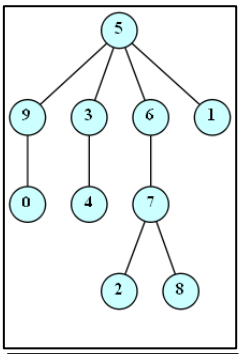
Avant de commencer : relire le notebook «Arbres représentés par un tableau des ancêtres»

Introduction : *On représente communément les arbres en mémoire en indiquant pour chaque nœud quels sont ses nœuds fils. C'est très pratique pour de nombreux algorithmes qui ont besoin que l'on accède aux fils d'un nœud (taille d'un arbre, hauteur d'un arbre, insertion et recherche dans un arbre binaire de recherche etc.).*

Il existe néanmoins des algorithmes qui ont besoin que l'on accède au père d'un nœud : dans ce cas la représentation de l'arbre à l'aide d'un tableau peut être pertinente. C'est le cas des algorithmes Union-Find.

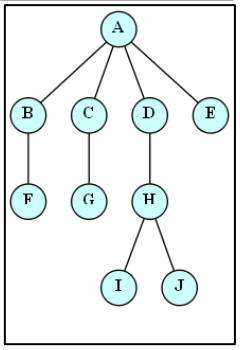
I : Représentation d'un arbre par un tableau ou dictionnaire des ancêtres



On rappelle qu'avec la représentation d'un arbre en tableau des ancêtres, l'arbre ci-contre est rerésenté par le tableau suivant :

```
arbre = [9, 5, 7, 5, 3, 5, 5, 6, 7, 5]
```

où l'on remarque que le père du nœud racine est indiqué comme étant lui-même (on aurait aussi pu choisir d'indiquer que le nœud racine avait None pour père).



Quant à l'arbre ci-contre on peut utiliser le même principe mais à l'aide d'un dictionnaire :

```
arbre = {'F':'B', 'B':'A', 'G':'C', 'C':'A', 'I':'H', 'H':'D', 'D':'A',
         'J':'H', 'E':'A', 'A':'A'}
```

Questions

Question 1 :

Déterminer une fonction python `pere` qui :
- prend en paramètres :
  - un tableau représentant un arbre des ancêtres (`list`),
  - un des nœuds (`int`);
- renvoie le père de ce nœud (`int`)

```
>>> t = [9, 5, 7, 5, 3, 5, 5, 6, 7, 5]
>>> pere(t, 3)
5
>>> pere(t, 5)
5
>>> pere(t, 2)
7
```

Question 2 :

Déterminer une fonction python `ancetres` qui :
- prend en paramètres :
  - un *tableau* représentant un arbre des ancêtres (`list`),
  - un des nœuds (`int`);
- renvoie un *tableau* contenant tous les ancêtres du nœud triés par ordre croissant d'ancienneté (d'abord le nœud, puis le père, puis le grand-père, puis l'arrière grand-père etc.)

```
>>> t = [9, 5, 7, 5, 3, 5, 5, 6, 7, 5]
>>> ancetres(t, 8)
[8, 7, 6, 5]
>>> ancetres(t, 7)
[7, 6, 5]
>>> ancetres(t, 0)
[0, 9, 5]
>>> ancetres(t, 5)
[5]
```

Question 3 :

On souhaite faire la même chose qu'aux questions 1 et 2 mais en prenant en paramètres un *dictionnaire* des ancêtres représentant un arbre et un des nœuds de l'arbre (comme ci-contre).

Y a-t-il un changement à apporter aux fonctions `pere` et `ancetres` ?

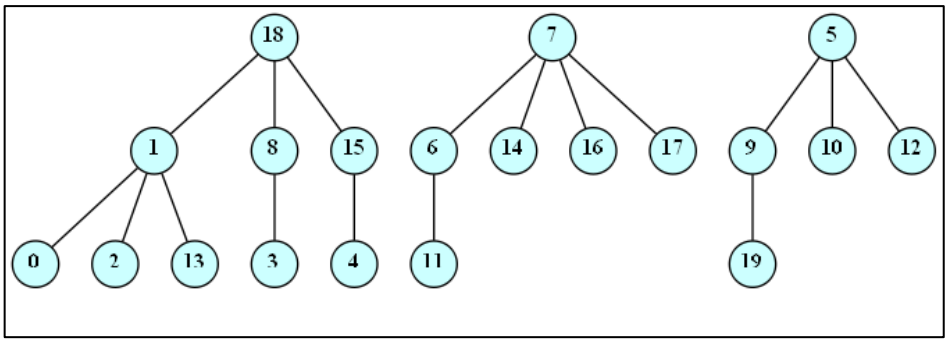
```
>>> d = {'F':'B', 'B':'A', 'G':'C', 'C':'A', 'I':'H',
         'H':'D', 'D':'A', 'J':'H', 'E':'A', 'A':'A'}
>>> pere(d, 'G')
'C'
>>> pere(d, 'A')
'A'
>>> ancetres(d, 'J')
['J', 'H', 'D', 'A']
>>> ancetres(d, 'A')
['A']
```

