

Comp 9334 Project 2021 T1

Heterogeneous server farms

1 Introduction

The project has been split into two main sections, first is creating a simulation program to produce departure time for all three servers given the chosen load balancing algorithm, a mode, interarrival time and service time. The report will thus follow a similar format.

2 Correctness of the simulation code

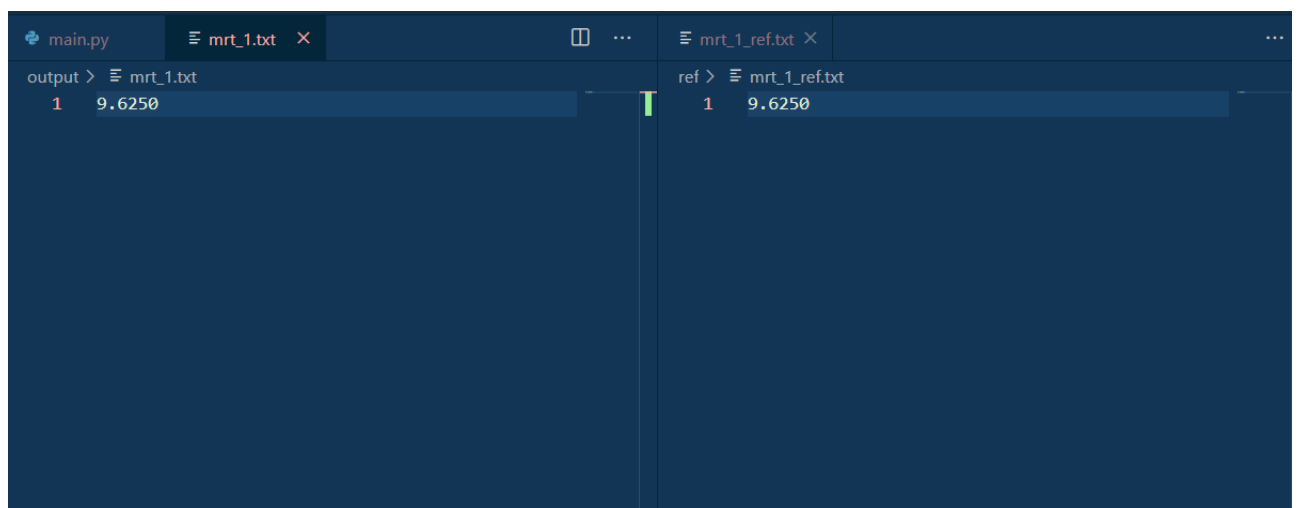
The simulation code has been written in python3 for the simplicity of running it without needing to compile. There are two modes as to how the code will be ran, random and trace, the results of trace can be compared more easily, so we will be looking at that first.

2.1 Correctness of the code running the trace mode

When running the trace mode, service time and interarrival time have been given, so only the mean response time and the departure time of 3 servers will be used to compare the correctness.

If version 1 of the load balancing algorithm is used, we can use the reference output from test 1 to compare for correctness.

First the mean response time.

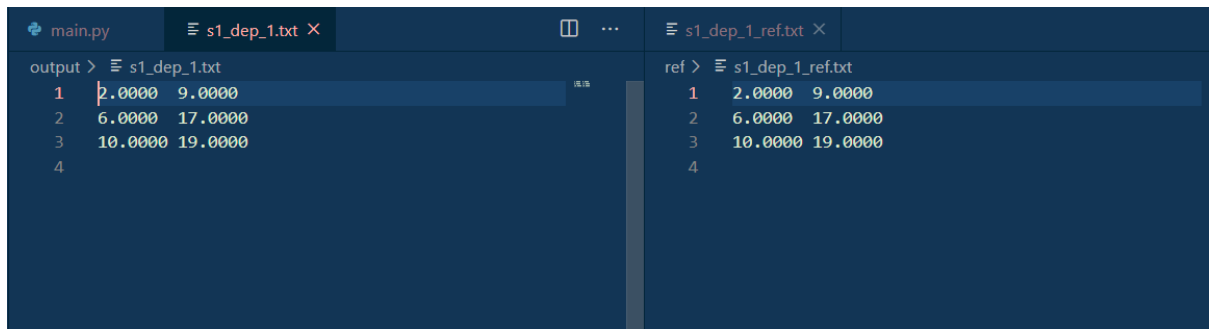


```
main.py  mrt_1.txt  mrt_1_ref.txt
output > mrt_1.txt
1 9.6250
ref > mrt_1_ref.txt
1 9.6250
```

Z5267511

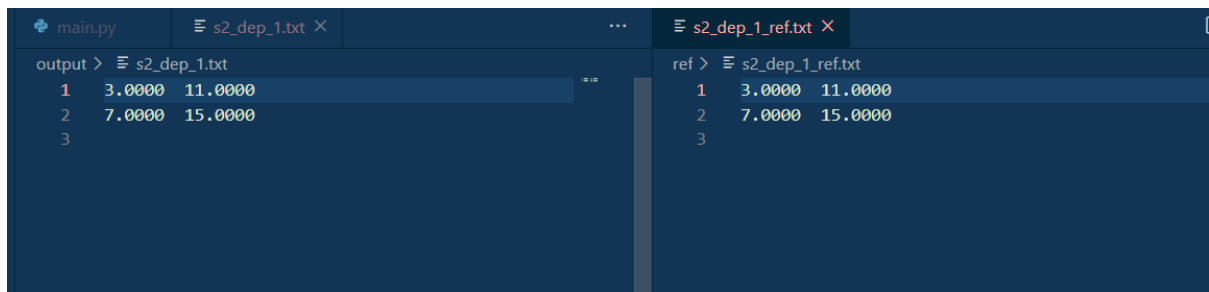
Then the departure time for each server.

