# CCP 6214 ALGORITHM DESIGN AND ANALYSIS

## ASSIGNMENT PRESENTATION SLIDES


TC4L T15L GROUP 7

Q1: 1211101007 | AISYAH BINTI AHMAD KASSIM
Q2: 1211200107 | AFIEZAR ILYAZ BIN ALFIE ISKANDAR
Q3: 1221303660 | WONG WAI YEE
Q4: 1211202025 | ABDULLAH BIN KAMARUDDIN (LEADER)

# ALGORITHM

| | |
|---|---|
| Q1 | DATASET 1 AND 2 |
| Q2 | HEAP AND SELECTION SORT |
| Q3 | DIJKSTRA’S AND KRUSKAL |
| Q4 | 0/1 KNAPSACK |

# Q1 DATASET 1 AND 2

# DATASET 1

Write an algorithm to generate 6 sets of data with random numeric seeding.
Use group leader's ID as the random seed reference

## 1. Extract Unique Digits

- Extracts unique digits from the given seed.
- Converts the seed into its individual digits and stores them in a set to ensure each digit is unique. The set is then converted to a vector.

## 2. Generate Dataset

- This function takes a seed for the random number generator, the size of the dataset, and a vector of allowed digits.
- It initializes the random number generator with the given seed and reserves space for the dataset.
- For each number to be generated, it constructs a 3-digit number by randomly selecting digits from the allowed_digits vector.
- It then adds the generated number to the dataset.

## 3. Save Dataset

- Writes each number in the dataset to the file, each on a new line.

## 4. Save & Generate Dataset

- Uses extract_unique_digits to get the allowed digits from id_leader.
- Iterates through the predefined dataset sizes
- Constructs a filename based on the current index.
- Generates a dataset using the generate_dataset function, with the seed incremented by the current index and the corresponding dataset size.
- Saves the dataset to the constructed filename using the save_dataset function.

# DATASET 2

Write an algorithm to generate a dataset for 20 locations of stars with 54 routes that connect between the stars. Each star connects to at least 3 other stars. Use the sum of other group members' ID numbers to generate data consisting of values for a star name, x, y, z-coordinate, weight and profit.

## 1. Star Structure

- Defines a Star with a name (char), three coordinates (x, y, z as double), and two attributes (weight and profit as int).

## 2. Extract Unique Digits

- Extracts unique digits from the given seed.
- Converts the seed into its individual digits and stores them in a set to ensure each digit is unique. The set is then converted to a vector.

## 3. Generate Unique Number

- Uses a custom random function to select digits from a set of allowed digits and constructs a number with a specified number of digits.

## 4. Generate Star

- Extracts unique digits from a sum of IDs
- Seeds the random number generator
- Generates stars with random attributes using these digits.

## 5. Calculate Distance Between Two Stars

- This function calculates the Euclidean distance between two stars based on their coordinates.

## 5. Generate Routes

- Uses a while loop to repeatedly generate random pairs of star names.
- Checks ensure that each route is unique and not duplicated.
- Increments a counter (connections) until exactly 54 routes are generated.

# MAIN.CPP

Provide options for user to choose either to generate dataset 1 or dataset 2

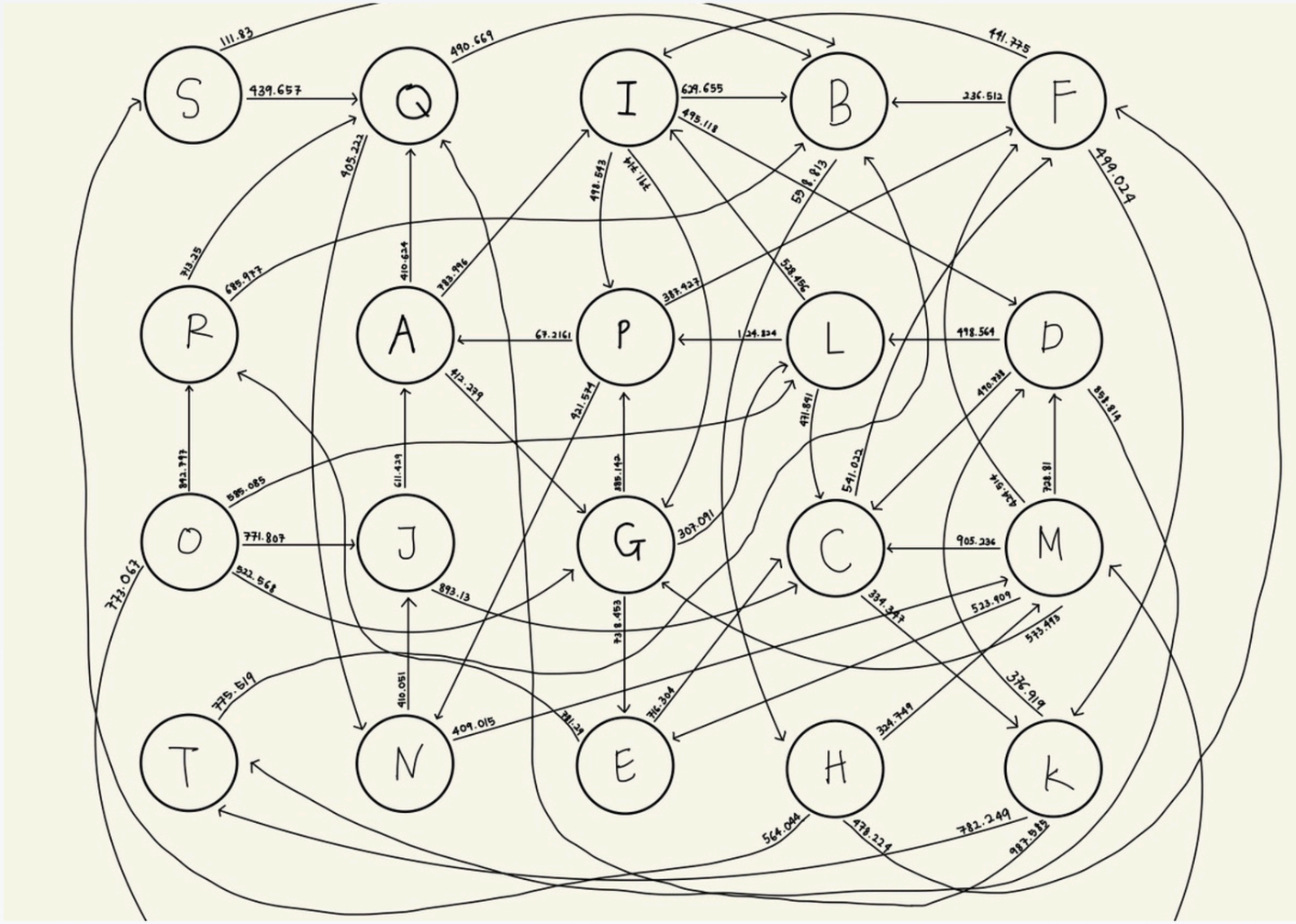
## 1. Generate Dataset 1

- Calls the generate_and_save_datasets() function.

