CCP6214: ALGORITHM DESIGN AND ANALYSIS TERM 2410

Assignment

Prepared by:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Num | Student ID | Student Name | Task Description | Percentage (%) |
| 1 | 1211103705 | Ter Zheng Bin | Dataset 1 2 Q4 | 25 |
| 2 | 1211101506 | Leong Jia Yi | Dataset 2 Q3 | 25 |
| 3 | 1211102809 | Liau Kai Ze | Dataset 2 Q3 Q4 | 25 |
| 4 | 1211102810 | Lai Cheng Yung | Dataset 1 Q2 | 25 |

Table of Contents

[**Question 1 (Dataset)** 4](#_Toc169366895)

[Dataset 1 4](#_Toc169366896)

[Dataset 2 6](#_Toc169366897)

[**Question 2** 12](#_Toc169366898)

[Code Explaination 12](#_Toc169366899)

[Libraries used: 12](#_Toc169366900)

[Function: read\_dataset 13](#_Toc169366901)

[Function: write\_to\_file 14](#_Toc169366902)

[Function: heapify 15](#_Toc169366903)

[Function: heap\_sort 16](#_Toc169366904)

[Function: selection\_sort 17](#_Toc169366905)

[Driver code 18](#_Toc169366906)

[Output 21](#_Toc169366907)

[Time Complexity 24](#_Toc169366908)

[**Question 3** 26](#_Toc169366909)

[Code Explanation (Dijkstra's Algorithm) 26](#_Toc169366910)

[Libraries used: 26](#_Toc169366911)

[Function: load\_data 26](#_Toc169366912)

[Function: dijkstra 27](#_Toc169366913)

[Function: get\_shortest\_path 28](#_Toc169366914)

[Driver Code (Dijkstra's Algorithm) 29](#_Toc169366915)

[Output (Dijkstra’s Algorithm) 30](#_Toc169366916)

[Time Complexity (Dijkstra’s Algorithm) 32](#_Toc169366917)

[Code Explanation (Kruskal's Algorithm) 33](#_Toc169366918)

[Function: find, union and kruskal 33](#_Toc169366919)

[Driver code (Kruskal's Algorithm) 34](#_Toc169366920)

[Output (Kruskal Algorithm) 36](#_Toc169366921)

[Time Complexity (Kruskal's Algorithm) 37](#_Toc169366922)

[**Question 4** 38](#_Toc169366923)

[Code Explaination 38](#_Toc169366924)

[Libraries used: 38](#_Toc169366925)

[Function: load\_star\_data 38](#_Toc169366926)

[Function: knapscak 39](#_Toc169366927)

[Function: save\_knapsack\_result 41](#_Toc169366928)

[Function: save\_knapsack\_result 41](#_Toc169366929)

[Driver Code 42](#_Toc169366930)

[Output 43](#_Toc169366931)

[Time Complexity 46](#_Toc169366932)

# **Question 1 (Dataset)**

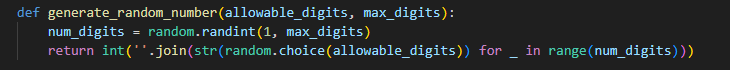
In Question 1, we will detail the algorithms designed to generate two datasets. The first dataset consists of random numeric values, seeded by the group leader's ID, to be used for further implementation. The second dataset is designed for a star conquer quest, containing star locations and routes connecting them. This dataset will include values such as coordinates, weight, and profit, essential for theoretical analyses and practical implementation in subsequent tasks.

## Dataset 1

The algorithm for Dataset 1 is designed to generate six sets of random numeric data, with each set having a different size. The sizes of the datasets are 100, 1,000, 10,000, 100,000, 500,000, and 1,000,000. The random numbers are generated using specific digits derived from the group leader’s ID.  
  
First, the algorithm imports the necessary libraries: random for generating random numbers and time for seeding the random number generator with the current time, ensuring variability in the generated numbers.



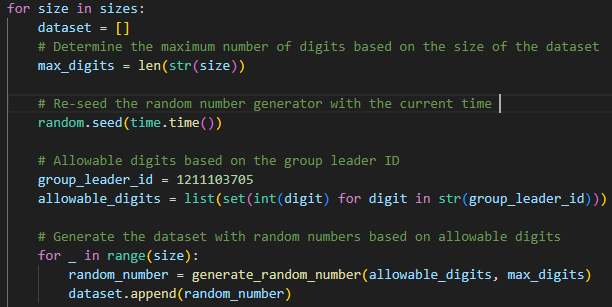
Next, it defines a function called **generate\_random\_number**. This function takes two parameters allowable digits, which is a list of digits derived from the group leader's ID, and **max\_digits**, which represents the maximum number of digits a random number can have. The function generates a random number by repeatedly selecting digits from the **allowable\_digits** list up to the **max\_digits** length. For example, if its dataset 100 and the **allowable\_digits** is 25763, it can generate the random number less than 3-digit numbers and the number are random choose from the **allowable\_digit** which is 25763 and it can duplicate. Example of dataset 100 with **allowable\_digits** can be 276, 666, 253, 2,57 and more.



The algorithm then specifies the sizes of the datasets to be generated. It iterates over each size, generating a dataset for each specified size. For each dataset size, an empty list called dataset is created to store the random numbers.

To ensure variability, the random number generator is seeded with the current time using **random.seed(time.time())**. This means that each time the algorithm runs, it will produce different random numbers.

The digits from the group leader's ID are extracted and stored in a list called **allowable\_digits**. These digits are used to generate random numbers.



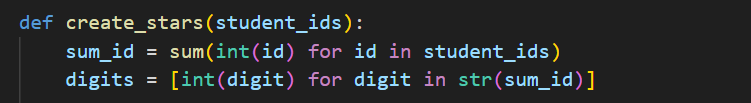
For each size, the algorithm runs a loop that iterates as many times as the size of the dataset. In each iteration, it generates a random number using the **generate\_random\_number** function and adds it to the dataset list. Once all random numbers are generated, the dataset is saved to a text file. The file is named based on the dataset size (e.g., dataset\_100.txt for a dataset of size 100).

## Dataset 2

The algorithm for Dataset 2 is designed to generate a dataset for a star conquer quest, consisting of 20-star locations (vertices) and 54 routes (edges) connecting them. Each star must connect to at least 3 other stars. The values for each star include a name, x-coordinate, y-coordinate, z-coordinate, weight, and profit, all derived from the sum of other group members' ID numbers.

First, the necessary libraries are imported: random for generating random values, numpy for numerical operations, networkx for graph operations, and matplotlib for visualization. The algorithm calculates the sum of the other group members' ID numbers and extracts the digits of this sum. These digits are used to generate random values for the stars' coordinates, weight, and profit.

The create\_stars function is called to generate 20 stars, each with a unique name (e.g., StarA, StarB) and random x, y, and z coordinates, as well as weight and profit values. The random number generator is seeded with the sum of the student IDs and the current time to ensure variability.



The create\_edges function creates a graph using networkx and adds each star as a node. It ensures that each star connects to at least 3 other stars by adding edges between stars until this condition is met. Then, it continues to add edges randomly until there are a total of 54 routes.

