TP 28 - Algorithme de Dijkstra
Appliqué au métro parisien

Exercice 1 (Algorithme de Dijkstra)

Le squelette de code fourni contient une implémentation des files de priorité, ainsi que la représentation du métro parisien sous forme de dictionnaire de listes d'adjacence.
Codez l'algorithme de Dijkstra vu en cours, et utilisez-le sur le graphe du métro parisien pour vérifier que la durée de trajet entre l'arrêt 89 et l'arrêt 253 est de 31 minutes et 58 secondes.

Exercice 2 (Chemin à emprunter)

Dans la vraie vie, il ne suffit pas de savoir quelle est la meilleure durée avec laquelle on peut aller d'un arrêt à un autre, mais on veut également savoir par quels arrêts intermédiaires il est nécessaire de passer. Dit en terme de graphes, parfois la distance entre deux sommets ne suffit pas, mais on veut l'intégralité d'un *chemin* de distance minimale.
Codez une fonction `chemin(depart, arrivee)` qui prend en entrée deux noms de stations `depart` et `arrivee` et renvoie la liste de tous les arrêts par lesquels il faut passer pour aller de l'un à l'autre le plus rapidement possible. Le résultat sera une liste de noms (et non pas de numéros), incluant l'arrêt de départ et l'arrêt d'arrivée.

Conseil. Copiez-collez puis modifiez votre code pour Dijkstra de telle sorte que :
- La file de priorité ne contiendra plus seulement des paires (u, d) où u est un sommet et d sa distance au sommet source depuis lequel Dijkstra a été lancé ; mais des triplets (u, v, d) où v est le sommet qui a permis d'enfiler ou de mettre à jour u.
- Quand on défile un sommet u, on renseigne dans une dictionnaire `predecesseur` le sommet v correspondant.

Il suffira alors, à la fin de l'exécution de l'algorithme, de remonter depuis `arrivee` jusqu'à `depart` dans le dictionnaire `predecesseur` pour reconstruire un plus court chemin de `depart` à `arrivee`.

Exercice 3 (Complexité de cette version des files prio et de Dijkstra)

1. Quelle est la complexité des différentes fonctions fournies implémentant des files de priorité ?
2. Quelle est la complexité de votre fonction `dijkstra` ?
3. Quelle est la complexité de votre fonction `stations_intermediaires` ?
4. **Bonus, difficile** Arrivez-vous à concevoir une autre implémentation des files de priorité, ou de l'algorithme de Dijkstra, ayant une meilleure complexité ?

Exercice 4 (Bonus : Détection de changements de ligne)

Quand on cherche à aller d'un arrêt de métro à un autre, plus que savoir quels sont tous les arrêts intermédiaires, ce qui nous intéresse est de savoir à quels arrêts il faut changer de ligne.
Codez une fonction `itineraire(depart, arrivee)` qui prend en entrée deux *noms* d'arrêts, et affiche un texte en français qui explique comment se rendre de `depart` jusqu'à `arrivee` (En particulier, qui donne les noms d'arrêts auxquels il faut changer de ligne).

Pour aller plus loin

Ici, nous avons supposé que l'itinéraire le plus rapide est forcément le meilleur, mais dans la vraie vie différentes personnes peuvent vouloir minimiser des critères incompatibles : temps de trajet, nombre de correspondances, prix (si on parle de trajets de trains), etc...
Il n'y a donc pas *un* meilleur itinéraire, mais différents meilleurs itinéraires, incomparables entre eux, optimisant différents critères. Le sujet 2025 d'Option Informatique du concours Centrale Supélec pour la filière MP étudie ce problème. Bien sûr, ce sujet utilise certaines notions qu'on ne verra qu'en fin de deuxième année en Option Informatique, mais vous pouvez regarder certaines des définitions introduites dans le sujet (optimums de Pareto, leur façon d'implémenter des files de priorité, etc...), si vous êtes curieux·ses.