Final Project

1. Project name: Food Delivery System

2. Description: An application helps the food delivery company to better arrange their delivery, which is to determine the order of delivery by sorting all order's deadlines for delivery (based on the ascending order of deadline). Then search for the corresponding customer's address for the order we plan to deliver. After finding relevant customer's address, select the shortest path from the food delivery company to customer's address. Operate the same operations until all orders have been delivered successfully.

3. Assumption:

1. Once the order has been made, the deadline to deliver the order will always stay the same, it won't be affected by any circumstances and changes
2. Each order has different deadline, and the application is to prioritize the order which has the soonest deadline.
3. Each order is delivered individually.
4. The order cannot be cancelled once it has been made.
5. Customers can no longer change their address after they have made and confirmed the order.
6. The shortest path is selected from all paths between food delivery company and customer's address.

4. Functionalities:

In this section, I will describe three functionalities and the role of each on the application.

4.1 Functionality1 - Process orders in the order of the soonest deadline.

Usage: This functionality helps the food delivery company sort all deadlines belong to different orders based on the ascending order, and then help the company determine the order of delivery.

4.2 Functionality2 - Search for corresponding customer's address for which order we plan to deliver.

Usage: This functionality helps the food delivery company find the corresponding customer's address, for which customer's order are scheduled to be delivered. Only when the company finds customer's address can it deliver the order.

4.3 Functionality3 - Find the shortest path between food delivery company and customer's address.

Usage: Given food delivery company's address, and the corresponding customer's address has already been found. This functionality helps the food delivery company select the shortest path to deliver the order. The reason why choosing the shortest path is that the company could save delivery costs and time.

5. Data structure/Algorithms applied in each functionality:

In this section, I will list the applied data structures/algorithms respectively, and then I will show the reason why they are suitable in the application.

5.1 Functionality1 - Data structure: heaps (max-heap)

Input: a series of deadlines which belong to different orders

Output: a series of sorted deadlines based on the ascending order

Supporting reasons:
heaps are quick to build, max-heap also provides help to find the soonest deadline which order is scheduled to be delivered first.
Furthermore, comparing to some other comparison-based sorting methods like insertion sort, although their space complexity is same, heapsort provides a smaller time complexity O(nlogn) in the worst case and average case, while insertion sort changes the time complexity to O(n^2) in the same cases, hence the time complexity of heapsort is relatively stable.(n is the number of deadlines need to be sorted)
Comparing to some other non-comparison-based sorting like counting sort, the space complexity for heapsort only need O(1), while counting sort needs the space complexity O(n+k), so heapsort is much more space saving. As for their time complexity, in all cases, heapsort only needs O(nlogn) while counting sort need O(n+k), so heapsort will be more efficient. (n is the number of deadlines need to be sorted and k is the input range in counting sort)
Furthermore, from my previous assumption, each order which cannot be changed after confirming the order has different deadlines and will be deliver individually. In this case, using heapsort to determine the order of delivery is able to avoid the situation where multiple orders have the same deadline, also, the order of delivery won't be dynamically changed by any factors.

5.2 Functionality2 - Data structure: Hashing
Input: a series of data, each data contains a deadline and an address respectively
Output: fetch the corresponding customer's address by searching for the deadline
Supporting reasons:
Hashing provides a fast approach to search for the target element.
Tree data structures are often used to implement searching operations, I will choose binary search tree as a specific example to compare with hashing. We want to search for the target element in all n elements. In binary search tree, if the tree is balanced, then each time we implement the search operation needs time complexity O(logn), but the expected time complexity in hashing is only O(1). From this point, hashing seems to be more efficient than binary search tree to search for the element.
Binary search is also a commonly used approach, but it still needs the time complexity O(logn) to search for the element, which is less effective than hashing.
However, there is drawback in hashing, as the number of inputs creasing, the probability of collision will also increase. Therefore, I come up with an approach to handle collision, which is to adopt simple uniform hashing and chaining. In this case, a successful searching operation still only needs a constant time.
My previous assumptions provide suitable constraints to the implementation of functionality2. For a successful searching, we have to avoid the situation that the target address is deleted with the cancelling of the order, as well as the continually changing customer's address.