## 2. Generate Dataset 2

- Sum of IDs: Calculates the sum of IDs
- Star Generation: Calls generate_stars function.
- Route Generation: Calls generate_routes function.
- Saving Dataset: Calls save_star_dataset function.

# DATASET 2 GRAPHS



# Q2 HEAP AND SELECTION SORT

# PROBLEM STATEMENT

Store the data in a priority queue using a heap and selection sort. Store sorted dataset(s)/output(s) in e.g. txt file(s).

## TASK 1

Record the time to insert all data into the priority queue. Dequeue the data to test the timing of the priority queue.

## TASK 2

Plot the graph of timing vs dataset size and discuss about the time and space complexity of the algorithm.

## TASK 3

# SORTING ALGORITHM

**Heap Sort and Selection Sort**

**1**
**Creates vectors for all dataset sizes**

**2**
**Uses a loop to iterate through each datasets**

INITIALISATION

**1**
**In a loop, code reads a dataset (readDataset)**

**2**
**Fills vectors with integers from read files**

READING

**1**
**Performs Heapify and Heap Sort**

**2**
**Performs Selection Sort**

EXECUTION

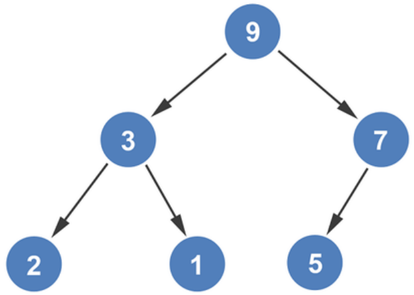
**1**
**Records the duration of time to complete**

**2**
**Prints the timing information**

TIME / OUTPUT

# FUNCTION EXPLANATION

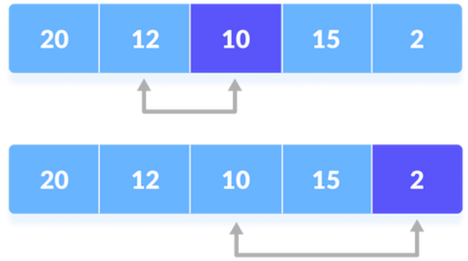
## *Heap Sort*




- readDataset: Reads integers from txt files, stores in vector
- writeSortedDataset: Sorted vector is put into a txt file
- heapify: Ensures that root is greater or equal to its children
- heapSort: Sort the vector in ascending order

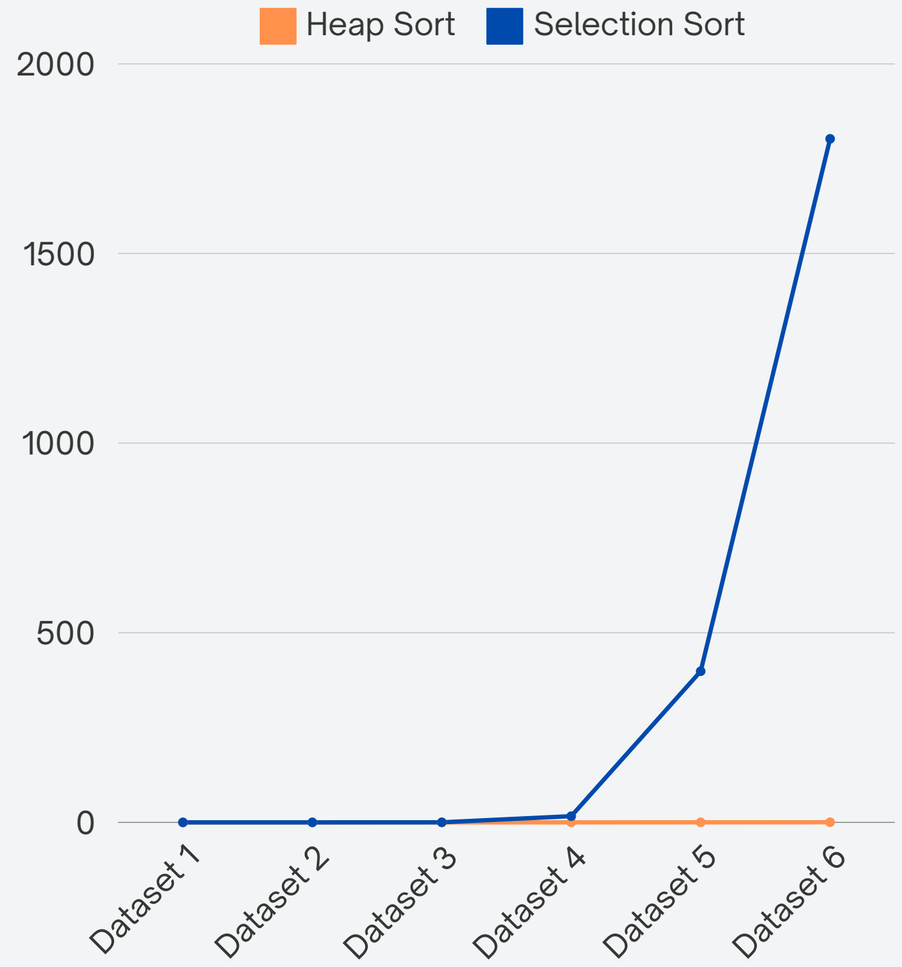
- readDataset: Reads integers from txt files, stores in vector
- writeSortedDataset: Sorted vector is put into a txt file
- selectionSort: Sorts the vector of integers in ascending order using the selection sort algorithm.