A screen shot of a computer code

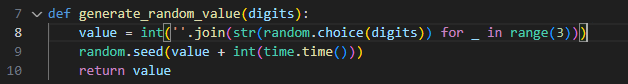
Description automatically generated

The **calculate\_distances** function calculates the Euclidean distance between each pair of connected stars. It iterates over the edges in the graph, retrieves the coordinates of the connected stars, and computes the distance using the Euclidean formula. The calculated distance is then stored as an attribute of the edge.

A computer code on a black background

Description automatically generated

The **generate\_random\_value** function takes a list of digits as input and generates a random three-digit number. It does this by randomly selecting three digits from the input list, concatenating them into a string, and converting the string to an integer. This integer value is then returned and used as the coordinates, weight, or profit for the stars in the dataset. The random number generator is seeded once at the beginning of the star creation process to ensure consistent randomness throughout.



In summary, the algorithm generates a dataset of 20 stars with random locations and attributes, ensures each star is connected to at least 3 others, calculates the distances between connected stars, saves the data to text files, and visualizes the graph of stars and routes.

This is the example of Stars Data Format that will be generated:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | x | y | z | Weight | Profit |
| StarA | 215 | 335 | 335 | 650 | 325 |
| StarB | 233 | 763 | 133 | 513 | 663 |
| StarC | 563 | 635 | 233 | 33 | 373 |
| StarD | 203 | 353 | 363 | 535 | 265 |
| StarE | 10 | 333 | 777 | 33 | 163 |
| StarF | 233 | 353 | 250 | 363 | 337 |
| StarG | 353 | 302 | 303 | 232 | 333 |
| StarH | 12 | 336 | 170 | 373 | 635 |
| StarI | 632 | 13 | 603 | 327 | 35 |
| StarJ | 25 | 257 | 310 | 125 | 235 |
| StarK | 653 | 133 | 137 | 231 | 665 |
| StarL | 330 | 723 | 655 | 717 | 367 |
| StarM | 732 | 573 | 523 | 666 | 530 |
| StarN | 226 | 313 | 733 | 172 | 701 |
| StarO | 370 | 763 | 360 | 530 | 310 |
| StarP | 333 | 325 | 602 | 316 | 537 |
| StarQ | 103 | 733 | 207 | 320 | 722 |
| StarR | 533 | 312 | 333 | 763 | 363 |
| StarS | 766 | 22 | 363 | 133 | 335 |
| StarT | 311 | 731 | 535 | 626 | 330 |

This is the example of Routes Data Format that will be generated where its display the connection of each start calculated by the distance and the distance formula is provided by the question:

|  |  |  |
| --- | --- | --- |
| Star\_1 | Star\_2 | Distance |
| StarA | StarL | 515.92 |
| StarA | StarB | 473.62 |
| StarA | StarJ | 206.90 |
| StarA | StarK | 521.39 |
| StarA | StarO | 455.89 |
| StarB | StarO | 265.14 |
| StarB | StarJ | 575.00 |
| StarB | StarH | 482.22 |
| StarB | StarR | 577.41 |
| StarC | StarQ | 471.04 |
| StarC | StarD | 475.42 |
| StarC | StarI | 727.01 |
| StarC | StarK | 518.96 |
| StarC | StarS | 658.69 |
| StarD | StarL | 488.15 |
| StarD | StarG | 169.41 |
| StarE | StarS | 916.33 |
| StarE | StarO | 698.85 |
| StarE | StarK | 929.00 |
| StarE | StarM | 802.12 |
| StarE | StarN | 221.34 |
| StarE | StarI | 720.81 |
| StarF | StarS | 637.51 |
| StarF | StarK | 487.41 |
| StarF | StarI | 631.99 |
| StarG | StarK | 382.25 |
| StarG | StarP | 300.55 |
| StarH | StarI | 822.32 |
| StarH | StarQ | 408.97 |
| StarH | StarK | 673.19 |
| StarH | StarR | 546.43 |
| StarI | StarJ | 716.82 |
| StarI | StarT | 789.42 |
| StarJ | StarS | 779.18 |
| StarJ | StarL | 655.14 |
| StarJ | StarM | 803.16 |
| StarJ | StarT | 597.58 |
| StarJ | StarR | 511.49 |
| StarJ | StarO | 614.46 |
| StarK | StarL | 848.97 |
| StarL | StarT | 121.76 |
| StarM | StarO | 440.13 |
| StarM | StarT | 449.83 |
| StarN | StarS | 716.37 |
| StarN | StarQ | 684.26 |
| StarO | StarT | 187.43 |
| StarO | StarQ | 309.19 |
| StarO | StarR | 480.31 |
| StarP | StarR | 335.45 |
| StarP | StarS | 580.02 |
| StarQ | StarR | 614.83 |
| StarQ | StarT | 388.40 |
| StarQ | StarS | 984.59 |
| StarR | StarS | 373.21 |

**Graph**

To visualize the network of stars, we calculate the distance between connected stars using the Euclidean distance formula in three-dimensional space:



In our implementation, the stars' coordinates and the routes between them are read from the provided dataset files. Each star's coordinates (x,y,z)(x, y, z)(x,y,z) are used to represent nodes in the graph, and the routes between stars are edges, with weights equal to the calculated distances. This ensures accurate spatial representation and clear visualization of the star network, showing the connections and distances between stars effectively.

A network of blue dots and lines

Description automatically generated

**Discussion of time complexity of generating both dataset**

The first script, **Q1\_dataset1.py**, involves generating datasets of various sizes by creating random numbers with several digits determined by the dataset size. For each dataset, the script generates n random numbers, where n is the size of the dataset, and writes them to a text file. The time complexity for generating each number **is dependent on the number of digits,** which is relatively small compared to n and can be considered constant. Therefore, the overall time complexity for generating and saving each dataset is 𝑂(𝑛).

Besides that, the second script**, Q1\_dataset2.py**, is designed to create a fixed number of star objects and establish connections between them to form a graph. The process of generating the stars and their attributes is fixed and independent of the input size, resulting in a time complexity of 𝑂(1) for creating the stars. Similarly, the creation of edges between the stars and the calculation of distances for each edge also involve a constant amount of work, as the number of stars and edges are fixed. Therefore, the overall time complexity for the second script is 𝑂(1) as all operations involve **a fixed and constant number of elements**.

Summary:

* **Q1\_dataset1.py:** O (N), where N is the size of the largest dataset.
* **Q1\_dataset2.py:** O (1), since the number of stars and edges are fixed.

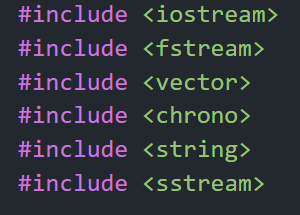
# **Question 2**

Before starting to solve the question, we need to know what we need to do. This question is separated into 2 parts which are Heap Sort and Selection Sort.

