CS/CE 102/171: Data Structures & Algorithms
Spring 2025 - Homework 02

Department of Computer Science

Dhanani School of Science and Engineering

Habib University

Contents

Introduction

1.1 Instructions

- This homework will contribute 5% towards your final grade.
- The deadline for submission of this homework is 09:00 am on Monday, April 7, 2025.
- The course team will address any queries regarding this homework for **one week** following its release. Queries submitted after this period may not receive a response.
- The homework must be submitted online via CANVAS. You are required to submit a zip file that contains all the *.py* files.
- **Do not** modify the original file structure of the homework and **do not** change the names of the files shared with you. You may be penalized if such changes are observed.
- The zip file should be named as *hw2_aa12345.zip* where *aa12345* will be replaced with your student id.
- **Files that don't follow the appropriate naming convention will not be graded.**

1.2 Grading Rubric

Your assignment will be graded according to the rubric given in the Excel file `dsa-spring25-hw2-rubric.xlsx`

1.3 Homework Objectives

Use Divide and Conquer, Graph, and Hashtable ADTs and algorithms effectively and efficiently to solve problems.

1.4 Late submission policy

You are allowed to submit late till 9:00 am on Tuesday, April 8, 2025, with a 20% penalty. No submissions after this will be accepted.

1.5 Use of AI

Taking help from any AI-based tools such as ChatGPT is strictly prohibited and will be considered plagiarism.

1.6 Viva

Course staff may call any student for Viva to provide an explanation for their submission.

1.7 Queries

You can post your Queries on the Homework 02 Discussion on Canvas or visit course staff during their office hours.

1.8 Processing Text Files

For each question, you have to implement a `main(...)` function that takes a filename as an argument. The `main(...)` function should read the input from the file and call the required functions.

You can use the following code to read the input from a file:

```
with open('input.txt') as f:
    lines = f.readlines()

input = []
for line in lines:
    line = line.strip() # remove leading and trailing spaces
    tokens = line.split() # split the line into tokens
    input.append(tokens[0]) # add the first token to the
   input list

main(input) # call the main function
```

Questions

2.1 Divide and Conquer

2.1.1 Decrypting Enemy Warehouse Information

A spy has been gathering intelligence on enemy warehouse storage capacities. In a previous mission, another spy used the Karatsuba algorithm (see Section 2.1.5 for details) to transmit coordinate-based information, but the method was compromised, leading to their capture. The current spy has devised a modified encryption method based on the Karatsuba algorithm to avoid detection.

This time, instead of sending simple coordinates, the spy needs to transmit the dimensions of the enemy's warehouse: Length, Width, and Height. The encryption process involves recursively encoding these dimensions using a variation of the Karatsuba algorithm. Specifically:

- The Length and Width values are encrypted using the Karatsuba algorithm, forming recursive call trees similar to the original method.
- The Height value is also included, but it is adjusted by a factor of $10^{(\text{tree level})}$. At each level of recursion, the height is divided by $10^{(\text{level})}$ before being included in the encryption process.
- The final encrypted message consists of only the leaf nodes from the recursive calls, each represented as a tuple (x, y, h). Here, x and y are intermediate values derived from the Karatsuba algorithm, while h is the modified height value at that level.

  Level of a Node in a Tree: The level of a node in a tree is defined as the number of edges in the path from the node to the root. The root is at level 0.

Your task is to understand the modified Karatsuba encryption process and design a recursive algorithm to decrypt the original dimensions (Length, Width, and Height) of the warehouse from the given leaf nodes.

2.1.2 Function(s) Description

You have to provide the implementation of the following function:

- `reverse_karatsuba(data, level) -> tuple`: A recursive function that receives the leaf nodes as a nested list and returns a tuple containing the original warehouse dimensions (Length, Width, Height). The parameter `level` tracks the recursion depth to correctly reconstruct the Height value.
- `main(filename) -> list[tuple[int, int, int]]`: It receives the input file name as a parameter and returns a list of values where each value is the volume of the warehouse calculated using the decrypted dimensions (Length, Width, Height) for given trees in the file.

2.1.3 Input Format

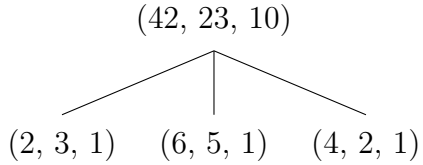
The first line of the input file is an integer N where $1 \leq N \leq 100$. For the next N lines, each line consists of a nested list, where each level of the tree is represented by a list. Each element in the list can be:

- A tuple (x, y, h), representing a leaf node containing encrypted dimensions.
- A nested list, representing a subtree, following the same structure as the outermost list.

