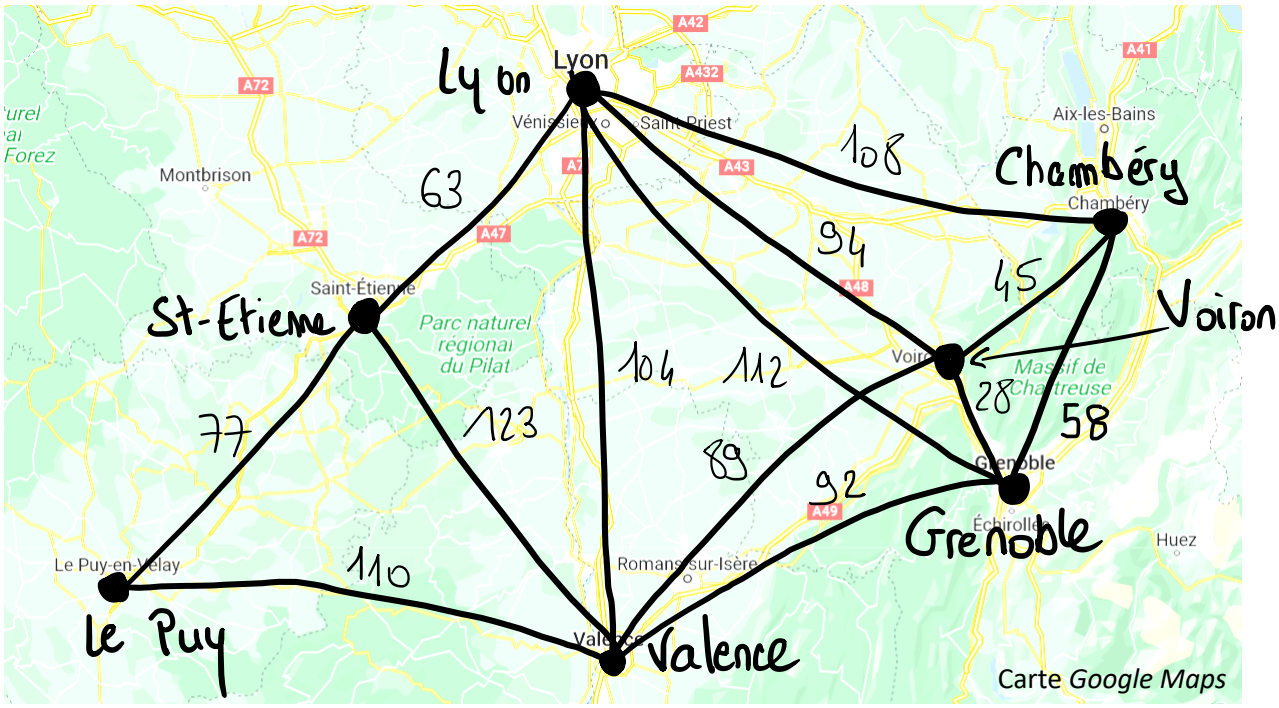
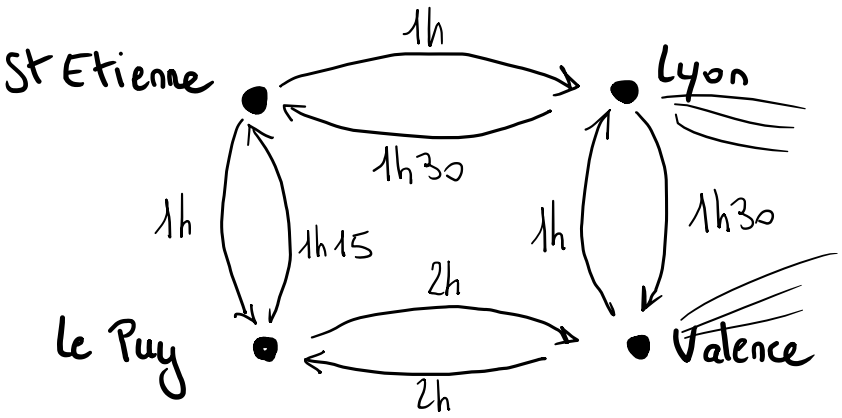
TD 23 – Partie 3 – Algorithme de Dijkstra – Recherche du plus court chemin dans un graphe pondéré

