Name:	Suhas Kamath	**Assignment No:**	Assignment 1
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS221
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Intro. to Scalable Systems
Date:	September 5, 2025	**Term:**	AUG 2025

Question 1

Solution Approach

In this question, we had to find the minimum weight of the duplicate parcels. The list of parcels was provided in the input file. An initial approach could be to run a nested for loop, which should read the list and identify the minimum weight of the duplicate parcels. This would be an inefficient algorithm. A much better algorithm is as follows:

1. Create a hash map to store the id and the minimum weight of each parcel.
2. Create a set to store the list of package ids that have duplicates.
3. Loop through the list of parcels and update the minimum weight with respect to each id in the above hash map.
4. During this loop, if a package is found to be duplicated, it is added to the set.
5. By the end of the for loop, we have two pieces of data: the id numbers of the duplicated parcels, and the minimum weight of each parcel.
6. Using the above two pieces of data, we create a separate two-dimensional vector array to store the list of duplicated packages with their minimum weight.
7. We then sort and return this vector array.

Time and Space Complexity Analysis

Time Complexity

First, let us analyze the program to determine its time complexity. The code can be divided into the following major sections, namely:

1. Creation and initialization of the required data structures.
2. The for loop that calculates the list of duplicate package ids and the minimum weight of each parcel.
3. The for loop that compiles the final result, i.e duplication parcel ids and their respective minimum weights.
4. The sort() function that sorts the package ids based on their id number.

Let us now analyze the time complexity of each section. Let us assume that n = number of parcels and m = number of duplicate parcels.

Section	Time Complexity	Reason
Section 1	$O(1)$	Initialization of variables and data structures takes constant time, independent of the values of n and m.
Section 2	$O(n)$	The loop executes n times. Hash map and set operations (insert, find, update) take $O(1)$ on average. Therefore, the total average time complexity is $O(n)$.
Section 3	$O(m)$	The loop iterates through the m duplicate IDs. For each ID, it performs an $O(1)$ hash map lookup. The total time is therefore $O(m)$.
Section 4	$O(m \log m)$	The complexity of the sorting function

Hence, we can state that the worst case complexity of the algorithm is $O(m \log m)$. Because we know that the number of duplicate parcels is definitely less than the total number of parcels, we can state that $m < n$. Hence, we can infer that the asymptotic worst case complexity of the code is $O(n \log n)$.

In the best case, the worst case complexity can be enhanced to $O(n)$. The best case is when there are absolutely no duplicate packages, thereby eliminating the sorting.

Space Complexity

There are three main data structures. Their space complexities are as follows:

1. Hash map - $O(n)$
2. Set - $O(n)$
3. Result vector - $O(m)$

Since we know that $m = O(n)$, the asymptotic space complexity of this algorithm can be considered as $O(n)$.

Experimental Setup

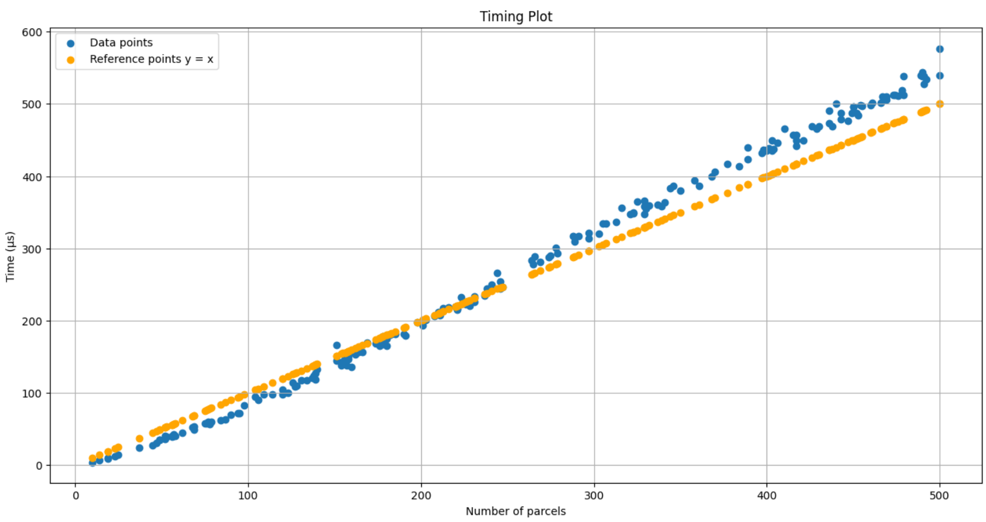
I first wrote the code on my personal computer. I had downloaded version 14 of the C++ compiler, and was using several new features provided in this version. This includes declaring variables with the auto keyword, and for-each loops. When I got access to the cluster, I realized that the cluster was still using C++ version 8.5. I then had to manually degrade the code, such that it can run on the cluster.
The code was then executed on the cluster provided. I created two files, namely q1_input.csv and q1_output.csv. I then wrote a program that does the following:

