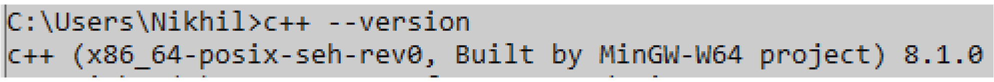
Final Project COMP3600– u5834991

Program Specs
Operating System: Windows 10
Language: C++



```
C:\Users\Nikhil>c++ --version
c++ (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0
```

Idea is similar to the first, delivery company, example provided in Piazza post @278.

Program Description

Scenario

A delivery company is given requests in the form of a list of items it needs delivered. The software used by this company manipulates that assortment of items into a usable format, providing information of delivery order and estimated delivery time to different locations. The program also allows manual entry of deliveries, lookup and deletion of orders for the delivery personnel to use.

On running the program (Can skip this and read Readme)

The program will begin by reading from a text file that will contain a 'delivery order' as well as a text file containing an adjacency matrix which will be used for Dijkstra's algorithm. The delivery order is an unsorted assortment of items meant to be delivered at various places on various times. The text file will be formatted as such:

```
Number of lines
Item1 DeliveryDate DeliveryLocation
Item2 DeliveryDate DeliveryLocation
...
```

The item name may be an ID or name and the delivery location must be a valid location.

The program will read this in and sort it by delivery date which shall be an integer in the format of (YYYYMMDD), into a list. This is to determine the delivery order or priority. We must assign which delivery run the item will be placed on, given that there is a limit to the number of parcels one truck can carry. For example, we may say that only the first 20 deliveries in this sorted list can be placed onto the first delivery run, then the next 20 on the next delivery run and so on. Hence, we assign each item a delivery run number, 1 meaning it will go out on the next run.

The program will then use a hashing function to store this information into a hashmap/hashtable using a hashing function that is *based on the item name/ID and delivery location* for fast lookup for the user as described in the following section.

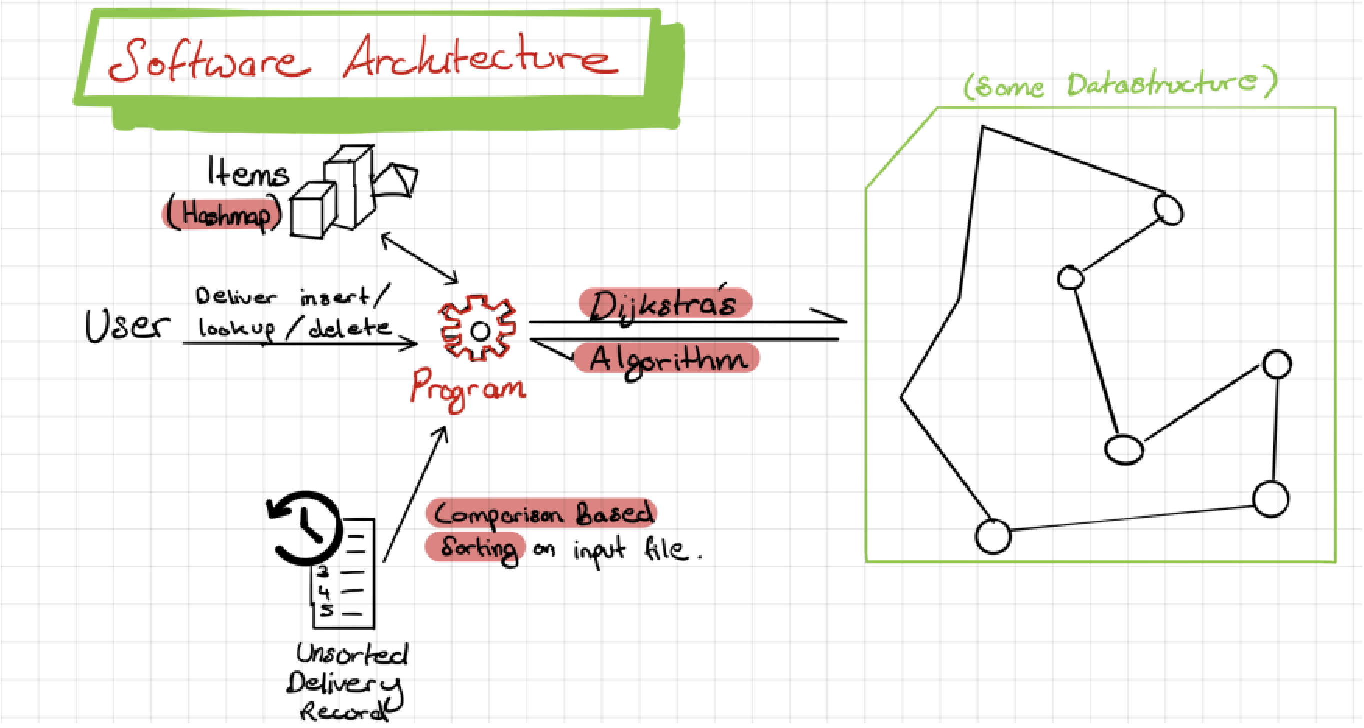
Now, furthermore the application will have a basic command line interface, which will:

(1) *Allow the user to input a set of new deliveries*, essentially storing their item along with delivery location and delivery date. The user may keep entering deliveries.
(2) Allow the user to search for an item to which the program provides the user with information on said item if the item exists, whether it has been delivered or not, along with estimated delivery time - explained in (4).
(3) Allow the user to delete an item from the delivery essentially cancelling the delivery database.
(4) Displays the list of deliveries in order of date from earliest to latest. Using a sorting algorithm.
(5) Provides the user with shortest distance to every location from a given source (which the user can input). So if the users at vertex 2, it will show the user the shortest path to every other vertex.
(6) The user will be able to exit out of the program

Closing Program

Once the user opts to close/exit the program it will store the resultant current delivery information back into a text with similar syntax to the input one, so that it can once again be opened.

The architecture is described in Figure 1 below, where the selected functionalities are highlighted in red.



Key Functionalities

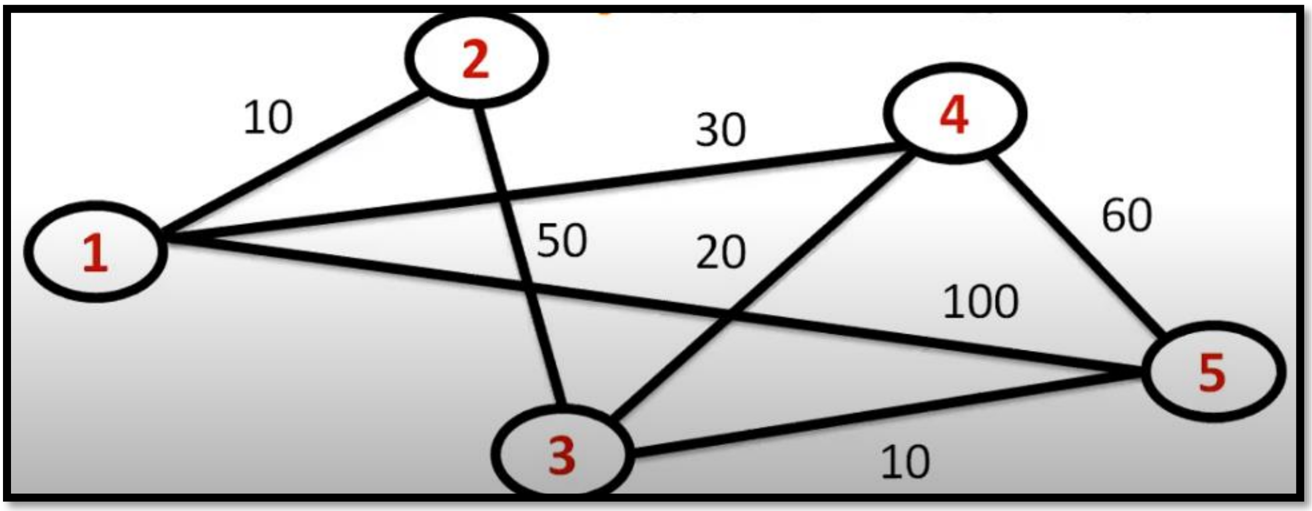
- Sorting a list of items by delivery date, using comparison-based sorting algorithm, specifically merge sort (B.1.) for the delivery to then be carried out in order. The program will take inputs from a delivery data text file. The user may want to see a list of all current deliveries in chronological order from earliest to be delivered to latest.
- Insertion, lookup, deletion of delivery items in a HashMap or hashed dataset, using hashing (B.7.) In order to store the item in a database, using a class 'item' we can store any unlimited amount of additional information and attributes to that item. Hashing allows an employee or person using the system to quickly lookup an item with only knowledge of its name/ID and delivery location. For example, this sort of system could be used by users that want to know when their item will be delivered, and they know their own location (address) and the name of the item they ordered. *Note: I have not added a lot of functionality to the item class as it was out of scope for this assignment.*
- Using Dijkstra's algorithm (B.9.) to provide a user with the shortest distance to all points given their location vertex. This information could be useful from the delivery company side to determine where to go first as it may not be apparent at first from a GPS or map due to other conditions affecting 'edge weight' . This information could also be potentially useful to estimate delivery time for the customer, although this could be a further implementation.