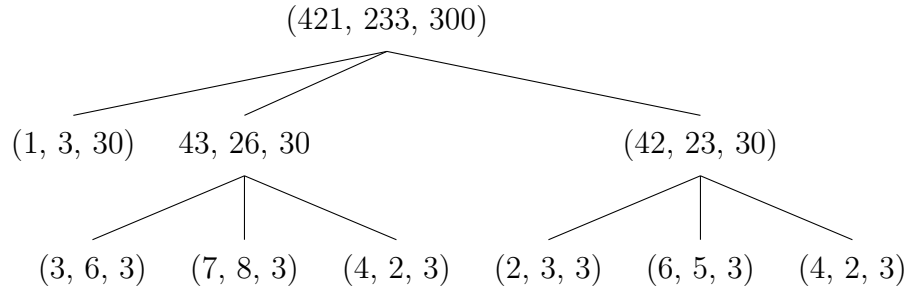
Sample Input

```
2
[(2, 3, 1), (6, 5, 1), (4, 2, 1)]
[(1, 3, 30), [(3, 6, 3), (7, 8, 3), (4, 2, 3)],
    [(2, 3, 3), (6, 5, 3), (4, 2, 3)]]
```

The following tree visualizes the calls of the Karatsuba algorithm for (42, 23, 10). The leaf nodes of this tree are (x=2, y=3, 1), (x=6, y=5, 1), and (x=4, y=2, 1). Note that x and y are calculated using Karatsuba algorithm where height is calculated by dividing the height by 10^level. The spy shares only this information. Your decryption algorithm must be able to generate the actual volume from these leaf nodes.



Similarly, the tree for the coordinates (421, 233, 300) is represented as follows:



2.1.4 Output

The output should return a list of N values, where each value is a tuple. The first element of the tuple is another tuple representing (L, W, H), while the second element is the volume of the warehouse.

The output of the above input file should be:

```
[((42, 23, 10), 9660) , ((421, 233, 300), 29427900)]
```

2.1.5 Karatsuba algorithm

The Karatsuba algorithm is a fast multiplication algorithm that efficiently multiplies two large numbers by dividing them into smaller parts, recursively solving sub-problems, and combining the results. It was discovered by Anatolii Alexeevitch Karatsuba in 1960 when he was a student.

The algorithm takes advantage of the divide-and-conquer strategy to reduce the number of arithmetic operations required for multiplication, especially for large numbers. Instead of directly multiplying two numbers digit by digit using the traditional "long multiplication" method, the Karatsuba algorithm breaks down the numbers into smaller parts, typically of equal size, and performs intermediate multiplications to compute partial products. These partial products are then combined to obtain the final result.

The key idea behind the Karatsuba algorithm is to exploit the fact that the product of two large numbers can be expressed as a sum of smaller products of their components. By recursively applying this approach to sub-problems, the algorithm achieves a significant reduction in the number of multiplications compared to traditional methods.

The Karatsuba algorithm can be described as follows:

1. Given two numbers x and y to be multiplied, express them as follows:

$$x = x_1 \cdot B^m + x_0$$
$$y = y_1 \cdot B^m + y_0$$

where B is the base of the number system (e.g., 10 for decimal numbers), and m is the number of digits in the smaller of the two numbers.

2. Compute the following intermediate products:

$$z_0 = x_0 \cdot y_0$$
$$z_1 = (x_1 + x_0) \cdot (y_1 + y_0)$$
$$z_2 = x_1 \cdot y_1$$

3. Compute the final result using the following formula:

$$x \cdot y = z_2 \cdot B^{2m} + (z_1 - z_2 - z_0) \cdot B^m + z_0$$

4. Recursively apply the Karatsuba algorithm to compute the intermediate products z_1 and z_2. The final result can be obtained by combining these intermediate products using the formula above.

5. The base case of the recursion occurs when the numbers are small enough to be multiplied directly using the traditional method.

6. The algorithm terminates when the base case is reached, and the final result is obtained by combining the partial products computed at each level of the recursion.

2.1.6 Pseudocode

The pseudocode for the Karatsuba algorithm can be expressed as follows:

```
    def karatsuba(x, y):
    if x < 10 or y < 10:
        return x * y

    n = max(len(str(x)), len(str(y)))
    n2 = n // 2

    high1, low1 = split(x, n2)
    high2, low2 = split(y, n2)

    z0 = karatsuba(low1, low2)
    z1 = karatsuba((low1 + high1), (low2 + high2))
    z2 = karatsuba(high1, high2)

    return (z2 * 10**(2 * n2)) + ((z1 - z2 - z0) * 10**n2) + z0

def split(num, n):
    high = num // 10**n
    low = num % 10**n
    return high, low
```

Credits: Muhammad Qasim Pasta
For queries, please reach out to Qasim Pasta and Nadia Nasir

2.2 Hash Tables

2.2.1 Stream Sphere Video History Manager

After graduating with a degree in Computer Science, you land your dream job at StreamSphere Co, a cutting-edge video streaming company. Your mentor, Alex Carter, assigns you an important task—designing a smart video history manager. Users watch videos, like or dislike them, and even remove them from history. You have to build a hashtable-powered solution that keeps StreamSphere ahead of the competition Your task is to read the watched Video Records from **watchedVideos.csv** and create a hash table representation from it. To test this new system you will be performing some Operations, which are mentioned in the text file **Operations1.csv**.

