STAT3989 IT Course Group Project Report

Name	UID
Huang Yifeng	3035334905
Ke Daqi	3035234600
Sun Xiaotong	3035448122
Tao Yufeng	3035447049

Queuing Efficiency for Banking Services
- Single Line vs Multiple Lines

Abstract

Queueing efficiency plays an important role in operational management, and the single-line and multiple-line queueing models generalize the majority of situations in real life. Taking the banking service as a specific topic, this project compares the efficiency of these two methods using daily average customer (customers who have been served) waiting time as an indicator. With the customer arrivals and service controlled, we perform simulations under the two models using a C++ program and do the statistical analysis by R. The simulation results show that the single-line model is more efficient than the multiple-line model in most cases, conforming to the fact that banks usually adopt the single-queue model.

Table of contents

Motivation

As a crucial branch of operations research, queueing theory has been extensively applied in a wide range of fields, such as business logistics, industrial engineering, project management, etc. In the financial industry, especially retail banking, queueing efficiency is also a critical issue, since the waiting time is one of the most important factors that customers would consider. The purpose of this project is to compare the queueing efficiency between the single-line case and the multiple-line case for banking services with daily average customer (customers who have been served) waiting time as the efficient indicator. In the single-line case, the system will assign a service window to each customer according to early availability on a first come, first served basis. While in the multiple-line case, we assume that customers will choose the shortest queue, and no jockeying happens, that is, customers would not switch between different queues.

Data

The algorithm requires three inputs, the number of customers (NC), the number of windows (NW), and the number of simulations (NS). In addition, we use uniform distribution in random number generation of customers' arrival time and service time which are thus predetermined to be the same under both the single-line model and the multiple-line model in each simulation. Seeds are set to ensure the result is reproducible.

Methodology

This section will firstly concentrate on the programming methodology for the single-line case and then highlight how the multiple-line case can be derived analogously.

As a fundamental unit, a customer is endowed with several important features: arrival time, waiting time, service time, etc., all of which are captured by a user-defined structure named customers. As mentioned above, random data generation provides known values of arrival time and service time to be modelled afterwards. What also needs to be predetermined are the number of service windows (NW) and the total count of customers (NC).

Following necessary initializations, all customers are sorted in ascending order based upon their arrival time previously generated. Selection sort is adopted among sorting algorithms for simplicity. Next, all customers are put in queues corresponding to their service windows. It merits the attention that in the single-line case we construct the hypothetically multiple lines which customers are distributed into obeying the rule: always choose the line with the least waiting time. This construction is for easy comparison purpose later with the multiple-line case and also logically equivalent to the actual single-line queuing scenario. Technically, the data structure queue is utilized because of its first-in-first-out principle and the struct customers are moved in and out accordingly. In the end, for single-line case and multiple-line case respectively, a daily average waiting time of all customers actually served for each given set of inputs can be calculated and output to two separate text files (.txt). For subsequent statistical analysis in R, we repeat the whole procedure for a specified number of times (NS) and append each outcome in the output text file with space as the delimiter.

The primary difference between the single-line case and the multiple-line case is the way to decide which queue to join for the new customer. For the single-line case, the new customer would join the queue where the total service time of all customers is the least. While for the multiple-line case, the new customer would join the queue where the total number of customers is the smallest. Whenever the numbers are the same in different queues, we assume that the new customer would join the left-most queue where the equality in customer numbers occurs.