Assumptions

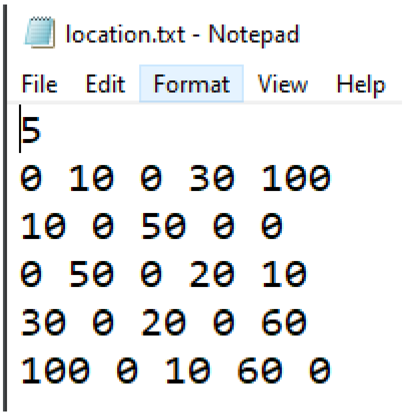
- Locations are restricted to a certain number of nodes; the delivery locations will be imaginary locations and the number of them and connecting locations will be pre-determined arbitrarily in some form. Some data structure will be used to define storing of these location and adjacency values.
- The user will input the item name, delivery date/priority, the sending location and the destination location as well as their username or a form of identification.
- The input file will contain at least 1 delivery order.
- There cannot be duplicate items of the same name/ID being sent to the same locations.
- The inputs must be well formed in order for the program to work correctly, but generally it
- We set a maximum input of 10 000 items, which is a rather large number for a local delivery company. I would expect that a normal list of deliveries would be needed for the coming week or fortnight and would contain maybe 1000 – 2000 items in the upper range for a given driver. *For the sake of time complexity analysis, I included large randomly generated (messy) data to test the run time of certain operations empirically but for the typical function of the program we would expect a smaller input.*
- The number of vertices or locations works up to 10 000 but takes a SIGNIFICANT amount of time. The typical limit for deliver locations would be around. According to Wired.com the average UPS driver makes 120 deliveries a day. We can say that in a fortnight that would be 120*10 working days = 1200 locations or 1200 vertices for our adjacency matrix for Dijkstra's algorithm. *(https://www.wired.com/2013/06/ups-astronomical-math/#:~:text=down%20to%20math.-,But%20some%20companies%20have%20tougher%20equations%20to%20solve%20than%20others,119%20*%20118%20*%20.%20.%20.)*

Functionalities in Action

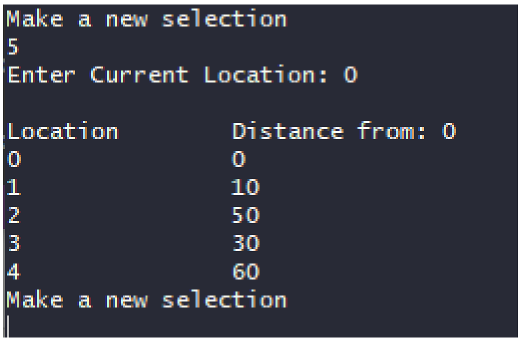
Dijkstra’s Algorithm was tested using example of graphs found online, one such example is from this youtube video, which shows the correct answer and it is the adjacency matrix in the file ‘location.txt’ https://www.youtube.com/watch?v=3C36O9gBMvI



Adjacency Matrix



```
5
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
```



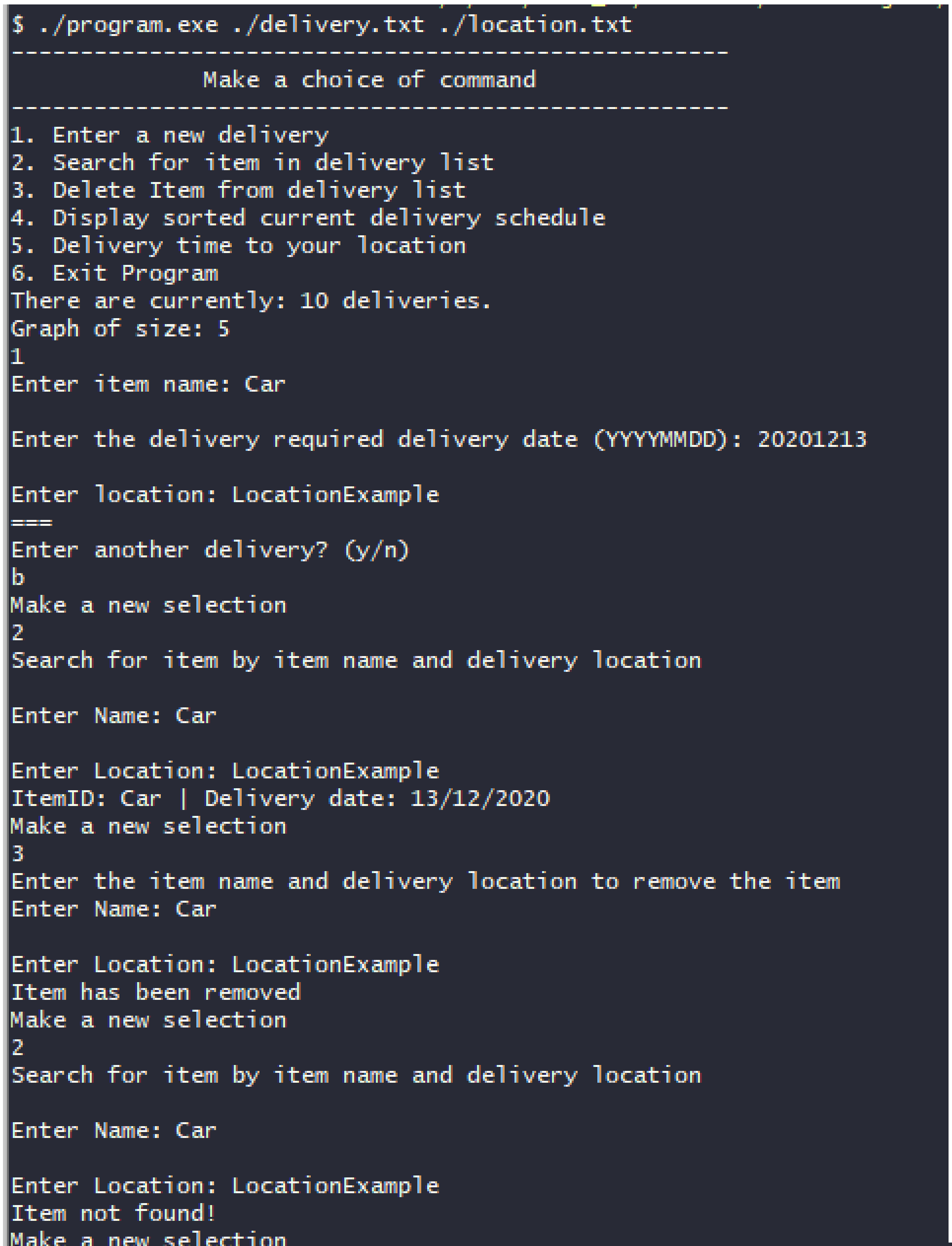
```
Make a new selection
5
Enter Current Location: 0

Location            Distance from: 0
0                   0
1                   10
2                   50
3                   30
4                   60
Make a new selection
```

We can see that the program points are mapped 1 less than in the image as we begin indexing from 0. However you can see simply that started from 0 or 1 in the image above. The shortest distances are correct we can see by observation, the best example of this is the path from 1 to 5, the direct edge is 100 but from 1 → 2 →3 → 5 we have a solution of 60 which is indeed the shortest path.

