Python Task 1

In [26]:
```
import pandas as pd
```

In [27]:
```
pip install --upgrade python-dateutil
```

```
Requirement already satisfied: python-dateutil in s:\soft\anaconda\lib\site-packages (2.8.2)
Requirement already satisfied: six>=1.5 in s:\soft\anaconda\lib\site-packages (from python-dateutil) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

Question 1: Car Matrix Generation

Under the function named generate_car_matrix write a logic that takes the dataset-1.csv as a DataFrame. Return a new DataFrame that follows the following rules:

values from id_2 as columns values from id_1 as index dataframe should have values from car column diagonal values should be 0.

In [3]:
```
df=pd.read_csv("dataset-1.csv")
```

In [9]:
```
df
```

Out[9]:

	id_1	id_2	route	moto	car	rv	bus	truck
0	829	827	1	2.05	4.14	4.14	10.1	15.2
1	829	821	4	6.63	13.26	13.26	32.4	48.5
2	829	804	7	14.41	28.92	28.92	64.7	97.0
3	829	822	6	5.90	11.81	11.81	28.8	43.2
4	829	826	9	2.87	5.81	5.81	14.2	21.2
...	...	...	...	...	...	...	...	...
336	803	802	3	1.70	3.40	3.40	6.9	10.3
337	803	805	4	3.00	6.00	6.00	12.0	17.9
338	803	825	3	11.59	23.28	23.28	50.1	75.2
339	803	806	9	3.80	7.70	7.70	15.3	23.0
340	803	830	1	16.18	32.47	32.47	72.6	108.8

341 rows × 8 columns

In [10]:

```python
import pandas as pd


def generate_car_matrix(df):
    # Pivot the DataFrame to create a matrix with id_1 as index, id_2 as columns, and car as values
    car_matrix = df.pivot(index='id_1', columns='id_2', values='car')