## *Selection Sort*



# GRAPH

Time taken to complete sorting for each dataset for Heap Sort and Selection Sort



# RESULTS AND DISCUSSION

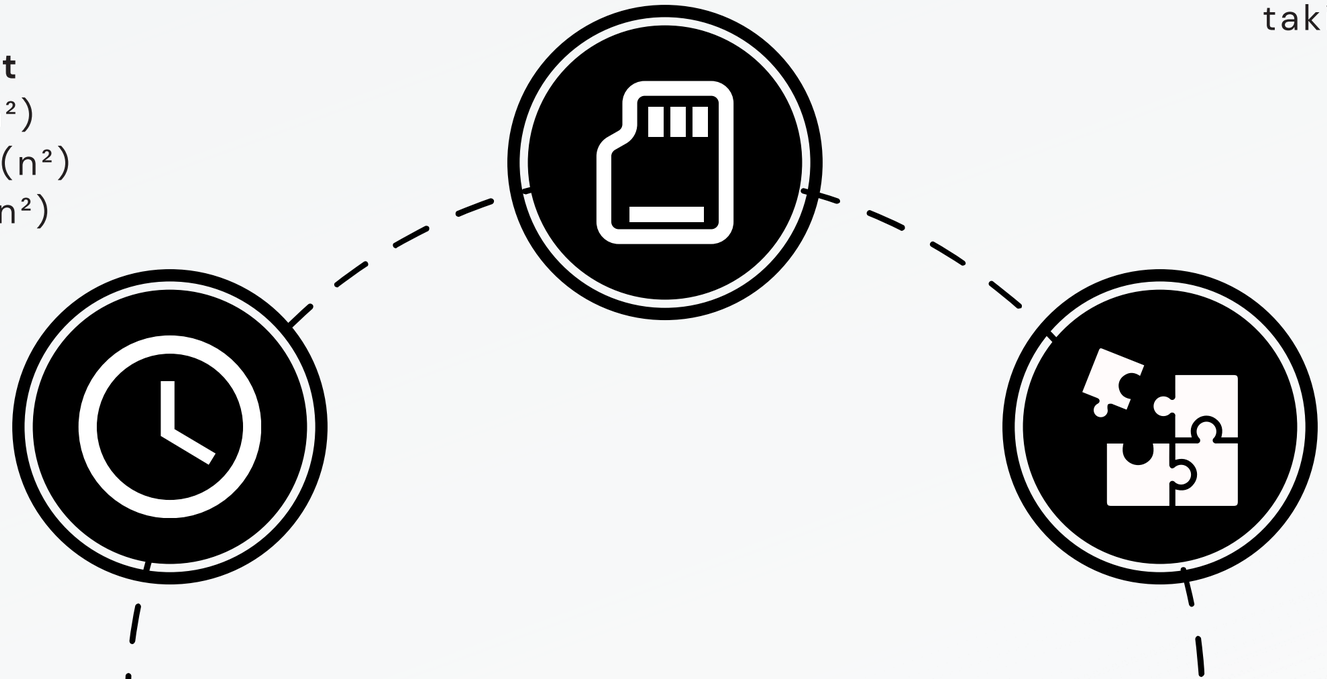
## Time Complexity

**Heap Sort**
Best Case: O(nlogn)
Average Case: O(nlogn)
Worst Case: O(nlogn)

**Selection Sort**
Best Case: $O(n^2)$
Average Case: $O(n^2)$
Worst Case: $O(n^2)$

## Space Complexity

Heap Sort: O(1)
Selection Sort: O(1)



## Conclusion

Heap Sort is far better than Selection Sort, since it has a better Time Complexity, taking less time to sort datasets

# Q3 DIJKSTRA'S AND KRUSKAL

# DIJKSTRA_OPERATIONS.H

## 1. Stars Generation

- DijkstraStar-identify star name, coordinates in x,y,z
- Edge-destination star, distance

## 2. Stores shortest distance and path

- PathInfo- stores shortest distance to star and vector of characters representing the path
- dijkstra_calculate_distance-calculate Euclidean distance between 2 stars in 3d space with parameters (x1, y1, z1, x2, y2, z2)

## 3. Read Star Dataset

- read_star_dataset- reads data from a file to populate vector of stars
- Starts by attempting to open the file specified by ‘filename’
- Each line is stored in ‘line’ String
- istringstream object-parse each line and read into ‘type’ String to determine if the line describe star or route

## 4. Determine whether Type is Star or Route

- If ‘Type’ is Star- remaining data in line (name, coordinate) is read into DijkstraStar Struct created and push to stars vector
- If ‘Type’ is Route - Edge struct created and push to graph for current and destination star

# DIJKSTRA_OPERATIONS.H

## 5. Starting Node

- char start- store starting node
- vector<Edge>> &graph- graph represented as adjacency list where each key is a node and the value is a list of edges connected to the node
- An unordered map ‘distances’ is initialized to store shortest distance from start node to each node
- Initially, all distances set to inifinity
- Set distance to start node as 0

## 6. Comparison in Priority Queue

- A priority queue ‘pq’ is initialized
- Start node is pushed to pq with distance 0
- Main loop runs while !pq.empty()
- Node with smallest distance(‘current’) is extracted from pq
- If current distance is greater than known shortest distance, the node is skip

## 7. Explore Neighbours of current node

- For each neighbour, ‘new_distance’ is calculated as the sum of ‘current_distance’ and ‘edge_distance’
- If ‘new_distance’ is shorter than previous known distance, the distance and path are updated and push into pq
- Function returns shortest distances and paths once all nodes have been processed

