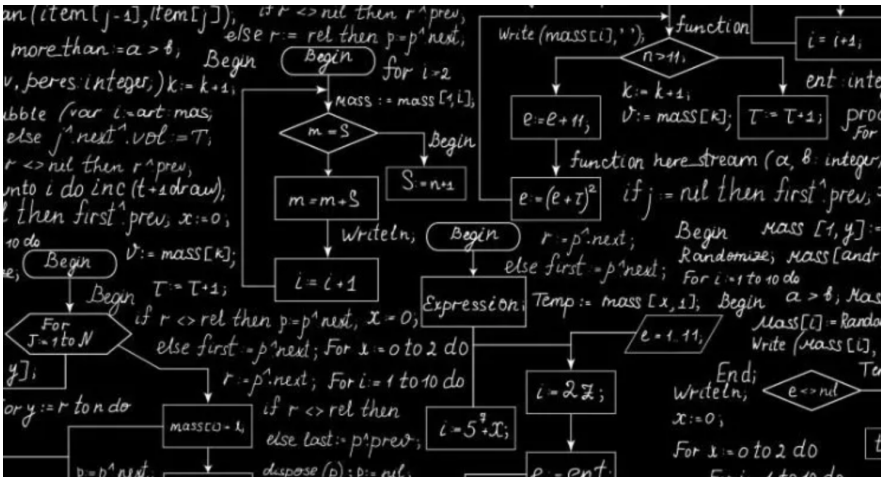
TP INFO 10 | Éléments d'algorithmique

Ce document rassemble quelques exercices et problèmes d'algorithmiques posés au concours.



1. Conseils de rédaction

Dans les exercices qui suivent, nous allons aborder des aspects plus théoriques de l'informatique. Il conviendra de respecter les consignes suivantes :

⇛ Tout algorithme non trivial doit être expliqué brièvement. Quelle est la stratégie globale de résolution ? Quels rôles les différentes variables jouent-elles ? L'idéal est de proposer un invariant de boucle afin d'éclairer le lecteur.

⇛ Il faut justifier avec soin les complexités, même lorsqu'elles sont évidentes : par exemple, écrire que le coût d'une boucle est linéaire en n car elle comporte n itérations est insuffisant, il convient d'estimer le coût d'une itération. De plus, il ne sert à rien de détailler à l'extrême vos calculs (le fait qu'une itération coûte 2, 4 ou 5 opérations n'est pas important, c'est l'ordre de grandeur qui compte : on se borne à mentionner un coût constant).

2. Exercices

1 — *Génération des codes binaires de longueur fixée f*

L'objectif est de générer la liste de tous les codes binaires de longueur $n \in \mathbb{N}^*$. Par exemple, pour $n = 3$, on obtient les codes suivants

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

qui sont les écritures binaires de tous les entiers de 0 à $2^3 - 1$.

1. Il n'est pas difficile de construire le code de l'entier $k+1$ à partir de celui de k. L'algorithme est celui que nous utilisons pour additionner en base dix, on propage une « retenue » de droite à gauche :

$$\begin{array}{rr} & 1\,1\,1\,1\,1 \\ & 1\,0\,1\,0\,1\,1\,1\,1\,1 \\ + & \qquad\qquad 1 \\ \hline = & 1\,0\,1\,1\,0\,0\,0\,0\,0 \end{array}$$

On voit que cela revient à trouver le « 0 » de poids minimal, le remplacer par « 1 » puis remplacer les « 1 » de poids inférieur par des « 0 ». En déduire une fonction `suivant` prenant en argument une liste binaire `t` de longueur n distincte de `[1]*n` et renvoyant le code binaire qui suit `t`.

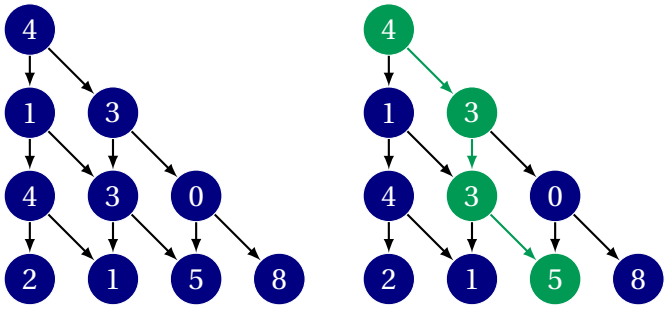
2. En déduire une fonction `gen` prenant en argument un entier n non nul et renvoyant la liste de tous les codes binaires de longueur n.