Server 1:



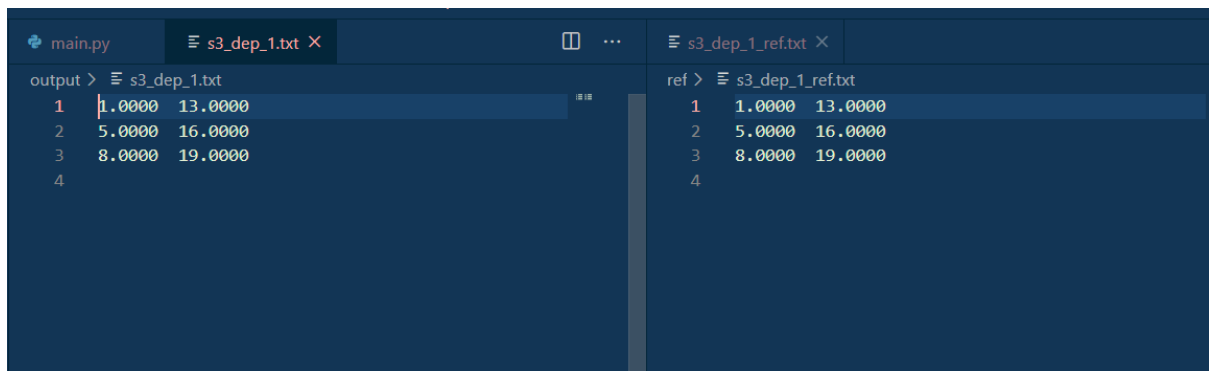
```
main.py s1_dep_1.txt s1_dep_1_ref.txt
output > s1_dep_1.txt
1 2.0000 9.0000
2 6.0000 17.0000
3 10.0000 19.0000
4
ref > s1_dep_1_ref.txt
1 2.0000 9.0000
2 6.0000 17.0000
3 10.0000 19.0000
4
```

Server 2:



```
main.py s2_dep_1.txt s2_dep_1_ref.txt
output > s2_dep_1.txt
1 3.0000 11.0000
2 7.0000 15.0000
3
ref > s2_dep_1_ref.txt
1 3.0000 11.0000
2 7.0000 15.0000
3
```

Server 3:

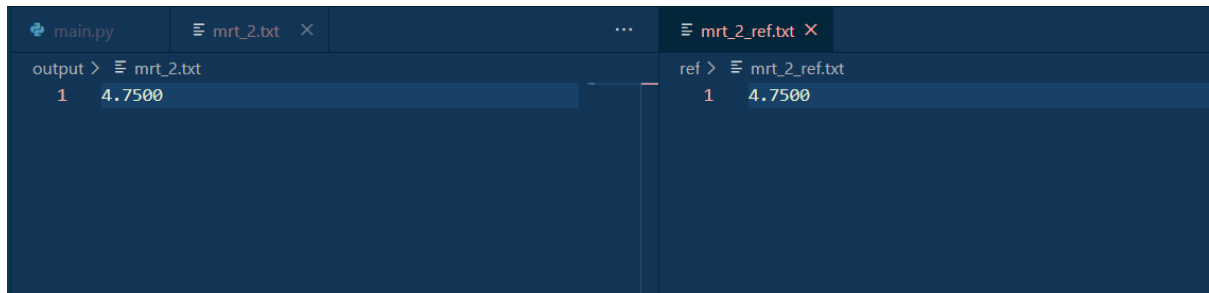


```
main.py s3_dep_1.txt s3_dep_1_ref.txt
output > s3_dep_1.txt
1 1.0000 13.0000
2 5.0000 16.0000
3 8.0000 19.0000
4
ref > s3_dep_1_ref.txt
1 1.0000 13.0000
2 5.0000 16.0000
3 8.0000 19.0000
4
```

The same can be done if version 2 of the loading balancing algorithm have been chosen, in this case, test 2 reference outputs will be looked at.

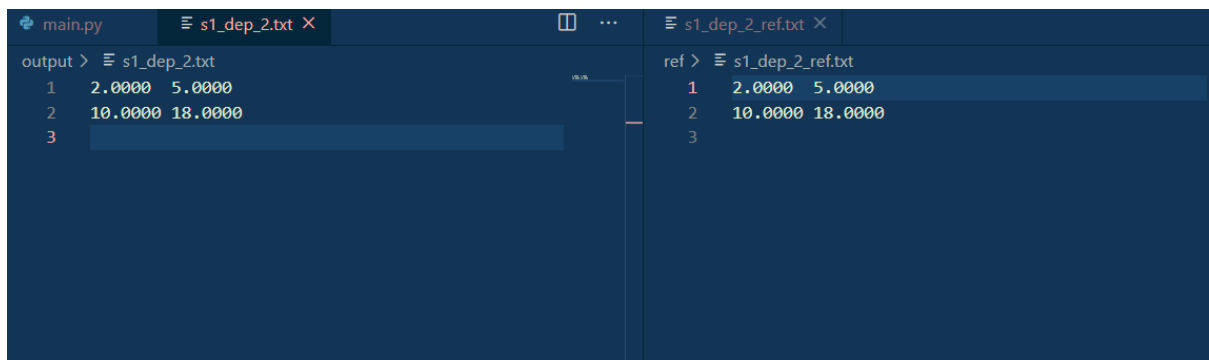
Z5267511

Mean response time:



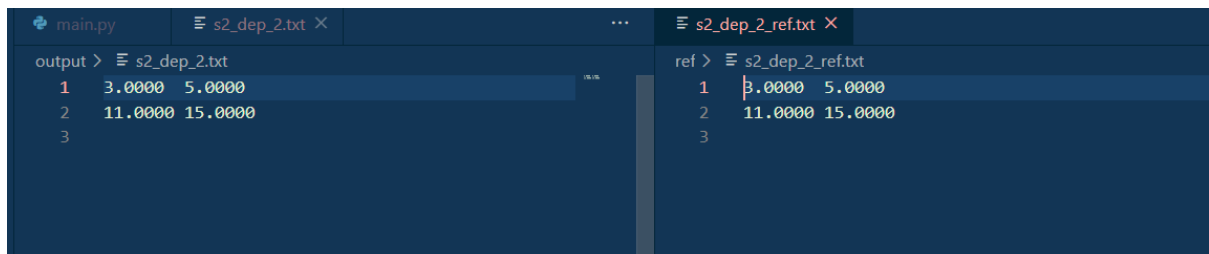
```
main.py | mrt_2.txt | ... | mrt_2_ref.txt
output > mrt_2.txt | ref > mrt_2_ref.txt
1 4.7500 | 1 4.7500
```

Server 1:



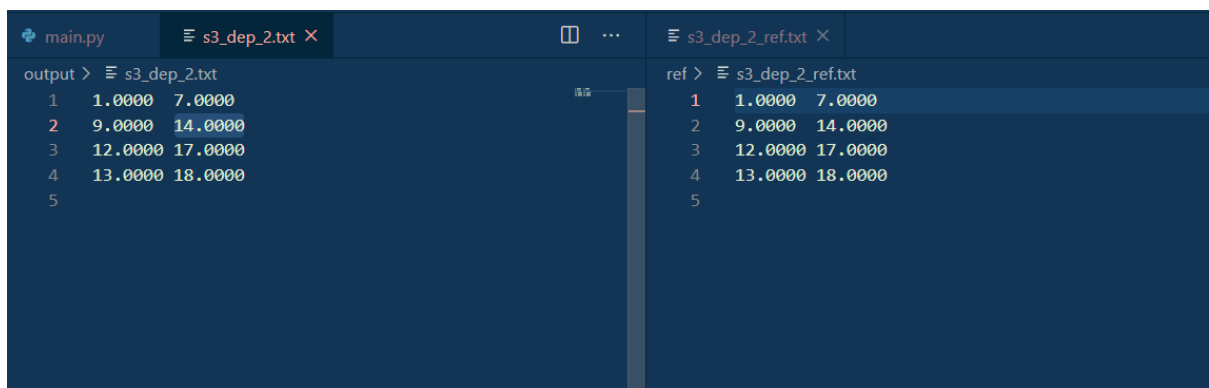
```
main.py | s1_dep_2.txt | ... | s1_dep_2_ref.txt
output > s1_dep_2.txt | ref > s1_dep_2_ref.txt
1 2.0000 5.0000 | 1 2.0000 5.0000
2 10.0000 18.0000 | 2 10.0000 18.0000
3 | 3
```

Server 2:



```
main.py | s2_dep_2.txt | ... | s2_dep_2_ref.txt
output > s2_dep_2.txt | ref > s2_dep_2_ref.txt
1 3.0000 5.0000 | 1 3.0000 5.0000
2 11.0000 15.0000 | 2 11.0000 15.0000
3 | 3
```

Server 3:



```
main.py | s3_dep_2.txt | ... | s3_dep_2_ref.txt
output > s3_dep_2.txt | ref > s3_dep_2_ref.txt
1 1.0000 7.0000 | 1 1.0000 7.0000
2 9.0000 14.0000 | 2 9.0000 14.0000
3 12.0000 17.0000 | 3 12.0000 17.0000
4 13.0000 18.0000 | 4 13.0000 18.0000
5 | 5
```

2.1 Correctness of the code running the random mode

Since the algorithm for the random and trace mode is identical, the correctness of the output for random depends on the correct inter-arrival probability distribution and service time distribution.

For the inter-arrival probability distribution, for λ , `random.expovariate` is used to generate exponentially distributed pseudo-random numbers.

```
def getArrivalTime(arrival_lines):
    lenda = float(arrival_lines[0])
    a1 = random.expovariate(lenda)

    # uniformly distributed in the interval [a2l, a2u]
    a2l = float(arrival_lines[1])
    a2u = float(arrival_lines[2])
    a2 = random.uniform(a2l, a2u)

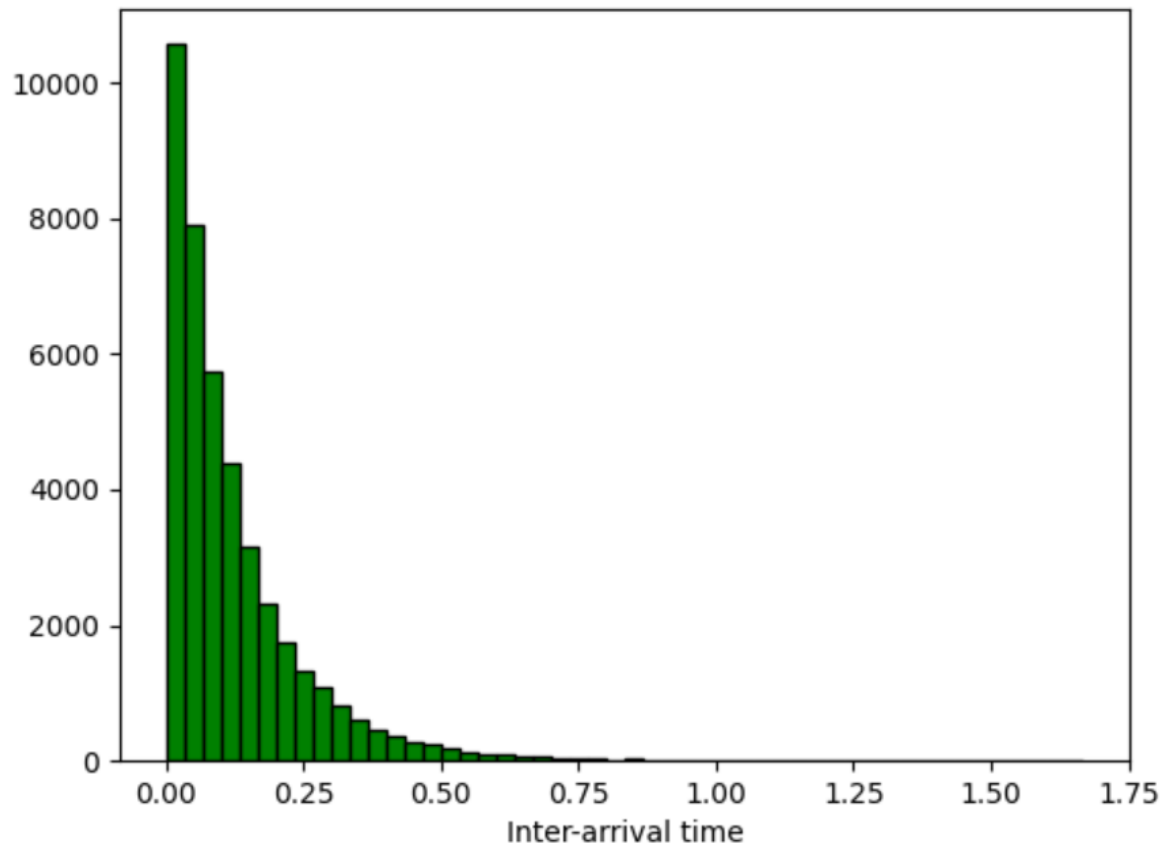
    # ak
    ak = a2 * a1

    return ak
```

Then, `Plot.py` is used to plot the inter-arrival distribution. The data used are:

Parameter	Value
Lambda	7.000
Time_end	5000
A2l	0.500
A2u	1.200
seed	1

The data produced an exponentially distributed histogram:



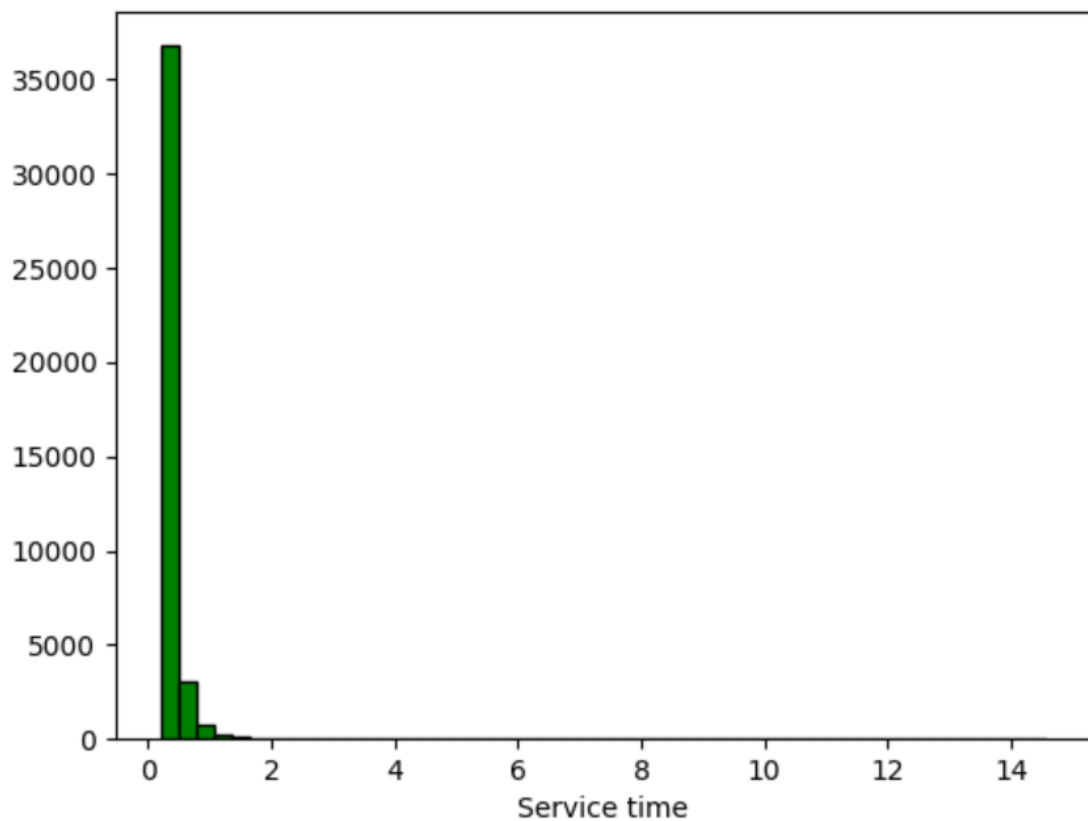
For the service time distribution, the function `getServiceTime` generates the service time using the probability density function in the project documentation combined with the alpha and beta values.

```
def getServiceTime(alpha, beta):
    gamma = (beta - 1)/(alpha**(1-beta))
    prob = numpy.random.uniform(0,1)
    while prob == 0:
        prob = numpy.random.uniform(0,1)
    prob = round(prob, 6)
    service_time = (prob*(1-beta)/gamma + alpha**(1-beta)) ** (1/(1-beta))
    return service_time
```

Then, Plot.py is used to plot the a histogram for the service time distribution. The data used are:

Parameter	value
Alpha	0.200
beta	3.500
Length of the list	40863
seed	0

The data produced the following histogram:



To test the correctness of the simulation for the random mode, the shell scrip run_many.sh is used to simulate the python program 100 times to get the mean response time for the test case 6.

Then the python program within_range.py is used to sort and find the largest and smallest output and to be compared to the range of mean response time for test case 6.

```

The largest mean response time is: 0.5595
The smallest mean response time is: 0.4686

```

The values are within the range of mean response time for test case 6 which is [0,4678, 0.5610].

We can repeat the method above for test case 5.

```

The largest mean response time is: 0.3082
The smallest mean response time is: 0.2774

```

The values are within the range of mean response time for test case 5 which is [0,276, 0.3110].

3 Reproducibility of the code

To prove that my code is reproducible, the program will be ran 100 times by run_many.sh and the mean response time will be appended to the mrt_6.txt.

```

Set as interpreter
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
This program reads in an input, triples it and then writes the result
to a file.
"""
import decimal
import sys
import os
import math
import random
import numpy

random.seed(10)
numpy.random.seed(10)
# run the load algo 1 on trace and random modes
def Load_Balancing_Algo1(p, s, mode, f, d, service_lines, arrival_lines, time_end):
    if mode == "trace":
        #simulate until all jobs have departed

        # for i in arrival lines
        sum_arrival = []

        # get the arrival time
        for i in range(len(arrival_lines)):
            if i == 0:
                sum_arrival.append(arrival_lines[i])
            else:
                sum_arrival.append(format(float(arrival_lines[i]) + float(sum_arrival[i-1]),'.4f'))
        new_list = list(zip(sum_arrival, service_lines))

```

```

main.py  mrt_6.txt  run_many.sh  plot.py  within_range.py
output > mrt_6.txt
1 0.4971
2 0.4971
3 0.4971
4 0.4971
5 0.4971
6 0.4971
7 0.4971
8 0.4971
9 0.4971
10 0.4971
11 0.4971
12 0.4971
13 0.4971
14 0.4971
15 0.4971
16 0.4971
17 0.4971
18 0.4971
19 0.4971
20 0.4971
21 0.4971
22 0.4971
23 0.4971
24 0.4971
25 0.4971
26 0.4971
27 0.4971
28 0.4971
29 0.4971
30 0.4971
31 0.4971
32 0.4971
33 0.4971
34 0.4971
35 0.4971
36 0.4971
37 0.4971
38 0.4971
39 0.4971
40 0.4971
41 0.4971
42 0.4971

```

4.0 Suitable values of d for load balancing

For solving this design problem, fixed values have been given as such:

Parameter	Values
f	1.5
β	3.5
α	0.2
$\alpha_{2\mu}$	1.1
$\alpha_{2\ell}$	0.8
λ	6.1
seed	0
Time_end	10000

Time end have been chosen to be 10000 to have enough jobs for mean response time to reach a steady state.

4.1 Best value of d for load balancing algorithm 1

Value of d	Mean response time
0	0.3822
1	0.3639
2	0.3735
3	0.3864
4	0.3977
5	0.4076
6	0.4161
7	0.4224
8	0.4264
9	0.4308
10	0.4340
11	0.4362
12	0.4385
13	0.4401
14	0.4411

First, I calculated the mean response time for different values of d without transient removal, from the results, it can be observed that the best mean response time was achieved at $d = 1$ for the load balancing algorithm 1.

