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...

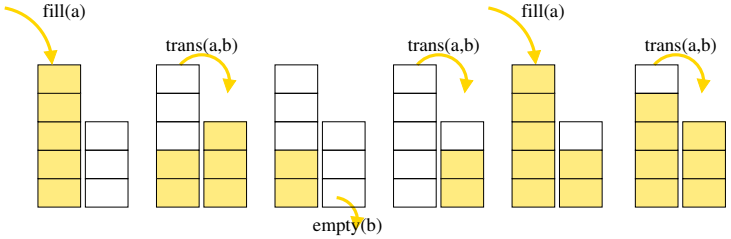
★ Exercice 1: Présentation du problème des récipients.

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.

La première étape est de réfléchir à comment représenter ce problème dans notre programme, puis à écrire les fonctions d'aide permettant par exemple de remplir un objet à la fontaine, de transvaser un objet dans un autre, et de vider un objet dans la fontaine. Le plus simple est probablement de s'inspirer de ce que nous avons fait dans les TP précédents.

★ Exercice 3: Parcours exhaustif simple (mais borné).

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 code ressemble à cela :

```
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 cette idée était mise en pratique dans chaque cas.

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

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

▷ **Question 5:** Il s'agit maintenant de sauvegarder l'état avant les modifications, et de le restaurer après la récursion. Le plus simple est de stocker dans des variables spécifiques le contenu des récipients modifiés à cet étage de récursion, puis de restaurer ces valeurs après la récursion. Modifiez votre code en ce sens.

▷ **Question 6:** En l'état, ce 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 change rien puisque que les deux sont vides), avant de plonger un étage plus bas dans la récursion. À l'infini. Pour résoudre cela, modifiez votre code pour ne pas effectuer les opérations qui ne mènent visiblement à rien. Attention à ne pas couper l'exploration trop brutalement, car sinon on risque de rater des branches importantes.

▷ **Question 7:** Malheureusement, les boucles infinies sont toujours possible malgré cette modification. Considérez l'historique suivant : remplir(0) ; transvaser(0,1) ; transvaser(1,0) ; transvaser(0,1) ; transvaser(1,0) ; transvaser(0,1) ; transvaser(1,0) ; transvaser(0,1) ; ...

Pour se prémunir contre ce problème, nous allons borner la profondeur de recherche. C'est-à-dire que vous devez arrêter la recherche après une quantité prédéterminée de transvasements.

▷ **Question 8:** Ajoutez maintenant ce qu'il faut pour afficher la séquence d'opérations menant à la solution une fois que vous l'avez trouvé. Comme souvent, plusieurs solutions sont possibles : on peut soit afficher les coups effectués (en sens inverse) lors de la remontée récursive, soit construire une chaîne de caractères (ou autres) lors de la descente décrivant les coups réalisés avec un accumulateur.

▷ **Question 9:** Sur machine, retrouvez la solution donnée en exemple d'introduction de ce sujet, où les capacités sont {3 et 5} pour une cible de 4.

▷ **Question 10:** Quel est le nombre minimal de transvasements pour retrouver 6 avec des récipients de tailles {8, 5, 3} ?

▷ **Question 11:** Même question pour retrouver 42 avec des récipients de tailles {100, 24, 25} [1]. Cette instance du problème est agaçante, car on peut la résoudre de tête puisque la séquence "vider 24 ; remplir 25 ; transvaser 25 dans 24" permet d'obtenir une unité. Il suffit alors de la répéter 42 fois pour trouver la cible. Mais notre programme est trop long pour parvenir à trouver une solution de profondeur 196 comme celle-ci.

★ **Exercice 4: Plus de contraintes pour couper plus de branches.**

Nous avons dû borner la profondeur maximale de la recherche car notre programme effectue parfois une quantité infinie d'opérations. Cela se produit quand il trouve un cycle d'actions consistant à faire et défaire, comme "A→B ; B→A". Pourtant, 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 auparavant), et couper l'exploration si l'on rencontre à nouveau une solution déjà vue. Pour déterminer si la situation actuelle a déjà été rencontrée auparavant, il suffit de stocker dans une liste toutes les situations rencontrées jusque là.

▷ **Question 1:** Écrivez les deux fonctions suivantes, qui teste l'égalité entre deux tableaux de même taille et qui teste si un élément donné est dans la liste donnée. `contient()` utilise naturellement `egal()`.