    # Fill NaN values with 0 (for cells where id_1 and id_2 do not intersect)
    car_matrix = car_matrix.fillna(0)

    # Set diagonal values to 0
    for idx in car_matrix.index:
        car_matrix.loc[idx, idx] = 0

    return car_matrix

# Example usage:
car_matrix_result = generate_car_matrix(df)
print(car_matrix_result)
```

id_2	801	802	803	804	805	806	807	808	809	821	\
id_1											
801	0.00	2.80	6.00	7.70	11.70	13.40	16.90	19.60	21.00	23.52	
802	2.80	0.00	3.40	5.20	9.20	10.90	14.30	17.10	18.50	20.92	
803	6.00	3.40	0.00	2.00	6.00	7.70	11.10	13.90	15.30	17.72	
804	7.70	5.20	2.00	0.00	4.40	6.10	9.50	12.30	13.70	16.12	
805	11.70	9.20	6.00	4.40	0.00	2.00	5.40	8.20	9.60	12.02	
806	13.40	10.90	7.70	6.10	2.00	0.00	3.80	6.60	8.00	10.42	
807	16.90	14.30	11.10	9.50	5.40	3.80	0.00	2.90	4.30	6.82	
808	19.60	17.10	13.90	12.30	8.20	6.60	2.90	0.00	1.70	4.12	
809	21.00	18.50	15.30	13.70	9.60	8.00	4.30	1.70	0.00	2.92	
821	23.52	20.92	17.72	16.12	12.02	10.42	6.82	4.12	2.92	0.00	
822	24.67	22.07	18.87	17.27	13.17	11.57	7.97	5.27	4.07	1.80	
823	26.53	23.93	20.73	19.13	15.03	13.43	9.83	7.13	5.93	3.67	
824	27.92	25.32	22.12	20.52	16.42	7.80	11.22	8.52	7.32	5.06	
825	29.08	26.48	23.28	21.68	17.58	15.98	12.38	9.68	8.48	6.22	
826	30.87	28.27	25.07	23.47	19.37	17.77	14.17	11.47	10.27	8.01	
827	32.53	29.93	26.73	25.13	21.03	19.43	15.83	13.13	11.93	9.43	
829	36.32	33.72	30.52	28.92	24.82	23.22	19.62	16.92	15.72	13.26	
830	38.27	35.67	32.47	30.87	26.77	25.17	21.57	18.87	17.67	15.17	
831	39.24	36.64	33.44	31.84	27.74	26.14	22.54	19.84	18.64	16.15	

id_2	822	823	824	825	826	827	829	830	831
id_1									
801	24.67	26.53	27.92	29.08	30.87	32.53	36.32	38.27	39.24
802	22.07	23.93	25.32	26.48	28.27	29.93	33.72	35.67	36.64
803	18.87	20.73	22.12	23.28	25.07	26.73	30.52	32.47	33.44
804	17.27	19.13	20.52	21.68	23.47	25.13	28.92	30.87	31.84
805	13.17	15.03	16.42	17.58	19.37	21.03	24.82	26.77	27.74
806	11.57	13.43	14.82	15.98	17.77	19.43	23.22	25.17	26.14
807	7.97	9.83	11.22	12.38	14.17	15.83	19.62	21.57	22.54
808	5.27	7.13	8.52	9.68	11.47	13.13	16.92	18.87	19.84
809	4.07	5.93	7.32	8.48	10.27	11.93	15.72	17.67	18.64
821	1.80	3.67	5.06	6.22	8.01	9.43	13.26	15.17	16.15
822	0.00	2.21	3.60	4.76	6.55	8.00	11.81	13.74	14.68
823	2.21	0.00	1.79	2.94	4.74	6.15	10.00	11.89	12.87
824	3.60	1.79	0.00	1.71	3.50	4.92	8.77	10.66	11.64
825	4.76	2.94	1.71	0.00	2.20	3.65	7.46	9.35	10.33
826	6.55	4.74	3.50	2.20	0.00	2.05	5.81	7.71	8.69
827	8.00	6.15	4.92	3.65	2.05	0.00	4.14	6.06	7.04
829	11.81	10.00	21.40	7.46	5.81	4.14	0.00	2.38	3.36
830	13.74	11.89	10.66	0.00	7.71	6.06	2.38	0.00	1.39
831	14.68	12.87	11.64	10.33	8.69	7.04	3.36	1.39	0.00

Question 2: Car Type Count Calculation

Create a Python function named get_type_count that takes the dataset-1.csv as a DataFrame. Add a new categorical column car_type based on values of the column car:

low for values less than or equal to 15, medium for values greater than 15 and less than or equal to 25, high for values greater than 25. Calculate the count of occurrences for each car_type category and return the result as a dictionary. Sort the dictionary alphabetically based on keys.

In [11]:
```
def get_type_count(df):
    # Add a new column 'car_type' based on the conditions
    df['car_type'] = pd.cut(df['car'], bins=[-float('inf'), 15, 25, float('inf')],
                            labels=['low', 'medium', 'high'], right=False)

    # Calculate the count of occurrences for each car_type category
    type_count = df['car_type'].value_counts().to_dict()

    # Sort the dictionary alphabetically based on keys
    type_count = dict(sorted(type_count.items()))

    return type_count

# Example usage:
type_count_result = get_type_count(df)
print(type_count_result)
```

```
{'high': 56, 'low': 196, 'medium': 89}
```

Question 3: Bus Count Index Retrieval

Create a Python function named get_bus_indexes that takes the dataset-1.csv as a DataFrame. The function should identify and return the indices as a list (sorted in ascending order) where the bus values are greater than twice the mean value of the bus column in the DataFrame.

In [12]:
```
def get_bus_indexes(df):
    # Calculate the mean value of the 'bus' column
    bus_mean = df['bus'].mean()

    # Identify indices where the 'bus' values are greater than twice the mean
    bus_indexes = df[df['bus'] > 2 * bus_mean].index.tolist()
```

```
    # Sort the indices in ascending order
    bus_indexes.sort()

    return bus_indexes

# Example usage:
bus_indexes_result = get_bus_indexes(df)
print(bus_indexes_result)
```

```
[2, 7, 12, 17, 25, 30, 54, 64, 70, 97, 144, 145, 149, 154, 160, 201, 206, 210, 215, 234, 235, 245, 250, 309, 314, 319, 322, 323,
334, 340]
```

Question 4: Route Filtering

Create a python function filter_routes that takes the dataset-1.csv as a DataFrame. The function should return the sorted list of values of column route for which the average of values of truck column is greater than 7.

In [13]:
```
def filter_routes(df):
    # Calculate the average of values in the 'truck' column for each 'route'
    route_avg_truck = df.groupby('route')['truck'].mean()

