

## Réponse

Cette semaine, on a un TD de (45mn de dichotomie/1h15 sac à dos), et un TP sur le sac à dos. But du jeu : ne pas écrire le code du TP en TD, mais faire assez d'exos d'échauffement sur le thème pour qu'ils écrivent le code seul sans soucis ensuite. Si tout va bien, faut se décarcasser en TD et ne plus rien avoir à faire en TP.

## Fin réponse

### ★ Exercice 1: Diviser pour régner : la dichotomie

## Réponse

L'objectif de cet exercice est d'appliquer les recettes de cuisine du cours pour (1) écrire une fonction récursive (voir le traitement appliqué aux tours de hanoi dans le cours) (2) dérécurser une fonction récursive terminale. Evidemment, sur un cas aussi simple, on peut faire directement au feeling, mais il faut insister sur la recette de cuisine histoire que tout le monde l'intègre.

## Fin réponse

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.

## Réponse

### Recette de cuisine :

- Paramètre : l'intervalle (et sa taille)
- Cas triviaux : si l'intervalle est de taille 1 ou 0
- Comment résoudre le pb avec l'aide d'un bon génie :
  - Couper l'intervalle en deux moitiés
  - Regarder dans quelle moitié peut être l'élément cherché (le tableau est trié)
  - Demander au génie de chercher pour de vrai dans cette moitié

**Complexité de l'algo :**  $\log(n)$  car on divise la quantité de chose à faire par 2 à chaque fois. Donc, en 1 étape, je gère 1 élément au max. En 2 étapes, 2 fois plus. En 3 étapes, encore 2 fois plus. En  $N$  étapes, je gère  $2^N$  éléments. Donc, pour gérer  $p$  éléments, il faut  $n$  étapes tel que  $2^n > p$ , ie,  $n > \log(p)$ . CQFD.

**Preuve de terminaison :** la taille de l'algo est divisé par 2 à chaque étape, et on s'arrête quand la taille est 1 (ou 0). Suite strictement monotone (attention aux divisions entières pour ça), effectivement convergeante vers le cas terminal, c'est bon.

**Code :**

## Version récursive

```

1 public class DichoRec {
2     public static int dichorec(int tab[], int min, int max, int elem) {
3         System.out.println("min: "+min+"; max: "+max);
4         if (max - min <= 1) {
5             if (tab[min] == elem) {
6                 return min;
7             } else if (tab[max] == elem) {
8                 return max;
9             } else {
10                return -1;
11            }
12        } else {
13            int milieu = (min + max) / 2;
14
15            if (elem < tab[milieu]) {
16                System.out.println("Debut (" + min + "; " + milieu + ")");
17                return dichorec(tab, min, milieu, elem);
18            } else {
19                System.out.println("Fin (" + milieu + "; " + max + ")");
20                return dichorec(tab, milieu, max, elem);
21            }
22        }
23    }
24    public static void main(String[] args) {
25        int tab[] = {1,3,6,7,8,9,10};
26        int min=0,max=6,elem;
27        System.out.println("RES: L'élément 0 se trouve en position "+
28                           dichorec(tab,min,max,0)+" Attendu: -1");
29        for (int cpt=0; cpt <=max; cpt++) {
30            System.out.println("RES: L'élément "+tab[cpt]+" se trouve en position "+
31                               dichorec(tab,min,max,tab[cpt])+" Attendu: "+cpt);
32        }
33        System.out.println("RES: L'élément 5 se trouve en position "+
34                           dichorec(tab,min,max,5)+" Attendu: -1");
35        System.out.println("RES: L'élément 11 se trouve en position "+
36                           dichorec(tab,min,max,11)+" Attendu: -1");
37    }
38 }

```

Je pense qu'on peut faire plus simple, mais pas sûr. Attention, si vous changez le test en  $min == max$ , ça merdoie grâce à cette saloperie de division entière contre-intuitive : ce cas (taille=0) n'arrive jamais si la taille initiale de l'intervalle est impaire et que l'élément cherché est dans une case impaire (ou un plan du genre).

Fin réponse

► **Question 2:** Dérécursivez la fonction précédente.

Réponse

Faut appliquer la recette de cuisine du cours. Tant que condition fausse, traitements du cas général. Ensuite, traitements du cas terminal.