4.2 Best value of d for load balancing algorithm 2

Value of d	Mean response time
0	0.3638
1	0.3741
2	0.3977
3	0.4160
4	0.4223
5	0.4265
6	0.4341
7	0.4366
8	0.4400

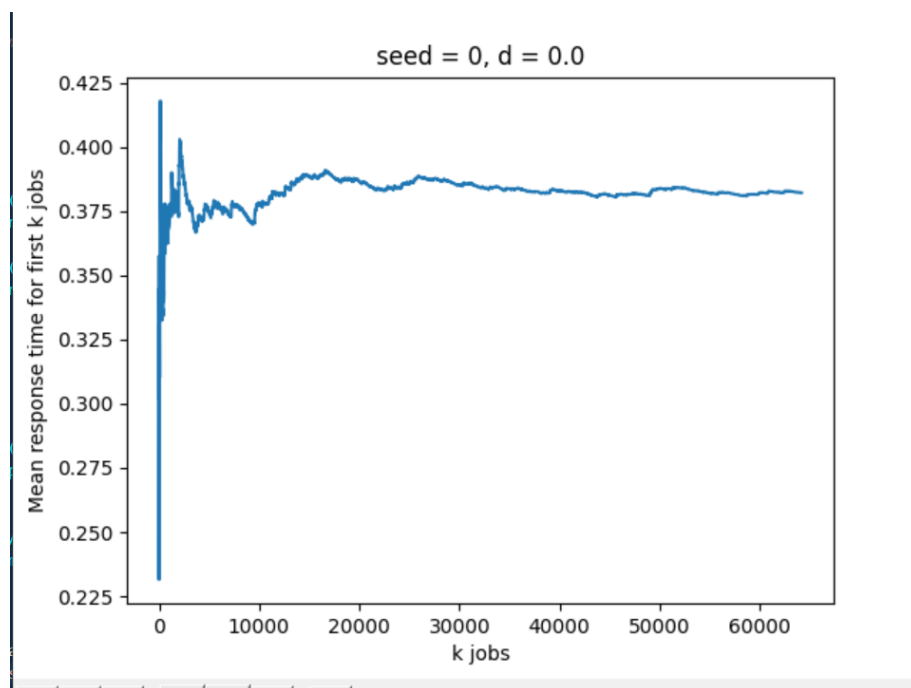
9	0.4415
10	0.4438
11	0.4449
12	0.4461
13	0.4461
14	0.4469

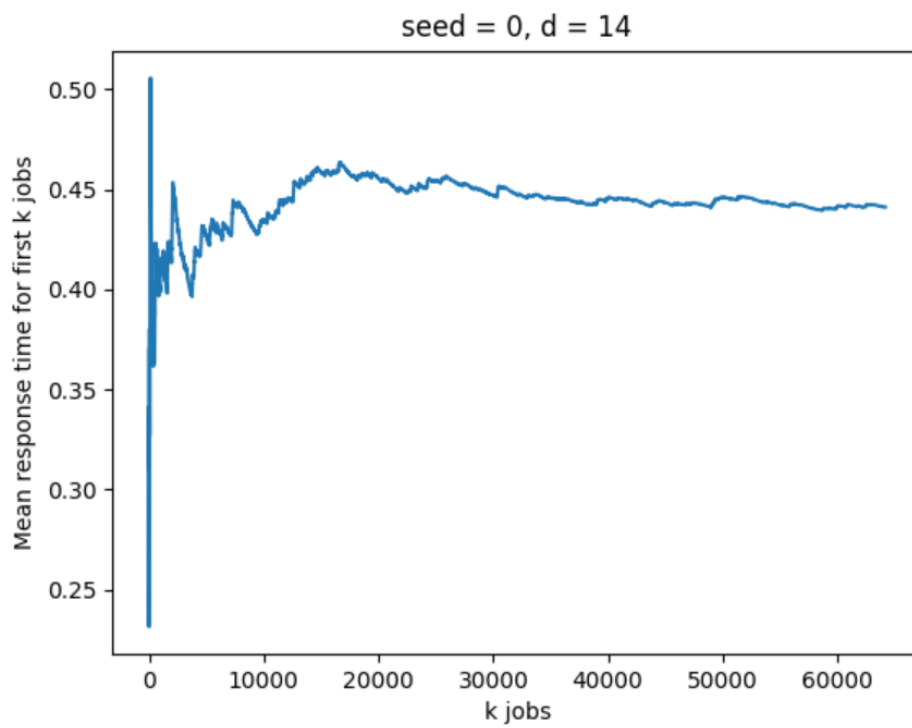
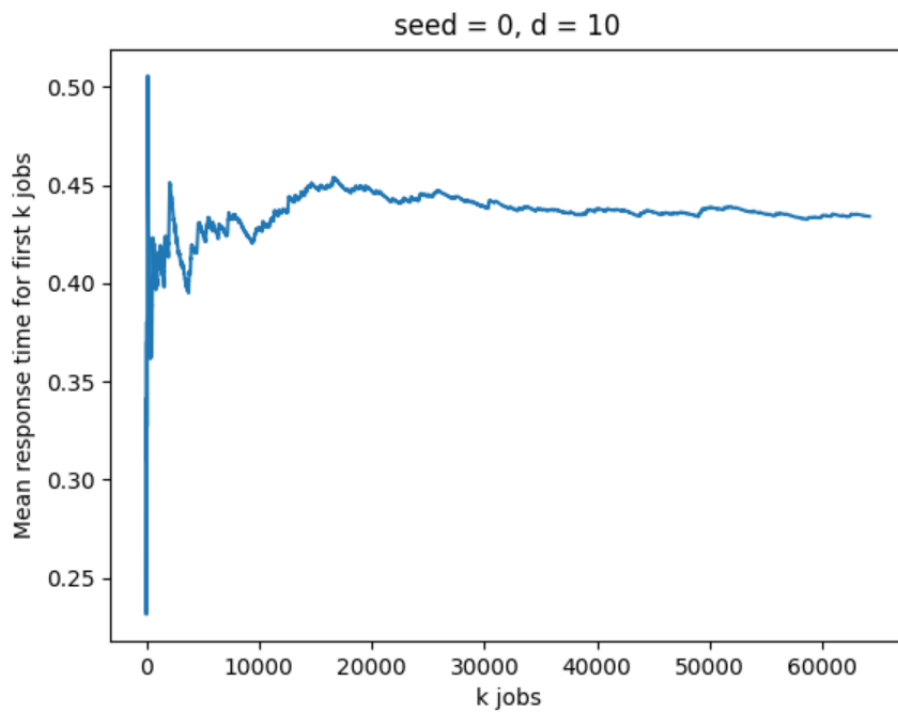
A similar approach is done for algorithm 2, without transient removal, the best mean response time is observed at $d = 0$.