Hashing

The hashing functionality is called immediately on running the program as it stores every item in the input delivery list text file into a hash table, by calling hashInsert on the items. I am using a hashing function based on the ASCII codes of the strings of item name or ID and delivery location, multiplying by prime numbers. By ensuring a load factor of less than 0.7 or 70% by ensuring that the size of the array is much larger than the number of input items. I utilise open addressing as it is much faster when we maintain a load factor, in comparison to chaining. Especially because there is greater probability of collisions with hashing strings which could cause issues for chaining because suppose we have anagrams in the strings for example.

I have shown the usage of insertion, lookup, deletion from the hash table below.



```
$ ./program.exe ./delivery.txt ./location.txt
----------------------------------------------------
              Make a choice of command
----------------------------------------------------
1. Enter a new delivery
2. Search for item in delivery list
3. Delete Item from delivery list
4. Display sorted current delivery schedule
5. Delivery time to your location
6. Exit Program
There are currently: 10 deliveries.
Graph of size: 5
1
Enter item name: Car

Enter the delivery required delivery date (YYYYMMDD): 20201213

Enter location: LocationExample
===
Enter another delivery? (y/n)
b
Make a new selection
2
Search for item by item name and delivery location

Enter Name: Car

Enter Location: LocationExample
ItemID: Car | Delivery date: 13/12/2020
Make a new selection
3
Enter the item name and delivery location to remove the item
Enter Name: Car

Enter Location: LocationExample
Item has been removed
Make a new selection
2
Search for item by item name and delivery location

Enter Name: Car

Enter Location: LocationExample
Item not found!
Make a new selection
```

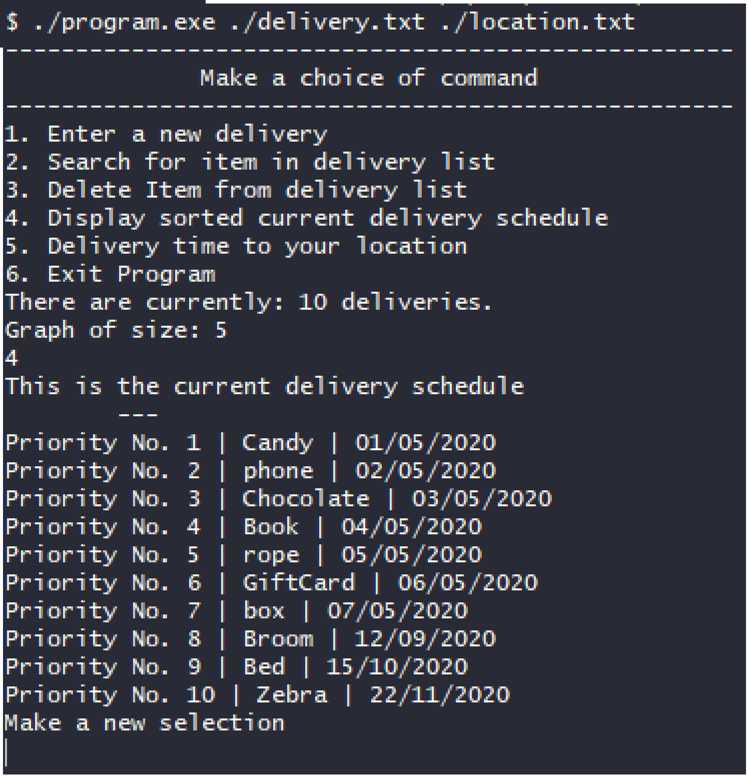
We can see on the left that the program automatically inserts all the input deliveries into a hashtable. Then we search for the item we want to deliver and the program returns the OBJECT for box indicating that the program can find the item based on its name and delivery location. The item object can then have many attributes such as delivery time, delivery run number and additional values if the program were to be more fleshed out.

Then we remove the delivery, which is essentially deletion this is typically done if the delivery is cancelled or item has been delivered hence removed from the hash table. And then when we try to search the item again it is not found.

The lookup and deletion also works in all the items placed into the hash table on startup of the program.

Merge Sort

Merge sort sorts the items based on a comparator within the merge sort function item.deliveryDate is compared as integer with another item. We can simply see it works by letter the dates be integers or as shown in the image below on delivery.txt, the dates have been sorted.



```
$ ./program.exe ./delivery.txt ./location.txt
----------------------------------------------------
              Make a choice of command
----------------------------------------------------
1. Enter a new delivery
2. Search for item in delivery list
3. Delete Item from delivery list
4. Display sorted current delivery schedule
5. Delivery time to your location
6. Exit Program
There are currently: 10 deliveries.
Graph of size: 5
4
This is the current delivery schedule
        ---
Priority No. 1 | Candy | 01/05/2020
Priority No. 2 | phone | 02/05/2020
Priority No. 3 | Chocolate | 03/05/2020
Priority No. 4 | Book | 04/05/2020
Priority No. 5 | rope | 05/05/2020
Priority No. 6 | GiftCard | 06/05/2020
Priority No. 7 | box | 07/05/2020
Priority No. 8 | Broom | 12/09/2020
Priority No. 9 | Bed | 15/10/2020
Priority No. 10 | Zebra | 22/11/2020
Make a new selection
```

Functionality Justification

Merge Sort

Merge sort is a simple and efficient way to sort lists and hence I chose it as a method of ordering the delivery item objects in ascending order of their delivery date. I took onboard the feedback from milestone two and used a class and use the < operator on its attribute for delivery date. For this any simple comparison sorting could have been used as we do not care so much about it being in-place and such. However one advantage of merge sort is that it is stable, so items with the same delivery date remain in the initial relative order (which could mean that a certain persons order came through first for whatever reason by a slight margin).

Hashing

I utilise hashing for its fast lookup and insertion time which allows the user to store and retrieve objects quickly as they desire. The items do not need or have any particular order per say in the hash table but they may have attributes attached to them that determine priority. It was pointed out that a AVL tree could also be used instead of hash table in milestone 2 feedback. This was a good suggestion as AVL tree allows for a sorted data structure but it requires a comparison operator between elements; and while it would entirely feasible to implement a AVL tree I decided that because it has O(logn) insertion. Whilst a hash table has O(1) insertion of elements as long as we keep a decent load factor, and having the items be sorted is not particularly useful in the sense of lookup and storing in that manner. Sorting would be a very infrequent use case, whilst insertion and deletion would be very common as items come in and out.

I utilise open addressing as it is faster as long as the load factor of the hash table is below 0.7 and since we typically won't have a large number of deliveries since a delivery company would not have that many deliveries in its system at one point in time and deliveries will be constantly being taken out of it and inserted and these operations tend to be faster using open addressing than chaining where multiple values could end up in the same linked list which must be traversed in order (O(n) time). *(Note this is outside of the time complexity testing of hash table for which I may need to test the maximum load factor where the entire table is filled)*

Dijkstra's Algorithm

Dijkstra's Algorithm is famous for finding the shortest path between a set of points with weighted paths between them. This can be used to represent locations and the time taken to travel between them or some sort of cost associated with travelling between them. Hence given a input of adjacency matrix which defined this cost associated with travelling between the set of vertices and if travel is even possible between two vertices, then we can use Dijkstra's algorithm to provide a user with the set of shortest paths to every point given their current position. Hence it made sense to use this algorithm for the use case shown in this report.