2.2.2 Function(s) Description

Following are the files which are provided to you,

- *watchedVideos.csv* (Input file containing Video' information)
- *Operations.csv* (Input file containing operations to be performed on hash table)
- *VideoHistoryHashTable.py* (Python file Containing helper functions for hash table)
- *VideoHistoryManager.py* (Python file for creating the hash table and performing operations on it)

You are required to implement the following functions in *VideoHistoryManager.py* file

- `def main(filename)` - Takes the input file (`watchedVideos.csv`) as argument and returns a list of Dictionaries in the following format: `[{'Video_ID': '88F7B33d2bcf9f5', 'Video_URL': 'https://StreamSphere.com/watch?v=88F7B33d2bcf9f5', 'Views': 664320, 'Likes': 43580, 'Dislikes': 1234}, ....]`
- `def create_VideoHistory(VideoRecords)`- Takes the list of Dictionaries from the main function and returns a hash table of size = 7 using the helper functions from `VideoHistoryHashTable.py`. Hash table is a list of dictionaries, and each dictionary contains two keys: **ID** and **DATA**. The Hash table is of the following format: `[{'ID': '1bA7A3dc874da3c', 'DATA': {'Video_URL': 'https://StreamSphere.com/watch?v=1bA7A3dc874da3c', 'Views': 860201, 'Likes': 11137, 'Dislikes': 784},...]}`. Note that the Views, Likes and Dislikes are integers.

- `def perform_Operations(H,operationFile)` - Takes the hash table and operations files (`Operations1.txt`) as input and performs all the operations on the hash table. It returns a dictionary of collision_path, and the resultant Hash table. The format of collision_path dictionary is mentioned in the output section.

You are required to implement the following functions in *VideoHistoryHashTable.py*

- `def resize_hashtable(hashtable,size,increase)` - Takes **Hashtable**, **size** and **increase** (a boolean variable). The purpose of this function is to resize (increase or decrease the size of) the hash table. If the value of **increase** variable is True, the size of the hash table will increase by multiplying it by 2 and extending it to the next Prime number. For example, 7 will become 7*2=14 and the nearest next prime number to number 14 will be 17 so this will be the new size of hash table.

  If the value of **increase** variable is False, the hash table size will be reduced to half and extended further to make it a Prime number. For example, if the size of the hash table was 17, then it will become 17//2= 8, and the nearest Prime number is 11 so it will be reduced to the size of 11.Note that the size of the hash table should not be less than 7.

- `def hash_function(key,size)` - It takes a **key** (Video_ID), and **size** (of the hash table) as inputs. The function will return the hash value by following these steps:

  1. Sum the ASCII equivalent of each character in the key
  2. Right shift 4 bits
  3. Return the modulo size for this new value and make sure the address remains non-negative in this process.

- `def collision_resolver(key,oldAddress,size)`- This function takes the **key**, **OldAddress** (the last address on which a collision occured), and the **size** of the hash table as inputs. It calculates and returns the new address using the Key Offset method using the following formula:

```
offset=Sum of Ascii Codes of key // size
address=((offset+oldAddress)%size)
```

- `def put(hashtable,key, data,size)` - The function takes **hashtable**, **key** to be inserted, the **data** associated with the key (a dictionary in this case) and **size** of the hash table as input values. This function also considers resizing/shrinking of hash table based on the load factor of **75%** and **30%**. When more than **75%** of hash table is already populated, increase the size, whereas decrease the size if it is less than **30%**. The function returns the hashtable and its size as a tuple.

- `def loadFactor(hashtable,size)` - Returns the load factor of the hashtable
- `def Update(hashtable,key, columnName, data,size,collision_path,opNumber)` It takes `hashtable`, `key` to be updated, the `ColumnName` whose value needs to be changed to `data`, a dictionary named `collision_path` and an integer `opNumber` as input values. `opNumber` is the sequence Number of the Operation being performed and this will act as a key to the `collision_path`. To see the format of `collision_path` see the output section.
- `def UpdateAtIndex(hashtable,index, columnName)` It takes `hashtable`, `index` to be updated, the `ColumnName` whose value needs to be changed to `data` as input values.
- `def get(hashtable,key,size,collision_path,opNumber)` returns the value associated with the key in the hashtable or prints "Item not found". This method also updates the collision_path.
- `def delete(hashtable, key, size,collision_path,opNumber)` - Deletes key from hashtable and places a tombstone marker #.

2.2.3 Input Format

There are 2 input files as mentioned earlier, the format of **input file** (`watchedVideos.csv`), is given below,

```
Video_ID;Video_URL;Views;Likes;Dislikes
88F7B33d2bcf9f5;https://StreamSphere.com/watch?v=88
   F7B33d2bcf9f5;664320;43580;1234
f90cD3E76f1A9b9;https://StreamSphere.com/watch?v=
   f90cD3E76f1A9b9;749879;46935;3456
```

The first row contains the Column names for the Video information to be stored in Video History Hashtable. The next rows are the values associated with these columns in the same order. You are provided the information of 10 Videos.

