TASK 1.1 Report

The function called remove\_duplicates\_file which reads in a file of numbers, removes duplicates, and writes the unique numbers to a new file. The input file path and output file path are provided as function arguments.

The function first reads in all the numbers from the input file, sorts them in ascending order, and initializes variables to keep track of the current number, its count, and last index. It then iterates over the sorted list and checks if the current number is the same as the previous one. If it is, it increments the count and updates the last index. If it is not, it adds the previous number to the output list based on its count and last index, and resets the variables to the new number.

After the loop, the function adds the final number to the output list and writes the list to the output file. Finally, it prints the output list.

This code is useful for removing duplicates from a list of numbers and storing only the unique ones in a file. The sorting step ensures that duplicates are always adjacent to each other, making them easy to identify and remove. The use of variables to keep track of the current number, its count, and last index helps to determine which unique number(s) should be added to the output list.

Overall, this function is a straightforward and efficient way to remove duplicates from a list of numbers and store the unique ones in a file.

TASK 1.2 Report:

The Task 1.2 code reads in a file of integers, sorts them in ascending order, and then performs three types of data operations on them: searching for a specific value, inserting a new value into the list in ascending order, and deleting all occurrences of a specified value from the list. The operations and their corresponding values are defined in a list of tuples, where the first element of each tuple represents the type of operation (1 for search, 2 for insertion, and 3 for deletion), and the second element represents the value involved in the operation.

The code defines functions for each of the data operations, as well as a function to perform the specified operations on the sorted list of integers and write the results to an output file. The time taken to complete each operation is also recorded and included in the output file.

The main function reads in the operations from a file and defines example operations to perform on the sorted list of integers. It then calls the perform\_data\_operations function to perform the specified operations and write the results to an output file. Finally, it prints the final state of the sorted list of integers.

Overall, this code demonstrates how to read in and manipulate data in Python using functions and file input/output. The use of time tracking allows for performance analysis and optimization of the data operations.

TASK 1.3 Report

This task 1.3 code performs a multi-threaded word frequency count for a given list of names in a text file. The program reads in a text file containing a large number of words, and a file containing a list of names. It then splits the list of words into multiple chunks and creates a separate thread to count the frequency of each name in each chunk. The count is stored in a results dictionary, which is accessed using a lock to avoid synchronization issues. Finally, the program writes the frequency dictionary to a new file.

The purpose of this program is to perform a quick and efficient count of how many times a given name appears in a large text file. The use of threads allows the program to perform the count in parallel, improving performance. The lock ensures that multiple threads do not try to modify the dictionary at the same time, avoiding errors and ensuring the correct results are stored. The resulting frequency file shows the number of times each name appears in the text file.

Overall, this program is a good example of how threading can be used to improve performance when working with large datasets. However, it is important to note that the performance improvement may depend on the specific dataset and hardware used, so it is always a good idea to test and evaluate the program on different datasets.

TASK 1.4 Report

The Task 1.4 code is a Python program that finds the cheapest route between two stations in a railway network. The program reads in the railway network from a CSV file, which contains the name of each station, the name of its neighboring stations, and the cost of traveling between them. The program then prompts the user to enter the name of the departure and destination stations. It uses Dijkstra's algorithm to find the cheapest cost of traveling between the two stations and the corresponding route.

The load\_railway\_network() function reads the railway network data from the CSV file and stores it in a dictionary where the keys are the station names and the values are the neighboring stations and their corresponding costs.

The dijkstra() function implements Dijkstra's algorithm, which calculates the shortest path from the start station to all other stations in the network. It uses a priority queue to keep track of the next station to visit and the cost of traveling to that station.

The get\_route() function finds the cheapest route between the start and end stations. It also uses a priority queue to keep track of the stations to visit and their corresponding paths.

The main() function is the main entry point of the program. It reads the railway network, prompts the user for the start and end stations, and prints the cheapest cost and route between the two stations. If there is no route between the two stations, it prints an appropriate message.

Overall, this program demonstrates how to use Dijkstra's algorithm to find the cheapest route between two stations in a railway network.

TASK 2.2 Sustainability

One important issue relating to computational sustainability that system developers should consider for national railway services is energy efficiency. According to the International Energy Agency, transportation accounts for 23% of global energy-related CO2 emissions, with railways being a significant contributor (IEA, 2020). As such, it is crucial for system developers to design railway systems that are energy-efficient, which can be achieved through the use of sustainable energy sources and the optimization of train schedules and routes.

In Task 1.4, the use of Dijkstra's algorithm to find the cheapest route between two stations in a railway network can contribute to energy efficiency by reducing the time and distance traveled by trains, which in turn reduces fuel consumption and CO2 emissions. Additionally, the use of a railway network dataset can aid in the identification of bottlenecks and areas where congestion is likely to occur, which can be addressed through the optimization of train schedules and routes.

System developers should also consider the long-term sustainability of the railway system, which includes the maintenance and upkeep of infrastructure and the minimization of waste. This can be achieved through the use of sustainable materials in construction and the implementation of recycling and waste reduction programs. Furthermore, developers should consider the impact of the railway system on local ecosystems and biodiversity and take steps to minimize negative effects, such as by implementing wildlife crossings and avoiding construction in sensitive areas.

Overall, system developers have a crucial role to play in promoting sustainability in national railway services through the use of energy-efficient technologies, the optimization of train schedules and routes, and the consideration of long-term sustainability factors such as infrastructure maintenance and waste reduction. By taking these factors into account, developers can help to ensure that railway systems are not only cost-effective and efficient but also sustainable and environmentally friendly.

Reference:

International Energy Agency. (2020). CO2 emissions from fuel combustion highlights. Retrieved from https://www.iea.org/reports/co2-emissions-from-fuel-combustion-highlights-2020