

Problème. Rendu de monnaie (Algorithmes gloutons, récursivité, programmation dynamique)

On dispose d'une somme $s \in \mathbb{N}^*$ d'argent à rendre, et d'une liste lv de valeurs possibles de pièces. Par exemple, pour les euros, la liste des valeurs serait $lv = [1, 2, 5, 10, 20, 50, 100, 200]$ (en cents, et en ne prenant pas en compte les billets). On appelle une telle liste un *système de pièces*, et on la supposera toujours triée par ordre croissant, et que son premier élément est toujours 1.

On souhaite calculer le nombre minimal de pièce permettant de rendre la somme s en utilisant un système de pièces donné. On suppose disposer en quantité illimitée de chaque type de pièces.

Par exemple, on peut rendre 290 en 4 pièces au minimum (à l'aide d'une pièce de 200, d'une de 50, et de deux de 20).

Stratégie gloutonne

On se propose de tenter une stratégie *gloutonne* (il s'agit d'une stratégie essayant d'effectuer le «meilleur choix» à chaque étape) pour trouver le nombre minimal de pièces permettant de rendre la monnaie pour une somme s .

Pour cela, on va systématiquement rendre la pièce la plus grande possible, la déduire de s , puis recommencer jusqu'à ce que s vaille 0.

1. Indiquer (en le calculant à la main) le résultat de cette stratégie pour $lv = [1, 2, 5, 10, 20, 50, 100, 200]$ et $s = 532$.
2. Écrire une fonction `plus_grande_piece(lv, s)` qui prend en argument un système de pièces (nécessairement trié par ordre croissant), et renvoie la plus grande valeur de pièce inférieure ou égale à s (supposé strictement positif). Quelle est sa complexité?
3. Écrire une fonction `monnaie_glouton(lv, s)` qui renvoie un entier indiquant le nombre de pièce utilisée pour rendre la somme s et la liste des pièces utilisées, à l'aide de la stratégie décrite ci-dessus. Tester votre fonction avec les données de la question 1).
4. Justifier que l'algorithme termine.
5. L'algorithme est-il correct, c'est-à-dire renvoie-t-il le nombre minimal de pièces que l'on peut utiliser? (On pourra considérer $lv = [1, 3, 4]$ et $s = 6$.)

Un algorithme récursif exact

On souhaite implémenter dans cette partie un algorithme récursif correct, basé sur les propriétés suivantes. Notons, pour une somme s , n_s le nombre minimal de pièces nécessaires pour rendre s .

Alors :

$$n_0 = 0; \quad \text{pour tout } s > 0, n_s = 1 + \min_{\substack{p \in lv \\ p \leq s}} (n_{s-p}).$$

6. Sur l'exemple de la question 5), écrire n_6 à l'aide de la formule de récurrence.
7. Écrire une fonction récursive `monnaie_recursive(lv, s)` qui renvoie n_s , le nombre de pièces utilisées pour rendre la somme s à l'aide de la stratégie décrite ci-dessus.

```
def monnaie_recursive(lv, s):
    if .....:      #Cas de base
        .....
    else:
        #on calcule le minimum
        m=..... #initialisation du minimum

        .....
        ....#autant de lignes que nécessaires
        .....

    return .....
```

8. Prouver qu'elle termine.

9. Modifier votre fonction afin qu'elle affiche la valeur de s au début de son exécution, et tester pour différentes valeurs de s . Quel est l'inconvénient de `monnaie_recursive`?

On parle de *chevauchement de sous-problèmes*.

Récurtivité avec mémoïsation de haut en bas

Afin d'y remédier, on se propose de stocker les résultats obtenus dans un dictionnaire, de façon à ne pas avoir à les recalculer.

Pour cela, afin de calculer n_s , on commencera par créer un dictionnaire vide que l'on remplit dès qu'une valeur est calculée; Les clés seront les sommes s les valeurs sont les valeurs n_s .