5.3 Functionality3 - Algorithm: Dijkstra's algorithms
Input: the address of the food delivery company and the address of customer's address are two nodes in weighted graph, and all weighted edges between them.
Output: the shortest path between food delivery company and customer's address
Supporting reasons:

Dijkstra's algorithm provides support to find the shortest path between two nodes in a weighted graph, and I will make comparison with other algorithms which provide the similar help to convince the reason why I choose this specific algorithm.
Dynamic programming is able to resolve the whole problem into sub programs, and those sub-problems are dependent, hence we need to solve all sub problems to get the optimal solution to the shortest path, the requirement for space is quite large and its total time complexity is also much greater than Dijkstra's algorithm.
As for Minimal spanning tree, although it is helpful to find the shortest path, it is not always as helpful as Dijkstra algorithm for the weighted graph which has many edges. Furthermore, although its time complexity is similar as Dijkstra algorithm, its space complexity is much greater so it cannot save the space. Therefore, in some cases, Dijkstra algorithm has relatively optimal time complexity and space complexity, it is more effective in solving the problem.
Dijkstra algorithm also have some limitations, it cannot handle the graph with negative weighted edges, but all weighted edges in this application are positive. Moreover, Dijkstra algorithm has good scalability and can adapt to many problems.
In order to find the shortest path between two sites, we have to first confirm the what the two sites are. Hence, In the last assumption, I have restrained the shortest path is selected between the food delivery company and corresponding customer's address.

6.Theoretical time complexity analysis

In this part, I will analyze the theoretical time complexity in both average case and worst case for each functionality.

Functionality1:

Worst case:

Worst case:

worst case happens when the heap is laugest imbalance

that is: when the bottom of the heap happens to be half full

In heapsort,

We mainly need two different operations:

① build a max heap ② heapify the rest of elements in the heap

I will analyse the time complexity of each operation

For building a max heap,

Assuming there are n elements (deadlines) in the max heap, then the height of the heap is $\log n$, a heap with the height h can contain at most $\frac{n}{2^{h+1}}$ elements

$\sum_{h=0}^{\log n} (\frac{n}{2^{h+1}}).\ O(h) = O(n \sum_{h=0}^{\log n} \frac{h}{2^h})$ $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

Hence $O(n \sum_{h=0}^{\log n} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(2n) = O(n)$

For heapify the rest of elements in the heap (h is the height of heap)

Size of left-subtree $= \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ size of right sub-tree $= \sum_{0}^{h-1} 2^i = 2^h - 1$

then the total number of elements in the heap $n = 1 + 2^{h+1} - 1 + 2^h - 1 = 2^h(2+1) - 1$

$2^h = \frac{n+1}{3}$

Hence we can define the reccurence for heapify: $T(n) \leq T(\frac{2n}{3}) + C$

Solve the reccurrence using the master's Theorem,

$T(n) \leq T(\frac{2n}{3}) + C = O(\log n)$

Everytime we operate building a max heap need the time complexity $O(n)$

and every n-1 time we operate max-heapifying need the time complexity $O(\log n)$

Then we need a total $O(n \log n)$ time to max-heapify

Hence, the total time complexity for heatsort (using max heap) is $O(n \log n)$

Average case:

Average Case:

Similar as what I analyzed in worst case.

I will anaylze the average time complexity for

① build a max heap

h is the height of max heap. assume there are n elements in the heap.

then $h = \log n$

starting from the nodes on the penultimate layer, they must be transferred with their child nodes based on their order (if the order is correct than do nothing)

And for upper layer, operate the same operation until the heap strictly follows the principle of a max heap.

Assuming X is the random variable.