## 8. Save distance into file

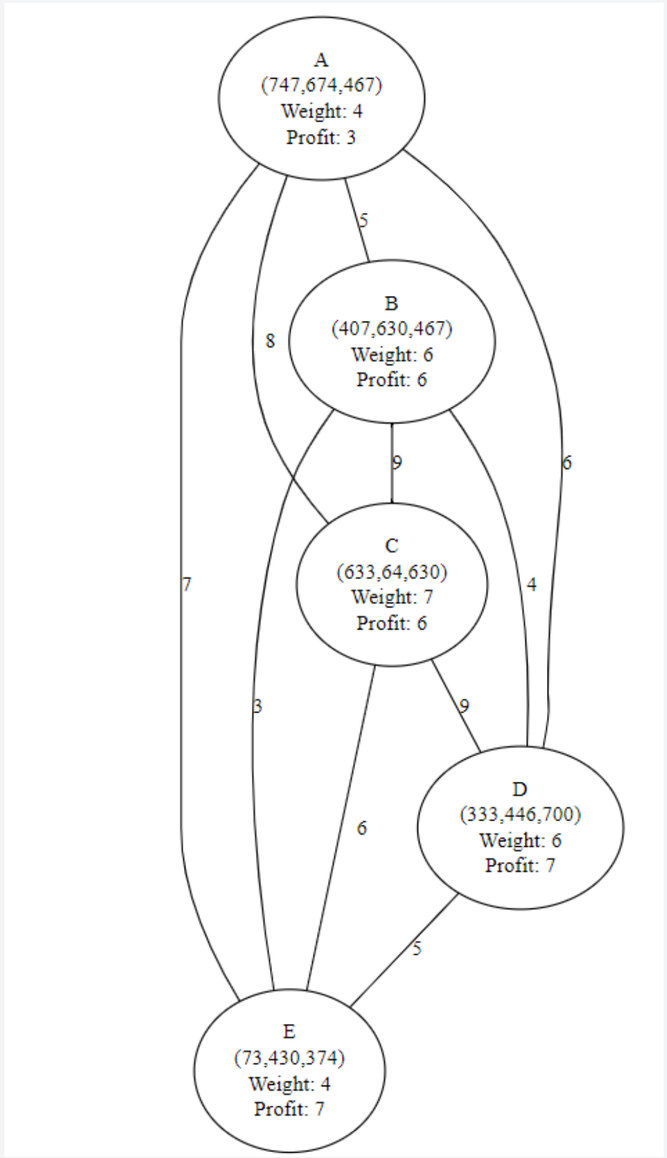
- Open file specified by ‘filename’
- Iterates through a for loop through each entry (entry.first- destination node, entry.second.distance-shortest distance to destination node, entry.second.path-path to reach destination node)
- Write each node to the file

# SHORTEST PATH

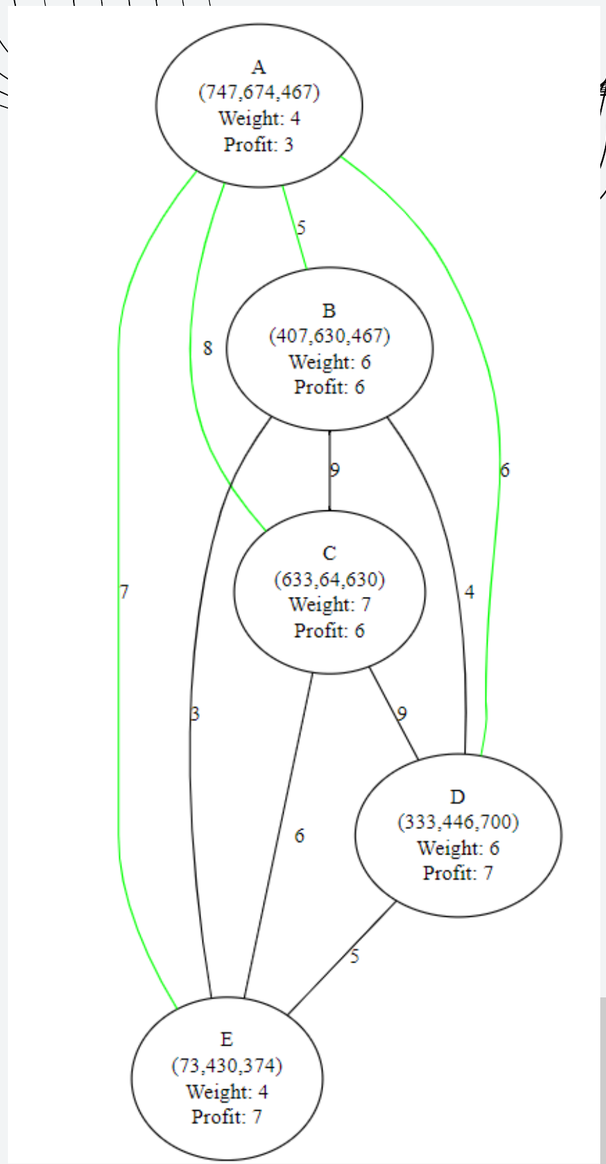
## 5 DATASET

Q3 Dijkstra and Kruskal > Dijkstra 5 dataset > shortest_paths.txt

```
1  Shortest distance from Star A to Star A is 0
2  Shortest distance from Star A to Star B is 5
3  Shortest distance from Star A to Star C is 8
4  Shortest distance from Star A to Star D is 6
5  Shortest distance from Star A to Star E is 7
6
```




