Step 1: To organize the tournament

1. Propose a data structure to represent a Player and its Score

A player is characterized by its name/pseudo, its score, and its rank. The name is going to identify and differentiate players. The rank is going to classify them in order to make the groups and eliminate the last ones. The score is mandatory to update the rank. We have 3 attributes so the best data structure to store them in one object player is to create a class Player.

When we initialize a Player, we give him a name, his score is set to 0 and his rank depends on how many players were initialized before him even though at the beginning it doesn't represent anything.

We actually took the time to create each one of the 100 players of the tournament. We picked the pseudo of the top 100 streamers in the world and wrote them in a text file participants.txt. Then with a dedicated function, we read this file and put every name in a list. After that we looped 100 times the Player class initialization (Player(name, rank=ith iteration)) to create our set of 100 Player objects.

We preferred this class structure to a simple dictionary (key:value)=(name:score) because we wanted to have this third attribute being the rank and also because we felt more comfortable using class. The data persistency of a class object, the possibility to add functions within the class related to the object (Update score, Reset Score, Update rank, etc.) and the ease of use and calling are all arguments that strengthened our choice.

2. Propose a most optimized data structures for the tournament (called database in the following questions)

Now that we have our Player structure, we need a tournament structure to organize and classify them through different games. The main purpose of this database is to be able to reach any player with a log complexity. And a simple list of the 100 players is not going to match this constraint because if we are looking for a player that is at the end of this list, then it is going to take longer than a log complexity to reach it.

Since we want to classify players according to their rank, we have an integer value which can be compared to others. In this case, a Binary Search Tree reveals itself to be suited to the situation. When we have to insert data in the tree, at each node (if not the root), the inserted data has two options depending on the value of the rank: if the rank of the inserted player is higher than the rank of the player in the node, then it goes in the right subtree; left otherwise. And if there is no subtree, then the inserted player creates a new one.

With this database, searching for an element becomes easier because we always have only two choices: either go right if the value we are looking for is higher than the one in the actual node, or go left, until we reach the exact same value (if it exists).

However, BST is not the best suited structure. And this is because if by any mean the values we insert in the tree are always higher than the previously inserted value, then we are going to find ourselves with a tree not really looking like a tree because it only has one right subtree at each node. Therefore, if the value we are seeking in the tree is at the end of it (the last right subtree), then our tree will be no better than a classic list and we will reach a linear complexity. In conclusion, we need to balance our tree.

In this case, the AVL tree reveals itself to be the most suited structure for our tournament. Indeed, AVL tree is a balanced Binary Search Tree, meaning the insertion of data works exactly like a BST but it differentiates itself in the balancing. If the right subtree of a node gets 2 subtrees higher than the left, then we apply rotations to rebalance the tree.

With this technic, the AVL tree search complexity cannot exceed log2(n) which is our requirement.

Finally, we have to notice that Nodes of our tree are not just composed by a left subtree, a right subtree, a height (for AVL) and a simple value data. In our case, it is more complex because our data are Players which are structures. The main difference is going to play inside the AVL tree functions : instead of comparing data (players) directly, we will need to compare players.rank.

3. Present and argue about a method that randomize player score at each game (between 0 point to 12 points)

In the Player Class, we implemented an UpdateScore method which, with a simple integer randomizer without weighted score, gives a score between 0 and 12. However, to keep the simulation as real as possible, we need to make scores like 10, 11 or 12 less probable than 4,5 or 6. We can use the technique of the dice increasing probability for average score and decreasing for extreme score (0,1,2,11,12). So here we have two “dices” containing numbers between 0 and 6 (included), hence a possible set of score in the range (0,12) (12 included).

Now, it is required that players score is not the sum of all scores from all their games but rather the mean of scores of all their games. Therefore, we cannot just add the randomized score from a game to the previous score, we have to multiply the previous score by (n-1)/n, n being the number of games played, and add to this the new score from this game divided by n. Then we have the mean score.

4. Present and argue about a method to update Players score and the database

After updating Players score, we must update the database because the ranking of each player has really strong chances of changing. And to do so, we cannot keep our current AVL tree and change each player’s position in it, this would be overly complicated. If the 100^{th} player scores 12 points, then we have to check if the player above him still has a higher score. If this is not the case, then player 100 exchanges places with player 99 and so on until player 100 finds a player that has a better score than his. And if you think this would be long, we must repeat this process as much times as there are players in the Tree. The conclusion is that it would take forever, and it is not a good solution to update players score within the tree.

However, if we just do an in-order search on the AVL tree, it will give us back the list of all players ordered by their ranking from the worst to the best. Then, we use this list to update players score simply looping iteratively and using UpdateScore function. Finally, with this list we create a new AVL tree which is going to class them upon their score and not their ranking (because it would not be updated at this time). Once all players are inserted inside the tree, another simple in-order search algorithm on the tree would give us the opportunity to rank them: the first one in the list being the 100^{th} of the tournament and the last of the list the first.

It is important to have this subsidiary list alongside the AVL tree because it is this one that we are going to use to update scores and ranking of players.

5. Present and argue about a method to create random games based on the database

The 3 first random games are warm up games. Their goal is to put players in the atmosphere of the tournament without any pressure of being kicked out at the end. To prepare them in the best way possible, they should not play again the exact same person 3 times in a row because this would not be representative of the average level: some groups of 10 might be more homogeneous than others, then giving the wrong impression to gamers (either the level is too high or too low). Hence the importance of well mixing the players in those 3 games.

For the first random game, all players score is set to 0 so we can't know their level. Plus, nobody played nobody so far so we can create games of 10 players however we want, there would not be redundancy. Then it is sold, players will play against each other depending on their ranking or more precisely the alphabetical order because this is how their first ranking was based on. And to get players in a ranking order from the AVL Tree, we have the in-order search algorithm.

Now for the second random game, scores are updated, so are ranks and the new AVL Tree is built. We want an algorithm that is going to give us back the list of players from this tree but in a different order from the previous one. And even though doing an in-order search algorithm would not give us the same list (in the same order) as earlier, this would not respect our wish of heterogeneity among the random games. To keep the splitting as random as possible we will use the fact that the ranking changes of course but also and mainly use a different search algorithm: pre-order or post-order. Here we use the pre-order search algorithm to organize the second random game.

For the last game, we repeat the same steps as earlier but apply the post-order search algorithm instead.

Even though we didn't really need scores and ranking for the creation of the 3 random games, we still needed to update them because they are mandatory for the rest of the tournament. Indeed, at the end of the 3 random games, the 100 players will be ranked according to their score so that the tournament and the disqualification games can begin. (the score of each player is not reset after the 3 games, it wouldn't be fair for top players).

6. Present and argue about a method to create games based on ranking

For the creation and repartition of players according to their rank, we had to make a choice:

- Rewarding the best players by making them play against the worst players
- Giving all players, good or bad, the same chance of success batching the best players together and the worst players together

The second option is simple: we get the list of players in-order from the AVL tree then split the list every ten players so that player ranked 1 play against 2, 3, 4 until 10, and players 91 to 100 play against. Unfortunately, this technic does not reward the best players, it evens disadvantages them. The 10^{th} player for example will be the worst player from his group and will not get the chance to score as many points as he could have if he had played against worse players.