Time Complexity Analysis

An excel spreadsheet has been attached containing the full details of empirical time complexity calculations please see that for more detail. I have summarised the graphs here feel free to look at excel documents and manipulate graphs as required.

Time complexities for reference.

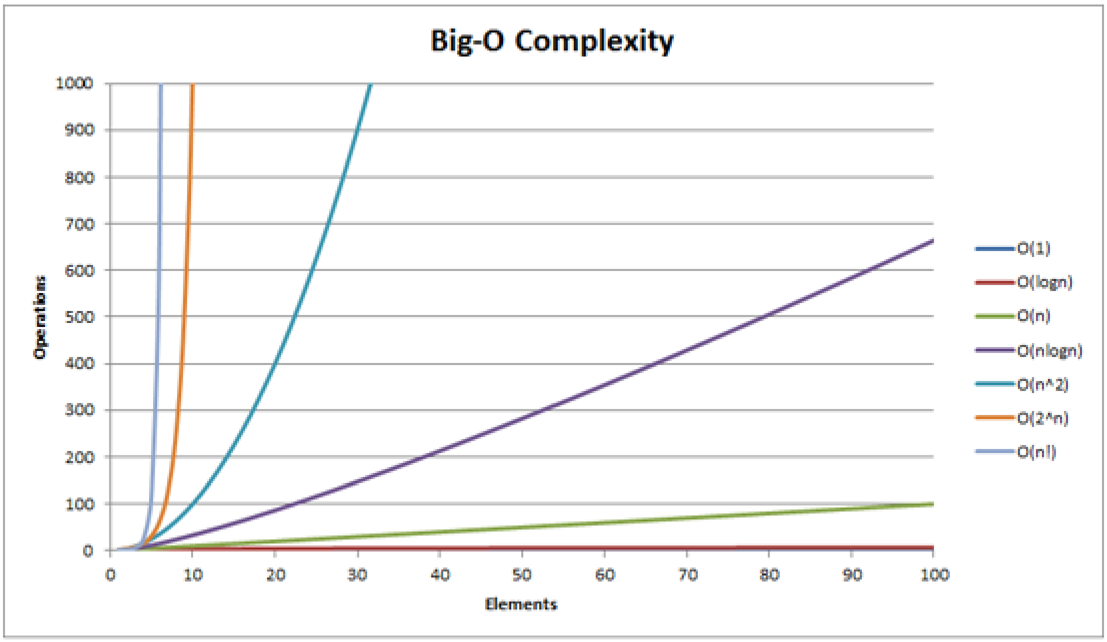
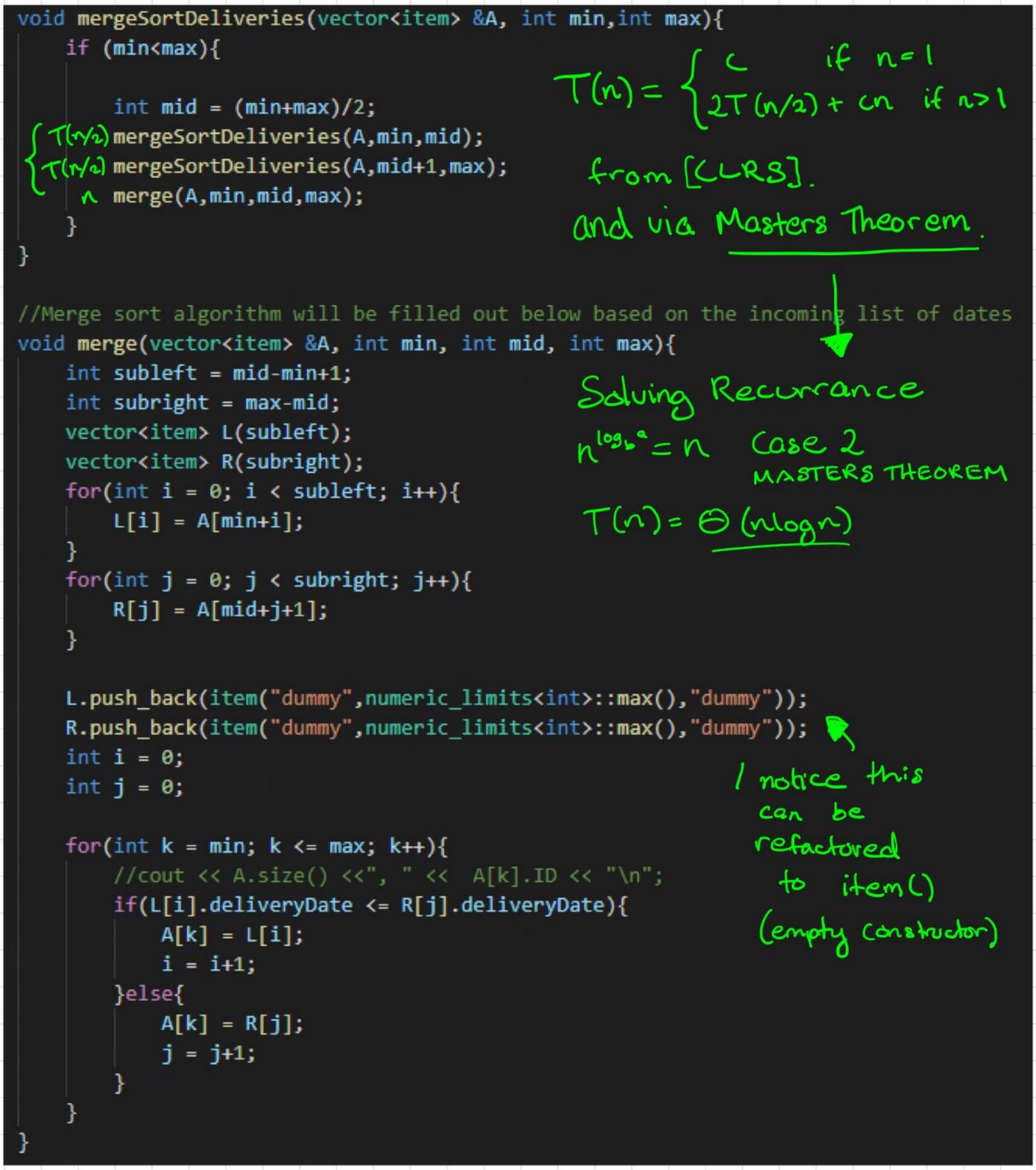


Image from: https://stackoverflow.com/questions/33214852/is-anything-less-than-n-is-log-n , 2015.

For the empirical analysis I used two external C++ programs I have provided to generate a massive data set of randomly generated names, numbers, strings, integers and the randomly generated adjacency matrix as well. Please note that these randomly generated test cases provided for empirical testing are not typically the real world use cases for this program as outlined by the assumptions. I have provided the delivery.txt and location.txt file which contain somewhat realistic small sample of data. But the randomly generated adjacency matrix will provide somewhat unrealistic shortest paths due to its random edge weights and connections, however the algorithm still functions. The large samples of data may take up to 5 seconds to execute on startup because the program inserts every item in the delivery list into a hash table on launch.

Merge sort ← Using CLRS implementation.



```
void mergeSortDeliveries(vector<item> &A, int min,int max){
    if (min<max){

        int mid = (min+max)/2;
        mergeSortDeliveries(A,min,mid);
        mergeSortDeliveries(A,mid+1,max);
        merge(A,min,mid,max);
    }
}

//Merge sort algorithm will be filled out below based on the incoming list of dates
void merge(vector<item> &A, int min, int mid, int max){
    int subleft = mid-min+1;
    int subright = max-mid;
    vector<item> L(subleft);
    vector<item> R(subright);
    for(int i = 0; i < subleft; i++){
        L[i] = A[min+i];
    }
    for(int j = 0; j < subright; j++){
        R[j] = A[mid+j+1];
    }

    L.push_back(item("dummy",numeric_limits<int>::max(),"dummy"));
    R.push_back(item("dummy",numeric_limits<int>::max(),"dummy"));
    int i = 0;
    int j = 0;

    for(int k = min; k <= max; k++){
        //cout << A.size() <<", " <<  A[k].ID << "\n";
        if(L[i].deliveryDate <= R[j].deliveryDate){
            A[k] = L[i];
            i = i+1;
        }else{
            A[k] = R[j];
            j = j+1;
        }
    }
}
```

$T(n/2)$

$T(n/2)$

n

$$T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2) + cn & \text{if } n>1 \end{cases}$$

from [CLRS].

and via Masters Theorem.