    # Filter routes where the average of 'truck' column is greater than 7
    selected_routes = route_avg_truck[route_avg_truck > 7].index.tolist()

    # Sort the list of selected routes
    selected_routes.sort()

    return selected_routes

# Example usage:
selected_routes_result = filter_routes(df)
print(selected_routes_result)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Question 5: Matrix Value Modification

Create a Python function named multiply_matrix that takes the resulting DataFrame from Question 1, as input and modifies each value according to the following logic:

If a value in the DataFrame is greater than 20, multiply those values by 0.75, If a value is 20 or less, multiply those values by 1.25. The function should return the modified DataFrame which has values rounded to 1 decimal place.

In [14]:
```python
def multiply_matrix(car_matrix_result):
    # Create a copy of the DataFrame to avoid modifying the original
    modified_matrix = car_matrix_result.copy()

    # Apply the specified logic to modify the values
    modified_matrix[modified_matrix > 20] *= 0.75
    modified_matrix[modified_matrix <= 20] *= 1.25

    # Round the values to 1 decimal place
    modified_matrix = modified_matrix.round(1)

    return modified_matrix

# Example usage:
modified_matrix_result = multiply_matrix(car_matrix_result)
print(modified_matrix_result)
```

id_2	801	802	803	804	805	806	807	808	809	821	822	823
id_1												
801	0.0	3.5	7.5	9.6	14.6	16.8	21.1	24.5	19.7	22.0	23.1	24.9
802	3.5	0.0	4.2	6.5	11.5	13.6	17.9	21.4	23.1	19.6	20.7	22.4
803	7.5	4.2	0.0	2.5	7.5	9.6	13.9	17.4	19.1	22.2	23.6	19.4
804	9.6	6.5	2.5	0.0	5.5	7.6	11.9	15.4	17.1	20.2	21.6	23.9
805	14.6	11.5	7.5	5.5	0.0	2.5	6.8	10.2	12.0	15.0	16.5	18.8
806	16.8	13.6	9.6	7.6	2.5	0.0	4.8	8.2	10.0	13.0	14.5	16.8
807	21.1	17.9	13.9	11.9	6.8	4.8	0.0	3.6	5.4	8.5	10.0	12.3
808	24.5	21.4	17.4	15.4	10.2	8.2	3.6	0.0	2.1	5.2	6.6	8.9
809	19.7	23.1	19.1	17.1	12.0	10.0	5.4	2.1	0.0	3.6	5.1	7.4
821	22.0	19.6	22.2	20.2	15.0	13.0	8.5	5.2	3.6	0.0	2.2	4.6
822	23.1	20.7	23.6	21.6	16.5	14.5	10.0	6.6	5.1	2.2	0.0	2.8
823	24.9	22.4	19.4	23.9	18.8	16.8	12.3	8.9	7.4	4.6	2.8	0.0
824	20.9	23.7	20.7	19.2	20.5	9.8	14.0	10.6	9.2	6.3	4.5	2.2
825	21.8	24.8	21.8	20.3	22.0	20.0	15.5	12.1	10.6	7.8	5.9	3.7
826	23.2	21.2	23.5	22.0	24.2	22.2	17.7	14.3	12.8	10.0	8.2	5.9
827	24.4	22.4	20.0	23.6	19.7	24.3	19.8	16.4	14.9	11.8	10.0	7.7
829	27.2	25.3	22.9	21.7	23.3	21.8	24.5	21.2	19.7	16.6	14.8	12.5
830	28.7	26.8	24.4	23.2	20.1	23.6	20.2	23.6	22.1	19.0	17.2	14.9
831	29.4	27.5	25.1	23.9	20.8	24.5	21.1	24.8	23.3	20.2	18.4	16.1

