UNIVERSITÉ DE NAMUR

Groupe 2

Algorithme II

Auteurs
Julien Castiaux
Denis Minne

Professeurs
M. James Ortiz
Pr. Pierre-Yves Schobbens

Diviser et conquérir

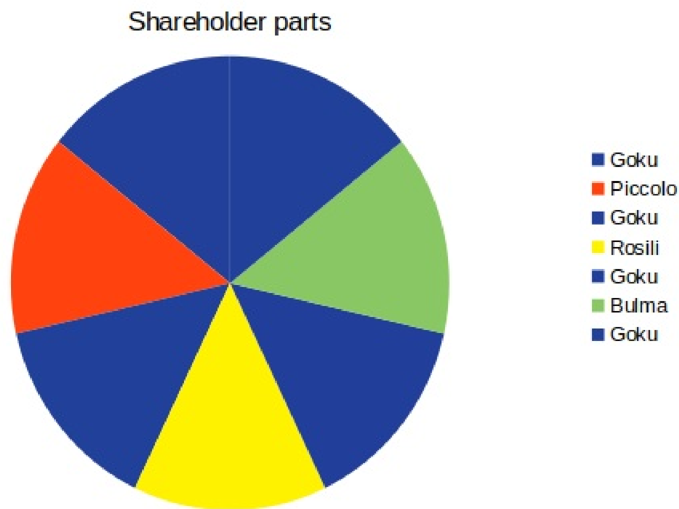
Analyse

> Analysez l'énoncé : détecter les ambiguïtés, les contradictions, etc,

Selon l'énoncé, un actionnaire est majoritaire s'il détient un plus grand nombre de parts que tous les autres réunis. Si aucun actionnaire ne respecte cette condition, il n'y a pas d'actionnaire majoritaire.

Dans les données d'entrée, chaque actionnaire est associé à une ou plusieurs parts et chaque part représente un même pourcentage de la valeur totale de sorte que la somme de toutes les parts égale la valeur totale. Les parts ne sont pas regroupées par actionnaire mais plutôt mélangées dans les données d'entrée.

Pour les données d'entrée suivantes: {Goku, Piccolo, Goku, Resili, Goku, Bulma, Goku}, on peut représenter le partage de la manière suivante:



Dans ce cas-ci l'actionnaire majoritaire est Goku car il détient 58% des parts là où les autres détiennent chacun 14%.

Spécification

```
/** Returns the name of the majority shareholder or null if there is none
  * @ normal_behavior
  * @ requires shareholders != null;
  * @ requires shareholders.length == 0;
  * @ ensures \result == null;
  * @ also
  * @ normal_behavior
  * @ requires shareholders != null;
  * @ requires shareholders.length > 0;
  * @ ensures \result != null ==> (
  * @   \sum int i; 0 <= i && i < shareholders.length && shareholders[i] == \result; 1
  * @ ) > shareholders.length / 2;
  * @ ensures \result == null ==> (
  * @   \forall int i; 0 <= i && i < shareholders.length; (
  * @     \sum int j; 0 <= j && j < shareholders.length && shareholders[j] == shareholders[i]; 1
  * @   ) <= shareholders.length / 2
  * @ );
  */
public static /* @ pure @ */ String majorityShareholder(String[] shareholders) {
```

i. Dans le cas où il n'y a pas d'actionnaire, il n'y a pas d'actionnaire majoritaire.

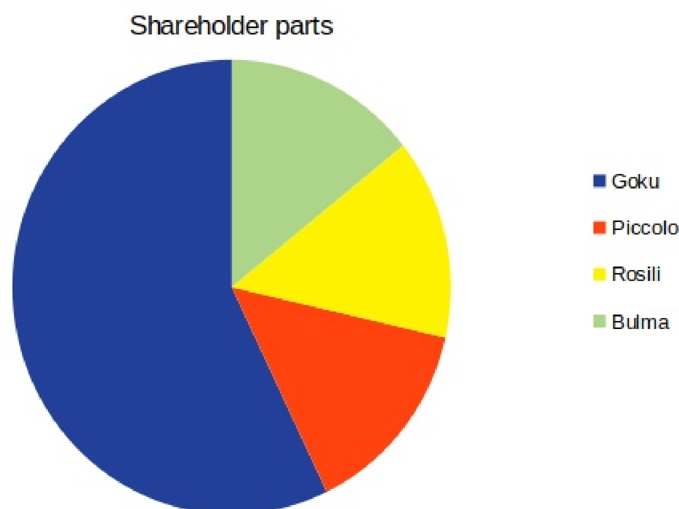
ii. Dans le cas où il y a un actionnaire majoritaire, cet actionnaire possède plus de 50% des parts.

iii. Dans le cas où il n'y a pas d'actionnaire majoritaire, chaque actionnaire possède au maximum 50% des parts.