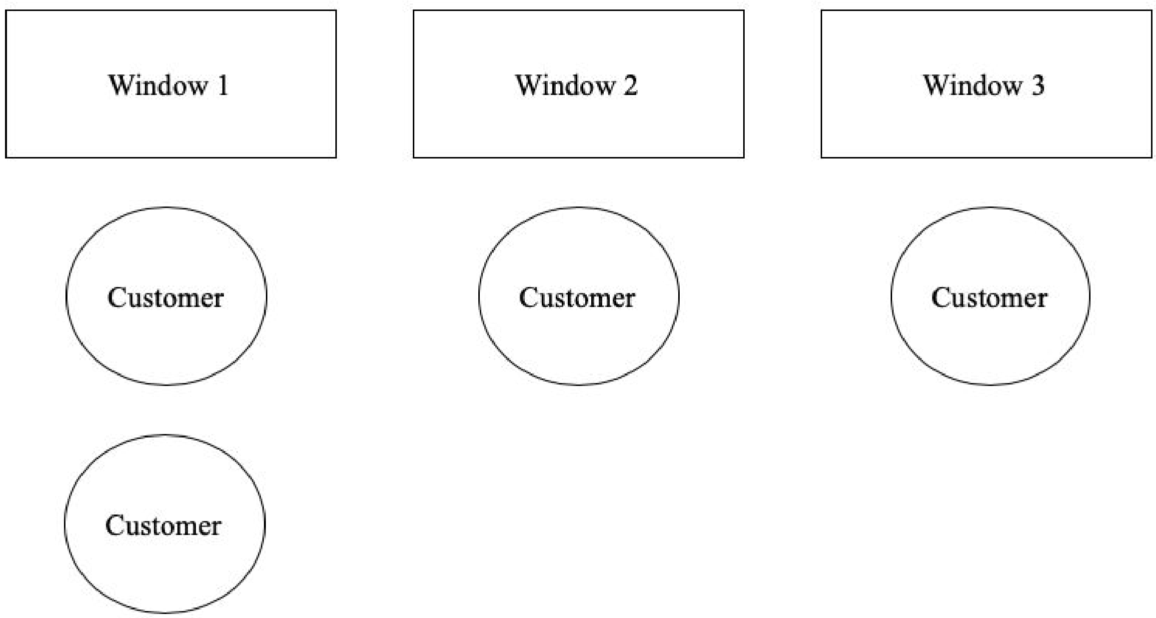

Fig 1. Illustration of equality issue in multiple lines case

In the multiple-line case, the next customer would join the 2nd queue, i.e. Window 2. The reason is that the number of customers in front of Window 2 and Window 3 are the same and the smallest, and the Window 2 is the left-most queue between these two.

Analysis

For each set of windows and customers, we run 100 simulations for both single-line case and multiple-line case, and the result is visualized using R. There are 1200 simulations in total, and the summary statistics is shown in Table 1. In this table, Q1, median, and Q3 of the average waiting time are identified, and "More efficient" column represents the percentage of the cases out of 100 simulations where the corresponding queueing method is more efficient.

Windows	**customers**	**Single/Multiple**	**Q1**	**Median**	**Q3**	**More efficient**

2	50	Single	9.7426	12.43835	17.926	100%
2	50	Multiple	22.13675	27.8719	38.71125	0
2	70	Single	53.09185	58.8116	63.7614	100%
2	70	Multiple	100.1679	108.2015	113.6265	0
3	50	Single	0.24	0.34	0.44	100%
3	50	Multiple	0.67	1.88	2.69	0
3	70	Single	6.222255	7.3	8.659245	100%
3	70	Multiple	15.4318	18.5683	21.63235	0
4	50	Single	0.18	0.31	0.58	Equal
4	50	Multiple	0.18	0.31	0.58	
4	70	Single	1.307145	1.75714	2.02857	100%
4	70	Multiple	4.611495	5.67143	6.58571	0

Table 1: Summary of Simulation Result

In most cases, the average waiting time is shorter if customers are in one single line. However, when the number of windows is 4 and the number of customers are 70, the average waiting time in single-line and multiple-line cases are the same. A possible explanation is that, the windows are usually available (i.e. customers do not need to wait) and there may be one person in the queue most of the time, and thus, the queue with the shortest service time in total and the queue with the smallest number of people waiting could be the same. This phenomenon is likely to occur when the ratio of total customers over the number of windows gets smaller.