3. Quelle est la complexité de la fonction `gen` ?

2 — *Un problème d'optimisation* ff

On considère des données entières structurées sous la forme d'un triangle (cf. ci-contre). En partant du sommet, on se déplace jusqu'à la dernière ligne en choisissant à chaque itération la direction sud ou sud-est.

Le poids de ce chemin est la somme des entiers rencontrés. Par exemple, le chemin figuré en vert a un poids de 15.



L'objectif est de déterminer le poids maximal d'un chemin. Les données seront codées par une liste de listes représentant les différentes lignes. Notre exemple sera donc représenté par :

```
[ [4], [1,3], [4,3,0], [2,1,5,8] ]
```

On notera n le nombre de lignes du triangle et $t_{i,j}$ ses coefficients.

1. On se propose de calculer le poids maximal par *force brute*, i.e. en itérant sur tous les chemins possibles.
   - **a.** Expliquer comment représenter un chemin par un code binaire à $n-1$ bits.
   - **b.** Écrire une fonction `cout` prenant en arguments un triangle `t`, un chemin `c` et qui renvoie son poids.
   - **c.** En utilisant l'exercice précédent, écrire une fonction `pm_brute` prenant en argument un triangle `t` et renvoyant le poids maximal d'un chemin du triangle.
   - **d.** Quelle est la complexité de votre fonction ?
2. Nous allons optimiser le calcul du poids maximal. Pour tout $i \in [\![0, n-1]\!]$, on note $\ell_{i,j}$ la longueur maximale d'un chemin dont l'origine est le coefficient $t_{i,j}$ (ligne i et colonne j du triangle).
   - **a.** Que vaut $\ell_{n-1,j}$ pour $j \in [\![0, n-1]\!]$?
   - **b.** Démontrer que, pour tout $i \in [\![0, n-2]\!]$ et $j \in [\![0, i]\!]$, $\ell_{i,j} = t_{i,j} + \max\left(\ell_{i+1,j}, \ell_{i+1,j+1}\right)$.
3. En déduire une fonction récursive `pm_rec` prenant en arguments la liste `t`, les indices i, j et renvoyant $\ell_{i,j}$. Comment obtient-on le chemin de coût maximal au moyen de cette fonction ?
4. Proposer une approche non récursive pour le calcul des $\ell_{i,j}$. Écrire une fonction `pm_ite` analogue à la précédente et déterminer sa complexité.

3 — *Élément majoritaire d'une liste en sept versions* ff – fff

Un élément `x` d'une liste `t` est dit majoritaire dans `t` s'il apparaît dans cette liste un nombre de fois strictement supérieur à $n/2$, où n est la longueur de la liste. Il est clair qu'une liste admet au plus un élément majoritaire. Considérons par exemple, les listes :

`t_1=[5,1,5,4,5,7,5,3,5]` et `t_2=[1,4,4,5,7,2,3,5]`

La liste `t_1` admet un élément majoritaire, l'entier `5`. En revanche, la liste `t_2` n'admet aucun terme majoritaire.

1. Une première solution consiste à compter le nombre d'occurrences des termes de la liste jusqu'à obtenir la réponse.

```
def elt_majo_1(t):
    n=len(t)
    for i in range(n):
        c=0
        for j in range(n):
            if t[j]==t[i]:
                c+=1
        if 2*c>n:
            return t[i]
    return None
```

a. Déterminer la complexité de la fonction `elt_majo_1`.

b. Quelle structure vous semble la plus adaptée pour dénombrer les termes d'une liste ?

c. En déduire une fonction `elt_majo_2` prenant en argument une liste `t` et renvoyant son élément majoritaire en cas d'existence et `None` sinon.

d. Quelle est la complexité de cette nouvelle fonction ?

2. Dans cette question, nous allons étudier le cas d'une liste `t` triée.

a. Proposer une autre méthode de calcul de l'élément majoritaire qui exploite le caractère trié de la liste.

b. En déduire une fonction `elt_majo_triee` prenant en argument une liste `t` triée dans l'ordre croissant et renvoyant son élément majoritaire en cas d'existence et `None` sinon.

c. Quelle sa complexité ?

d. Donner le principe d'une fonction `elt_maj_3` prenant en argument une liste `t` et renvoyant son élément majoritaire en cas d'existence et `None` sinon. Comparer sa complexité à celles des fonctions `elt_maj_1` et `elt_maj_2`.