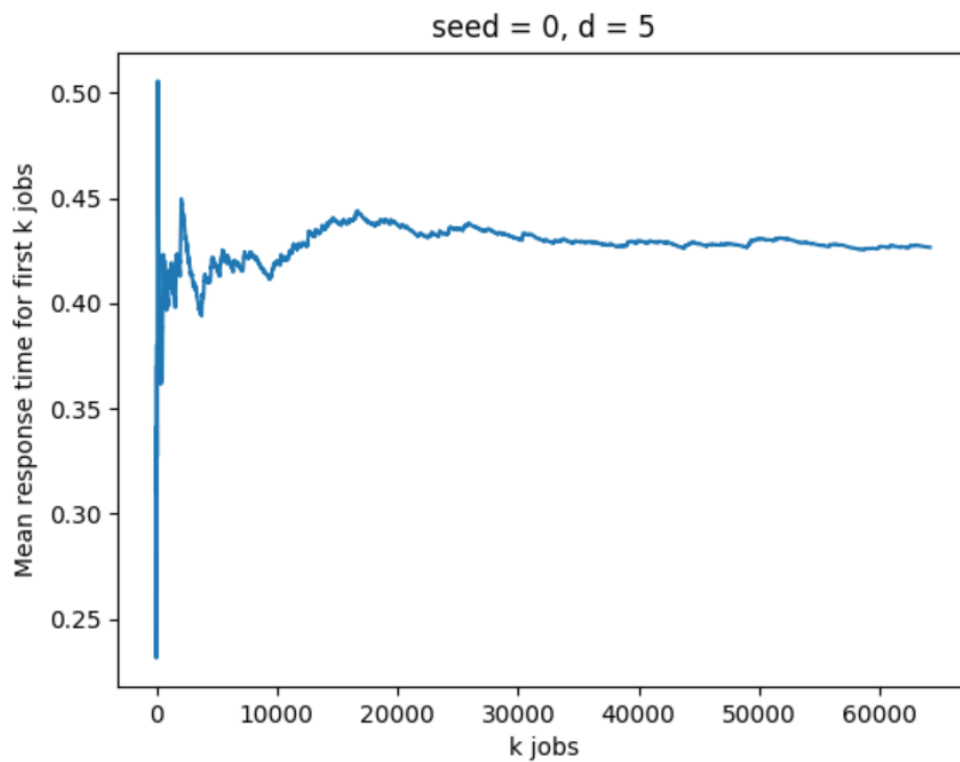
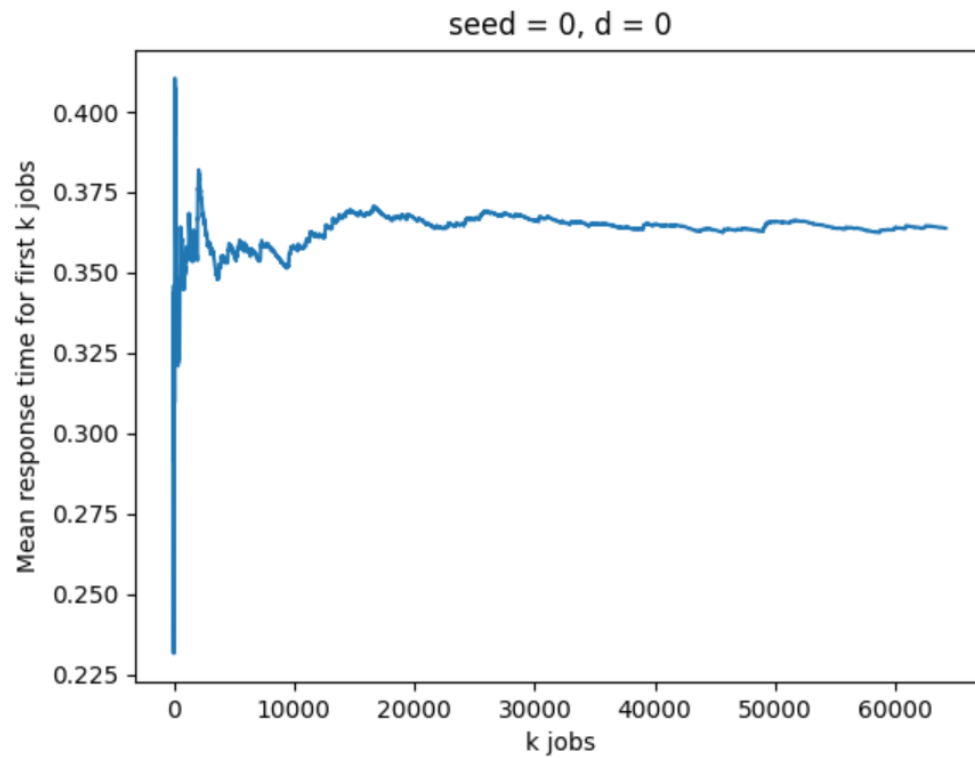
4.3 Transient removals