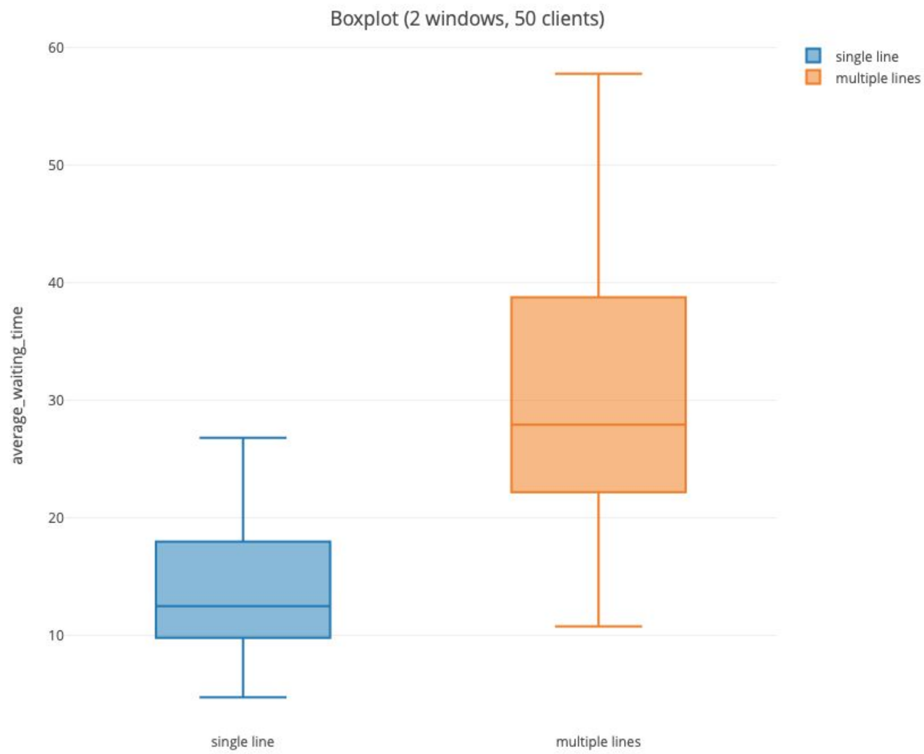

Fig 1. Boxplot for average waiting time (2 windows, 50 customers)

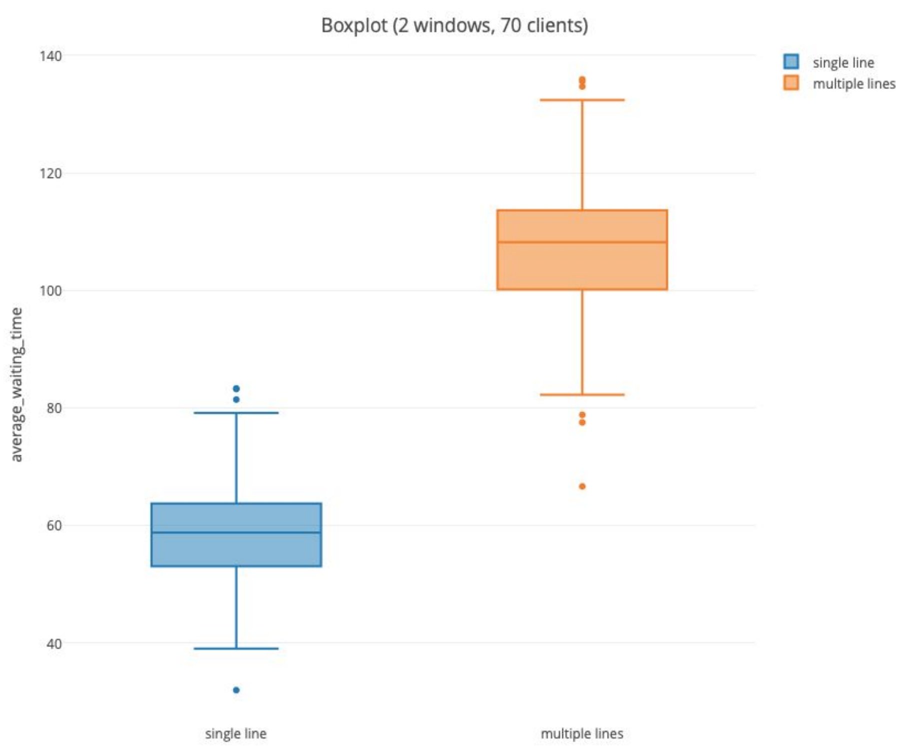

Fig 2. Boxplot for average waiting time (2 windows, 70 customers)