However, since we are giving points randomly to each player and we do not respect some type of structured pointing system, this logic of rewarding gets blown away. Therefore, we decided to go for the simplest way of splitting groups from 10 to 10 rank-wisely. The process is quite straight forward:

get the list of rank-ordered players with the in-order algorithm, go through the list with a for loop and place players in a group (list), changing list every 10 players.

7. Present and argue about a method to drop the players and to play game until the last 10 players

Serious things begin. Shuffle games are over, and the official tournament takes place: 9 rounds of one game after each the 10 last players (least score) are permanently eliminated. 9 rounds because we need to eliminate 90 players to keep the 10 best for the final step of the tournament.

The scheme is the following: for the first round, we have our AVL Tree and our affiliated rank-ordered list.

- we update the scores (with the list) after the game
- we create the new AVL Tree
- we get the list of players ordered from the lowest score to the highest
- we update the ranks
- we delete the 10 first players of the list (10 worst)

So, we repeat this action 9 times.

The reason we do not practice a deletion directly on the AVL Tree is because when we create the AVL tree after the score update, we do not know the new rank of players. So, we cannot go through the AVL tree searching for the 10 worst players according to their ranking because it will not be up to date and we could find ourselves to eliminate the 89th player because his rank was 91 before the game. Furthermore, we cannot do the same thing with the score because we cannot know the exact score for which the 10 last players are below.

Hence the need to get the new list of ordered players (in-order algorithm) from the AVL Tree to update the ranks and delete the 10 worst players from it. At this point, we could question the utility of the AVL Tree since we don't practice any action on it like deletion. But in fact, the AVL Tree is super effective for classifying players and getting the list of players in order quickly.

8. Present and argue about a method which display the TOP10 players and the podium after the final game.

When only the 10 best players are remaining, the tournament organization changes. All scores are reset to 0 to not disadvantage the 10th player who has a lower score than the 1st. Players are going to fight for the podium around five games. After each game, scores are always updated, but ranks and also the AVL tree don't need to be because we don't attach any importance to the classification of players until the fifth and last game is played since there are no disqualifications.

Therefore, after the final game, with the list of 10 players, we create our AVL Tree inserting players according to their score and not to their not updated rank. Then we get the same list but ordered from the worst to the best with the in-order algorithm. Finally, we loop through the list starting from the end and display the rank, name and score of players from the best (1st) to the worst (10th).

Step 2: Professor Layton < Guybrush Threepwood < Us

1. Represent the relation (have seen) between players as a graph, argue about your model.

We are given several information:

- When a player sees another player, they always both see each other, there can't be one player seeing another without being seeing. This has for consequence that the adjacency matrix will be symmetrical.
- Impostors can't see each other

To register this information on who saw who, we decided to create an adjacency matrix after each crewmate's death. This matrix is a Boolean matrix saying if a player saw (True) or not (False) another one (1 if seen; 0 if not seen).

- This matrix is squared of size n*n, n being the number of players in the game (10 in our case).
- Since players can't see themselves, we have only 0 on the diagonal
- Since Impostors can't see each other, the value of the box those two share will always be equal to 0.
- Since one player can't see another without being seen by this latter, the matrix is logically symmetrical, meaning that we always have $n_{ij} = n_{ji} \; \forall \; i, j \in [0,9]$
- At each round, we have a dead player and the matrix is recalculated to see who saw who. The matrix is still going to be a n*n matrix except there will be only 0 on the rows and columns of the dead player.

For example, if we have 10 players and one got killed, we have a 10*10 matrix of Boolean. If player 0 who died has seen the players 1, 4 and 5, then the value n_{01} of the matrix is set to 1, same for n_{04} and n_{05}.

We also have the n_{10}, n_{40} and n_{50} values of the matrix set to 1 but it is not necessary to write them since the matrix is symmetrical. However, in the python code, it will be easier to complete the whole matrix because when we will count who saw who, it will be easier to count on one column or one line instead of one part on one line n and the other on column n.

Here is the representation of the Adjacency Matrix of the example given in the subject.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	0	1	0	0	1	1	0	0	0	0
1		0	1	0	0	0	1	0	0	0
2			0	1	0	0	0	1	0	0
3				0	1	0	0	0	1	0
4					0	0	0	0	0	1
5						0	0	1	1	0
6							0	0	1	1
7								0	0	1
8									0	0
9										0

2. Thanks to a **graph theory problem**, present how to find a set of probable impostors.

In the following explanation, we will show you different technics all based or using the adjacency matrix to unmask impostors before they kill every crewmate.

Logically, if a crewmate dies, then the impostor has to be among the members he has previously seen. Therefore, if 0 is found dead, then the impostor is either 1, 4 or 5, or maybe two of them. With this probable set of impostors we can estimate who is going to be the second impostor. Indeed, if for example, 1 is the impostor, and we know impostors don't see each other at all, the set of probable impostors is all the players 1 didn't see : 3, 4, 5, 7, 8 and 9. In our case, it isn't really helpful, but in a common game players ought to see more than 3 other players increasing our chances of finding the impostors with this technic.

Furthermore, we can't make assumptions after the first kill, unless a dead player saw only one other player. A better and more efficient way would be to store this information and then add it to the new information we get after the second kill. To do so, we decided to store the information on who saw who at each round. This means storing the adjacency matrices of each round in a list. This list has several utilities:

- First of all, it allows us to know who saw who at each round and update those information after each kill. In the case 0 was killed and saw 1, 4 and 5 but 4 and 5 died later in the game, we could deduce from this matrix that 1 is the impostor by updating the status of the player. A player can either be dead or innocent to not be considered suspect. This is the main feature of this list of matrices that we will get to use in almost all our functions.
- Then we can use this list to calculate the sum of all adjacency matrices to be able to count how many times players saw each other. This may seem useless but if we consider that impostors can't see each other and all other alive players saw all other alive ones at least once, then this means they can't be impostors. And at this moment, except from the 0 is the diagonal, there will be only one 0 in the matrix of summed adjacency matrices, and it will be in the box shared by the two impostors.

We just presented you our first technic to unmask impostors. This one can only be used when 2 impostors are remaining. But we have more than one trick up our sleeve and we will enumerate all of them.

The simple length of suspect list

When a player dies, we look upon the newly created adjacency matrix to get the list of players he saw also being the list of suspects for this murder. If the list contains only one player, then it has to be an Impostor.

The Impostorness coefficient

This technic is the only one above all others that eliminates players based on probabilities and not certainty. The Impostorness coefficient is a coefficient that increases of a certain amount depending on the number of other players the dead one saw. To take back our example, if 0 dies, then 1, 4 and 5 see their coefficient increased by 1 divided by 3. After the second kill, if 9 dies and he saw 1 and 4 then those players see their coefficient increased by 0.5. At this point, 1 and 4 both have coefficient equal to 0.83. They can both be impostors or only one of them.

