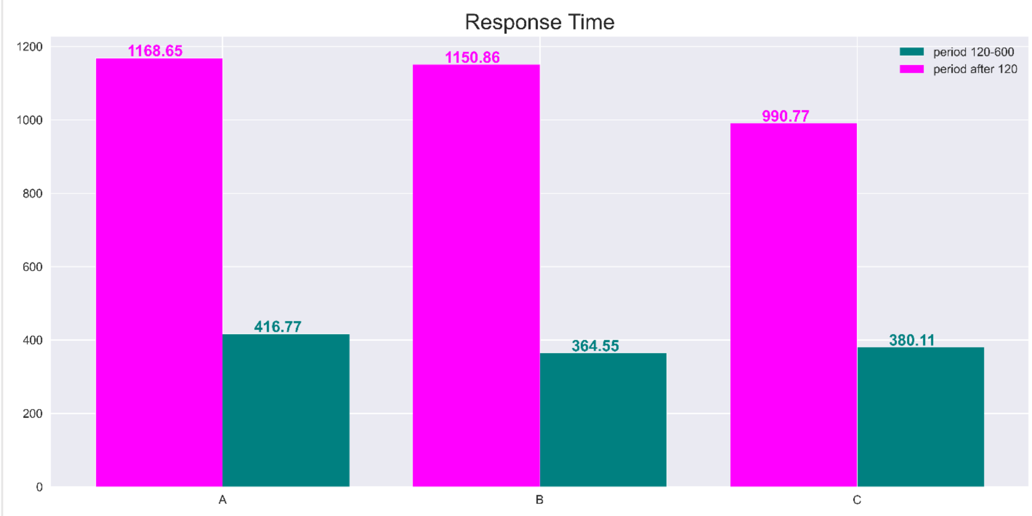


Figure 21- Response time comparison

We can see that by adding an additional server to station A, the overall response time decreases, the same thing is true by reducing the service time of server C. The interesting thing is why adding a server between 120-600 is better while it is better to decrease server C's service time in the period after 120. We think this is because the number of simulations is small and we have to increase the number of simulations to more than 10 times to get more accurate results.

	From	To	Mean Response Time A	Mean Response Time B	Difference A & B	Mean Response Time C	Difference A & C
0	120.0	180.0	883.122812	921.521733	-38.398921	794.831268	88.291544
1	180.0	240.0	1091.189042	867.601442	223.587600	770.997594	320.191448
2	240.0	300.0	1070.128864	1043.113450	27.015414	901.716930	168.411935
3	300.0	360.0	1085.326311	1169.371606	-84.045295	940.098736	145.227575
4	360.0	420.0	1200.475807	1150.931125	49.544682	994.833176	205.642631
5	420.0	480.0	1280.703528	1267.721387	12.982141	1102.327034	178.376494
6	480.0	540.0	1349.557847	1329.085258	20.472589	1155.939970	193.617877
7	540.0	600.0	1417.245218	1437.965409	-20.720191	1233.534793	183.710426

Figure 22-Response time in different time intervals - All Parts

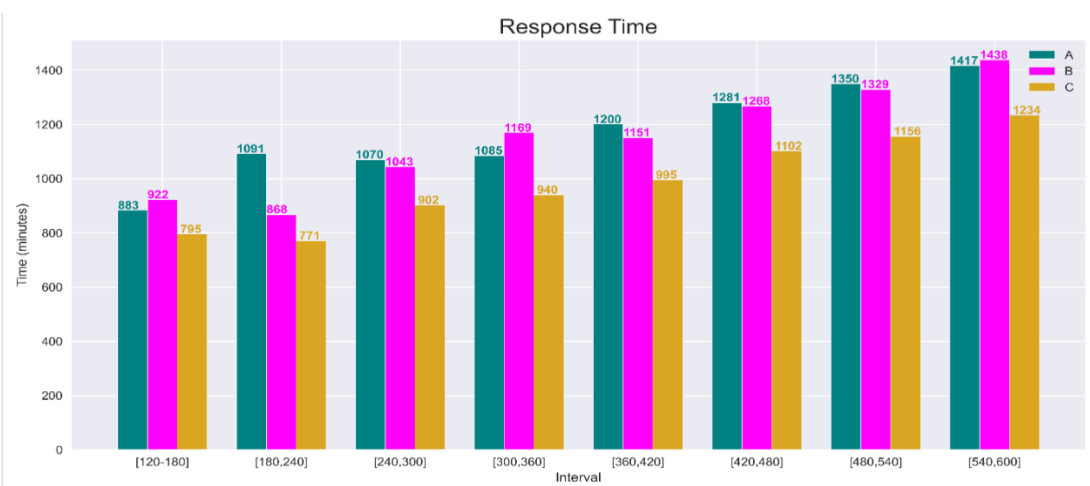


Figure 23-Response time in different time intervals comparison - All Parts

It is obvious that as time goes on, the response time increases. This is because the number of machines that will be in a queue increase and therefore, they have to stay in the queue longer, and therefore the response time increases. But this time is the shortest for the third case, which is about reducing the service time of server C.