A / Contexte : GPS pour la recherche du meilleur itinéraire

Soit un graphe constitué de 7 sommets, qui représentent 7 villes de Rhône-Alpes. On décide de représenter une arrête entre deux sommets, non-orientée et pondérée par la distance par voie rapide entre deux villes. On remarque que le graphe est connexe.



Cependant, pour la navigation, les temps de transport pourraient aussi être intéressants, pour déterminer non pas le trajet le plus court mais le plus rapide. Le souci est qu'à cause des embouteillages et des travaux, les temps de trajet ne sont pas forcément similaires dans les deux temps de trajet, voire même parfois un itinéraire peut être momentanément inenvisageable car trop long… On décide donc de représenter la même carte mais avec un graphe orienté, où les arcs représentent les temps de transport. Étant donné le nombre d'arcs à représenter, voici une partie du graphe alors représentée, les temps de trajet ayant été mesurés sur *Google* à heure de pointe (18h en semaine).



Pour ces deux problèmes, nous avons un graphe **pondéré** avec des coefficients tous positifs (les temps, les distances entre deux sommets). Pour déterminer le plus court chemin à partir d'un sommet, on peut utiliser l'algorithme de *Dijkstra* (1959).

B / Deux fonctions auxiliaires

Il y a plusieurs façons d'écrire et d'implémenter l'algorithme de Dijkstra, mais c'est toujours un algorithme assez long en termes de lignes de code. Je vous invite fortement, dans ce genre de cas, à ***utiliser des fonctions auxiliaires***, auxquelles vous ferez appel dans l'algorithme principal.

B.1 / Tri des sommets par ordre décroissant de proximité au sommet initial

À chaque étape de l'algorithme de Dijkstra, on a vu qu'on sélectionne le sommet le plus proche du nœud dinitial `N_d` parmi ceux qu'il reste à traiter. En pratique, nous allons trier la liste L des sommets à traiter par ordre du plus lointain au plus proche (car, souvenez-vous, le parcours en largeur s'appuie sur une pile, et on va décider de sortir de la pile à droite – si on décidait de dépiler à gauche, on trierait par ordre croissant ici).

→ Etant donné que la liste change relativement peu entre deux itérations, mais qu'on va devoir re-trier à chaque itération, on opte pour un tri **par sélection**, qui a une complexité $O(N)$ quand il est appliqué à une liste presque déjà triée (*N* étant le nombre d'éléments dans la liste).

Soit D un dictionnaire des distances entre le nœud initial `N_d` et chacun des nœuds du graphe.

- **Mathématiquement** : D représente une application surjective de S (ensemble des sommets) vers $\mathbb{R}_+$
- **Informatiquement** : on décide ici de l'implémenter avec un dictionnaire, pour lequel pour tout sommet `S` du graphe (qui correspond à une clé), `D[S]` est la distance du chemin actuel entre `N_d` et `S`.

Q1 – Écrire une fonction `ind_insert(Noeud,L_triee, D)` prenant en argument un Nœud du graphe, une liste `L_triee` contenant certains nœuds du graphe, liste qui est déjà **triée par ordre décroissant de distance au nœud initial**, et `D` le dictionnaire des distances au nœud initial. Cette fonction renverra l'index (position) auquel insérer le sommet `Noeud` si on voulait l'insérer de sorte à maintenir l'ordre de la liste ainsi construite.

