TUTORIAL 8-9

MINI-PROJECT: THE JOKER

BY:

VAYSHNAVI SIVARAJAH & SOUMAYA SABRY

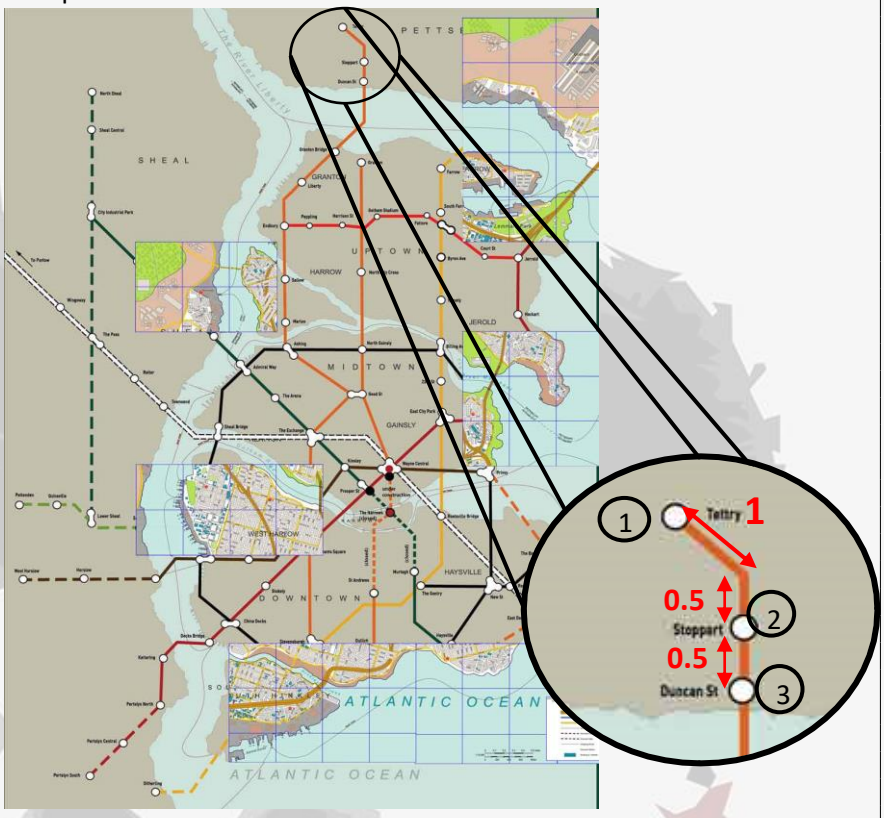
Summaries

Part 1: Connecting the people

Gotham City, August 1982, the city is subject to fire and sword. The whole district is under the Jokers and the police browse across the battlefield like scared dogs. Your new organization need a communication network to build a lasting resistance to the corrupted government. You decide to place some radio substation in each railway stations, each substation have a wired connection to at least another substation to create a connected network (in Uptown, Midtown and Downtown, see the map above).

Code all algorithm and justify your answer in the report.

We represented the map below in a non-oriented weighted graph. We consider that the weight is the distance between two metro stations. We calculated the distance between stations with a rule and multiplied it by the scale (we consider that 1 cm = 1km).
We represented a node (a metro station) by a number.
(For more details, Cf. map.pdf)



Here is the head of our database in which we store the edges of the graph:

node1	node2	distance
1	2	1500
2	3	500
3	4	2700
4	5	4500
5	6	1700
6	7	800
7	8	1100
8	9	1100
9	10	1600

1. Create and display a graph representing the Gotham City's railways and subways below.

   To represent the Gotham City's railways and subways, we created a class graph which is composed of 3 elements:

   - **Edges**: which is a list in which we indicate with which nodes are connected every node
     Example: the node 1 is connected to the nodes 2 and 4.
   - **Distances**: which is a dictionary in which we indicate the distance between 2 nodes.
     Example: the distance between the node 1 and the node 2 is 1500.
   - **Nodes**: which is a set with all the nodes.
     Example: in our graph, there are the nodes 1,2,3,4, …, 91.

Here is a representation of our graph in a matrix:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91
1	0	1500	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1500	0	500	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	500	0	2700	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	2700	0	4500	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	4500	0	1700	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
..	...	...	...	...	...	...	..	..	..	..	..	..	..	..	..	...	..	..	..	...	..	..	..	..	..	...	...	...	...	...	...
87	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	2500	0	0	0	0	0	800	0	1600	0	5000	0
88	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	1600	0	0	0	0
89	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	0	500	0
90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	5000	0	500	0	2000
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0	0	2000	0

We also represented the graph with the class networkx. But the graph is not very comprehensible:

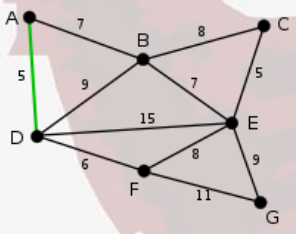
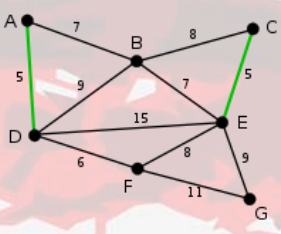
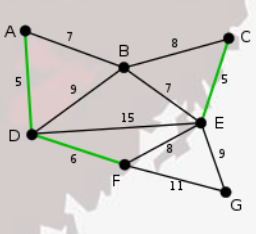
(source:https://networkx.github.io/documentation/stable/auto_examples/drawing/plot_weighted_graph.html?fbclid=IwAR0yWW5mcpxJatnw73LDr1UE6maZ3VN1wB6r1HjcQEjnFlGblghZiggeWDs#sphx-glr-auto-examples-drawing-plot-weighted-graph-py).