```
1  def egal(A:Array[Int], B:Array[Int]):Boolean = {...}
2  def contient(liste:List[Array[Int]], elm:Array[Int]):Boolean= {...}
```

La complexité asymptotique de ces fonctions est clairement un grand polynôme, mais en pratique, elles s'avèrent suffisantes dans certains cas.

▷ **Question 2:** Modifiez votre code pour stocker la liste de tous les états visités dans une variable globale. Il ne faut faire l'appel récursif que si l'état actuel est nouveau. Sinon, il faut couper l'exploration. Il n'est plus nécessaire de borner la profondeur de parcours, puisque les branches menant à des cas déjà visités seront coupées, rendant impossible toute recherche infinie.

▷ **Question 3:** Sur machine, testez votre code sur l'instance d'Hounkonnou. Une solution est trouvée en moins d'une minute, bien que la fonction `contient` soit en $O(n^3)$. Ce bon résultat est probablement dû à la chance, où nous n'avons pas besoin d'explorer toutes les branches pour trouver la solution.

En revanche, la solution trouvée de cette façon implique 259 opérations alors que l'on sait qu'il existe une solution en 196 opérations.

1. Récipients={100, 24, 25} ; Cible=42 : Instance proposée par Oswald Hounkonnou, promo 2012.

★ Exercice 5: Parcours en largeur.

On peut constater que même dans les cas où il fonctionne bien, notre code fait souvent du travail inutile. Par exemple, si on lui demande de chercher l'exemple de l'énoncé, il trouve la solution ci-contre en 10 coups alors qu'on en connaît une en 7 coups seulement. C'est que les cinq premiers coups sont une façon bien compliquée de remplir le récipient B !

Coup	Résultat
Remplir A	(5, 0)
A → B	(0, 5)
Remplir A	(5, 5)
A → B	(3, 7)
Vider A	(0, 7)
B → A	(5, 2)
Vider A	(0, 2)
B → A	(2, 0)
Remplir B	(2, 7)
B → A	(5, 4)

Le problème, qui explique également que la solution trouvée pour l'instance d'Hounkonnou ne soit pas optimale, vient de l'ordre de parcours des solutions possibles. Nous avons choisi un ordre qui s'appelle classiquement "en profondeur" (depth first en anglais). Cela explique que notre programme trouve d'abord la solution ci-contre avant de trouver la solution de l'énoncé, qui est certes plus courte mais rencontrée plus tard lors d'un parcours en profondeur.

Une piste pour éviter ce problème, proposée par Arthur Brongniart (promo 2013) est de réaliser un parcours en largeur au lieu d'un parcours en profondeur. Il s'agit d'explorer entièrement un étage de l'arbre avant de descendre au niveau suivant. Ce type de parcours est un peu plus compliqué à réaliser.

Pour cela, chaque étage de la récursion prend la liste de toutes les situations que l'on peut atteindre à la profondeur précédente, et construit la liste de celles que l'on peut atteindre avec une étape de plus.

▷ **Question 1:** Écrivez une telle fonction récursive qui calcule la profondeur minimale permettant de résoudre une instance donnée du problème. Il semble difficile d'afficher les coups ayant mené à la solution dans le cas d'un parcours en largeur pour l'instant. Contentez vous d'afficher la profondeur minimale nécessaire. De même, n'apportez aucune optimisation pour l'instant.

▷ **Question 2:** On voit clairement que cette solution correcte va trouver la réponse la plus courte, puisqu'elle évalue les solutions dans l'ordre de leur longueur. En revanche, elle ne permet pas de trouver la solution pour l'instance Hounkonnou car l'ordinateur arrive à court de mémoire pour stocker toutes les solutions en cours d'élaboration avant d'atteindre la solution.

▷ **Question 3:** Modifiez votre parcours en largeur pour sauvegarder la liste de tous les états visités afin de ne pas revisiter des sous-arbres déjà explorés. Une solution simple pour optimiser la consommation mémoire est de ne stocker la nouvelle solution partielle que si elle n'est pas déjà présente dans la liste des solutions partielles connues. Cette modification est très proche de celles apportées lors de l'exercice précédent.

Sur machine, cette variante permet de trouver une solution à l'instance Hounkonnou. Surprise, elle ne demande que 20 transvasements ! En revanche, cela ne dit pas quelle est cette solution... Saurez-vous déterminer les opérations nécessaires pour résoudre l'instance Hounkonnou ?

▷ **Question 4:** Testez votre code sur l'instance où l'on cherche 1 avec des récipients de taille {34, 55, 89, 144} [2]. Cette instance est plus complexe, et le code précédent ne suffit pas.

Saurez-vous déterminer le nombre minimal d'opération pour résoudre cette instance ? Quelles sont les opérations nécessaires pour la résoudre ?

▷ **Question 5:** Saurez-vous trouver une instance encore plus compliquée que celles connues à ce jour ? La complexité d'une instance du problème se mesure à la longueur de la plus courte solution permettant de la résoudre.

2. Récipients={34, 55, 89, 144} ; Cible=1 : Instance proposée par E. Thomé, chercheur en cryptographie au Loria.