TP5: Récursivité (récipients)

TOP : Techniques and tOols for Programming – 1A

L'objectif de ce nouveau TDP est de résoudre un nouveau problème par backtracking. La spécificité de ce problème-ci est que si vous agissez sans précaution, votre programme peut entrer en boucle infinie...

★ **Exercice 1: Présentation du problème des récipients.**

On dispose d'un certain nombre de récipients dont on connaît la capacité, et d'une fontaine d'eau. On cherche quels sont les transvasements à réaliser pour passer faire en sorte que l'un des récipients contienne une quantité d'eau donnée. Seules les opérations suivantes sont autorisées :

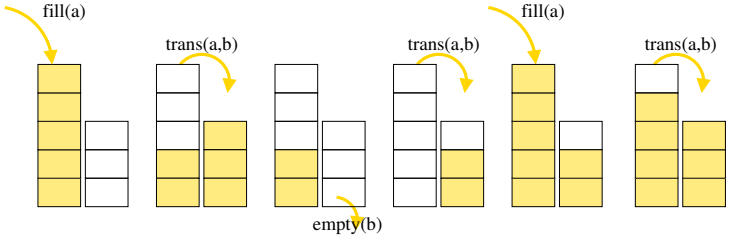
A. Remplir complètement un récipient depuis la fontaine ;

B. Vider complètement un récipient dans la fontaine ;

C. Transvaser un récipient dans un autre jusqu'à ce que la source soit complètement vide ou que la destination soit complètement pleine.

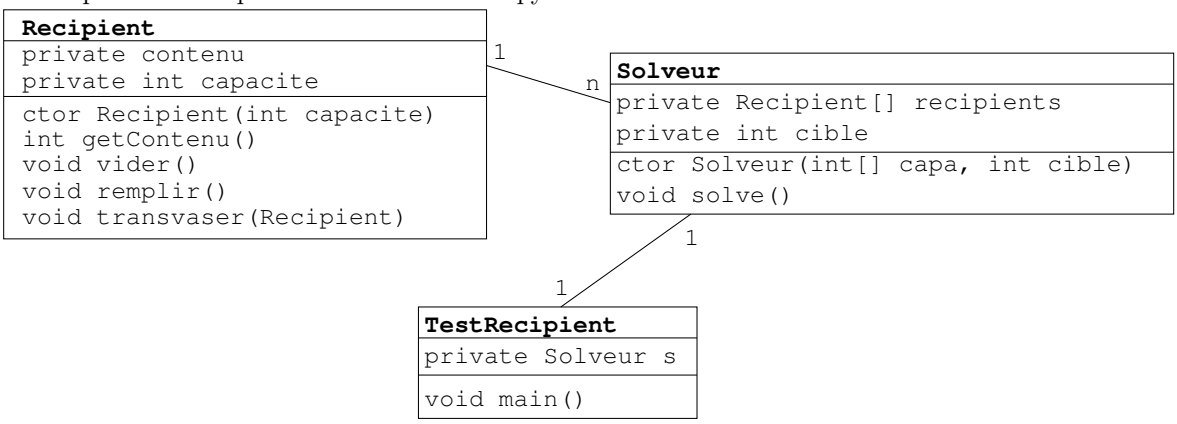
▷ **Exemple.** On suppose avoir deux récipients de capacité respective 5 et 3. On veut mesurer un volume de 4. D'après une situation initiale où les deux récipients sont vides, notée $(0, 0)$, les opérations suivantes permettent d'y parvenir.

- Remplir A à la fontaine : (5,0)
- Transvaser A dans B : (2,3)
- Vider le contenu de B : (2,0)
- Transvaser A dans B : (0,2)
- Remplir A à la fontaine : (5,2)
- Transvaser A dans B : (4,3)
- On a bien 4 unités dans A.



▷ **Question 1:** Pouvez-vous trouver une instance de ce problème n'admettant pas de solution ?

★ **Exercice 2: Réflexions sur le codage.** Pour simplifier, nous vous proposons d'organiser votre code de la façon suivante. Il s'agit d'un schéma très classique et relativement proche de ce que nous avons mis en œuvre précédement pour le sac à dos ou les pyramides.



