TDP3: Récursivité (dichotomie et sac à dos)

TOP : Techniques and tOols for Programming – 1A

★ Exercice 1: Diviser pour régner : la dichotomie

La dichotomie (du grec « couper en deux ») est un processus de recherche où l'espace de recherche est réduit de moitié à chaque étape.

Un exemple classique est le jeu de devinette où l'un des participants doit deviner un nombre tiré au hasard entre 1 et 100. La méthode la plus efficace consiste à effectuer une recherche dichotomique comme illustrée par cet exemple :

- Est-ce que le nombre est plus grand que 50 ? (100 divisé par 2)
- Oui
- Est-ce que le nombre est plus grand que 75 ? ((50 + 100) / 2)
- Non
- Est-ce que le nombre est plus grand que 63 ? ((50 + 75 + 1) / 2)
- Oui

On réitère ces questions jusqu'à trouver 65. Éliminer la moitié de l'espace de recherche à chaque étape est la méthode la plus rapide en moyenne.

▷ **Question 1:** Écrivez une fonction récursive cherchant l'indice d'un élément donné dans un tableau trié. La recherche sera dichotomique.

Données :

— Un tableau trié de n éléments (`tab`)
— Les bornes inférieure (`borne_inf`) et supérieure (`borne_sup`) du tableau
— L'élément cherché (`élément`)

Résultat : l'indice où se trouve l'élément dans tab s'il y est, -1 sinon.

▷ **Question 2:** Calculez la complexité de cette fonction.

▷ **Question 3:** Montrez la terminaison de cette fonction.

▷ **Question 4:** Dérécursivez la fonction précédente.

★ Exercice 2: Présentation du problème du sac à dos

L'objectif du prochain TP sera de réaliser un algorithme de recherche avec retour arrière dans un graphe d'états. Nous allons maintenant nous familiariser avec ce problème.

Pour **éviter de boire la tasse sur machine**, il ne faut pas se lancer dans l'implémentation sur machine tête baissée à l'exercice 4. Avant cela, il faut absolument avoir fini et compris les exercices 2 et 3, qui préparent à cette implémentation. Notez également qu'un code à compléter est fourni pour l'exercice 4. Comme c'est la première fois que vous allez être amené à écrire du scala hors de la PLM, il faut aussi prendre le temps de vérifier que votre environnement de travail fonctionne. Mais avant tout, il s'agit de prendre les choses dans l'ordre et réfléchir ensemble pour la suite de l'exercice 2.

Le problème du sac à dos (ou *knapsack problem*) est un problème d'optimisation classique. L'objectif est de choisir autant d'objets que peut en contenir un sac à dos (dont la capacité est limitée). Des problèmes similaires apparaissent souvent en cryptographie, en mathématiques appliquées, en combinatoire, en théorie de la complexité, *etc.*

Problème :

> Étant donné un ensemble d'objets ayant chacun une valeur v_i et un poids p_i, déterminer quels objets choisir pour maximiser la valeur de l'ensemble sans que le poids du total ne dépasse une borne N.
> (on pose dans un premier temps $\forall i, v_i = p_i$. Imaginez qu'il s'agit de lingots d'or de tailles différentes)

Données :

— Le poids de chaque objet `i` (rangés dans un tableau `poids[0..n-1]`)
— La capacité du sac à dos `N`

Résultat :

— un tableau `pris[0..n-1]` de booléens indiquant pour chaque objet s'il est pris ou non

Le seul moyen connu [1] de résoudre ce problème est de tester différentes combinaisons possibles d'objets, et de comparer leurs valeurs respectives. Une première approche serait d'effectuer une recherche exaustive d'absolument toutes les remplissages possibles.

1. Ceci est du moins vrai dans la forme non simplifiée du problème et en utilisant des valeurs non entières.

▷ **Question 1:** Calculez le nombre de possibilités de sac à dos possible lors d'une recherche exaustive.

Une approche plus efficace consiste à mettre en œuvre un algorithme de recherche avec retour arrière (lorsque la capacité du sac à dos est dépassée) tel que nous le verrons en cours. Elle permet de couper court au plus tôt à l'exploration d'une branche de l'arbre de décision. Par exemple, quand le sac est déjà plein, rien ne sert de tenter d'ajouter encore des objets.

La suite de cet exercise vise à vous faire mener une réflexion préliminaire au codage, que vous ferez dans l'exercise suivant, lors du prochain TP.

▷ **Question 2:** Comment modéliser ce problème en Scala ? Par quel type de données allez vous représenter un sac à dos donné ?

▷ **Question 3:** Écrivez les méthodes `mettreDansSac()`, `retireDuSac()` et `estPris()` qui simplifient l'usage de votre structure de données.

▷ **Question 4:** Écrivez une fonction `valeurTotale`, prennant un sac à dos et un tableau d'entiers `poids` en paramètre (`poids` indique le poids de chaque objet possible du sac à dos) et renvoyant le poids total de tous les objets sélectionnés pour le sac à dos.

★ Exercice 3: Résolution récursive du problème du sac à dos

▷ **Question 1:** Dessinez l'arbre d'appel que votre fonction doit parcourir lorsqu'il y a quatre objets de tailles respectives $\{5, 4, 3, 2\}$ pour une capacité de 15.

▷ **Question 2:** Même question avec un sac de capacité 8.

Réfléchissez à la structure de votre programme. Il s'agit d'une récursion, et il vous faut donc choisir un paramètre portant la récursion. Appliquez le même genre de réflexion que celui mené en cours à propos des tours de Hanoï. Il est probable que vous ayez à introduire une fonction récursive dotée de plus de paramètres que la méthode `cherche()`.

▷ **Question 3:** Quel sera le prototype de la fonction d'aide ? Quel sera le prototype de la fonction récursive ?

Dans le même temps, l'objectif de votre méthode récursive est de trouver le meilleur nœud de l'arbre visité. D'une certaine façon, c'est assez similaire à une recherche de l'élément minimal dans un tableau : on parcours tous les éléments, et pour chacun, s'il est préférable au meilleur candidat connu jusque là, on décrète qu'il s'agit de notre nouveau meilleur candidat. Mais contrairement à l'algorithme d'une recherche d'élément minimum, les valeurs ne sont pas de simples entiers.

▷ **Question 4:** Écrivez une fonction qui recopie une structure représentant un sac à dos dans une autre. On l'utilisera pour sauvegarder le nouveau meilleur candidat.

▷ **Question 5:** Explicitez en français l'algorithme à écrire. Le fonctionnement en général (en vous appuyant sur l'arbre d'appels dessiné à la question 3), puis l'idée pour chaque étage de la récursion.

★ Exercice 4: Implémentation d'une solution au problème du sac à dos

Récupérez les fichiers **knapsack.jar** et **knapsack.scala** depuis la page du cours [2] ou depuis le dépôt [3].

▷ **Question 1:** Expérimentez avec le fichier **knapsack.jar**. Il s'agit d'une version compilée exécutable de la solution. Exécutez-le en lui passant diverses instances du problème en paramètre (en ligne de commande) afin de mieux comprendre comment fonctionne l'algorithme que vous devez écrire.

▷ **Question 2:** Le fichier **knapsack.scala** également fourni est un template de la solution, c'est-à-dire une sorte de texte à trou de la solution que vous devez maintenant compléter. Testez votre travail sur plusieurs instances du problème.

▷ **Question 3:** Combien d'appels récursifs effectue votre code ?

▷ **Question 4:** Testez votre travail pour des instances plus grandes du problème et observez les variations du temps d'exécution lorsque la taille du problème augmente.

▷ **Question 5:** Généralisez votre solution pour résoudre des problèmes où la valeur d'un objet est décorrélée de son poids (on ne suppose donc plus que $v_i = p_i$). Il s'agit maintenant de maximiser la valeur du contenu du sac en respectant les contraintes de poids. Vous serez pour cela amené à modifier toutes les fonctions écrites.

2. Page du cours : `http://www.loria.fr/~quinson/Teaching/TOP`
3. Dépot : `/home/depot/1A/TOP/knapsack` sur les serveurs de l'école.