Exercice Amis parfais : graphe d’amis (6 points)

1. **Décrivez votre approche pour résoudre ce problème**

**1ère étape :**

Analyser les entrées : On commence par lire l'entrée qui contient le nombre de personnes et les relations d'amitié entre elles. On peut stocker ces informations dans une structure de données appropriée, telle qu'un dictionnaire ou une liste de listes.

**2ème étape :**

Trouver les groupes existants : En utilisant les relations d'amitié données, on peut construire les groupes existants en trouvant les composantes connexes du graphe non orienté qui représente les relations d'amitié entre les personnes.

**3ème étape :**

Compter le nombre de groupes existants : Le nombre de groupes existants correspond simplement au nombre de composantes connexes du graphe non orienté.

**4ème étape :**

Trouver toutes les combinaisons possibles de nouveaux groupes : Pour trouver toutes les combinaisons possibles de nouveaux groupes, on peut utiliser une approche de combinaison en prenant un élément de chaque groupe existant. Pour cela, on peut utiliser une boucle pour parcourir chaque groupe existant et une autre boucle pour parcourir les membres de chaque groupe.

**5ème étape :**

Compter le nombre de nouveaux groupes possibles : Le nombre de nouveaux groupes possibles correspond simplement au nombre total de combinaisons de membres de chaque groupe existant. On peut utiliser la formule mathématique "nouveau nombre de groupes = produit des tailles des groupes existants" pour calculer ce nombre.

**6ème étape :**

Afficher les résultats : Enfin, on peut afficher le nombre de groupes existants et le nombre de nouveaux groupes possibles.

Le code utilise une liste d'adjacence pour représenter le graphe et compte le nombre de groupes existants et de nouveaux groupes qui peuvent être formés. La méthode **addRelation** est utilisée pour ajouter des arêtes entre les nœuds et la méthode **countUtil** utilise un parcours en profondeur pour compter le nombre de nœuds visités. La méthode **countGroups** utilise la méthode **countUtil** pour compter le nombre de groupes existants et le nombre de nouveaux groupes pouvant être formés. La méthode **main** crée un objet **Graph**, ajoute des relations et appelle la méthode **countGroups** pour afficher le nombre de groupes existants et de nouveaux groupes pouvant être formés.

1. **Voir code**
2. **Déterminez la complexité temporelle de votre solution**

La complexité temporelle de cette solution est de où N est le nombre de nœuds dans le graphe et M est le nombre d'arêtes. La méthode **countGroups** parcourt tous les nœuds du graphe une seule fois en utilisant la méthode **countUtil**, qui a une complexité de De plus, pour chaque nœud visité, il effectue une recherche en profondeur sur ses voisins pour compter le nombre total de nœuds appartenant à ce groupe. La méthode **addRelation** a également une complexité de car elle ajoute simplement une arête entre deux nœuds en modifiant la liste d'adjacence. Par conséquent, la complexité temporelle totale de cette solution est de

Exercice : Nombre d’îles (8 points)

1. **Décrivez votre approche pour résoudre ce problème**

Pour résoudre ce problème, je vais utiliser l'algorithme DFS (Depth First Search) qui permet de parcourir en profondeur tous les sommets d'un graphe ou d'une matrice en explorant chaque branche aussi loin que possible avant de revenir en arrière. Dans notre cas, chaque case de la matrice représente un sommet et les 8 cases adjacentes à chaque sommet représentent les branches à explorer en profondeur. Ainsi, pour compter le nombre d'îles dans la matrice, je vais parcourir toutes les cases de la matrice en utilisant l'algorithme DFS pour explorer chaque île. Pour chaque île explorée, j'ajouterai 1 au compteur et je continuerai à parcourir les cases restantes jusqu'à ce que toutes les îles soient explorées.

1. **Voir code**
2. **Voir code**
3. **Déterminez la complexité temporelle de votre solution**

La complexité temporelle de l'algorithme est car dans le pire des cas, il parcourt chaque élément de la matrice une fois. De plus, la fonction DFS peut être appelée plusieurs fois, mais elle ne traite jamais plus de N*\**M éléments au total. Par conséquent, la complexité temporelle totale de l'algorithme est de .

Exercice : Vols les moins chers avec k arrêts (6 points)

1. **Décrivez votre approche pour résoudre ce problème**

L'approche utilisée pour résoudre ce problème est une variante de l'algorithme de recherche en largeur avec une file de priorité. Voici les étapes de l'approche :

1. Création du graphe : Convertir le tableau de vols donné en un graphe sous forme de liste d'adjacence. Chaque ville est représentée par un entier, et chaque arête est représentée par une paire (destination, prix). Le graphe est stocké dans une HashMap, où la clé est la ville source et la valeur est une liste de paires (destination, prix).

2. Initialisation : Créer une file de priorité (min-heap) pour stocker les nœuds à explorer. Les nœuds sont des tuples (ville, nombre d'arrêts, coût total). La file de priorité est ordonnée par coût total croissant. Ajouter le nœud source avec 0 arrêts et un coût total de 0 à la file de priorité.

3. Exploration : Tant que la file de priorité n'est pas vide, effectuer les étapes suivantes :

a. Extraire le nœud avec le coût total le plus bas de la file de priorité.

b. Si la ville du nœud extrait est la destination, renvoyer le coût total.

c. Si le nombre d'arrêts du nœud extrait est inférieur ou égal à K, ajouter les voisins de la ville du nœud extrait à la file de priorité avec leur coût cumulé et le nombre d'arrêts incrémenté.

4. Résultat: Si aucune route n'est trouvée après avoir exploré tous les nœuds possibles, renvoyer -1.

Cette approche explore les chemins possibles en tenant compte du coût et du nombre d'arrêts. Elle garantit de trouver le coût le moins cher pour voyager de la ville source à la ville destination avec au maximum K arrêts, si une telle route existe.

1. **Voir code**
2. **Déterminez la complexité temporelle de votre solution**

La complexité temporelle de la solution dépend du nombre de villes (n), du nombre d'arêtes (m) et du nombre maximum d'arrêts autorisés (K).

Création du graphe : La construction du graphe prend un temps proportionnel au nombre d'arêtes, donc O(m).

Initialisation : L'initialisation de la file de priorité prend un temps constant, donc O(1).

Exploration : Dans le pire des cas, chaque ville peut être visitée K+1 fois (puisque nous autorisons au maximum K arrêts). À chaque itération, nous extrayons un nœud de la file de priorité et ajoutons ses voisins. L'extraction d'un nœud de la file de priorité prend un temps O(log n) et l'ajout de voisins prend un temps O(log n) pour chaque voisin. Dans le pire des cas, chaque ville est connectée à toutes les autres villes, donc il y a n-1 voisins pour chaque ville. Ainsi, la complexité temporelle de l'exploration est O((K+1) \* n \* (n-1) \* log n) = O(K \* n^2 \* log n).

Résultat : La vérification du résultat prend un temps constant, donc O(1).

En combinant toutes ces étapes, la complexité temporelle totale de la solution est O(m) + O(1) + O(K \* n^2 \* log n) + O(1), qui est dominée par le terme O(K \* n^2 \* log n). Ainsi, la complexité temporelle de la solution est O(K \* n^2 \* log n).

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**