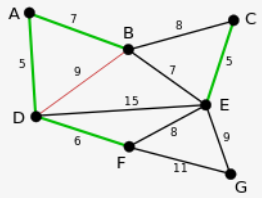
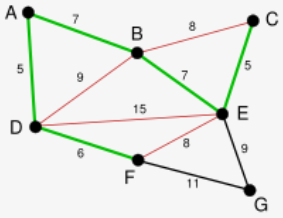
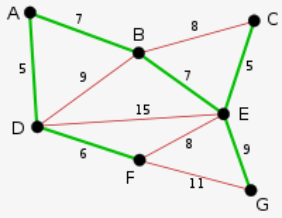
2. Which kind of algorithm create a connected graph while minimizing the total amount of distance? Show the algorithm? What is its complexity?

A. Explanation of the algorithm

A spanning tree T of an undirected graph G is a subgraph that is a tree which includes all the vertices of G. A minimum spanning tree (MST) connects all the vertices together with minimal total weighting for its edges.

The problem we must resolve is a Minimal Spanning Tree. For that, we can use the Kruskal method or the Prim method.
Here, we are going to use the Kruskal's algorithm:

		
1- AD and CE are the shortest edges, with length 5, and AD has been arbitrarily chosen, so it is highlighted.	2- CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.	3- The next edge, DF with length 6, is highlighted using much the same method

		
4- The next-shortest edges are AB and BE, both with length 7. AB is chosen arbitrarily and is highlighted. The edge BD has been highlighted in red, because there already exists a path (in green) between B and D, so it would form a cycle (ABD) if it were chosen.	5- The process continues to highlight the next-smallest edge, BE with length 7. Many more edges are highlighted in red at this stage: BC because it would form the loop BCE, DE because it would form the loop DEBA, and FE because it would form FEBAD.	6- Finally, the process finishes with the edge EG of length 9, and the minimum spanning tree is found.

B. Explanation of the code

To code this Kruskal method, we need to understand the logic. We are going to explain it with an example:

We initialize the tables root[] and rank[] : the values of root[] will be the values of the nodes and the values of rank[] will be 0.
Root[] is a table which indicates which is the root of the node and rank[] is a table which indicates which is the height of the node.

node	1	2	3	4	5	6	7	8	9	10
root	1	2	3	4	5	6	7	8	9	10
rank	0	0	0	0	0	0	0	0	0	0

The first edge is {7,4} which weight is 1. We place it in the result graph and we connect the nodes 4 and 7.
We assume that 4 is the root of 7 so we put the value 4 in the root(7) and now the rank of 4 is 1.

node	1	2	3	4	5	6	7	8	9	10
root	1	2	3	4	5	6	4	8	9	10
rank	0	0	0	1	0	0	0	0	0	0

The second edge is {8,10} which weight is 1. 8 and 4 are both roots. So we don't create cycles by adding this edge.
We place it in the result graph and we connect the nodes 8 and 10: we assume that 8 is the root of 10 so we put the value 8 in the root(10) and now the rank of 8 is 1.

node	1	2	3	4	5	6	7	8	9	10
root	1	2	3	4	5	6	4	8	9	8
rank	0	0	0	1	0	0	0	1	0	0

The next edge is {9,10} which weight is 2. 9 is a root and 10 has for root 8 (which is different from 9). So, we don't create cycle by adding this edge.
We place it in the result graph and we connect the nodes 9 and 10: the rank of 8 is 1 and the rank of 9 is 0.
To maintain the rank of 8 at 1, we must put the value 8 in the root(9) : we give to 9 the same root as 10 which he got connected to.

node	1	2	3	4	5	6	7	8	9	10
root	1	2	3	4	5	6	4	8	8	8
rank	0	0	0	1	0	0	0	1	0	0

And so on…

Source : *http://www.unit.eu/cours/EnsROtice/module_avance_thq_voo6/co/algoKruskal.html*

The complexity of the Krushkal's algorithm is **O(nlog(n))**, where n is the total number of edges:

- First, we sort the edges by croissant weights order. This sort could be do with a complexity of O(nlog(n)).
- Then, we need to verify if the adding of an edge product a cycle or not. For that, we need to maintain the values of the root and of the rank of each nodes by using the Union-Find structure. We can verify if the edge could be added or not with a complexity of O(log(n)).

