INTRODUCTION

The aim of this project was to develop a structure of data containers for efficient addition and access of data items. The project was an extension of Project 1[1] to allow customer data to be stored in these containers and provide access to the data given certain user input constraints. This program would aim to improve targeted advertising in towards customers with specific needs who reside in specific cities. The development goals were focused around building an efficient container set to allow for specific user input queries. Efficiently processing and responding to each of the user input commands was focused upon. The analysis goals included analyzing the response times for the program while responding different types of commands under different constraints.

Certain assumptions were made[2] to simplify the task at hand and allow the programmer to focus on the key requirements and areas of importance. Some of them were as follows-

- The user inputs to the program would be made through a terminal based interface.
- The inputs to the terminal would follow a specific format to allow the client to perform various operations associated with the database.
- The cities for each customer were to be randomly generated along a uniform distribution from a given set of cities that can be modified by the client.
- The customer generation times for a simulation run and the individual customer needs would be determined based on values from different Gaussian distributions. They are described as follows:

Customer generation times-

Mean	Standard Deviation
61 seconds	31 seconds

Percentage of customers who would need types of items-

	Mean	Standard Deviation
Baked	37%	17%
Meat	53%	13%
Dairy	59%	19%
Fruit	47%	13%
Vegetable	71%	29%
Beverage	43%	11%

It was hypothesized that as more customers would be added to the database, the time taken for most operations would increase. As the number of simulation runs would increase, the time taken for 'run' would increase linearly. For 'list' a linear complexity was also expected as the size of the database

[1] Liew, Chun W. "CS 150: Project 1 - Managing A Farmer's Market." (n.d.): n. pag. 23 Sept. 2015. Web. 25 Sept. 2015. <https://moodle.lafayette.edu/pluginfile.php/189193/mod_resource/content/5/p1.pdf>.

[2] Liew, Chun W. "CS 150: Project 2 Description." (n.d.): n. pag. 27 Oct. 2015. Web. 29 Oct. 2015. <https://moodle.lafayette.edu/pluginfile.php/194458/mod_resource/content/3/p2.pdf>.

increased, more customers would need to be accessed and listed. For 'findgoods' a near constant-time probability would be expected because the operation uses direct pointer access to obtain necessary values. The number of items being accessed is constant irrespective of the database size. For 'findcities' a probability between linear time and quadratic time would be expected depending on the parameters input. For 'findcities' a method needs to be run for every city based on parameters to obtain the requisite values for ordering cities. The time taken for the this command would be expected to run between linear time and quadratic time as it depends on both the number of cities and the number of customers for each item in the city.

APPROACH

The developed solution was divided into several classes, each serving a different functionality-

- Customer class- This class was used to generate Customer objects with a randomly assigned home city and set of needs. Provides functionality to query customer needs.
- City class – This class was used to generate city objects.
- Item class – This class was used to generate item objects.
- CityContainer class – This class was used to store multiple city objects to be conveniently passed as parameters to/from methods.
- ItemContainer class – This class was used to store multiple item objects to be conveniently passed as parameters to/from methods.
- ItemNeedGenerator class – This class was used to randomly generate random needs for each type of item based on the fixed constraints.
- CityGenerator class – This class was used to randomly generate random cities for customers based on the client provided list of cities.
- TopLevelContainer class – This class was used as the top level container with a HashMap[3] as the data structure for the container. The keys for each entry in the hash map were city names and the values were CityHolder objects that acted as sub containers. This class provides functionality to output the data according to user input commands and constraints.
- CityHolder class – This class was used as a secondary container with a HashMap as the data structure for the container. The keys for each entry in the hash map were item types and the values were ItemHolder objects that acted as sub containers.
- ItemHolder class – This class was used as a tertiary container with a LinkedList[3] as the data structure for the container. The actual Customer objects were stored in the linked lists.
- CityHolderComparator class – This class was used as a comparator to compare CityHolder objects based on the number of customers in each container who needed a specific set of items.
- ItemHolderComparator class – This class was used as a comparator to compare ItemHolder objects based on the number of customers in each container.
- Simulation class – This class was used to actually run the market simulation to randomly generate customers and add them to the existing top level container.

[3] *Java Platform SE 8 API Specification*. Oracle, n.d. Web. 12 Nov. 2015. <https://docs.oracle.com/javase/8/docs/api/>.