Solving Recurrance

$n^{\log_b a} = n$ Case 2

MASTERS THEOREM

$T(n) = \Theta(n \log n)$

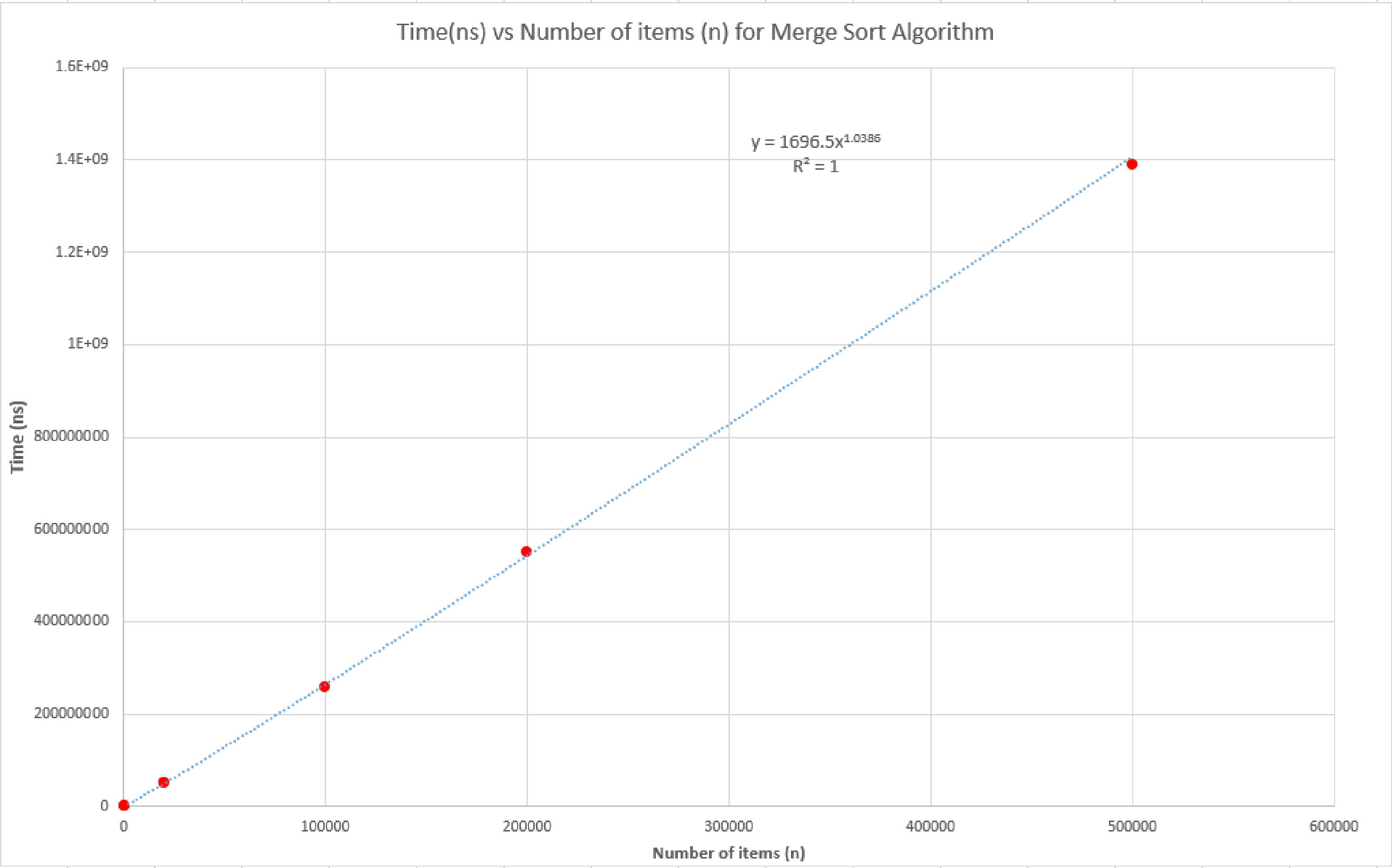
I notice this can be refactored to item() (empty constructor)

We are getting 2 subproblems of size $\frac{n}{2}$.

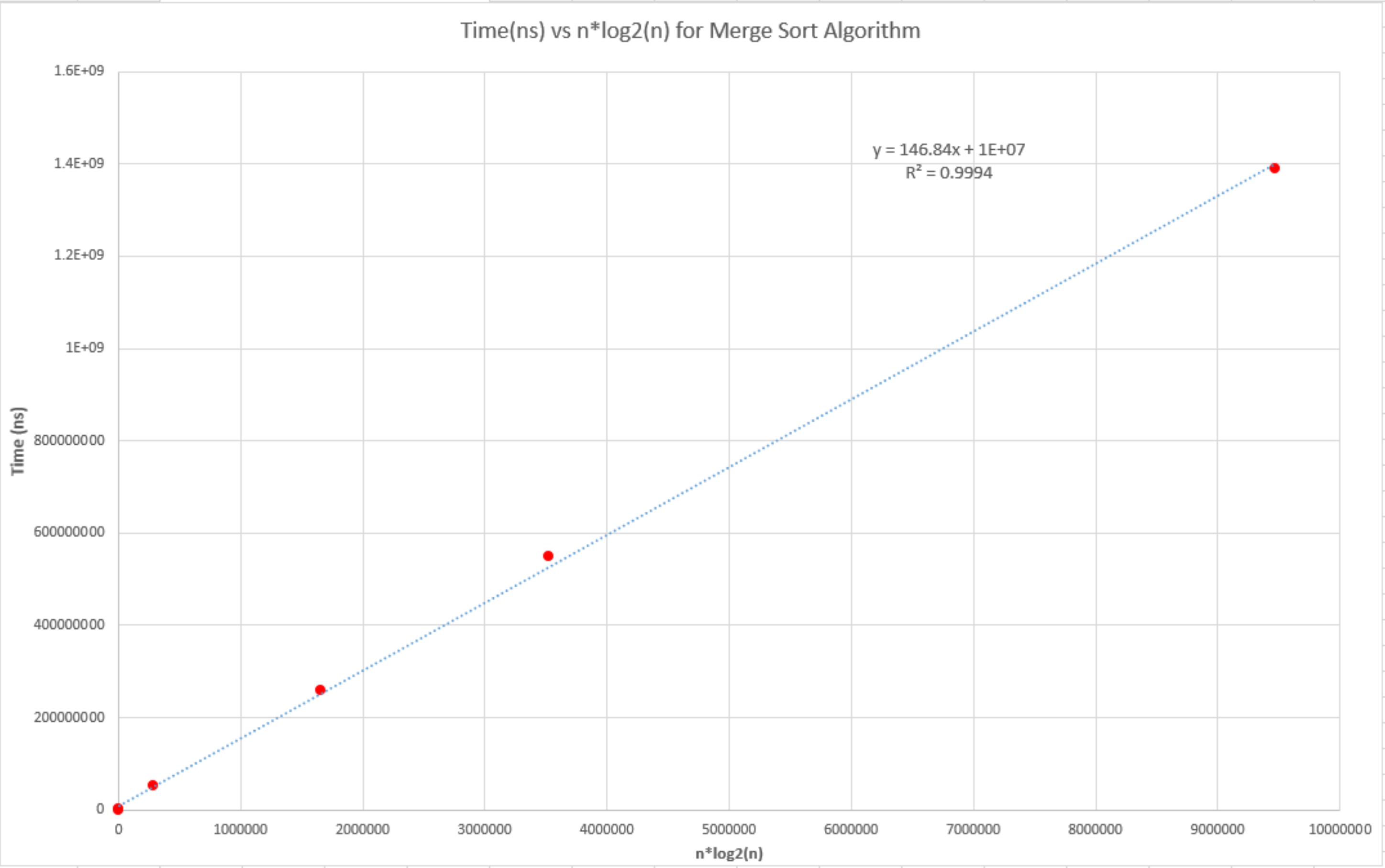
$a=2 \quad b=2$

The whole merge function is in cn time where c is a constant.