Q2 – Écrire alors une fonction `tri_insert_D(L, D)` qui réalise le **tri en place, par insertion** d'une liste de nœuds quelconque `L`, en la triant par ordre décroissant de distance au nœud initial, selon les informations contenues dans le dictionnaire des distances `D`.

Rappel : pour un tri **en place**, vous n'avez droit qu'à des permutations, du type `L[i],L[j] = L[j],L[i]` et la fonction sera procédurale (pas de **return**).

Test : Soit la liste des distances au sommet 'blanc' :

```
D = {'blanc': 0, 'bleu': 3, 'gris': 5, 'rouge': 6, 'jaune': 12}
```

Et les listes de sommets :

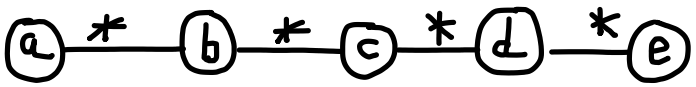
```
L1 = ['blanc', 'jaune', 'gris']
L2 = ['jaune', 'bleu', 'rouge', 'blanc']
```

Les appels ne doivent rien renvoyer, mais après les appels :

```
tri_insert_D(L1, D) → print(L1) → ['jaune', 'gris', 'blanc']
tri_insert_D(L2, D) → print(L2) → ['jaune', 'rouge', 'bleu', 'blanc']
```

B.2 / Notion de parent, et relâchement

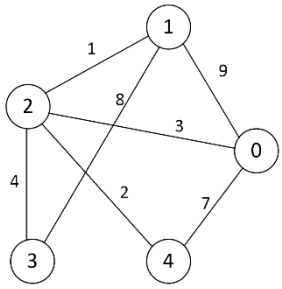
En pratique, au cours des itérations, en plus de vouloir tenir à jour le dictionnaire des distances `D`, nous voudrons également tenir à jour un second dictionnaire `P` qui associera à chaque sommet `S` du graphe : le sommet qui le précède dans le chemin le plus court qu'on ait trouvé.

Exemple : Soit un arbre 

→ Quelles que soient les valeurs des poids (∗), en fin d'algorithme on voudra que `P` associe à *e* : *d*, à *d* : *c*, à *c*, *b* et à *c* : *a*. Ainsi, on aura `P = { "a":None, "b":"a", "c":"b", "d":"c", "e":"d"}`

Le graphe est implémenté sous forme de dictionnaires de dictionnaires, par exemple :

```
Graphe = { 0:{1:9,2:3,4:7}, 1:{0:9,2:1,3:8}, 2:{0:3,1:1,3:4,4:2},
     3:{1:8,2:4}, 4 :{0:7,2:2} }
```



Soit un nœud de départ `N_d` et un de ses voisins `N_a`.

Q3 – Que faut-il écrire en *Python* pour qu'il renvoie la distance allant de `N_d` à `N_a` ?

On souhaite implémenter une fonction `relachement(Graphe,N_d,N_a,D,P)` prenant en argument

- un graphe
- deux dictionnaires `D` (des distances par rapport au sommet initial) et `P` (des parents)
- et un nœud de départ `N_d` ainsi qu'un nœud d'arrivée `N_a` pris parmi ses voisins.

Cette fonction **procédurale** effectuera le relâchement de l'arc (ou de l'arrête, selon que le graphe est orienté ou non), en mettant à jour `D` et `P` avec une portée globale. Le processus de relâchement est décrit en page 2.

Q4 – Écrire une fonction `relachement(Graphe,N_d,N_a,D,P)` décrite ci-dessus.