BEFORE



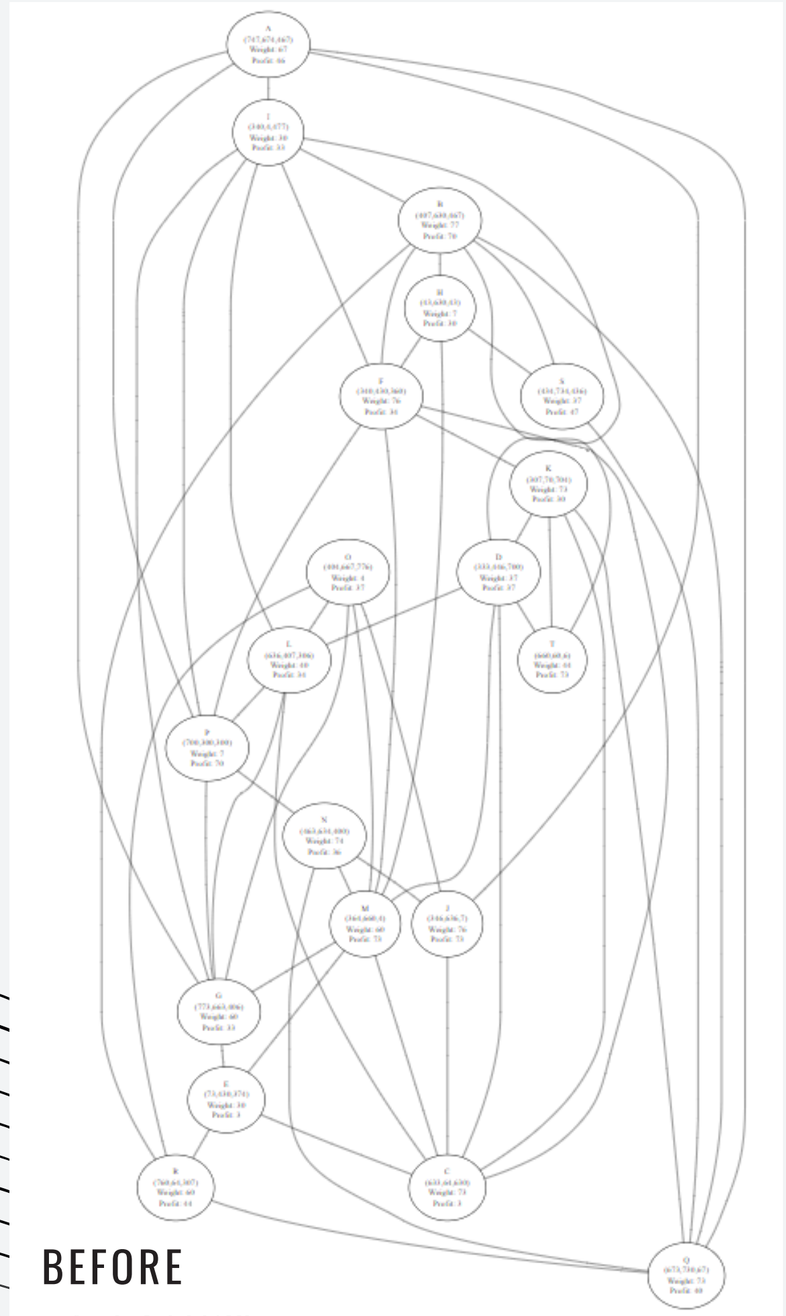

AFTER

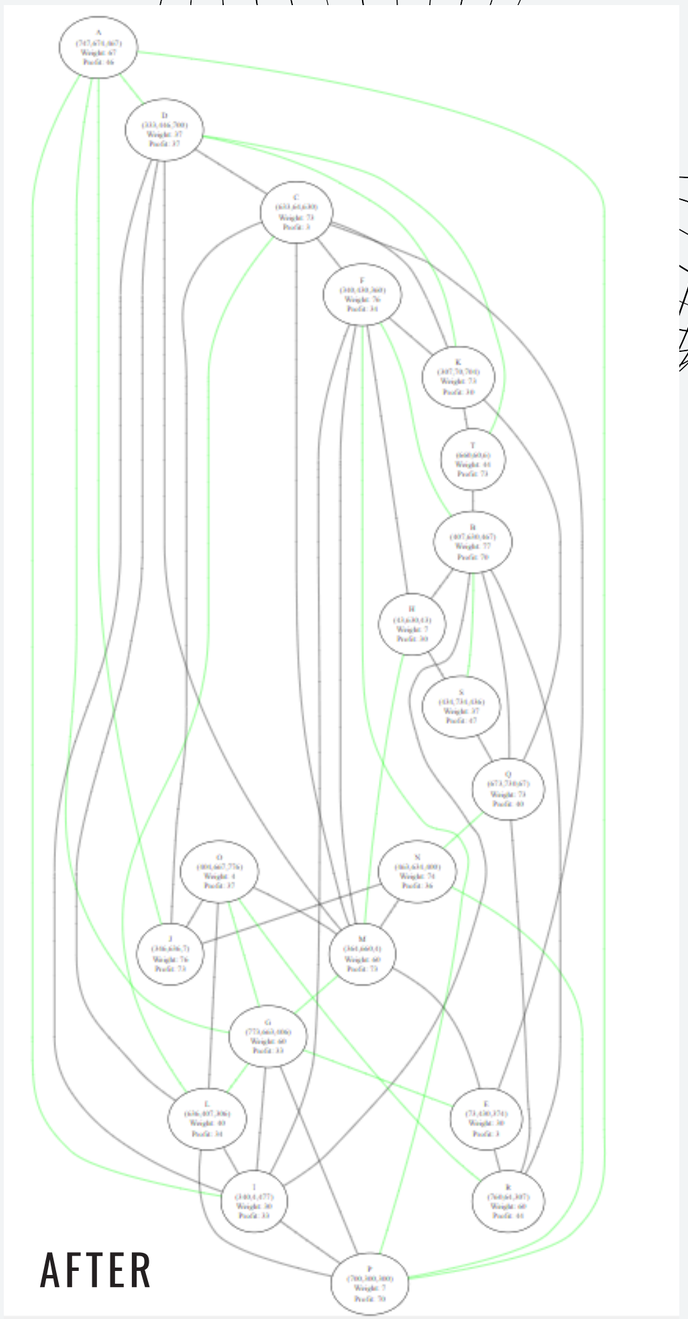
# SHORTEST PATH

Q3 Dijkstra and Kruskal > Dijkstra > shortest_paths.txt

```
Shortest distance from Star A to Star A is 0, path: {A}
Shortest distance from Star A to Star B is 1036.22, path: {A, P, F, B}
Shortest distance from Star A to Star C is 846.148, path: {A, G, L, C}
Shortest distance from Star A to Star D is 410.624, path: {A, D}
Shortest distance from Star A to Star E is 805.669, path: {A, G, E}
Shortest distance from Star A to Star F is 799.706, path: {A, P, F}
Shortest distance from Star A to Star G is 67.2161, path: {A, G}
Shortest distance from Star A to Star H is 965.458, path: {A, G, M, H}
Shortest distance from Star A to Star I is 783.996, path: {A, I}
Shortest distance from Star A to Star J is 611.429, path: {A, J}
Shortest distance from Star A to Star K is 787.543, path: {A, D, K}
Shortest distance from Star A to Star L is 374.307, path: {A, G, L}
Shortest distance from Star A to Star M is 640.709, path: {A, G, M}
Shortest distance from Star A to Star N is 833.853, path: {A, P, N}
Shortest distance from Star A to Star O is 589.784, path: {A, G, O}
Shortest distance from Star A to Star P is 412.279, path: {A, P}
Shortest distance from Star A to Star Q is 1239.08, path: {A, P, N, Q}
Shortest distance from Star A to Star R is 1432.58, path: {A, G, O, R}
Shortest distance from Star A to Star S is 1148.05, path: {A, P, F, B, S}
Shortest distance from Star A to Star T is 1269.44, path: {A, D, T}
```