3. On se propose de réaliser un nouvel algorithme de résolution. Le point de départ est de coder une boucle `for` sur un indice `j` variant de 0 à $n-1$ (où n est la longueur de la liste) dont la propriété suivante

$$\begin{cases} \text{La liste } \texttt{t[:i]} \text{ ne contient aucun élément majoritaire} \\ \text{Le terme } \texttt{t[i]} \text{ est majoritaire dans } \texttt{t[i:j+1]} \\ \text{Le terme } \texttt{t[i]} \text{ apparaît } \texttt{c} \text{ fois dans } \texttt{t[i:j+1]} \end{cases}$$

est un invariant. À chaque itération, on met à jour l'indice `i` et le compteur `c` de la façon suivante :

- ⇛ Si `t[j]` est égal à `t[i]`, alors on incrémente `c`.
- ⇛ Sinon, dans le cas où `t[i]` n'est plus majoritaire dans `t[i:j+1]`, on affecte `i` à `n-1` ou `j+1` selon que la liste est épuisée ou non puis on réinitialise `c`.

a. Considérons une liste `L` contenant un élément majoritaire `x` qui n'est pas majoritaire dans une sous-liste de la forme `L[:i]`. Justifier que `x` est majoritaire dans `L[i:]`.

b. On suppose qu'une liste `L` admet un élément majoritaire `x`. Montrer que si on ajoute un élément à `L`, la nouvelle liste ne peut admettre un élément majoritaire différent de `x`.

c. Expliciter cette boucle avec son initialisation sous la forme d'un script python puis justifier l'invariant annoncé.

d. En déduire une fonction `elt_majo_4` prenant en argument une liste `t` et renvoyant son élément majoritaire en cas d'existence et `None` sinon.

e. Quelle est sa complexité ?

4. Un algorithme très simple a été publié en 1980 par Boyer et Moore.

- On utilise deux variables :
  - `candidat` : dont la valeur est le candidat actuel pour l'élément majoritaire
  - `compte` : qui est entière et initialisée à 0.
- On parcourt la liste de gauche à droite, et pour chaque valeur `v`
  - si `compte` est nulle,
    - on affecte candidat à la valeur `v`
    - on incrémente `compte`
  - sinon,
    - si `candidat` est égal à `v`, alors on incrémente `compte`
    - sinon, on décrémente `compte`

Si la liste admet un élément majoritaire, alors nécessairement il est égal à la valeur de `candidat`.

a. En déduire une fonction `elt_majo_5` ayant les mêmes caractéristiques que les précédentes.

b. Évaluer sa complexité.

c. Démontrer la correction de l'algorithme de Boyer-Moore.

5. Proposer une fonction `elt_majo_6` fondée sur un principe récursif et la remarque suivante que l'on justifiera : si une liste contient un élément majoritaire, alors celui-ci est nécessairement majoritaire dans au moins de une deux moitiés de la liste.

6. En utilisant l'algorithme de Quick-select (cf. le TP dédié aux tris), proposer une nouvelle fonction `elt_majo_7` ayant les mêmes caractéristiques que les précédentes.

4 — *Longueur d'une plus grande sous-liste croissante* ***fff***

On appelle sous-liste d'une liste `t` toute liste de la forme

`[t[n0],t[n1],...,t[nk]]` où $0 \leqslant$ `n0` $<$ `n1` $< \cdots <$ `nk` $<$ `n` où `n` est la longueur de `t`

Elle est dite croissante si de plus `t[n0]` $\leqslant$ `t[n1]` $\leqslant \cdots \leqslant$ `t[nk]`. Par convention, une liste de longueur 1 est croissante. On s'intéresse dans le sujet à la longueur maximale d'une sous-liste croissante de `t`. Pour `[11,6,2,24,25,12,21,41,34,30]`, celle longueur maximale vaut 4 : par exemple, `[2,24,25,41]` réalise ce maximum. On propose un algorithme efficace, fondé sur la programmation dynamique, pour calculer la longueur d'une plus grande sous-liste croissante.

Notons `t` la liste donnée en argument. On calcule un nouvelle liste `M` de la manière suivante : s'il n'existe pas d'élément inférieur ou égal à `t[i]` avant `t[i]`, alors `M[i]=1`, sinon `M[i]` vaut 1 plus le maximum des `M[j]`, pour `j` < `i` tel que `t[j]` $\leqslant$ `t[i]`.

1. Que représente `M[i]` en termes de sous-listes croissantes de `t` ?
2. Comment trouver la longueur d'une plus grande sous-liste croissante de `t` en utilisant `M` ?
3. Écrire une fonction `longMaxBis(t)` renvoyant la longueur d'une plus grande sous-liste croissante de `t`.
4. Quelle est la complexité de la fonction `longMaxBis` ?
5. En vous inspirant de la fonction précédente, écrire une fonction `seqMax(t)` renvoyant une sous-liste de `t`, croissante et de longueur maximale.