## Code Explaination

### Libraries used:

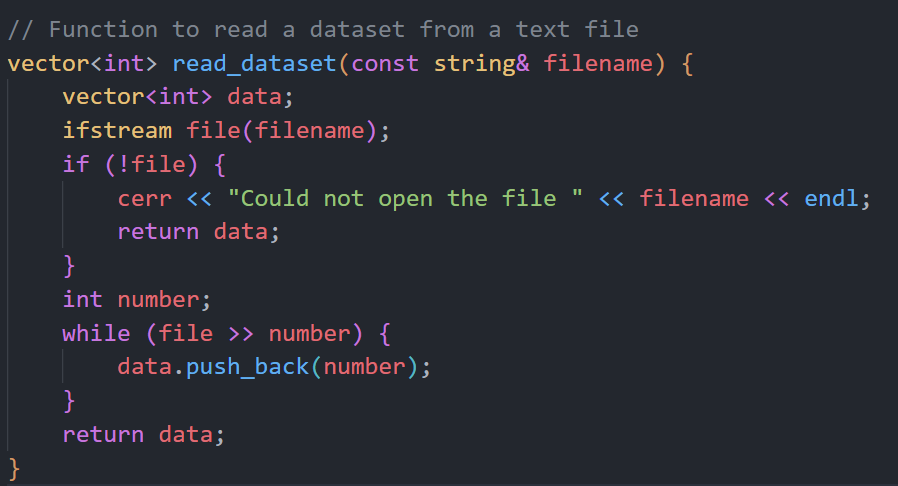
We using the programming language C++ to solve the question. Therefore, before solving the questions, we will need to import some required libraries. The libraries that we will be using are fstream, vector, chrono, string and sstream.



These are the imported libraries.

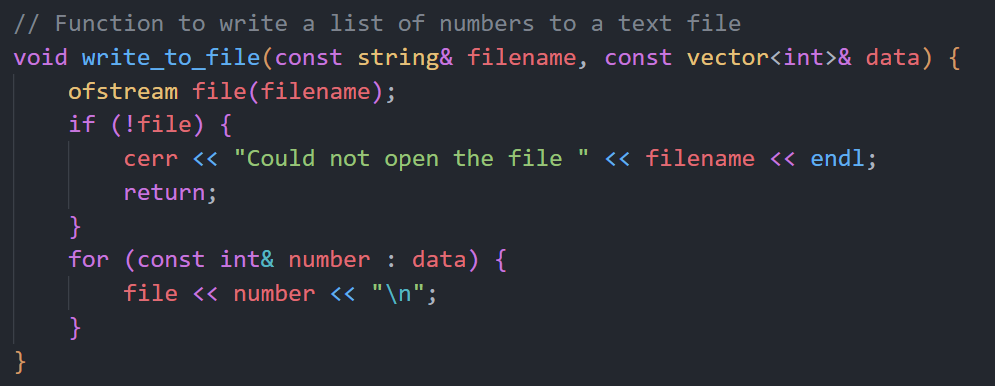
The program can support file operation thanks to the fstream package. Read, write, and create files are a few examples. Second, dealing with dynamic arrays will be supported by the vector package. Unlike a typical fixed size array, a dynamic array can be resized. The chrono package comes in third. We can measure the function's execution time with this package. Also included is the string package. Functions for manipulating strings are included in this package. Lastly, the sstream which is employed in string transformations.

### Function: read\_dataset



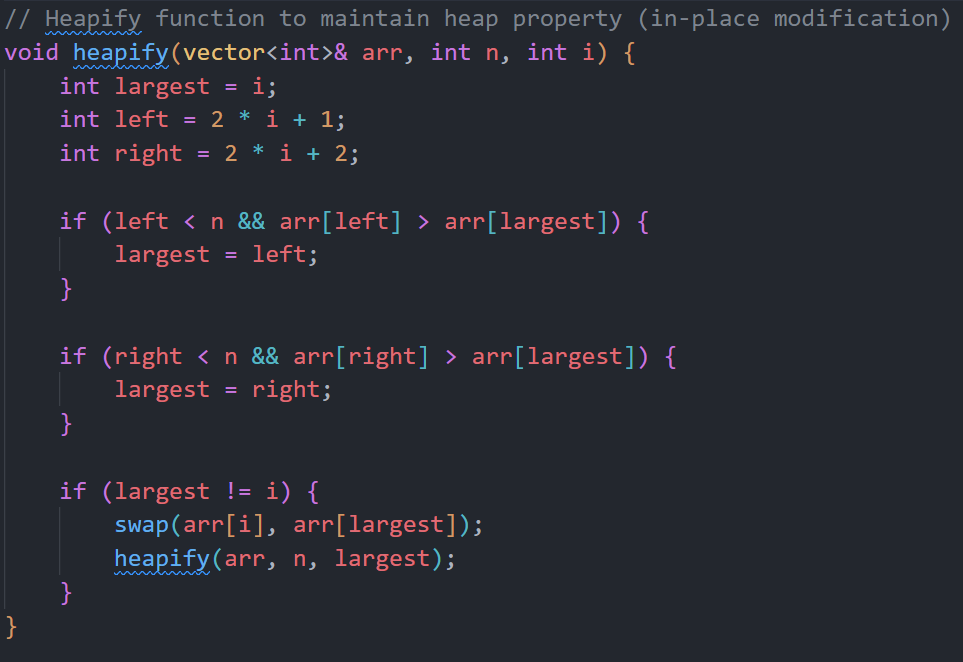
This is the C++ code which the function is to read the dataset from a text file. The name of the function is **read\_dataset**. This function is to read the dataset that has been randomly generated from Q1. The program will then check if the file name is equivalent to the size of the dataset. If it is not, the system will pop up “Could not open the file”. Else the system will read through the text and read all the numbers into the number variable using a while loop. The number will then be pushed back to the end of the data vector.

### Function: write\_to\_file



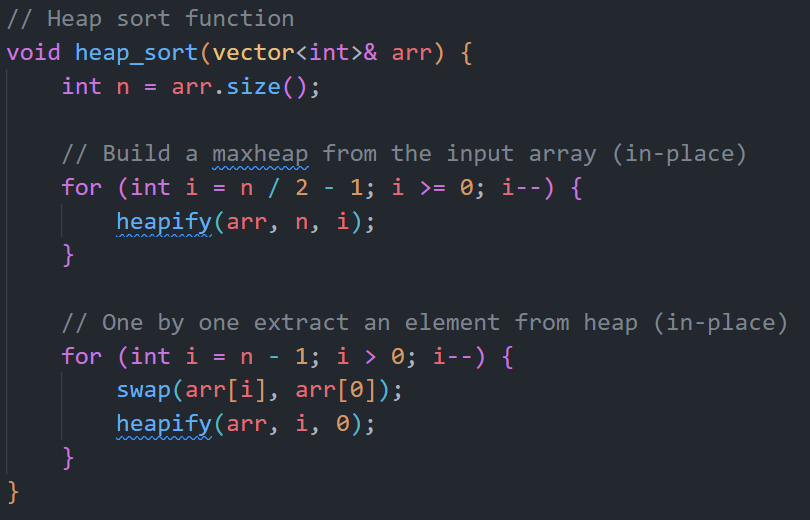
This function name is **write\_to\_file**. The program will check if the file is successfully opened. If the file is unable to be opened, the program will display “Could not open the file”. The for loop iterates each integer (number) into the data vector.