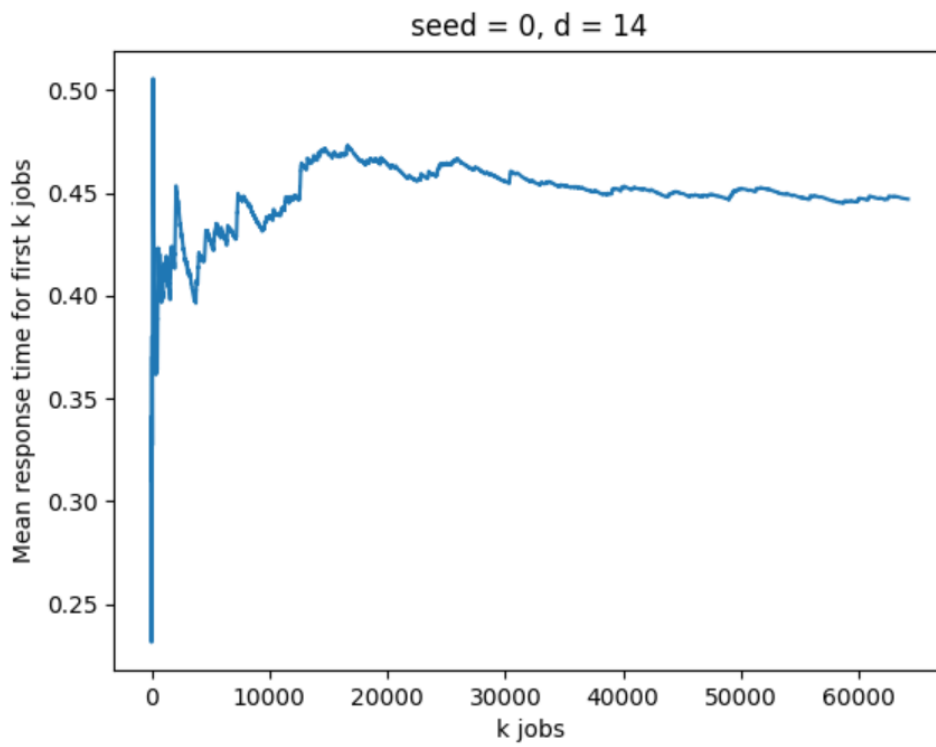
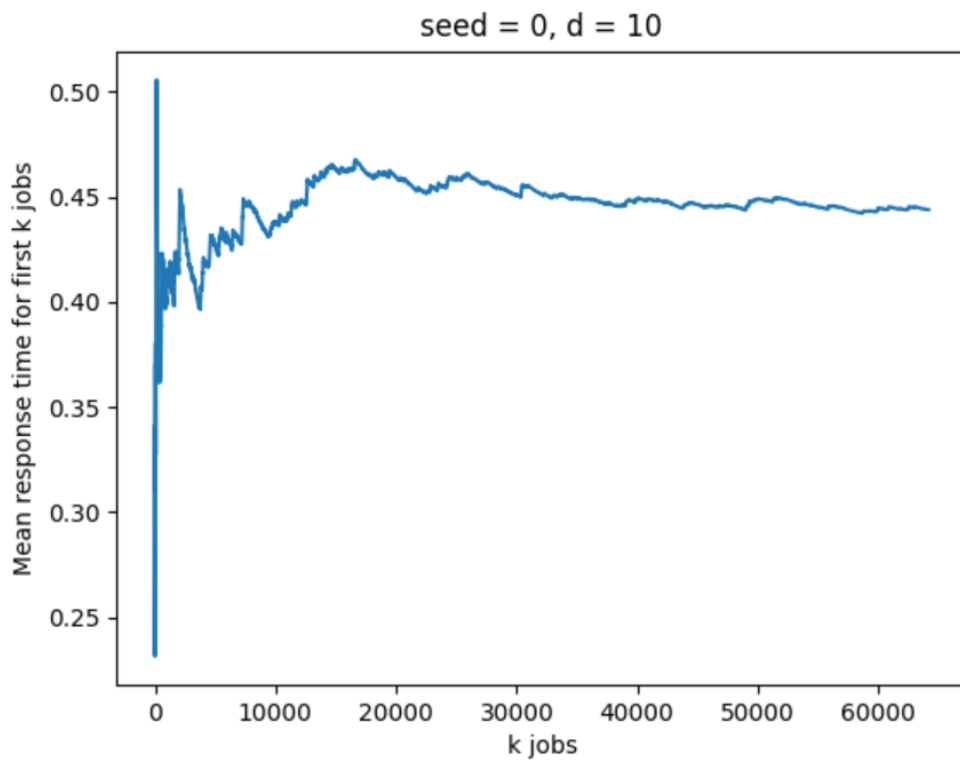
From the histogram made for service time, the service time ranges from 0 to 14. When graphing the mean response time for k jobs for seed = 0, and different values of d , it can be observed that the response time reaches a steady state after fluctuations at the beginning. Hence, the transient part can be removed, and it can be observed that the first 20000 jobs can be removed for both algorithm 1 and algorithm 2.

Algorithm 1 graphs:





Algorithm 2 graphs:



4.4 Confidence interval for Algorithm 1

The number of replications chosen is 60 because that is half of the number of replications for the t-distribution table. Since the number of replications is 60, according to the t distribution table, $t = 2$.

The formula for confidence interval is:

$$\left[\hat{T} - t_{n-1, 1-\frac{\alpha}{2}} \frac{\hat{S}}{\sqrt{n}}, \hat{T} + t_{n-1, 1-\frac{\alpha}{2}} \frac{\hat{S}}{\sqrt{n}} \right]$$

We can solve for the formula by solving the mean and the standard deviation first using the following formulas:

$$\hat{T} = \frac{\sum_{i=1}^n T(i)}{n}$$

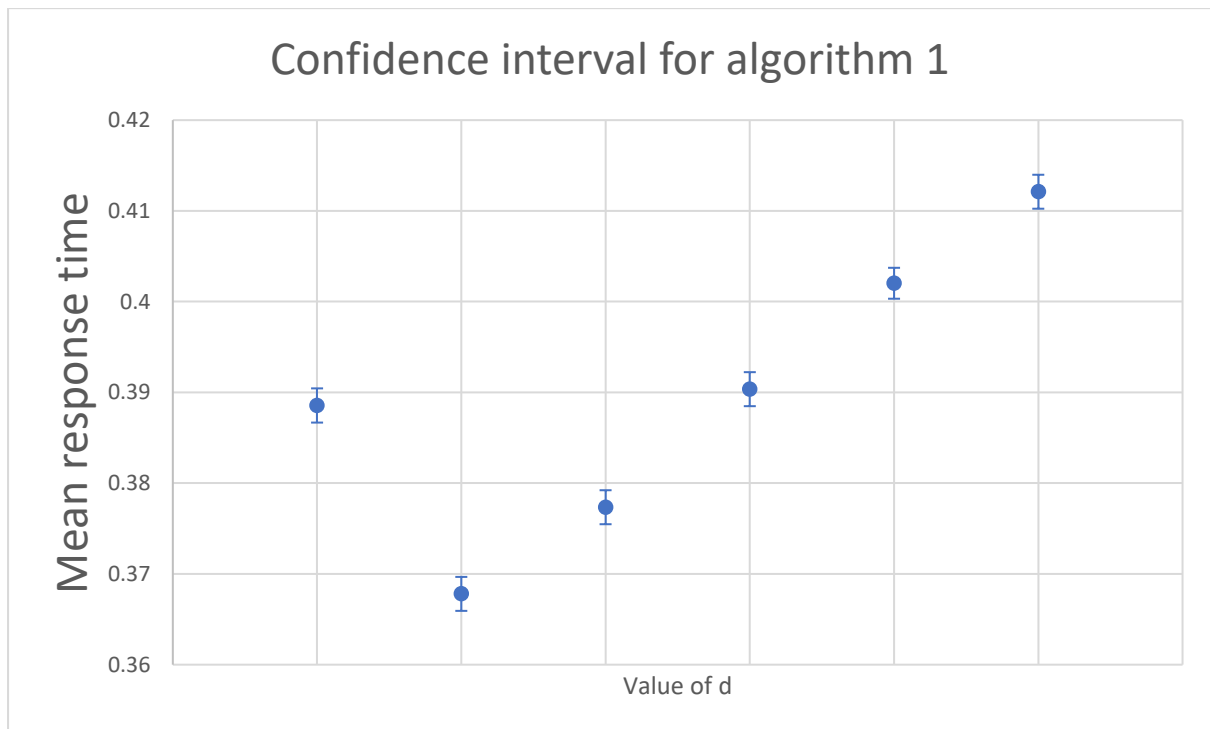
$$\hat{S} = \sqrt{\frac{\sum_{i=1}^n (\hat{T} - T(i))^2}{n - 1}}$$

To find the confidence interval, `run_seed.sh` is used first to run `main.py` 60 times with different seeds, then `mrt.py` to add the mean response time to `mrt.txt`, which will contain all 60 mean response time values. Then, `confidence.py` is used to find the confidence interval for the value of d .

