TP5: Récursivité (récipients)

TOP: Techniques and tOols for Programming

Première année

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 :

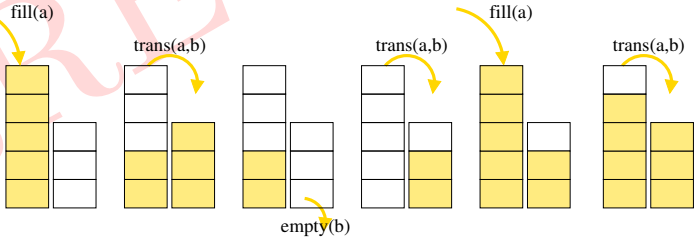
A. Remplir complètement un récipient depuis la fontaine ;

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.



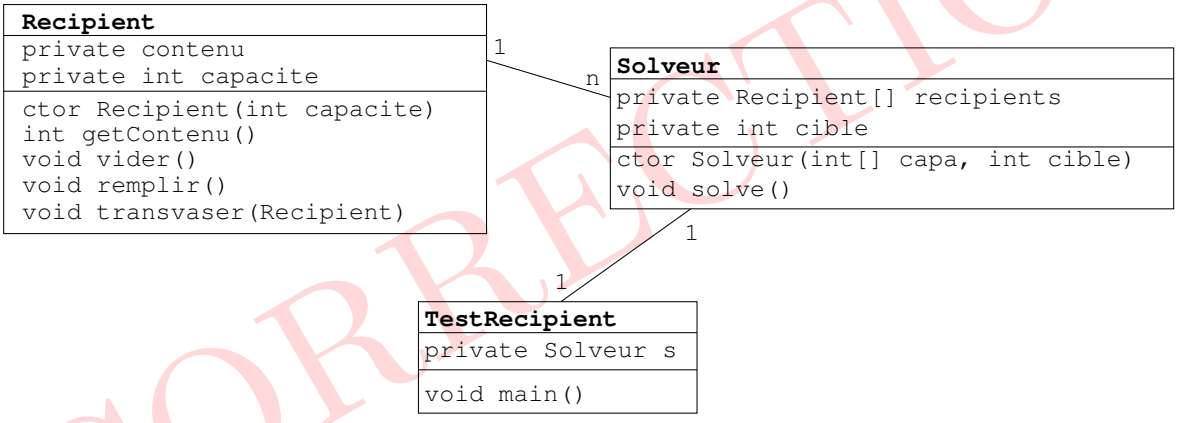
▷ **Question 1:** Pouvez-vous trouver une instance de ce problème n'admettant pas de solution ?

Ben ca va être plus dur, si, au choix :

- Si la cible est plus grande que n'importe quel récipient
- Si toutes les capacités sont égales les unes aux autres, et différentes de la cible cherchée

Fin réponse

★ **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)`.

Réponse

```
1 transvase(src,dst)
2   quantite = Min(src.contenu,dst.capacite-dst.contenu)
3   src.contenu -= quantite
4   dst.contenu += quantite
```

Fin réponse

▷ **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 ?

Réponse

Pour le paramètre, on a pas de bonne idée selon les données. La seule possibilité est le nombre de transvasements faits jusqu'à présent. Pour les cas triviaux, c'est si un récipient contient ce qu'on veut.

Fin réponse

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.

Réponse

```
for (a=0;a<=NB_RECIPIENTS;a++) { // on choisit un recipient source
  for (b=0;b<=NB_RECIPIENTS;b++) { // et un recipient cible ...
    // Si a==NB_RECIPIENTS, on remplit depuis la fontaine
    // Si b==NB_RECIPIENTS, on vide dans la fontaine
    if (a != b) {
      // Le corps ici
    }
  }
}
```

Fin réponse

▷ **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.