### Function: heapify



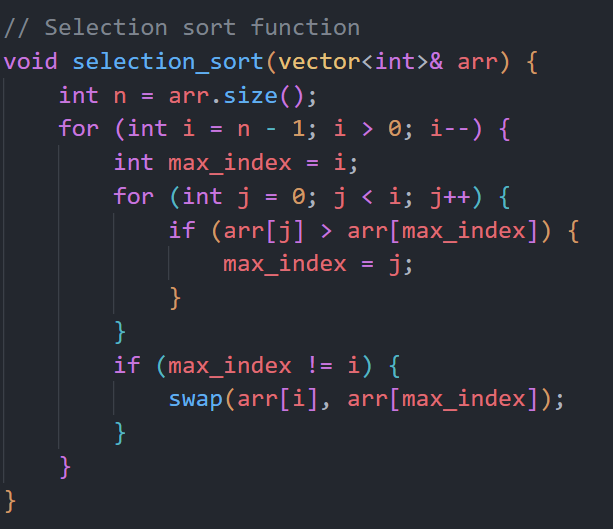
This is the heapify process code before performing heap sort. **arr** refer as a vector of integer, **n** is the size of the heap and **i** represent the index of the element in the heap that has changed. The function will find the largest element that will be stored in the largest which is set to the index of the current node **i.** The left and right are the variables that stores the left and right child node of the current node by using formulas. (2 \* i + 1) is for the left child node while (2 \* i + 2) is for the right child node. Then we will start to compare whether the left or the right node has the larger number compared to the current largest. If left or child node is larger than current largest, the largest variable will be updated to store the index of the child node. Then the program will swap the value of the current node and the largest element using swap function. The heapify function is called recursively on the subtree at the new largest element to ensure that the heap property is maintained.

### Function: heap\_sort



The first for loop in the picture above is to build a maxheap by iterating through the non-leaf nodes of the tree. The second for loop is to extracting elements from the heap by iterating through the array from the end and repeatedly swapping the root element which is the largest element with the last element in the unsorted part of the array then call the function heapify to maintain the max-heap property. The array is then sorted in descending order.

### Function: selection\_sort



This is the code for the function selection sort. There are 2 loops in the code. The outer loop iterates through the vector from the second to last element down to the first element. Each iteration, one element is placed in its correct position. While the inner loop iterates through the unsorted sub list which are the elements before the current **i**. It compares every element in the unsorted sub list with the current assumed maximum element and if a larger element is found, it will be updated to the **max\_index.** After the inner loop finishes execute, if the **max\_index** is not equal to **i**, it means a larger element was found in the unsorted. The element at **i** will then be swapped with the **max\_index** to place the larger element at the end of the unsorted sub-list. The process will repeat until the entire vector is completely sorted.

### Driver code

A screen shot of a computer code

Description automatically generated

The first line of code declares a vector named dataset\_sizes with type int, representing the sizes of datasets that will be processed by the program. A second vector, dataset\_filenames with type string, is used to store filenames corresponding to each dataset size. The for loop iterates over each integer in dataset\_sizes, constructing a filename string based on "dataset\_" + the string representation of the size + ".txt", and adds it to dataset\_filenames.

A screen shot of a computer program

Description automatically generated

This section of the code handles the heap sort for each dataset. It first reads the dataset from the file using the read\_dataset function. If the dataset is successfully read (i.e., it is not empty), the heap sort process begins, and the execution time is recorded. Once sorted, the dataset is saved to a file named "heap\_sorted\_" followed by the original dataset filename. The execution time for the heap sort enqueue and deques is then displayed.

A screen shot of a computer program

Description automatically generated

After completing heap sort on all datasets, the program then processes each dataset with the selection sort algorithm. It reads the dataset again, checks if it is not empty, and proceeds with the selection sort while recording the execution time. Once sorted, the dataset is saved under a filename prefixed by "selection\_sorted\_". The selection sort execution time for each dataset is displayed after sorting.

## Output

The output displays the sorting times for datasets ranging from 100 to 1,000,000 elements of Help sort enqueue, dequeue and selection time. Here are the details for each dataset:

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

The collected data will be used to analyze the time complexity of Heap Sort and Selection Sort. By plotting the times against the dataset sizes.

The Sorted Dataset will be saved as a .txt file and saved in the same folder. Below are the file example that both datasets will be saved as:

A screen shot of a computer

Description automatically generated

Heap Sort files

A screen shot of a computer screen

Description automatically generated

Selection Sort files

Below are the examples of the largest datasets (1,000,000) sorted in two different algorithms.