1. Reads one input from the q1_input.csv file.
2. Processes it using the function present in user_code.h.
3. Compares the output generated by the user_code.h file with the real output present in q1_output.csv.
4. Notes down the time taken with respect to the number of nodes in the q1_timings.csv file.

I then moved the q1_timings.csv file to my personal computer, where I drew a graph based on the data in the csv file. For this, I decided to use Python and its matplotlib library, which I am familiar with. The plotted graph is attached the next section.

Empirical Observations

The following time graph was obtained:



Name:	Suhas Kamath	**Assignment No:**	Assignment 1
SR No:	06-18-01-10-51-25-1-25945	**Course Code:**	DS221
Email ID:	suhaskamath@iisc.ac.in	**Course Name:**	Intro. to Scalable Systems
Date:	September 5, 2025	**Term:**	AUG 2025

Additional Insights

Some further optimizations can be made if the input data was sorted on basis of ID. In such a case, no sorting shall be required. This would essentially enhance the time complexity to $O(n)$. The space complexity, however, shall remain at $O(n)$.

Question 2

Solution Approach

In this problem, we are given the pre-order and in-order traversals of a binary tree, along with a list of parcels stored at each leaf. We are also given several queries, each asking for the highest junction in the tree where all the parcels in that query are present in the sub-tree of that junction.

The most obvious first thing to do is to reconstruct the tree using the in-order and pre-order traversals. We can assign parcels to their respective leaves. Then, to actually solve the problem, we can propagate the parcels upward to each ancestor. Then, we can check the queries against each node to get the solution. This solution, however is very wasteful.

We can optimize the code further by using the fact that each parcel can be present only on one leaf node. Just like before, we reconstruct the tree from the pre-order and in-order traversals. Next, we pre-process the tree using a depth-first search to record the depth of each node and to fill a binary lifting table. This binary lifting table is used to quickly access the ancestor of a particular node. After this, we traverse the leaves and map every parcel to the ID of the leaf node where it resides.

Now that the tree is constructed and the parcels are assigned to the respect leaf nodes, we can start processing queries. For each query, we look up the leaf nodes corresponding to the parcels in the query and compute their lowest common ancestor. The node we obtain at the end is the junction we are looking for.

This method avoids the redundancy of storing parcel sets at many nodes and essentially reduces the problem to a series of LCA computations.

Time and Space Complexity Analysis

Time Complexity

First, let us analyze the program to determine its time complexity. The code can be divided into the following major sections, namely:

1. Building the tree from the in-order and pre-order traversals.
2. Traversing the tree in DFS to assign a ID to each node.
3. Pre-processing the required parameters for Lowest Common Ancestor (LCA).
4. Mapping each parcel to a leaf node.
5. Solving each query one by one.

Let us now analyze the time complexity of each section. Let us assume that n = number of nodes, p = number of parcels and q = number of queries.