id_2	824	825	826	827	829	830	831
id_1							
801	20.9	21.8	23.2	24.4	27.2	28.7	29.4
802	23.7	24.8	21.2	22.4	25.3	26.8	27.5
803	20.7	21.8	23.5	20.0	22.9	24.4	25.1
804	19.2	20.3	22.0	23.6	21.7	23.2	23.9
805	20.5	22.0	24.2	19.7	23.3	20.1	20.8
806	18.5	20.0	22.2	24.3	21.8	23.6	24.5
807	14.0	15.5	17.7	19.8	24.5	20.2	21.1
808	10.6	12.1	14.3	16.4	21.2	23.6	24.8
809	9.2	10.6	12.8	14.9	19.7	22.1	23.3
821	6.3	7.8	10.0	11.8	16.6	19.0	20.2
822	4.5	5.9	8.2	10.0	14.8	17.2	18.4
823	2.2	3.7	5.9	7.7	12.5	14.9	16.1
824	0.0	2.1	4.4	6.2	11.0	13.3	14.6
825	2.1	0.0	2.8	4.6	9.3	11.7	12.9
826	4.4	2.8	0.0	2.6	7.3	9.6	10.9
827	6.2	4.6	2.6	0.0	5.2	7.6	8.8
829	20.1	9.3	7.3	5.2	0.0	3.0	4.2
830	13.3	0.0	9.6	7.6	3.0	0.0	1.7
831	14.6	12.9	10.9	8.8	4.2	1.7	0.0

Question 6: Time Check

You are given a dataset, dataset-2.csv, containing columns id, id_2, and timestamp (startDay, startTime, endDay, endTime). The goal is to verify the completeness of the time data by checking whether the timestamps for each unique (id, id_2) pair cover a full 24-hour period (from 12:00:00 AM to 11:59:59 PM) and span all 7 days of the week (from Monday to Sunday).

Create a function that accepts dataset-2.csv as a DataFrame and returns a boolean series that indicates if each (id, id_2) pair has incorrect timestamps. The boolean series must have multi-index (id, id_2).

In [29]:
```
df=pd.read_csv("dataset-2.csv")
```

In [30]:
```
df.info
```

Out[30]:

```
<bound method DataFrame.info of             id        name     id_2  startDay startTime     endDay   endTime  \
0      1040000  Montgomery       -1    Monday  05:00:00  Wednesday  10:00:00
1      1040010       Black       -1    Monday  10:00:00     Friday  15:00:00
2      1040020     Emerald       -1  Thursday  15:00:00     Friday  19:00:00
3      1040030       Foley       -1    Monday  19:00:00     Friday  23:59:59
4      1050000    Whittier  1050001  Saturday  00:00:00     Sunday  23:59:59
...        ...         ...      ...       ...       ...        ...       ...
39509  1031012     Baldwin  1031030    Monday  19:00:00     Friday  23:59:59
39510  1031012     Baldwin  1031032  Saturday  00:00:00     Sunday  23:59:59
39511  1031014    Thickson  1031016  Saturday  00:00:00     Sunday  23:59:59
39512  1031014    Thickson  1031018    Monday  05:00:00  Wednesday  10:00:00
39513  1031014    Thickson  1031020    Monday  10:00:00     Friday  15:00:00

       able2Hov2  able2Hov3  able3Hov2  able3Hov3  able5Hov2  able5Hov3  \
0            3.0        3.0       -1.0         -1          3          3
1            6.0        6.0       -1.0         -1          6          6
2            3.0        3.0       -1.0         -1          3          3
3            6.0        6.0       -1.0         -1          6          6
4            6.0        6.0        NaN         -1          6          6
...          ...        ...        ...        ...        ...        ...
39509       11.0       11.0        4.0          4         11         11
39510       11.0       11.0        4.0          4         11         11
39511       11.0       11.0        4.0          4         11         11
39512        8.0        8.0        4.0          4          8          8
39513       11.0       11.0        4.0          4         11         11

       able4Hov2  able4Hov3
0              3          3
1              6          6
2              3          3
3              6          6
4              6          6
...          ...        ...
39509         11         11
39510         11         11
39511         11         11
39512          8          8
39513         11         11

[39514 rows x 15 columns]>
```