Operations1.csv file has the following format

```
Watch DbeAb8CcdfeFC2c;https://fakevideo.com/watch?v=
   DbeAb8CcdfeFC2c;653568;32921;24
Like 1bA7A3dc874da3c
Dislike 8e4FB470FE19bF0
Delete 8e4FB470FE19bF0
```

Sample operations format

- **Watch DbeAb8CcdfeFC2c;**
  https://fakevideo.com/watch?v=DbeAb8CcdfeFC2c;653568;32921;24
  This will find the ID = DbeAb8CcdfeFC2c from the hashtable. If it is found in the video history it increases the view count by 1. If it is not found, add it to the history and increase the view count. Also print the Video Info associated with this ID (in the form of a Dictionary).
- **Like 1bA7A3dc874da3c**
  This will find the ID = 1bA7A3dc874da3c in the hashtable and increase the likes of this video by 1. This operation assumes that the ID is present in the Hashtable.
- **Dislike 8e4FB470FE19bF0**
  This will find the ID = 8e4FB470FE19bF0 in the hashtable and increase the dislikes of this video by 1. This operation assumes that the ID is present in the Hashtable.
- **Delete 2EFC6A4e77FaEaC**
  This will delete the Video with ID= 2EFC6A4e77FaEaC from the Video History Hashtable.

2.2.4 Output

Besides intermediate printing of statements mentioned in the function description section, under each function the `perform_Operations()` will maintain a dictionary called `collision_path` and returns it. The purpose of this dictionary is to maintain a log of collision path for each Operation.
The dictionary should look like this:

```
{1: [15], 2: [12], 3: [13, 6], 4: [13, 6], 5: [13, 6], 6: [13,
6], 7: [13, 6], 8: [9], 9: [14], 10: [13, 5], 11: [7], 12: [5],
13: [0]}
```

Keys of this dictionary are the operation number in the order mentioned in *Operation.txt*, the values are the list of indexes on which collisions occur.

Credits: Saba Saeed
For queries, please reach out to Saba Saeed and Amber Altaf

2.3 Graphs

SwiftMart Corporation is one of the largest retail chains in the country, having an extensive supply chain network consists of multiple warehouses and hundreds of retail stores spread across different countries around the world. To ensure efficient operations, the company must carefully manage shipping routes, delivery times, and costs to keep store shelves stocked while maintaining profitability. Every day, thousands of orders are processed, and SwiftMart's logistics team must make strategic decisions on how to distribute inventory from one warehouse to another. The task is to create a graph network (using Adjacency Maps, as done in labs) for their supply chain.

2.3.1 Dataset Description

SwiftMart Logistics data set is defined in the given CSV file called `supply_chain.csv`, which contains the following columns:

- **Outbound Nexus** – This represents the warehouse or distribution center where the shipment originates. Each warehouse is uniquely identified by an alphanumeric code (e.g., W1, W10).
- **Inbound Hub** – This represents the destination warehouse, retail store, or distribution center where the shipment is sent. It is also uniquely identified by an alphanumeric code (e.g., W11, W16).
- **Shipment Time** – The estimated number of days (an integer value) required for a shipment to travel from the outbound nexus (source) to the inbound hub (destination).
- **Shipment Cost** – The total cost of transporting the shipment from the outbound nexus to the inbound hub, measured in U.S. dollars (a floating-point value). This cost includes transportation expenses such as fuel, labor, handling, and potential tariffs.
- **Shipping Method** – The mode of transportation used for shipping (a string value), such as:
  - **Air**: Fast but expensive.
  - **Sea**: Cost-effective for large shipments but slower.
  - **Ground**: Includes trucking and rail, balancing cost and speed.
- **Transit Trust Ratings** – A percentage value (a floating-point value) representing the reliability of the Outbound Nexus (warehouse) based on its historical shipping performance. The rating is expressed as a percentage and reflects factors such as on-time deliveries, consistency in shipping durations, past disruptions, and overall efficiency.

The given dataset will produce a graph that will be used for all the following problems.

2.3.2 Graph Construction

Q1) Define the following functions to generate the required graph:

1. `list_of_vertices(filename)` – Takes the filename of the dataset as an argument and generates a list of vertices. Each vertex is defined by a tuple that contains the warehouse ID and the Transit Trust Ratings for that warehouse. **Please note that the Transit Trust Ratings are defined for all warehouses in the supply chain based on historical shipping performance. However, this dataset only records values for Outbound Nexus Warehouses.**

   For the given dataset, the list of vertices would result in the following:

   ```
   [('W1', 76.14), ('W10', 94.39), ('W11', 70.81), ('W12', 76.02),
   ('W13', 73.13), ('W14', 93.7), ('W15', 86.31), ('W16', 73.24),
   ('W17', 82.5), ('W18', 94.33), ('W19', 91.9), ('W2', 78.09),
   ('W20', 89.33), ('W3', 82.95), ('W4', 73.83), ('W5', 87.41),
   ('W6', 77.81), ('W7', 86.43), ('W8', 86.2), ('W9', 91.85)]
   ```

2. `list_of_edges(filename)` – Takes the filename of the dataset as an argument and generates a list of edges. The edges are defined between every Outbound Nexus and Inbound Hub warehouses. Make sure you follow the correct data types for each of these values.