Solution naïve

> Construisez une solution naïve permettant de trouver l'actionnaire majoritaire

La solution naïve consiste à regrouper les actionnaires par nom et de déterminer quelle section est la plus grande.



Invariant et variant

Le cas traité ci-dessous nécessite que la liste des actionnaires de `sParamTable` ait été regroupée. Il s'agit de l'algorithme permettant de déterminer qui possède la plus grosse part.

```
public static String findActioMaj(String [] psParamTable)
{
    int iMax = 0, iTmpValue = 0;
    String sResult = null, sTmpName = null;
    sTmpName = psParamTable[0];
    /**
      * @ loop_invariant 0 <= iCpt && iCpt <= psParamTable.length;
      * @ loop_invariant 0 <= iTmpValue && iTmpValue <= psParamTable.length;
      * @ loop_invariant 0 <= iCpt + 1 - iTmpValue;
      * @ loop_invariant (
      * @   \forall int i; iCpt - iTmpValue < i && i < iCpt; psParamTable[i] == sTmpName
      * @ );
      * @ loop_invariant \forall int i; 0 <= i && i < iCpt; iMax >= (
      * @   \sum int j; 0 <= j && j < iCpt && psParamTable[i] == psParamTable[j]; 1
      * @ );
      * @ loop_invariant iMax == (
      * @   \sum int i; 0 <= i && i < iCpt && psParamTable[i] == sResult; 1
      * @ );
      * @ decrease psParamTable.length - iCpt;
      */
    for(int iCpt = 0; iCpt < psParamTable.length; iCpt ++) //>
```

- `iMax` est la taille de la plus grande section courante
- `sResult` est le nom du plus grand actionnaire courant
- `iTmpValue` est la taille de la section actuelle
- `sTmpName` est le nom de l'actionnaire actuel

L'algo est un algorithme de calcul du maximum modifié, au lieu de donner l'élément le plus grand, il donne le nombre d'actions de la plus longue série d'éléments du même actionnaire. On obtiendra au final l'actionnaire ayant le plus grand nombre d'actions, reste à vérifier que ce nombre d'actions est majoritaire (+ de 50%).

Complexité de la solution naïve

> Donnez l'ordre de grandeur du temps et de la place mémoire de cet algorithme naïf,

La solution naïve repose sur deux sous-algorithmes distincts: un algorithme de tri et un algorithme qui calcule la plus longue série d'éléments identiques. Les deux algorithmes sont utilisés l'un après l'autre.

Ordre de grandeur :

- L'algorithme de tri a une boucle qui traite l'ensemble des actionnaires imbriquée dans une seconde boucle qui traite également l'ensemble des actionnaires.
- L'algorithme de la plus grande série fait un simple passage sur les actionnaires regroupés.

La complexité nous donne donc `O(n²) + O(n)` ce qui revient à **$O(n^2)$**

Espace requis:

- L'algorithme de tri nécessite une copie de la liste des actionnaires de départ. La taille est donc de `O(2n)` où n est le nombre de parts.
- L'algorithme de la plus grande série travaille directement sur les données d'entrée. La taille est donc de `O(n)`.

L'espace requis par cet algorithme est donc `max(2n, n)` ce qui donne **2x la taille des données de départ**.

Solution optimisée

> Construisez une solution plus efficace basée sur le principe "diviser pour régner"

Comme nous le constatons, trouver quelle est la plus longue série est une approche qui permet de résoudre le problème avec un ordre de grandeur O(n) et sans espace supplémentaire. Mais cet algorithme nécessite de trier les données à priori ce qui fait perdre du temps de calcul et nécessite de la mémoire supplémentaire. Utiliser un autre algorithme de tri pourrait résoudre ce problème. Il serait possible de réduire l'ordre de grandeur total en utilisant un autre algorithme de tri qui a un ordre de grandeur en O(n log_2_n) comme *Merge sort*, *Quick sort* ou *Heap sort*.

Une autre solution est de ne pas déterminer le plus grand actionnaire en passant par le tri et la longueur des séries mais de réaliser un dénombrement des parts de chaque actionnaire.

Le dénombrement peut se faire au moyen d'une map. On associe le nom de chaque actionnaire au nombre de parts qu'il possède. Pour déterminer si un actionnaire est majoritaire, il suffit alors de vérifier que le plus grand actionnaire possède plus de la moitié des parts.

Dénombrer les actionnaires est très simple :

- Dénombrer 1 actionnaire revient à incrémenter le compteur de cet actionnaire de 1 dans la map servant au dénombrement.
- Dénombrer les autres revient à faire le dénombrement des autres actionnaires.

Le *diviser pour régner* a la forme "La 1-tête et la (n-1)-queue)"

Spécification

Les spécifications sont identiques à la solution naïve.

Algorithme optimisé

> Ecrivez un algorithme basé sur le principe du diviser pour régner pour trouver l'actionnaire majoritaire dans S,

L'algorithme ci-dessous fait le dénombrement des parts pour chaque actionnaire. Les algorithmes qui suivent (plus grand actionnaire et actionnaire majoritaire) sont sur github, fichier `Pools.java`

```
/** Internal recurcive counter.
  *
  * @ public static model HashMap<String, Integer> count_spec(
  * @     String[] people, int index, HashMap<String, Integer> counter);
  * @ normal_behavior
  * @ requires people != null;
  * @ requires counter != null;
  * @ requires counter instanceof HashMap<String, Integer>;
  * @ requires head_index == people.length;
  * @ ensures \result isinstanceof HashMap<String, Integer>;
  * @ ensures \forall String person; (\result).containsKey(person); (
  * @   (\result).get(person).intValue() == (
  * @       \sum int i; 0 <= i && i < people.length && people[i] == person; 1
  * @   )
  * @ );
  * @ also
  * @ normal_behavior
  * @ requires people != null;
  * @ requires counter != null;
  * @ requires 0 <= head_index && head_index < people.length;
  * @ measured_by people.length - head_index;
  * @ ensures \forall String person; (\result).containsKey(person); (
  * @   (\result).get(person).intValue() <= (
  * @       \sum int i; 0 <= i && i < people.length && people[i] == person; 1
  * @   )
  * @ );
  * @ ensures \result == count_spec(people, index + 1, counter);
  */
private static /* @ pure @ */ HashMap<String, Integer> count(
    String[] people, int head_index, HashMap<String, Integer> counter) {
    if (head_index == people.length)
        return counter;
    counter.put(people[head_index], counter.getOrDefault(people[head_index], 0) + 1);
    return count(people, head_index + 1, counter);
}
```

Complexité de la solution optimisée

La solution optimisée se base sur trois sous-algorithmes effectués l'un après l'autre. Le premier va faire le dénombrement des parts des actionnaires, le second déterminer le plus grand actionnaire et le dernier vérifie que le plus grand actionnaire est majoritaire.

Ordre de grandeur

- Faire le dénombrement se fait de manière linéaire. L'ordre de grandeur est donc O(n).
- Trouver la plus grande paire (actionnaire, nombre de part) dans le dénombrement se fait également de manière linéaire. L'ordre de grandeur est donc O(n).
- Vérifier que le plus grand actionnaire est majoritaire vérifie simplement que ses parts couvrent plus de la moitié de l'ensemble des parts.

L'ordre de grandeur final de l'algorithme est donc `O(n) + O(n) + O(1)` ce qui donne **O(n)**

Espace requis

- Faire le dénombrement nécessite une structure de données supplémentaire. Cette structure sera aussi grande que le nombre d'actionnaires distincts. Dans le pire cas où il n'y a que des actionnaires différents, la taille de la map sera égale à la taille de la structure d'entrée.

- Trouver l’actionnaire ayant la plus grande part travaille directement sur les données d’entrée sans structure (mis à part quelques variables primitives) supplémentaire.
- Vérifier que le plus grand actionnaire est majoritaire nécessite de connaître le nombre de parts total. Vu que l’ensemble des parts sont partagées par l’ensemble des actionnaires, le nombre de parts total est simplement la taille du tableau des actionnaires qui est une donnée existante.

L’espace maximum requis est donc `max(2n, n, 0)` ce qui donne **2n**

Programmation dynamique

Analyse

> Analysez l’énoncé : détecter les ambiguïtés, les contradictions, etc.