In [32]:

```
from datetime import datetime, timedelta
```

```
def check_time_completeness(df):
    # Combine date and time columns to create datetime objects with explicit format
    df['start_datetime'] = pd.to_datetime(df['startDay'] + ' ' + df['startTime'], format='%A %H:%M:%S')
    df['end_datetime'] = pd.to_datetime(df['endDay'] + ' ' + df['endTime'], format='%A %H:%M:%S')

    # Function to check if a time range is valid (covers 24 hours and spans all 7 days)
    def is_valid_time_range(group):
        start_time = group['start_datetime'].min()
        end_time = group['end_datetime'].max()

        expected_start_time = datetime.strptime("00:00:00", "%H:%M:%S")
        expected_end_time = datetime.strptime("23:59:59", "%H:%M:%S")

        expected_days = set(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'])

        return start_time <= expected_start_time and end_time >= expected_end_time and set(group['startDay'].unique()) == expecte

    # Apply the function to check time completeness for each (id, id_2) group
    completeness_series = df.groupby(['id', 'id_2']).apply(is_valid_time_range)

    return completeness_series

# Example usage:
completeness_result = check_time_completeness(df)
print(completeness_result)
```

```
id       id_2
1014000  -1          False
1014002  -1          False
1014003  -1          False
1030000  -1          False
          1030002    False
                     ...
1330016   1330006    False
          1330008    False
          1330010    False
          1330012    False
          1330014    False
Length: 9254, dtype: bool
```

In [33]:

```
df=pd.read_csv("dataset-3.csv")
df.info
```

Out[33]:

```
<bound method DataFrame.info of     id_start   id_end  distance
0    1001400  1001402       9.7
1    1001402  1001404      20.2
2    1001404  1001406      16.0
3    1001406  1001408      21.7
4    1001408  1001410      11.1
5    1001410  1001412      15.6
6    1001412  1001414      18.2
7    1001414  1001416      13.2
8    1001416  1001418      13.6
9    1001418  1001420      12.9
10   1001420  1001422       9.6
11   1001422  1001424      11.4
12   1001424  1001426      18.6
13   1001426  1001428      15.8
14   1001428  1001430       8.6
15   1001430  1001432       9.0
16   1001432  1001434       7.9
17   1001434  1001436       4.0
18   1001436  1001438       9.0
19   1001436  1001437       5.0
20   1001438  1001437       4.0
21   1001438  1001440      10.0
22   1001440  1001442       3.9
23   1001442  1001488       4.5
24   1001488  1004356       4.0
25   1004356  1004354       2.0
26   1004354  1004355       2.0
27   1004355  1001444       0.7
28   1001444  1001446       6.6
29   1001446  1001448       9.6
30   1001448  1001450      15.7
31   1001450  1001452       9.9
32   1001452  1001454      11.3
33   1001454  1001456      13.6
34   1001456  1001458       8.9
35   1001458  1001460       5.1
36   1001460  1001461      12.8
37   1001460  1001462      17.9
38   1001461  1001462       5.1
39   1001462  1001464      26.7
40   1001464  1001466       8.5
41   1001466  1001468      10.7
```

```
42   1001468  1001470      10.6
43   1001470  1001472      16.0>
```

Question 1: Distance Matrix Calculation

Create a function named calculate_distance_matrix that takes the dataset-3.csv as input and generates a DataFrame representing distances between IDs.

The resulting DataFrame should have cumulative distances along known routes, with diagonal values set to 0. If distances between toll locations A to B and B to C are known, then the distance from A to C should be the sum of these distances. Ensure the matrix is symmetric, accounting for bidirectional distances between toll locations (i.e. A to B is equal to B to A).

In [36]:
```python
def calculate_distance_matrix(df):
    # Convert IDs to integers to avoid floating-point precision issues
    df['id_start'] = df['id_start'].astype(int)
    df['id_end'] = df['id_end'].astype(int)

    # Get all unique IDs
    all_ids = list(set(df['id_start'].unique()).union(set(df['id_end'].unique())))

    # Create an empty DataFrame with IDs as both index and columns
    distance_matrix = pd.DataFrame(index=all_ids, columns=all_ids)

    # Fill diagonal with zeros
    distance_matrix.values[[range(len(distance_matrix))]*2] = 0

    # Iterate through the DataFrame and calculate cumulative distances
    for index, row in df.iterrows():
        start_id = int(row['id_start'])
        end_id = int(row['id_end'])
        distance = row['distance']

        # Update the distance matrix with cumulative distances
        distance_matrix.loc[start_id, end_id] = distance_matrix.loc[start_id, end_id] + distance
        distance_matrix.loc[end_id, start_id] = distance_matrix.loc[end_id, start_id] + distance

    return distance_matrix

# Example usage:
```

```
distance_matrix_result = calculate_distance_matrix(df)
print(distance_matrix_result)
```