Empirically Graphed

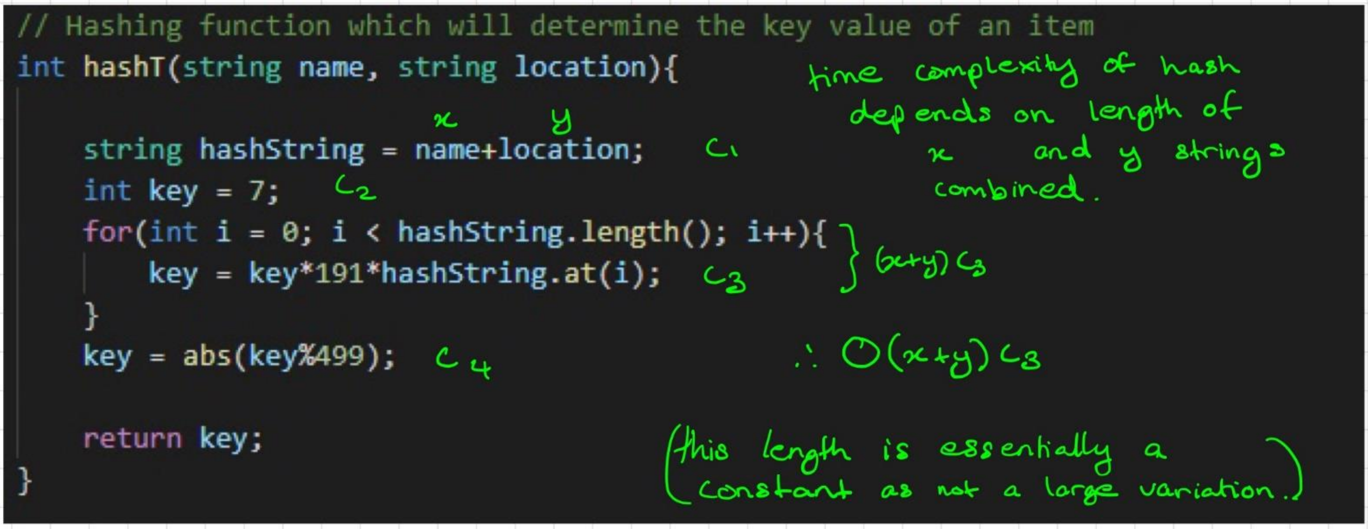


The 6 points graphed show that the runtime increases slightly faster than a linearly increasing function. This is what we would expect looking at the reference image for O(nlogn) time complexity. Furthermore graphing the time against nLogn as shown below gives a linear relationship as expected. The correlation between points is strong giving a good indication that the time complexity is indeed nLog$_2$(n).



The empirical analysis reflects what we would expect from the theoretical analysis and the time complexity of merge sort is increasing almost linearly according to empirical results, hence there is a good match between what we expect.

Hashing



```
// Hashing function which will determine the key value of an item
int hashT(string name, string location){

    string hashString = name+location;
    int key = 7;
    for(int i = 0; i < hashString.length(); i++){
        key = key*191*hashString.at(i);
    }
    key = abs(key%499);

    return key;
}
```

x y

c_1

c_2

c_3 } $(x+y)\,c_3$

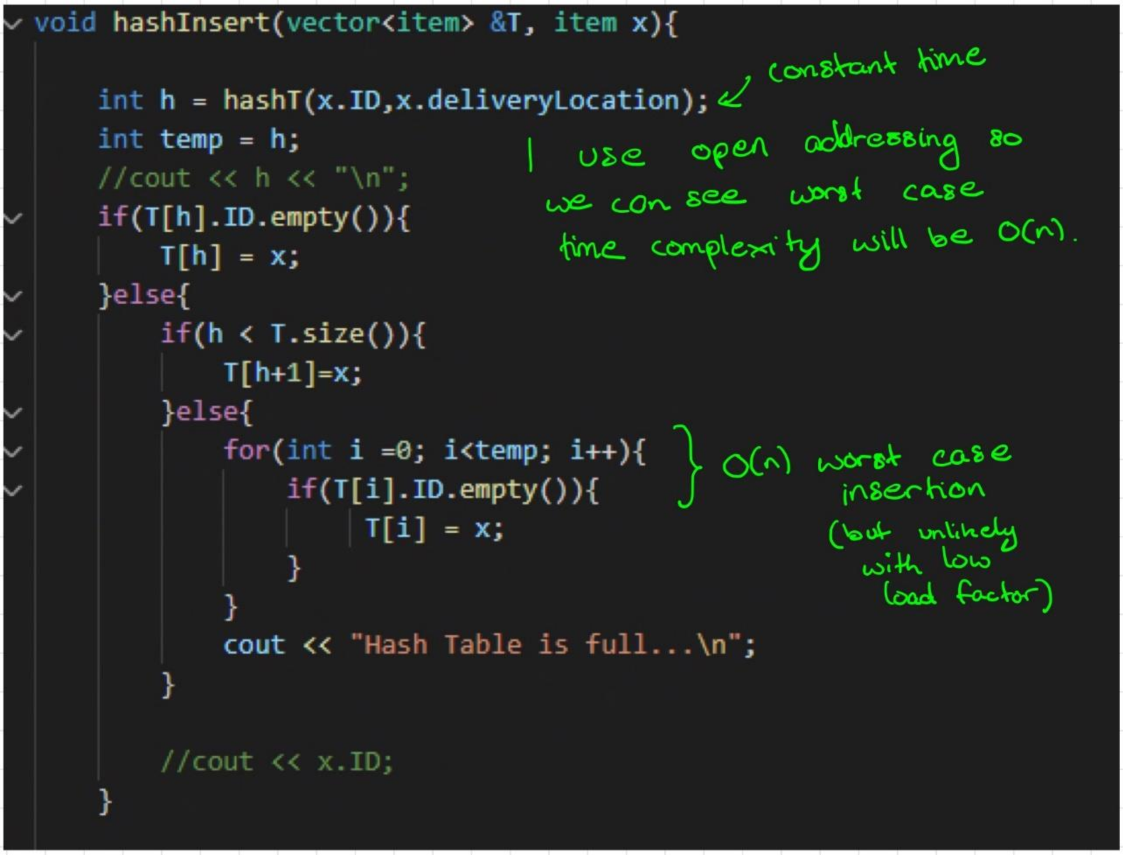
c_4

time complexity of hash depends on length of x and y strings combined.

$\therefore\ O(x+y)\,c_3$

(this length is essentially a constant as not a large variation.)