Lorsqu'on doit calculer n_k , on commence par regarder si cette valeur a déjà été calculée en testant la présence de k dans le dictionnaire. Sinon, on la calcule en utilisant la méthode récursive ci-dessus, et, avant de la renvoyer, on la stocke dans le dictionnaire.

10. Écrire une fonction `monnaie_memoisation1(lv, s)`, qui prend en argument `lv` et `s` comme précédemment et qui réalise ce travail :
- on définira un dictionnaire destiné à contenir les valeurs de n_s : les clés sont les sommes s et les valeurs les entiers n_s .
 - on écrira une fonction récursive interne dont le dictionnaire ci-dessus est une variable globale.

Version en programmation dynamique de bas en haut

Dans cette question on donne une version itérative de la fonction en tabulant les valeurs n_s dans un tableau `T` où `T[s]` contient la valeur n_s .

11. Écrire une fonction `monnaie_dynamique(lv, s)` qui renvoie le résultat voulu. Quelle est sa complexité?

Exercice. Hachage par division Dans cet exercice, nous manipulerons des dictionnaires sous forme de listes Python avec des tables de hachage. La fonction de hachage utilisée étant la division par un entier m . On place à l'indice k la liste des couples (c, v) tels que le reste de c par m est égal à k . Par exemple, le dictionnaire `d` (en notation Python) égal à

$$\{2:3, 4:10, 0:2, 9:11, 12:13, 10:3\}$$

correspondra à la liste

$$[[(4,10), (0,2), (12,13)], [(9,11)], [(2,3), (10,3)], []]$$

Ici, on a choisi $m = 4$.

Dans ce qui suit, quand on parlera de dictionnaire, il s'agira d'une telle liste.

1. Écrire une fonction `presCouple(l, c)` prenant en argument une liste `l` de couples (c_i, v_i) , un entier `c` et renvoyant un booléen exprimant l'existence de i tel que $c = c_i$.
2. Écrire une fonction `present(dic, c)` prenant un dictionnaire `dic` et un entier `c` et renvoyant un booléen exprimant la présence de c dans le dictionnaire.
Que peut-on dire de la complexité de cette fonction?
3. Écrire une fonction renvoyant la valeur associée à une clef dans un dictionnaire.
4. Écrire une fonction prenant en argument un dictionnaire et renvoyant le nombre d'éléments.
5. Écrire une fonction ajoutant un couple (c, v) à un dictionnaire. On supposera (sans avoir à le vérifier) que la clef n'est pas déjà présente).

Exercice 2. Coût d'un chemin

```
      7
     2 1
    1 1 3
   3 2 8 2
  5 1 6 2 1
```

On considère une pyramide de nombres comme celle de l'exemple ci-dessus. En partant de celui du haut (ici 7), le but est de trouver le chemin vers le bas qui maximise la somme de tous les nombres parcourus. Chaque entier peut être suivi d'une des deux cases inférieures. Dans cet exemple, on peut espérer obtenir : $25 = 7 + 1 + 3 + 8 + 6$

Pour représenter la pyramide précédente, on utilise une liste de listes :

```
P = [
    [7],
    [2, 1],
    [1, 1, 3],
    [3, 2, 8, 2],
    [5, 1, 6, 2, 1]
]
```

On peut aussi créer une pyramide aléatoire de n lignes, avec la commande :

```
from random import randint
P = [ [ randint (0,9) for j in range(i) ] for i in range(1,n+1) ]
```

On cherche à écrire une fonction prenant comme argument la liste P , une ligne i et une colonne j , et renvoyant la somme maximale vers le bas de la pyramide à partir du nombre d'indices i et j .

- 1) Questions préliminaires sur la liste de listes P .
 - a- Quelle valeur renvoie $P[3][1]$?
 - b- Quels sont les nombres situés en dessous de $P[i][j]$?
- 2) Que doit renvoyer la fonction dans le cas où i correspond à la dernière ligne ?
- 3) À partir de la case (i, j) , quelles sont les deux possibilités à envisager? Laquelle faut-il choisir? Exprimer récursivement ce que doit renvoyer la fonction.
- 4) Écrire la fonction récursive `somme_rec(P, i, j)` qui renvoie la somme maximale vers le bas de la pyramide à partir du nombre d'indices i et j .
- 5) Comment obtenir la somme maximale de la pyramide?
- 6) Évaluer la complexité de la fonction précédente.