	1001472	1001488	1004354	1004355	1004356	1001400	1001402	1001404	\
1001472	0	0	0	0	0	0	0	0	
1001488	0	0	0	0	4.0	0	0	0	
1004354	0	0	0	2.0	2.0	0	0	0	
1004355	0	0	2.0	0	0	0	0	0	
1004356	0	4.0	2.0	0	0	0	0	0	
1001400	0	0	0	0	0	0	9.7	0	
1001402	0	0	0	0	0	9.7	0	20.2	
1001404	0	0	0	0	0	0	20.2	0	
1001406	0	0	0	0	0	0	0	16.0	
1001408	0	0	0	0	0	0	0	0	
1001410	0	0	0	0	0	0	0	0	
1001412	0	0	0	0	0	0	0	0	
1001414	0	0	0	0	0	0	0	0	
1001416	0	0	0	0	0	0	0	0	
1001418	0	0	0	0	0	0	0	0	
1001420	0	0	0	0	0	0	0	0	
1001422	0	0	0	0	0	0	0	0	
1001424	0	0	0	0	0	0	0	0	
1001426	0	0	0	0	0	0	0	0	
1001428	0	0	0	0	0	0	0	0	
1001430	0	0	0	0	0	0	0	0	
1001432	0	0	0	0	0	0	0	0	
1001434	0	0	0	0	0	0	0	0	
1001436	0	0	0	0	0	0	0	0	
1001437	0	0	0	0	0	0	0	0	
1001438	0	0	0	0	0	0	0	0	
1001440	0	0	0	0	0	0	0	0	
1001442	0	4.5	0	0	0	0	0	0	
1001444	0	0	0	0.7	0	0	0	0	
1001446	0	0	0	0	0	0	0	0	
1001448	0	0	0	0	0	0	0	0	
1001450	0	0	0	0	0	0	0	0	
1001452	0	0	0	0	0	0	0	0	
1001454	0	0	0	0	0	0	0	0	
1001456	0	0	0	0	0	0	0	0	
1001458	0	0	0	0	0	0	0	0	
1001460	0	0	0	0	0	0	0	0	
1001461	0	0	0	0	0	0	0	0	
1001462	0	0	0	0	0	0	0	0	
1001464	0	0	0	0	0	0	0	0	
1001466	0	0	0	0	0	0	0	0	
1001468	0	0	0	0	0	0	0	0	
1001470	16.0	0	0	0	0	0	0	0	

	1001406	1001408	...	1001454	1001456	1001458	1001460	1001461	1001462	\
1001472	0	0	...	0	0	0	0	0	0	
1001488	0	0	...	0	0	0	0	0	0	
1004354	0	0	...	0	0	0	0	0	0	
1004355	0	0	...	0	0	0	0	0	0	
1004356	0	0	...	0	0	0	0	0	0	
1001400	0	0	...	0	0	0	0	0	0	
1001402	0	0	...	0	0	0	0	0	0	
1001404	16.0	0	...	0	0	0	0	0	0	
1001406	0	21.7	...	0	0	0	0	0	0	
1001408	21.7	0	...	0	0	0	0	0	0	
1001410	0	11.1	...	0	0	0	0	0	0	
1001412	0	0	...	0	0	0	0	0	0	
1001414	0	0	...	0	0	0	0	0	0	
1001416	0	0	...	0	0	0	0	0	0	
1001418	0	0	...	0	0	0	0	0	0	
1001420	0	0	...	0	0	0	0	0	0	
1001422	0	0	...	0	0	0	0	0	0	
1001424	0	0	...	0	0	0	0	0	0	
1001426	0	0	...	0	0	0	0	0	0	
1001428	0	0	...	0	0	0	0	0	0	
1001430	0	0	...	0	0	0	0	0	0	
1001432	0	0	...	0	0	0	0	0	0	
1001434	0	0	...	0	0	0	0	0	0	
1001436	0	0	...	0	0	0	0	0	0	
1001437	0	0	...	0	0	0	0	0	0	
1001438	0	0	...	0	0	0	0	0	0	
1001440	0	0	...	0	0	0	0	0	0	
1001442	0	0	...	0	0	0	0	0	0	
1001444	0	0	...	0	0	0	0	0	0	
1001446	0	0	...	0	0	0	0	0	0	
1001448	0	0	...	0	0	0	0	0	0	
1001450	0	0	...	0	0	0	0	0	0	
1001452	0	0	...	11.3	0	0	0	0	0	
1001454	0	0	...	0	13.6	0	0	0	0	
1001456	0	0	...	13.6	0	8.9	0	0	0	
1001458	0	0	...	0	8.9	0	5.1	0	0	
1001460	0	0	...	0	0	5.1	0	12.8	17.9	
1001461	0	0	...	0	0	0	12.8	0	5.1	
1001462	0	0	...	0	0	0	17.9	5.1	0	
1001464	0	0	...	0	0	0	0	0	26.7	
1001466	0	0	...	0	0	0	0	0	0	
1001468	0	0	...	0	0	0	0	0	0	