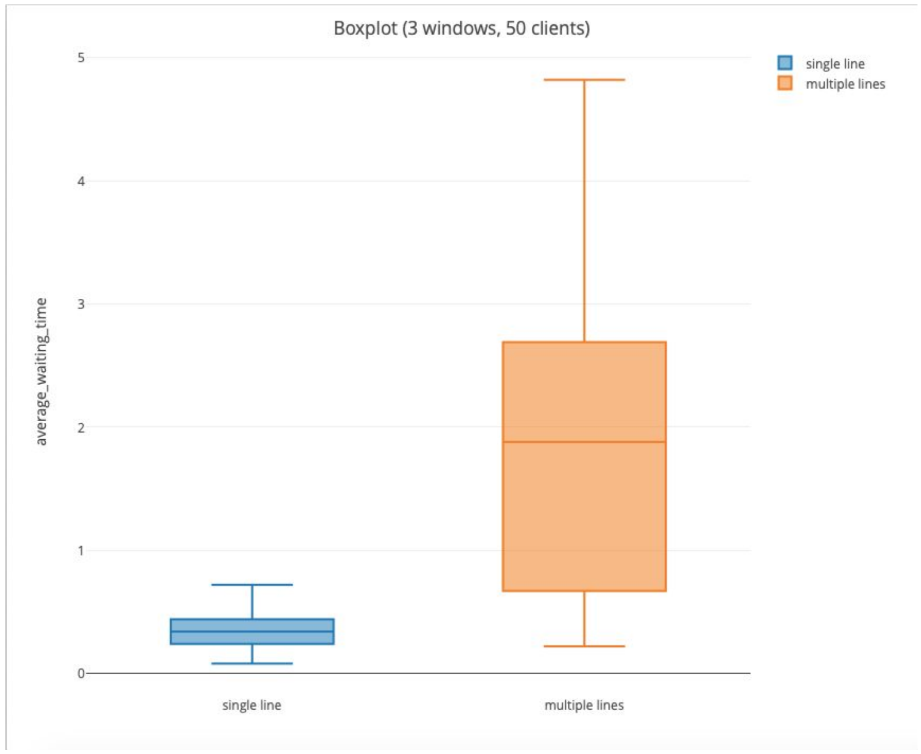

Fig 3. Boxplot for average waiting time (3 windows, 50 customers)

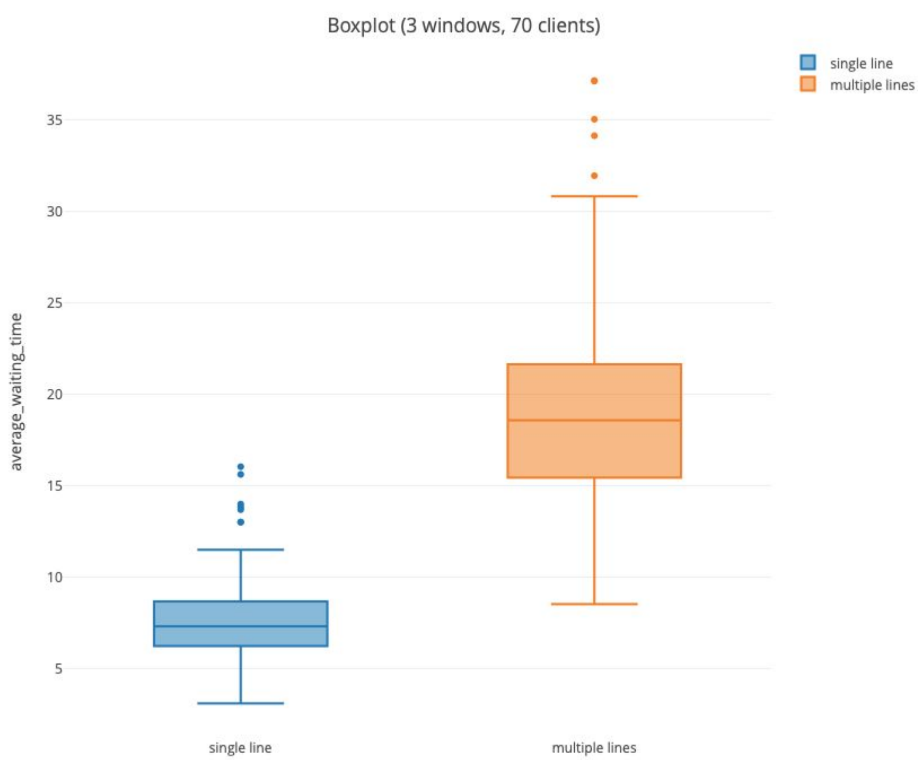

Fig 4. Boxplot for average waiting time (3 windows, 70 customers)