$E[X] = \sum_{i=1}^{h} i \cdot (h-i) \cdot 2^{i-1}$ (where 2^{i-1} is the number of elements in a layer, and $h-i$ is the number of comparisions)

$= 1 \cdot (h-1) \cdot 2^0 + 2 \cdot (h-2) \cdot 2^1 + \cdots + n \cdot (h-h) \cdot 2^{h-1}$

$= 2^{h-1} + 2^{h-2} + 2^{h-3} + \cdots + 2 - (h-1)$

$= 2^h - h - 1$

$h = \log n$. $E[X] = 2^{\log n} - \log n - 1 = n - \log n - 1$

let c donates the cost

$T(n) = cn - c\log n - c = O(n)$

② heapify the rest of max heap

the operation will be excuted n-1 times

$E = \sum_{i=1}^{h} i \cdot \frac{2^{i-1}}{n} = \frac{1}{n} \sum_{i=1}^{h} i \cdot 2^{i-1}$

$= \frac{1}{n} (1 \times 2^0 + 2 \times 2^1 + \cdots + n \cdot 2^{n-1})$

$= \frac{1}{2n} n \cdot \log n = \frac{1}{2} \log n$

$T(n) = \frac{1}{2} c \log n = O(\log n)$

Total Average time complexity = $O(n) \times O(\log n) = O(n \log n)$

Hence heapsort's time complexity in the average case is $O(n \log n)$

Functionality2:

The implementation of Functionality1 and Functionality2 is independently. Actually, Functionality1 helps the food delivery company to arrange the order of delivery, and each time the company deliver the order, Functionality2 helps the food delivery company to fetch the corresponding customer's address based on the deadline of order.

Hence, in this part, I will analyze the time complexity on implementing Functionality2.

Average Case:

Average Case:

I use simple uniform hashing and chaining to solve collision

The whole process could be seen as two parts:

① Inserting all data into the hashtable ② search for the target element

For Inserting all data into the hashtable:

the time complexity is $O(1)$

For a successful searching of the target element:

Given a hashtable with m slots which could contain n elements.

Then we can define the load factor $\alpha = \frac{n}{m}$

time to compute hash $h(k)$ is $O(1)$

In order to search for an element (k, d)

First define an indicator random variable

$$X_{ij} = \begin{cases} 1 & h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$$

and the collision probability is $E[X_{ij}] = P(X_{ij}=1) = \frac{1}{m}$

Hence we can compute:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}])$$

$$= \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) = 1+\frac{1}{mn}\sum_{i=1}^{n}(n-i)$$

$$= 1+\frac{1}{mn}(\sum_{i=1}^{n}n - \sum_{i=1}^{n}i)$$

$$= 1+\frac{1}{mn}(n^2-\frac{1}{2}n^2-\frac{1}{2}n) = \frac{1}{2}+\frac{1}{mn}(\frac{1}{2}n^2-\frac{1}{2}n)$$

$$= \frac{1}{2}+\frac{n}{2m}-\frac{1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Then the total time complexity is $O(1) + O(1+\frac{\alpha}{2}-\frac{\alpha}{2n}) = O(1+\alpha)$

However, if the number of slots in the hashtable is proportional to the elements in the hashtable

$n = O(m) \Rightarrow \alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$

Back to $O(1+\alpha)$, then the average total time complexity will also be a constant time. which is $O(1)$

Combining the above two operations, the time complexity is $O(1)$

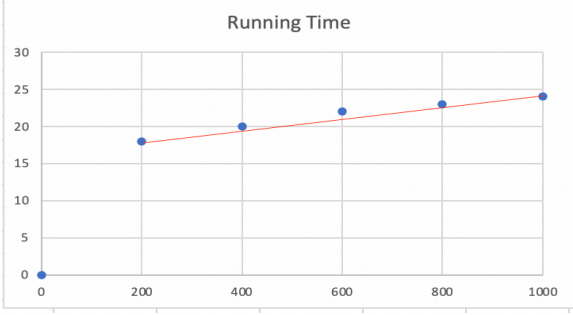
Worst case:

Worst case happens when we have to traverse the whole linked list to find the target element. Similar as what I analyzed in the average case, the whole implementation process could be seen as two parts: insert all data in the chained hash table; search for the target element. I will analyze the time complexity of each.