C. Python code

Here is the python code of the Kruskal's algorithm:

```
def Kruskal(self):
    result = {}

        i=0
        e=0
        d = sorted(self.distances.items(),key=lambda t:t[1])
        root = []
        height = []

        number = len(self.nodes)

        for node in range(1,number+1):
            root.append(node)
            height.append(0)

        while e < len(self.nodes) - 1:
            l = list(list(d[i])[0])
            source = l[0]
            destination = l[1]
            weight = list(d[i])[1]

            i=i+1
            rs = self.find(root,source)
            rd = self.find(root,destination)

            if rs!=rd:
                e=e+1
                result[(source,destination)]=weight
                self.connect(root,height,rs,rd)

        for edge_n in result:
            print(str(edge_n[0]) + " ------- " + str(edge_n[1]) + " =
    " + str(result[edge_n]))
        return result
```

3. Show and display a solution of the connected network.

Finally, with the method of Kruskal, we got the following result (representation of the first edges):
We also represented the edges in a graph with the class network.

(source: https://networkx.github.io/documentation/stable/auto_examples/drawing/plot_weighted_graph.html?fbclid=IwAR0yWW5mcpxJatnw73LDr1UE6maZ3VN1wB6r1HjcQEjnFlGblghZiggeWDs#sphx-glr-auto-examples-drawing-plot-weighted-graph-py).

```
57 ------- 58 = 200
2 ------- 3 = 500
10 ------- 11 = 500
89 ------- 90 = 500
44 ------- 45 = 700
70 ------- 71 = 700
6 ------- 7 = 800
32 ------- 33 = 800
45 ------- 55 = 800
86 ------- 87 = 800
48 ------- 49 = 900
45 ------- 46 = 1000
24 ------- 53 = 1000
53 ------- 54 = 1000
54 ------- 23 = 1000
72 ------- 71 = 1000
```

Part 2: Spread the revolution

Since Gotham is in your hand, the revolution needs to be disseminated into other cities. You have many volunteers, but you want to optimize the process. Following the matrix of distance below, you want to compute the shortest path from to Gotham City (first line/column) to the others. At each path from Gotham to an endpoint, a volunteer is sent.

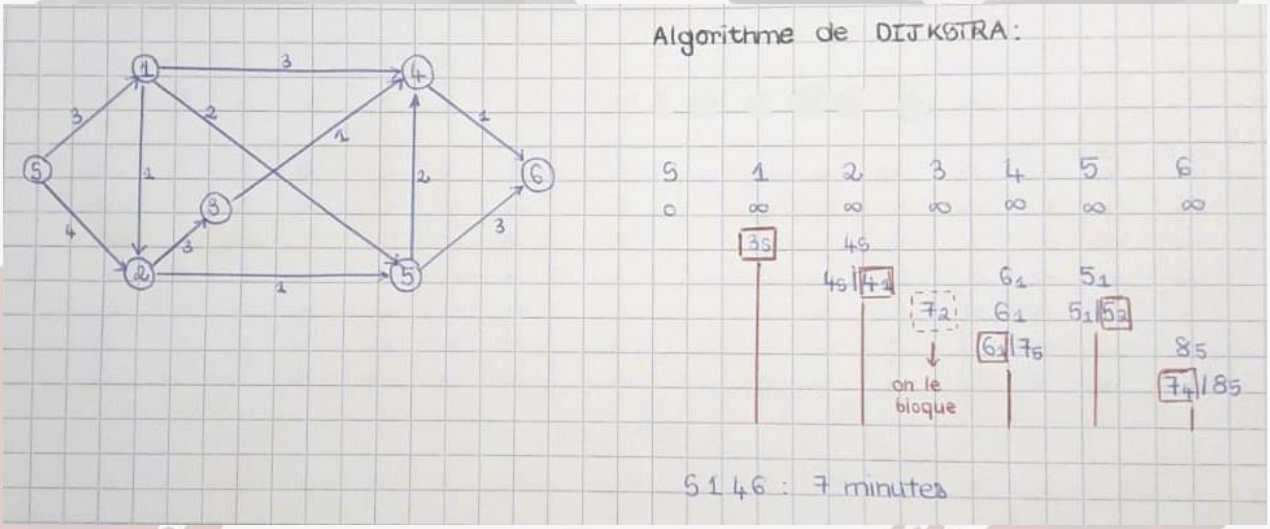
	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}
g_1	0.0	8.1	9.2	7.7	9.3	2.3	5.1	10.2	6.1	7.0
g_2	8.1	0.0	12.0	0.9	12.0	9.5	10.1	12.8	2.0	1.0
g_3	9.2	12.0	0.0	11.2	0.7	11.1	8.1	1.1	10.5	11.5
g_4	7.7	0.9	11.2	0.0	11.2	9.2	9.5	12.0	1.6	1.1
g_5	9.3	12.0	0.7	11.2	0.0	11.2	8.5	1.0	10.6	11.6
g_6	2.3	9.5	11.1	9.2	11.2	0.0	5.6	12.1	7.7	8.5
g_7	5.1	10.1	8.1	9.5	8.5	5.6	0.0	9.1	8.3	9.3
g_8	10.2	12.8	1.1	12.0	1.0	12.1	9.1	0.0	11.4	12.4
g_9	6.1	2.0	10.5	1.6	10.6	7.7	8.3	11.4	0.0	1.1
g_{10}	7.0	1.0	11.5	1.1	11.6	8.5	9.3	12.4	1.1	0.0

Code all algorithm and justify your answer in the report.

1. Which kind of algorithm create a connected graph while minimizing the distance to a unique city? What is its complexity?