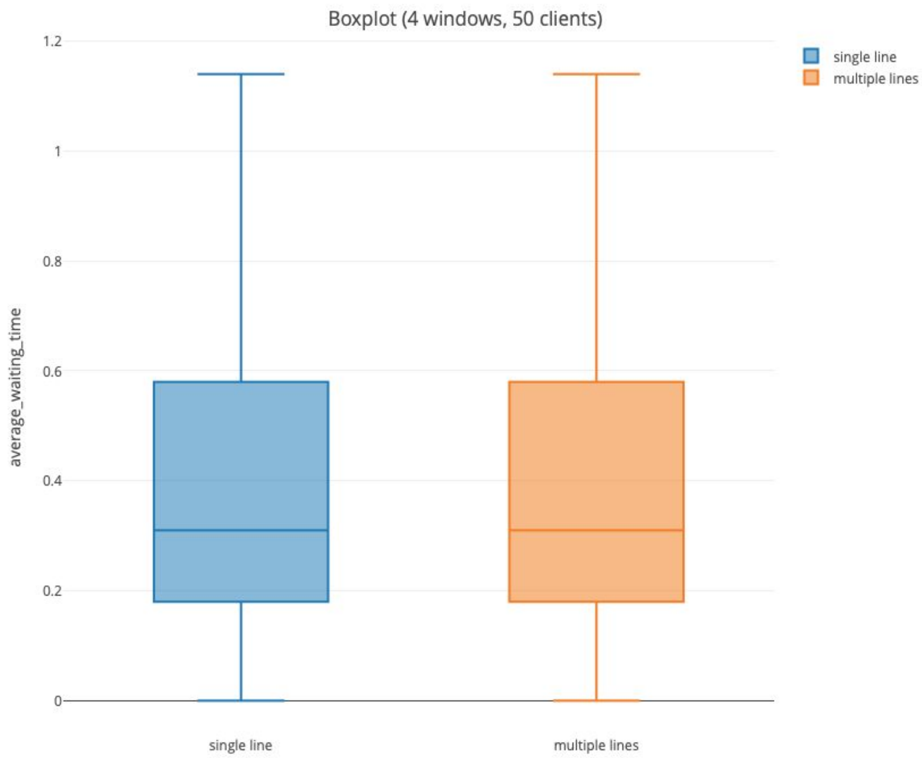

Fig 5. Boxplot for average waiting time (4 windows, 50 customers)

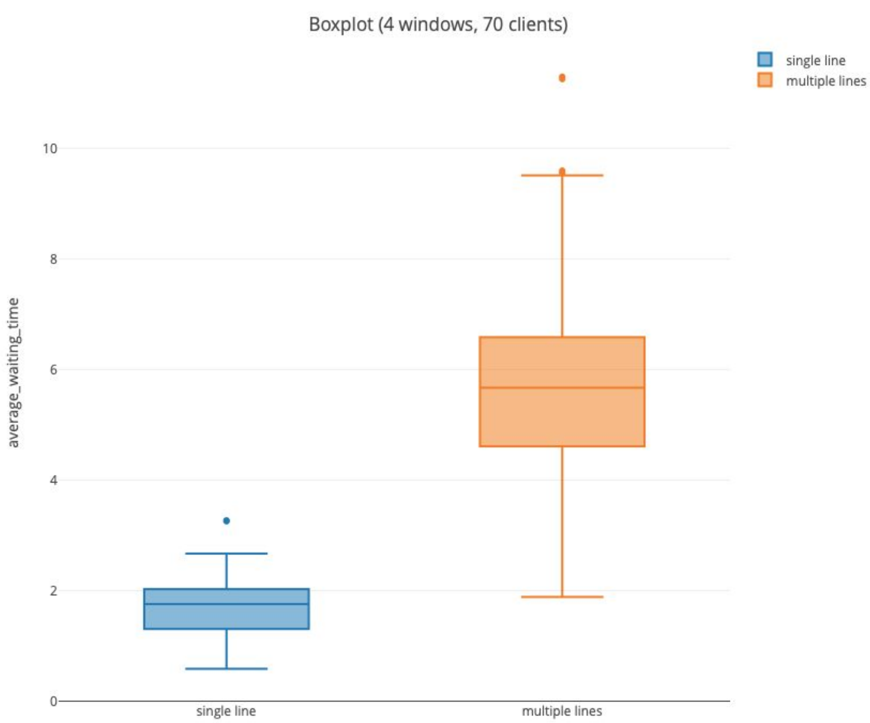

Fig 6. Boxplot for average waiting time (4 windows, 70 customers)

Figure 1 to 6 indicates that generally speaking, single-line cases tend to outperform multiple-line cases.

Discussion

1. Extension

Our project successfully supports that single line is more efficient in most cases which is in accordance with the existing queueing system in the majority of banks. Nevertheless, if we generalize the application of queuing theory to other fields, multiple-line queueing systems could also prevail. For instance, multiple-line models are usually adopted in supermarket checkout procedure, given that the service time is significantly short and calling numbers one by one which is required in single-line system but not in multiple-line system is relatively long in proportion, thus it is more efficient for customers queue right after the one who is being .

Another scenario where the multiple-line case may outperform the single-line case is when the customers' service time varies significantly. Assume there are two windows and each of them is occupied by 1 customer with serving time 2 and 1 respectively. Assume later there will be only three more customers (customer 1, 2, 3) coming at time 0, 1 and 2. Assume customer 1's serving time is long enough to block the window he chooses and customer 2's serving time is 1. The tables below shows if these three customers are waiting or being served.