A screenshot of a computer

Description automatically generated

heap\_sorted\_dataset\_1000000.txt

A screenshot of a computer

Description automatically generated

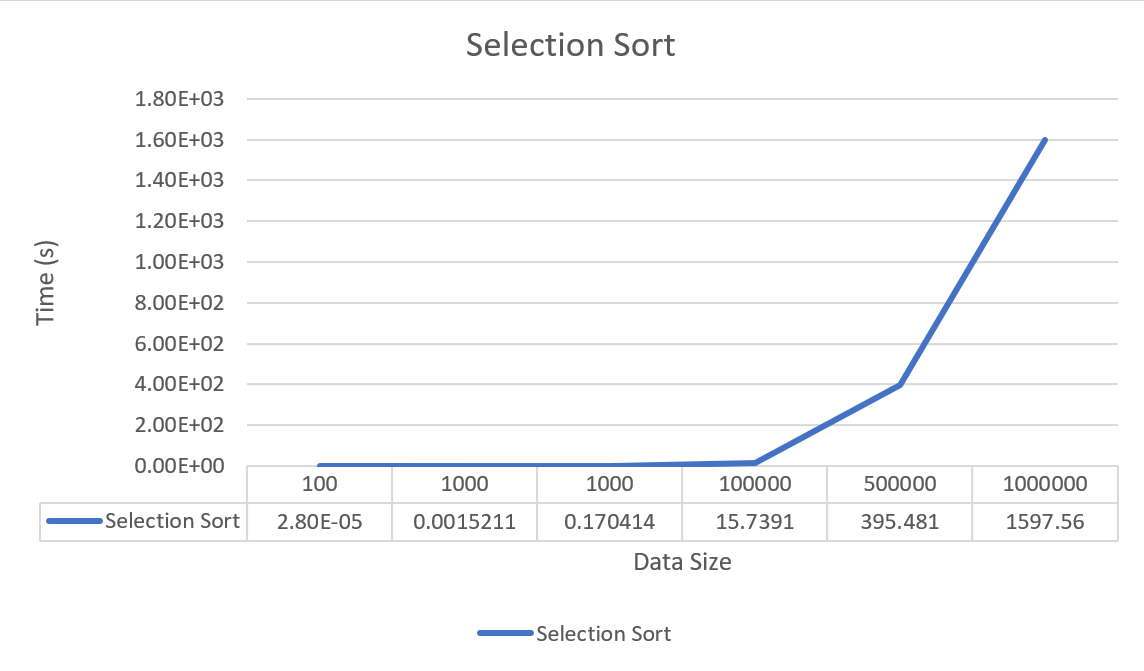
selection\_sorted\_dataset\_1000000.txt

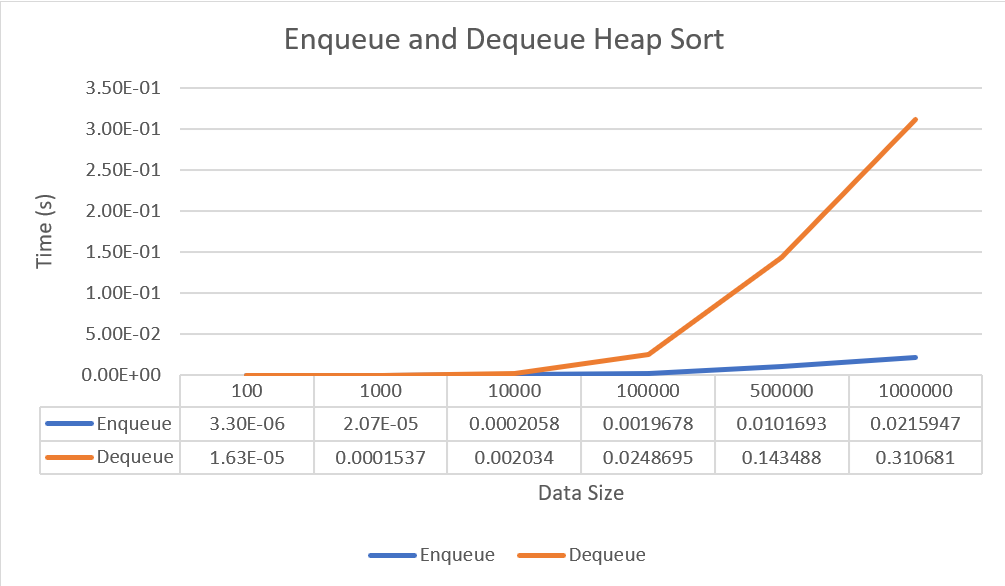
The output of both sorting algorithms (Heap Sort and Selection Sort) is the same, meaning they both correctly sort the datasets. However, there is a significant difference in the execution times:

**Heap Sort is considerably faster than Selection Sort for all dataset sizes tested.**

This output clearly shows that while both algorithms produce the same sorted output, Heap Sort is much **more efficient** in terms of execution time, especially as the size of the dataset increases. This efficiency makes Heap Sort a more suitable choice for larger datasets.

## Time Complexity

  
This is the graph of selection sort. It shows the relationship between the data size and time used to finish sorting. The **time complexity** of selection sort is **O(n^2)** while **space complexity** is **O(1)**. The time complexity of O(n^2) is because the outer loop iterates from n-1 down therefore it runs **n-1** times. The inner loop will iterate from 0 to i-1 where i is the current index of the outer loop. The worst case of the inner loop will be i times. One average, the inner loop iterates roughly n/2 times. It is because the largest element is moved to the correct position at the outer loop and reduces the size of the unsorted sub-array by 1. Therefore, the time complexity is approximately **(n-1) \* (n/2)** which is equal to O(n^2).

This is the graph of enqueue and dequeue of heap sort. It shows the relationship between time used for enqueue and dequeue. The time complexity of heap sort is O(n log n) while space complexity is O(1). The heapify process uses time complexity of O(log n) because it compares element at most log n levels deep and perform constant-time swaps. On the other hand, heap sort function called heapify to build the initial max heap where the time complexity will be O(n log n). Therefore, time complexity will be O(n log n).

# **Question 3**

This program identifies the shortest paths from Star A to the other stars using Dijkstra’s Algorithm, displays the shortest distance to each star, and stores the outputs in text files. It also draws the graph representing the shortest paths and discusses the time and space complexity of the algorithm. Additionally, Kruskal's Algorithm is implemented to find the Minimum Spanning Tree (MST) of the graph.

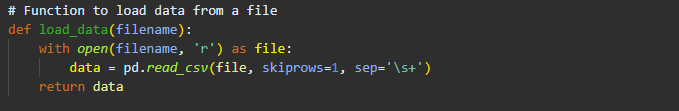
## Code Explanation (Dijkstra's Algorithm)

### Libraries used:



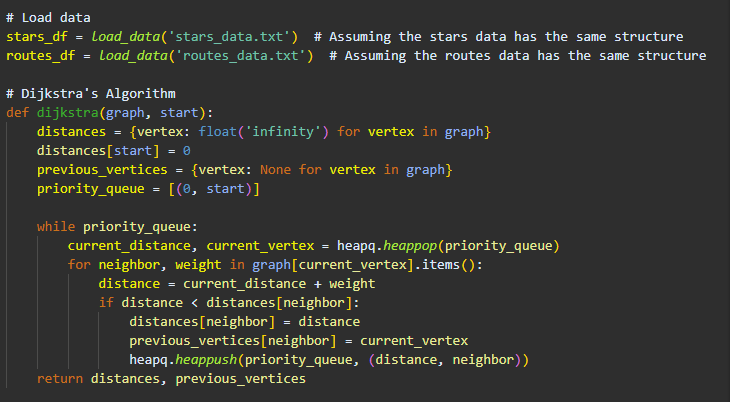
‘pandas’ is imported for data manipulation and analysis. ’heap’ provides an implementation of the heap queue algorithm, also known as the priority queue algorithm. ’matplotlib.pyplot’ is a plotting library used for creating static, interactive, and animated visualizations. 'networkx' is a library for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

### Function: load\_data



This function reads a text file into a pandas DataFrame. The skiprows=1 parameter indicates that the first row of the file should be skipped (assuming it contains headers).

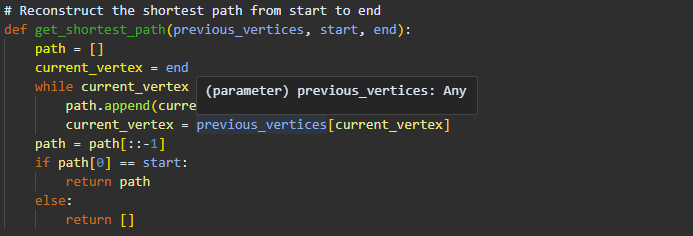
### Function: dijkstra

These lines load data from two files (’stars\_data.txt’ and ’routes\_data.txt‘) into DataFrames ’stars\_df’ and ’routes\_df’.

Dijkstra's algorithm is implemented in the dijkstra function, which calculates the shortest paths from a starting vertex to all other vertices in a graph. Initially, the function sets the distance to all vertices as infinity, except for the starting vertex which is set to zero. It also maintains a dictionary of previous vertices to reconstruct the shortest path later. A priority queue is used to select the vertex with the smallest distance at each step.