   For the given dataset, the list of edges would result in the following:

   ```
   [('W1', 'W11', 2, 757.96, 'Sea'), ('W1', 'W4', 11, 913.21, 'Air'),
   ('W10', 'W16', 1, 418.59, 'Sea'), ('W10', 'W11', 9, 556.73, 'Sea'),
   ('W11', 'W1', 5, 339.96, 'Ground'), ('W12', 'W7', 9, 582.59, 'Ground'), ...]
   ```

   Please note, the list of edges does not include the Transit Trust Ratings, as they are characteristics of the vertices and not the edges. This is only a sample; your program should process all edges in the dataset.

3. `create_supply_chain(filename)` – that takes the filename of the dataset as an argument. It uses this filename to generate the list of vertices first, by using the helper function defined above (`list_of_vertices`); as well as to generate the list of edges, by using the helper function defined above (`list_of_edges`). The function then uses these lists to create the supply chain network, which is a weighted graph defined as Adjacency Map, as follows:

   - The keys are represented by the vertices, which are a tuple as generated before

- The values are tuples representing the other end point, and edge weight. Each edge weight is defined as a tuple containing three values: (Shipment Time, Shipment Cost, Shipping Method).

For the given dataset, the graph will result in the following:

```
{('W1', 76.14): [('W11', (2, 757.96, 'Sea')), ('W4', (11, 913.21, 'Air'))],
('W10', 94.39): [('W16', (1, 418.59, 'Sea')), ('W11', (9, 556.73, 'Sea'))],
('W11', 70.81): [('W1', (5, 339.96, 'Ground'))],
('W12', 76.02): [('W7', (9, 582.59, 'Ground')), ('W9', (8, 485.8, 'Sea')),
('W2', (12, 974.87, 'Ground'))],
('W13', 73.13): [('W20', (6, 195.26, 'Sea')),
('W17', (15, 284.62, 'Ground')),('W6', (12, 552.92, 'Ground')),
('W14', (11, 536.73, 'Sea')), ('W12', (2, 661.32, 'Sea'))], ...}
```

Please note that this is not the complete graph for the given dataset. Only part of it is shown to define how the graph must be defined by your program. However, it should be done for all the data. The function then returns the respective graph at the end.

2.3.3 Warehouse Connectivity

Q2) Define the following functions to get information on connected warehouses in the supply chain:

1. `get_warehouse_connections(graph, warehouse, option)` – that takes three arguments: **graph** which is the graph created in Q1, **warehouse** which is a string representing the alphanumeric warehouse ID part of the vertex, and **option** which is a character: "i" for inbound supply, "o" for outgoing supply, and "b" for both inbound and outbound connections. The function then finds all the connected vertices based on the option selected and returns the connected warehouses from the supply chain without the repeating values, in the form of a list. If there is no connection to or from this warehouse, the result will be an empty list.

   - For example, if the function is called for warehouse ID = "W1" with option = 'o', then the list of connected warehouses will be: [**'W11', 'W4'**]
   - Similarly, if the function is called for the same warehouse ID, i.e., "W1" but with option = 'i', then the list of connected warehouses will be: [**'W11', 'W14', 'W5'**]
   - And if this function is called for warehouse ID "W1" with option = 'b', then the list of connected warehouses will be: [**'W11', 'W4', 'W14', 'W5'**]. Please note that for option = 'b', it ignores repeated warehouse connections.

2. `get_number_of_warehouse_connections(graph, warehouse, option)` – that takes three arguments: **graph** which is the graph created in Q1, **warehouse** which is a string representing the alphanumeric warehouse ID part of the vertex, and **option** which is a character: "i" for inbound supply, "o" for outgoing supply, and "b" for both inbound and outbound connections. The function then calculates the total number of connected warehouses depending on the option selected and returns that number. If any of the connecting warehouse is being repeated, it is only counted once. If there are no connections or the warehouse does not exist, then the function returns 0.

    - For example, if the function is called for warehouse ID = "W1" with option = 'o', then the list of connected warehouses will be: [**'W11', 'W4'**], and this function will return the count as 2
    - Similarly, if the function is called for the same warehouse ID, i.e., "W1" but with option = 'i', then the list of connected warehouses will be: [**'W11', 'W14', 'W5'**], and this function will return the count as 3
    - And if this function is called for warehouse ID "W1" with option = 'b', then the list of connected warehouses will be: [**'W11', 'W4', 'W14', 'W5'**], and this function will return the count as 4

2.3.4 Warehouse Relationship Analysis

Q3) Define the following functions to find information about any two specific warehouses connected with each other:

1. `list_of_incident_edges(graph, warehouse)` – that takes two arguments: **graph** which is the graph created in Q1 and **warehouse** which is a string representing the alphanumeric warehouse ID part of the vertex. The function then returns the list of incident edges to the vertex defined by the argument, warehouse.

    - For example, if the function is called for warehouse ID = "W1", the list of incident edges will be the outgoing edges of this vertex, resulting in the following list: [**('W1', 'W11', 2, 757.96, 'Sea'), ('W1', 'W4', 11, 913.21, 'Air')**]
    - Please note, the list of edges does not include the Transit Trust Ratings, as they are the characteristics of the vertices, and not the edges.

    If a particular warehouse does not have incident edges, it returns an empty list.