1001470	0	0	...	0	0	0	0	0	0

	1001464	1001466	1001468	1001470
1001472	0	0	0	16.0
1001488	0	0	0	0
1004354	0	0	0	0
1004355	0	0	0	0
1004356	0	0	0	0
1001400	0	0	0	0
1001402	0	0	0	0
1001404	0	0	0	0
1001406	0	0	0	0
1001408	0	0	0	0
1001410	0	0	0	0
1001412	0	0	0	0
1001414	0	0	0	0
1001416	0	0	0	0
1001418	0	0	0	0
1001420	0	0	0	0
1001422	0	0	0	0
1001424	0	0	0	0
1001426	0	0	0	0
1001428	0	0	0	0
1001430	0	0	0	0
1001432	0	0	0	0
1001434	0	0	0	0
1001436	0	0	0	0
1001437	0	0	0	0
1001438	0	0	0	0
1001440	0	0	0	0
1001442	0	0	0	0
1001444	0	0	0	0
1001446	0	0	0	0
1001448	0	0	0	0
1001450	0	0	0	0
1001452	0	0	0	0
1001454	0	0	0	0
1001456	0	0	0	0
1001458	0	0	0	0
1001460	0	0	0	0
1001461	0	0	0	0
1001462	26.7	0	0	0
1001464	0	8.5	0	0
1001466	8.5	0	10.7	0

```
1001468       0    10.7       0    10.6
1001470       0       0    10.6       0

[43 rows x 43 columns]
```

Question 2: Unroll Distance Matrix

Create a function unroll_distance_matrix that takes the DataFrame created in Question 1. The resulting DataFrame should have three columns: columns id_start, id_end, and distance.

All the combinations except for same id_start to id_end must be present in the rows with their distance values from the input DataFrame.

In [41]:
```python
def unroll_distance_matrix(distance_matrix):
    # Extract column and index labels from the distance matrix
    ids = distance_matrix.index.tolist()

    # Create an empty list to store dictionaries
    unrolled_data = []

    # Iterate over the distance matrix to populate the unrolled data list
    for id_start in ids:
        for id_end in ids:
            # Skip same id_start to id_end combination
            if id_start == id_end:
                continue

            # Get the distance value from the distance matrix
            distance = distance_matrix.loc[id_start, id_end]

            # Append the data to the list as a dictionary
            unrolled_data.append({'id_start': id_start, 'id_end': id_end, 'distance': distance})

    # Convert the list of dictionaries to a DataFrame
    unrolled_df = pd.DataFrame(unrolled_data)

    return unrolled_df

# Example usage:
unrolled_distance_df = unroll_distance_matrix(distance_matrix_result)
print(unrolled_distance_df)
```

```
      id_start   id_end  distance
0      1001472  1001488       0.0
1      1001472  1004354       0.0
2      1001472  1004355       0.0
3      1001472  1004356       0.0
4      1001472  1001400       0.0
...        ...      ...       ...
1801   1001470  1001461       0.0
1802   1001470  1001462       0.0
1803   1001470  1001464       0.0
1804   1001470  1001466       0.0
1805   1001470  1001468      10.6

[1806 rows x 3 columns]
```

Question 3: Finding IDs within Percentage Threshold

Create a function find_ids_within_ten_percentage_threshold that takes the DataFrame created in Question 2 and a reference value from the id_start column as an integer.