But we need to update this coefficient after each kill and this is where the list of adjacency matrices reveals useful. For each round, we are going to see who died, then see who he saw, then among this list of seen players get the ones alive and not innocent, to finally update Impostorness coefficient of this round. To take back our example, if 5 dies on round 2, then it leaves the suspect of player 0 killed in round 1 to 1 and 4. So their Impostorness on this round coefficient needs to be updated from 0.33 to 0.5. And further is the game, if 4 dies, then with the Impostorness coefficient, we can bust player 1 who has now a coefficient equals to 1 for round 1 which is an enough condition to know for sure a player is an impostor.

When we said this technic was unsure and sometimes unreliable, it's because players have an Impostorness coefficient by round and also an overall Impostorness coefficient summing up coefficients of all rounds. And in the game, if we reach a critical situation when if impostors kill one more player, then the number of crewmates equals the number of impostors, crewmates need to make a decision not based on certainty but rather on probability. They will look at overall Impostorness coefficient of each alive players and eliminate the one who has the highest coefficient.

There also exists one situation where players would eliminate a crewmate. To keep on our example, if 1 and 6 are suspects on murder of player 4, and 1 is unmasked as an impostor, the algorithm is not going to exonerate 6 if he wasn't previously proven innocent. Then, after 1's death, 6's Impostorness coefficient would equals 1 and he would be eliminated. This is a unique situation where 6 really wasn't lucky and was in the wrong places each time. I don't see this situation as flawed but rather as a good thing since it doesn't make Crewmates win all the time but give Impostors a chance instead.

The Innocentation

We can prove players are innocent. To do so, we need to prove they saw every other alive players in the game. If this is the case, they can't be impostors, so they need to be considered as innocent. This state will be taken into consideration for Impostorness coefficient. We apply this technic after each kill and newly created adjacency matrix.

The Exoneration

This technic looks like the previous one by name, but it is different. We only apply it when one impostor is unmasked, and one is still remaining. If one impostor is found, then we can exonerate all players this impostor saw since impostors don't see each other. This exoneration is more efficient than the Innocentation technic because conditions to prove a player is innocent are less important (a player can only have seen one player being the impostor and he would be exonerated while for the Innocentation, he would have to see every other alive players).

3. Argue about an algorithm solving your problem.

To make sure our investigation technics work, we wanted to try it on a real situation. However, a real situation would be to play actual games of Among Us, be able to detect when players see each other and mostly that they respect all the assumptions we made (for example, the fact impostors can't see each other isn't representative of the reality because they can sometimes meet up).

The plan is that we choose the 2 impostors before the program launches so we know who they are but the computer doesn't and his goal is to unmask them. So, we randomly choose 2 impostors in the set of 10 players and then repeat the following steps until no impostors are remaining or the number of impostors equals the number of crewmates:

1. Create Adjacency matrix after one kill and add this new adjacency matrix to the list of adjacency matrices:
   - We randomly set values of 0 and 1 in the matrix
   - We make sure the matrix is symmetrical
   - We also are careful to not putting 1 between 2 impostors (meaning they saw each other) and take into consideration dead players who can't see nor been seen but are still mentioned in the adjacency matrix.
2. Innocentation of players:
   - If one player saw every other player alive, he is innocent
   - If all alive players are innocent except 2 and there are still 2 impostors remaining, then we bust them.
3. Check the summed adjacency matrix:
   - Only if the 2 impostors are remaining
   - Can bust either the two impostors or none but never just one
4. Kill one crewmate:
   - We make sure the player killed is alive
   - He also has to be one of the players seen by impostors (so we only take one player among the players seen by impostors)
   - Get the list of suspects for this kill to try to bust one impostor with length of list
   - If one impostor is busted and one is remaining, we Exonerate players seen by this dead impostor
5. Update Impostorness coefficients for this round and the previous ones:
   - If players got exonerated or killed, their Impostorness coefficient needs to be updated
   - If one updated coefficient reaches 1, the player is an impostor (in most cases)
   - If one impostor is busted and one is remaining, we Exonerate players seen by this dead impostor
6. If we are in a critical situation where there is only one crewmate more than there are impostors alive, crewmates need to decide to kill a player based on probability even if they are not sure:
   - We are at a point where if the argument doesn't unmask an impostor, the game ends (for example 3 crewmates and 2 impostors remain, if no impostor gets discovered, then next round there will be one crewmate left making it impossible for cremates to win).
   - Decision will be based on overall Impostorness coefficient of players: the one who has the highest gets eliminated
   - If an Impostor is unmasked, then we loop once again; otherwise, Impostors win.

After the description in natural language of our algorithm, let us show you the description in symbolic language:

```
players=[0,1,2,3,4,5,6,7,8,9]
Impostors=[1,7]
While (not game_ended)
        New_adjacency_matrix= Get_Matrix_meetings()
        List_matrices.append(New_adjacency_matrix)
        Innocentation()
        Check_Impostors(sum_matrices)
        In this function we do most of the work (like Exoneration and Update_Coeff)
        Kill_player()
        If (critical_situation)
                Eliminate_player_based_on_probability (overall Imp coeff)
```

4. Implement the algorithm and show a solution.

Until there, we often referred at our players as dead or innocent or suspect. In order to better register their state and store information on them, we created a special class Player which could store the following information : ID (integer between 0 and 9), alive (bool), innocent (bool), imp_coeff_round (list of floats between 0 and 1), overall_imp_coeff (float). This class, among with functions related to manipulation or recuperation of elements of this class, represents one file of the program.

The second file of this program contains the main functions like Create_Adjacency_Matrix, Kill_crewmate, Update_Coefficients, Check_Impostors, Exoneration or Innocent.

The third and last file is the main.py which runs the program and is the transcription in python code of the algorithm described in question 3.

Now, we are going to run several simulations to show how the program runs and works:

10 players and 2 impostors, we display new adjacency matrix, summed one and player killed:

```
Players remaining : 10
Impostors remaining : 2
Here is the adjacency matrix of round 1:
[[0. 0. 1. 1. 0. 0. 1. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0. 1. 0. 1. 1.]
 [1. 1. 0. 1. 0. 1. 0. 1. 1. 1.]
 [1. 0. 1. 0. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 1. 0. 1. 1. 0. 1. 1.]
 [0. 0. 1. 1. 1. 0. 1. 1. 1. 1.]
 [1. 1. 0. 1. 1. 1. 0. 1. 0. 0.]
 [0. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [1. 1. 1. 1. 1. 1. 0. 1. 0. 1.]
 [0. 1. 1. 1. 1. 1. 0. 1. 1. 0.]]
Here is the summed adjacency matrix of round 1:
[[0. 0. 1. 1. 0. 0. 1. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0. 1. 0. 1. 1.]
 [1. 1. 0. 1. 0. 1. 0. 1. 1. 1.]
 [1. 0. 1. 0. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 1. 0. 1. 1. 0. 1. 1.]
 [0. 0. 1. 1. 1. 0. 1. 1. 1. 1.]
 [1. 1. 0. 1. 1. 1. 0. 1. 0. 0.]
 [0. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [1. 1. 1. 1. 1. 1. 0. 1. 0. 1.]
 [0. 1. 1. 1. 1. 1. 0. 1. 1. 0.]]


Player 9 was killed by impostors.
He saw the following suspects (alive and not proven innocent) : [1, 2, 3, 4, 5, 7, 8]
```