2. `is_connected(graph, warehouse_o, warehouse_i)` – that takes three arguments: **graph** which is the graph created in Q1, **warehouse_o** which is a string representing the alphanumeric warehouse ID part of the origin vertex, and **warehouse_i** which is a string representing the alphanumeric warehouse ID part of the destination vertex. The function then searches the

graph to find if the two warehouses are connected to each other or not. It returns True if there is a direct connection between them, and False otherwise. Remember, a shipment cannot originate and end at the same warehouse, so that connection will be considered False. For example, if this function is called for warehouse_o = 'W1' and warehouse_i = 'W11', then it will return True, as there is a connection from 'W1' to 'W11' in the graph.

3. `if_disconnected(graph, warehouse_o, warehouse_i)` – that takes three arguments: graph which is the graph created in Q1, warehouse_o which is a string representing the alphanumeric warehouse ID part of the origin vertex, and warehouse_i which is a string representing the alphanumeric warehouse ID part of the destination vertex. The function first checks if the connection between the two warehouses exists using the is_connected function above. If it does, then assume that this edge is temporarily not available, and find out the fastest connection (based on Shipment time) using the same mode of Shipping method. If there is no such warehouse_o in the dataset, it returns None. If there is no connection of the same shipping method left after ignoring the first one, then the function returns -1.

   For example, if this function is called for warehouse_o = "W15" and warehouse_i = "W5", the mode of shipping between the two warehouses is by "Sea". The function then looks for the fastest alternate edge that uses the same Shipping Method, i.e., by "Sea". That is, the list of edges for "W15" are given as follows:

   - `[('W15', 'W5', 3, 320.43, 'Sea'),`
     `('W15', 'W19', 1, 596.11, 'Sea'),`
     `('W15', 'W10', 15, 368.41, 'Ground'),`
     `('W15', 'W4', 2, 654.81, 'Ground'),`
     `('W15', 'W12', 7, 509.52, 'Sea')]`

   - Apart from "W5", there are two other outgoing edges that uses the Shipping method of "Sea", i.e., [**('W15', 'W19', 1, 596.11, 'Sea'), ('W15', 'W12', 7, 509.52, 'Sea')**]
   - The alternate faster edge will then be **('W15', 'W19', 1, 596.11, 'Sea')**, as this has the smallest shipping time, i.e., the fastest connection

2.3.5 Route Analysis and Graph Modifications

Q4: Define the following functions to get information regarding the connected supply route between any two warehouses in the graph:

1. `get_supply_route_details(graph, warehouse_o, warehouse_i, option)` – which takes four arguments: **graph** which is the graph created in Q1, **warehouse_o** which is a string representing the alphanumeric warehouse ID part of the origin vertex, **warehouse_i** which is a string representing

the alphanumeric warehouse ID part of the destination vertex, and **option** which is a string with range of values as either "time", "cost", or "mode" to get the respective details. The function returns the respective edge weight value, that is, if "time" is selected, it returns the Shipment Time; if "cost" is selected, it returns the Shipment Cost; and if "mode" is selected, it returns the Shipping Method between the origin and destination warehouses. If the origin warehouse is not in the graph, it returns None. If the destination warehouse is not connected to the origin warehouse, it returns -1.

- For example, if the function is called for warehouse_o = "W1" and warehouse_i = "W11" with option = "time": as a connection does exist between the two, so it will return the Shipment time of that connection i.e., **2**, which is the number of days
- If the function is called for warehouse_o = "W1" and warehouse_i = "W11" with option = "cost": as a connection does exist between the two, so it will return the Shipment Cost of that connection i.e., **757.96** in US dollars.
- If the function is called for warehouse_o = "W1" and warehouse_i = "W11" with option = "mode": as a connection does exist between the two, so it will return the Shipping Method of that connection i.e., by **"Sea"**.
- If the function is called for warehouse_o = "W34" and warehouse_i = "W4" with any option irrespectively, as warehouse W34 does not exist in the graph, it will return **None**.
- Similarly, if the function is called for warehouse_o = "W1" and warehouse_i = "W13" with any option irrespectively, as there is no direct connection between these valid warehouses, it will return **-1**.

Q5: Define the following functions to analyze the supply route between any two warehouses in the graph:

1. `get_cheapest_outgoing_supply_route(graph, warehouse)`
2. `get_expensive_outgoing_supply_route(graph, warehouse)`

- Both these functions take two arguments: **graph**, which is the graph created in Q1, and **warehouse** which is a string representing the alphanumeric warehouse ID part of the outgoing vertex.
- Each of these functions searches its incident edges and/or connections to find the cheapest/expensive outgoing supply route from the given warehouse, and returns a tuple of origin warehouse and destination warehouse. If there's no such edge, then it returns None
- For example, if this function is called for warehouse = "W14", the cheapest outgoing route from W14 will return the tuple: **('W14', 'W17')**, and the expensive outgoing route will return: **('W14', 'W1')**

- If any of these functions are called for warehouse ID = "W24" – as W24 does not even exist in the graph, both the functions would return **None**