- ExperimentController class – This class was used to store the top level container object and set of cities for every instance of the program. It provides functionality to link operation commands between the Interface and the top level container.
- Interface class – This class provides functionality to accept user input terminal commands and parse constraints for further processing.
- GaussianGenerator class – This class provides functionality to generate random numbers in a Gaussian distribution with a given mean and standard deviation.[4]
- RandomGenerator class – This class provides functionality to generate random numbers within a given range of bounds.[4]

A combination of 2 Hash Maps and 1 Linked List were used as the main underlying container structure to provide fast access to customers based on their city of residence and items purchased. For the 'findgoods' and 'findcities' operations TreeSet[5] objects were used to provide ordered access to the required data (based on user input parameters) obtained from the main containers.

METHODS

The main parameter used for the experiments was the size of the database determined by the number of times the simulation was run. Additionally for parameter based commands, two cities and two types of items were used. The following fixed parameters were used for all trials-

1. 'list'
    a. item: fruits
    b. item: vegetables
    c. city: Easton
    d. city: Allentown
2. 'findgoods'
    a. Easton
    b. Allentown
3. 'findcities'
    a. fruits
    b. vegetables

As described in the introduction, the statistical parameters to determine customer generation and city generation were set as fixed constraints.

The experiments were run by varying the parameter sequentially. A total of five different database sizes were tested. Every successive parameter size was increased by 5-10 times the current size.

The five configurations tested are listed as follows-

1. 1 run (approx. 200 customers)

---

[4] *Java Practices - Generate Random Numbers*. Hirondelle Systems, n.d. Web. 9 Oct. 2015. <http://www.javapractices.com/topic/TopicAction.do?Id=62>.

[5] *Java Platform SE 8 API Specification*. Oracle, n.d. Web. 12 Nov. 2015. <https://docs.oracle.com/javase/8/docs/api/>.

2. 10 runs (approx. 2,000 customers)
3. 100 runs (approx. 20,000 customers)
4. 500 runs (approx. 100,000 customers)
5. 1000 runs (approx. 200,000 customers)

For every configuration the performance of four commands was tested. The commands tested are listed as follows-

1. 'run'
2. 'list'
3. 'findgoods'
4. 'findcities'

Each experiment configuration was run 10 times to obtain **average values.** For each configuration a comparative analysis of timings for different commands was performed. An overall comparison of timings for each of those operations over each configuration was also performed.