Selon l’énoncé, le bûcheron se trouve sur la première ligne côté gauche indice (0,0) et se déplace vers la droite (0,1), (0,2) ou vers le bas(1,0), (2,0), etc.
S’il se déplace vers le bas, il change de direction. On peut donc en déduire que:

- Si la hauteur est un nombre pair, on se déplace vers la droite, c’est-à-dire avec une valeur croissante. A l’inverse, si la hauteur est un nombre impair, on se déplace vers la gauche, c’est-à-dire avec une valeur décroissante.
- Il ne peut jamais remonter
- Pour chaque case, il n’existe au maximum que 2 solutions possibles : on peut se déplacer d’une case vers la gauche ou vers la droite selon la « hauteur » à laquelle on se trouve, ou descendre d’une case (et changer de sens). Ces solutions ne sont possibles que si le contenu de leur case respective le permet.
- Si une case propose les 2 solutions, il faut tester les 2 « chemins » et choisir le meilleur. Ces 2 chemins risquent de se « rencontrer » plus tard dans la résolution. On se retrouvera donc à effectuer plusieurs fois la même opération.
- Si on arrive dans une cellule en passant par plusieurs chemins, le résultat obtenu dans cette cellule sera toujours identique.

Spécification

> Spécifiez le problème en utilisant JML (pré- et post-conditions).

En annexe

Implémentation naïve

> Construisez une solution naïve permettant de trouver le nombre maximal d’arbres que le bûcheron peut couper.

La solution naïve va exploiter une récursion, la technique se fait selon le calcul suivant : on traite la valeur (x, y) et il reste à traiter toutes les valeurs accessibles depuis cette position. Comme expliqué précédemment, les passages dans une cellule peut se faire par plusieurs chemins différents, la solution naïve ne va pas s’en soucier et recalculer une cellule autant de fois que l’on va y passer, et donc recalculer les cellules suivantes.

Invariant et variant

Invariant

```
pcTableData != null;
pcTableData.length > 0;
pcTableData[0].length > 0;
\forall int x, y; x >= 0 && x <= pcTableData.length && y => 0 && y <= pcTableData[0].length; (
    pcTableData[x][y] == 'T'
    || pcTableData[x][y] == '#'
    || pcTableData[x][y] == '0'
);
```

Variant

La variable `rx` correspond au nombre de cellules maximal restantes à traiter sur une ligne et `ry` au nombre de lignes restantes à traiter.

Dans le cas où iParamPositionY est alors, `rx` prend la valeur:

```
rx =  pcTableData.length – iParamPositionX
```

Dans le cas contraire, cette variable vaut:

```
rx = iParamPositionX
```

La variable `ry` correspond au nombre de lignes (complètes) restantes à traiter. Cette variable prend toujours la valeur

```
ry  = pcTableData[0].length – (rx + 1)
```

Le variant résultant (noté `i`) qui prend en compte ces deux variables est donc :

```
i = ry * pcTableData.length + rx
```

Ordre de grandeur et espace mémoire requis

> Donnez l'ordre de grandeur du temps et de la place mémoire de cet algorithme naïf.

Au plus simple la case (0,0) contient `#` , le bûcheron ne peut rien faire. L'ordre de grandeur est de O(1)

Au pire, toutes les cases sont accessibles et pour chaque case, il faut considérer les 2 chemins possibles à chaque fois (pour peu que les limites du terrain le permettent) on obtient donc un ordre de **$O(2^n)$** ce qui est vraiment très lent !

Côté mémoire, cet algorithme est peu gourmand. Il travaille directement sur la structure donnée en entrée et ne nécessite pas de générer des structures de travail intermédiaires. L'espace mémoire nécessaire est donc (à une constante près) identique à la taille de la structure donnée en entrée.

Solution optimisée

> Déterminez la sous-structure optimale dont on a besoin pour résoudre ce problème.

Pour résoudre ce problème de façon plus efficace, il peut être utile de ne pas devoir recompter les cellules par lesquelles on est déjà passé. Il faut donc créer un 2ème tableau de type entier et de dimensions identiques au tableau du problème. Dans un premier temps, toutes les cellules doivent contenir une valeur indiquant que la cellule n'a pas été calculée. On choisira une valeur impossible à obtenir par un calcul → Le JML dans le code nous indique qu'une valeur ne peut contenir moins que 0 ou plus que le nombre de cellules. Dans ce code, on choisira la valeur -1. A chaque passage dans une cellule, on va contrôler le contenu du tableau de résultats (à la même position que notre cellule à calculer). Si celui-ci contient -1, on effectue l'opération de calcul et on va stocker le résultat du calcul dans le tableau de résultats. Sinon, la cellule a déjà été calculée et on peut renvoyer le résultat directement.

