

## TD ITC n°1

## Parrainages

La société secrète **FDMEDP** (Fans de Mathématiques et de Physique) fonctionne avec un système de parrainages : chaque membre peut avoir un-e parrain/marraine (ou ne pas en avoir), et peut avoir au maximum trois filleul-e-s.

La société contient  $n$  personnes, chaque individu étant représenté-e par un nombre entier entre 0 et  $n - 1$ . On représente la structure de parrainage par une liste `parrains` telle que

- `parrains[i]` contient  $i$  si l'individu  $i$  n'est pas parrainé ;
- `parrains[i]` contient le numéro de son parrain/sa marraine sinon.

Par exemple, la liste

```
[0, 0, 0, 4, 4, 2]
```

correspond à la structure ci-dessous :

```
-+-----+
 0   4
+---+ +
1  2  3
+
5
```

Dans tout l'énoncé,  $n$  désigne le nombre d'individus dans la société, la liste `parrains` est définie comme ci-dessus, et on note l'intervalle  $[0 : k]$  l'intervalle allant de 0 inclus à  $k$  exclus. On nomme « ancêtres de  $i$  » les individus dont descend l'individu  $i$ .

**1** – Préparation : récupérer le module `module_parrainages.py` sur cahier de prépa, puis ouvrir un **nouveau fichier python** dans le même dossier que le module. Dans ce nouveau fichier, écrire l'instruction

```
import module_parrainages as par
```

puis utiliser la console pour créer une société de test et l'afficher (ici pour 20 individus) :

```
In [1]: soc = par.créer_liste_parrains(20)
```

```
In [2]: par.afficher(soc)
```

On peut également récupérer une liste pré-générée, assez équilibrée pour les tests :

```
In [1]: soc = par.récupérer_liste_préalable()
```

```
In [2]: par.afficher(soc)
```

**2** – Écrire une fonction `nb_parrainés` qui prend en paramètre une liste `parrains` et renvoie le nombre d'individus qui sont parrainés par un autre. Estimer la complexité de cet algorithme.

**3** – Écrire une fonction `nb_filleuls` qui prend en paramètres une liste `parrains` et un entier `individu` compris dans  $[0 : n]$ , et renvoie le nombre de filleuls de `individu`.

**4** – Écrire une fonction `ancêtre` qui prend en paramètres une liste `parrains` et un entier `individu` compris dans  $[0 : n]$ , et renvoie le numéro de l'individu tout en haut de sa généalogie. Estimer la complexité dans le meilleur et le pire cas pour cet algorithme.

**5** – Écrire une fonction `longueur_chaine` qui prend en paramètres une liste `parrains` et un entier `individu` compris dans  $[0 : n]$ , et renvoie la longueur de la chaîne de parrainages qui le relie à son ancêtre.

**6** – Écrire une fonction `lister_filleuls` qui prend en paramètres une liste `parrains` et un entier `individu` compris dans  $[0 : n]$ , et renvoie une liste contenant les filleuls de `individu`.

**7** – Écrire une fonction `liste_filleuls_complète` qui prend en paramètre une liste `parrains`, et renvoie une liste contenant en case  $i$  la liste des filleuls de l'individu  $i$ .

**8** – Écrire une fonction `ligne_directe` qui prend en paramètres une liste `parrains` et deux entiers `jeune` et `vieux` compris dans  $[0 : n]$ , et renvoie `True` si `jeune` est un descendant de `vieux`, et `False` sinon.

**9** – Écrire une fonction `plus_proche_ancêtreCommun` qui prend en paramètres une liste `parrains` et deux entiers `i1` et `i2` compris dans  $[0 : n]$ , et renvoie le numéro de leur plus proche ancêtre commun ; s'ils ne sont pas apparentés, on renverra `-1`. On pourra s'aider de la fonction `ligne_directe`.