We need an algorithm which can resolve the problem of Shortest Path.
Here we have only positive weights and the graph is non-oriented, so the best algorithm will be the Dijkstra's algorithm.

Let's explain the Dijkstra's algorithm with an example:



We start from the node **S** and we want to go to the node 6. We are searching for the **shortest path**. The coefficients on the edges of the graph represent the travel time between two nodes.
First, from the point S to the point S, the travel time is 0 and for all the other points it's infinite.
We can go from S to 1 with a travel time of 3 minutes (we note in the column 1: 3_S). We can also go from S to 2 with a travel time of 4 minutes (we note in the column 2: 4_S).
We choose the shortest which is going to 1 (we circle the value of the column 1). Now we are on the node 1. We can go to 2 with a travel time of 1 minutes, so the entire travel time from S will be 4 minutes (we note in the column 2: $4_S|4_1$ which are the two possibilities to go to 2). We can also go to 4 with a travel time of 3 minutes, so the entire travel time will be 6 minutes (we note in the column 4: 6_1). We can also go to 5 with a travel time of 2 minutes, so the entire travel time will be 5 minutes (we note in the column 5: 5_1).

We choose the shortest path which is going to 2: we can go by either the S point or the 1 point (we circle one of the values of the column 2).
And so on, until you arrive to the node 6.

By using this logic, in the worst-case scenario, we must calculate the distance with a loop for the first time, doing n iteration (n is the number of nodes), then we block one of them (the minimal). We continue by looping (n-1) this time, then block one, then loop (n-2), then block one, and so on…
To finish this algorithm, we will do n iteration where we do the looping for distance, it will be:
n+ (n-1) + (n-2) ……+1 = $\sum_{n}^{1} i = \frac{n(n-1)}{2}$. In conclusion, the complexity of this algorithm is **O(n(n-1)/2) = O(n^2)**.

We applied the Dijkstra's algorithm on our matrix of distance (by hand) and here is the result:

G1	G2	G3	G4	G5	G6	G7	G8	G9	G10
0	8.1 g1	9.2 g1	7.7 g1	9.3 g1	**2.3 g1**	5.1 g1	10.2 g1	6.1 g1	7 g1
|	**8.1** g1/11.8 g6	**9.2** g1/13.4 g6	**7.7** g1/11.5 g6	**9.3** g1/13.5 g6	|	**5.1** g1/7.9 g6	**10.2** g1/14.4 g6	**6.1** g1/10 g6	**7** g1/10.8 g6
|	**8.1** g1/15.2 g7	**9.2** g1/13.2 g7	**7.7** g1/14.6 g7	**9.3** g1/13.6 g7	|	|	**10.2** g1/14.2 g7	**6.1** g1/13.4 g7	**7** g1/14.4 g7
|	8.1 g1/**8.1 g9**	**9.2** g1/16.6 g9	7.7 g1/**7.7 g9**	**9.3** g1/16.7 g9	|	|	**10.2** g1/17.5 g9	|	**7** g1/7.2 g9
|	8.1 g9/**8 g10**	**9.2** g1/18.5 g10	**7.7** g9/8.1 g10	**9.3** g1/18.6 g10	|	|	**10.2** g1/19.4 g10	|	|
|	**8** g10/8.6 g4	**9.2** g1/18.9 g4	|	**9.3** g1/18.9 g4	|	|	**10.2** g1/19.7 g4	|	|
|	|	**9.2** g1/20 g2	|	**9.3** g1/20 g2	|	|	**10.2** g1/20.8 g2	|	|
|	|	|	|	**9.3** g1/9.9 g3	|	|	**10.2** g1/10.3 g3	|	|
|	|	|	|	|	|	|	**10.2** g1/10.3 g5	|	|
Node depart	G10	G1	G9	G1	G1	G1	G1	G1	G1

2. Show and display a solution of the proposed algorithm.
   Here is the python code of the Dijkstra's algorithm

```
def dijsktra(self, initial):
        current_n = {initial: 0}
        path = {}
        nodes = set(self.nodes)

        #the n iteration of the algo (n there is nodes.lenght)
        while nodes:
            node_min = None
            #we find the minimal node(the loop of distance)
            for node in nodes:
                if node in current_n:
                    if node_min is None:
                        node_min = node
                    elif current_n[node] < current_n[node_min]:
                        node_min = node

            if node_min is None:
                break

            # we remove the mode with minimal distance (aka block one)
```

```
            nodes.remove(node_min)

            current_weight = current_n[node_min]

          #we calculate the distance and attribute the best value to current
          for edge in self.edges[node_min]:
                weight = current_weight
                         + self.distances[frozenset({node _min, edge})]
              if edge not in current_n or weight < current_n[edge]:
                    current_n[edge] = weight
                    path[edge] = node_min

        return current_n, path
```

3. Show and display a solution of the problem.

The problem is to find how many people we must send. This number is the number of leaves of the shortest path tree that we found with Dijkstra's algorithm.