Le code et les spécifications se trouvent en annexe.

Ordre de grandeur de la solution optimisée

> Donnez l'ordre de grandeur du temps et de la place mémoire de cet algorithme basée sur le principe de la "programmation dynamique".

Si on sauvegarde en mémoire le résultat du calcul d'une cellule, il n'est plus nécessaire de refaire le calcul. Donc, si on passe par un autre chemin et qu'on retombe dans cette cellule, le résultat est obtenu directement sans devoir recalculer la suite. L'ordre de complexité de l'algorithme passe de cette manière de $O(2^n)$ à **O(n)** ce qui est un gain considérable.

Algorithme glouton

Analyse

> Analysez l'énoncé : détecter les ambigüités, les contradictions, etc,

Selon l'énoncé, un cercle d'amis est possible s'il est possible de construire un graphe où chaque individu est lié à autant d'autres individus du réseau qu'il a d'amis. Une personne ne peut donc pas être amie avec elle-même ou avoir plusieurs liens avec un autre individu. De plus, comme nous connaissons l'ensemble de notre réseau, personne ne peut être ami avec des personnes extérieures. L'ensemble des relations doivent donc impérativement aller par paires car si un individu a un ami, cet ami doit également être ami avec le premier individu. Ne pas avoir d'ami ne pose pas de problème, on peut imaginer un individu qui vient tout juste de rejoindre le réseau et qui a une liste d'ami vide, il agit alors comme un noeud orphelin qui peut ne pas être pris en considération. Dans la même logique, un réseau social tout juste créé n'aura aucun individu et sera également valide.

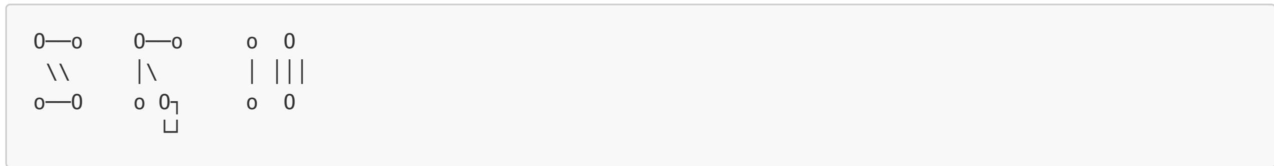
Selon l'ensemble de ces considérations, un réseau social est invalide si:

- 1e propriété, La somme de toutes les relations est impaire.
- 2e propriété, Un individu a au moins autant d'amis que la taille du réseau.
- 3e propriété, Un individu est ami avec un même individu plusieurs fois.

Ce à quoi on ajoute:

- 4e propriété, Le réseau vide est valide.
- 5e propriété, Les noeuds orphelins ne sont pas à prendre en compte.

Les deux premières conditions peuvent se vérifier très facilement mais leur contraire n'assure pas la validitéd'un réseau. Exemple, le cerle de 4 amis où les individus ont respectivement 3, 3, 1 et 1 ami(s) n'est pas un réseau valide car il nécessite qu'un des deux individus ayant 3 amis soit ami avec lui-même ou que les deux individus ayant 3 amis soient ami l'un avec l'autre plus d'une fois.



Solution naïve

> Construisez une solution naïve.
>
> Donnez l'ordre de grandeur du temps et de la place mémoire de cet algorithme naïf.

Nous avons manqué de temps pour répondre à cette question. L'algorithme qui est implémenté actuellement est incorecte car il ne vérifie pas la 3e propriété, ainsi le réseau `3 3 1 1` est considéré comme valide.

De plus cet algorithme prend juste notre solution glutone à l'envers, il part d'un réseau de la même taille que le réseau d'origine et crée des liens entre les individus. Il s'agissait d'une tentative d'implémentation d'un algorithme en *générer et tester* mais dans les fait il s'agit d'un algorithme glouton.

Choix glouton

> Déterminez la propriété gloutonne dont on a besoin pour résoudre ce problème,

Comme montré dans l'analyse, il est possible d'invalider simplement un réseau de deux manières : la somme du nombre d'amis de chaque individu est impaire ou un individu a au moins autant d'amis que la taille du réseau.