# SHORTEST PATH



# KRUSKAL.H

## 1. Edge Generation

- Defines KruskalEdge to store starting star, ending star and distance of the edge

## 2. Merge Disjoint Set

- Declare ‘parent’ Vector that keeps track of parent of each element
- Declare ‘rank’ Vector helps in balancing tree
- Resize ‘parent’ and ‘rank’ vectors to hold ‘n’ element(stars)
- Initialized each element to be its parent and set rank of each element to 0

## 3. Determine Rank of Star

- Find roots of int u and int v-elements that need to be connected
- If roots are different, it means element u and element v are in different sets and need to be merged
- Compare rank of root- if rank of root u is smaller than rank of root v, it means root v is deeper than root u. Hence, root u is the parent to root v

## 4. read_kruskal_dataset

- Open file specified by ‘filename’
- Each line is store in ‘line’ String
- Declare start node, destination node and distance
- Use ‘sscanf’ to parse each line according to the specified format, it constructs KruskalEdge object and push to edge Vector and return edges

# KRUSKAL.H

## 5. Initialize MST

- mst Vector is initialized as an empty vector of KruskalEdge and will eventually contain edges
- sorted_edges- sort edges based on distance in ascending order
- get_index- convert star name to index to operate on integer indices

## 6. Process Sorted Edges

- Initialize UnionFund with parameter of vertex_count
- Iterate through sorted_edges and convert int u and int v into index
- Check if adding edge between u and v would form cycle in mst
- If u and v are from different sets, then the sets containing u and v will merge
- After merging, the current edge is push to minimum spanning tree

## 8. Save mst into file

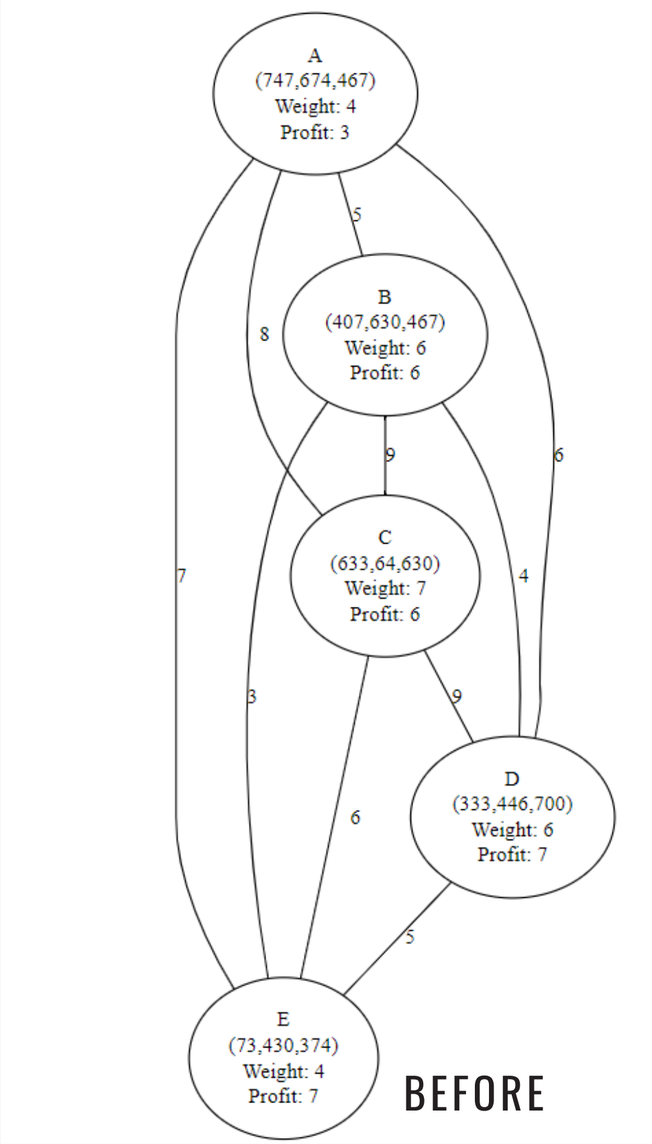
- Result is save into minimum_spanning_tree.txt

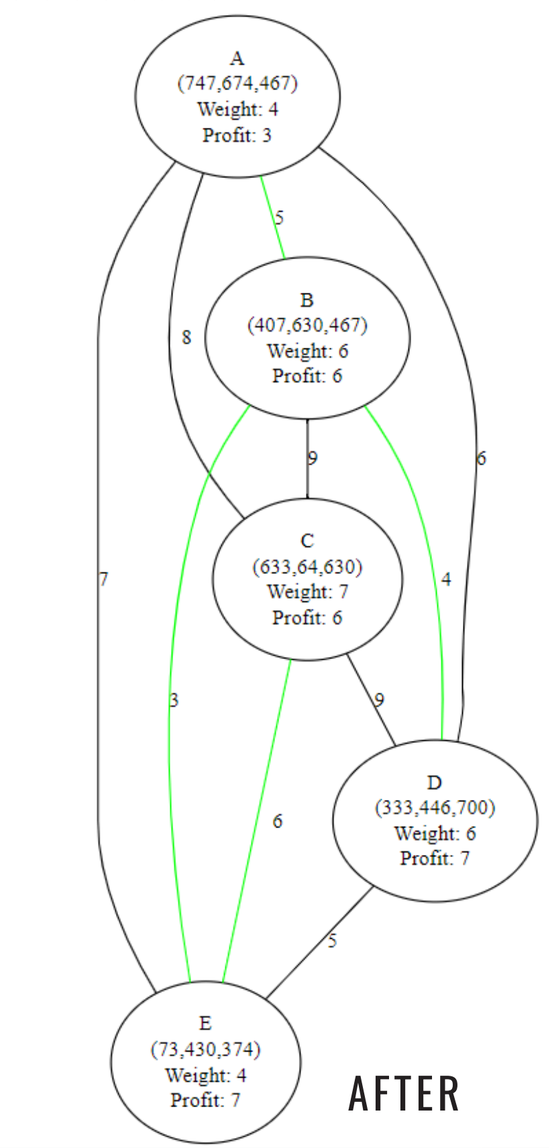
# MINIMUM SPANNING TREE

## 5 DATASET

Q3 Dijkstra and Kruskal > Kruskal 5 dataset > minimum_spanning_tree.txt

```
1  Connected Stars: B - E Distance: 3
2  Connected Stars: B - D Distance: 4
3  Connected Stars: A - B Distance: 5
4  Connected Stars: C - E Distance: 6
5
```