DATA & ANALYSIS

The data obtained has been described and analyzed as follows-

1) <u>1 run-</u>

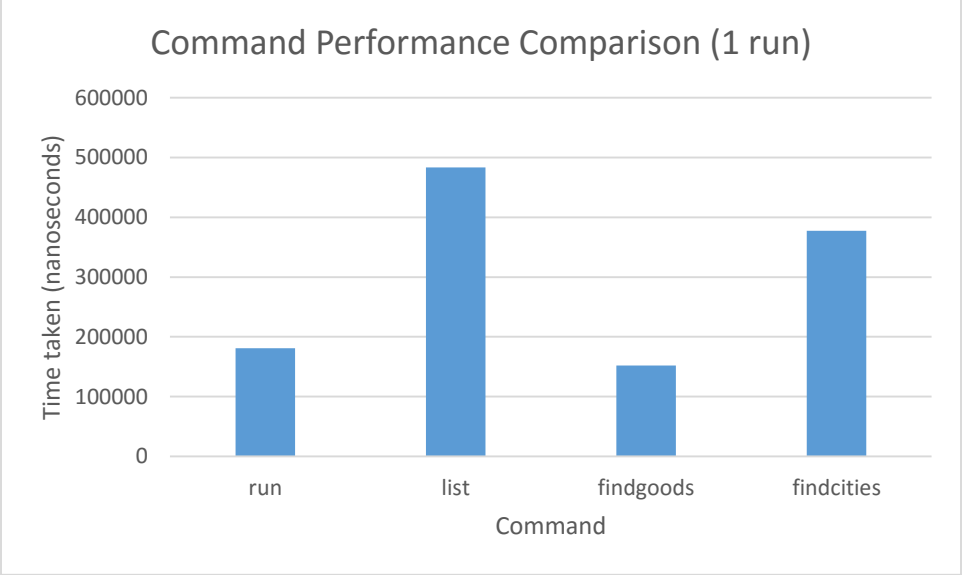

Figure 1

2) <u>10 runs-</u>

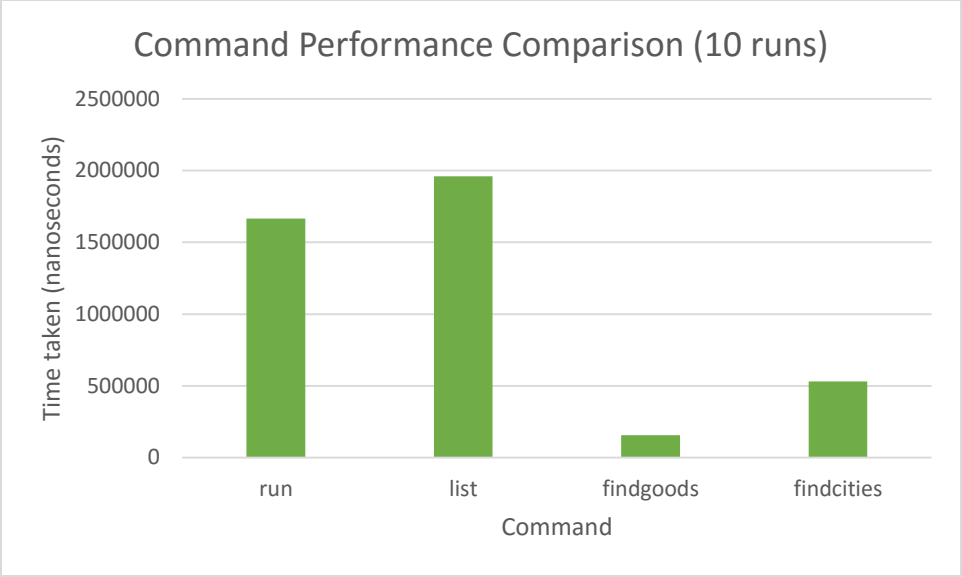

Figure 2

3) <u>100 runs-</u>

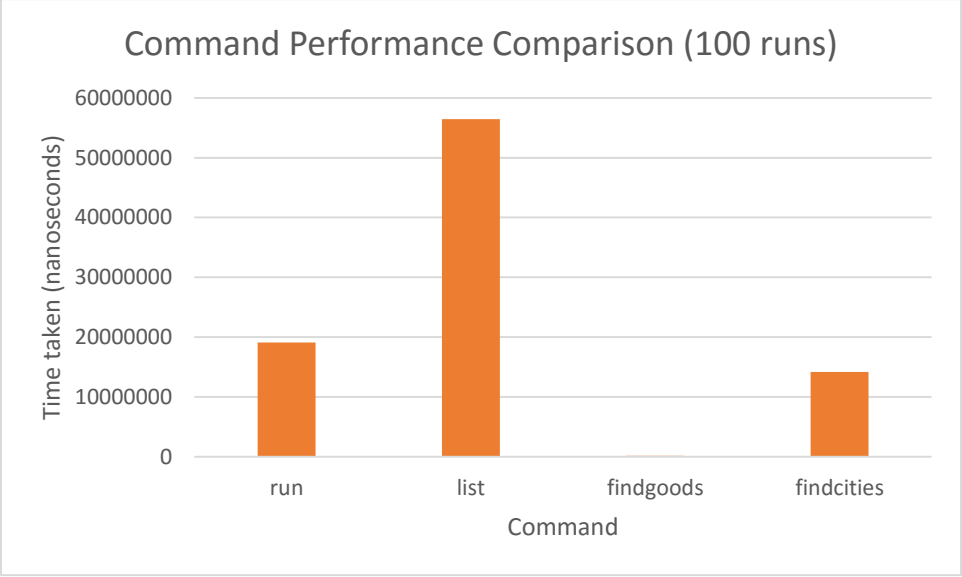

Figure 3

4) <u>500 runs-</u>

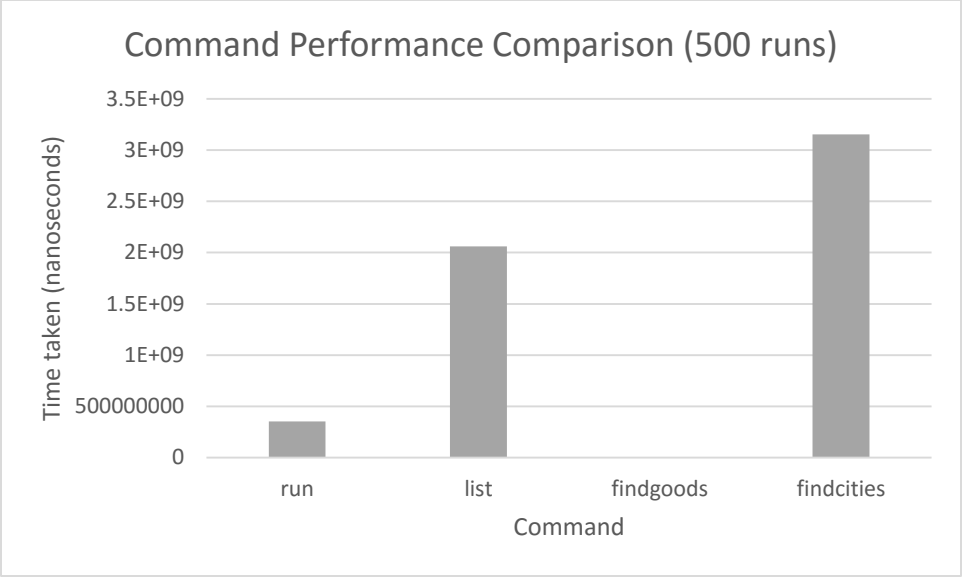