## Version récursive

```

1 public class DichoIter {
2     public static int dicholter(int tab[], int min, int max, int elem) {
3         while (max - min > 1) {
4             int milieu = (min + max) / 2;
5             System.out.println("min: "+min+"; max: "+max+"; milieu: "+milieu);
6
7             if (elem < tab[milieu]) {
8                 System.out.println("Debut (" + min + "; " + milieu + ")");
9                 max=milieu;
10            } else {
11                System.out.println("Fin (" + milieu + "; " + max + ")");
12                min=milieu;
13            }
14        }
15
16        if (tab[min] == elem) {
17            return min;
18        } else if (tab[max] == elem) {
19            return max;
20        }
21    }
22 }

```

```

20     } else {
21         return -1;
22     }
23 }
24 public static void main(String[] args) {
25     int tab[] = {1,3,6,7,8,9,10};
26     int min=0,max=6,elem;
27     System.out.println("RES: L'élément 0 se trouve en position "+
28         dichoter(tab,min,max,0)+" Attendu: -1");
29     for (int cpt=0;cpt <=max; cpt++) {
30         System.out.println("RES: L'élément "+tab[cpt]+" se trouve en position "+
31             dichoter(tab,min,max,tab[cpt])+" Attendu: "+cpt);
32     }
33     System.out.println("RES: L'élément 5 se trouve en position "+
34         dichoter(tab,min,max,5)+" Attendu: -1");
35     System.out.println("RES: L'élément 11 se trouve en position "+
36         dichoter(tab,min,max,11)+" Attendu: -1");
37 }
38 }

```

---

**Fin réponse**

### ★ 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<sup>1</sup> 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 exhaustive d'absolument toutes les remplissages possibles.

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

---

**Réponse**

On devrait s'en sortir avec un  $C_n^p$ , mais on peut aussi constater qu'on a  $n$  objets, et que chacun est soit pris, soit pas pris. Ce qui nous fait  $n$  caractères sur un alphabet à 2 lettre, soit  $2^n$  possibilités.

---

**Fin réponse**

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 exercice vise à vous faire mener une réflexion préliminaire au codage, que vous ferez dans l'exercice 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. Pour de vrai, dans le cas qu'on présente ici, on peut le claquer en temps polynomial par programmation linéaire. Mais c'est hors sujet...



```
void cherche() {
    cherche.helper(int profondeur, KnapsackSolution etat_courant);
}
```

---

**Fin réponse**

---

▷ **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.

---

**Réponse**

---

Ce qui semble mal passer, c'est que ce n'est pas un if/then/else, mais une séquence. Je prend l'objet, je descend [à gauche], je le pose, je descend [à droite]. À chaque étage, c'est très similaire une fois qu'on a vu que c'est à l'objet N qu'on s'intéresse à l'étage N.

Un autre piège à expliciter est qu'on a vu en cours des backtracking avec une boucle, et ici y'a pas de boucle. C'est qu'on parcourt l'ensemble {pris, pas pris} alors on fait pas une vraie boucle. A la place, on teste les deux cas explicitement.

Il faut qu'ils comprennent car la semaine prochaine, on refait un autre backtracking.

---

**Fin réponse**

---

★ **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.

---

**Réponse**

---

Il y a 3 classes :

**Test.java** contient le main

**Knapsack.java** fait le parcourt récursif avec retour arrière

**KnapsackSolution.java** réifie les solutions au problème (on en manipule plusieurs lors du parcours : la courante, la meilleure rencontrée jusque là, etc)

**Ce sont les corrections du TP suivant, alors c'est pas une bonne idée de tout leur donner : faut qu'ils cherchent, un peu, aussi.**

---

Test.java

---

```
1 public class Test {
2
3     public static void main(String[] args) {
4         int valeurs [] = {5,4,3,2};
5         int capacite = 10;
6
7         Knapsack KS = new Knapsack(capacite, valeurs);
8         KS.cherche();
9         System.out.println("Meilleure solution: "+KS);
10    }
11 }
12 }
```

---

Knapsack.java

---

```
1 public class Knapsack {
2
3     /** Les données du problème,
4      * On copie ici plutôt que de les passer en paramètre de la récursion,
5      * puisqu'ils ne changent pas au cours de la récursion. */
6     private int [] valeur;
7     private int capacite;
8
9     /** La meilleure solution connue pour l'instant.
10     * La fonction récursive a (entre autres) des paramètres du même type
11     * qu'elle modifie lorsqu'elle construit sa solution. Lorsqu'une solution
12     * meilleure que la meilleure connue est trouvée, elle est stockée ici.
13     */
14     private KnapsackSolution meilleure;
15
16     /** Constructeur */
17     public Knapsack(int c, int [] vals){
18         capacite = c;
19     }
```

```

20     valeur = vals;
21
22     meilleure = new KnapsackSolution(vals);
23 }
24
25 /* quelques Getter pour des attributs privés */
26 public int getMeilleureValeur(){return meilleure.getValeur();}
27 public boolean [] getMeilleureSolution(){return meilleure.getPris();}
28
29
30 /* change l'objet en chaine de caractères pour le visualiser. Pratique pour debugger */
31 public String toString(){
32     return toString(valeur.length);
33 }
34
35 /* idem, mais n'affiche que les premiers objets (ie, une solution partielle) */
36 public String toString(int objMax) {
37     String s = "";
38     s = s + "\nValeurs: ";
39     for (int i=0; i<objMax; i++)
40         s = s + valeur[i] + " ";
41     s = s + meilleure.toString();
42     s = s + "\nCapacité: " + capacite;
43     return s;
44 }
45
46 private void chercheRec(int profondeur, KnapsackSolution courante) throws Exception {
47     System.out.print("(n="+profondeur+" Exploreqq"+courante.toString(profondeur));
48     if (courante.getValeur() > capacite) {
49         System.out.println(" *** Oups, ca deborde (backtrack!) ***");
50         return;
51     }
52
53     if (courante.getValeur() > meilleure.getValeur()) {
54         System.out.print(" Nouvelle meilleure solution ");
55         meilleure = courante.duplique();
56     } else {
57         System.out.print(" ");
58     }
59
60     if (profondeur == valeur.length) {
61         System.out.println("(Cas terminal)");
62         return;
63     } else {
64         System.out.println("(Cas général)");
65     }
66
67
68     /*
69     if (courante.getValeur() == capacite) {
70         System.out.println(" *** Solution parfaite ***");
71         throw new Exception();
72     }*/
73
74     /* Prend l'objet et récurse */
75     courante.prendObjet(profondeur);
76     chercheRec(profondeur+1, courante);
77
78     /* Pose l'objet et récurse */
79     courante.poseObjet(profondeur);
80     chercheRec(profondeur+1, courante);
81 }
82
83 public void cherche() {
84     try {
85         chercheRec(0, new KnapsackSolution(valeur));
86     } catch (Exception e) {}
87 }
88 }

```

KnapsackSolution.java

```

1 /** Classe implémentant une solution au problème du sac à dos.
2  *
3  * Elle peut également être utilisée pour manipuler des solutions partielles
4  * (ie, en cours de construction).
5  */
6

```

```

7 public class KnapsackSolution {
8     /** pour chaque objet i, il est dans le sac à dos ssi pris[i] est vrai */
9     private boolean [] pris;
10    /** valeur totale du contenu du sac */
11    private int valeurTotale;
12
13    /** valeurs de tous les objets. Pratique pour les methodes prendObjet et poseObjet */
14    private int valeur[];
15
16    /** Constructeur et fonctions basiques */
17
18    /** Initialise une solution à partir de l'instance du problème passée en paramètre */
19    public KnapsackSolution(int valeur[]) {
20        pris = new boolean[valeur.length];
21
22        for (int i=0; i<pris.length; i++)
23            pris[i]=false;
24
25        valeurTotale = 0;
26        this.valeur = valeur;
27    }
28
29    /** un getter ou deux */
30    public boolean [] getPris() { return pris; }
31    public /*@pure@*/ boolean getPris(int i) { return pris[i]; }
32    public /*@pure@*/ int getValeur() { return valeurTotale; }
33
34    public String toString() {
35        return toString(pris.length);
36    }
37
38    /** N'affiche que jusqu'à un certain objet (affiche une solution partielle) */
39    public String toString(int objMax) {
40        String s = "";
41        //s += "Solution Courante: ";
42        for (int i=0; i<objMax; i++)
43            if (pris[i])
44                s += " O ";
45            else
46                s += " N ";
47        while (s.length() < valeur.length*3)
48            s+= "...";
49        s += "; Valeur: " + valeurTotale;
50        return s;
51    }
52
53    /** Methodes d'usage et d'accès */
54
55    /**@ requires !getPris(i) ;
56    public void prendObjet(int i) {
57        pris[i] = true;
58        valeurTotale += valeur[i];
59    }
60
61    /**@ requires getPris(i) ;
62    public void poseObjet(int i) {
63        pris[i] = false;
64        valeurTotale -= valeur[i];
65    }
66
67    /** Fonctions avancées */
68
69    /** Crée une copie de l'objet courant */
70    public KnapsackSolution duplique() {
71        KnapsackSolution res = new KnapsackSolution(valeur);
72        for (int i=0; i<valeur.length; i++) {
73            if (pris[i]) {
74                res.prendObjet(i);
75            }
76        }
77        return res;
78    }
79
80 }
81
82 }

```

---

**Fin réponse**

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

**Réponse**

Y'a des étudiants qui vérifient la validité de leur travail en comptant le nombre d'appels récursifs effectués. Bonne idée. Le bon résultat est 15. Pas le nombre de feuilles de l'arbre, mais le nombre de nœuds internes...

---

**Fin réponse**

▷ **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.

**Réponse**

Oui, c'est exponentiel. Un peu mieux que  $2^n$ , mais ça reste très douloureux très vite.

---

**Fin réponse**