Here we see that players 2, 3 and 5 were proven innocent. Therefore, we won't see them in the list of suspects displayed after each killed player.

```
Players remaining : 9
Impostors remaining : 2
Here is the adjacency matrix of round 2:
[[0. 0. 1. 1. 0. 1. 1. 0. 0. 0.]
 [0. 0. 1. 1. 1. 1. 1. 1. 1. 0.]
 [1. 1. 0. 1. 1. 0. 1. 1. 1. 0.]
 [1. 1. 1. 0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 0. 1. 1. 0. 1. 0.]
 [1. 1. 0. 1. 1. 0. 1. 0. 1. 0.]
 [1. 1. 1. 1. 1. 1. 0. 1. 0. 0.]
 [0. 1. 1. 1. 0. 0. 1. 0. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Here is the summed adjacency matrix of round 2:
[[0. 0. 2. 2. 0. 1. 2. 0. 1. 0.]
 [0. 0. 2. 1. 1. 1. 2. 1. 2. 1.]
 [2. 2. 0. 2. 1. 1. 1. 2. 2. 1.]
 [2. 1. 2. 0. 2. 2. 2. 2. 2. 1.]
 [0. 1. 1. 2. 0. 2. 2. 0. 2. 1.]
 [1. 1. 1. 2. 2. 0. 2. 1. 2. 1.]
 [2. 2. 1. 2. 2. 2. 0. 2. 0. 0.]
 [0. 1. 2. 2. 0. 1. 2. 0. 2. 1.]
 [1. 2. 2. 2. 2. 2. 0. 2. 0. 1.]
 [0. 1. 1. 1. 1. 1. 0. 1. 1. 0.]]

Player 2 was innocented because he saw every other players
Player 3 was innocented because he saw every other players
Player 5 was innocented because he saw every other players

Player 6 was killed by impostors.
He saw the following suspects (alive and not proven innocent) : [0, 1, 4, 7]
```

Nothing new happened at this round, no more players proven innocent and summed matrix still has more than 1 zero in it.

```
Players remaining : 8
Impostors remaining : 2
Here is the adjacency matrix of round 3:
[[0. 0. 1. 0. 0. 1. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 1. 0. 1. 1. 0.]
 [0. 0. 0. 0. 1. 1. 0. 1. 1. 0.]
 [0. 0. 1. 1. 0. 0. 0. 1. 1. 0.]
 [1. 1. 1. 1. 0. 0. 0. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 1. 1. 0. 0. 1. 0.]
 [1. 0. 1. 1. 1. 1. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Here is the summed adjacency matrix of round 3:
[[0. 0. 3. 2. 0. 2. 2. 0. 2. 0.]
 [0. 0. 2. 1. 1. 2. 2. 1. 2. 1.]
 [3. 2. 0. 2. 2. 2. 1. 3. 3. 1.]
 [2. 1. 2. 0. 3. 3. 2. 3. 3. 1.]
 [0. 1. 2. 3. 0. 2. 2. 1. 3. 1.]
 [2. 2. 2. 3. 2. 0. 2. 2. 3. 1.]
 [2. 2. 1. 2. 2. 2. 0. 2. 0. 0.]
 [0. 1. 3. 3. 1. 2. 2. 0. 3. 1.]
 [2. 2. 3. 3. 3. 3. 0. 3. 0. 1.]
 [0. 1. 1. 1. 1. 1. 0. 1. 1. 0.]]

Player 2 was innocented because he saw every other players
Player 3 was innocented because he saw every other players
Player 5 was innocented because he saw every other players

Player 8 was killed by impostors.
He saw the following suspects (alive and not proven innocent) : [0, 4, 7]
```

There interesting things happened. Players 4 and 7 were proven innocent making 5 of them now. Since only 7 players are remaining and we also know 2 impostors are remaining, this means we know impostors are the two alive players not proven innocent. Therefore, we can deduce who they are.

```
Here is the adjacency matrix of round 4:
[[0. 0. 1. 1. 1. 1. 0. 1. 0. 0.]
 [0. 0. 1. 1. 1. 1. 0. 1. 0. 0.]
 [1. 1. 0. 1. 1. 1. 0. 1. 0. 0.]
 [1. 1. 1. 0. 1. 1. 0. 0. 0. 0.]
 [1. 1. 1. 1. 0. 1. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Here is the summed adjacency matrix of round 4:
[[0. 0. 4. 3. 1. 3. 2. 1. 2. 0.]
 [0. 0. 3. 2. 2. 3. 2. 2. 2. 1.]
 [4. 3. 0. 3. 3. 3. 1. 4. 3. 1.]
 [3. 2. 3. 0. 4. 4. 2. 3. 3. 1.]
 [1. 2. 3. 4. 0. 3. 2. 1. 3. 1.]
 [3. 3. 3. 4. 3. 0. 2. 3. 3. 1.]
 [2. 2. 1. 2. 2. 2. 0. 2. 0. 0.]
 [1. 2. 4. 3. 1. 3. 2. 0. 3. 1.]
 [2. 2. 3. 3. 3. 3. 0. 3. 0. 1.]
 [0. 1. 1. 1. 1. 1. 0. 1. 1. 0.]]

Player 2 was innocented because he saw every other players
Player 3 was innocented because he saw every other players
Player 4 was innocented because he saw every other players
Player 5 was innocented because he saw every other players
Player 7 was innocented because he saw every other players

Player 0 was eliminated by crewmates because he was an impostor.
Busted by Innocentation of all crewmates.


Player 1 was eliminated by crewmates because he was an impostor.
Busted by Innocentation of all crewmates.
```

When we kill one impostor, we precise the reason why they were busted.

We don’t forget to display a winning (or defeating) message at the end.

```
Congratulations to crewmates, you found and eliminated all impostors!
```

Now we run a second simulation. This one took only two rounds to bust both impostors. What happened is the following:

On the first picture, everything is normal, player 4 was killed and he saw a lot of suspects (0, 2, 3, 5, 6, 8, and 9) nobody was proven innocent.

On the second picture, we have 6 players proven innocents at once: 0, 2, 3, 5, 6, 9. Since player 8 was killed, when we updated the list of suspects of player 4’s kill, it only remains player 0 who was neither proven innocent nor dead. After the unmasking of 0, it is not hard to unmask the other impostor since 0 saw every other players alive except one: 1.