3. Problèmes

5 — *Structure d'Union-Find et application à un réseau social* ***f–ff***

Partie I – Réseaux sociaux

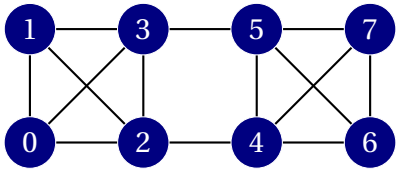
Nous supposerons que les individus sont numérotés de 0 à $n-1$ où n est le nombre total d'individus. Nous représenterons chaque lien d'amitié entre deux individus i et j par une liste contenant leurs deux numéros dans un ordre quelconque, c.-à -d. par la liste `[i,j]` ou par la liste `[j,i]` indifféremment.

Un réseau social R entre n individus sera représenté par une liste `reseau` à deux éléments où :

⇛ `reseau[0]` est le nombre d'individus appartenant au réseau ;

⇛ `reseau[1]` la liste non-ordonnée (et potentiellement vide) des liens d'amitié déclarés entre les individus.

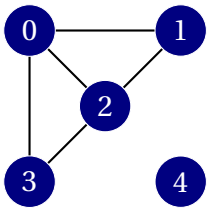
La figure suivante donne un réseau social sous la forme d'un graphe relationnel : chaque lien d'amitié entre deux personnes est représenté par une arête entre elles.



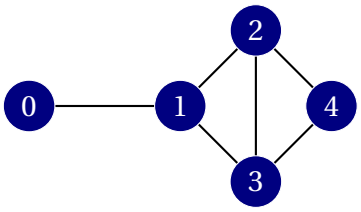
Une représentation possible de ce réseau sous la forme d'une liste au format [Nombre d'individus, liste des liens d'amitié] :

```
[8,[[0,1],[1,3],[3,2],[2,0],[0,3],[2,1],[4,5],[5,7],[7,6],[6,4],[7,4],[6,5],[2,4],[5,3]]]
```

1. Donner une représentation sous forme de listes pour chacun des deux réseaux sociaux ci-dessous :



Réseau A



Réseau B

2. Écrire une fonction `creerReseauVide` prenant en argument un entier n et qui crée, initialise et renvoie la représentation sous forme de liste du réseau à n individus n'ayant aucun lien d'amitié déclaré entre eux.
3. Écrire une fonction `estUnLienEntre` prenant en argument une liste `paire` à deux termes, et `i`, `j` deux entiers, et qui renvoie `True` si les deux éléments contenus dans paire sont i et j dans un ordre quelconque et renvoie `False` sinon.
4. Écrire une fonction `sontAmis` prenant en arguments un réseau `res`, deux entiers `i`, `j` et qui renvoie `True` s'il existe un lien d'amitié entre les individus i et j dans `reseau` et renvoie `False` sinon. Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?
5. Écrire une fonction `declareAmis` de paramètres `reseau`, `i`, `j` qui modifie `reseau` pour y ajouter le lien d'amitié entre les individus `i` et `j` si ce lien n'y figure pas déjà. Quelle est la complexité en temps de votre procédure dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?
6. Écrire une fonction `listeDesAmisDe` prenant en paramètres `reseau` et `i` qui renvoie la liste des amis de `i` dans `reseau`. Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?

Partie II – Structure d'Union-Find

Une partition en k groupes d'un ensemble A à n éléments consiste en k sous-ensembles disjoints non-vides $A_1, \ldots, A_k$ de A dont l'union est A, c.-à -d. tels que $A_1 \cup \cdots \cup A_k = A$ et pour tout $i \neq j, A_i \cap A_j = \varnothing$. Par exemple $A_1 = \{1,3\}, A_2 = \{0,4,5\}, A_3 = \{2\}$ est une partition en trois groupes de $A = [\![0,5]\!]$. Dans cette partie, nous implémentons une structure de données très efficace pour coder des partitions de $[\![0, n-1]\!]$.

Le principe de cette structure de données est que les éléments de chaque groupe sont structurés par une relation filiale : chaque élément a un (unique) parent choisi dans le groupe et l'unique élément du groupe qui est son propre parent est appelé le représentant du groupe. On s'assure par construction que chaque élément i du groupe a bien pour ancêtre le représentant du groupe, c.-à -d. que le représentant du groupe est bien le parent du parent du parent etc. (autant de fois que nécessaire) du parent de l'élément i. La relation filiale est symbolisée par une flèche allant de l'enfant au parent dans la figure 2 qui présente un exemple de cette structure de données.