1) Multiple lines

	Time 0-1	Time 1-2	Time 2-3	Time 3-4
Window1	1wait	1wait	1serve	1serve
Window2		2serve	3serve	

2) Single line

	Time 0-1	Time 1-2	Time 2-3	Time 3-4
Window1		2wait	2serve, 3wait	3serve
Window2	1wait	1serve	1serve	1serve

The waiting time of customer 1, 2, 3 for multiple lines is 2 which is shorter than 3 for single line. Similar as the prisoner dilemma, although each one is making the best choice for oneself, the overall result may not be the optimal one.

2. Limitations

Our project uses uniform distribution to generate the arrival time for customers which may fail to reflect reality very well as Poisson distribution is better for the independent arrival model. Also the mean arrival numbers vary among different time during each day. A better solution may be generating data by using Poisson distribution with different means for different time periods.

The service time in this project is uniformly generated. To predict more accurate average waiting time, each bank can collect the service data and use the simulated distribution instead.

The criterion in this project to judge whether waiting in single line or multiple lines is better is the average waiting time. Other criteria such as waiting space, whether it provides a better sense of fairness are not considered. To discuss the latter topic, further studies may calculate the variance of each customers' waiting time. The way with smaller variance is more fair to every customer.

In the real world, banks may open some vip windows. These windows only work for vip customers when there are vip waiting but can also serve for general customers when they are free. This project only considers the case with windows open to all customers.

Conclusion

This project generates data of arrival and service time for each customer in a bank queuing model and calculates the average waiting time for single line and multiple lines with C++. Simulation is done multiple times under different window numbers and different customer number levels. The results are then analyzed using R and waiting in single line is considered to be more efficient in most cases.

Reference

HachiLin. (2018, March 31). Queue- bank queuing model simulation [Blog post]. Retrieved from https://blog.csdn.net/Hachi_Lin/article/details/79774526

Appendix

C++

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <queue>
#include <fstream>
using namespace std;
//declare the number of service windows as K, the total count of customers as N, the number of
simulations required as S
int K,N,S;
//define a struct to represent time by hour and minute
struct thetime
{
```

```
	int h;  //hour
	int m;  //minute
};
//define all attributes of a customer in a struct
struct customers
{
	struct thetime arrive_time;	//arrival time
	struct thetime wait_time;	//waiting time
	struct thetime start_time;	//service starting time
	int business_time;	//duration of the corresponding customer service(Assumed as integers
and between 1-30 mins)
	struct thetime end_time;	//ending time of the service
	int in_bank_number;		//queuing number obtained upon arrival
};

//Function declaration
void customers_time(struct customers &c, int index);
void customer_sort(customers customer[]);

void customers_in_queue(queue<customers> cus_queue[],customers customer[],int
each_queue_cus_number[], char label);
void leave_queue(queue<customers> cus_queue[],customers customer);

int judge_queue_in_M(queue<customers> cus_queue[],customers &customer,int
each_queue_cus_number[],int index);
int judge_queue_in_S(queue<customers> cus_queue[],customers &customer,int
each_queue_cus_number[],int index);

void output (customers customer[], int each_queue_cus_number[], ofstream &oufile);

int main()
{
	//interactively receive the required inputs
	cout<<"Number of Windows (NW): ";
	cin>>K;
	cout<<"Number of Customers (NC): ";
	cin>>N;
	cout<<"Number of Simulations (NS): ";
	cin>>S;
	//file output stream to create two new files and write the simulation data in them
	ofstream oufile_m;//record the multiple-line data
	ofstream oufile_s;//record the single-line data
```

```
oufile_m.open ("simulation_data_m.txt", ios::app);//write in by appending one row each time in the following for-loop
oufile_s.open ("simulation_data_s.txt", ios::app);
if(oufile_m.fail()||oufile_s.fail())
cout << "File failed to open!" << endl;//signify file creation failure
else
{
oufile_m<<"NW: " << K << " NC: " << N << " NS: " << S << endl;
oufile_s<<"NW: " << K << " NC: " << N << " NS: " << S << endl;
char label;//to be used later to distinguish between multiple-line cases and single-line cases
oufile_m << "Multiple case:" << endl;
oufile_s << "Single case:" << endl;
//loop of simulation
for (int i = 0; i<S; i++)
{
customers customer_m[N];//an array of user-defined structures to represent N customers' information
queue<customers> cus_queue_m[K];//an array of queues to model the queues of each service window
int each_queue_cus_number[K];//count the number of queuing customers at each window (K windows in total)
//initialization of attributes of each customer
for(int j=0; j<N; j++)
customers_time(customer_m[j],i+j);
//initialization of numbers of queuing customers at each window
for(int j=0; j<K; j++)
each_queue_cus_number[j]=0;
//sort according to customers' arrival time
customer_sort(customer_m);
label = 'M';//multiple-line cases
//assign queuing numbers upon arrival
for(int j=0; j<N; j++)
customer_m[j].in_bank_number = j + 1;
//put customers in queues
customers_in_queue(cus_queue_m,customer_m,each_queue_cus_number,label);
output(customer_m,each_queue_cus_number, oufile_m);