There are a lot of outcomes different for this algorithm (impostors can win, we can have a critical situation, other busting processes) but we chose to only show you 2 to make it short.

```
Players remaining : 10
Impostors remaining : 2
Here is the adjacency matrix of round 1:
[[0. 0. 0. 1. 1. 1. 1. 1. 1. 0.]
 [0. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [0. 1. 0. 1. 1. 1. 1. 1. 0. 1.]
 [1. 1. 1. 0. 1. 0. 0. 1. 0. 1.]
 [1. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [1. 1. 1. 0. 1. 0. 1. 1. 1. 1.]
 [1. 1. 1. 0. 1. 1. 0. 0. 0. 1.]
 [1. 0. 0. 1. 0. 1. 0. 0. 1. 1.]
 [1. 1. 0. 0. 1. 1. 0. 1. 0. 0.]
 [0. 1. 1. 1. 1. 1. 1. 1. 0. 0.]]
Here is the summed adjacency matrix of round 1:
[[0. 0. 0. 1. 1. 1. 1. 1. 1. 0.]
 [0. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [0. 1. 0. 1. 1. 1. 1. 1. 0. 1.]
 [1. 1. 1. 0. 1. 0. 0. 1. 0. 1.]
 [1. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [1. 1. 1. 0. 1. 0. 1. 1. 1. 1.]
 [1. 1. 1. 0. 1. 1. 0. 0. 0. 1.]
 [1. 0. 0. 1. 0. 1. 0. 0. 1. 1.]
 [1. 1. 0. 0. 1. 1. 0. 1. 0. 0.]
 [0. 1. 1. 1. 1. 1. 1. 1. 0. 0.]]


Player 4 was killed by impostors.
He saw the following suspects (alive and not proven innocent) : [0, 2, 3, 5, 6, 8, 9]
```

```
Players remaining : 9
Impostors remaining : 2
Here is the adjacency matrix of round 2:
[[0. 0. 1. 0. 0. 0. 0. 1. 0. 1.]
 [0. 0. 0. 0. 0. 0. 1. 0. 1. 1.]
 [1. 0. 0. 1. 0. 1. 1. 1. 1. 1.]
 [0. 0. 1. 0. 0. 1. 1. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 0. 0. 1. 1. 1. 1.]
 [0. 1. 1. 1. 0. 1. 0. 1. 1. 1.]
 [1. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [0. 1. 1. 1. 0. 1. 1. 1. 0. 1.]
 [1. 1. 1. 1. 0. 1. 1. 1. 1. 0.]]
Here is the summed adjacency matrix of round 2:
[[0. 0. 1. 1. 1. 1. 1. 2. 1. 1.]
 [0. 0. 1. 1. 0. 1. 2. 0. 2. 2.]
 [1. 1. 0. 2. 1. 2. 2. 1. 1. 2.]
 [1. 1. 2. 0. 1. 1. 1. 2. 1. 2.]
 [1. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [1. 1. 2. 1. 1. 0. 2. 2. 2. 2.]
 [1. 2. 2. 1. 1. 2. 0. 1. 1. 2.]
 [2. 0. 1. 2. 0. 2. 1. 0. 2. 2.]
 [1. 2. 1. 1. 1. 2. 1. 2. 0. 1.]
 [1. 2. 2. 2. 1. 2. 2. 2. 1. 0.]]

Player 2 was innocented because he saw every other players
Player 3 was innocented because he saw every other players
Player 5 was innocented because he saw every other players
Player 6 was innocented because he saw every other players
Player 8 was innocented because he saw every other players
Player 9 was innocented because he saw every other players

Player 8 was killed by impostors.
He saw the following suspects (alive and not proven innocent) : [1, 7]
```

```
Player 0 was eliminated by crewmates because he was an impostor.
Busted by Length of suspects list updated (after removing dead or innocent suspects).


Player 1 was eliminated by crewmates because he was an impostor.
Busted by Length of suspects list updated (after removing dead or innocent suspects).



Congratulations to crewmates, you found and eliminated all impostors!
```

Here is one example of Impostors winning the game:

```
We have a critical situation! Crewmates need to take a decision:
3 was eliminated because he had an Impostorness coefficient equals to 1.75
He was not an impostor!
There is now as much crewmates as impostors, crewmates can no longer win...


Impostors [0, 1] got the best of you and killed everyone!
```

Step 3: I don't see him, but I can give proofs he vents!

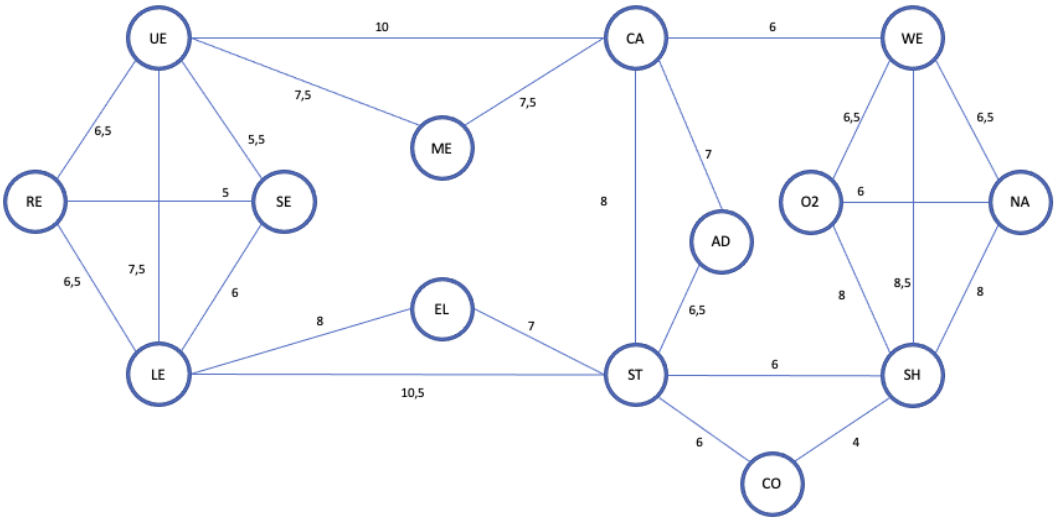
Presents and argue about the two models of the map.

To represent the map, we are going to use undirected graph for both models. To calculate the weight of each edge we say that 1cm has a cost of 1. To determine all the weight, we made an impression of the map and measured all the distance between each room.

First model: crewmate without vents

In the first model, players can go in each room in both ways (from B to A and A to B), each way (aka edges) between rooms (aka nodes) is weighted by the amount of time needed to go from one room to another. The weight is going to be the same both way, and anyone can travel from a room to another both way that is why we use an undirected graph.

ASDA Map for the crewmate:

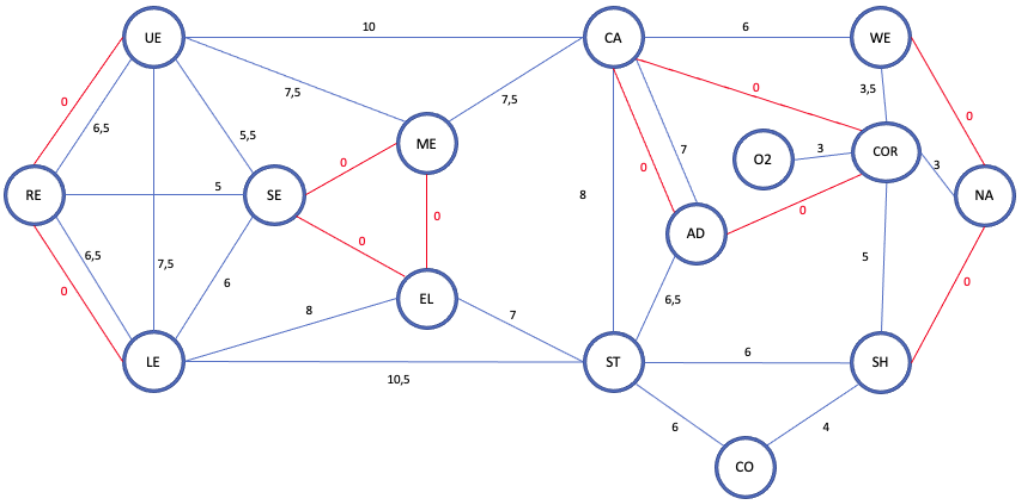


Second model: impostor with vents

This graph is a subgraph of the previous one, meaning we take the same graph and add edges that represent vents to it. Vents are new edges that impostor can take to travel faster through rooms and because the time to travel in vents is null, the weights on these new edges are all 0.