▷ **Question 1:** Donnez le (pseudo-)code de la méthode `Recipient.transvase(Recipient)`.

▷ **Question 2:** Implémentez ces classes, à l'exception de la méthode solve() du solveur, qui constitue le cœur du TP et sera implémenté plus tard. Dans la classe de test, utilisez l'instance du problème présentée dans l'exemple ci-dessus. L'intéret est que vous avez la garantie qu'une solution existe pour cette instance.

★ **Exercice 3: Réflexions algorithmiques.** En absence de meilleure idée, nous allons établir une recherche exaustive des solutions de transvasement jusqu'à trouver une situation où l'un des récipients contienne la bonne quantité de liquide. Comme vous vous en doutez, nous allons le faire par backtracking.

▷ **Question 1:** Quel est le paramètre de récursion ? Quels sont les cas triviaux ?

Comme souvent lors d'une recherche combinatoire, nous allons donc utiliser une récursion dont chaque étage est une boucle parcourant toutes les possibilités existantes. Le pseudo-code serait quelque chose comme ça :

```
fonction_recursive(parametres)
  Si je suis sur un cas trivial, j'arrête
  Pour chaque décision D que je peux prendre maintenant
    appliquer D
    fonction_recursive(paramètres modifiés)
    annuler les changements dûs à D
```

▷ **Question 2:** Relisez le cours à propos du placement des reines, le code que vous avez écrit pour le sac à dos et celui pour les pyramides pour voir comment ce pseudo-code était mis en pratique dans chaque cas.

▷ **Question 3:** Écrire la liste des actions autorisées à chaque étape de la recherche exaustive.
Indication : il y a sans doute deux boucles `for` imbriquées, et un `if` dedans.

▷ **Question 4:** Écrivez le pseudo-code de la fonction récursive en combinant les deux questions précédentes. Ne cherchez pas encore à résoudre le problème posé par la ligne "annuler les changements dûs à D" dans le pseudo-code.

Il s'agit maintenant de sauvegarder l'état avant les modifications, et de le restaurer après la récursion. Le plus simple pour cela est d'ajouter un *copy-constructor* à la classe Recipient, c'est à dire un constructeur prenant un autre récipient en argument et faisant en sorte que l'objet nouvellement créé soit une copie conforme de l'argument.

▷ **Question 5:** Écrivez ce constructeur (sans ajouter d'autre méthode ni rendre les champs publiques), et utilisez le pour finir la fonction solve.

Codé comme cela, notre code "fonctionnerait", mais entrerait en boucle infinie. À chaque étage de la récursion, nous déciderions de transvaser le premier récipient dans le second (ce qui ne sert à rien vu que les deux sont vides), avant de plonger un étage plus bas dans la récursion. À l'infini.

▷ **Question 6:** Pour résoudre ce problème, modifiez votre pseudo-code pour arrêter la recherche après un nombre donné de transvasements.

★ **Exercice 4: Première implémentation (bounded depth first).**

▷ **Question 1:** Vous ne devriez pas être loin d'une solution fonctionnelle, et implémenter votre pseudo-code ne devrait pas poser beaucoup de problème.

La seule chose manquante est une idée pour afficher le transvasement ayant mené à la solution lorsque vous en trouvez une. Pour cela, on peut construire lors de la descente une chaînes de caractères décrivant les opérations effectuées. Une autre idée est d'ajouter à la fonction récursive un paramètre une liste d'opérations de transvasement et l'afficher quand on trouve, mais cela demande de définir une classe représentant ces opérations.

▷ **Question 2:** Retrouvez la solution à l'instance du problème donnée en exemple ci-dessus.

▷ **Question 3:** Trouvez une solution si les capacités valent {8,5,3} et que l'on cherche à obtenir 6 volumes en moins de 3 transvasements. Si c'est impossible, augmentez progressivement le nombre de transvasements autorisés au maximum jusqu'à trouver une solution. Répondez aux questions suivantes le temps que votre programme trouve une réponse.

▷ **Question 4:** Même question si les capacités valent {100, 25,24} et que l'on cherche à obtenir 42 volumes [1]. Il est probable que votre programme ne parvienne pas à la solution tant que vous n'aurez pas implémenté les améliorations de l'exercice suivant.