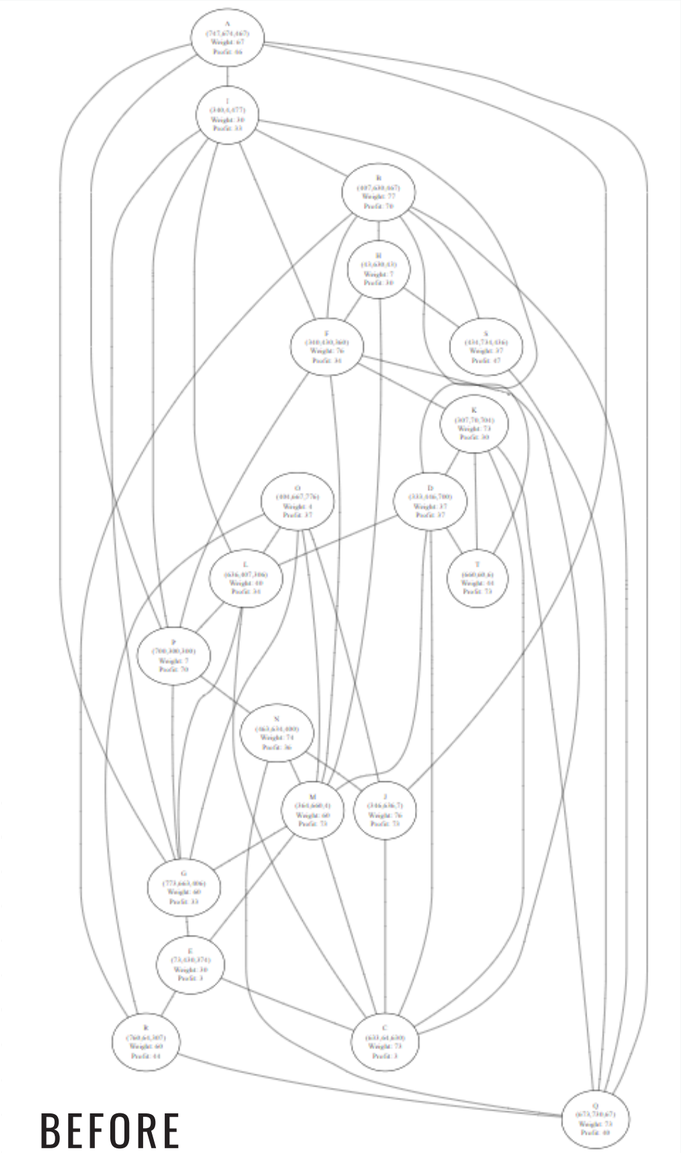



# MINIMUM SPANNING TREE

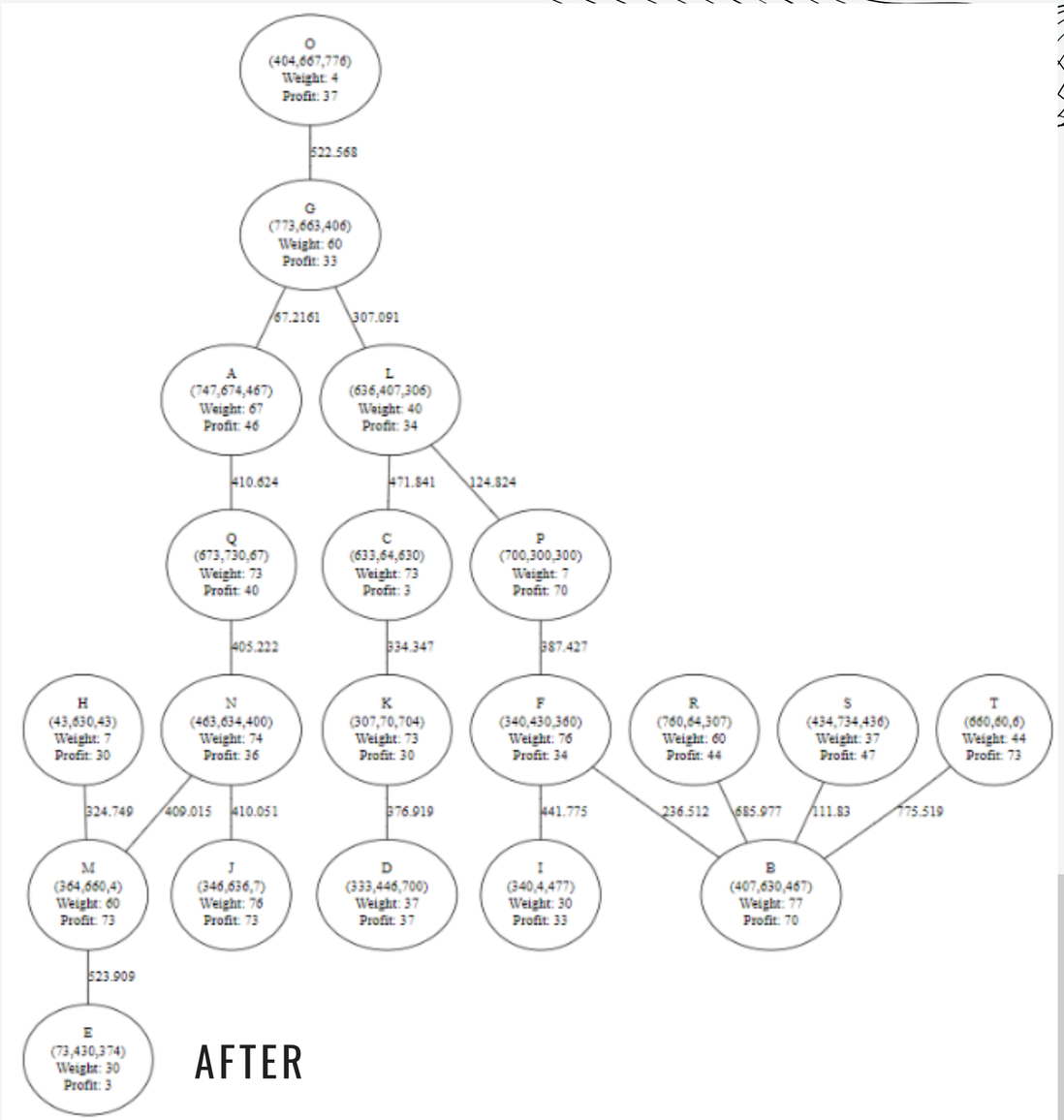
Q3 Dijkstra and Kruskal > Kruskal > minimum_spanning_tree.txt

```
1  Connected Stars: G - A Distance: 67.2161
2  Connected Stars: S - B Distance: 111.83
3  Connected Stars: L - P Distance: 124.824
4  Connected Stars: F - B Distance: 236.512
5  Connected Stars: G - L Distance: 307.091
6  Connected Stars: H - M Distance: 324.749
7  Connected Stars: C - K Distance: 334.347
8  Connected Stars: K - D Distance: 376.919
9  Connected Stars: P - F Distance: 387.427
10 Connected Stars: Q - N Distance: 405.222
11 Connected Stars: N - M Distance: 409.015
12 Connected Stars: N - J Distance: 410.051
13 Connected Stars: A - Q Distance: 410.624
14 Connected Stars: F - I Distance: 441.775
15 Connected Stars: L - C Distance: 471.841
16 Connected Stars: O - G Distance: 522.568
17 Connected Stars: M - E Distance: 523.909
18 Connected Stars: R - B Distance: 685.977
19 Connected Stars: T - B Distance: 775.519
20
```