Figure 4

5) <u>1000 runs-</u>

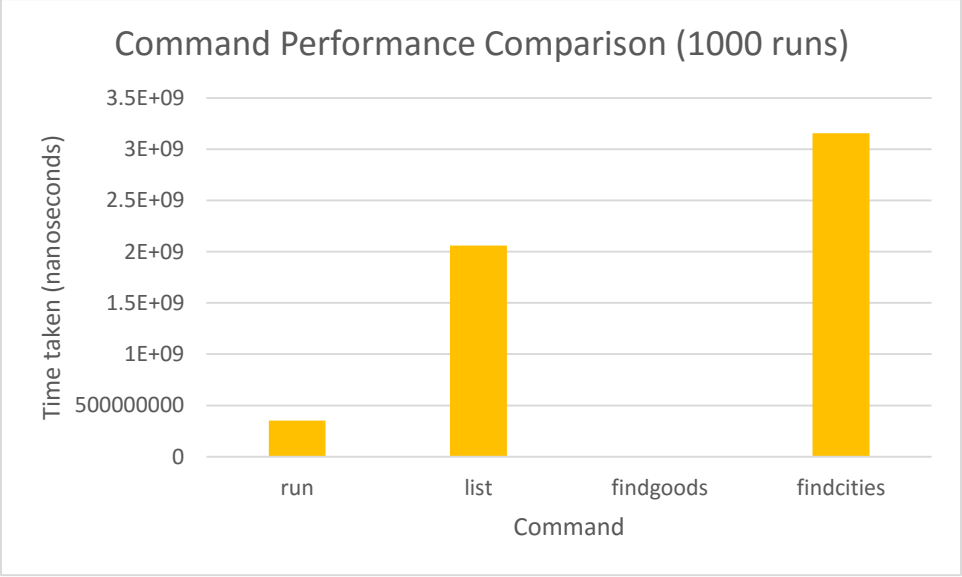

Figure 5

<u>Overall statistics-</u>

A comparison of the timing values obtained from each of the configurations has been presented as follows-

- Timing comparison for ‘run’ command-

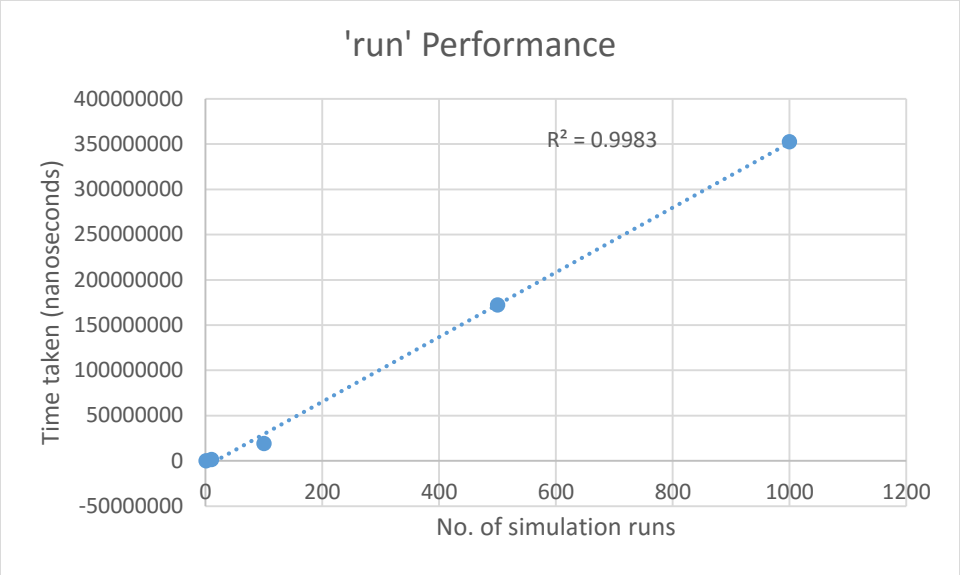

Figure 6

The run command operates in O(n) time.