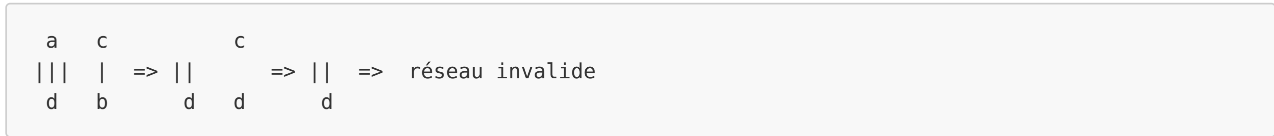
Déterminer qu'un individu a plusieurs fois le même ami est par contre beaucoup plus compliqué. Selon nous, il n'est pas possible de déterminer si un réseau ne possède aucun "tricheur" à priori. Il y a cependant moyen de ramener le réseau dans un cas suffisamment simple que pour vérifier cette 3e propriété : en retirant, un à un, tous les individus d'un réseau, on arrive à terme à n'avoir plus qu'une seule personne dans le réseau. Si cette personne n'a pas d'ami, le réseau est valide. Si cette personne a des amis, elle ne respecte pas la seconde propriété selon laquelle elle devrait avoir moins d'amis que la taille du réseau et la 3e propriété est ainsi invalidée.

Afin de vérifier la 3e propriété, il est donc nécessaire de réduire le réseau de manière à le ramener à un réseau assez simple pour être validé par la 4e propriété ou être invalidé par la 2e propriété. Réduire le graphe peut se faire en retirant un à un les individus qui le constituent ce qui signifie en réalité retirer autant de liens sur le graphe que cet individu n'a d'amis.

Retirer les liens doit prendre en compte la 3e propriété en retirant des liens entre personnes différentes. Ne pas prendre en compte cette propriété pourrait mener à des suppressions de liens entre mêmes individus et rendre valide un réseau qui ne l'est pas à la base.

Exemple, le réseau (invalide) `3 1 1 3` qui respecte les 1e et 2e propriétés auquel on supprime `a` ainsi que ses trois relations à `d` sans respecter la 3e propriété:

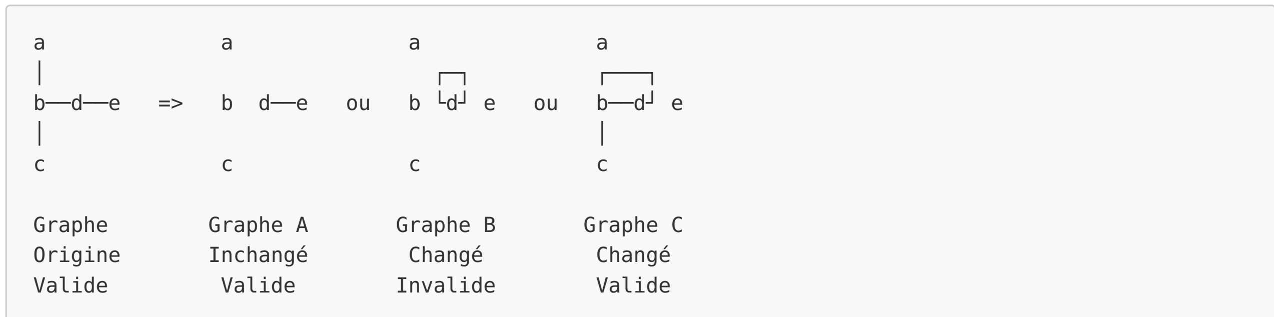
Même réseau où la 3e propriété est prise en compte:



```
 a   c          c
|||  |  => ||      => ||  =>  réseau invalide
 d   b      d   d      d
```

La réduction du graphe doit également se faire de manière à maintenir sa forme générale. Les liens existants ne peuvent être que supprimés, de nouveaux liens ne peuvent pas apparaitre et les liens existants ne peuvent pas être déplacés. Le maintien de la forme est primordiale car elle permet de ne pas fausser un graphe valide (voir l'exemple du graphe b ci-dessous). Choisir le bon individu à supprimer ainsi que les personnes qui étaient amis avec lui est donc déterminant. Deux choix sont légitimes, supprimer la personne la plus/la moins populaire et décider de retirer un ami aux plus/aux moins populaires.