The algorithm iterates by popping the vertex with the smallest distance from the priority queue. For each neighbor of this vertex, it calculates the distance through the current vertex. If this new distance is smaller than the known distance, it updates the shortest distance and the previous vertex for this neighbor and pushes the updated distance into the priority queue

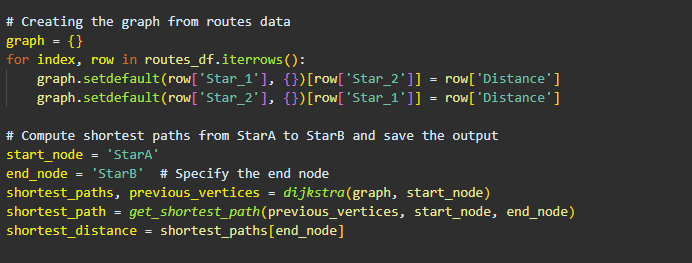
### Function: get\_shortest\_path

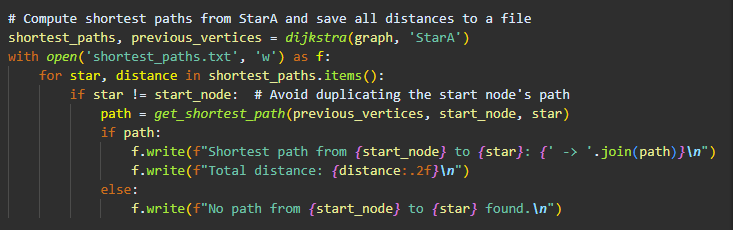
 To reconstruct the shortest path from the start vertex to any other vertex, the get\_shortest\_path function is used. This function traces back from the end vertex to the start vertex using the previous\_vertices dictionary, building the path in reverse order. Once the path is reconstructed, it is reversed to get the correct order from start to end.

This function reconstructs the shortest path from the start vertex to the end vertex using the’ previous\_vertices ’dictionary:

* ‘previous\_vertices’: Dictionary from Dijkstra's algorithm.
* ‘start’: The starting vertex.
* ‘end’: The ending vertex.
* ‘path’: List to store the path from start to end.
* Reverses the path list to get the correct order from start to end.

### Driver Code (Dijkstra's Algorithm)





The graph is created from the routes data, where each star is a key in the dictionary and its value is another dictionary containing neighboring stars and the distances to them. This representation is essential for efficiently accessing the distances between any two connected stars. The code creates a graph from the routes data:

* ‘routes\_df.iterrows()’: Iterates over the rows of the DataFrame.
* ‘setdefault’: Adds the key with an empty dictionary if it does not exist.
* Adds edges with distances between stars to the graph.

This code computes the shortest paths from start\_node (StarA) to all other stars and saves the results to a file:‘shortest\_paths’: Shortest distances from StarA.‘previous\_vertices’: Previous vertices for each vertex in the shortest path.At last,writes the shortest path and distance to shortest\_paths.txt. For each star, it reconstructs the shortest path and writes the path and total distance to the file. If no path is found, it notes this in the file.

## Output (Dijkstra’s Algorithm)

A close-up of a network

Description automatically generated

Output Txt File:

A screenshot of a computer program

Description automatically generated

## Time Complexity (Dijkstra’s Algorithm)

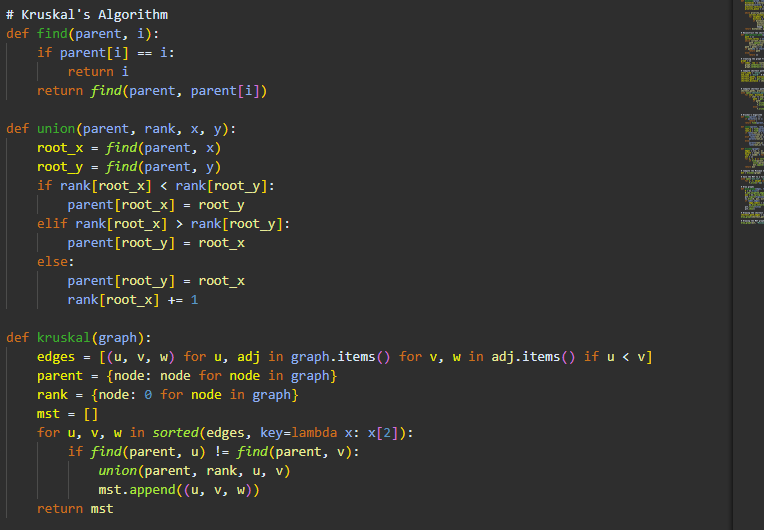
The priority queue's data structure determines how time-consuming Dijkstra's algorithm is. The complexity of a basic array is O(V2)O(V^2)O(V2). The complexity decreases to O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV) when a binary heap is used, with VVV representing the number of vertices and EEE representing the number of edges.

Space complexity of the algorithm.

O(V+E)O(V + E)O(V+E) is the space complexity of Dijkstra's algorithm. This comprises the amount of space needed to hold the priority queue, the distance array, and the graph (adjacency list or matrix).

## Code Explanation (Kruskal's Algorithm)

### Function: find, union and kruskal



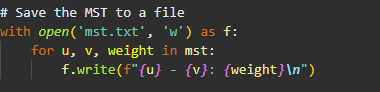
We use ‘find’ to find the root of the set containing ‘i’ and ‘union’ to unions two sets containg ‘x’ and ‘y’.In order to find the Minimum Spanning Tree(MST) of a graph, The first line creates a list of all the edges in the graph. Each edge is represented as a tuple (u, v, w), where u and v are the vertices and w is the weight of the edge. The condition if u < v ensures that each edge is only included once in the list (assuming the graph is undirected).

The parent dictionary is where each node is its own parent initially. This is used for the union-find data structure to keep track of the connected components. The rank dictionary keeps the rank (an upper bound on the height) of the trees in the union-find data structure. Initially, the rank of each node is 0. The mst list is empty at first and will store the edges of the minimum spanning tree.

The sorted(edges, key=lambda x: x[2]) line sorts the edges by their weight in non-decreasing order. For each edge (u, v, w) in the sorted list:

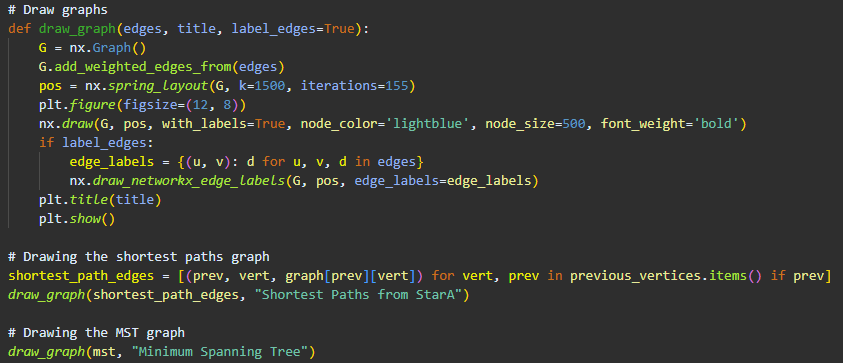
* ‘find(parent, u) != find(parent, v)’: Checks if the vertices ‘u’ and ‘v’ are in different connected components. The ‘find’ function returns the root of the set containing the vertex.
* If they are in different components,’ union(parent, rank, u, v)’ unites the two components (by attaching one tree under the root of the other) and adds the edge ‘(u, v, w)’ to the’ mst’ list.

Finally, the function returns the mst list, which contains the edges of the minimum spanning tree.



This code computes the MST of the graph and saves the result to a file.

### Driver code (Kruskal's Algorithm)



The function ‘draw\_graph’ is designed to visualize a graph using the ‘networkx’ and ‘matplotlib’ libraries. The function takes three parameters: ‘edges’, ‘title’, and an optional ‘label\_edges’ parameter which defaults to ‘True’.

First, the function creates a new graph’ G’ using ‘networkx’. It then adds the edges to the graph with the weights using ‘G.add\_weighted\_edges\_from(edges)’. The edges are expected to be in the form of a list of tuples, where each tuple is (u, v, w), representing an edge between vertices u and v with weight w.

