Craig Lombardo

CS150 Data Structures and Algorithms

Professor Liew

Project 1 Writeup

3/13/15

1. Introduction:

One of the biggest problems that business owners have today is managing the customer demand they receive. This project focuses specifically on the restaurant industry and the difficult task they are set with of seating every, or most of the customers, they are presented with. More specifically managing the wait list, particularly at popular restaurants. This task is simple at first, i.e. when the restaurant initially opens and all of the tables are open; however, once those tables are filled, things become a little trickier. Managing hundreds and hundreds of customers is never easy; managing hundreds and hundreds of hungry customers is even harder. For the purposes of this project we are experimentally testing the task of seating arrangements for popular restaurants such as Ippudo in New York City. Ippudo is a perfect example, as the restaurant does not take reservations, thus making the task of efficient seating a necessity. Customers show up and put their name on a wait list and they are called when a table is available. The goal of this project is to come up with a seating strategy that will maximize revenue and minimize wait time. It is also important to ensure a correct wait time approximation. It is not fair for a group to be told they will be seated for 5 minutes and then wait for 5 hours, that is just poor business and there will be no returning customers. The goal of the project

is also to deal with object-oriented programming and construct a program that will treat customers as objects that can interact with the environment they are put in. For this project I made a few assumptions. The first being that the table structure that we were given is beneficial for the project we are trying to complete. I also had to assume, given I don't have anyone else's data to go based off of, that the seating strategies I made are the best (this includes seating and wait list handling). The important data to be collected in experimentation included the number of customers served, how many meals were given to those waiting past 2 am (4 free meals a person), how many people were still eating after 2 am, and how accurate my time estimate was. Customer arrival times, and group sizes varied. Small groups, groups of people from 1-4, arrived on average 20 seconds apart from each other (standard deviation of 10 seconds), staying on average 45 minutes (standard deviation of 10 minutes). Large groups, groups of people from 5-8, arrive on average 30 seconds apart from each other (standard deviation of 15 seconds), staying on average 60 minutes (standard deviation of 15 minutes). Upon arrival of new customers we have several seating options: five tables that held four people apiece, five booths holding eight people each, and an oval that can hold 30 people (continuously around) and a bar that seats 20 people, that gives us 110 seats. Should we not be able to seat a group together, we will put them on the waiting list and give them an estimated waiting time. Should they get a table they will be seated right away and given a wait time of 0 seconds; however, should their estimated wait time be longer than the restaurant is open then they will unfortunately be turned away as we are not going to stay open all night. With this all being said, we want to find out what seating strategy gives us the most money and the least amount of unhappy customers.

2. Approach:

For the purposes of this project I decided to break things up into multiple parts, sort of implementing a divide and conquer strategy so as to ensure that no one class was doing too much. There were two reasons behind this; the first being that I wanted to limit the potential for problems (although it does not change the project it makes the logic and debugging easier), the second being I wanted my program to run more like a machine and less like a piece of code. For the purposes of the approach I will be discussing the parent classes and the purpose of each of those classes, as all of their children extend their parent and thus inherit all logic that will be explained and any class that is neither a parent nor child. The first class is the Restaurant class; this class serves as the actual "restaurant" we are implementing and will be filled with varieties of seating options and customers. The restaurant will keep track of things like the total customer count, the seating and waiting list preference (which when varied supply me with different strategies), lists that indicate who is on what list. Most importantly, the restaurant brings all of the other classes together as a sort of bridge between the classes. I realized post submission of my lab that I did not hide enough of the implementation from the user. I have unintentionally put a fair amount of code in the restaurant class rather than completely allocating these tasks to outside classes. I have also come across the mistake of dropping out my third seating scenario; it is coded up however I neglected to implement it in the code so seating option 3 essentially does not work. To fix this one line of code is needed however given I cannot fix it and resubmit it I just wanted to provide proper notice in regards to this blunder. The seating options are as follows, scenario 1, in

