TDP3: Récursivité (dichotomie et sac à dos)

TOP: Techniques and tOols for Programming

Première année

★ 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:** 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.

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[i..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.

▷ **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 l'avons vu 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.

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

▷ **Question 2:** De combien de classes Java avez vous besoin ? Dessinez le diagramme de classes.

▷ **Question 3:** 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 4:** Même question avec un sac de capacité 10.

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 5:** Quel sera le prototype de la fonction d'aide ?

▷ **Question 6:** 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 3: Implémentation d'une solution au problème du sac à dos**

Vous trouverez dans `/home/depot/1A/TOP/knapsack` une classe `Knapsack` contenant des éléments pour vous aider. Elle utilise la classe `KnapsackSolution`, également fournie.

▷ **Question 1:** Complétez la fonction `Knapsack.cherche()` afin de résoudre le problème du sac à dos. Vous utiliserez la classe `KnapsackSolution`. Vérifiez avec la classe `Test` fournie la validité de votre travail.

▷ **Question 2:** Combien d'appels récursifs effectue votre code ?

▷ **Question 3:** 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 classes fournies.

▷ **Question 4:** Vous trouverez dans le répertoire du dépôt un script `cree_instance.sh` capable de créer une nouvelle instance du problème à écrire dans le fichier `Test.java`. Observez les variations du temps d'exécution lorsque la taille du problème augmente.