//analogous procedure for single-line cases
customers customer_s[N];
queue<customers> cus_queue_s[K];
for(int j=0; j<N; j++)
customers_time(customer_s[j],i+j);
for(int j=0; j<K; j++)
```

```
        each_queue_cus_number[j]=0;
        customer_sort(customer_s);
        label = 'S';
        for(int j=0; j<N; j++)
        customer_s[j].in_bank_number = j + 1;
        customers_in_queue(cus_queue_s,customer_s,each_queue_cus_number,label);
        output(customer_s,each_queue_cus_number, oufile_s);
        }
        }
        return 0;
}
//initialization via random generation
void customers_time(struct customers &c, int index)
{
        //set seeds for different customers using distinct index parameters
        srand(index);
        //randomly generate customers' arrival time and service durations
        c.arrive_time.h=9+rand()%8;
        c.arrive_time.m=rand()%60;
        c.business_time=rand()%30+1;
}
//sort all customers in ascending order of randomly generated arrival time
void customer_sort(customers customer[])
{
        int max_time_index;   //record the customer index with the latest arrival time
        customers max_time_cus,swap_cus;
        //selection sort
        for(int i=N-1; i>0; i--)
        {
        max_time_cus=customer[i];
        max_time_index=i;
        //locate the customer who arrives the latest
        for(int j=0; j<i; j++)
        {
        if((customer[j].arrive_time.h)*60+customer[j].arrive_time.m >
(max_time_cus.arrive_time.h)*60+max_time_cus.arrive_time.m)
        {
        max_time_cus=customer[j];
        max_time_index=j;
        }
        }
        if(i!=max_time_index)
        {
```

```
            //the swap part of selection sort
            swap_cus=customer[i];
            customer[i]=max_time_cus;
            customer[max_time_index]=swap_cus;
            }
            }
}
//for multiple-line cases, judge which queue has the fewest people waiting
int judge_queue_in_M(queue<customers> cus_queue[],customers &customer,int
each_queue_cus_number[],int index)
{
            //record waiting time of each window in an array
            int each_queue_wait_time[K];
            for(int i=0; i<K; i++)
            {
            //the waiting time of an individual depends on the ending service time of the previous customer in
its queue
            int wait_h=cus_queue[i].back().end_time.h-customer.arrive_time.h;
            int wait_m;
            if (wait_h == 0)
            wait_m=cus_queue[i].back().end_time.m-customer.arrive_time.m;
            else //wait_h > 0
            wait_m=cus_queue[i].back().end_time.m-customer.arrive_time.m + 60;
            each_queue_wait_time[i]=wait_h*60+wait_m;
            }
            //determine the queue with the fewest people waiting
            int min_cus_number_index=0;
            for(int j=1; j<K; j++)
            {
            if(cus_queue[j].size() < cus_queue[min_cus_number_index].size())
            min_cus_number_index=j;
            }
            //update data
            customer.wait_time.h=each_queue_wait_time[min_cus_number_index]/60;
            customer.wait_time.m=each_queue_wait_time[min_cus_number_index]%60;
            customer.start_time.h=cus_queue[min_cus_number_index].back().end_time.h;
            customer.start_time.m=cus_queue[min_cus_number_index].back().end_time.m;

customer.end_time.h=customer.start_time.h+(customer.start_time.m+customer.business_time)/60;
            customer.end_time.m=(customer.start_time.m+customer.business_time)%60;
            //push the customer in queue
            //if a customer's starting time (not necessarily arrival time) is later than the bank's closing time
            //then he/she would not join any queue or receive any service
```