Section	Time Complexity	Reason
Section 1	$O(n)$	Construction of a binary tree from its in-order and pre-order takes $O(n)$ time.
Section 2	$O(n)$	DFS traversal requires $O(n)$ time.
Section 3	$O(n \log n)$	For each node, we will add a maximum of $(\log n)$ ancestors. Because we have n nodes, this will require $O(n \log n)$ time.
Section 4	$O(p)$	For each parcel, we will assign to its respective leaf node. This will require $O(p)$ time.
Section 5	$O(qp \log n)$	For each parcel in a query, we identify its leaf node. Because there are q queries and a maximum of p parcels per query, we will have to obtain a maximum of $q \times p$ leaf nodes. We then take these leaf nodes and find their common ancestor, thereby taking $O(qp \log n)$ time.

From this, we can state that the asymptotic worst case complexity of the code is $O(n + n + n \log n + p + qp \log n)$. Because we know that $n = O(n \log n)$ and $p = O(qp \log n)$, we can simplify the asymptotic worst case complexity of the code to $O(n \log n + qp \log n)$. This can also be re-written as $O(max(n, qp) \times \log n)$.

Space Complexity

There are four main data segments. Their space complexities are as follows:

1. Tree - $O(n)$
2. Binary lifting table - $O(n \log n)$
3. Parcel mapping - $O(p)$
4. Query storage - $O(q \times p)$

The asymptotic space complexity of this algorithm can be considered as $O(n + n \log n + p + qp)$. Because we know that $n = O(n \log n)$ and $p = O(qp)$, we can simplify the asymptotic worst case complexity of the code to $O(n \log n + qp)$. This can also be re-written as $O(max(n \log n, qp))$.

Experimental Setup

I followed a similar setup as Question 1 for the experimental setup.

I first wrote the code on my personal computer. I had downloaded version 14 of the C++ compiler, and was using several new features provided in this version. This includes declaring variables with the auto keyword, and for-each loops. When I got access to the cluster, I realized that the cluster was still using C++ version 8.5. I then had to manually degrade the code, such that it can run on the cluster.
The code was then executed on the cluster provided. I created two files, namely q2_input.csv and q2_output.csv. I then wrote a program that does the following:

1. Reads one input from the q2_input.csv file.
2. Processes it using the function present in user_code.h.
3. Compares the output generated by the user_code.h file with the real output present in q2_output.csv.
4. Notes down the time taken with respect to the number of nodes in the q2_timings.csv file.

I then moved the q2_timings.csv file to my personal computer, where I drew a graph based on the data in the csv file. For this, I decided to use Python and its matplotlib library, which I am familiar with.

Later, I realised that matplotlib is also capable of drawing three-dimensional graphs. I hence wrote some more Python code, in order to generate this 3-D graph.

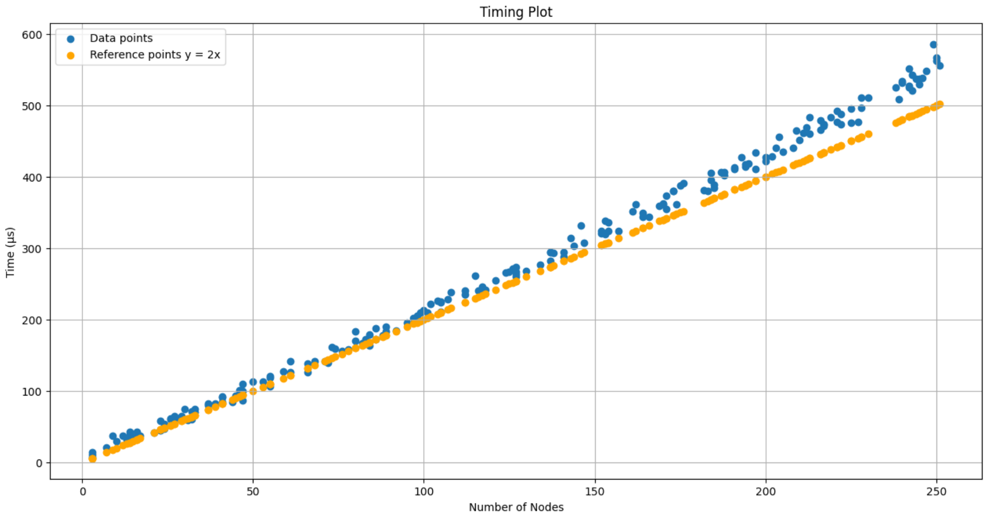
Both graphs are attached in the next section.