In fact, we first delete all the edges that connect the room “WE”, “O2”, “NA” and “SH” because we are going to add a new ‘fake’ room “COR” (for corridor). We need to create this new room because one vent leads to no room but just a corridor between rooms. So, once we delete those edges and add the new ‘fake’ room, we add new edges from this ‘fake’ room with cost that respect and correspond to the crewmate map. With this method, we indeed added a new ‘fake’ room “COR” to allow us to create new vents in respect of the previous map so the cost for all path stay the same even with this new ‘fake’ room.

ASDA Map for the impostor :



Vents are represented in red with a cost of 0.

We can see that on the impostor map we indeed added a 'fake room' which is 'COR' for corridor (on the right part of the map). We added this room because the impostor has a vent which leads to a corridor and not a room, so we needed to add this 'fake room' so that the impostor map respect as much as possible the original map. This 'fake room' will not be counted in the list of rooms to do our research to unmask the impostor, it is only there as a path for the vent to respect the original map. This 'fake room' doesn't change the weight of the path between the reals rooms which is very important for the rest of the project.

How do we create the map?

Here we choose to create the map with 2 lists: one that contain the list of nodes and the other one the list of edges. In fact, we use this method instead of a traditional adjacency matrix because when we add vent to our crew mate graph to create the impostor graph, we don't want to replace the cost of the already existing edge by zero, because this is not the case. The real case is that we add edges (vents with cost equals to zero) to the already existing edges. This addition is doable with list of edges but not with adjacency matrix because we can't put 2 different weight for one edge with adjacency matrix. We will see later how our list are created.

How to unmask the impostor?

To unmask the impostor, we need to calculate the time a player takes to go from one room to another. If this time is less than the time a crewmate is supposed to have in the best case, then the player is an impostor.

Argue about a pathfinding algorithm to implement.

The goal here is to find a pathfinding algorithm that best fit the model with the minimum complexity. Since both our graph doesn't contain negative weights, we don't need to use the Floyd-Warshall and

Bellman-Ford algorithm, and since we have more edges than nodes it is better to use the Dijkstra algorithm for our problem. Indeed, Dijkstra algorithm has equal or better space complexity than the two other algorithm. Dijkstra also has a time complexity of $O(N^2)$ compare to O(MN) for Bellmen-Ford and $O(N^3)$ for Floyd-Warshall (with N number of nodes and M edges). So, we should take the Dijkstra algorithm. We need to take the shortest path between two room so that we can compare them between an impostor and a crewmate to find the impostor.

Dijkstra's algorithm allows us to have the minimum cost from an initial node to all the nodes of the map. The goal now is to implement this method

Implement the method and show the time to travel for any pair of rooms for both models.

Method:

They are 5 different files in this part for 5 different functions: GraphClass, Create_map_function, Q2_Dijkstra, Q3_Show_result and the main.

First, before we calculate the Dijkstra algorithm we need to build the map.

GraphClass:

This file is here to create a class of graph. This class will allow us to create a graph and have access to its nodes and edges. Here we don't use adjacency matrix for the reasons said in previous part. Instead of an adjacency matrix we use a dictionary to store the nodes and a list for the edges. This class doesn't take any parameter but has indeed 2 variables:

- Dico_nodes : which is a dictionary that takes the name of the node as a key and the value of the node as a value. The value of the node will be use in the Dijkstra algorithm and this is why we use a dictionary. The value is initialized to zero.
- List_edges : which is a list composed of every edges that compose the graph. Each element of this list is an edge stored in this form: (start node, end node, cost). When referring to a node we use its name (meaning its key value)

Because it doesn't take any parameter, we need to add constructors. To add node and edge to our graph we create two functions to this class:

- Add_node : to add a node to dico_node of a graph, we add a node with the name of the node (it will be the key value)
- Add_edge : to add an edge list_edge of a graph, we add an edge with its start node, end node and its cost (weight)

This file will allow us to create an instance of graph for the crew mate and one for the impostor. This method is very important to create the impostor graph because of the reasons said before. This method is respectful for both graphs. We will use those graphs for the Dijkstra's algorithm.

Create_map_function

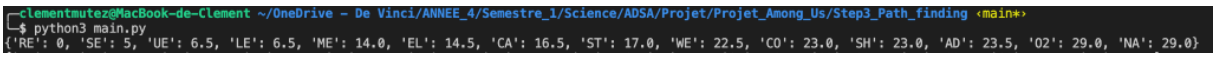
This file allows us to create instance of the 2 different maps (impostor and crewmate) from the GraphClass. To do so, we use the 2 function from the graph class to add node and edges to a graph. We created these 2 graphs in respect of the 2 maps showed in the first part.

Q2_dijkstra

In this file we have the main function of all this part: the Dijkstra algorithm. Dijkstra function takes 2 parameters: a graph and an initial node. This function will return a list of all the node of the graph with their minimum cost to travel from the initial node to them. Here is how it works:

- ⇒ First, we initialize the value of the initial node to zero and the other to infinity. The value of each node will represent its distance to travel from the initial node to it. That is why the initial node has a value/cost of zero. And that is also why we use a dictionary to store node.
- ⇒ Then while there is an unvisited node, we search for Dijkstra which means:
  - For each node that we already visited we look at their adjacent node (their neighbour)
    - For each adjacent node we look at their cost, which is the cost of the node already visited + the cost of the edge between the node already visited and the adjacent node.
    - The goal then is to visit every neighbour, calculate their cost, and take the minimum
  - Once we have the minimum, we store it (name of the node and value) in a list of already visited node
- ⇒ Finally, we return the list of already visited node that contain all the node with their cost to travel from the original node to them.

Here is an example of the result we obtain with this Dijkstra algorithm:



Here the original node is “RE” since it is the first in the list, and it has a cost of zero. Then to understand the list, each node has a value, this value is the minimal cost a player has to take to travel from the node “RE” to each node. For example here to go from “RE” to “WE” a player takes 22.5 seconds.

For more details on how this function works, please take a look at the comment of the python’s program that explains clearly how it works.

Q3_show_result

This file is here to present the time to travel for any pair of room for both models. It is composed of 2 main functions:

- All_time : take a graph as parameter and return the list of all the pair of room possible with their cost to travel from one to another. We use the Dijkstra function here to calculate all the different cost.
- Print_the_result : allows us to take the list of result from the previous function and display the result in a good way. In fact, it prints the result in matrix form. Each row and column have a room name (we use data frame). Each value of the matrix corresponds to the minimal distance to travel from its column’s name to its row’s name.