Average Waiting Time

We expect that by adding an additional server or reducing the service time of server C, the total waiting time decreases:

	Mean Waiting Time A	Mean Waiting Time B	Difference A & B	Mean Waiting Time C	Difference A & C
0	37662.412037	35862.596177	1799.81586	33379.138769	4283.273268

Figure 24-Average total waiting time - All parts

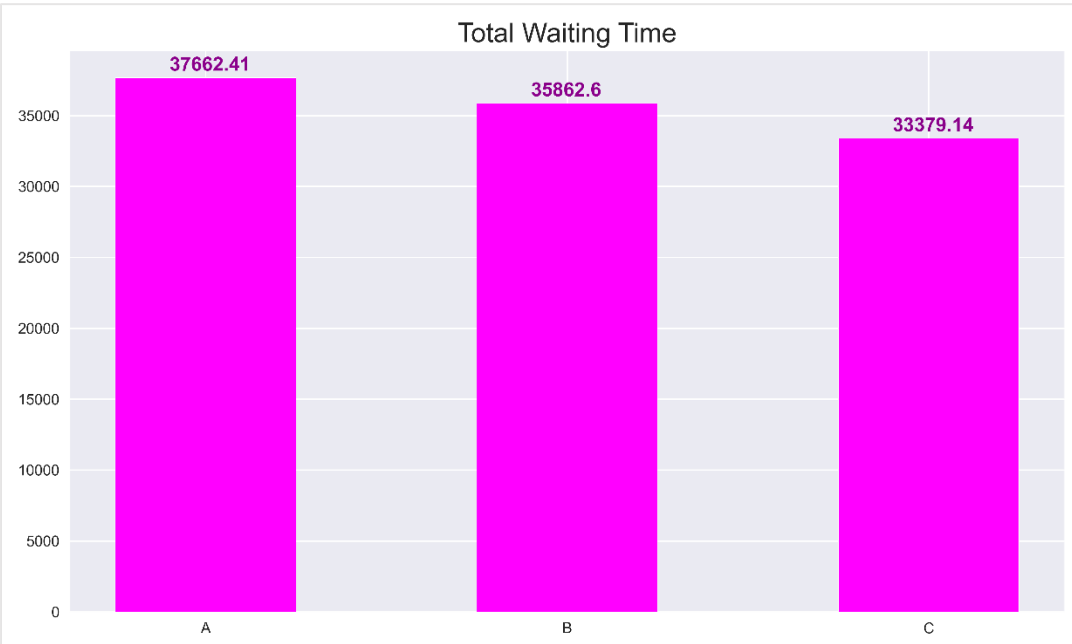


Figure 25-Average total waiting time comparison - All parts

Average Service Time

This statistic should not change, and even if it changes, it is because of randomness:

	Average Service Time A	Average Service Time B	Difference A & B	Average Service Time C	Difference A & C
A	9.875886	10.950076	-1.074190	10.843784	-0.967898
B	35.079490	32.143491	2.935999	31.534079	3.545411
C	19.189698	19.661423	-0.471725	14.859677	4.330021
D	42.541796	40.740847	1.800949	41.010446	1.531350
E	31.326775	25.475710	5.851065	31.399722	-0.072947

Figure 26-Average service time - All Parts

To make sure whether service time significantly changes in states B or C, we can perform statistical hypothesis tests like ANOVA. We only perform it on station A (because it's a repetitive task):

$$\begin{cases} H_0: \mu_A = \mu_B = \mu_C \\ H_1: ow \end{cases}$$

```
lis1 = np.array([x[0] for x in SVR_list_A])/np.array([x[0] for x in N_list_A])
lis2 = np.array([(x[0]+x[1]) for x in SVR_list_B])/np.array([(x[0]+x[1]) for x in N_list_B])
lis3 = np.array([x[0] for x in SVR_list_C])/np.array([x[0] for x in N_list_C])
print(lis1.mean() , lis2.mean() , lis3.mean() )
st.f_oneway(lis1 , lis2 , lis3)
```

executed in 13ms, finished a minute ago

```
9.875886022464524 10.950076440141164 10.843784356135604

F_onewayResult(statistic=2.4297912930561987, pvalue=0.10707047162455058)
```

Figure 27-ANOVA test over service time

For α=5%, we cannot reject the null hypothesis, so service time does not change in different states.

Also, the question may arise why server C's service time is not an integer (given that the service time is fixed)? This is because service time is only collected in periods 120 to 600.

To better explain, assume a machine enters server C at 590, and with a service time of 20 minutes, the service is completed at 610, but instead of adding 20 minutes of service time, only 10 minutes are added, because from 600 to 610 our simulation system should no longer collect data.

