TDP3: Récursivité (dichotomie et sac à dos)

TOP : Techniques and tOols for Programming – 1A

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) dérécursiver une fonction récursive terminale. Evidement, 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'intervale (et sa taille)
- Cas triviaux : si l'intervale est de taille 1 ou 0
- Comment résoudre le pb avec l'aide d'un bon génie :
  - Couper l'intervale 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

Code : Je pense qu'on peut faire plus simple que le code ci-après, 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'intervale est impaire et que l'élément cherché est dans une case impaire (ou un plan du genre).

Remarquez que dans ce code, je définis la fonction récursive **dans** la fonction publique. On peut tout à fait déclarer des fonctions dans les fonctions en scala, ca réduit leur visibilité comme on s'y attend. Dans ce cas, la fonction récursive est de la cuisine interne, et l'appellant n'a pas à savoir comment c'est implémenté dedans (ca lui évite de savoir initialiser les arguments récursifs).

Version récursive

```
def dicho(tab:Array[Int], elm:Int):Int = {

    def dichoRec(tab:Array[Int], elm:Int, min:Int, max:Int):Int = {
        if (max-min <= 1) {
            if (tab(min) == elm) {
                return min
            } else if (tab(max) == elm) {
                return max
            } else {
                return -1
            }
        } else {
            val milieu = (min+max) / 2;

            if (elm < tab(milieu)) {
                // println("Debut  ("+min+";"+milieu+")");
                return dichoRec(tab,elm,min,milieu);
            } else {
                // println("Fin    ("+milieu+";"+max+")");
                return dichoRec(tab,elm,milieu,max);
            }
        }
    }


    return dichoRec(tab, elm,0, tab.length-1);
}

def test(tab:Array[Int]) {
    def cherche(searched:Int, expected:Int) {
        val found = dicho(tab, searched);
        if (found != expected)
            println("ERREUR: elm "+searched+" trouvé en "+found+" (attendu en "+expected+").");
        else
            println("elm "+searched+" trouvé en "+found+" comme attendu.");
    }
    cherche(0,-1);
    for (cpt <- 0 to tab.length-1)
        cherche(tab(cpt),cpt);
    cherche(5,-1);
    cherche(11,-1);
}

test(Array(1,3,6,7,8,9,10))
test(Array(1,3,6,7,9,10))
```

Fin réponse

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

$O(log(n))$ 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.

Fin réponse

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

Quand on fait une fonction récursive, il est très important de montrer qu'elle s'arrête tout le temps, pour 2 raisons : (1) écrire une fonction récursive qui boucle à l'infini est facile et courant ; (2) montrer ceci est relativement simple, c'est pas une preuve avancée.

Une condition suffisante pour montrer la terminaison d'un algo recursif (mais pas forcément nécessaire), c'est de montrer qu'il y a une grandeur qui varie de façon strictement monotone, et qu'elle atteint à coup sûr les cas d'arrêt de la récursion.

Ici : la taille du tableau est divisé par 2 à chaque étape, et on s'arrête quand la taille est inférieure ou égale à 1. Suite strictement monotone (attention aux divisions entières pour ca), effectivement convergeante vers le cas terminal, c'est bon.

Si on avait marqué explicitement que la longueur entre min et max est 0 ou 1, on aurait pu se faire avoir avec des cas où max passe à gauche de min. Peut-être bien que cela aurait pu passer à travers, ie

partir se promener chez les négatifs et donc partir à l'infini ($-\infty$). Mais cette simple précaution (écrire $i <= 1$ quand on pense à $i \in [0,1]$) nous assure que ce ne sera pas le cas.

Fin réponse

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

Réponse

Cette question est optionnelle. Si votre groupe a posé trop de questions sur le début, passez à la suite. On refera de la dérécursivation, et il faut absolument défricher le knapsack en TD car c'est le sujet du TP.