- Timing comparison for 'list' command-

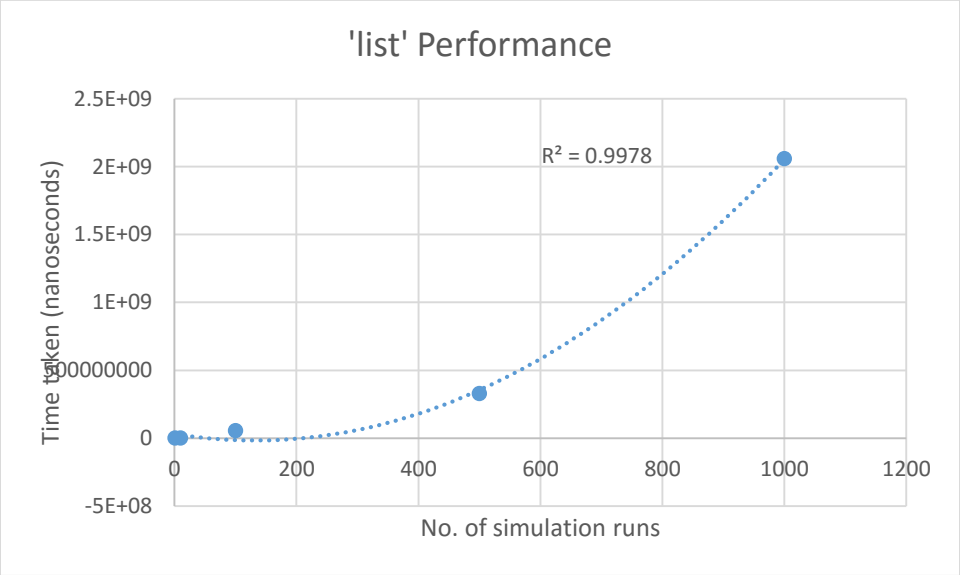

Figure 7

The list command operates in $O(n^2)$ time.

- Timing comparison for 'findgoods' command-

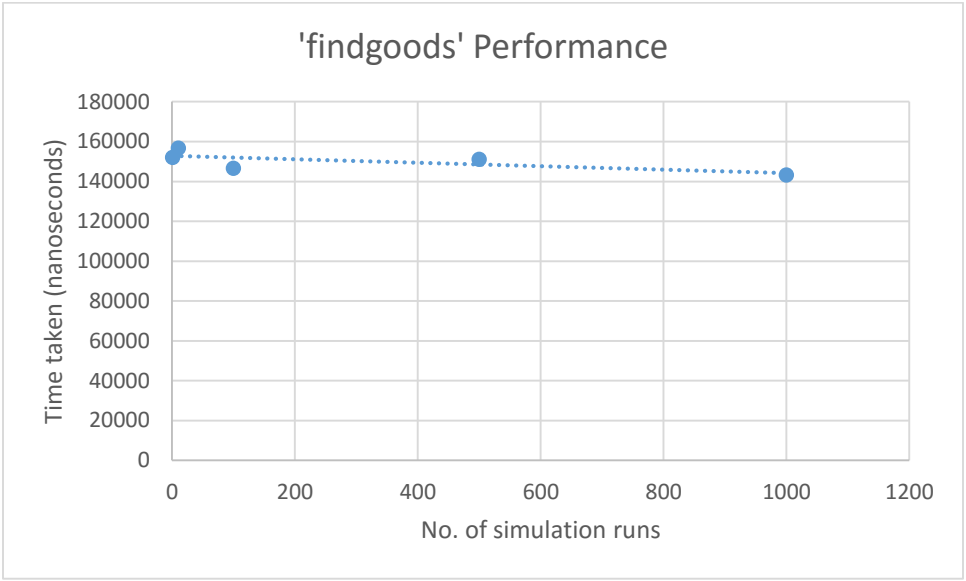

Figure 8

The findgoods command operates in O(1) time.