Ci-dessous, un graphe `3 2 1 1 1` réduit de trois manières différentes. Le graphe A choisis les plus populaires, le graphe C choisis les moins populaires et le graphe B supprime le plus populaire et retire un ami aux moins populaires.



```
a             a             a             a
|                          ┌─┐           ┌───┐
b──d──e   =>   b  d──e   ou   b └d┘ e   ou   b──d┘ e
|                                        |
c             c             c             c

Graphe       Graphe A      Graphe B      Graphe C
Origine      Inchangé       Changé        Changé
Valide        Valide       Invalide       Valide
```

Le choix glouton nous permettant de maintenir la forme du graphe après réduction et ainsi assurer que l'état du réseau ne changera pas est donc : **supprimer l'individu le plus populaire et retirer 1 ami aux personnes les plus populaires suivantes pour autant de personnes que le 1er individu avait d'amis**

Spécification

> Spécifiez le problème en utilisant JML (pré et post-conditions)

Afin d'optimiser la réduction, il a été décidé de ne travailler que sur un réseau **trié**. De cette manière, récupérer les personnes les plus populaires peut se faire immédiatement.

```java
/** Private gluton function that validate a complexe mesh
  *
  * @ public static model boolean recIsValid_spec(int[] network, int length);
  * @ normal_behavior
  * If the network is empty, it is valid.
  * @ requires length == 0;
  * @ assignable \nothing;
  * @ ensures \result == true;
  * @ also normal_behavior
  * If there is just one person in the network, the network is valid if he doesn't have friends.
  * @ requires oldNetwork != null;
  * @ requires 0 <= length && length <= oldNetwork.length;
  * @ requires length == 1;
  * @ assignable \nothing;
  * @ ensures \result == (oldNetwork[0] == 0);
  * @ also normal_behavior
  * If there are at least two people in the network, the network is invalid if someone is
  *  friend with more people than the size of the network.
  * @ requires oldNetwork != null;
  * @ requires 2 <= length && length <= oldNetwork.length;
  * @ requires (
  * @   \forall int i; 0 < i && i < oldNetwork.length; oldNetwork[i - 1] <= oldNetwork[i];
  * @ requires (
  * @   \forall int i; 0 <= i && i < oldNetwork.length; oldNetwork[i] != 0
  * @ ) ==> \max int i; 0 <= i && i < length; oldNetwork[i] >= length;
  * @ requires (
  * @   \exists int i; 0 <= i && i < oldNetwork.length; oldNetwork[i] == 0
  * @ ) ==> \max int i; 0 <= i && i < length; oldNetwork[i] >= length - (
  * @   \max int j; 0 <= j && j < oldNetwork.length; network[j] == 0
  * @ ) - 1;
  * @ \fresh network;
  * @ ensures \result == false;
  * @ also normal_behavior
  * If the actual network is valid, then a network without one of its node (and the
  * links of that node) must be valid too. Removing the most popular node
  * and his friend ensures we keep the shape of the network.
  * @ requires 2 <= length && length <= oldNetwork.length;
  * @ requires (
  * @  \forall int i; 0 < i && i < oldNetwork.length; oldNetwork[i - 1] <= oldNetwork[i]
  * @ );
  * @ requires (
  * @   \forall int i; 0 <= i && i < oldNetwork.length; oldNetwork[i] != 0
  * @ ) ==> \max int i; 0 <= i && i < length; oldNetwork[i] < length;
  * @ requires (
  * @   \exists int i; 0 <= i && i < oldNetwork.length; oldNetwork[i] == 0
  * @ ) ==> \max int i; 0 <= i && i < length; oldNetwork[i]) < length - (
  * @   \max int j; 0 <= j && j < oldNetwork.length; oldNetwork[j] == 0
  * @ ) - 1;
  * @ \fresh network;
  * @ ensures \forall int i; 0 <= i && i < network.length; network[i] >= 0;
  * @ ensures \forall int i; 0 < i && i < network.length; network[i - 1] <= network[i];
  * @ ensures network.length <= oldNetwork.length;
  * @ ensures \result == recIsValid_spec(network, network.length - 1);
  * @ measured_by length;
  */
private static boolean recIsValid(int[] oldNetwork, int length)
```

Implémentation

> Ecrivez un algorithme glouton pour trouver si la liste des degrés est possible ou non.