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

Version récursive

```
def dicho(tab:Array[Int], elm:Int):Int = {
    var min = 0;
    var max = tab.length-1;
    while (max-min>1) {
        val milieu = (min+max) / 2;
        if (elm < tab(milieu)) {
            max = milieu;
        } else {
            min = milieu;
        }
    }
    if (tab(min) == elm) {
        return min
    } else if (tab(max) == elm) {
        return max
    } else {
        return -1
    }
}

def test(tab:Array[Int]) {
    def cherche(searched:Int, expected:Int) {
        val found = dicho(tab, searched);
        if (found != expected)
            println("ERREUR: elm "+searched+" trouvé en "+found+" (attendu en "+expected+").");
        else
            println("elm "+searched+" trouvé en "+found+" comme attendu.");
    }
    cherche(0,-1);
    for (cpt <- 0 to tab.length-1)
        cherche(tab(cpt),cpt);
    cherche(5,-1);
    cherche(11,-1);
}

test(Array(1,3,6,7,8,9,10))
test(Array(1,3,6,7,9,10))
```

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[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.

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

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 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 ?

Réponse

Dans le template qui leur sera donné, j'ai modélisé le sac à dos comme un tableau de boolean, chacun représentant si l'objet correspondant est pris ou non. Il faut donc parvenir à les faire converger vers cette solution ;)

Fin réponse

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

Réponse

Ces méthodes sont triviales, et cette question a pour objectif principal que tout le monde comprend bien la modélisation utilisée.

```
/** indique dans le paramètre que l'objet spécifié est maintenant pris
    (cette fonction est juste là pour se simplifier la vie) */
def prendObjet(objets:Array[Boolean], obj:Int) {
    if (objets(obj))
        println("L'objet "+obj+" est déjà pris; ignore la requete");
    objets(obj) = true
}
/** indique dans le paramètre que l'objet spécifié est maintenant posé */
def poseObjet(objets:Array[Boolean], obj:Int) {
    if (!objets(obj))
        println("L'objet "+obj+" est déjà posé; ignore la requete");
     objets(obj) = false
}
/** ben oui, c'est juste un acces au tableau, quoi */
def estPris(objets:Array[Boolean], obj:Int) = objets(obj)
```

Fin réponse

▷ **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.

Réponse

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...

Il faut bien les laisser ramer avant de leur filer ce code, sinon ils vont s'ennuyer en TP. Si vous donnez tout le code au tableau en TD, ils vont râler (à juste titre) en réunion pédagogique que le travail en TD est de recopier le tableau sans réfléchir, puis que le travail en TP est de taper le TD sur l'ordi sans réfléchir.

Je ne serais pas choqué si vous ne corrigiez pas cette méthode au tableau, au fond. Donner l'idée générale suffira. Par exemple : "Je remarque que les deux tableaux doivent être de même taille pour que ça marche. Pour trouver la valeur, j'initialise une variable `total` à 0, puis pour tous les indices possibles, si l'objet est pris, j'ajoute son poids au total en cours de calcul. A la fin, la valeur renvoyée est `total`".

```
/** Calcule le gain d'un sac à dos donné */
def valeurTotale(objets:Array[Boolean], poids:Array[Int]):Int = {
  var total = 0
  for (i <- 0 to objets.length -1)
    if (objets(i))
      total += poids(i)
  return total
}
```

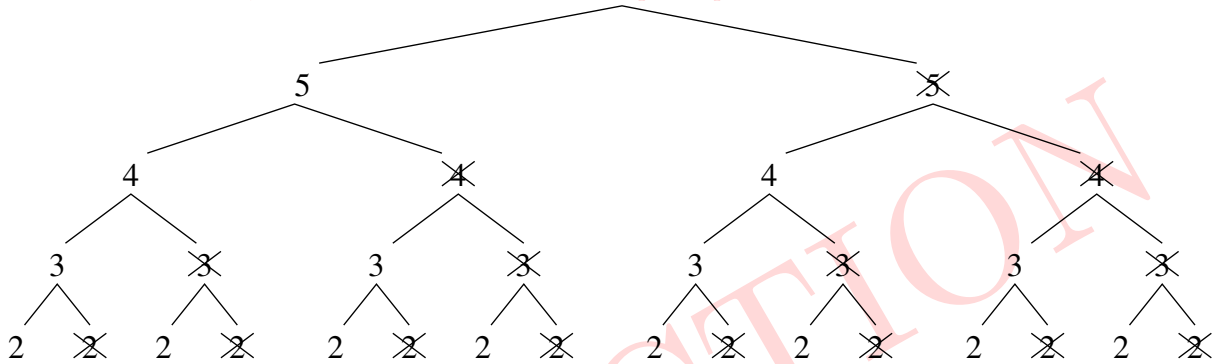
Fin réponse

★ 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.