Calculate average distance for the reference value given as an input and return a sorted list of values from id_start column which lie within 10% (including ceiling and floor) of the reference value's average.

In [43]:
```python
def find_ids_within_ten_percentage_threshold(df, reference_value):
    # Filter rows with the specified reference value in id_start
    reference_df = df[df['id_start'] == reference_value]

    # Calculate the average distance for the reference value
    reference_avg_distance = reference_df['distance'].mean()

    # Calculate the threshold values
    lower_threshold = reference_avg_distance * 0.9
    upper_threshold = reference_avg_distance * 1.1

    # Filter rows within the 10% threshold
    within_threshold_df = df[(df['distance'] >= lower_threshold) & (df['distance'] <= upper_threshold)]

    # Get unique values from id_start column and sort them
    result_ids = sorted(within_threshold_df['id_start'].unique())

    return result_ids
```

```
# Example usage:
reference_value = 1001437  # Replace with the desired reference value
result_ids = find_ids_within_ten_percentage_threshold(unrolled_distance_df, reference_value)
print(result_ids)
```

```
[]
```

Question 4: Calculate Toll Rate

Create a function calculate_toll_rate that takes the DataFrame created in Question 2 as input and calculates toll rates based on vehicle types.

The resulting DataFrame should add 5 columns to the input DataFrame: moto, car, rv, bus, and truck with their respective rate coefficients. The toll rates should be calculated by multiplying the distance with the given rate coefficients for each vehicle type:

0.8 for moto 1.2 for car 1.5 for rv 2.2 for bus 3.6 for truck

In [45]:
```
def calculate_toll_rate(df):
    # Define rate coefficients for each vehicle type
    rate_coefficients = {'moto': 0.8, 'car': 1.2, 'rv': 1.5, 'bus': 2.2, 'truck': 3.6}

    # Add columns for each vehicle type with their respective rate coefficients
    for vehicle_type, rate_coefficient in rate_coefficients.items():
        df[vehicle_type] = df['distance'] * rate_coefficient

    return df

# Example usage:
toll_rate_df = calculate_toll_rate(unrolled_distance_df)
print(toll_rate_df)
```

	id_start	id_end	distance	moto	car	rv	bus	truck
0	1001472	1001488	0.0	0.00	0.00	0.0	0.00	0.00
1	1001472	1004354	0.0	0.00	0.00	0.0	0.00	0.00
2	1001472	1004355	0.0	0.00	0.00	0.0	0.00	0.00
3	1001472	1004356	0.0	0.00	0.00	0.0	0.00	0.00
4	1001472	1001400	0.0	0.00	0.00	0.0	0.00	0.00
...	...	...	...	...	...	...	...	...
1801	1001470	1001461	0.0	0.00	0.00	0.0	0.00	0.00
1802	1001470	1001462	0.0	0.00	0.00	0.0	0.00	0.00
1803	1001470	1001464	0.0	0.00	0.00	0.0	0.00	0.00
1804	1001470	1001466	0.0	0.00	0.00	0.0	0.00	0.00
1805	1001470	1001468	10.6	8.48	12.72	15.9	23.32	38.16

[1806 rows x 8 columns]

Question 5: Calculate Time-Based Toll Rates

Create a function named calculate_time_based_toll_rates that takes the DataFrame created in Question 3 as input and calculates toll rates for different time intervals within a day.

The resulting DataFrame should have these five columns added to the input: start_day, start_time, end_day, and end_time.

start_day, end_day must be strings with day values (from Monday to Sunday in proper case) start_time and end_time must be of type datetime.time() with the values from time range given below. Modify the values of vehicle columns according to the following time ranges:

Weekdays (Monday - Friday):

From 00:00:00 to 10:00:00: Apply a discount factor of 0.8 From 10:00:00 to 18:00:00: Apply a discount factor of 1.2 From 18:00:00 to 23:59:59: Apply a discount factor of 0.8 Weekends (Saturday and Sunday):

Apply a constant discount factor of 0.7 for all times. For each unique (id_start, id_end) pair, cover a full 24-hour period (from 12:00:00 AM to 11:59:59 PM) and span all 7 days of the week (from Monday to Sunday).

In [ ]: