

**Structure d'Union-Find** . Barème sur 22 pts, durée 1h15mn

Il s'agit de gérer efficacement **les partitions** de  $\llbracket n \rrbracket = \{0, 1, 2, \dots, n - 1\}$ . Une partition de  $E$  est une famille de parties de  $E$  non vides, deux à deux disjointes et dont la réunion est  $E$ . On les appelle **les classes** de la partition.

L'objectif est de construire une structure permettant d'effectuer les opérations suivantes :

- déterminer si deux éléments sont dans la même classe (*find*)
- modifier la relation pour fusionner les classes de deux éléments (*union*).

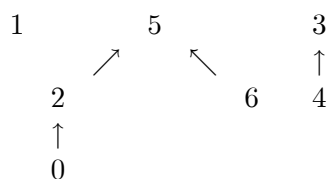
Une partition en  $k$  groupes d'un ensemble  $E = \llbracket n \rrbracket$  est une famille de  $k$  parties (appelées classes) de  $E$  disjointes non vides et dont l'union vaut  $E$ . Par exemple,  $\mathcal{P} = (\{1, 2, 5\}, \{0, 3\}, \{4\})$  est une partition de  $E_6$ .

Le principe de cette structure de données est de structurer les éléments de chaque groupe par une relation filiale (c'est-à-dire un arbre) : **chaque élément admet un unique parent choisi dans le groupe et l'unique élément du groupe qui est son propre parent est appelé le représentant du groupe (il correspond à la racine de l'arbre)**. Autrement dit, on représente la partition par une famille d'arbres (appelée forêt) dont les racines sont les représentants des classes.

**Partie I. Implémentation naïve**

Pour coder cette structure, on utilise un tableau `parent` de longueur  $n$  où la case `parent[i]` contient le numéro du parent de  $i$ .

Par exemple, `parent = [2, 1, 5, 3, 3, 5, 5]` représente la forêt :



Ainsi, le tableau `parent` code la partition de  $\llbracket 6 \rrbracket$  en les trois classes  $\{0, 2, 5, 6\}$ ,  $\{1\}$  et  $\{3, 4\}$ .

Les opérations fondamentales sur les partitions sont :

- FIND : Déterminer si deux éléments appartiennent à la même classe
- UNION : Fusionner deux classes pour n'en faire plus qu'une (ce qui modifie la partition).

1) [1 pt] Ecrire une fonction `creerPartitionEnSingletons(n)` qui crée et renvoie un tableau `parent` a  $n$  éléments codant la partition de  $\llbracket n \rrbracket$  en  $n$  singletons (= groupes à un seul élément).

2) [2.5 pts] Ecrire une fonction `representant(parent, i)` qui utilise le tableau `parent` pour trouver et renvoyer l'indice du représentant du groupe auquel appartient  $i$  dans la partition encodée par le tableau `parent`.

Quelle est la complexité dans le pire cas en fonction du nombre total  $n$  d'éléments ?

Donner sans justification un exemple de tableau `parent` qui atteint 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 :

*Etape 1* : Calculer les représentants  $p$  et  $q$  des deux groupes contenant  $i$  et  $j$  respectivement.

*Etape 2* : Faire `parent[p] = q`.

3) [2 pts] Ecrire une procédure `fusion(parent, i, j)` qui modifie le tableau `parent` pour fusionner les deux groupes contenant respectivement  $i$  et  $j$ .

4) [2 pts] Proposer sans justification une suite de  $(n - 1)$  fusions dont l'exécution a partir de la partition en  $n$  singletons de  $E$ , nécessite de l'ordre de  $n^2$  opérations élémentaires.

5) [2.5 pts] (★) Ecrire une fonction `listeDesGroupes(parent)` qui renvoie la liste des différents groupes codés par le tableau `parent` sous la forme d'une liste des classes, les éléments de chaque classe étant classés par ordre croissant.

Par exemple, `listeDesGroupes([2, 1, 2, 0, 1])` renvoie `[[0, 2, 3], [1, 4]]`.

## Partie II. Amélioration de la structure par équilibrage des arbres

La fusion de deux classes revient à fusionner deux arbres. L'idée est de fusionner ces deux arbres de sorte à minimiser la hauteur de l'arbre obtenu.

On considère donc un tableau supplémentaire `rang` qui donne **pour chaque représentant** la longueur maximale d'un chemin (autrement dit la hauteur de l'arbre représentant la classe).

*Exemple* : `parent = [2, 1, 5, 3, 3, 5, 5]` et `rang = [* , 0, * , 1, * , 2, *]` représentent la partition de  $\llbracket 6 \rrbracket$  en les classes  $\{0, 2, 5, 6\}$ ,  $\{1\}$  et  $\{3, 4\}$  dont les représentants sont 0, 3 et 5, et les valeurs du tableau `rang` en positions 0, 2, 4, 6 ne sont pas significatives (leurs valeurs n'ont pas de rôle dans l'algorithme).

6) [2 pts] Ecrire une nouvelle procédure `fusion(parent, rang, i, j)` qui fusionnent (si elles sont distinctes) les classes auxquelles appartiennent  $i$  et  $j$ , en prenant comme représentant de la réunion des deux classes celui des deux

représentants qui a le rang maximum. Lorsque les deux représentants ont le même rang, on choisit arbitrairement l'un des deux comme représentant de la réunion.

7) [2 pts] On appelle rang d'une classe le rang de son représentant.

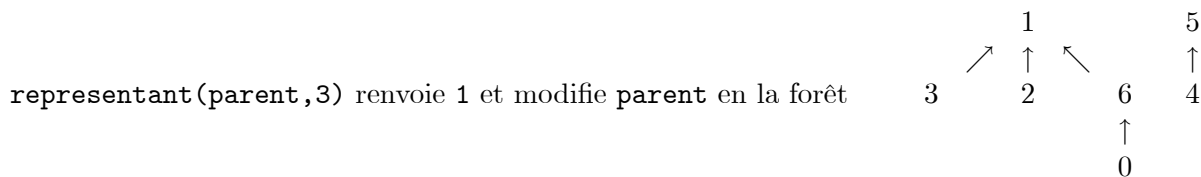
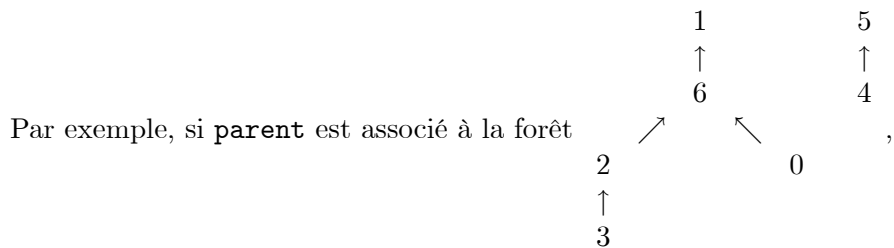
Montrer que `fusion` préserve l'invariant : < toute classe de rang  $k$  admet au moins  $2^k$  éléments >.

8) [2 pts] Considérons une partition de  $\llbracket n \rrbracket$  construite à partir de la partition en singletons et d'appels à la la fonction `fusion` précédente. Montrer qu'alors la fonction `FIND` permettant de déterminer le représentant d'un élément est de complexité  $O(\log n)$ .

### Partie III. Amélioration de la structure par compression des arbres

Pour remédier aux mauvaises performances de la structure précédente, on pourrait envisager une structure où le parent de chaque élément est le représentant de sa classe. Cette structure permettrait de résoudre `FIND` en temps  $O(1)$ , mais hélas, le cout de `UNION` resterait trop élevé, car il faudrait à chaque fusion modifier la totalité des parents de l'une des deux classes fusionnées.

Une astuce pour résoudre ces différents problèmes consiste à compresser la relation filiale après chaque appel a la fonction `representant(parent, i)`. L'opération de compression consiste a faire la chose suivante : si  $p$  est le résultat de l'appel à la fonction `representant(parent, i)`, cette fonction modifie **en même temps** le tableau `parent` de façon à ce que chaque ancêtre (c'est-à-dire le parent du parent . . . du parent) de  $i$ , dont  $i$  lui-même, ait pour parent direct  $p$ . Ainsi, un appel à `representant(parent, i)` renvoie le représentant de  $i$  et modifie également le tableau `parent`.



9) [2 pts] Modifier votre fonction `representant(parent, i)` pour qu'elle modifie le tableau `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 ?

**10)** [2 pts] (★) On reprend la fonction `listeDesGroupes(parent)` de la question 5).

Proposer une fonction de complexité linéaire  $O(n)$  et justifier la complexité obtenue.

#### Partie IV. Exemple d'utilisation

On donne ici un exemple d'utilisation (en fait lié à l'algorithme de Kruskal).

On se donne un graphe non-orienté  $G = (S, A)$ . On cherche un sous-ensemble d'arêtes  $B \subset A$  de cardinal maximal de sorte que le sous-graphe  $H = (S, B)$  soit acyclique, c'est-à-dire sans cycle (ce qui revient à dire que  $H$  est le graphe d'une forêt d'arbres disjoints).

Si tel est le cas, les composantes connexes de  $H$  sont les mêmes que celles de  $G$ , et  $H$  est la réunion d'arbres associés à chaque des composantes connexes de  $H$ . Il s'agit donc d'un algorithme permettant de déterminer les composantes connexes de  $G$ .

On suppose :

- les sommets de  $S$  numérotés de 0 à  $n - 1$ , donc on peut supposer  $S = \llbracket n \rrbracket$
- une arête  $(x, y)$  est stockée par le couple  $(x, y)$ , avec  $x < y$
- l'ensemble des arêtes de  $G$  est donné par la liste des couples  $(x, y) \in A$ .

On suppose que chaque arête  $(x, y) \in A$  n'apparaît qu'une fois dans cette liste.

On va utiliser UNION-FIND pour coder la partition associée à  $B$  (c'est-à-dire les composantes connexes de  $H$ ).

Par exemple, au départ, si  $B = \emptyset$ , la partition est composée des  $n$  singletons. La partition définie par les composantes connexes de  $B$  est comme précédemment codée par le tableau `parent`.

**11)** [1 pt] Ecrire une fonction `cycle(parent, a)` qui étant donné un tableau `parent` représentant la partition définie par  $B$  et une arête  $a \notin B$  renvoie `True` ssi le graphe  $(S, B \cup \{a\})$  admet un cycle.

**12)** [1 pt] Ecrire une fonction `foretCouvrante(n, A)` qui étant donnés un entier  $n$  et une liste  $A$  d'arêtes dans  $\llbracket n \rrbracket$ , renvoie la liste  $B$  souhaitée. On utilisera le tableau `parent` comme indiqué ci-dessus.

Préciser la complexité en fonction de  $n$  et  $m = \text{card } A$ , en supposant ici que les arbres obtenus ont tous une hauteur en  $O(\log n)$ .