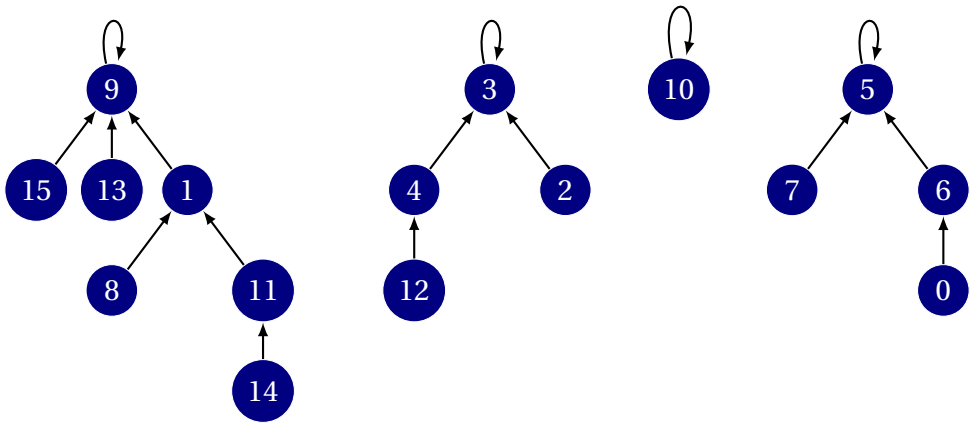


Figure 2 : une représentation filiale de la partition suivante de $[\![0,15]\!]$ en quatre groupes : $\{1,8,9,11,13,14,15\}$, $\{2,3,4,12\}$, $\{10\}$ et $\{0,5,6,7\}$ dont les représentants respectifs sont 9,3, 10 et 5.

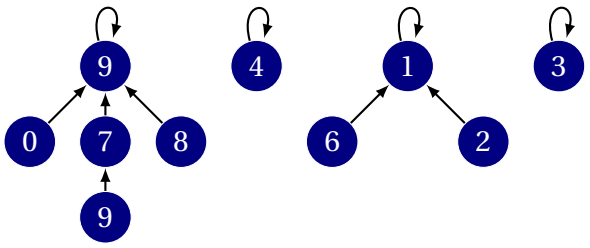
Dans l'exemple de cette figure, 14 a pour parent 11 qui a pour parent 1 qui a pour parent 9 qui est son propre parent. Ainsi, 9 est le représentant du groupe auquel appartiennent $14, 11, 1$ et 9 . Notons que ce groupe contient également 8, 13 et 15.

On notera que la représentation n'est pas unique (si l'on choisit un autre représentant pour un groupe et une autre relation filiale, on aura une autre représentation du groupe).

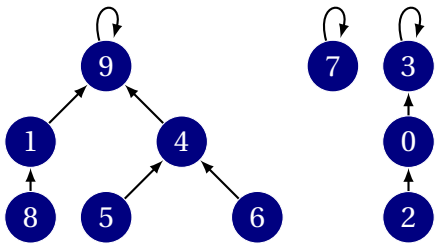
Pour coder cette structure, on utilise un liste `parent` à n éléments où `parent[i]` contient le numéro du parent de `i`. Par exemple, les valeurs du liste parent encodant la représentation filiale donnée dans la figure 2 sont :

`i`	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`parent[i]`	6	9	3	3	3	5	5	5	1	9	10	1	4	9	11	9

1. Donner les valeurs de la liste `parent` encodant les représentations filiales des deux partitions de $[\![0,9]\!]$ ci-dessous, et préciser les représentants de chaque groupe :



Représentation filiale A



Représentation filiale B

Initialement, chaque élément de $[\![0, n-1]\!]$ est son propre représentant et la partition initiale consiste en n groupes contenant chacun un individu. Ainsi, initialement, `parent[i]` vaut `i` pour tout $i \in [\![0, n-1]\!]$.

2. Écrire une fonction `creerPartitionEnSingletons` d'argument n qui renvoie un liste `parent` à n éléments dont les valeurs sont initialisées de sorte à encoder la partition de $[\![0, n-1]\!]$ en n groupes d'un seul élément.

Nous sommes intéressés par deux opérations sur les partitions :

⇛ Déterminer si deux éléments appartiennent au même groupe dans la partition.

⇛ Fusionner deux groupes pour n'en faire plus qu'un. Par exemple, la fusion des groupes $A_1 = \{1,3\}$ et $A_3 = \{2\}$ dans la partition de $[\![0,5]\!]$ donnée en exemple au tout début de cette partie donnera la partition en deux groupes $A_2 = \{0,4,5\}$ et A_4 où $A_4 = A_1 \cup A_3 = \{1,2,3\}$.