Average working Time

	Average working time A (for the whole station)	Average working time A (per server in the station)	Average working time B (for the whole station)	Average working time B (per server in the station)	Difference for whole station A & B	Difference per server A & B	Average working time C (for the whole station)	Average working time C (per server in the station)	Difference for whole station A & C	Difference per server A & C
A	476.862826	476.862826	948.164466	474.082233	-471.301639	2.780594	473.985833	473.985833	2.876994	2.876994
B	1327.089665	442.363222	1359.811666	453.270555	-32.722000	-10.907333	1299.640285	433.213428	27.449381	9.149794
C	380.134782	380.134782	414.847059	414.847059	-34.712278	-34.712278	331.276769	331.276769	48.858013	48.858013
D	1453.954481	363.488620	1469.579497	367.394874	-15.625017	-3.906254	1576.474928	394.118732	-122.520447	-30.630112
E	963.733170	321.244390	784.297922	261.432641	179.435248	59.811749	1069.887542	356.629181	-106.154372	-35.384791

Figure 28-Average working time - All Parts

The reason behind this negativity in rows D and E is that, in state A, machines enter servers D and E a little bit late and they are idler (due to the queues that are formed in posterior workstations).

But in state B and C, because we add either an additional server or reduces the service time, machines reach out to stations D and E faster and they work more.

In the following table you can see that in the specified time 120-600, more machines entered servers D and E.

	Number of Machines PART A	Number of Machines PART B	Number of Machines PART C
A	48.7	86.8	44.7
B	38.2	43.0	41.5
C	19.8	21.1	22.3
D	34.4	36.1	38.6
E	30.9	30.9	34.0

Figure 29-Number of machines entered each station in the specific period

Average Total Queue Length

	Average total queue length A	Average total queue length B	Difference A & B	Average total queue length C	Difference A & C
A	127.3	132.7	-5.4	128.0	-0.7
B	46.6	90.4	-43.8	42.2	4.4
C	18.5	24.1	-5.6	18.6	-0.1
D	16.9	20.7	-3.8	23.2	-6.3
E	12.4	8.1	4.3	19.3	-6.9

Figure 30-Average total queue length - All Parts

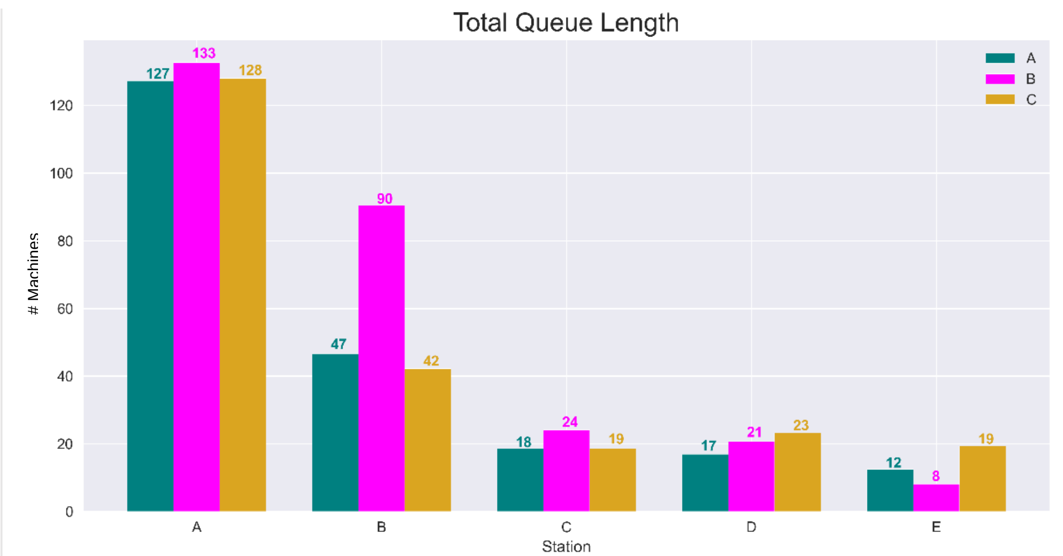


Figure 31-Total queue length comparison - All Parts

As it was previously mentioned, by adding another server to workstation A or reducing service time in C, there will be more machines to be served in workstations C and D and E (in the specified time of 120-600), so there will be an increase in queue lengths, working time of each station and number of servers being busy simultaneously.

Maximum Waiting Time

	Max WT PART A	Max WT PART B	Difference A & B	Max WT PART C	Difference A & C
0	480.0	480.0	0.0	480.0	0.0

Figure 32-Maximum waiting time - All Parts

Obviously, maximum waiting time occurred in workstation C.

Also, we can infer that reducing the service time of server C doesn't change the maximum waiting time.

Final Conclusion:

Our top measurement for selecting which improvement is better, is response time. Since metrics like queue length, maximum waiting time, and …, are not as important as response time is. Because this metric considers almost everything and our first goal must be to minimize the time each machine spends in the system.

So, as was mentioned earlier, according to Figure 23, **reducing the service time of server C by replacing it with a newer machine is the best option we can choose.**

Work Breakdown

Name	Work
Pedram Peiro Asfia	Python code and code description – Result & conclusion – Checking the report
Mahdi Mohammadi	Diagrams – Problem statement & modeling – Integrating the report – Checking the code