Empirical Observations

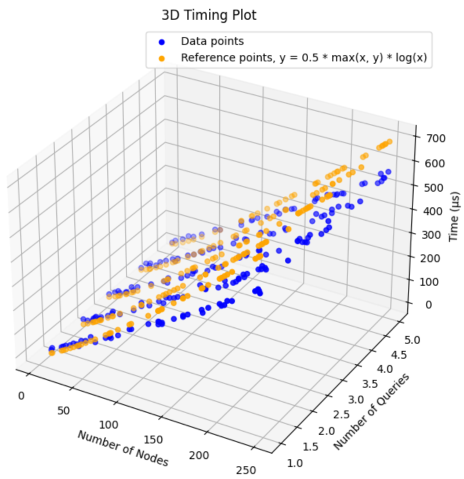
As mentioned in the above section, the time varies with respect to three variables:

1. Number of nodes (junctions) in the tree
2. Number of queries
3. Total number of parcels

The following time graph was obtained when the time was plotted against the number of nodes:



The following three-dimensional graph was obtained when the time was plotted against the number of nodes and number of queries:



Because we are unable to draw a four-dimensional graph, we are assuming that the number of parcels is constant. Furthermore, it was noticed that the number of parcels has a lesser effect on the overall time relative to the number of nodes and queries.

Additional Insights

Currently, we are using binary lifting, such that each lowest common ancestor problem is answered in $O(n \log n)$. If there are much more queries than nodes, i.e. $q >> n$, we can use Euler Tour and Range Minimum Queries such that each lowest common ancestor problem can be solved in constant time.

Another improvement that can be made is that we can modify the code such that k-ary trees are supported, instead of just binary trees.

Question 3

Solution Approach

The first step in the solution is to represent the network efficiently. The input is given as an edge list, and we convert it into an adjacency list so that we can quickly traverse all neighbors of a given node.

The next step is to determine travel times for both trucks. We will use Dijkstra's algorithm for this. However, an issue arises with this. The original (vanilla) Dijkstra's algorithm does not have support for the booster concept of the question. Hence, we need to modify the algorithm. In this modified algorithm, each state includes both the node and whether the booster has already been used. Thus, the distance array becomes dist[node][booster_state].

We will have to execute this modified Dijkstra algorithm twice:

1. Once from node 1 (the starting city of the first truck)
2. Once from node n (the starting city of the second truck)

When moving from a node to the next node, there are two cases:

1. The truck has not used its booster yet:
   (a) If the current node is a metro city, we allow the truck to apply the booster. After this, whenever we traverse an edge, we will cut down the travel time to half.
   (b) If the current city is not a metro city, we have no choice but to use the complete travel time.
2. The truck has used its booster: In this case, there is no point in checking if the current city is a metro city or not. We just continue to the next node, with half of the travel time.

At last, we can determine the meeting point of the trucks. Basically, we calculate the meeting time of the trucks at every single node. Because both trucks have to be there at the node to meet, the meeting time is basically max of the time each truck takes to get there. From this list, we basically identify the minimum time taken. This will be the fastest that the two trucks can meet at a node.

Time and Space Complexity Analysis

Time Complexity

First, let us analyze the program to determine its time complexity. The code can be divided into the following major sections, namely:

1. Building the graph by converting from list of edges to adjacency list.
2. Running the modified Dijkstra algorithm.
3. Finding the meeting time at each node, and taking the minimum of these times.