- Timing comparison for 'findcities' command-

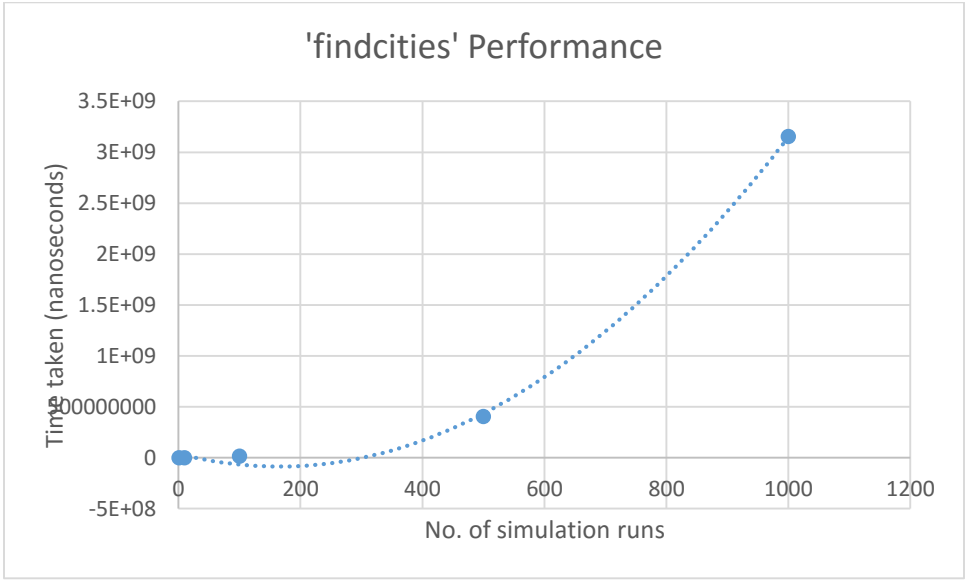

Figure 9

The findcities command operates in $O(n^2)$ time.

CONCLUSION

The comparative performance analysis of the different experiment configurations provided a good idea of performance of the database operations as the size of the database increased.

As hypothesized, we can confirm from Figure 6 that the 'run' command runs in O(n) time as the number of simulation runs increases. Even the 'findgoods' command runs as expected with a near O(1) complexity, as can be seen from Figure 8. Due to such performance, the 'findgoods' command appears to run the fastest amongst the tested operations.

The other two commands, however, do not perform exactly as hypothesized. Both the 'list' (Figure 7) and 'findcities' (Figure 9) operations seem to run in $O(n^2)$ time. A possible explanation for such performance is that a linear increase in customers results in a super-linear increase in the demand for certain goods. Considering the fixed testing parameters for items, 'fruits' and 'vegetables', we can conclude that the total proportion of customers purchasing this combination of goods increases as the customer sampling size increases. This behavior is influenced directly by the fixed constraints governing the percentage of customers who need to purchase different types of items.

While this is the case, if we consider the actual practicality of the runtime operations we notice that the longest 'list' operation takes 2 seconds and the longest 'findcities' operation takes 3 seconds (both for 1000 simulation runs) In practice, thus, the use such a database structure would not seem to be too inefficient for these operations, at least while the size of the database does not increase our testing maximum (approx. 200,000 customers)

REFERENCES

Weiss, Mark Allen. *Data Structures and Problem Solving Using Java*. Boston, MA: Pearson/Addison-Wesley, 2010. Print.

Java Practices - Generate Random Numbers. Hirondelle Systems, n.d. Web. 9 Oct. 2015. <http://www.javapractices.com/topic/TopicAction.do?Id=62>.

Java Platform SE 8 API Specification. Oracle, n.d. Web. 12 Nov. 2015. <https://docs.oracle.com/javase/8/docs/api/>.

Liew, Chun W. "CS 150: Project 2 Description." (n.d.): n. pag. 27 Oct. 2015. Web. 29 Oct. 2015. <https://moodle.lafayette.edu/pluginfile.php/194458/mod_resource/content/3/p2.pdf>.

Liew, Chun W. "CS 150: Project 1 - Managing A Farmer's Market." (n.d.): n. pag. 23 Sept. 2015. Web. 25 Sept. 2015. <https://moodle.lafayette.edu/pluginfile.php/189193/mod_resource/content/5/p1.pdf>.