The results are shown below.

Value of d	Confidence interval	Mean time
0	[0.386311, 0.390765]	0.388538
1	[0.365522, 0.369818]	0.36767
2	[0.370719, 0.374268]	0.372493
3	[0.388483, 0.392231]	0.390357
4	[0.400324, 0.403729]	0.402026
5	[0.410256, 0.413967]	0.412111

After computing the mean response time for values of d up to 5, I have noticed that the trend is that intervals will continue to increase, and the minimum point was at 1.



The observation from the graph is that the minimum point should occur between 0 to 2.

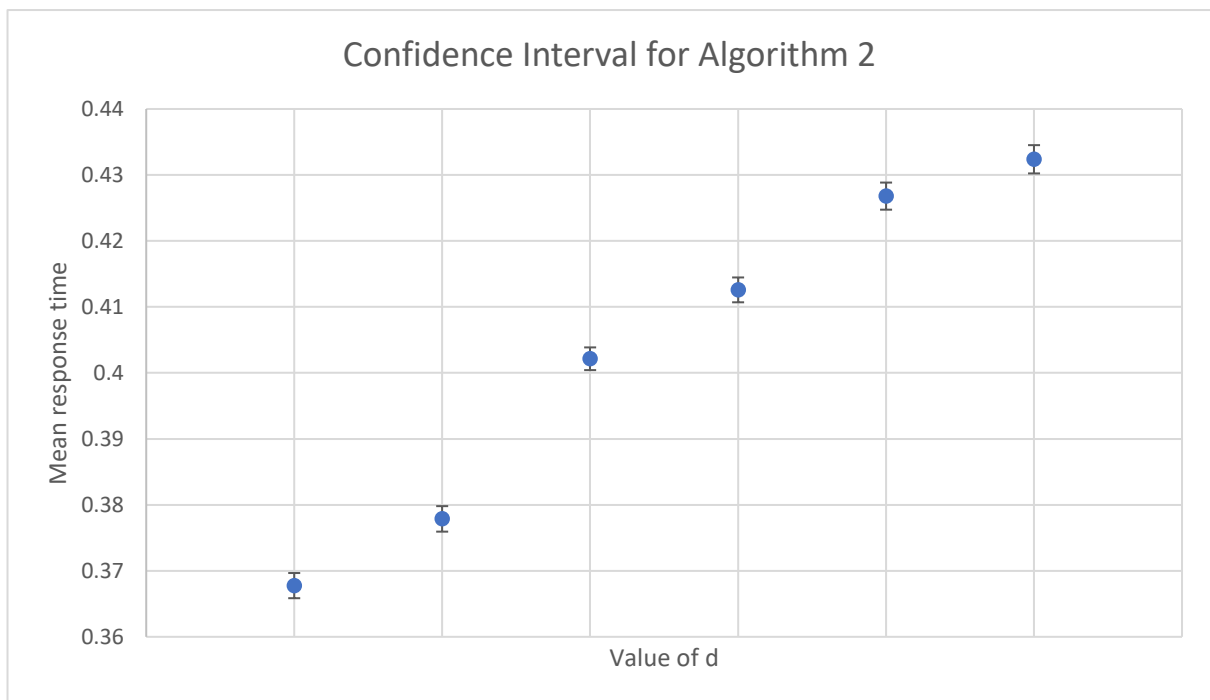
However, if we recall the if statement containing the value of d in algorithm 1:

else if $n_s == 0$ or $n_s \leq n_3 - d$ then

The value of d being 1 is exactly same as being $[0.1, 0.9]$, because n_3 will only ever be an integer. Therefore, the value of d that gives the best mean response time for load balancing algorithm 1 is $d = 1$.

4.5 Confidence Interval for Algorithm 2

Value of d	Confidence Interval	Mean time
0	[0.365458, 0.369871]	0.367664
1	[0.375558, 0.380191]	0.377875
2	[0.400247, 0.404379]	0.402313
3	[0.410701, 0.414466]	0.412584
4	[0.42474, 0.428848]	0.426794
5	[0.430234, 0.43452]	0.432377

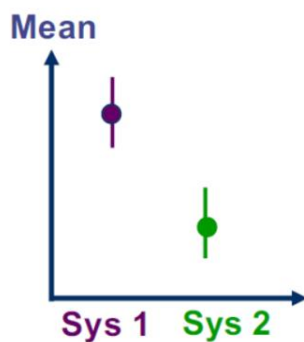


Because there are no overlaps, the least mean response time is achieved when the value of $d = 0$.

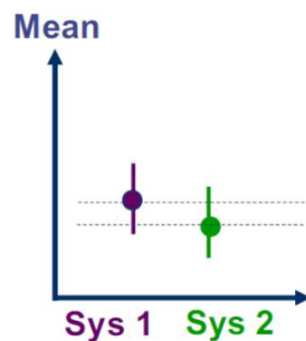
4.6 Can algorithm 2 achieve a lower mean response time compared to algorithm 1

	Algorithm 1	Algorithm 2
Value of d	1	0
Mean time	0.36767	0.367664
Confidence interval	[0.365522, 0.369818]	[0.365458, 0.369871]

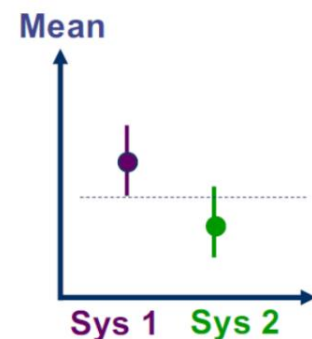
Statistically speaking, algorithm 2 can achieve a lower mean response time than algorithm 1 because the lower bound for algorithm 2 is less than the upper bound for algorithm 1, it also has a mean time that is slightly less than the mean time for algorithm 1.



CIs do not overlap
Mean of System 1
> Mean of Sys. 2



CIs overlap and
mean of a system
is in the CI of the
other: System are
not different



CIs overlap and
mean of any one is
not in the CI of the
other: do *t*-test

However, the confidence interval of algorithm 2 overlaps the mean of algorithm 1, we can actually conclude that system are not different, hence algorithm 2 is no different to algorithm 1.