We know from lectures and CLRS that avg time is $\Theta(1+\frac{n}{m})$, statistically.



```
void hashInsert(vector<item> &T, item x){

    int h = hashT(x.ID,x.deliveryLocation);
    int temp = h;
    //cout << h << "\n";
    if(T[h].ID.empty()){
        T[h] = x;
    }else{
        if(h < T.size()){
            T[h+1]=x;
        }else{
            for(int i =0; i<temp; i++){
                if(T[i].ID.empty()){
                    T[i] = x;
                }
            }
            cout << "Hash Table is full...\n";
        }

        //cout << x.ID;
    }
```

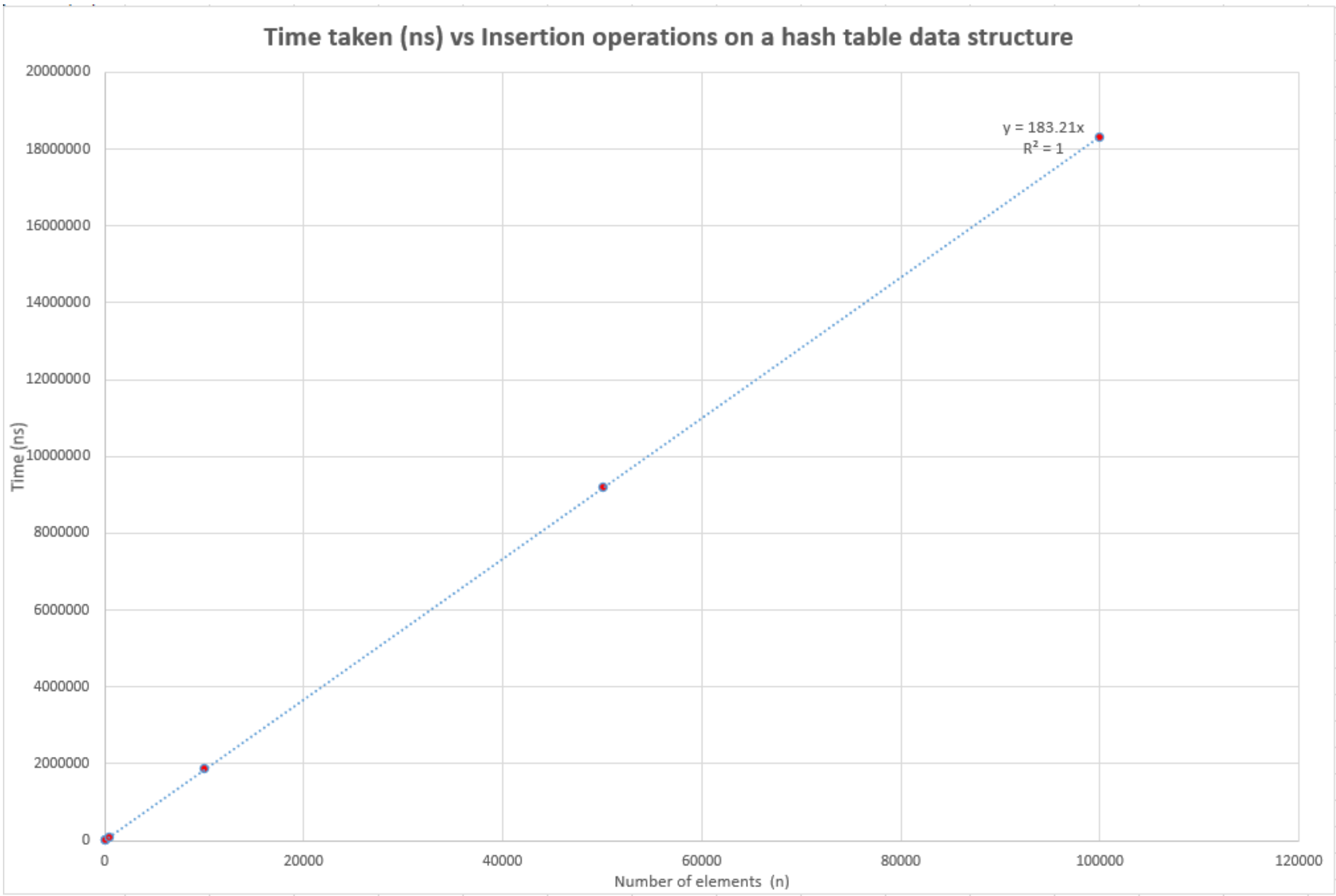
constant time

I use open addressing so we can see worst case time complexity will be O(n).

} O(n) worst case insertion (but unlikely with low load factor)

Empirical Analysis

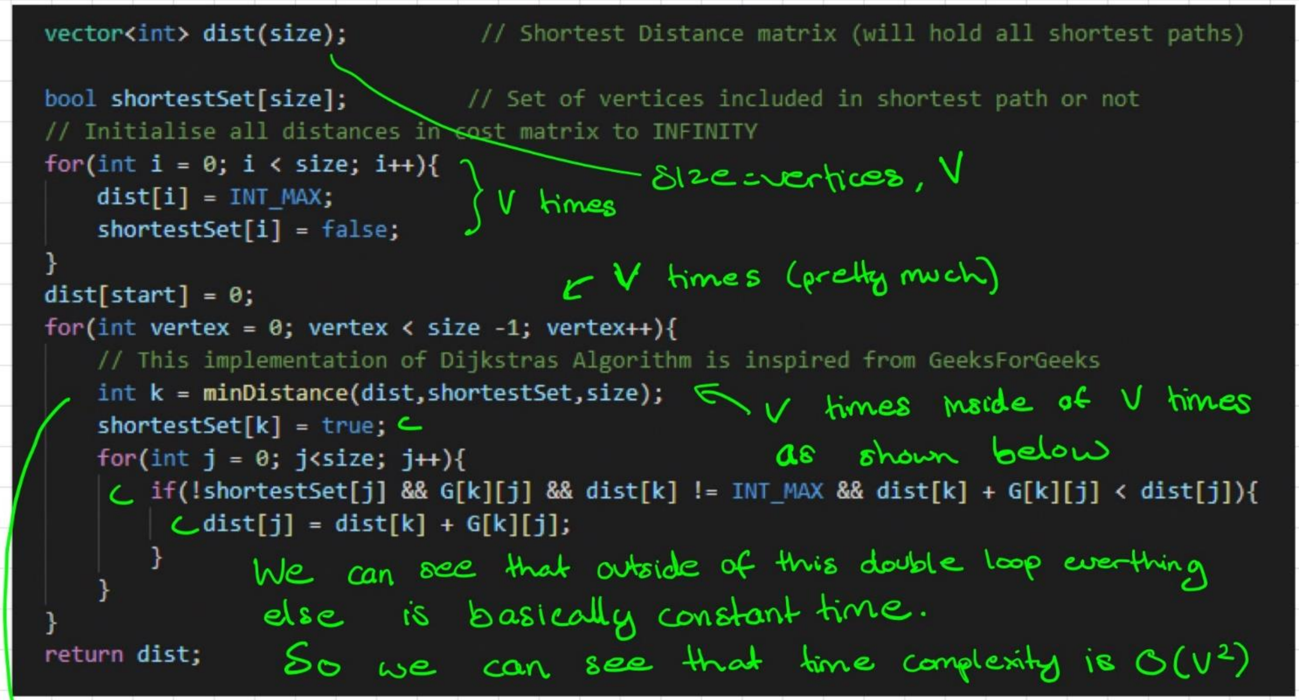
The program repeatedly inserts elements into the hash table on launch based on the length the list of items. So, I ran it using massive sample sizes up to 100000 items.



As we can see this is perfectly linear which means that insertion is constant time since n insertions takes c*n time as shown by the figure above. Deletion and lookup would be the same except that in the cases of reaching a higher load factor the worst case could be O(n). However I implemented the program such that the array is sized depending on a prime number multiple of the initial number of elements. So unless a LARGE number of elements are inserted without any being removed (which is unlikely for delivery company use case), it is highly unlikely that anywhere near the worst case would be achieved.

In general it seems that the hashing function provides constant time operations for all operations within the expected use case of the program. And the theoretical analysis seems to line up with what we expect.

