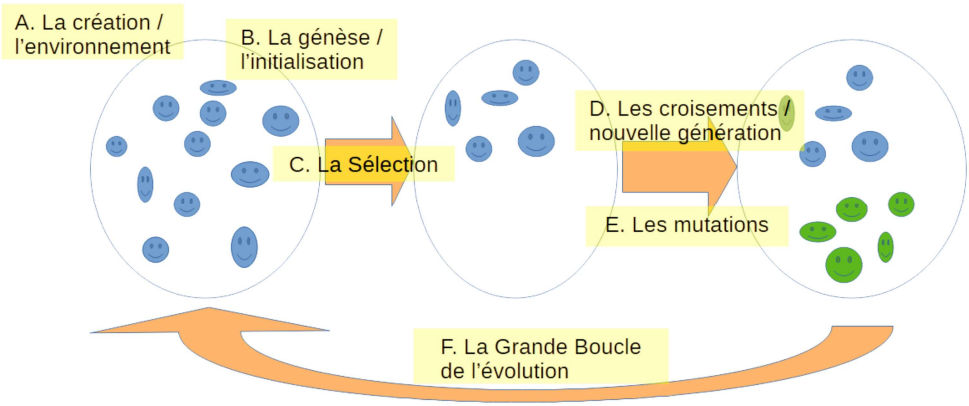
TP2 - Algorithme génétique pour le Problème du voyageur de commerce



On importe les librairies suivantes, que vous avez déjà croisées :

```
import numpy as np
import numpy.random as rd
```

Vous trouverez une aide complète sur la librairie `numpy.random` à l'adresse :
`docs.scipy.org/doc/numpy-1.14.0/reference/routines.random.html`
↪ adresse que vous pouvez retrouver parmi les premiers résultats en tapant "numpy.random" dans votre moteur de recherche.

On travaillera dans tout le TP avec des tableaux. On choisira de les coder au format `list` comme des listes de listes (= listes des lignes du tableau).

A. La "création" / l'environnement

L'algorithme s'applique sur un jeu de données :

- pour un "vrai problème" il est donné
- pour faire des tests, on peut créer des données aléatoirement

Pour savoir quels sont les meilleurs trajets pour le voyageur, on va avoir besoin de connaitre la distance entre les différentes villes de notre problème. Par exemple, si le voyageur doit aller à Brest, Quimper, Rennes et Saint Brieuc, on remplit un tableau de cette forme :

	Rennes	Brest	St Brieuc	Quimper
Rennes	0	210	95	180
Brest		0	126	53
St Brieuc			0	113
Quimper				0

Puis on rentre dans Python le tableau associé :

0	210	95	180
210	0	126	53
95	126	0	113
180	53	113	0

1. Rentrer dans Python le tableau ci-dessus, en l'appelant `Carte0`.

   Puis télécharger sur Moodle le tableau Python `Carte1` des distances entre 15 villes de France.

B. La "génèse" / initialisation aléatoire

On initialise l'algorithme avec un ensemble de N solutions admissibles créées aléatoirement.
Dit avec le vocabulaire génétique : on crée une population de départ constituée de N individus aléatoires.

Soit n le nombre de villes à parcourir, qu'on appellera dorénavant par leurs numéros de 0 à n-1. Un trajet peut alors être donné sous la forme d'une liste des nombres de 0 à n-1 dans l'ordre du parcours.
Par exemple, si on travaille avec 8 villes, numérotée de 0 à 7, un trajet possible serait la liste `[0,3,5,1,6,4,2,7]`

2. Créer une fonction `TrajetAlea` qui prend en paramètre le nombre n de villes et renvoie un trajet aléatoire sous forme de liste.

   On conviendra que la ville 0 est la ville où habite le voyageur, et donc la ville de départ et d'arrivée. On peut donc choisir de la mettre au début et à la fin, ou bien choisir de la sous-entendre.

   Utiliser l'aide de la librairie `numpy.random` *à l'adresse donnée ci-dessus : une fonction de la librairie permet de faire la fonction en 1 ou 2 lignes...*

3. Créer une fonction `PopAlea` qui prend en paramètres le nombre n de villes et la taille p de la population qu'on veut générer, et renvoie un tableau à p lignes dont chaque ligne est un trajet aléatoire.

   Tester vos fonctions.

C. La "sélection naturelle" / sélection des meilleures solutions

Parmis la population de départ, on sélectionne la "meilleure moitié". C'est-à-dire qu'on calcule la valeur de la fonction-coût pour chacun des N individus de la population, et on garde les $N/2$ individus pour lesquelles la valeur est plus petite.

Pour sélectionner les meilleurs trajets créés lors de la génèse, on a d'abord besoin de calculer la longueur des trajets fabriqués.

4. Créer une fonction `LTrajet` qui prend en paramètres une liste t correspondant à un trajet, ainsi qu'un tableau `Carte`, et renvoie la longueur du trajet. N'oubliez pas que le voyageur doit rentrer chez lui à la fin du voyage.
5. Vérifier qu'un voyageur habitant à Rennes et allant à Saint Brieuc puis Brest et Quimper (et rentrant chez lui) parcourt 454km.
6. Créer une fonction `LPop` qui prend en paramètre un tableau P contenant une population de trajets, ainsi qu'un tableau `Carte`, et renvoie un tableau contenant les longueurs de chacun des trajets (dans l'ordre du tableau P).
7. Tester votre fonction avec `Carte0` et `Carte1`, pour des trajets créés par votre fonction `PopAlea`.

Ensuite, on va imposer que le nombre p de trajets créés lors de la génèse soit un nombre pair. Le but de la question suivante est de ne garder que $p/2$ trajets, en choisissant les $p/2$ trajets les plus courts : par exemple, si on a créé une population de $p = 20$ trajets lors de la génèse, on gardera 10 trajets à l'issue de la sélection.

8. Créer une fonction `Selection` qui prend en paramètres un tableau P contenant une population de trajets, ainsi qu'un tableau `Carte`, et renvoie un tableau $P1$ contenant la moitié des trajets de P ayant les longueurs les plus courtes.

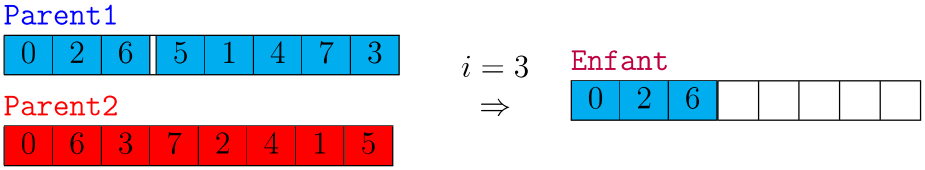
   On pourra utiliser les fonctions numpy `np.sort` *et/ou* `np.argsort`.

D. Les croisements

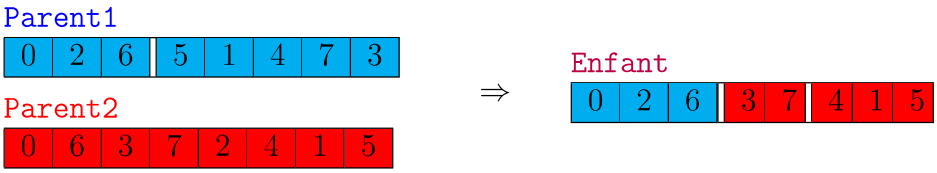
A partir des $N/2$ individus sélectionnés à l'étape précédente, on crée une "descendance" de $N/2$ nouveaux individus.
Pour cela, avec des "couples" d'individus sélectionnés (les "parents"), on crée un nouvel individu (l'"enfant") en mélangeant les caractéristiques des 2 "parents". On crée ainsi $N/2$ "enfants".

Il n'y a pas de méthode générale pour réaliser les croisements, on peut faire des choix très différents... plus ou moins efficaces. Voici le croisement qu'on se propose de coder ici.
On va créer un "enfant" en faisant un croisement à partir de 2 "parents" `Parent1` et `Parent2` de la manière suivante (+ compréhensible avec les schémas en couleur, sur Moodle) :

- La liste `Enfant` commencera comme la liste `Parent1`. Précisément, on choisit aléatoirement un indice i entre 0 et n-1, et les i premières villes de la liste `Enfant` seront les i premières villes de `Parent1`.



- Ensuite, on ajoute les villes de `Parent2` qui ne sont pas encore dans `Enfant`, dans l'ordre dans lequel elles apparaissent dans `Parent2`.



9. Créer une fonction `Croisement`, qui prend en paramètres 2 trajets (parents), et renvoie 1 trajet (enfant) par croisement des trajets parents de la manière décrite ci-dessus. L'entier i de coupure sera choisi aléatoirement à l'intérieur de la fonction.

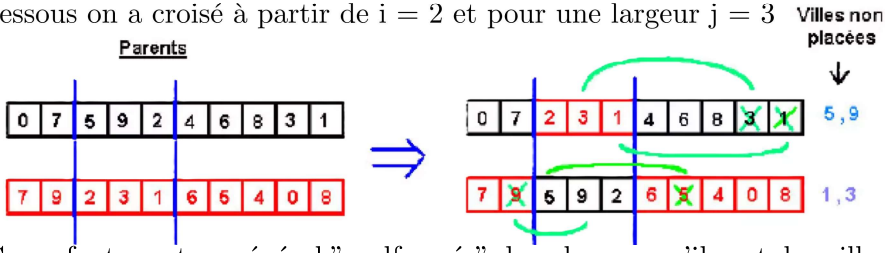
   Tester votre fonction avec les parents ci-dessus, en forçant i à valoir 3, puis avec un i aléatoire.

10. Créer une fonction `PopCroisement` qui prend en paramètre un tableau `P` contenant un nombre pair de trajets (parents), et renvoie un tableau contenant autant de trajets (enfants), obtenus par croisements des trajets (parents) entrés en paramètres.

    Pour cela, vous regrouperez aléatoirement en "couples" de parents les trajets du tableau `P` rentré en paramètre. Pour chaque "couple", vous ferez 2 enfants avec la fonction `Croisement`, en échangeant l'ordre des 2 parents. **Tester votre fonction.**

11. *Question facultative / + difficile.* D'autres choix de croisements peuvent être faits. Ci-dessous, une 2e manière de créer un croisement entre 2 "parents" `Parent1` et `Parent2` (+ compréhensible avec les schémas en couleur, sur Moodle). La coder dans une 2e fonction `Croisement2`, qui prend 2 parents en paramètres, et renvoie cette fois-ci 2 enfants.

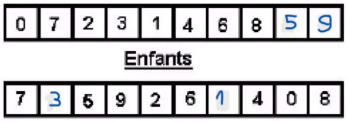
- On commence par prendre une partie de chaque parent, et par les intervertir. On obtient ainsi 2 enfants... "malformés". Dans l'exemple ci-dessous on a croisé à partir de i = 2 et pour une largeur j = 3



Ces enfants sont en général "malformés" dans le sens qu'ils ont des villes en double, et d'autres manquantes. Dans l'exemple ci-dessus le 1e enfant possède en double les villes 1 et 3, et il lui manque les villes 5 et 9.

- On convient alors de modifier les villes doublons **dans la partie principale** (la partie venant du parent 1 pour l'enfant 1, en noir sur le schéma ci-dessus). On remplace ces villes doublons par les villes manquantes (ici 5 et 9). On ajoute les villes manquantes dans **le même ordre que celui où elles apparaissent dans le parent principal** (le parent 1 pour l'enfant 1). Attention, on ne modifie donc pas la partie croisée.

  Pour l'exemple ci-dessus, on obtient les enfants :



E. Les mutations

On fait des "mutations" sur une certaine proportion de la population. C'est-à-dire qu'on choisit aléatoirement un certain pourcentage de la population, et qu'on applique de petites modifications aléatoires à ces individus.

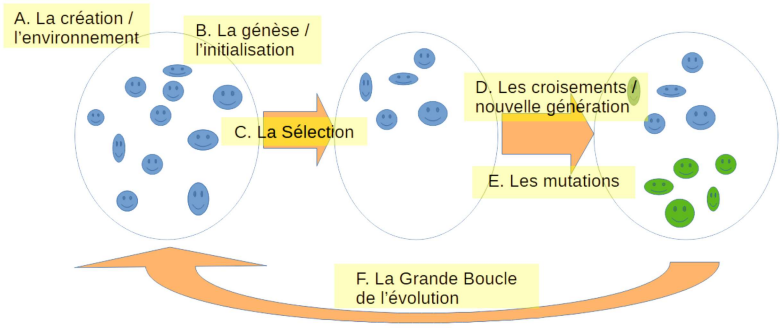
Dans l'évolution, l'ADN mute aussi aléatoirement suite à des facteurs environnementaux ou des erreurs de réplication.

9. Faire une fonction `Mutation` qui prend en paramètre un trajet, et renvoie ce trajet avec une modification : on se limitera au cas le plus simple, on permutera au hasard deux villes de l'individu en entrée (à l'exclusion de la ville 0 si elle apparait).
10. Faire une fonction `PopMutation` qui prend en paramètre un tableau contenant une population de trajets et un taux de mutation `r`, et renvoie un tableau contenant ces trajets après mutations d'une portion `r` de la population de départ.

    On arrondira la proportion pour avoir un nombre entier d'individus : par exemple, si on a en entrée une population de 10 individus, et un taux de mutation de 0.25, le nombre d'individus qui subiront une mutation devrait être de 0.25*10=2.5, on fera muter 2 individus. On choisira ensuite aléatoirement les 2 individus qui muteront parmis les 10 de la population.

F. La grande boucle de l'évolution

On a maintenant tout ce qu'il faut pour faire tourner un algorithme complet !



11. Créer une fonction `Genetique` qui prend en paramètres un tableau `Carte`, la taille p des populations à manipuler, le nombre g de générations souhaité, et le taux r de mutation, et renvoie le trajet le plus court obtenu après la g-ième itération.

G. Un bel affichage

Il peut être intéressant d'avoir un affichage qui permet de voir l'avancement de notre algorithme. Il y a plusieurs possibilités.

12. Un 1e affichage qui peut être intéressant est de renvoyer, à chaque itération, la longueur du trajet le plus court de la génération en cours.
13. Un affichage un peu plus complexe serait de représenter graphiquement les positions des villes, puis de représenter, à chaque itération, le trajet le plus court obtenu (en reliant les points), en indiquant en légende le numéro de la génération associée (et éventuellement la longueur de ce trajet).