Let us now analyze the time complexity of each section. Let us assume that n = number of nodes (cities) and m = number of edges.

Section	Time Complexity	Reason
Section 1	$O(n+m)$	Construction of an adjacency list with n nodes and m edges requires $O(n+m)$ time.
Section 2	$O((n+m) \times \log n)$	Running Dijkstra algorithm on an adjacency list requires $O((n+m) \times \log n)$. Because we are running the Dijkstra algorithm twice, the actual time complexity is $O(2 \times (n+m) \times \log n)$. However, because we want to measure asymptotic time complexity, not absolute, we will simply ignore the constant term.
Section 3	$O(n)$	For each node (city), we will calculate the meeting time. In the same loop, we can also calculate the minimum meeting time.

From this, we can state that the asymptotic worst case complexity of the code is $O((n+m)+((n+m)\times\log n)+n)$. Because we know that $(n+m) = O((n+m)\log n)$ and $n = O((n+m)\log n)$, we can simplify the asymptotic worst case complexity of the code to $O((n+m)\log n)$. This can also be re-written as $O(max(m,n) \times \log n)$.

Space Complexity

There are three main data segments. Their space complexities are as follows:

1. Adjacency list - $O(n+m)$
2. Distance table - $O(n)$
3. Priority Queue - $O(n+m)$

The asymptotic space complexity of this algorithm can be considered as $O((n+m)+n+(n+m))$. This can also be re-written as $O(max(m,n))$.

Experimental Setup

I followed a similar setup as Question 1 and 2 for the experimental setup.

I first wrote the code on my personal computer. I had downloaded version 14 of the C++ compiler, and was using several new features provided in this version. This includes declaring variables with the auto keyword, and for-each loops. When I got access to the cluster, I realized that the cluster was still using C++ version 8.5. I then had to manually degrade the code, such that it can run on the cluster.
The code was then executed on the cluster provided. I created two files, namely q3_input.csv and q3_output.csv. I then wrote a program that does the following:

1. Reads one input from the q3_input.csv file.
2. Processes it using the function present in user_code.h.
3. Compares the output generated by the user_code.h file with the real output present in q2_output.csv.
4. Notes down the time taken with respect to the number of nodes in the q3_timings.csv file.

I then moved the q3_timings.csv file to my personal computer, where I drew two graphs based on the data in the csv file. One is time taken vs. number of nodes. The other is time taken vs. number of edges. For this, I decided to use Python and its matplotlib library, which I am familiar with. I wrote some additional Python code, in order to generate a 3-D graph.

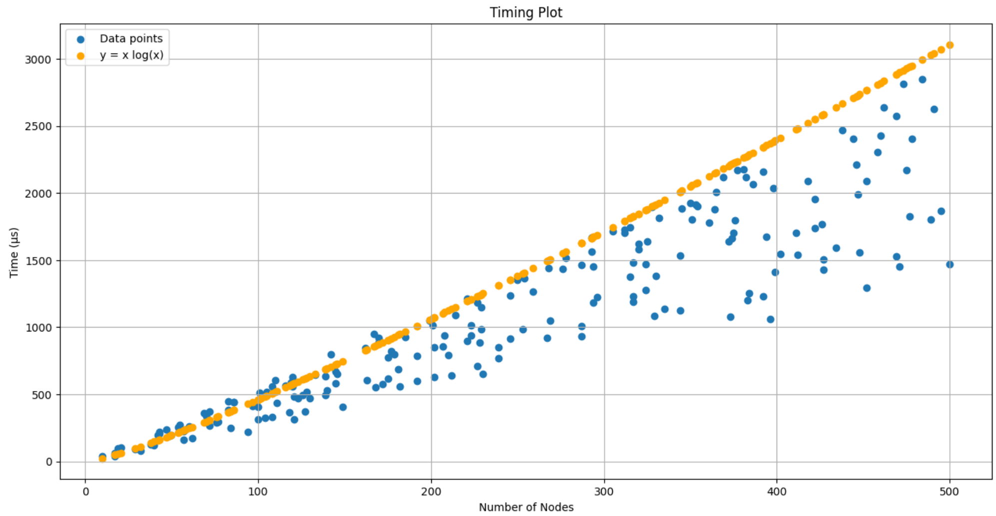
All three graphs are attached in the next section.