Réponse

```
boolean solve(int n) {
  for (src=0;src<=NB_RECIPIENTS;src++) { // on choisit un recipient source
    for (dst=0;dst<=NB_RECIPIENTS;dst++) { // et un recipient cible ...
      if (src != dst) {
        // Choisi et applique une décision
        if (src==NB_RECIPIENTS) { // on remplit depuis la fontaine
          recipients[dst].remplir();
        } else if (dst==NB_RECIPIENTS) { // on vide dans la fontaine
          recipients[dst].vider();
        } else {
          recipients[src].transvase(recipients[dst]);
        }

        solve(n+1);

```

```
            // TODO: annule la décision
        }
    }
  }
}
```

Fin réponse

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.

Réponse

Souvent, ca leur pose problème car ils pense qu'un champ privé est privé à une instance d'objet. Et non, la frontière est classe par classe et je peux tout à fait aller taper sur les champs privés d'un autre objet, du moment qu'ils sont de la même classe que moi.

```
public Recipient(Recipient other) {
  this.capacite = other.capacite;
  this.contenu = other.contenu;
}
```

Fin réponse

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.

Réponse

On fait tjs une recherche en profondeur, mais bornée cette fois, donc ca va aller : le code va faire n'imp sur N étapes, puis tester des choses un peu plus variées en remontant.

Il faut ajouter la profondeur max pour `new Solveur()` dans la classe de tests, ou bien passer à `solve()` le nombre d'étapes qu'il a encore le droit de faire. Et ca nous fait un nouveau cas trivial : si la profondeur courante dépasse la profondeur autorisée (ou si le nb d'étapes restantes arrive à 0), on coupe.

Fin réponse

★ 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.

Réponse

Je l'ai pas codé, j'ai pas la solution. J'espère que cette instance est pas triviale sinon, c'est dommage. Voici ce que j'ai trouvé sur internet (attention, ca semble juste, mais j'ai lu rapidement)

```c
http://www.yaronet.com/en/posts.php?sl=&s=4798&p=2

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
#define NB_RECIPIENTS 4
#define MAX_VOLUME 50
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
typedef struct S_recipient {
  int capacite; // Capacité totale du recipient
  int volume_ricard; // Volume de ricard présent dans le récipient
} T_recipient;
typedef struct S_transvasement {
  T_recipient *src; // Le recipient source // Si NULL alors fontaine
  T_recipient *dst; // Le recipient cible // Si NULL alors fontaine
  struct S_transvasement *suiv; // Pointe vers le transvasement suivant, NULL si pas de suivant
} T_transvasement;
T_recipient tab_recipient[NB_RECIPIENTS]; // Tableau representant NB_RECIPIENTS récipients
unsigned char buffer[MAX_VOLUME + 1];
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
int mini(int a,int b) // Retourne le min de 2 entiers
 { return(a<b?a:b); }
int maxi(int a,int b) // Retourne le max de 2 entiers
 { return(a>b?a:b); }
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
int get_capacite(T_recipient *recipient) // Retourne la capacité totale d'un récipient
 { return(recipient->capacite); }
int get_volume_ricard(T_recipient *recipient) // Retourne le volume de ricard présent dans le récipi
 { return(recipient->volume_ricard); }
int get_libre(T_recipient *recipient) // Quantité de ricard pouvant encore etre versée
 { return(recipient->capacite - recipient->volume_ricard); }
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
void suite_transvasement(T_transvasement *suite) { // effectue une liste chainée de tranvasements su
 int a;
 while (suite) { // tant qu'il reste des transvasements a effectuer on continue  :)
   if (suite->src && suite->dst) { // transvasement entre 2 recipients
     a = mini(get_libre(suite->dst),get_volume_ricard(suite->src)); // on determine la quantite de l
     suite->src->volume_ricard -= a; // on vide la source ...
     suite->dst->volume_ricard += a; // ... de la meme quantite que l'on remplit la destination !
   } else if (suite->src) { // transvasement entre recipient -> fontaine
     suite->src->volume_ricard = 0; // on vide le recipient
   else // transvasement entre fontaine -> recipient
     suite->dst->volume_ricard = get_capacite(suite->dst); // on remplit le recipient
   suite = suite->suiv; // on passe au transvasement suivant
  }
}
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------

void kestion5(int n)
{
  int a,b;
  T_transvasement trans;
  T_recipient save_src,save_dst;

  if (n==0) { // on vient de faire les n transvasements possibles  :)  ... on note le resultat des v
    for (a=0;a<NB_RECIPIENTS;a++)
      buffer[tab_recipient[a].volume_ricard]=1;
    return;
  }

  // il reste encore des transvasements a faire ...

  for (a=0;a<=NB_RECIPIENTS;a++) { // on choisit un recipient source
    for (b=0;b<=NB_RECIPIENTS;b++) { // et un recipient cible ... ensuite on va transvaser et voir c
      if (a != b) { // on evite les transvasements d'un recipient dans lui meme parce que c pas tres
```

```
75          if (a==NB_RECIPIENTS) {        // Tout ce micmac sert juste a
76            trans.src = NULL;            // initialiser la
77          } else {                       // structure trans avec une source
78            trans.src = &tab_recipient[a];              // (NULL si fontaine)
79            save_src.capacite = tab_recipient[a].capacite;      // et une destination
80            save_src.volume_ricard = tab_recipient[a].volume_ricard;// (NULL si fontaine)
81          }
82
83          if (b==NB_RECIPIENTS) {        // et le pointeur sur le transvasement suivant
84            trans.dst = NULL;            // a NULL pour ne faire qu'un transvasement par etape
85          } else {
86            trans.dst = &tab_recipient[b];
87            save_dst.capacite = tab_recipient[b].capacite;
88            save_dst.volume_ricard = tab_recipient[b].volume_ricard;
89          }
90
91          trans.suiv = NULL;
92          suite_transvasement(&trans);  // Ensuite on appelle la fonction suite_transvasement() avec l
93
94          kestion5(n-1);                        // Le transvasement a eut lieu ... on passe a un autre
95
96          if (trans.src) {               // Ici on est revenu de l'appel de fonction kestion5() !
97            tab_recipient[a].capacite = save_src.capacite;      // C'est donc que l'on est tombé sur u
98            tab_recipient[a].volume_ricard = save_src.volume_ricard;// les transvasements possible dan
99          }                              // et explorer les autres possibilites
100         if (trans.dst)                        //
101         {                              // Et pour revenir une etape en arriere comme si il s'etait r
102           tab_recipient[b].capacite = save_dst.capacite;      // la table des récipients a la valeur
103           tab_recipient[b].volume_ricard = save_dst.volume_ricard;// Du coup le programme n'y voit q
104         }                              //
105       }
106     }
107   }
108   return;
109 }
110
```

Fin 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.

★ **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.

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

▷ **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).

Ben c'est pas bon : on crée des chaines à tout bout de champs et on les stoque. On va faire sauter la mémoire.

En plus, on hache les chaines de représentation. On risque d'avoir pleins de conflits vu que les chaines se ressemblent toutes. Mais ca, vu qu'ils ont pas vu ce qu'est une table de hachage, on peut pas leur demander.

Fin réponse

★ **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 ? **Réponse**

Injective (il me semble, je me goure tjs). Il faut que $\forall x, y, (x \neq y) \Rightarrow (code(x) \neq code(y))$

Fin réponse

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 ?

Le hachage va mieux se passer, et on a beaucoup moins de chaines construites. C'est bien mieux. Mais on se trimbale maintenant avec un tableau de booléens de dimension de folie.

On peut améliorer en stockant l'information sous forme d'une liste (chainée) triée des valeurs déjà rencontrées. Le stockage sera plus efficace, la recherche est en $O(\log(n))$, et l'insertion en 0(1).

Le mieux est de faire un stockage creux du vecteur (c'est comme ca qu'on dit en calcul scientifique, rien de grave). On économise des pointeurs dans la liste ci-dessus en stockant des vecteurs dans chaque case du tableau, et en fusionnant les cellules adjacentes dans le meme vecteur.

$(1) \leftrightarrow (2) \leftrightarrow (3) \leftrightarrow (76) \leftrightarrow (77) \leftrightarrow 205$ devient $(1,2,3) \leftrightarrow (76,77) \leftrightarrow 205$, ce qui sauve une poignée de vecteurs. Ca risque d'être précieux si la structure grandit.

Fin réponse