For inserting all data into the chained hash table:
If the pre-insert elements doesn't appear in the table, then inserting the elements into the table only needs a constant time, which is O(1).
Searching for the target element:
The worst time complexity in searching is related to the length of linked list. Assuming all n elements are hashed into the same slot, then there will be a linked list with length n. In this case, the time complexity to compute hash function is O(1), but the time complexity to successfully search for an element is O(n).
Combining above results, in the worst case, a successful searching needs the time complexity O(n).

Functionality3:
Given a weighted directed graph G = (V, E), the start node is the food delivery company and the end node is corresponding customer's address.
V is the number of nodes in the graph, and E is the number of edges. For each v belongs to V, d is the upper bound of weight on the shortest path from start node s to v.
Actually, the implementation of Dijkstra algorithm depends on the implementation of minimal priority queues.
There are three priority queue operations implemented to maintain the minimal priority queue, which are insert, extract-min and decrease-key respectively. I will analyze the time complexity of each operation to generate the overall time complexity.
A set S contains all nodes in the graph, assume all shortest paths from the start node to each node in the set has already been found, then the algorithm will select the node which has the shortest path and add it to the set S. Hence, for each node in the graph, the algorithm will perform insert and extract-min operation only once because each node is added to the set S only once. The number of edges in the graph is |E|, so Dijkstra algorithm implements decrease-key operation for at most |E| times.
Using the numbered node from 1 to |V|, and put v, d into the corresponding v-th item in the list. In this case, the time complexity to implement insert and decrease-key operation is O(1) each time, while the time complexity to implement extract-min operation is O(V). So we can conclude the total time complexity for Dijkstra algorithm is O(V^2+E) = O(V^2).
Moreover, we could use heaps to implement minimal priority queue and consequently optimize the time complexity. Building a heap need the time complexity O(V). For extract-min operation, there are |V| operation times, so the time complexity is O(logV). For decrease-key operation, the time complexity is O(logV), and there are at most |E| operations.
Therefore, the total time complexity is O((V+E)logV). If all nodes in the graph id reachable from the start node, then the total time complexity will change to O(ElogV).

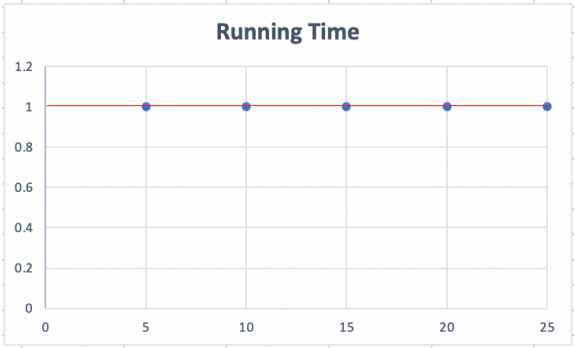
7.Empirical Time Complexity:
Functionality1:



The five different test files have the input size 1000, 800, 600, 400, 200 respectively. I test functionality based on each test file and get the running time of each.

The above graph shows the running time of each test file, where the horizontal axis is the input size (the maximal test input size is 1000), and the vertical axis is the corresponding running time(use ms to show running time). It is relatively linear.
And the empirical time complexity is around O(nlogn), which is similar as the theoretical time complexity.

Functionality2:



The five provided test files have the input size 5, 10, 15, 20, 25 respectively. And I test each of them to get the running time of implementing the functionality.
The graph above shows the running time of each test file, where the horizontal axis is the input size(the maximal test input size is 25), and the vertical axis is the corresponding running time. From observation, no matter what the input size is, the running time is always 1ms.
Hence, the empirical time complexity of functionality2 is always a constant time, the time complexity is O(1), which is the same as the average case theoretical time complexity.

Reference:

[CLRS]6.1 6.2 6.3 6.4 (For the codes in Functionality1, I referenced the pseudocode provided in this section, as well as the theoretical time complexity analysis)
[CLRS]11.2 11.3
[CLRS]24.3
https://stackoverflow.com/questions/5343689/java-reading-a-file-into-an-arraylist
https://blog.csdn.net/u010452388/article/details/81283998
https://www.cnblogs.com/ysocean/p/8032656.html
https://www.cnblogs.com/didiaodidiao/p/9249485.html

Note: The programming language is Java and will run on MacOS.