which small groups look to be seated at a table first, then the bar, and finally the oval. Large groups look to be seated at a booth first, then the oval and then finally the bar. We have scenario 2, in which small groups look to be seated at the bar and then the oval and then finally a table. Large groups look to be seated at the oval and then the bar, and then finally the booths. We also have two seating options in which groups are either taken when available (option 1), or precedence is given to larger groups as they are harder to seat (option 2). The next class is Seating, this class is the parent class for all seating types (i.e. bar, oval, booth, table) and handles customer seating, or attempted customer seating. The children of this class each control their own seating, each keeping track of the number of seats available and which seating positions are available for a given group type. The next class is the Customer class; this class is another parent class. When instantiated, as either a SmallGroup or LargeGroup, each group is assigned a random size (1-4 for small groups, 5-8 for large groups) and the type of group is tracked for later use in seating implementation. Each group also keeps track of pertinent information such as how many people are in the group, what table they are sitting at, what time they arrived/the wait time they were told (if applicable). Past this, the Customer class does not have much more functionality, it is intended to be like an actual group, keeping track of what it's told; however, it doesn't control implementation, it merely changes how the implementation of the restaurant is handled. There is a class called TimedEvents; which acts like the clock/hostess of the restaurant. The clock notifies the restaurant when a new group has arrived, small or large, and checks to see when a group has finished eating their meal/ready to leave. From here customers are either added or removed from the waiting list accordingly as well as seats the customers when the time is appropriate. There are

also two random number generator classes, one is a RandomGaussian and the other is the Wheel. The RandomGaussian number generator generates random arrival and stay times for the two groups, based on a normal Gaussian distribution. The Wheel class generates random numbers, which will be used to give the random group sizes used. For all lists used, I used ArrayLists, even at the oval table; although there is not normally continuity I ensured that there will be a wrap around. There will be ArrayLists that will contain Customers i.e. waiting lists and seated lists, and ArrayLists that contain Integers such as available seating indexes.

3. Methods:

For the purposes of this project I decided to run things 10-15 times for a concrete result/ average. The data collected included how many customers were served, how many meals were given to those waiting past 2 am (4 free meals a person), how many people were still eating after 2 am, how many customers of each type came to the store front, and how accurate my time estimate was. Given the breakdown of five tables that held four people apiece, five booths holding eight people each, and oval that can hold 30 people (continuously around) and a bar that seats 20 people, that gave me 110 seats to work with. Small groups arrived on average 20 seconds apart from each other (standard deviation of 10 seconds), staying on average 45 minutes (standard deviation of 10 minutes). Large groups arrived on average 30 seconds apart from each other (standard deviation of 15 seconds), staying on average 60 minutest (standard deviation of 15 minutes). It is obvious to see that the waiting list quickly jammed up and this is why it is so important to create a good simulation. With this given data I created different

scenarios, these scenarios ranged from table preferences for the two groups and seating strategies. Each strategy was tested and revamped several times before deciding whether or not it was a valid option. As mentioned in the prior section, a line of seating option code was omitted which cuts its functionality from working in the overall project. Aside from that there are still four different options for customer handling, two options for seating and two options for wait list handling; these were used for my experimental data. For the testing process a simple for loop was used and within the for loop was functionality for my TimedEvents class. Each iteration of this loop was considered to be one second, starting from time 0 and continuing until the restaurant closed at some later specified second value; for my tests I used 28,800 seconds, or 8 hours as the restaurant we are modeling is open from 6pm to 2am. I felt that this would be the best as with this setup I would not need to worry about fractional time or anything like this, simply using uptime in seconds will guarantee I will not have any customer handling issues/any time issues.

4. Data and Analysis:

For the analysis of my data I have compared my results to the mathematical expectations of the simulation. Given we have 110 seats, if we assume there is an average of 4-5 customers per group that means we can get on average (ideal conditions) 22 groups seated. If large groups take on average 45 minutes, and large groups take on average 60 minutes, that means the average group eating time (for both) is 52.5 minutes. If the restaurant is open for a stretch of 8 hours, or 480 minutes, that means that in ideal conditions we can have approximately 9 total group rotations. At the end of the night, the

group rotations equate to an average of about 201 groups, or 804-1005 customers; for arguments sake I will say 900 customers, where each customer gives me a net profit of 1 meal. As seen by figure 1 on the next page, unfortunately none of the seating options gave me quite maximum profit. The expected profit is shown to be higher than the calculated value expected. The bars for seating options take payout into consideration as well, as customers who were promised a table but were not provided one, received four free meals each. We can see by the graph that seating option 1 waiting option 2 is the best strategy of the ones I have created. This is only marginally the case as seating option 2 waiting option 1 is virtually the same net gain; in fact the meal difference is only 4 meals.

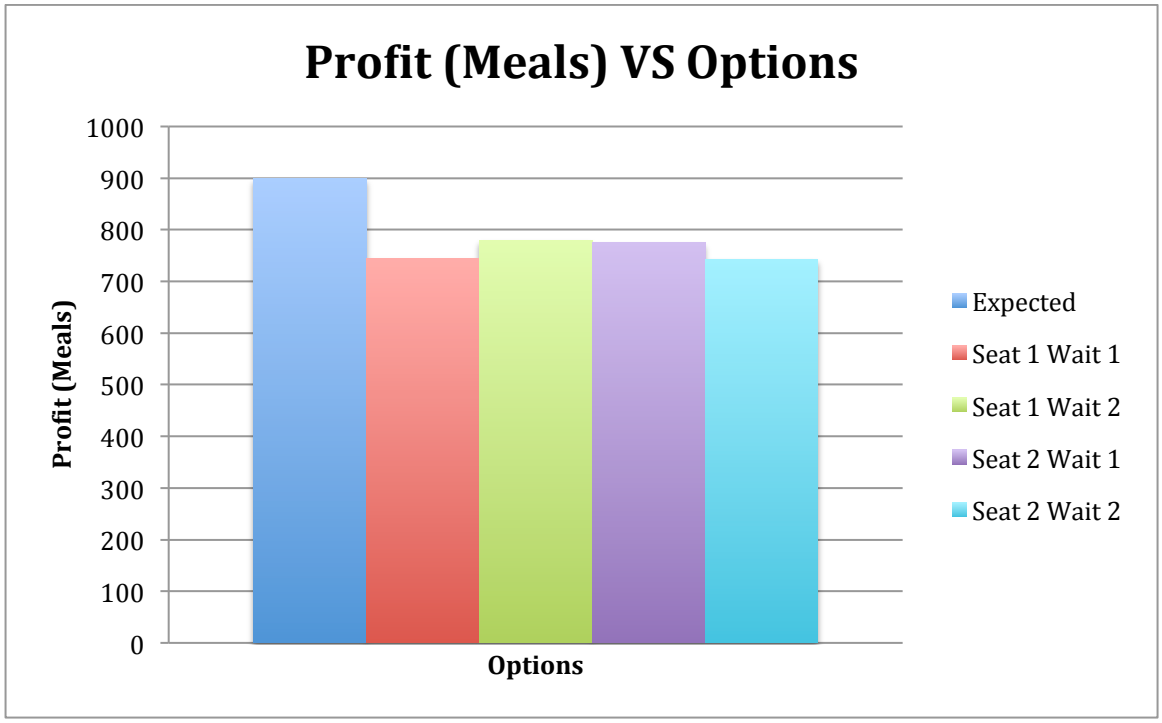

Figure 1

This graph shows the different payouts for the varied seating and waiting list options, the none Expected bars also take payout into account.

The average wait time cannot be generalized, as the wait time has to do entirely with arrival time and group size. In an ideal situation, we can perfectly estimate wait time, and successfully seat all customers who arrive (with a wait time prior to closing time); however, we know this is not the case, and that is why we rely on experimental data. Through experimentation I have realized that my model is not ideal; however, the results are still important. Given my experimental data, it can be seen that seating option 2 paired with seating option 2 yields the lowest average estimated wait time, this can be seen by the graph below in figure 2.

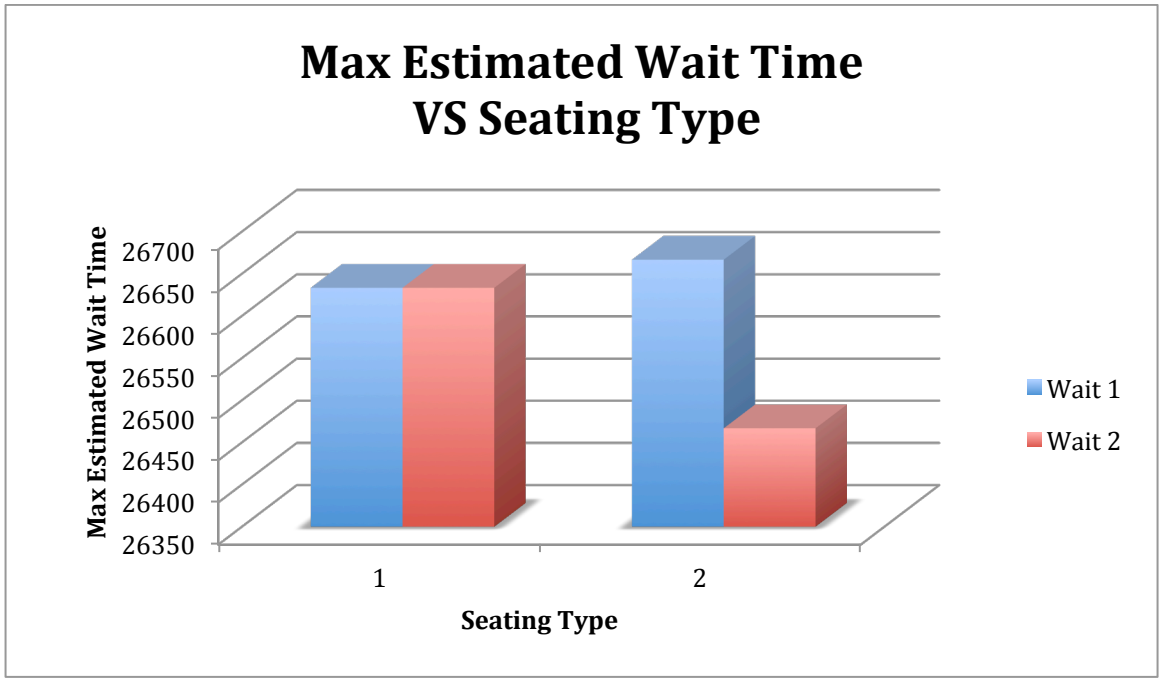

Figure 2

This chart shows a comparison between the different wait time estimates for the various seating options.

Although this was the lowest estimated wait time, this strategy had the highest actual wait time. The shortest actual wait time strategy was seating option 1 wait option 2, this can be seen by the Figure 3 below. We can also see that waiting option 1 was the best for the more accurate time estimates. Figure 4 on the next page shows the comparison between the estimates. This graph serves to show the accuracy to the actual wait time. Seating option 1 wait option 2 had the highest average estimated wait time and seating option 2 wait option 1 had the lowest average; however, as we can see from our actual calculations, seating option 2 wait option 2 had the lowest average actual wait time, which is usually what is most important. We can see these relationships in figures 5 and 6 respectively on the next two pages.

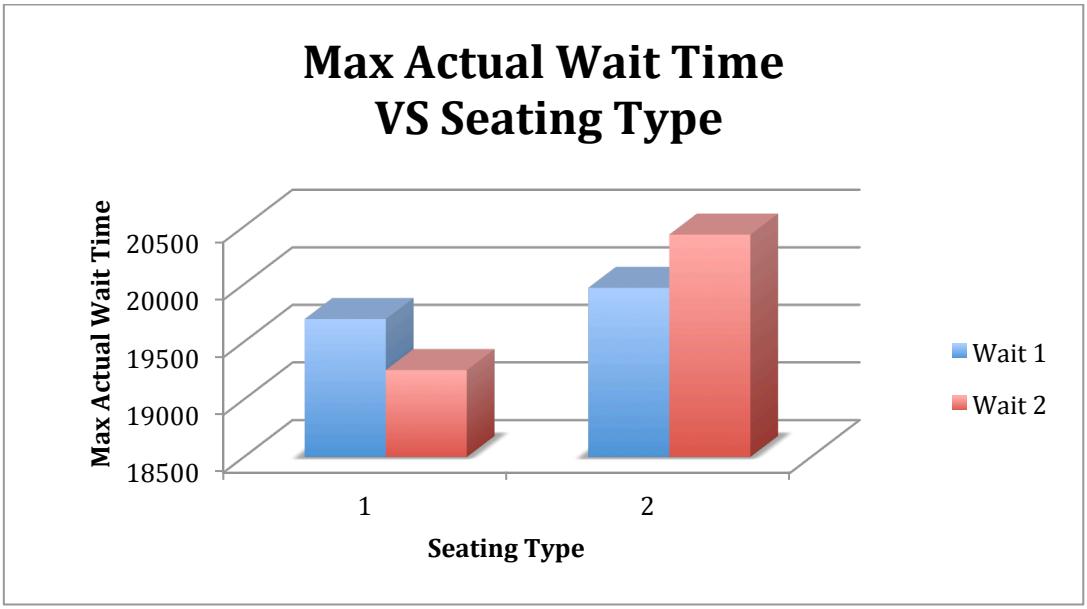

Figure 3

This chart shows a comparison between the different actual wait times for the various seating options.

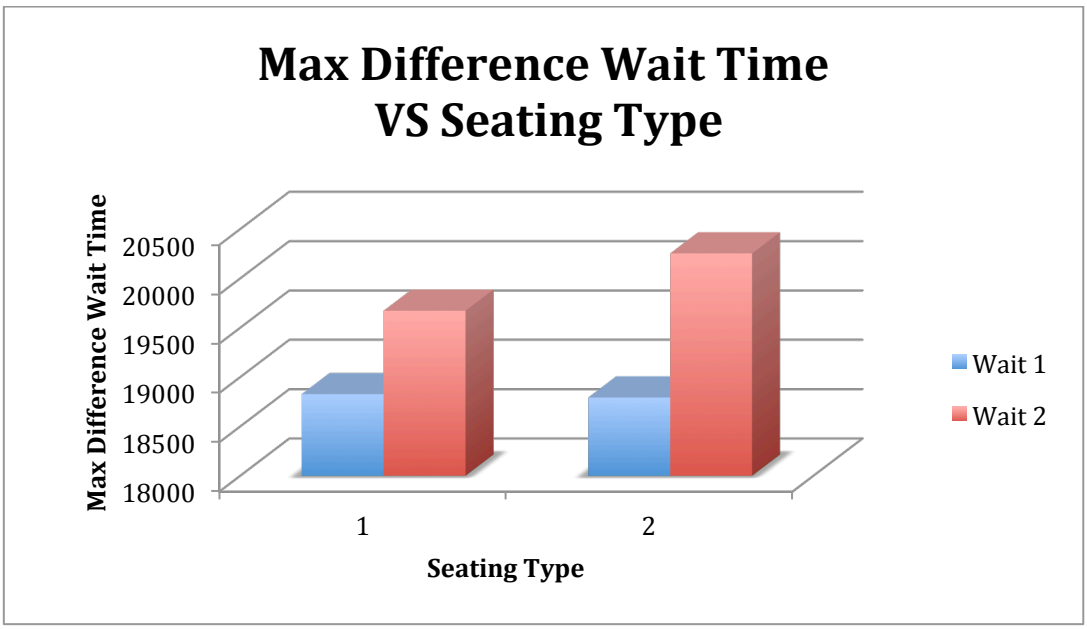

Figure 4

This chart shows a comparison between the time estimates for the various seating options, this shows accuracy of estimate vs actual.

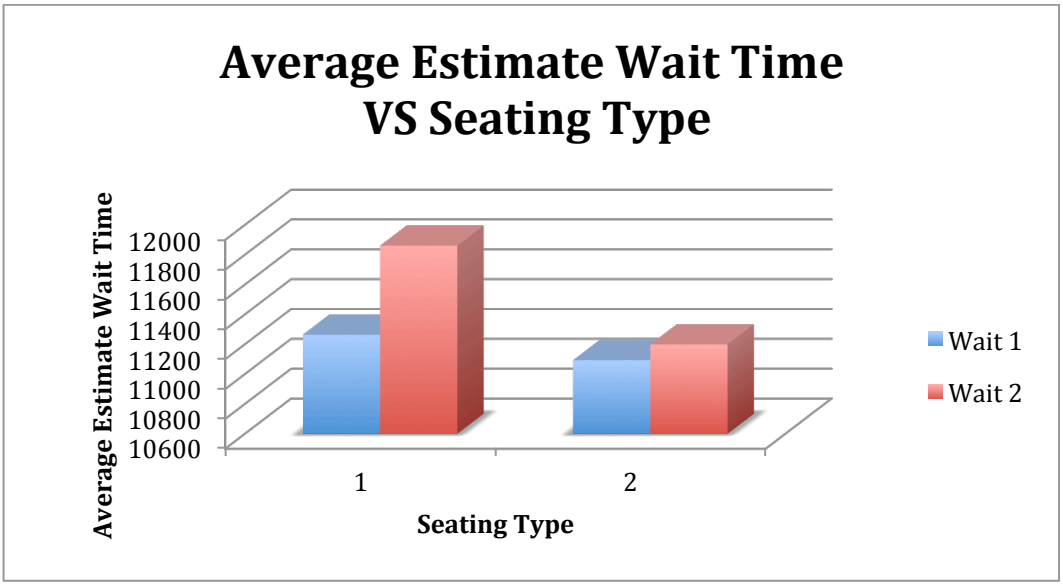

Figure 5

This chart shows a comparison between the average time estimates for the various seating options, this shows overall performance.