C / Algorithme de Dijkstra et détermination d'un meilleur chemin entre deux nœuds

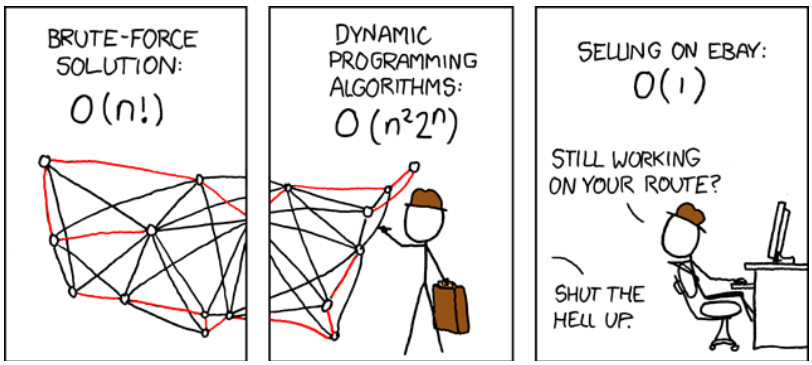
On rappelle qu'on a vu l'algorithme de **parcours en largeur** dans un graphe. L'algorithme de Dijkstra, comme les autres algorithmes de recherche de meilleur chemin dans un graphe, s'appuie sur ce parcours.

Rappel :

```
def parcours_largeur_ite(Graphe, N_d):
    A_visiter = [N_d]
    Deja_traites = []
    while A_visiter :
        Noeud_depart = A_visiter.pop()
        if Noeud_depart not in Deja_traites :
            Deja_traites.append(Noeud_depart)
            for Voisin in Graphe[Noeud_depart]: # attention, on avait un dictionnaire
                                    # de listes d'adjacences. Ici le graphe est
                                    implémenté sous
                                    # forme de dictionnaire de dictionnaire des
                                    distances…
                    A_visiter.insert(0,Voisin)
    return ...
```

Q5 – En vous inspirant fortement du code ci-dessus, et à l'aide des fonctions codées en **Q2** et **Q4**, écrire la fonction `Dijkstra(Graphe, N_d)` qui prenne en argument un Graphe et un sommet initial `N_d`, et qui renvoie les deux dictionnaires `D, P` (distances au sommet `N_d` et parent `P` – rappel `P` associe à chaque sommet S_i le dernier sommet S_j parcouru dans le meilleur chemin menant à ce sommet S_i).

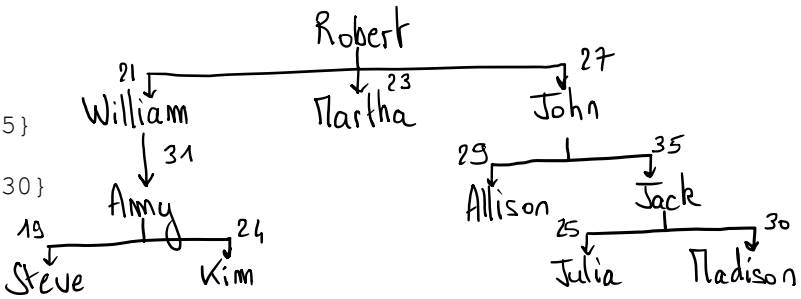
Q6 – En utilisant la fonction précédente, écrire une fonction `plus_court_chem(Graphe,N_d,N_a)` prenant en argument un graphe, un nœud initial (départ) `N_d` et un nœud d'arrivée `N_a`, et qui renvoie la liste des sommets parcourus successivement, menant de `N_d` à `N_a` en suivant le chemin le plus court.



D / Test sur le graphe décrit en début de sujet

Pour le test, on donne une variante pondérée du graphe vu dans le cours 2 sur les graphes :

```
Genealogie= { }   # Graphe orienté des âges auxquels chaque parent a eu chacun de ses enfants
Genealogie["Robert"]={"William":21, "Martha":23, "John":27}
Genealogie["William"]={"Amy":31}
Genealogie["Amy"]={"Steve":19,"Kim":24}
Genealogie["Steve"]={ }
Genealogie["Kim"]={ }
Genealogie["Martha"]={ }
Genealogie["John"]={"Allison":29,"Jack":35}
Genealogie["Allison"]={ }
Genealogie["Jack"]={"Julia":25,"Madison":30}
Genealogie["Julia"]={ }
Genealogie["Madison"]={ }
```