On cherche à présent à mettre en place une programmation dynamique. Pour cela, il faudra mémoriser les résultats intermédiaires à l'aide d'un dictionnaire dont la clé est un couple des deux indices du nombre correspondant et la valeur la somme maximale à partir de ce nombre.

- 7) Modifier la fonction précédente afin d'écrire sa version dynamique `somme_mem(P,i,j,dico={})`.

Problème 2. Occupation du sol (tableaux à deux dimensions, complexité, programmation dynamique)

On considère dans cet exercice un terrain de côté n mètres représenté de façon discrète par une matrice de taille $n \times n$ de 1 et de 0:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

1 signifie qu'il y a un arbre dans le m^2 correspondant, et 0 que le terrain y est vide.

On souhaite construire une maison de base carrée la plus grande possible, sans couper d'arbre. Par exemple, sur le terrain ci-dessus, le plus grand côté possible de la maison sera de 3 mètres.

1. (a) Écrire une fonction `carre_vide(t, i, j, c)` prenant en argument un terrain t , des coordonnées i et j correspondant à un emplacement du terrain (par convention, $(0,0)$ est en haut à gauche, et i est l'indice de ligne), et $c \in \llbracket 1, n \rrbracket$ et renvoie `True` si le carré de côté c dont le coin en haut à gauche est en (i, j) dans le terrain t est constructible. On ne vérifiera pas la validité des paramètres. Quelle est sa complexité en fonction de n ?
- (b) En déduire une fonction `plus_grand_carre_vide(t, i, j)` qui détermine le côté du plus grand carré vide dans le terrain dont le coin en haut à gauche est en (i, j) . Quelle est sa complexité?
- (c) En déduire une fonction `cote_maison_naif(t)` prenant en argument un terrain t (représenté par une liste de listes), et renvoyant le plus grand côté possible d'une maison qui y est constructible. Quelle est sa complexité en fonction de n ?
- (d) En utilisant une dichotomie, proposer un algorithme (qu'il n'est pas demandé de coder (dans un premier temps) transformant l'un des n de la complexité précédente en un $\log(n)$). Cela reste peu satisfaisant.

Une idée de programmation dynamique va permettre d'améliorer la complexité obtenue.

2. Pour tout couple d'indices $(i, j) \in \llbracket 0, n-1 \rrbracket^2$, on note $s_{i,j}$ le nombre de 1 contenus dans la partie rectangulaire du terrain délimitée par les cases d'indice $(0,0)$ et (i, j) .
 - (a) On cherche à calculer $s_{i,j}$ pour tout $(i, j) \in \llbracket 0, n-1 \rrbracket^2$.
 - i. Comment calculer les valeurs de $s_{i,j}$ sur la première ligne et la première colonne?
 - ii. Pour tout (i, j) tel que $i \neq 0$ et $j \neq 0$, exprimer $s_{i,j}$ en fonction de $s_{i-1,j}$, $s_{i,j-1}$, $s_{i-1,j-1}$ et $t[i][j]$.
 - iii. En utilisant les deux questions précédentes, écrire une fonction `carte_s(t)` de complexité $O(n^2)$ qui prend en argument un terrain t et renvoie la matrice des $(s_{i,j})$ correspondante. Quelle est sa complexité?
 - (b) Écrire une fonction `plus_grand_carre_vide_bis(s, i, j)` qui renvoie, en complexité au plus linéaire, le plus grand côté d'un carré vide dont le coin supérieur gauche est en (i, j) .

Indication: à partir de s , il est possible de calculer *en temps constant* le nombre de 1 dans n'importe quel carré!

Quelle est sa complexité?
 - (c) En déduire un algorithme plus efficace que l'algorithme naïf pour répondre à la question de départ. Il n'est pas demandé de l'implémenter. Quelle est sa complexité?