Réponse

Dans le schema ci-dessous, la moitié gauche, sous "5" est les cas où on prend le premier objet, de taille 5. La moitié droite, sous "5 barré" est les cas où on prend pas 5.



Ouaip, il faut explorer tout l'arbre car le sac est trop grand, tout tient dedans – même pas drôle.

Fin réponse

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

Réponse

Là, on peut couper pas mal de branches. Par exemple, on n'a pas besoin d'aller au delà de 5,4 car le sac éclate déjà avec seulement 5 et 4. Ça ne risque donc pas d'aller mieux en ajoutant plus de choses. Donc pas besoin d'explorer le sous-arbre gauche.

Attention cependant, nous n'avons pas vu la notion de backtracking en cours. Mais c'est pas grave, car ce problème n'est pas vraiment un backtracking vu qu'il n'y a pas de boucle.

Fin réponse

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 ?

Réponse

Le mieux est de faire un prototype qui soit exactement celui de l'énoncé pour la fonction d'aide. Une ruse est de définir la fonction récursive **dans** la fonction d'aide. Comme ça, on est sûrs que personne ne va l'appeller en initialisant les paramètres n'importe comment, et en plus, ça lui permet de voir les paramètres de `cherche()`

```
/** La fonction publique, pour chercher la meilleure solution */
def cherche(poids:Array[Int] , capacite:Int) {

  /* Appel à chercheRec ici, et affichage du résultat */

  def chercheRec(profondeur:Int, courante:Array[Boolean]) {
    /* Code récursif ici */
  }
}
```

Fin réponse

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.

Réponse

```
def dupplique(src:Array[Boolean], dst:Array[Boolean]) {
  for (i <- 0 to src.length-1)
    dst(i) = src(i)
}
```

Fin réponse

▷ **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.

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.

Tâchez de ne pas prononcer le mot backtracking car (1) ils ne savent pas ce que c'est (2) ce n'est pas vraiment un backtracking comme on verra en cours vu qu'on a pas de boucle au sein de la récursion. Au lieu de parcourir l'ensemble {pris, pas pris} avec une vraie boucle, on teste les deux cas explicitement.

Il faut qu'ils comprennent l'idée de couper des branches dans une recherche exhaustive, car on recommence la semaine prochaine...

Fin réponse

★ 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.

Réponse

Pas de correction, faut juste qu'ils fassent mumuse avec le binaire fourni jusqu'à être à l'aise.

Fin réponse

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

▷ **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.

Réponse

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.

```
/** Problème du sac à dos */

object knapsack { // KILLLINE

// Diverses fonctions d'aide
//////////////////////////////

/** Calcule le gain d'un sac à dos donné */
def valeurTotale(objets:Array[Boolean], poids:Array[Int]):Int = {
  var total = 0
  for (i <- 0 to objets.length -1)
    if (objets(i))
      total += poids(i)
  return total
}

/** Affiche si les objets sont pris ou non dans une solution partielle (en cours de construction)
 *
 * Le second argument est la taille déja construite.
 */
def affiche(objets:Array[Boolean], objetMax:Int) {

  // on utilise un min pour se protéger du cas (fréquent) où cette
  // fonction est appellée avec un second argument supérieur à la
  // taille du tableau.
  for (i <- 0 to math.min(objetMax, objets.length-1))
    if (objets(i))
      print(" O ")
    else
      print(" N ")
  if (objetMax < objets.length-1)
    for (i <- objetMax to objets.length-2)
      print("...")
  print(";")
}


/** Dupplique un sac dans un autre, pour se souvenir de la meilleure solution connue */
def dupplique(src:Array[Boolean], dst:Array[Boolean]) {
  for (i <- 0 to src.length-1)
    dst(i) = src(i)
}

/** indique dans le paramètre que l'objet spécifié est maintenant pris
    (cette fonction est juste là pour se simplifier la vie) */
def prendObjet(objets:Array[Boolean], obj:Int) {
    if (objets(obj))
        println("L'objet "+obj+" est déjà pris; ignore la requête.");
    objets(obj) = true
}
/** indique dans le paramètre que l'objet spécifié est maintenant posé */
def poseObjet(objets:Array[Boolean], obj:Int) {
    if (!objets(obj))
        println("L'objet "+obj+" est déjà posé; ignore la requête.");
     objets(obj) = false
}

// La fonction principale
//////////////////////////

/** La fonction publique, pour chercher la meilleure solution */
def cherche(poids:Array[Int] , capacite:Int) {

  // on va beaucoup utiliser cette valeur, alors on fait un alias
  // pour simplifier les écritures suivantes
  val len = poids.length