Dijkstras Algorithm



```
vector<int> dist(size);          // Shortest Distance matrix (will hold all shortest paths)

bool shortestSet[size];          // Set of vertices included in shortest path or not
// Initialise all distances in cost matrix to INFINITY
for(int i = 0; i < size; i++){
    dist[i] = INT_MAX;
    shortestSet[i] = false;
}
dist[start] = 0;
for(int vertex = 0; vertex < size -1; vertex++){
    // This implementation of Dijkstras Algorithm is inspired from GeeksForGeeks
    int k = minDistance(dist,shortestSet,size);
    shortestSet[k] = true;
    for(int j = 0; j<size; j++){
        if(!shortestSet[j] && G[k][j] && dist[k] != INT_MAX && dist[k] + G[k][j] < dist[j]){
            dist[j] = dist[k] + G[k][j];
        }
    }
}
return dist;
```

size=vertices, V

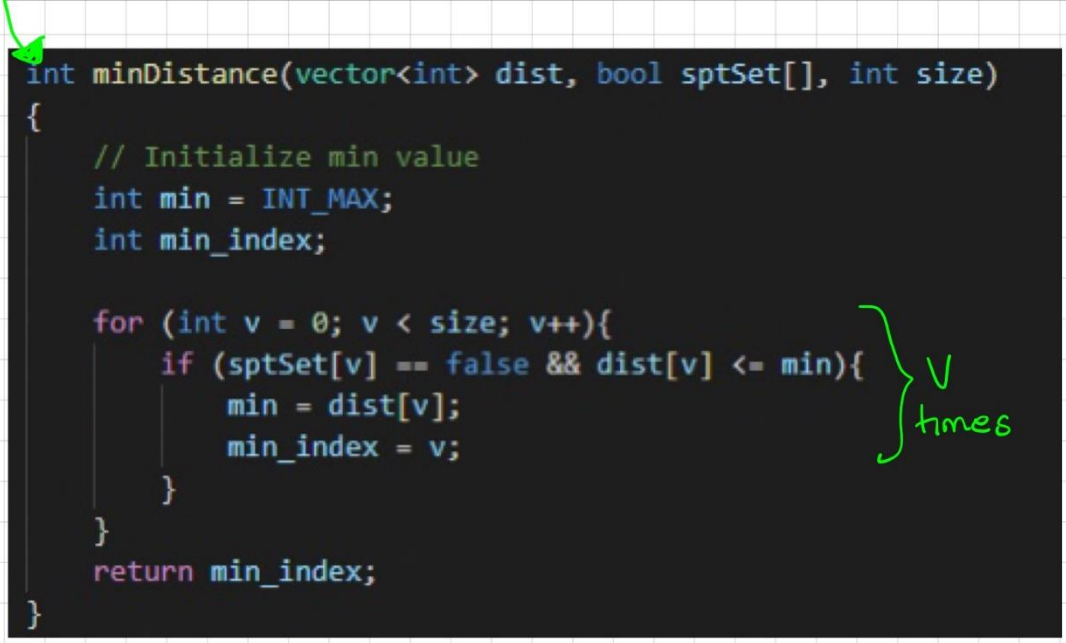
V times

V times (pretty much)

V times inside of V times as shown below

We can see that outside of this double loop everthing else is basically constant time.

So we can see that time complexity is $O(V^2)$

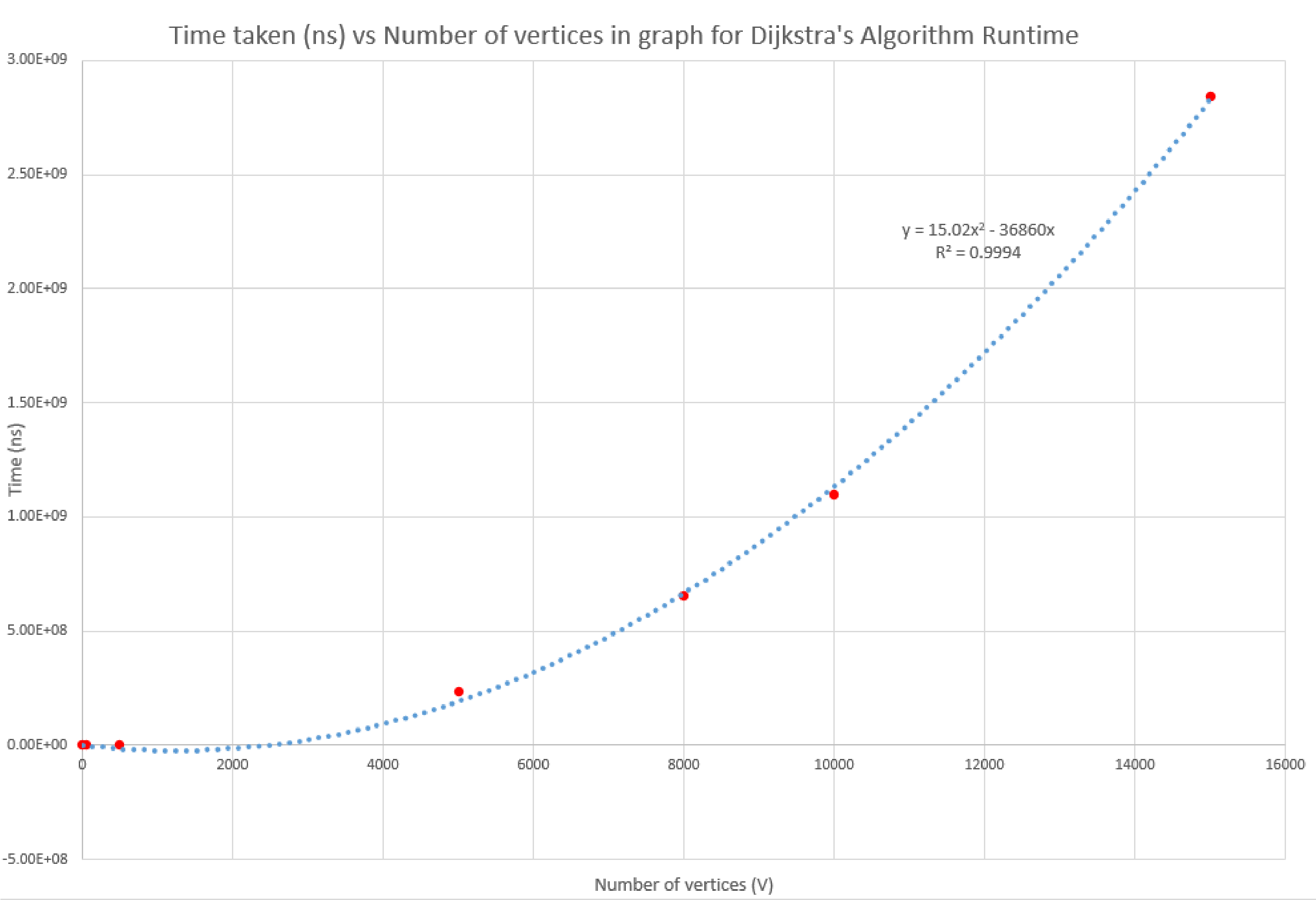


```
int minDistance(vector<int> dist, bool sptSet[], int size)
{
    // Initialize min value
    int min = INT_MAX;
    int min_index;

    for (int v = 0; v < size; v++){
        if (sptSet[v] == false && dist[v] <= min){
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
```

V times

If I used min heap instead of this function the time complexity would be $O(V\log(V))$

The theoretical time complexity of my implementation of Dijkstra's is $O(V^2)$ because I have not used a min-heap.



As shown above we see that the time complexity is increasing in a ‘power’ relationship indicating that is n raised to some exponent. The theoretical analysis indicates O(V^2) time complexity and we can see from the above graph that it is approximately at that value with excel calculating a R^2 of almost 1, meaning the line is a very good fit for the data.

Hence the function performs as we would expect after empirical analysis.

Improvements

An improvement to the program would be to create a location class and then have additional data associated with that delivery location and then Dijkstras algorithm would have added layer of complexity as the vertices are not provided by a simple adjacency matrix but by a number of objects of type location and connecting locations. I initially thought to do this however decided that it would be too time consuming.

References

Dijkstra’s Algorithm implementation - https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Other references made for random number generation for generating test files, specific references are in the source files.