# Task 2.1

## Hamidur Rahman – 2009146

### Task 1.1 - remove duplicated elements with a special rule

In the provided solution, I opted for a straightforward method utilising a dictionary and a list to efficiently manage the removal of duplicate numbers while adhering to the given rule. The design is broken down as follows:

1. Read the numbers from the input file and save them to a list.
2. Establish a dictionary, counts, to keep track of the indexes of each number.
3. Iterate throughout the numbers and populate the counts dictionary. Add the index of each number to the dictionary's corresponding list.
4. Determine the indices of the retained numbers based on the rule by iterating over the counts dictionary.
5. Sort the retained indexes ascendingly.
6. Using the retained indices, retrieve the retained numbers from the initial list.
7. Write the resultant values to the output file.

This program has a space complexity of O(N), where N is the number of input numbers, because we store the numbers in a list and use a dictionary to keep track of the indices. Time complexity is also O(N) because we iterate over the numbers twice (once to populate the dictionary and again to retrieve the retained numbers).

By storing the indices directly and avoiding superfluous comparisons, this solution performs well in terms of efficiency. We only store the original numbers and a dictionary with a limited number of entries, resulting in a relatively low memory footprint.

### Task 1.2 – data operations

In this solution, I have used a combination of merge sort and binary search algorithms to efficiently sort and perform operations on the given set of numbers. Here is an explanation of the design:

1. Importing and Sorting:

* The numbers are imported from the provided file and stored in a list.
* Merge sort algorithm is used to sort the numbers in ascending order.
* Merge sort has a time complexity of O(n log n) in all cases, making it an efficient choice for sorting large sets of data.

1. Search

* Binary search is used to search for a value in the sorted numbers list.
* Binary search has a time complexity of O(log n), which is efficient for searching in a sorted list.

1. Insertion

* Binary search is used to find the correct position to insert the value while maintaining the ascending order of the list.
* The time complexity for binary search-based insertion is O(log n), and the insertion itself takes O(n) time on average, resulting in an overall efficient insertion operation.

1. Deletion

* Binary search is used to find the first occurrence of the value in the sorted list.
* All occurrences of the value are then deleted from the list by shifting elements.
* The time complexity for binary search-based deletion is O(log n), and the deletion operation itself takes O(n) time on average, resulting in an overall efficient deletion operation.

This solution prioritises the use of effective sorting and searching algorithms to manage large number sets. The selection of merge sort for sorting ensures an acceptable time complexity, whereas binary search enables efficient search, insertion, and deletion operations.

### Task 1.3 – search with parallelism

## The code employs Python's multiprocessing module to enable parallel processing for frequency counting. Here is a detailed analysis of the design decisions and algorithmic reasoning:

## The function count\_frequency(name, text) computes the frequency of the string name in the text. The function returns a tuple containing the name and its frequency.

## The `process\_file` function is tasked with reading the text file and the file containing names. The code initialises a multiprocessing pool by determining the number of available CPU cores and the quantity of names to be processed.

## For each name, an asynchronous task is submitted to the pool using the apply\_async method. This implements workload distribution among the processes, enabling parallel execution.

## The programme collects frequency results from the asynchronous tasks using the "get" method and stores them in a dictionary with the name as the key and the frequency as the value.

## The output frequencies are written to a file named 'task1\_3\_results.txt'.

## The tracemalloc module is utilised to track memory usage prior to and post the frequency counting process.

## Utilising multiprocessing enables parallel execution of frequency counting tasks, leveraging multiple CPU cores to enhance performance. A dictionary data structure is utilised for storing the frequencies, enabling efficient retrieval and storage of the pairs of name-frequency.

### Task 1.4 – cheapest train tickets

## In this train ticket search system, I've implemented Dijkstra's algorithm to determine the cheapest price and route between two stations. Here is a summary of the layout:

## Importing the CSV File: I read the data from the CSV file using the csv module in Python and created a dictionary to depict the railway network. Each station is a dictionary key, and its value is a second dictionary containing neighbouring stations and their associated costs.

## The user must enter the departure and destination stations.

## I implemented Dijkstra's algorithm using a priority queue to monitor the least expensive route and cost. The algorithm iteratively selects the station with the lowest cost, updates the costs of its neighbouring stations if a shorter path is discovered, and reconstructs the route by keeping note of the previous station for each station.

## If the user wishes to modify the raw data, they can either input the specifics of a new route or modify the expense of an existing route. Accordingly, the railway network dictionary is revised, and the updated data are written back to the CSV file.

## I chose a dictionary to symbolise the railway system because it provides quick access to locations and their neighbours. By prioritising stations with reduced costs, the Dijkstra algorithm with a priority queue allows for efficient network exploration.

## The time complexity of Dijkstra's algorithm is O((V + E) log V), where V represents the number of stations and E represents the number of connections. The priority queue ensures efficient station selection at the lowest possible cost. The use of a dictionary enables rapid access to neighbouring stations, and the total amount of memory is measured and reported in order to analyse and optimise memory consumption.