3. Écrire une fonction `representant` d'argument `parent` et `i` qui renvoie l'indice du représentant du groupe auquel appartient i dans la partition encodée par le liste `parent`. Quelle est la complexité dans le pire cas de votre fonction en fonction du nombre total n d'éléments ? Donnez un exemple de liste parent à n éléments qui atteigne cette complexité dans le pire cas.

Pour réaliser la fusion de deux groupes désignés par l'un de leurs éléments i et j respectivement, on applique l'algorithme suivant :

⇛ Calculer les représentants p et q des deux groupes contenant i et j respectivement.

⇛ Affecter `parent[p]` à la valeur `q`.

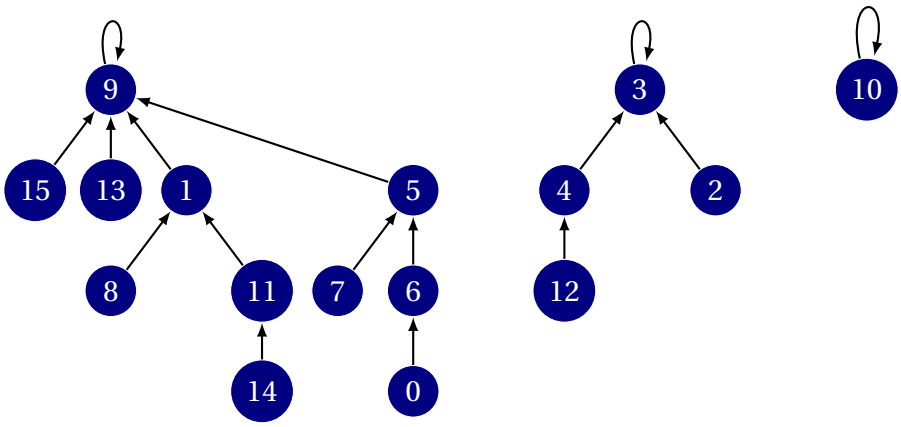


Figure 3 : structure filiale obtenue après la fusion des groupes contenant 6 et 14 de la figure 2.

4. Écrire une fonction `fusion` d'arguments `parent`, `i`, `j` qui modifie `parent` pour fusionner les deux groupes contenant `i` et `j` respectivement.

Pour l'instant, la structure de données n'est pas très efficace comme le montre la question suivante.

5. Proposer une suite de $n-1$ fusions dont l'exécution à partir de la partition en n singletons de $[\![0, n-1]\!]$, nécessite de l'ordre de n^2 opérations élémentaires.

Pour remédier à cette mauvaise performance, une astuce consiste à compresser la relation filiale après chaque appel `representant(parent,i)`. L'opération de compression consiste à faire la chose suivante : si p est le résultat de l'appel à la fonction `representant(parent,i)`, modifier `parent` de faqn à ce que chaque ancêtre de i (y compris i) ait pour parent direct p. Noter bien que même si un appel `representant(parent,i)` renvoie le représentant de i elle modifie également `parent`. Si l'on reprend l'exemple de la figure 2, le résultat de l'appel `representant(parent,14)` est 9, que l'on a calculé en remontant les ancêtres successifs de 14 : 11, 1 puis 9. L'opération de compression consiste alors à donner la valeur 9 aux cellules d'indices 14, 11, et 1 de la liste `parent`.

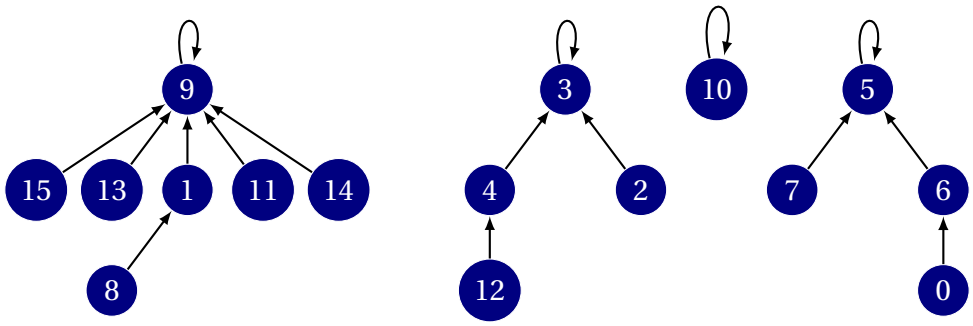


Figure 4 :structure filiale obtenue après l'opération de compression menée depuis 14.