```

```
68    // Affiche l'instance du problème
69    print("Poids des objets: ")
70    for (i <- 0 to poids.length-1)
71      print(" "+poids(i)+" ")
72    println("; Capacite: "+capacite)
73
74    // variable locale pour sauvegarder la meilleure solution connue à tout moment
75    var meilleure:Array[Boolean] = Array.fill(len)(false)
76
77    // BEGINKILL TODO: Placez ici l'appel récursif, avec les valeurs initiales des paramètres
78    // Initialise l'appel récursif
79    chercheRec(0,  // On commence le remplissage au rang 0
80               Array.fill(len)(false) // Cette écriture crée un tableau
81                                      // dans lequel la valeur 'false'
82                                      //est répétée 'len' fois.
83    ) // fin des paramètres de chercheRec. La récursion est lancée.
84    // ENDKILL
85
86    // Affiche la meilleure solution trouvée
87    println
88    print("Meilleure solution trouvée:")
89    for (i <- 0 to meilleure.length-1)
90      print(" "+poids(i)+":"+(if (meilleure(i)) "O" else "N")+";")
91    println(" Valeur:"+valeurTotale(meilleure,poids)+" (la capacité était "+capacite+")")
92
93
94    // BEGINKILL TODO: Definissez ici l'appel récursif à proprement parler
95    // Défini l'appel récursif
96    def chercheRec(profondeur:Int, courante:Array[Boolean]) {
97
98      val valeur = valeurTotale(courante,poids)
99
100     print(" (prof="+profondeur+") Explore ")
101     affiche(courante,profondeur)
102     print(" Valeur: "+valeur)
103
104     if (valeur > capacite) {
105       println(" *** Oups, ca deborde (backtrack!) ***")
106       return
107     }
108     if (valeurTotale(courante, poids) > valeurTotale(meilleure, poids)) {
109       dupplique(courante,meilleure)
110       print(" Nouvelle meilleure solution ");
111     } else {
112       print("   ")
113     }
114     if (profondeur == poids.length) {
115       println("(Cas terminal)");
116       return;
117     } else {
118       System.out.println("(Cas général)");
119     }
120
121
122     /* Prend l'objet et récurse */
123     prendObjet(courante,profondeur);
124     chercheRec(profondeur+1, courante);
125
126     /* Pose l'objet et récurse */
127     poseObjet(courante, profondeur);
128     chercheRec(profondeur+1, courante);
129
130   }
131   // ENDKILL
132 } // Fin de la fonction cherche() principale
133
134
135
136 // Le code de test, qui appelle la fonction publique
137 ////////////////////////////////////////////////
138 /* KILLLINE */def main(args:Array[String]) {
139 /* KILLLINE */  if (args.length == 0) {
140 /* KILLLINE */    println("Usage: knapsack <capacite> <obj1> <obj2> ... <objN>")
141 /* KILLLINE */    println("Arguments par défaut: 10 5 3 2")
142 cherche(Array(5,4,3,2), 10)
143 /* KILLLINE */
```

```
144 /* KILLLINE */  } else {
145 /* KILLLINE */    val capa = args(0).toInt
146 /* KILLLINE */    val objets:Array[Int] = new Array(args.length-1)
147 /* KILLLINE */    for (i <- 0 to objets.length-1)
148 /* KILLLINE */      objets(i) = args(i+1).toInt
149 /* KILLLINE */    cherche(objets, capa)
150 /* KILLLINE */  }
151 /* KILLLINE */}
152 /* KILLLINE */
153 /* KILLLINE */}
```

Fin réponse

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

Réponse

Certains étudiants vérifient la validité de leur travail en comptant le nombre d'appels récursifs effectués. Bonne idée. Le bon résultat pour l'instance {5,4,3,3} (capacité 10) est 15. Pas le nombre de feuilles de l'arbre, mais le nombre de nœuds internes...

Fin réponse

▷ **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.

Réponse

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

Fin réponse

▷ **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.