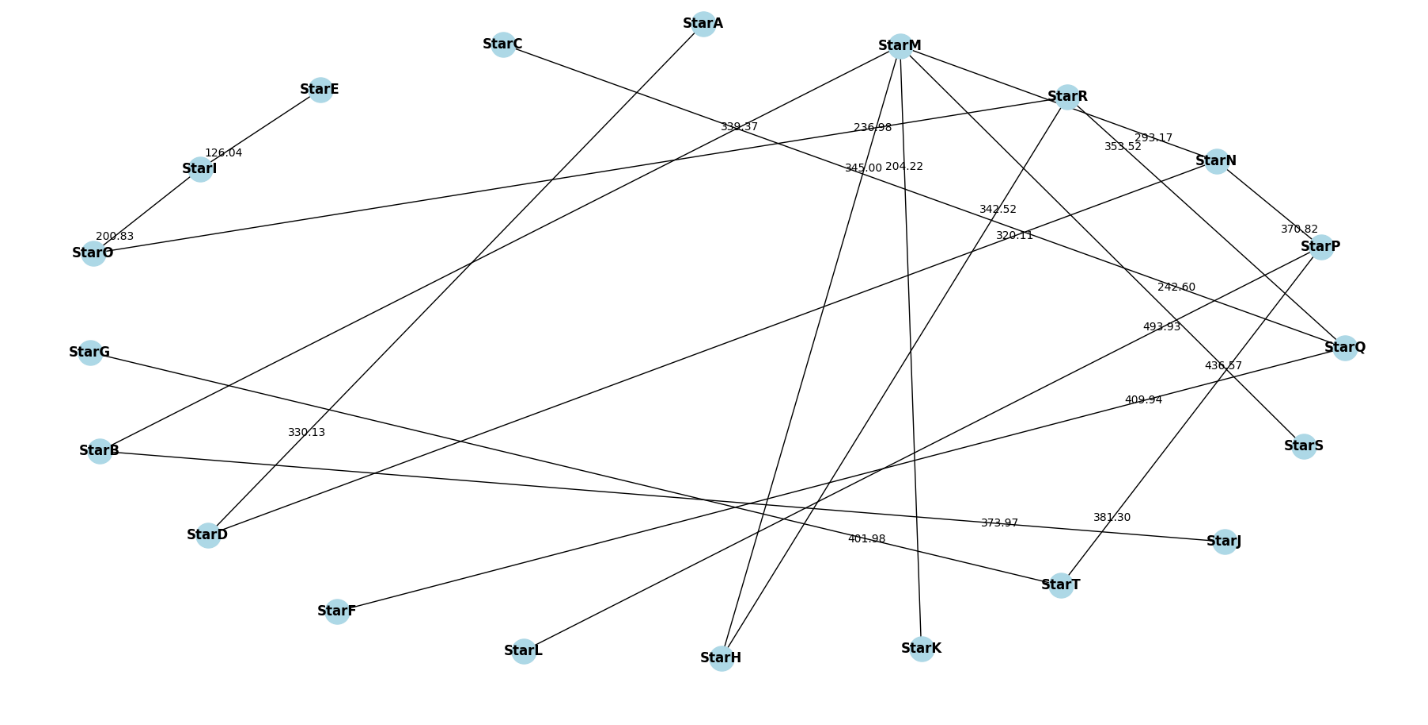
The position of the nodes in the graph is determined using ‘nx.spring\_layout(G, k=1500, iterations=155)’, which positions nodes using the Fruchterman-Reingold force-directed algorithm. This helps in generating a visually appealing layout for the graph.

The graph is plotted using matplotlib. The figure size is set to 12x8 inches with ‘plt.figure(figsize=(12, 8))’. The graph is drawn with labels using’ nx.draw’, where nodes are colored light blue, have a size of 500, and the labels are in bold font.

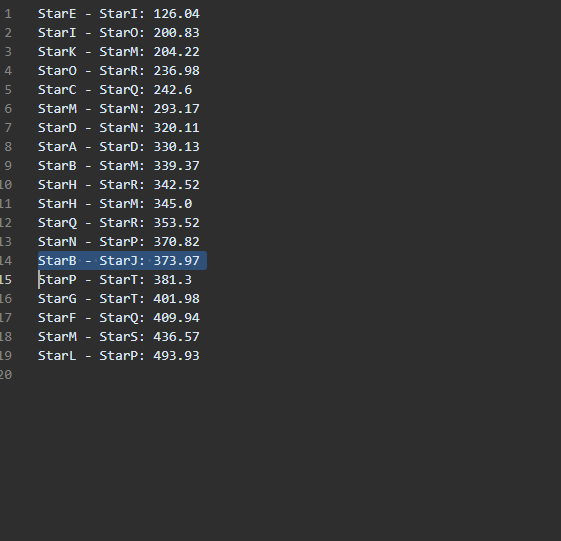
If ‘label\_edges’ is True, the edge labels are created as a dictionary where the keys are the edges (u, v) and the values are the weights d. These labels are then drawn on the graph using ‘nx.draw\_networkx\_edge\_labels’.

The title of the plot is set using’ plt.title(title)’, and the plot is displayed with ‘plt.show()’.

## Output (Kruskal Algorithm)



Output Txt File:



## Time Complexity (Kruskal's Algorithm)

Kruskal's algorithm's time complexity is mostly determined by the edge sorting and union-find procedures. O(E log E), where E is the number of edges, is needed to sort the edges. With α representing the inverse Ackermann function, which develops very slowly and is virtually constant in real-world circumstances, the union-find procedures (find and union) have an essentially constant time complexity of O(α(V)) per operation. As a result, the algorithm's overall time complexity is O(E log E).

Space complexity of the algorithm.

O(E+V)O(E + V)O(E+V) is the space complexity of Kruskal's method. This contains storage for the MST itself, the union-find data structure (used for managing vertex sets), and the edge list (used for sorting).

# **Question 4**

Q4 file implements the 0/1 Knapsack algorithm to help participants in a space conquest scenario maximize their profits by selecting the optimal set of stars to visit without exceeding their spaceship's weight capacity of 800 kg. The program reads star data, including names, weights, and profits, from an input file. It then uses dynamic programming to determine the maximum profit achievable and identifies the specific stars to visit. The results, including the decision matrix, are saved to output files for detailed analysis. This implementation ensures efficient and effective decision-making for maximizing profits while adhering to the weight constraints.

## Code Explaination

### Libraries used:

**A screenshot of a computer

Description automatically generated**

NumPy provides the tools for efficient numerical computations required for the dynamic programming solution. The CSV module ensures that the DP table can be saved and reviewed in a clear and structured format. The time module helps in performance measurement, allowing for a better understanding of the algorithm's efficiency

### Function: load\_star\_data

A screen shot of a computer code

Description automatically generated

The load\_star\_data function is responsible for reading the star data from a file. It skips the first two lines of the file (assumed to be headers or irrelevant information) and then reads each subsequent line to extract the name, weight, and profit of each star.

### Function: knapscak

The primary purpose of the knapsack function is to find the optimal set of stars to visit that maximizes the total profit while keeping the total weight within a specified capacity.