# MINIMUM SPANNING TREE



BEFORE




AFTER

# TIME AND SPACE COMPEXITY

## Time Complexity Dijkstra

- Reading Dataset- O( S+R), where S is number of Stars and R is number of Routes
- Dijkstra function- O((S + R) log S) using a priority queue

Overall Complexity

**O((S+R) log S)**

## Time Complexity Kruskal

- Sorting Edges- O(E log E) where E is number of edges
- Union Find Operations- O(α(V)), where V is the number of vertices

Overall Complexity

**O( E log E + Eα(V))**

## Space Complexity Kruskal

- Data Structure- O( S+R), where S is number of Stars and R is number of routes
- Dijkstra function- O(S) for storing distances and paths
- Priority Queue- O(S) , where S is number of Stars and distances are stored for each star

Overall Complexity

**O(S+R)**

## Space Complexity Kruskal

- Union Find Operations- O(V) for storing parent and rank arrays
- Store Vector of Edges- O(V + E)

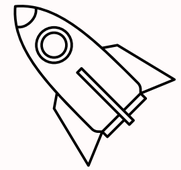
Overall Complexity

**O( V + E)**

# Q4 0/1 KNAPSACK

# INTRO AND PROBLEM STATEMENT

- **Objective**: Maximize the total profit without exceeding the weight capacity.
- **Constraints**: Each item can be selected or not (0 or 1).



- Participants travel in a spaceship with a maximum capacity of 800 kg.
- Identify the optimal set of stars to plunder to maximize profit without returning to the beginning star location.

# 0/1 KNAPSACK ALGORITHM OVERVIEW

01

## DP APPROACH

1. **Define Subproblems**:
   - Use a DP table dp[i][w] where i is the number of items considered and w is the weight capacity.
   - dp[i][w] represents the maximum profit for the first i items with a weight limit of w.
2. **Transition Relation**:
   - If the star's weight is less than or equal to the current capacity, consider including it.
   - dp[i][w] = max(dp[i-1][w], dp[i-1][w - stars[i-1].weight] + stars[i-1].profit)
3. **Initialization**: The dp table is initialized with all zeros, representing zero profit for zero capacity.

02

## DP TABLE

- **Rows**: Items (stars)
- **Columns**: Weight capacity from 0 to 800 kg
- Each cell dp[i][w] stores the maximum profit for i items and capacity w.

03

## DATASET2_1.TXT CALC

- **Stars Data**: (Weight and Profit for 20 stars respectively)
- **Weight**: [43, 63, 73, 63, 43, 4, 63, 67, 7, 67, 64, 43, 7, 34, 0, 36, 4, 33, 66, 0]
- **Profit**: [43, 60, 67, 76, 67, 46, 63, 43, 6, 76, 33, 73, 6, 40, 64, 30, 74, 46, 47, 76]
- **Result**:
- **Total Weight & Profit**: 780 & 1036
- **Stars Selected**: All 20 stars within the 800 kg limit.

# BRIEF CODE WALKTHROUGH

**Reading Input**:

- Function to read stars data from a file.

STEP 1

**DP Table Initialization**:

- Initialize a 2D vector with zeros.

STEP 2

**Filling the DP Table**:

- Nested loops to fill the DP table based on the weight and profit conditions.

STEP 3

**Traceback to Select Stars**:

- Determine the selected stars and compute total profit.

STEP 4

**Saving Results**:

- Save the resulting stars, DP table and other necessary information to a file.

STEP 5

# RESULT & DISCUSSION

1. The time complexity of the knapsack problem is O(n*W), where n is the number of items (stars) and W is the capacity of the knapsack.
   - This is because the dynamic programming approach fills a table with dimensions (n+1)*(W+1), iterating over each item and each capacity increment.
2. The space complexity is also O(n*W), as the DP table requires storage proportional to the product of the number of items and the knapsack capacity.
3. This approach ensures that each subproblem is solved only once and its result is reused, optimizing both time and space but still being dependent on the problem's input size.

## TIME AND SPACE COMPLEXITY:

# O(N*W)

# THE END