Here is the solution of the problem: (Each node with the shortest distance to go there)

```
_____________________ Applying Dijsktra _____________________
There is the Tree of the path :::
g1 (0)
|---g3 (9.2)
|---g5 (9.3)
|---g6 (2.3)
|---g7 (5.1)
|---g8 (10.2)
|---g9 (6.1)
|---|---g4 (7.7)
|---g10 (7.0)
|---|---g2 (8.0)
Based on our Calculating, We will send 7 Volunteers
```

Part 3: Organize the Jokers

The organization is growing so fast that some corrupted contact infested your new world. Most of the supporters are registered in a database which is informed by confirmed Jokers. While meeting another supporter, one may identify the other one in the database.

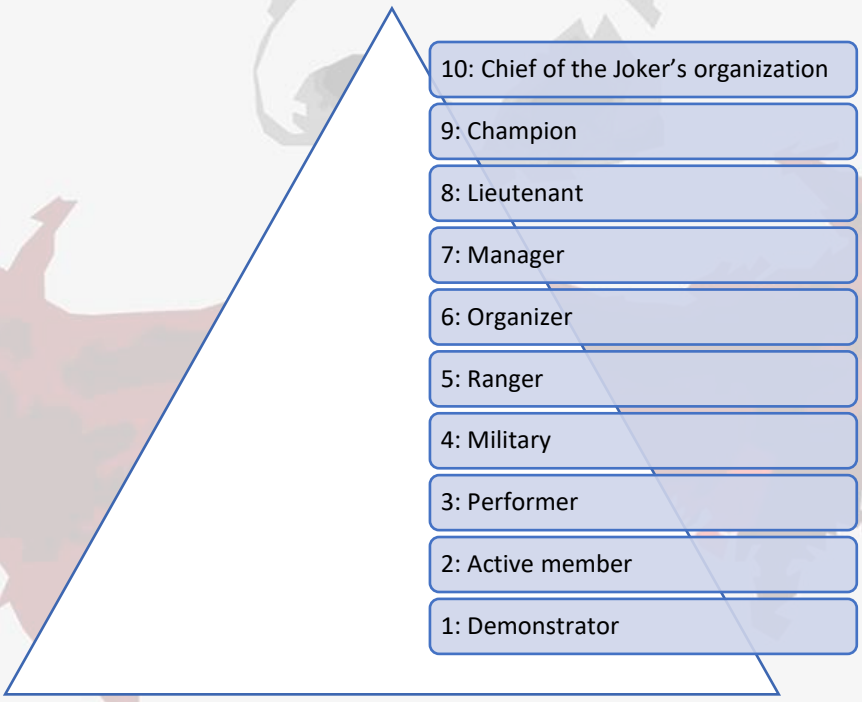
Code all algorithm and justify your answer in the report.

In our top-secret folders, we have many data about the Jokers: it's the Joker's database. Only the main Joker have the authorization to modify this database or to add new data in this database. Other Jokers can only access to the data in read-only mode: they can, for example, search information about another Joker member.

This database contains the following data:

ID	The ID number is a unique number which is used to identify each Joker member.
Nickname	It's not the real name of the Joker member because in the world of Jokers, everyone must be anonymous.
Location	Corresponds to one of the 5 main areas of Gotham City it indicates in which area the Joker member takes actions.
Trust Level *	Inspired by the S.H.I.E.L.D organization of Avengers, Joker's accreditations depend on a code ranging from 1 to 10, giving access to equipment, data and secret information.

* Trust Level: here is a detailed description of the different existing levels in the Joker's world:



Here is the head of our database:

ID	Nickname	Location	Trust Level
123	Jake	MidTown	8
20	Johnny	Uptown	7
34	Cats Eyes	Uptown	3

1. Suggest and show a greedy method to find an ID in a (linear) database. What is its complexity?

A greedy method to find an ID in the database is to browse the database until we found the correct ID. It's the iterative method which is the least efficient method.
It is an algorithm that look over all the items in the data, so it does a loop of n iteration. Its complexity is **O(n)** where n is the number of Jokers members who are in the Database.

Here is the python code of the iterative method:

```
def Iteratif(ResearchedID):
      is_Found=False
      for line in Database:
            if line.ID == ResearchedID:
                  print(line)
                  is_Found=True
                  break
      if is_Found == False:
            print("/!\ /!\ He isn't a Joker /!\ /!\ ")
```

This method is simple to code but can take a lot of time if the database is very big.

2. Suggest and show a divide and conquer method to find an ID in a (linear) database. What is its complexity?

To find an ID in a (linear) database, we can use a divide and conquer method.

The divide & conquer algorithm works by recursively breaking down a problem into a sub-problem of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

In worst case scenario, the researched id will be in the last sub-list, so the algorithm will run through all the items in the data base (aka n items). Its complexity is **O(n)** where n is the number of Jokers members who are in the Database.

Here is the python code of the Divide and Conquer method:

```
def DivideAndConquer(ResearchedID,left,right):
      ind0 = CI.Individu(0,"None","None",0)
      if left==right:
            return Database[left]
      else:
            if (left+right)%2==0:
                  moitie = (left+right)/2
            else:
                  moitie = (left+right-1)/2
            temp1 = DivideAndConquer(ResearchedID, left,int(moitie))
```

```
            temp2 = DivideAndConquer(ResearchedID, int(moitie)+1,right)
            if temp1.ID == ResearchedID:
                  return temp1
            else:
                  if temp2.ID == ResearchedID:
                        return temp2
                  else:
                        return ind0
```

3. Suggest and show a method that sort the (linear) database and find an ID. What is its complexity?

First, we will apply a sorting algorithm on the list of Jokers: The merge sort algorithm *(algorithme de tri-fusion)*. Since, the list will be sorted, the ID will be more easily found (we must verify if the middle value is superior or inferior to the researched value and then research in the correct part of the database): It's the Binary Search *(recherche dichotomique)*.