Main

In this file we can test all the previous functions:

- Question 2: to show the Dijkstra algorithm result. You can try to change the original room to see the different cost

- Question 3: to display the result of the Dijkstra algorithm for any pair of room for both graphs.

Little reminder to understand well our result. Each value in the matrix correspond to the minimal time to travel from its row's name (which is a room) to its column's name (which is also a room).

Here is the result for the crew mate map:

For example, first line ("ME") and second column ("AD") the value 7 means that the time to travel from "ME" to "AD" for a crew mate is 14,5.

```
Here are the result for the crew mate :

      ME    AD    UE    RE    SE    LE    EL    ST    CO    SH    O2    WE    NA    CA
ME   0.0  14.5   7.5  14.0  13.0  15.0  22.5  15.5  21.5  21.5  20.0  13.5  20.0   7.5
AD  14.5   0.0  17.0  23.5  22.5  17.0  13.5   6.5  12.5  12.5  19.5  13.0  19.5   7.0
UE   7.5  17.0   0.0   6.5   5.5   7.5  15.5  18.0  24.0  24.0  22.5  16.0  22.5  10.0
RE  14.0  23.5   6.5   0.0   5.0   6.5  14.5  17.0  23.0  23.0  29.0  22.5  29.0  16.5
SE  13.0  22.5   5.5   5.0   0.0   6.0  14.0  16.5  22.5  22.5  28.0  21.5  28.0  15.5
LE  15.0  17.0   7.5   6.5   6.0   0.0   8.0  10.5  16.5  16.5  24.5  23.5  24.5  17.5
EL  22.5  13.5  15.5  14.5  14.0   8.0   0.0   7.0  13.0  13.0  21.0  21.0  21.0  15.0
ST  15.5   6.5  18.0  17.0  16.5  10.5   7.0   0.0   6.0   6.0  14.0  14.0  14.0   8.0
CO  21.5  12.5  24.0  23.0  22.5  16.5  13.0   6.0   0.0   4.0  12.0  12.5  12.0  14.0
SH  21.5  12.5  24.0  23.0  22.5  16.5  13.0   6.0   4.0   0.0   8.0   8.5   8.0  14.0
O2  20.0  19.5  22.5  29.0  28.0  24.5  21.0  14.0  12.0   8.0   0.0   6.5   6.0  12.5
WE  13.5  13.0  16.0  22.5  21.5  23.5  21.0  14.0  12.5   8.5   6.5   0.0   6.5   6.0
NA  20.0  19.5  22.5  29.0  28.0  24.5  21.0  14.0  12.0   8.0   6.0   6.5   0.0  12.5
CA   7.5   7.0  10.0  16.5  15.5  17.5  15.0   8.0  14.0  14.0  12.5   6.0  12.5   0.0
```

And here is the result for the impostor map:

For example, first line ("ME") and second column ("AD") the value 7 means that the time to travel from "ME" to "AD" for an impostor is 7,5.

```
Here are the result for the impostor :

      ME    AD    UE    RE    SE    LE    EL    ST    CO    SH    O2    WE    NA    CA
ME   0.0   7.5   5.0   5.0   0.0   5.0   0.0   7.0  13.0  10.5  10.5  10.5  10.5   7.5
AD   7.5   0.0  10.0  10.0   7.5  10.0   7.5   6.5   7.0   3.0   3.0   3.0   3.0   0.0
UE   5.0  10.0   0.0   0.0   5.0   0.0   5.0  10.5  16.5  13.0  13.0  13.0  13.0  10.0
RE   5.0  10.0   0.0   0.0   5.0   0.0   5.0  10.5  16.5  13.0  13.0  13.0  13.0  10.0
SE   0.0   7.5   5.0   5.0   0.0   5.0   0.0   7.0  13.0  10.5  10.5  10.5  10.5   7.5
LE   5.0  10.0   0.0   0.0   5.0   0.0   5.0  10.5  16.5  13.0  13.0  13.0  13.0  10.0
EL   0.0   7.5   5.0   5.0   0.0   5.0   0.0   7.0  13.0  10.5  10.5  10.5  10.5   7.5
ST   7.0   6.5  10.5  10.5   7.0  10.5   7.0   0.0   6.0   6.0   9.5   6.0   6.0   6.5
CO  13.0   7.0  16.5  16.5  13.0  16.5  13.0   6.0   0.0   4.0  10.0   4.0   4.0   7.0
SH  10.5   3.0  13.0  13.0  10.5  13.0  10.5   6.0   4.0   0.0   6.0   0.0   0.0   3.0
O2  10.5   3.0  13.0  13.0  10.5  13.0  10.5   9.5  10.0   6.0   0.0   6.0   6.0   3.0
WE  10.5   3.0  13.0  13.0  10.5  13.0  10.5   6.0   4.0   0.0   6.0   0.0   0.0   3.0
NA  10.5   3.0  13.0  13.0  10.5  13.0  10.5   6.0   4.0   0.0   6.0   0.0   0.0   3.0
CA   7.5   0.0  10.0  10.0   7.5  10.0   7.5   6.5   7.0   3.0   3.0   3.0   3.0   0.0
```

Display the interval of time for each pair of room where the traveler is an impostor.

For this part we add another file: Q4_Compare.

Q4_compare

The goal here is to display the difference of time to travel between any pair of room between a crew mate and an impostor. Here we use the function 'all_time' from the previous python file (Q3) for both

models. Then we calculate the difference for each pair of room. Finally, we display the difference of travel time between each pair of room.

We also add this function to the main file so we can show our result with the function Question 4.

Here is the result:

To read the matrix, it is approximately the same method as before, but this time the value corresponds to the difference of the minimum time to travel (between rooms) between a crew mate and an impostor.

For example, first line (“ME”) and second column (“AD”) the value 7 means that the difference of time to travel from “ME” to “AD” between a crew mate and an impostor is 7.

```
Here are the result for the difference of travel time between impostor and crew mate :

      ME    AD    UE    RE    SE    LE    EL    ST    CO    SH    O2    WE    NA    CA
ME   0.0   7.0   2.5   9.0  13.0  10.0  22.5   8.5   8.5  11.0   9.5   3.0   9.5   0.0
AD   7.0   0.0   7.0  13.5  15.0   7.0   6.0   0.0   5.5   9.5  16.5  10.0  16.5   7.0
UE   2.5   7.0   0.0   6.5   0.5   7.5  10.5   7.5   7.5  11.0   9.5   3.0   9.5   0.0
RE   9.0  13.5   6.5   0.0   0.0   6.5   9.5   6.5   6.5  10.0  16.0   9.5  16.0   6.5
SE  13.0  15.0   0.5   0.0   0.0   1.0  14.0   9.5   9.5  12.0  17.5  11.0  17.5   8.0
LE  10.0   7.0   7.5   6.5   1.0   0.0   3.0   0.0   0.0   3.5  11.5  10.5  11.5   7.5
EL  22.5   6.0  10.5   9.5  14.0   3.0   0.0   0.0   0.0   2.5  10.5  10.5  10.5   7.5
ST   8.5   0.0   7.5   6.5   9.5   0.0   0.0   0.0   0.0   0.0   4.5   8.0   8.0   1.5
CO   8.5   5.5   7.5   6.5   9.5   0.0   0.0   0.0   0.0   0.0   2.0   8.5   8.0   7.0
SH  11.0   9.5  11.0  10.0  12.0   3.5   2.5   0.0   0.0   0.0   2.0   8.5   8.0  11.0
O2   9.5  16.5   9.5  16.0  17.5  11.5  10.5   4.5   2.0   2.0   0.0   0.5   0.0   9.5
WE   3.0  10.0   3.0   9.5  11.0  10.5  10.5   8.0   8.5   8.5   0.5   0.0   6.5   3.0
NA   9.5  16.5   9.5  16.0  17.5  11.5  10.5   8.0   8.0   8.0   0.0   6.5   0.0   9.5
CA   0.0   7.0   0.0   6.5   8.0   7.5   7.5   1.5   7.0  11.0   9.5   3.0   9.5   0.0
```