A screen shot of a computer program

Description automatically generated

The function signature includes into parameter it functions as:

* weights: A list of integers representing the weights of the stars.
* profits: A list of integers representing the profits of the stars.
* capacity: An integer representing the maximum weight capacity of the spaceship.



First step, the function initializes a dynamic programming table (DP table) with dimensions based on the number of stars and the capacity. Specifically, it sets up a 2D array initialized with zeros, where the rows represent the stars, and the columns represent capacities from 0 to the given capacity.

A black background with white text

Description automatically generated

The nested loops in the function fill this DP table by iterating over each star and each possible capacity. If the weight of the current star is less than or equal to the current capacity, the function computes the maximum profit by considering whether to include or exclude the star.

If the weight of the current star weights[i-1] is less than or equal to the current capacity w, the function computes the maximum profit by considering two scenarios:

* Excluding the current star: dp[i - 1][w].
* Including the current star: dp[i - 1][w - weights[i - 1]] + profits[i - 1].

If the weight of the current star exceeds the current capacity, the star is excluded: dp[i][w] = dp [i - 1][w].

A screenshot of a computer

Description automatically generated

After filling the DP table, the function traces back through the table to determine which stars are included in the optimal solution.

* Starting from the last cell of the DP table, it checks if the value at dp[i][w] is different from the value at dp[i - 1][w]. A difference indicates that the current star was included.
* The index of the included star is added to the items list, and the remaining capacity w is reduced by the weight of the included star.
* This process continues until all included stars are identified.



Finally, the function returns the filled DP table and the list of indices of the included stars. By constructing a DP table to keep track of the maximum profit for each sub-problem and tracing back through the table to identify the optimal set of stars to visit, the knapsack function ensures that the solution maximizes profit while adhering to the weight constraints, making it a crucial component of the program.

### Function: save\_knapsack\_result

A screen shot of a computer code

Description automatically generated

The save\_knapsack\_result function is designed to save the results of the knapsack algorithm to a text file. It writes the maximum profit obtained, the names, weights, and profits of the selected stars, and the total weight and profit of the selected stars. Additionally, it records the execution time of the knapsack computation. This function ensures that the results are clearly documented and can be reviewed later for analysis.

### Function: save\_knapsack\_result

A screen shot of a computer screen

Description automatically generated

The save\_knapsack\_table\_to\_csv function saves the entire DP table generated by the knapsack algorithm to a CSV file. This function creates a structured format for the DP table, with the first row representing the capacities and subsequent rows representing the items and their respective values in the table. This CSV file allows for a detailed examination of the intermediate values and decisions made during the dynamic programming process. Print this out is for us easy to check and determine any error.

### Driver Code

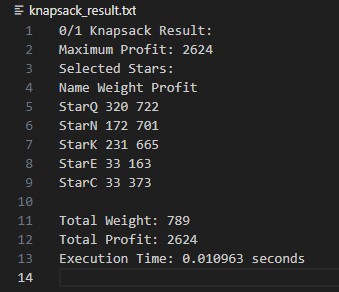
A screen shot of a computer program

Description automatically generated

The main block of the code orchestrates the execution of these functions. It starts by loading the star data using the load\_star\_data function. It then extracts the weights and profits of the stars and defines the maximum capacity of the knapsack. The main block measures the execution time of the knapsack algorithm by calling the knapsack function, then saves the results and the DP table using the save\_knapsack\_result and save\_knapsack\_table\_to\_csv functions, respectively. This sequence ensures that all necessary data is processed, analyzed, and saved efficiently and comprehensively.

## Output

The output of the 0/1 Knapsack algorithm applied to the star data from the provided dataset reveals the optimal set of stars to visit for maximum profit without exceeding the spaceship's weight capacity of 800 kg. Using the dataset from Q1 dataset 2, the algorithm identifies the following stars as part of the optimal solution: StarQ, StarN, StarK, StarE, and StarC.

  
*Output*

The maximum profit achieved is 2624. This value represents the highest possible profit that can be obtained by selecting a subset of stars whose total weight does not exceed the capacity of the spaceship. The stars included in the optimal set are:

* StarQ: Weight 320 kg, Profit 722
* StarN: Weight 172 kg, Profit 701
* StarK: Weight 231 kg, Profit 665
* StarE: Weight 33 kg, Profit 163
* StarC: Weight 33 kg, Profit 373

These stars collectively offer the highest profit while their combined weight remains within the capacity limit.

**Total Weight and Profit**:

* **Total Weight**: 789 kg. This is the sum of the weights of the selected stars and is just below the maximum allowed capacity of 800 kg.
* **Total Profit**: 2624. This is the sum of the profits from the selected stars and represents the optimal profit obtained.

The execution time of 0.010963 seconds demonstrates the efficiency of the dynamic programming approach. This result highlights the effectiveness of the 0/1 Knapsack algorithm in making optimal decisions within resource constraints, as illustrated by the given dataset.

A CSV file will also provide as what shows below. The table's primary purpose is to allow us to verify the correctness of the algorithm by showing the maximum profit achievable for each weight capacity. It helps in analyzing the incremental benefit of adding each star to the knapsack and in identifying the optimal solution for the full capacity of 800 kg. The table's structure includes a header row with capacity values from 0 to 800 kg and rows representing each star with its weight and profit, where the cells show the maximum profit achievable for each capacity with the given stars.

A screenshot of a computer

Description automatically generatedThis comprehensive view of the algorithm's calculations ensures clarity and facilitates easy verification and understanding of the results.

A screenshot of a table

Description automatically generatedThe rows in the table represent different stages of including items in the knapsack. Each row corresponds to a specific item and shows how the inclusion of that item affects the maximum profit for each weight capacity from 0 to 800 kg. The format of the entries in the "Item/Capacity" column is "index (weight, profit)," where "index" is the position of the item in the list, "weight" is the weight of the item, and "profit" is the profit associated with the item.

For example, if the entry is "1 (650,325)," it means this row is considering the first item, which has a weight of 650 kg and a profit of 325. The corresponding cells in this row show the maximum profit that can be achieved for each weight capacity when including this item in the knapsack.

## Time Complexity

The time complexity of the 0/1 Knapsack algorithm implemented in your **code is 𝑂(𝑛×𝑊)**, where 𝑛 is the number of stars (items) and 𝑊 is the maximum weight capacity of the knapsack. The time complexity 𝑂(𝑛×𝑊) of the 0/1 Knapsack algorithm indicates that the algorithm scales linearly with both the number of items and the maximum weight capacity. This is generally efficient for problems where the product of 𝑛 and 𝑊 is manageable. However, for very large 𝑛 or 𝑊, the algorithm may become computationally expensive, as the size of the DP table and the number of operations grow quadratically.

In our implementation, where 𝑛 represents the number of stars and 𝑊 is 800 (the maximum weight capacity of the spaceship), the algorithm performs efficiently within this constraint. The execution time of approximately 0.010963 seconds demonstrates that the dynamic programming approach is effective and can handle the problem size well.

In conclusion, the 𝑂(𝑛×𝑊) time complexity of your knapsack implementation reflects a balance between computational efficiency and problem-solving capability, making it suitable for practical applications within the given constraints.