6. Modifier votre fonction `representant` pour qu'elle modifie `parent` pour faire pointer directement tous les ancêtres de i vers le représentant de i une fois qu'il a été trouvé. En quoi cette optimisation de la structure filiale peut-elle être considérée comme « gratuite » du point de vue de la complexité ?

Afin d'afficher de manière lisible la partition codée par une liste parent, on souhaite calculer à partir de la liste parent la liste des listes des éléments des différents groupes. Une sortie possible pour la liste des groupes correspondant à la figure 2 serait :

```
[[15,8,1,9,11,13,14],[4,3,2,12],[7,5,6,0],[10]]
```

7. Écrire une fonction `listeDesGroupes` d'argument `parent` qui renvoie la liste des différents groupes codés par `parent` sous la forme d'une liste des listes des éléments des différents groupes.

Partie III – Algorithme randomisé pour la coupe minimum

Nous allons à présent trouver une partition des individus d'un réseau social en deux groupes qui minimise le nombre de liens d'amitiés entre les deux groupes.

Pour résoudre ce problème nous allons utiliser l'algorithme randomisé suivant pour un réseau social à n individus :

- ⇛ Créer une partition P en n singletons de $[\![0, n-1]\!]$;
- ⇛ Initialement aucun lien d'amitié n'est marqué ;
- ⇛ Tant que la partition P contient au moins trois groupes et qu'il reste des liens d'amitié non-marqués dans le réseau faire :
  - ☭ Choisir un lien uniformément au hasard parmi les liens non-marqués du réseau, notons-le `[i,j]` ;
  - ☭ Si i et j n'appartiennent pas au même groupe dans la partition P, fusionner les deux groupes correspondants.
  - ☭ Marquer le lien `[i,j]`.
- ⇛ Si P contient $k \geqslant 3$ groupes, faire $k-1$ fusions pour obtenir deux groupes.

⇛ Renvoyer la partition P.

1. Écrire une fonction `coupeMinimumRandomisee` d'argument `reseau` qui renvoie la liste `parent` correspondant à la partition calculée par l'algorithme ci-dessus.

   INDICATION : Au lieu de marquer explicitement les liens déjà vus, on pourra avantageusement les positionner à la fin de la liste non-ordonnée des liens du réseau et ainsi pouvoir tirer simplement les liens au hasard parmi ceux placés au début de la liste.

   Quelle est la complexité de votre procédure en fonction de n, m et $\alpha(n)$, où m est le nombre de liens d'amitié déclarés dans le réseau et où $\alpha(n)$ désigne la complexité d'un appel à la fonction `representant` ?

2. Écrire une fonction `tailleCoupe` d'arguments `reseau` et `parent` qui renvoie le nombre de liens entre les différents groupes de la partition représentée par parent dans `reseau`.

Culture

⇛ On peut démontrer que cet algorithme renvoie une coupe de taille minimum avec une probabilité supérieure à $1/n$, ce qui fait que la meilleure parmi n exécutions indépendantes de cet algorithme est effectivement minimum avec probabilité supérieure à $1/e \approx 0.36787\ldots$.

⇛ La structure de données filiale avec compression pour les partitions est particulièrement efficace aussi bien en pratique qu'en théorie. En effet, la complexité de k opérations est de $O\big(k\alpha(k)\big)$ opérations élémentaires où $\alpha(k)$ est l'inverse de la fonction d'Ackermann, une fonction qui croit extrêmement lentement vers l'infini (par exemple $\alpha\big(10^{80}\big) = 5$).

6 — *Plus longue sous-liste équilibrée* ***ff–fff***

On considère une liste $s = (s_i)_{0 \leqslant i \leqslant n-1}$ ne contenant que deux valeurs : -1 et 1. Une sous-liste équilibrée de s est une liste d'éléments consécutifs de s où l'élément -1 et l'élément 1 apparaissent exactement le même nombre de fois.

Par exemple, si s est la liste $(-1,-1,1,-1,1,1,-1)$, alors les sous-listes $(-1,1,-1)$ et $(-1,1,-1,1)$ de s sont respectivement non-équilibrée et équilibrée.

L'objectif de ce sujet est de concevoir deux algorithmes de calcul de la longueur maximale d'une sous-liste équilibrée de s.

Par exemple, si s est la liste $(-1,-1,1,-1,1,1,-1)$, alors la longueur maximale des sous-listes équilibrées de s est 6 : $(-1,-1,1,-1,1,1)$ et $(-1,1,-1,1,1,-1)$ sont deux sous-listes équilibrées de s, de longueur 6.

Dans ce sujet, on considérera que la liste s est codée par une liste Python `t`, de longueur n.