Merge sort is a divide and conquer algorithm. That have of 3 main steps: The divide step computes the midpoint of each of sub list, the conquer step recursively sorts two subarrays and the merge step merges n elements which takes O(n) time.
For each level from top to bottom Level n calls merge method on n sub-arrays. The complexity here is **O(nlog(n))**.

Binary Search reduces the effort to search for the element by half each time so, the complexity of the is **O(log(n))**.

Finally, the complexity of the totality is max (nlog(n), log(n)) ==> **O(nlog(n)),** where n is the number of Jokers members who are in the Database.

Here is the python code of the merge sort algorithm:

```
def MergeSort(ArrayL):
    if len(ArrayL) == 1:
        return [ArrayL[0]]
    else :
        L = MergeSort([ArrayL[i]for i in range
(0,int(len(ArrayL)/2))])
        G = MergeSort([ArrayL[i]for i in range (int(len(ArrayL)/2),
len(ArrayL))])
        res = []
        indexL =0
        indexG =0
        AcomparerUN = L[indexL]
        AcomparerDEUX = G[indexG]
        estFini = 0
        while estFini == 0 :
            estFini = 1
            if AcomparerUN.ID < AcomparerDEUX.ID :
                res.append(AcomparerUN)
                indexL = indexL +1
                if indexL < len(L):
                    estFini = 0
                    AcomparerUN = L[indexL]
```

```
            else :
                res.append(AcomparerDEUX)
                indexG = indexG +1
                if indexG < len(G):
                    estFini = 0
                    AcomparerDEUX =  G[indexG]


        if len(res) < len(L)+len(G) :
            if indexL <len(L):
                for i in range(indexL , len(L)):
                    res.append(L[i])
            else :
                for i in range(indexG , len(G)):
                    res.append(G[i])

        return res
```

Here is the python code of the Binary Search:

```
def BinarySearch(ResearchedID,JokersList):
    ind0 = CI.Individu(0,"None","None",0)
    l=len(JokersList)
    m = int(len(JokersList)/2)
    if l == 1 :
        if JokersList[m].ID == ResearchedID:
            return JokersList[m]
        else:
            return ind0
    elif JokersList[m].ID == ResearchedID:
        return JokersList[m]
    elif JokersList[m].ID > ResearchedID:
        return BinarySearch(ResearchedID, JokersList[1:m])
    else:
        return BinarySearch(ResearchedID,JokersList[m:l])
```

4. Show and display the three methods with an example of database of your choice (at least 50 data). (10%)

We suppose that a Joker named Jake needs to meet another Joker named Johnny to give him some stuff. When meeting, Johnny gives his ID to Jake. Jake must search this ID in the database to verify that Johnny is not a corrupted contact.

To do that, Jake can use three methods of search. Since it's his first time to meet another Joker, he's going to use the three methods and note which is the faster so then he'll use only this method.

Johnny's ID is 150. Here are the results of the three methods:

```
150 Bully South Hinkley 1
The iterative method takes 0.000999 seconds.
150 Bully South Hinkley 1
The divide and conquer method takes 0.006999 seconds.
150 Bully South Hinkley 1
The Binary Search method takes 0.001022 seconds.
```

Here is a recapitulative of the three methods we can use to find an ID in the list of the Jokers:

N°	Name of the method	Complexity	Time (n = 150) *	Time (n = 1000) *
1	Iterative linear method	O(n)	0.000999	0.001003
2	Divide and Conquer recursive method	O(n)	0.006999	0.007002
3	Merge sort and Binary Search method	O(nlog(n))	0.001022	0.001021

* The time represents the time in seconds that takes each method to find an ID in a list of n Jokers.

Since, the Joker's organization is growing faster and faster, we also tested the algorithms of search with a dataset of 1 000 elements:

```
1000 Pancho Uptown 9
The iterative method takes 0.001003 seconds.
1000 Pancho Uptown 9
The divide and conquer method takes 0.007002 seconds.
1000 Pancho Uptown 9
The Binary Search method takes 0.001021 seconds.
```

We can say that the Divide and Conquer method takes more time than the Iterative method because we divide the problem in sub-problems but in our case it's not necessary: it's faster to browse simply the list.

Finally, the Merge sort and Binary Search method will be the faster (when there is a very big number of elements): it's logic since it has the **smallest** complexity.

Part 4: The Jokefather

Your organization is worldwide, you are the Jokefather, the cradle of crimes and filths. The Empire is so big and manage so much people and malicious business, dishonest acts and unfair actions that ever the best IT crews have a hard time to handle so much information. You need to find a better management of those data. Knowing the great engineering school named ESILV, you ask them to find a solution.

Code all algorithm and justify your answer in the report.

The database is the same that in the Part 3: The Joker's database.

1. Suggest and show a method to manage a database in a binary tree; present a method to find any value in the tree (show both complexity). Show the methods with your own example (at least 50 data).