3. `get_cheapest_incoming_supply_route(graph, warehouse)`

4. `get_expensive_incoming_supply_route(graph, warehouse)`

    - Both these functions take two arguments: **graph**, which is the graph created in Q1, and **warehouse** which is a string representing the alphanumeric warehouse ID part of the outgoing vertex.
    - Each of these functions searches its incident edges and/or connections to find the cheapest/expensive incoming route for the given warehouse, and returns a tuple of origin warehouse and destination warehouse. If there's no such edge, then it returns None
    - For example, if this function is called for warehouse = "W1", the cheapest incoming route for W1 will return the tuple: **('W11', 'W1')**, and the expensive incoming route will return: **('W14', 'W1')**
    - If any of these functions are called for warehouse ID = "W24" – as W24 does not even exist in the graph, both the functions would return **None**

Q6: Adding a connection between existing warehouses:

1. `add_supply_link(graph, warehouse_o, warehouse_i, weight)` – which takes four arguments: **graph** which is the graph created in Q1, **warehouse_o** which is a string representing the alphanumeric warehouse ID part of the origin vertex, **warehouse_i** which is a string representing the alphanumeric warehouse ID part of the destination vertex, and **weight** which is a tuple representing the weight of this edge comprised of (Shipment Time, Shipment Cost, Shipping Method). This creates a direct connection between the **warehouse_o** and **warehouse_i** warehouses, adds the weight to the connection, and prints a message stating **"Supply link added successfully"**. If any of the origin or destination warehouses are not present in the supply chain, it should print a message stating that the particular warehouse is not in the supply chain. If the connection already exists, it just updates the weight of the connection, and prints a message stating **"Supply link updated successfully"**

    - For example, if the function is called for warehouse_o = "W1" and warehouse_i = "W5" – there is no connection between the two warehouses from before, so it adds a new connection and adds the given weight tuple to it in the graph, and prints the message **"Supply link added successfully"**. The list of incident edges may be printed to see the updated result.

- If the function is called for warehouse_o = "W1" and warehouse_i = "W11" – there is already a connection between the two warehouses from before, so it only updates the connection with the given weight tuple in the graph, and prints the message **"Supply link updated successfully"**. The list of incident edges may be printed to see the updated result.
- If the function is called for warehouse_o = "W24" and warehouse_i = "W11" – as there is no such node as "W24" in the graph, it will print an error message **"W24 is not in the supply chain"**, and the graph stays the same

Q7: Adding a new warehouse with at least one connection with one of the other existing destinations:

1. `add_warehouse(graph, warehouse_o, tr_trust, warehouse_i, weight)` – which takes five arguments inputs: **graph** which is the graph created in Q1, **warehouse_o** which is a string representing the alphanumeric warehouse ID part of the origin vertex, **tr_trust** which is the transit trust percentage of this new warehouse, **warehouse_i** which is a string representing the alphanumeric warehouse ID part of the destination vertex, and **weight** which is a tuple representing the weight of this edge comprised of (Shipment Time, Shipment Cost, Shipping Method). If the warehouse_o is already in graph, just print a message stating that the warehouse already exists. If the destination warehouse, i.e., warehouse_i does not exist, the new warehouse cannot be added, so it prints an error message stating **"The destination warehouse does not exist"**. Otherwise, it creates a new entry in the graph for the new warehouse with the given transit trust percentage and an initial connection to another warehouse, and prints a success message **"The warehouse was added successfully"**

    - For example, if the function is called for warehouse_o = "W21" with all the other parameters having valid values, it will create a new entry in the graph with one new single connection, and print the message: **"The warehouse was added successfully"**
    - However, if the function is called for warehouse_o = "W1" with any of the parameter values, it will print an error message stating **"The warehouse already exists"**
    - Also, if the function is called for warehouse_i = "W25" with any parameter values, it will print an error message stating **"The destination warehouse does not exist"**

Q8: Removing any connections between warehouses:

1. `remove_supply_link(graph, warehouse_o, warehouse_i)` – which takes three arguments: **graph** which is the graph created in Q1, **warehouse_o** which is a string representing the alphanumeric warehouse ID part of the

origin vertex, and **warehouse_i** which is a string representing the alphanumeric warehouse ID part of the destination vertex. The function simply removes the connection originating from warehouse_o to warehouse_i, and prints a success message, stating: **"Supply link removed successfully"**. If any of the origin or destination warehouses are not present in the supply chain, it should print a message stating that the particular warehouse is not present in the supply chain. If the origin is not linked with destination warehouse, this means there is no connection to be removed, so print an appropriate error message (See skeleton files for exact messages)

- For example, if the function is called for warehouse_o = "W1" and warehouse_i = "W11" – as there is an existing connection between the two warehouses, which means it can be removed. Hence, it is removed from the graph, while leaving all the other connections intact, and prints the message: **"Supply link removed successfully"**
- If the function is called for warehouse_o = "W24" – as there is no such warehouse ID in the graph, it should print an error message stating: **"W24 is not in the supply chain"**
- If the function is called for warehouse_o = "W1" and warehouse_i = "W5", as there is no direct connection between these valid vertices, it will print the error message: **"W5 is not linked to W1"**

Q9: Removing the warehouse itself:

1. `remove_warehouse(graph, warehouse)` – which takes two arguments: **graph** which is the graph created in Q1, and **warehouse** which is a string representing the alphanumeric warehouse ID part of the vertex to be removed. The function removes the warehouse passed as argument, and all its connections from the graph if it exists, and prints a success message. If the given warehouse is not present in the network, it should print a message stating that the particular warehouse is not in the supply chain.