Partie I – Une approche par « force brute »

Un premier algorithme consiste à passer en revue toutes les sous-listes de s en testant si elles sont équilibrées ou non.

1. Écrire une fonction `estEqui(i,j,t)` d'argument la liste `t` et deux indices i et j de la liste, tels que $0 \leqslant i < j < n$ où n est la longueur de `t`, et renvoie `True` si la sous-suite $(s_k)_{i \leqslant k \leqslant j}$ est équilibrée, `False` sinon.
2. Écrire une fonction `longMax(t,i)` d'argument la liste `t` et un indice i valide et renvoie la longueur de la plus grande sous-suite équilibrée de `t` commençant à l'indice i. Par convention, cette fonction doit renvoyer 0 s'il n'existe aucune sous-suite équilibrée commençant à l'indice i.
3. En déduire une fonction `longMaxSSE(t)` d'argument la liste `t` et qui renvoie la longueur maximale d'une sous-suite équilibrée de s.
4. Quel est la complexité de la fonction `longMaxSSE(t)` ?

Partie II – Un algorithme optimisé

On propose ici un algorithme plus efficace qu'à la partie I. Pour cela, on définit l'ordre lexicographique sur $\mathbb{R}^2$ par :

$$(x_1, y_1) \preccurlyeq (x_2, y_2) \iff (x_1 < x_2) \text{ ou } (x_1 = x_2 \text{ et } y_1 \leqslant y_2)$$

1. Écrire une fonction `cumul` d'argument la liste `t` et renvoyant `aux`, liste de longueur $n+1$ où n est celle de `t`, telle que `aux[k]` $= \left[\sum_{j=0}^{k-1} s_j, k-1\right]$ pour tout $k \in [\![0, n]\!]$, avec la convention $\sum_{j=0}^{-1} s_j = 0$.

   Par exemple, pour `t1=[1,1,-1,-1,1,-1,-1,1]`, l'appel de `cumul(t1)` doit renvoyer :

   ```
   aux1 = [ [0,-1], [1,0], [2,1], [1,2], [0,3], [1,4], [0,5], [-1,6], [0,7] ]
   ```

   On rappelle que `aux[i][j]` désigne l'élément d'indice j de `aux[i]`. Ainsi, sur l'exemple précédent, `aux[6]` est égal à la liste `[0,5]`, `aux[6][0]` et `aux[6][1]` valent respectivement 0 et 5.
2. Pour que $(s_k)_{i \leqslant k \leqslant j}$ soit équilibrée, quelle cns `aux[i][0]` et `aux[j+1][0]` doivent-ils vérifier ?
3. Dans cette question, on suppose donnée une fonction `tri` d'argument la liste `aux` et qui renvoie cette liste triée dans l'ordre croissant pour l'ordre lexicographique. Par exemple, pour la liste `aux1` de ci-dessus, la fonction `tri` doit renvoyer

   ```
   [ [-1,6], [0,-1], [0,3], [0,5], [0,7], [1,0], [1,2], [1,4], [2,1] ]
   ```

   En déduire une fonction `longMaxESSOpt(t)` d'argument la liste `t` renvoyant la longueur maximale des sous-suites équilibrées de s.

Partie III – Programmation de l'algorithme de tri et conclusion

On reprend les notations du préambule.

1. Justifier que $\left\{ \sum_{k=i}^{j} s_k \,;\, (i,j) \in [\![0, n-1]\!]^2 \right\} \subset [\![-n, n]\!]$.

2. Écrire une fonction `freq(n,s)` d'argument un entier naturel n et une liste `s`, dont chaque cellule est une liste numérique de longueur deux avec une première cellulle à valeurs dans $[\![-n, n]\!]$, et renvoyant la liste `f` de longueur $2n+1$ telle que, pour tout $k \in [\![0, 2n]\!]$, `f[k]` est égal à la liste croissante (éventuellement vide) des indices `i` tels que `s[i][0]=k-n`.

3. En déduire une version de la fonction `tri`, décrite à la partie II, de complexité linéaire.

4. Calculer la complexité de la fonction `longMaxESSOpt(t)`.

4. Indications

1

On peut écrire une fonction `suivant` de complexité linéaire.

2

On peut coder une descente verticale (resp. en diagonale) par 0 (resp. 1).

3

Au 1., l'instruction `x in t` a un coût linéaire car `t` est une liste. On peut optimiser via un dictionnaire.

4

On trouve une complexité quadratique au 4.

5

On pourra se demander pour certains algorithmes s'il plus avantageux d'itérer sur les individus ou sur leurs liens.

6

L'optimisation proposée permet de passer d'une complexité cubique à un coût linéaire.