A. Binary Search Tree

We are going to represent the database in a Binary Search Tree.

A Binary Search Tree is a binary tree with the following properties.

- There is a total order relation on the members in the tree.
- At every node:
  - Every member of the *left subtree* is *less* than or equal to the node value.
  - Every member of the *right subtree* is *greater* than or equal to the node value.

For browsing a binary tree, in a worst-case scenario, we will browse a tree with only one long branch with all the nodes on it, more likely a linear data base. So, the algorithm will run through every node, it will be a complexity of **O(n)**, where **n** is the number of nodes.

B. Insert in the BST

First, we must insert the data in the Binary Search Tree.
For that, we need to begin at the root node and apply the following steps recursively:

- Compare the value to the node data.
- If it is less (or equal), continue with the left subtree.
- If it is greater, continue with the right subtree.
- When no child, attach the node as a subtree.

The complexity of the algorithm for inserting a Joker in the Binary Search Tree is **O(n)**, where **n** is the number of Jokers members who are in the Database.

Here is the python code for inserting a Joker in the Binary Search Tree:

```
def _add(self, val, node):
    if(val < node.v):
        if(node.l != None):
            self._add(val, node.l)
        else:
            node.l = Node(val)
    else:
        if(node.r != None):
            self._add(val, node.r)
        else:
            node.r = Node(val)
```

```
def add(self,val):
      if(self.root == None):
            self.root = Node(val)
      if(self.find(val) == False):
            self._add(val,self.root)
```

C. Search in the BST

Then, we must search the ID of the Joker in the Binary Search Tree.
For that, we need to begin at the root node and apply the following steps recursively:

- Compare the value to the node data.
- If it is equal, find!
- If it is less, search the left subtree.
- If it is greater, search the right subtree.
- If the subtree is empty, the value is not in the tree.

The complexity of the algorithm for searching a Joker in the Binary Search Tree is **O(n)**, where **n** is the number of Jokers members who are in the Database.

Here is the python code for searching a Joker in the Binary Search Tree:

```
def _find(self, val, node):
      if(val == node.v):
            return True  ##node
      elif(val < node.v and node.l != None):
            return self._find(val, node.l)
      elif(val > node.v and node.r != None):
            return self._find(val, node.r)
      else :
            return False #None

def find(self, val):
      if(self.root == None):
            return None
      else:
            return self._find(val,self.root)
```

2. Suggest and show a method to improve the database management thanks to the works of the soviets Adelson-Velsky and Landis with the previous database.

A. Explanation of the AVL method

We can improve the database management by using the AVL method. Using AVL method changes our tree to be more balanced.

A binary search tree is an AVL tree if:

- It is an *empty tree* or if its sub-trees are AVL trees and the difference in *height* between its *left* and *right* sub-tree is between -1 and +1.

Balancing factor = | Height(left sub-tree) – Height(right sub-tree) | ≤ 1

First, we must repair the Binary Tree with the AVL method.
For that, we are going to follow the steps:

- After inserting a node, it is necessary to check each of the node's ancestors for consistency with the AVL rules.
- For each node checked, if the balance factor remains 1, 0, or -1 then no rotations are necessary. Otherwise, it's unbalanced.
- After each insertion, at most two tree rotations are needed to restore the entire tree.

We did an inserting method to check after each element to verify that it's an AVL tree:

```
def addByList_AVL(self,root,List):
    self.add(root)
    self.CalculBF()
    for val in List:
        self.add(val)
        self.CalculBF()
        while self.LoopCheckAVL(self.root,True)==False :
            self.Repair()
```

B. Code of the AVL method

```
def _Repair(self,node):
     if self.is_Leaf(node)== False:
          if node.r:
               self._Repair(node.r)
               if self.CheckIfAVL(node.r) == False :
                    node.r = self.Rotation(node.r)
                    #recount of BF
                    self.CalculBF()
          if node.l:
               self._Repair(node.l)
               if self.CheckIfAVL(node.l) == False :
                    node.l = self.Rotation(node.l)
                    #recount of BF
                    self.CalculBF()
def Repair(self):
     if self.root:
          self._Repair(self.root)
          if self.CheckIfAVL(self.root)==False :
               self.root = self.Rotation(self.root)
               #recount of BF
               self.CalculBF()
```

Then we can search in this tree, with the searching method we saw in the 1st question.
In the case of a balanced tree, the complexity of the searching will be, **O(h)** where h is the maximum height of the AVL tree($n=2^{h+1}-1$). Where the worst-case possible height of AVL tree with n nodes is 1.44*log(n), so the complexity **O(h)=O(1.44*log(n))=O(log(n)).**

Finally, we can see on the following example that by inserting in a Binary Search Tree with the normal method create an unbalanced tree inserting by using the AVL method create a balanced tree:

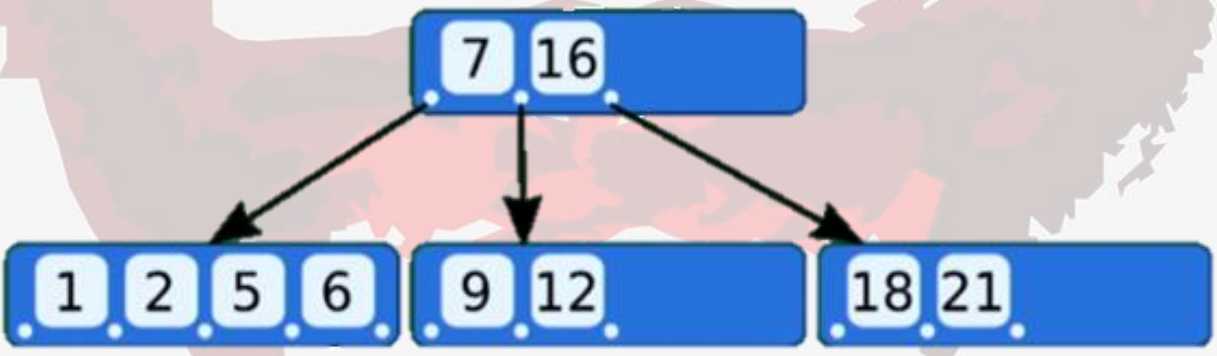
```
------------------Tree------------------
13
|---7
|---64
|---|---15
|---|---|---X
|---|---|---24
|---|---|---|---X
|---|---|---|---29
|---|---|---|---|---26
|---|---|---|---|---46
|---|---|---|---|---|---33
|---|---|---|---|---|---|---X
|---|---|---|---|---|---|---39
|---|---|---|---|---|---58
|---|---|---|---|---|---|---53
|---|---|---|---|---|---|---|---48
|---|---|---|---|---|---|---|---|---X
|---|---|---|---|---|---|---|---|---51
|---|---|---|---|---|---|---|---55
|---|---|---|---|---|---|---X
|---|---74
|---|---|---72
|---|---|---|---69
|---|---|---|---X
|---|---|---122
|---|---|---|---81
|---|---|---|---|---X
|---|---|---|---|---114
|---|---|---|---136
|---|---|---|---|---133
|---|---|---|---|---|---131
|---|---|---|---|---|---X
|---|---|---|---|---138
```

```
------------------Inserting with AVL--------------
55
|---29
|---|---24
|---|---|---13
|---|---|---|---7
|---|---|---|---15
|---|---|---26
|---|---39
|---|---|---33
|---|---|---|---15
|---|---|---|---46
|---|---|---51
|---|---|---|---46
|---|---|---|---|---X
|---|---|---|---|---48
|---|---|---|---53
|---122
|---|---58
|---|---|---53
|---|---|---|---48
|---|---|---|---X
|---|---|---74
|---|---|---|---69
|---|---|---|---|---64
|---|---|---|---|---72
|---|---|---|---81
|---|---|---|---|---X
|---|---|---|---|---114
|---|---136
|---|---|---133
|---|---|---|---131
|---|---|---|---X
|---|---|---138
```

3. Suggest and show a method that sort the database in a tree where the node can store multiple value like the works of Bayer and McCreight. Take your own example (at least 100 data) where each node can store up to 5 values.

A. Explanation of the B-Tree method

The works of Bayer and McCreight led to the creation of a new model of Binary Search Tree: The B-Tree. The B-Tree has the following form:



A B-Tree of order M has the following properties:

- Every node has at most m children

- A non-leaf node with k children contains k-1 keys
- The root has at least two children if it is not a leaf node
- Every non-leaf node (except root) has hat least m/2 children
- All leaves appear in the same level

The B-trees have the big advantage of being balanced , and all the leaves are at the same height, which makes it possible to increase in the height and thus better complexity of the searching, inserting and deleting algorithms on the B-Tree are **O(log(n)).**

B. Insert in the B-Tree

First, we need to insert value into the B-Tree. (to see the code go to B_tree.py => _add(self, val, node, n))
For that, we must:

- Find the leaf node where the item should be inserted.
- If the leaf node can accommodate another item (it has no more than m-1 items), insert the item into the correct location in the node.
- If the leaf node is "full", split the node in two, with the smaller half of the items in one node and the larger half in the other. "Promote" the median item to the parent node. If the parent node is full, split it and repeat... It this reaches the root node; the height of the B-Tree will grow by one.

C. Search in the B-Tree

Here is the Python code for searching in a B-Tree:

```
def _find (self,id , node):
    if id in node.listVal:
        return True
    elif id < node.listVal[0] and node.listKey[0] != None:
        return self._find(id, node.listKey[0])

    elif id > node.listVal[len(node.listVal)-1]
                          and node.listKey[len(node.listVal)] != None:
        return self._find(id,node.listKey[len(node.listVal)])

    elif id > node.listVal[0]
                            and id < node.listVal[len(node.listVal)-1]:
        for i in  range(len(node.listVal)-1) :
            if node.listVal[i] < id and id < node.listVal[i+1]:
                if node.listKey[i+1] != None :
                    return self._find(id, node.listKey[i+1])
                else: return False
    else :
        return False #None
def find (self, ID):
    if(self.root == None): return None
    else:
        return self._find(ID,self.root)
```

Note: This algorithm was coded from scratch without being inspired by any code on internet.
To code this algorithm, we only used the link https://www.cs.usfca.edu/~galles/visualization/BTree.html : the animation on this website was very useful to help to understand the B-Tree method.
(For more details on this algorithm, Cf. B-Tree.py).