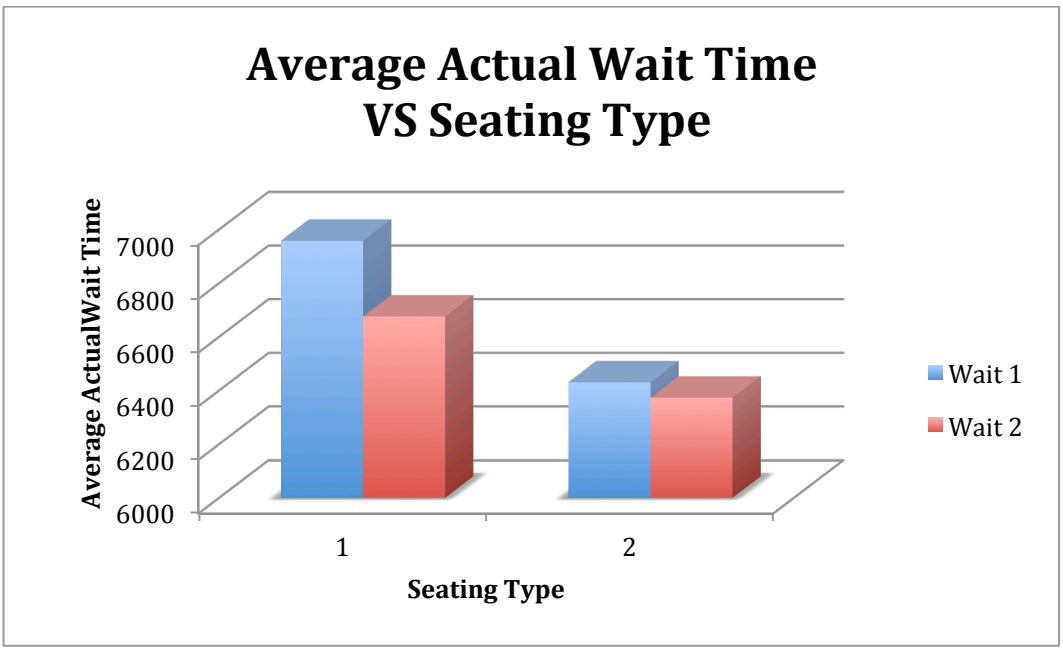

Figure 6

This chart shows a comparison between the actual wait times for the various seating options, this shows overall performance.

5. Conclusion

Based on this data it is somewhat tough to say which strategy is the best. It all comes down to the compromise of would I rather get more profit, or more happy customers; however, I feel I may have come up with a winner. It appears to me that seating option 1 waiting option 2 will get me the most profit and keep people the happy at the same time. As seen by the graphs above, the performance of this option is very good. I may not reach theoretical yield; however, the data collected is on the lower end, if not the lowest for all averages displayed which means more people get fed and I get more money, win win. Although there is not a graph to display this information, the average payout to customers still waiting was the second lowest, and only one meal more than the

lowest payout. It may not be serving thousands of groups in a night; however, given the parameters of the restaurant we were given, I am fairly confident that the strategies I have put forth are reasonably sound. Although there is no one perfect seating strategy for every restaurant I feel as though the program I have constructed allows for flexibility to be reused for a different restaurant with different table types and through adaptation of the seating strategies it may be used to test functionality and efficiency. Given the nature of the program, there is not a concrete time as to when I turn people away, I am not saying every night I will close the doors to new customers at 10 pm because the wait time is based entirely on the size of the waiting list and the size of the group trying to sit down. As more customers are on the waiting list the overall wait time for new customers should increase; however, given the exact same flow of customers is not guaranteed, we cannot generalize the door closing time.

6. References:

CS 150: Guidelines For Writing Lab and Project Reports:

https://moodle.lafayette.edu/pluginfile.php/151579/mod_resource/content/1/write up-guidelines.pdf

CS 150: Project I - Restaurant Management Project Description:

https://moodle.lafayette.edu/pluginfile.php/154315/mod_resource/content/3/p1.pdf

"Java Platform SE 7." Java Platform SE 7. N.p., n.d. Web. 12 Mar. 2015.

<http://docs.oracle.com/javase/7/docs/api/>.

"Java Practices - Generate Random Numbers." Java Practices - Generate Random Numbers. N.p., n.d. Web. 13 Mar. 2015. <http://www.javapractices.com/topic/TopicAction.do?Id=62>.

Weiss, Mark Allen. *Data Structures & Problem Solving Using Java*. Boston: Pearson/Addison Wesley, 2010. Print. Fourth Edition.