II : Représentation de plusieurs familles à l'aide de plusieurs arbres

Considérons le tableau ci-dessous :

```
[1, 18, 1, 8, 15, 5, 7, 7, 18, 5, 5, 6, 5, 1, 7, 18, 7, 7, 18, 9]
```

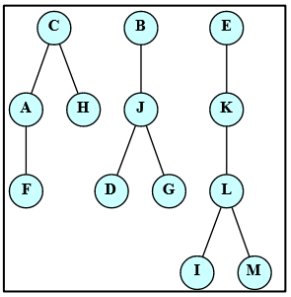
Si on reprend le principe des arbres représentés par des tableaux des ancêtres, on constate que ce tableau correspond à trois arbres, comme indiqué ci-contre.



Ce tableau correspond donc à trois familles différentes que l'on peut identifier par les indices de leur plus vieil ancêtre. On a ainsi la famille d'identifiant 18, la famille d'identifiant 7 et la famille d'identifiant 5.

Nous pouvons dire exactement la même chose avec le dictionnaire ci-dessous :

```
{'A': 'C', 'B': 'B', 'C': 'C', 'D': 'J', 'E': 'E', 'F': 'A', 'G': 'J',
'H': 'C', 'I': 'L', 'J': 'B', 'K': 'E', 'L': 'K', 'M': 'L'}
```



qui correspond aux trois familles d'identifiants C, B et E représentées ci-contre.

Remarque : pour la suite nous parlerons de «tableau des familles» ou de «dictionnaire des familles».

Questions

Question 0 :

On considère le code ci-contre.

Quelle est la valeur des variables `nb1` et `nb2` à la fin de l'exécution du code ? À quoi correspondent ces valeurs ?

```
>>> def mystere(tab):
        dico = {}
        for v in tab:
            dico[v] = True
        return len(dico)
>>> tab1 = [7, 6, 7, 7, 9, 6, 9, 7, 7]
>>> nb1 = mystere(tab1)
>>> tab2 = ['A', 'K', 'K', 'A', 'K', 'K']
>>> nb2 = mystere(tab2)
```

Question 1 :

Dans le tableau des familles ci-dessous, déterminer si les sommets 8 et 14 font partie de la même famille. Pour vous faciliter la tâche les indices multiples de 3 ont été notés sous le tableau.

```
[0, 8, 14, 7, 6, 19, 12, 15, 3, 18, 0, 18, 12, 18, 9, 15, 16, 0, 16, 12]
 ^          ^          ^          ^          ^          ^          ^
 0          3          6          9          12         15         18
```

Question 2 :

a) Déterminer un très court algorithme rédigé en français qui permet de savoir si deux sommets présents dans un tableau des familles font partie de la même famille.
Cet algorithme utilisera la fonction `ancetres(sommet)` telle qu'elle a été définie dans la partie I.

b) Cet algorithme fonctionne-t-il également avec un dictionnaire des familles ?

Question 3 :

Déterminer le nombre de familles – ainsi que leurs identifiants – présentes dans le tableau des familles ci-dessous. Pour vous faciliter la tâche les indices multiples de 3 ont été notés sous le tableau.

```
[8, 1, 17, 12, 16, 16, 14, 7, 18, 16, 12, 1, 1, 6, 7, 12, 7, 17, 19, 19]
 ^          ^          ^          ^          ^          ^          ^
 0          3          6          9          12         15         18
```

Question 4 :

a) Déterminer un court algorithme rédigé en français qui permet de compter le nombre de familles présentes dans un tableau des familles.
On pourra s'inspirer de la question 0 en comptant le nombre d'identifiants distincts des familles de chaque sommet.
Cet algorithme utilisera également la fonction `ancetres(sommet)` telle qu'elle a été définie dans la partie I.

b) Cet algorithme fonctionne-t-il également avec un dictionnaire des familles ?

Question 5 :

On dispose du tableau des familles ci-dessous qui comporte quatre familles.

```
[4, 1, 6, 13, 10, 6, 6, 1, 8, 2, 10, 7, 7, 8, 21, 18, 4, 3, 11, 13, 4, 15]
 ^         ^          ^         ^          ^          ^          ^          ^
 0         3          6         9          12         15         18         21
```

a) Vérifier que les sommets 20 et 18 ne font pas partie de la même famille.

b) Suite à une union (à un mariage), on apprend que les sommets 20 et 18 font désormais partie de la même famille. Modifier le tableau pour que le tableau *muté* corresponde à cette nouvelle situation. En particulier :
   - Le tableau *muté* ne comportera plus que trois familles,
   - Les liens de parenté au sein du tableau devront être modifiés, la seule chose qui importe c'est que :
     - les deux familles non concernées par l'union ne soient pas modifiées,
     - tous les nœuds des deux familles concernées par l'union devront désormais appartenir à la même famille, c'est-à-dire avoir le même identifiant, c'est-à-dire avoir le même plus vieil ancêtre. Cela nécessite donc que certains sommets changent de plus vieil ancêtre (puisqu'il ne restera qu'un seul plus vieil ancêtre pour ces deux familles).

On pourra faire un schéma des arbres concernés avant de commencer à modifier le tableau (la solution est très simple : ne cherchez pas une solution compliquée).

Question 6 :

a) Déterminer un très court algorithme rédigé en français qui, à partir d'un tableau des familles, permet de muter ce tableau des familles pour unir (marier) les familles de deux nœuds.
Cet algorithme pourra utiliser la fonction `ancetres(sommet)` telle qu'elle a été définie dans la partie I.

b) Cet algorithme fonctionne-t-il également avec un dictionnaire des familles ?

Implémentations en python

Pour les questions suivantes on effectuera l'implémentation en travaillant avec un dictionnaire des familles.
On notera que les implémentations de ces fonctions peuvent être très courtes (Q7 : 3 lignes de code, Q8 : 5 lignes de code, Q9 : 3 lignes de code)

Question 7 :

Déterminer une fonction python `meme_famille` qui implémente l'algorithme de la question 2.

Question 8 :

Déterminer une fonction python `nombre_familles` qui implémente l'algorithme de la question 4.

Question 9 :

Déterminer une fonction python `union_familles` qui implémente l'algorithme de la question 6.

Question 10 :

a) Plus les arbres des familles sont hauts, plus les algorithmes précédents sont lents. Expliquer pourquoi.

b) Réfléchir à une façon d'implémenter la fonction ancetres de la partie I de sorte qu'elle permette lors de son appel de diminuer la hauteur de certaines des branches des arbres en jeu.
Pour cela, on considèrera que la seule chose qui importe est que les familles soient respectées : on pourra donc modifier le père de certains sommets en le remplaçant par un sommet (bien choisi) de la même famille.

III : Application : analyse d'images

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 1, 1, 0, 0, 1, 0, 0],
 [0, 1, 1, 1, 0, 0, 1, 1, 0],
 [0, 1, 0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 1, 1, 0, 0, 0, 0],
 [0, 0, 1, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

On dispose d'une matrice correspondant à une image en noir et blanc (noir : 1 et blanc : 0).

On dit que deux pixels noirs sont de la même famille s'ils sont voisins (chaque pixel comporte quatre voisins : N, E, S, O).
On cherche à compter le nombre de familles présentes dans l'image, c'est-à-dire à compter le nombre de «tâches» noires présentes dans l'image (en théorie des graphes ou en topologie mathématique on parlerait de *composantes connexes*).

Par exemple, la matrice ci-contre en haut possède trois tâches noires, alors que celle-ci-contre en bas en possède sept.

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0],
 [0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0],
 [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0],
 [0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
 [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

On cherche à programmer une fonction permettant de compter les tâches noires d'une image.

Un pixel sera représenté par un 2-uplet `(lig, col)`.

1) Programmer une fonction `mat_2_dico` qui prend en paramètre une matrice comme celles-ci-dessus et renvoie un dictionnaire des familles où chaque pixel noir constitue une famille à lui tout seul (les pixels blancs ne figurent pas dans le dictionnaire).
   (6 lignes de code)

2) Programmer une fonction `mat_2_familles` qui prend en paramètre une matrice comme celles-ci-dessus et renvoie un dictionnaire des familles où tous les pixels noirs d'une même tâche noire constituent une seule famille. Pour cela on pourra partir d'un dictionnaire renvoyé par `mat_2_dico` et effectuer des unions de familles pour tous les pixels noirs avec leurs voisins.
   Bien évidemment on utilisera avec profit la fonction union_familles de la partie II.
   *(11 lignes de code dont 4 * 2 = 8 lignes avec 4 instructions conditionnelles* `if … :`*)*

3) Programmer une fonction `nombre_de_taches` qui prend en paramètre une matrice comme celles-ci-dessus et renvoie le nombre de tâches noires présentes dans l'image.
   Bien évidemment on utilisera avec profit la fonction nombre_familles de la partie II.
   (1 ligne de code)

Remarque 1 : pour faciliter l'implémentation, on suppose que les pixels situés au bord de la matrice sont blancs. Cela permet d'effectuer un parcours de la matrice en excluant les bords : en faisant un tel parcours, on est certain que les voisins existent bien, ce qui évitera les erreurs "Index out of range" ou l'utilisation de tests pour vérifier l'existence des voisins (tests qui alourdissent le code).

Dernière question : Si les clefs du dictionnaire sont des couples (lig, col) de coordonnées de pixels, peut-on prendre des objets de type list – par exemple [6, 8] – pour représenter ces coordonnées ?