    - For example, if the function is called for warehouse = "W1" – as W1 has three edges, which will be removed first using remove_supply_link function from the previous question, which will print its success messages, followed by removal of the warehouse "W1" itself from the graph, and then prints its success message, as follows: **"Supply link removed successfully"" "Supply link removed successfully"" "Supply link removed successfully"" "W1 is successfully removed from the supply chain""**
    - If the function is called for warehouse = "W24" – as there is no such warehouse ID in the graph, it should print an error message stating: **"W24 is not in the supply chain"**

2.3.6 Graph Traversal and Analysis

Q10: Traversing the supply chain

1. `count_all_supply_chain_connections(graph, warehouse_o, warehouse_i)` – which takes three arguments: **graph** which is the graph created in Q1, **warehouse_o** which is a string representing the alphanumeric warehouse ID part of the origin vertex, and **warehouse_i** which is a string representing the alphanumeric warehouse ID part of the destination vertex. The function then finds all the routes connecting the origin warehouse to the destination warehouse, as nested list, using Depth First Search, and returns the count of all connection paths. If the origin or destination warehouse does not exist in the network, it returns **None**. If both warehouse_o and warehouse_i refer to the same warehouse, then the count should be 0.

Q11: Warehouse progress

1. `find_bottlenecks(graph, threshold, option)` – which takes three arguments: **graph** which is the graph created in Q1, **threshold** which is a valid threshold value, and **option** which is either "time" or "cost". This function finds all the possible warehouses that are causing the bottleneck in the progress, i.e., the average value is above the given threshold. Bottlenecks can be created in terms of delay in shipping, i.e., due to Shipment Time, as well as Shipment Cost. If the user enters option as "time" in the function, it returns a list of warehouses that are causing the bottleneck in terms of Shipment Time, and if the choice is "cost", it returns a list of warehouses that are causing the bottleneck in terms of Shipment Cost. If there is no bottleneck found, it returns an empty list.

    - For example, if this function is called with threshold = 5 and option = "time", it will check for warehouses whose average Shipment Time is beyond the threshold, which will give us the following result: [**'W1', 'W12', 'W13', 'W14', 'W15', 'W16', 'W17', 'W18', 'W19', 'W2', 'W20', 'W3', 'W4', 'W5', 'W6', 'W7', 'W8', 'W9'**]
    - Similarly, if the function is called with threshold = 330.00 and option = "cost", it will return the warehouses whose average Shipment Cost is above 330.00: [**'W1', 'W10', 'W11', 'W12', 'W13', 'W14', 'W15', 'W16', 'W17', 'W18', 'W19', 'W2', 'W20', 'W3', 'W4', 'W5', 'W6', 'W7', 'W9'**]

Q12: Calculating and understanding graph density

Graph Density is a measure of how well connected a graph is connected to the maximum possible connections it can have. It is defined for undirected graphs as:

$$GD = \frac{2E}{V(V-1)}$$

where GD = Graph Density, E = total number of edges in the graph and V = total number of vertices in the graph.

For directed graph, however, it is slightly different, which is given as:

$$GD = \frac{E}{V(V-1)}$$

This is because a directed graph can have $V(V-1)$ edges at most.

To do this define the following functions:

1. `total_edges(graph)` which takes one argument: **graph** which is the graph created in Q1, and returns the total count of edges in the graph
2. `total_vertices(graph)` which takes one argument: **graph** which is the graph created in Q1, and returns the total count of vertices in the graph
3. `graph_density(graph)` – which takes one argument: **graph** which is the graph created in Q1, calculates the graph density. Based on the graph density, it deduces the following and prints the output:
    - If $GD = 0 \rightarrow$ The graph has no edges and it is completely disconnected, hence it prints the message stating **"Disconnected Graph"**
    - If $GD = 1 \rightarrow$ The graph is supposed to be a complete graph, which is fully connected, hence it prints a message stating **"Fully Connected Graph"**
    - If $GD > 0.5 \rightarrow$ The graph is considered relatively dense, hence it prints a message stating **"Relatively Dense Graph"**
    - If $GD < 0.5 \rightarrow$ The graph is considered relatively sparse, hence it prints a message stating **"Relatively Sparse Graph"**

Q13: Warehouse Rankings

1. `find_best_warehouse(graph)` – which takes one argument: **graph** which is the graph created in Q1, and finds the top performing warehouse, i.e., the warehouse with the highest transit trust factor. If the graph is empty, returns -1.
2. `find_worst_warehouse(graph)` – which takes one argument: **graph** which is the graph created in Q1, and finds the worse performing warehouse, i.e., the warehouse with the lowest transit trust factor.

Credits: Maria Samad

For queries, please reach out to Maria Samad and Unaiza Ahsan