Par ailleurs, on donne les deux graphes correspondant à la description donnée en partie **A** :

```
Temps_traj = {}  # Graphe orienté des temps de transport (en h) entre 2 villes par voie rapide
Temps_traj["Lyon"]={"St-Etienne":1.5,"Valence":1.5,"Grenoble":1.5,"Voiron":1.25, "Chambéry":1.15}
Temps_traj["Voiron"]={"Lyon":1.25, "Grenoble":0.8, "Valence":1,"Chambéry":0.8}
Temps_traj["Chambéry"]={"Lyon":1.5, "Grenoble":0.5, "Voiron":0.75}
Temps_traj["St-Etienne"]={"Lyon":1,"Valence":1.5,"Le Puy":1}
Temps_traj["Valence"]={"Lyon":1,"St-Etienne":1.75,"Le Puy":2,"Grenoble":1.25, "Voiron":1}
Temps_traj["Le Puy"]={"St-Etienne":1.25,"Valence":2}
Temps_traj["Grenoble"]={"Lyon":2.25,"Valence":1,"Voiron":0.75, "Chambéry":0.75 }

km_traj = {}  # Graphe non orienté des distances entre 2 villes par voie rapide
km_traj["Lyon"]={"St-Etienne":63,"Valence":104,"Grenoble":112,"Voiron":94,"Chambéry":108}
km_traj["Voiron"]={"Lyon":94, "Grenoble":28, "Valence":89,"Chambéry":45}
km_traj["Chambéry"]={"Lyon":108, "Grenoble":58, "Voiron":45}
km_traj["St-Etienne"]={"Lyon":63,"Valence":123,"Le Puy":77}
km_traj["Valence"]={"Lyon":104,"St-Etienne":123,"Le Puy":110,"Grenoble":92,"Voiron":89}
km_traj["Le Puy"]={"St-Etienne":77,"Valence":110}
km_traj["Grenoble"]={"Lyon":112,"Valence":92, "Voiron":28, "Chambéry" :58}
```

Q7 – La commande `print(Dijkstra(Genealogie,"Robert")[0])` doit afficher les âges qu'avait Robert à la naissance de chacun des membres de sa famille, sous forme d'un dictionnaire `D` :

```
{'Robert': 0, 'William': 21, 'Amy': 52, 'Steve': 71, 'Kim': 76, 'Martha': 23, 'John': 27,
'Allison': 56, 'Jack': 62, 'Julia': 87, 'Madison': 92}
```

Q8 – La commande `print(Dijkstra(Genealogie,"Robert")[1])` doit afficher le nom du parent de chaque membre de la famille (sauf pour Robert où le parent est inconnu), sous la forme d'un dictionnaire `P` :

```
{'Robert': None, 'William': 'Robert', 'Amy': 'William', 'Steve': 'Amy', 'Kim': 'Amy', 'Martha':
'Robert', 'John': 'Robert', 'Allison': 'John', 'Jack': 'John', 'Julia': 'Jack', 'Madison': 'Jack'}
```

Q9 – L'itinéraire le plus court en temps, pour aller du Puy-en-Velay à Chambéry, et le trajet retour, sont :

Itinéraire aller : `print(plus_court_chem(Temps_traj,"Le Puy","Chambéry"))`

→ `['Le Puy', 'St-Etienne', 'Lyon', 'Chambéry']`

Itinéraire retour : `print(plus_court_chem(Temps_traj, "Chambéry","Le Puy"))`

→ `['Chambéry', 'Grenoble', 'Valence', 'Le Puy']`

… alors que le plus court en distance est : `print(plus_court_chem(km_traj,"Le Puy","Chambéry"))`

→ `['Le Puy', 'Valence', 'Voiron', 'Chambéry']`