★ **Plus de contraintes pour plus de backtracking.** Notre approche "recherche en profondeur, avec profondeur maximale" est intéressante en ceci qu'elle permet d'éviter la boucle infinie consistant à réaliser la première action autorisée à chaque étage de récursion, même si cela ne mene nul part (cf. question 6 ci-dessus).

En revanche, son gros défaut est qu'il est difficile de déterminer la valeur à utiliser comme profondeur maximale. S'il est trop petit, on ne trouvera pas la solution. S'il est trop grand, l'espace grandit très (trop) vite. De plus, si on relance le programme avec une valeur plus grande, tous les calculs effectués avec une profondeur inférieure sont refaits. Cette situation est clairement sous-optimale...

On peut constater que notre programme effectue certes une quantité infinie d'opérations, mais qu'il n'existe qu'un nombre fini de situations. En effet, pour n récipients de capacité c_i chacun, nous savons que le premier récipient contient 0 unité ou bien 1 unité ou bien 2 ...ou bien c_1. De même, le nombre de façon de remplir chacun des autres récipients est borné. On en déduit que le nombre de plateaux (de remplissage de l'ensemble des récipients) n'est pas infini. Notre code fait donc des choses inutiles.

Pour changer cela, nous allons faire en sorte de ne parcourir que des solutions originales (jamais rencontrées auparavent), et couper par backtracking si on rencontre à nouveau une solution déjà vue. Pour déterminer si la situation actuelle a déjà été rencontrée auparavent, constatons tout d'abord qu'une situation est un vecteur de nombres, indiquant le remplissage de chaque récipients.

1. D'après une instance trouvée par Oswald Hounkonnou, promo 2012.

★ **Exercice 5: Stockage par liste de vecteurs.** Une première approche consiste à faire une liste de tous les vecteurs de valeur rencontrés. À chaque fois que l'on rencontre une nouvelle situation (au début de notre fonction récursive), on parcourt la liste pour voir si le vecteur courant est original, et on coupe court à la recherche si non.

▷ **Question 1:** Vous pouvez implémenter cette solution, ou constater sa mauvaise efficacité (tant en temps de calcul qu'en consomation mémoire).

★ **Exercice 6: Stockage par hachage.** Avec un peu plus de connaissances en Java que ce qui est demandé en TOP, une autre approche est d'utiliser une structure de données classique pour profiter des bonnes propriétés des tables de hachage (qui sont au programme du module de SD). Il suffit créer une variable de type `HashTree<String,Boolean>`, et d'utiliser ensuite les fonctions permettant de d'insérer un élément, et chercher si un élément donné existe dans la table. La clé des éléments sera le résultat de la méthode `toString()` appliquée à la solution courante.

▷ **Question 1:** Implémentez cette solution en vous appuyant sur la documentation des `HashTree`.

▷ **Question 2:** Discutez l'efficacité de cette solution (en particulier en terme de mémoire).

★ **Exercice 7: Stockage par tableau booléen.** Avec un minimum de connaissances mathématiques, une autre approche est de coder chaque vecteur de remplissage sous forme d'un entier unique. Il faut pour cela trouver une fonction allant de l'ensemble des vecteurs possibles dans l'ensemble des entiers. On utilisera la fonction `int hashCode()` pour cela.

▷ **Question 1:** Quelle propriété doit avoir cette fonction ?

Pour construire une telle fonction, on peut par exemple multiplier chaque élément du vecteur par un nombre premier différent. Par exemple, si le vecteur est de longueur 3, on peut utiliser les nombres premiers suivants : {3,5,7}. Ainsi, le vecteur [1,14,4] sera représenté par l'entier $3\times1+14\times5+7\times4 = 101$. Il suffit alors de disposer d'un tableau booléen nommé par exemple `dejaVu`, et de stocker sous `dejaVu[101]` si l'on a déjà vu le vecteur [1,14,4].

▷ **Question 2:** Implémenter cette solution.

▷ **Question 3:** Discutez l'efficacité de cette solution. Comment peut-on améliorer les choses ?