Seule la fonction gloutone est affiché, les autres fonctions utilisés par l'algorithme se trouvent sur github dans le fichier `SocialNetwork.java`

```java
public static boolean isValid(int[] network) {
    if (network.length == 0)
        return true;
    int min_ = min(network);
    if (min_ < 0)
        return false;
    int max_ = max(network);
    if (max_ >= network.length)
        return false;
    counting_sort(network, min_, max_);
    return recIsValid(network, network.length);
}

private static boolean recIsValid(int[] oldNetwork, int length) {
    if (length == 0) {
        return true;
    }
    int mostFriendsCount = oldNetwork[length - 1];
    if (length == 1) {
        // If there is just one person in the network, the network is
        // valid if that person doesn't have any friends
        return mostFriendsCount == 0;
    }

    int start = skipLeadingZeros(oldNetwork, 0, length - 1);
    int[] network = new int[length - start];
    System.arraycopy(oldNetwork, start, network, 0, length - start);
    if (mostFriendsCount >= network.length) {
        // If the most popular person has more friends than the size
        // of the network, this network is not possible
        return false;
    }
    removeRelations(network, 0, network.length - 1);
    shift_sort(network, 0, network.length - 2);
    return recIsValid(network, network.length - 1);
}
```

Variant et Invariant

> Exprimez l'invariant et variant de boucle. Prouvez la correction de l'invariant et la terminaison de boucle,

Le variant est la taille du graphe, celui-ci va diminuer progressivement au fur et à mesure que le réseau sera réduit (par l'étape de réduction gloutone ainsi que par la 5e propriété).

L'invariant est le fait que le graphe, réduction après réduction, garde sa nature de graphe valide ou invalide comme expliqué en large dans notre choix glouton.

La terminaison est assuré par la vérification de la 2e propriété réduction après réduction et de la 4e propriété lors de la toute dernière étape de la réduction lorsque le graphe final est vide.

L'ensemble des fonctions utilisés pour résoudre ce problème a été spécifié en JML avec des pré-conditions, post-conditions, variant, invariant et invariant de boucle en ce sens.

Complexité

> Donnez l'ordre de grandeur du temps et de la place mémoire de cet algorithme basé sur le principe de "glouton".

Ordre de grandeur

L'algorithme repose sur 12 sous algorithmes:

i. Vérification de la 4e propriété sur le réseau total (réseau vide): **O(1)**
ii. Recherche sur le réseau total de la personne la plus populaire pour la vérification de la 2e propriété: **O(n)**
iii. Calcul de la somme totale pour la vérification de la 1e propriété: **O(n)**
iv. Tri de la structure par le tri du dénombrement (*counting sort*): **O(n)**
v. De manière récursive, pour chaque réduction du réseau: **O(n)**
    i. Vérification de la 4e propriété sur le réseau réduit: **O(1)**
    ii. Application de la 5e propriété (ignorer les personnes sans amis), première étape de la réduction: **O(n)**
    iii. Vérification de la 2e propriété sur le réseau réduit (plus populaire que la taille du réseau): **O(1)**
    iv. Copie du réseau original dans un réseau réduit: **O(n)**
    v. Suppression de la personne la plus populaire et de ses liens vers les autres, seconde étape de la réduction: **O(n)**
    vi. Passe unique du tri par insertion: **O(n)**
    vii. Appel récursif terminal: **O(1)**

Nous totalisons donc: **O(n^2)**

Espace mémoire

Donné `n`, la taille du réseau original.

i. Vérification de la 4e propriété sur le réseau total (réseau vide): **0**
ii. Recherche sur le réseau total de la personne la plus populaire pour la vérification de la 2e propriété: **1**
iii. Calcul de la somme totale pour la vérification de la 1e propriété: **1**
iv. Tri de la structure par le tri du dénombrement (*counting sort*): **n**
v. De manière récursive, pour chaque réduction du réseau: **1**
    i. Vérification de la 4e propriété sur le réseau réduit: **0**
    ii. Application de la 5e propriété (ignorer les personnes sans amis), première étape de la réduction: **1**
    iii. Vérification de la 2e propriété sur le réseau réduit (plus populaire que la taille du réseau): **0**
    iv. Copie du tableau original dans un tableau réduit: **n**
    v. Suppression de la personne la plus populaire et de ses liens vers les autres, seconde étape de la réduction: **1**
    vi. Passe unique du tri par insertion: **n**
    vii. Appel récursif terminal: **1**.

Nous totalisons donc: **4n**