Step 4: Secure the last tasks

Presents and argue about the model of the map.

In this part, we only need the crew mate graph since they are the only one to have task to do. So here, compare to the previous part, we can use adjacency matrix to create our graph because no edges are going to have 2 different weight. It seems to be a good option to use the adjacency matrix here since we want to find a path and we are going to look at adjacent room to find it. Here is the adjacency matrix we are going to use:

Colon	ME	AD	UE	RE	SE	LE	EL	ST	CO	SH	O2	WE	NA	CA
ME	0	0	7,5	0	0	0	0	0	0	0	0	0	0	7,5
AD	0	0	0	0	0	0	0	6,5	0	0	0	0	0	7
UE	7,5	0	0	6,5	5,5	7,5	0	0	0	0	0	0	0	10
RE	0	0	6,5	0	5	6,5	0	0	0	0	0	0	0	0
SE	0	0	5,5	5	0	6	0	0	0	0	0	0	0	0
LE	0	0	7,5	6,5	6	0	8	10,5	0	0	0	0	0	0
EL	0	0	0	0	0	8	0	7	0	0	0	0	0	0
ST	0	6,5	0	0	0	10,5	7	0	6	6	0	0	0	8
CO	0	0	0	0	0	0	0	6	0	4	0	0	0	0
SH	0	0	0	0	0	0	0	6	4	0	8	8,5	8	0
O2	0	0	0	0	0	0	0	0	0	8	0	6,5	6	0
WE	0	0	0	0	0	0	0	0	0	8,5	6,5	0	6,5	6
NA	0	0	0	0	0	0	0	0	0	8	6	6,5	0	0
CA	7,5	7	10	0	0	0	0	8	0	0	0	6	0	0

This adjacency matrix takes exactly the same node, edges and weight than the graph in the previous part (Go to Step 3, question 1 if you want to check).

To build this graph with an adjacency matrix we use the library pandas that allows us to use data frame. Data frame allows us to create a matrix with labelled axes (row and column). This way we can access each row or column either by its index or by its name (room name here). This way we would be able in the future to use the index of each room but also to have access to the name of each row or column (that correspond to the name of the room).

Thanks to a graph theory problem, present how to find a route passing through each room only one time.

The goal here is to find a graph theory problem that find a route passing through each room only one time. The Hamiltonian problem fit well to our problem. Hamiltonian path works for undirected graph and is a path that visits each node exactly once. So, it is perfect for our problem.

In this context we only need to use the Hamiltonian path and not the Hamiltonian cycle since we don't need to go back to the original node to finish the different task.

Since we always 're start' a game after an emergency call at the Cafeteria, we will take one of the adjacent rooms of the Cafeteria as a starting node for our Hamiltonian path. Then from one of those adjacent rooms, we will use Hamiltonian path algorithm to find a route passing through every room to make sure every last crew mate can finish their task. Since crew mates will form a pack to travel through rooms, it is not a problem if they visited rooms with no task to do, because as a group they can't be killed.

So, the goal is in fact to find a Hamiltonian path that will be efficient at the end of the game and works for every game.

Argue about an algorithm solving your problem.

First of all, we need to create a map, the crew mate map. As we said before we create the map with an adjacency matrix. To do so we create a class Graph in the file GraphClass. The class graph takes one parameter: the number of nodes in the graph, and has two variables:

- Graph: which is the adjacency matrix in the form of a data frame
- N: which is the number of nodes in the graph

Once we have created our graph, we want to find the Hamiltonian path. To do so we use 3 functions: Ham_Solution, Hamiltonian, isGood.

Hamiltonian:

Main function of this part, it computes the Hamiltonian algorithm. It is a recursive function that takes a path and a position in the path as a parameter. It works like this:

→ First, we check if the state is final
→ Then if it is not final, for each node in the adjacency matrix we verify if the node is a good candidate, which means not already in the path and adjacent to the previous room added in the path. If the room is good, we add the room in the path and lunch the Hamiltonian again with a new path and a position in the path incremented of one. (this is the recursive part)
→ If it has found a Hamiltonian path it returns true otherwise false

Is Good:

This function is here to verify that a candidate (a room) is a good one. As said previously a good candidate is not already in the path and is adjacent to the previous node.

Ham_Solution:

This function is here to lunch the Hamiltonian algorithm from an initial node and print the result. To do so we first initialized the path with the starting node, and then we lunch the Hamiltonian algorithm with as parameter the path and as position 1 (since one node is already in the path)

Adjacency matrix was indeed a good choice since we want to check in the Hamiltonian path algorithm if a node is adjacent to another.

Finally, we have created a function Question4 and adjacent_room in the main function to lunch the Hamiltonian algorithm and show the result. The function adjacent_room is just here to find the

adjacent room of Cafeteria. Then in Question4 we create the crew mate graph and then run the Hamiltonian algorithm for each adjacent room.

For more detail on how it works, please take a look at the comment on the python's program.

Implement the algorithm and show a solution.

After we lunch the Question4 function in the main we obtain this result:

```
clementmutez@MacBook-de-Clement-1 ~/OneDrive - De Vinci/ANNEE_4/Semestre_1/Science/ADSA/Projet/Projet_Among_Us/Step4_Secure_Last_task <main*>
$ python3 main.py
One Hamiltonian path has been found from the original node : ME
ME UE RE SE LE EL ST AD CA WE O2 NA SH CO

One Hamiltonian path has been found from the original node : AD
AD ST EL LE RE SE UE ME CA WE O2 NA SH CO

Solution does not exist when we take UE as the original room

Solution does not exist when we take ST as the original room

One Hamiltonian path has been found from the original node : WE
WE O2 NA SH CO ST AD CA ME UE RE SE LE EL
```

As a final result we can see that there are 3 Hamiltonian path that exist from the 5 different adjacent rooms of Cafeteria. So, in late game, crew mate should take one of these paths if they want to be sure to win the game.

Important

To launch the 4 different programs, here is the list of python libraries to install:

- Pandas
- Random

And don't forget we separated the project into 4 different repositories. So if you want to launch one part from the command line, you have to go in the right repository and then to type "python main.py".