Empirical Observations

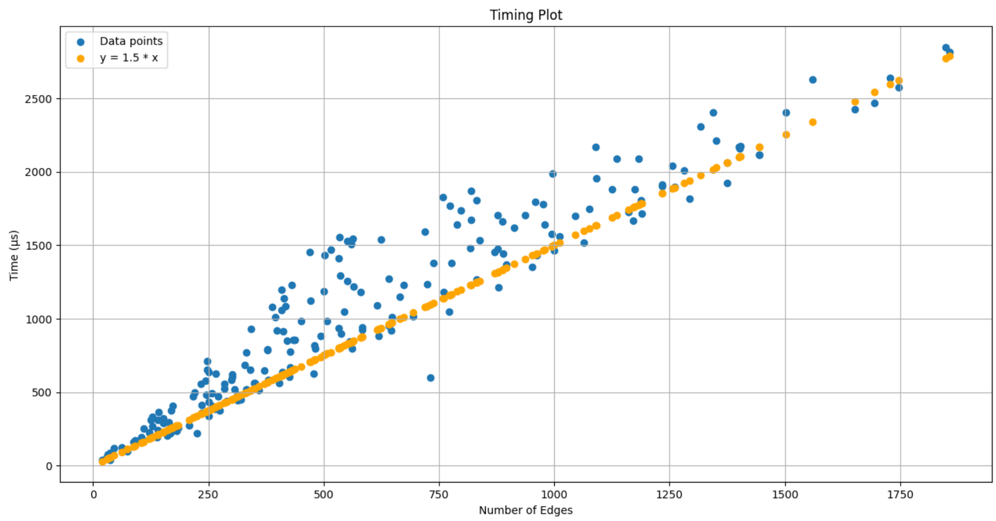
As mentioned in the above section, the time varies with respect to two variables:

1. Number of nodes (cities) in the graph
2. Number of edges

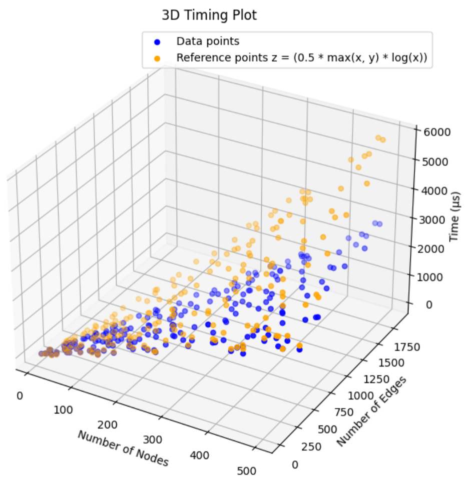
The following time graph was obtained when the time was plotted against the number of nodes (cities):



The following time graph was obtained when the time was plotted against the number of edges (roadway between cities):



The following three-dimensional graph was obtained when the time was plotted against the number of nodes and number of edges:



Additional Insights

If the algorithm is on an edge case, it can be further optimized to reduce the time. Some of the possible optimizations are as follows:

1. If the graph is disconnected, and the two trucks are on disconnected segments of the graph, there is no point in doing any calculation. The two trucks simply cannot meet, and we can return -1 instantly.
2. If the starting city of both trucks are metro cities, we can apply the boosters in the first step itself. After this, we do not need to accommodate for two different scenarios. Similarly, if none of the cities are metro cities, we can never apply the booster. Again, there shall be no need to accommodate two scenarios.