```
	if((customer.start_time.h)*60+customer.start_time.m < 17*60)
	{
	cus_queue[min_cus_number_index].push(customer);
	each_queue_cus_number[min_cus_number_index]++;
	}
	return min_cus_number_index;
}
//for single-line cases, judge which queue has the least waiting time
int judge_queue_in_S(queue<customers> cus_queue[],customers &customer,int
each_queue_cus_number[],int index)
{
	//analogous procedure for single-line cases
	int each_queue_wait_time[K];
	for(int i=0; i<K; i++)
	{
	int wait_h=cus_queue[i].back().end_time.h-customer.arrive_time.h;
	each_queue_wait_time[i]=wait_h*60+cus_queue[i].back().end_time.m-customer.arrive_time.m;
	}
	//determine the queue with the least waiting time
	int min_time_queue_index=0;
	for(int j=1; j<K; j++)
	{
	if(each_queue_wait_time[j] < each_queue_wait_time[min_time_queue_index])
	min_time_queue_index=j;
	}
	customer.wait_time.h=each_queue_wait_time[min_time_queue_index]/60;
	customer.wait_time.m=each_queue_wait_time[min_time_queue_index]%60;
	customer.start_time.h=cus_queue[min_time_queue_index].back().end_time.h;
	customer.start_time.m=cus_queue[min_time_queue_index].back().end_time.m;

customer.end_time.h=customer.start_time.h+(customer.start_time.m+customer.business_time)/60;
	customer.end_time.m=(customer.start_time.m+customer.business_time)%60;
	if((customer.start_time.h)*60+customer.start_time.m < 17*60)
	{
	cus_queue[min_time_queue_index].push(customer);
	each_queue_cus_number[min_time_queue_index]++;
	}
	return min_time_queue_index;
}
//when the next customer arrives, determine whether those customers currently in queues have ended
//services and move out of queues accordingly
void leave_queue(queue<customers> cus_queue[],customers customer)
{
```

```
	for(int i=0; i<K; i++)
	{
	if(!cus_queue[i].empty())
	{
	while((cus_queue[i].front().start_time.h)*60+cus_queue[i].front().start_time.m+
		cus_queue[i].front().business_time <=
(customer.arrive_time.h)*60+customer.arrive_time.m)
	{
	cus_queue[i].pop();
	if(cus_queue[i].empty())
		break;
	}
	}
	}
}
//put customers in queues
void customers_in_queue(queue<customers> cus_queue[],customers customer[],int
each_queue_cus_number[], char label)
{
	int queue_number;//locate an empty window
	for(int i=0; i<N; i++)
	{
	bool queue_free=false;
	//move those in the front of queues out if necessary
	leave_queue(cus_queue,customer[i]);
	for(int j=0; j<K; j++)
	{
	//when there are available windows
	if(cus_queue[j].empty())
	{
	//update data and join queues
	customer[i].wait_time.h=0;
	customer[i].wait_time.m=0;
	customer[i].start_time.h=customer[i].arrive_time.h;
	customer[i].start_time.m=customer[i].arrive_time.m;

customer[i].end_time.h=customer[i].start_time.h+(customer[i].start_time.m+customer[i].business_time)/6
0;
	customer[i].end_time.m=(customer[i].start_time.m+customer[i].business_time)%60;
	cus_queue[j].push(customer[i]);
	each_queue_cus_number[j]++;
	queue_free=true;
	break;
```

```
        }
        }
        //when there are no available windows
        if(queue_free==false)
        {
        if (label=='M')//multiple-line cases
        queue_number = judge_queue_in_M(cus_queue,customer[i],each_queue_cus_number,i);   //judge
which queues to join
        else
        queue_number = judge_queue_in_S(cus_queue,customer[i],each_queue_cus_number,i);
        }
        }
}
void output (customers customer[], int each_queue_cus_number[], ofstream &oufile)
{
        int sum_cus_wait_time=(customer[0].wait_time.h)*60+customer[0].wait_time.m;
        int actual_cus_numbers=0;
        for(int i=0; i<K; i++)
        actual_cus_numbers+=each_queue_cus_number[i];
        for(int i=1; i<actual_cus_numbers; i++)
        sum_cus_wait_time+=(customer[i].wait_time.h)*60+customer[i].wait_time.m;
        oufile << (double)sum_cus_wait_time/actual_cus_numbers << endl;
}
```

R

```
library(plotly)
setwd("~/Desktop/YEAR 4/summer/IT course timetable/project/simulation_final")

sraw <- read.delim("4_100_s.txt",skip=1,header=TRUE)
s <- data.matrix(sraw)
mraw <- read.delim("4_100_m.txt",skip=1,header=TRUE)
m <- data.matrix(mraw)

average_waiting_time <- c(s)
p <- plot_ly(y = ~average_waiting_time, type = "box",name = "single line") %>%
  add_trace(y = ~c(m),name="multiple lines")%>%
  layout(title = "Boxplot (4 windows, 70 clients)")
p

sefficient <- vector()
for (i in 1:100){
  if (s[i] < m[i]){
```

```
    sefficient[i] <- 1
  }else{
    sefficient[i] <- 0
  }
}
sbetter <- sum(sefficient)/100
mbetter <- 1-sbetter
```