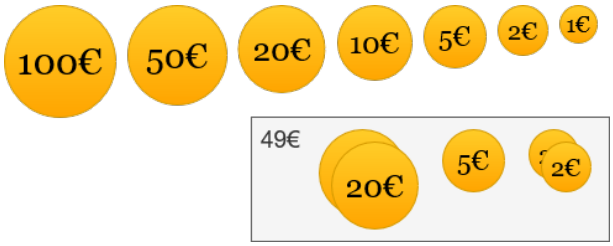
Programmation dynamique

Objectifs

Comprendre le principe de la programmation dynamique et de la mémoïsation.

Rendu de monnaie



Méthode gloutonne

Vous avez à votre disposition un nombre illimité de pièces de 2 cts, 5 cts, 10 cts, 50 cts et 100 cts (1 euro). Vous devez rendre une certaine somme. Le problème est le suivant : « Quel est le nombre minimum de pièces qui doivent être utilisées pour rendre la monnaie ».

La résolution « gloutonne » de ce problème peut être la suivante :

- on prend la pièce qui a la plus grande valeur (il faut que la valeur de cette pièce soit inférieure ou égale à la somme restant à rendre)
- on recommence l'opération jusqu'au moment où la somme à rendre est égale à zéro.

1. Que donne cette résolution pour une somme à rendre de 11 cts ?
2. Existe-t-il une solution exacte pour cet exemple ?
3. La résolution gloutonne donne-t-elle une solution ? Une solution optimale ?
4. Toujours sur l'exemple des 11 cts, représenter sur un brouillon un arbre de décisions ou chaque nœud correspond à la somme restante à rendre, le nœud racine correspondant à la somme initiale à rendre et où le poids des arrêtes correspond à la pièce rendue. Combien de feuilles possède cet arbre ? Combien de solutions exactes existe-t-il ? Comment en déduire le nombre de pièce utilisée ? Quelle est la solution optimale ?

Cet exemple marque une caractéristique importante des algorithmes gloutons : une fois qu'une « décision » a été prise, on ne revient pas « en arrière » (on a choisi la pièce de 10 cts, même si cela nous conduit dans une « impasse »). On retiendra qu'un algorithme glouton ne donne pas nécessairement la solution optimale.

Vous avez maintenant à votre disposition un nombre illimité de pièces (ou billets) de 100, 50, 20, 10, 5, 2 et 1 euros. Vous devez rendre une certaine `somme`. Le problème est le suivant : « Quel est le nombre minimum de pièces qui doivent être utilisées pour rendre la monnaie ».

5. Ecrire une fonction `rendu_glouton(systeme,somme)` qui prends en argument une liste `systeme` de pièces (euros ici), un entier `somme` à rendre et qui retourne la liste `monnaie` des pièces et billets à rendre par la méthode gloutonne. La tester pour le système euros avec `somme=16`.

La réponse optimale est celle qui utilise le nombre minimal de pièces. On a vu que l'algorithme glouton échoue à la trouver en général.

Méthode récursive

Une approche récursive permet de résoudre le problème : soit x une valeur faciale de l'une des pièces du système monétaire. Pour rendre une somme s de façon optimale, si l'on veut utiliser au moins une fois la pièce x, il suffit de rendre x et la somme $s-x$ de façon optimale. Il n'y a plus qu'à choisir, parmi tous les choix possibles de x, celui qui permet d'utiliser le minimum de pièces. Autrement dit, si on appelle $f(s)$ le nombre minimal de pièces et billets qu'il faut utiliser pour payer la somme x, on a simplement :

$$\begin{cases} f(0) = & 0 \\ f(s) = & \min_{x \leq s}(1 + f(s-x)) \end{cases}$$

le minimum étant calculé sur toutes les valeurs x d'une pièce.

6. A partir du pseudo-code en annexe 1, écrire une fonction `rendu_recursif(systeme,somme)` qui prends en argument une liste `systeme` de billets (euros ici), un entier `somme` à rendre et qui retourne le `mini` de pièces et billets à rendre. On importera `inf` du module `math` (quel que soit `a`, `a<inf` est `True`).

Cette solution admet un problème de taille, son appel est extrêmement lent car les mêmes calculs sont effectués de manière répétée, la complexité est exponentielle.

Méthode avec mémoïsation

Une idée classique consiste à mémoriser les résultats des appels pour être sûr qu'on n'aura pas besoin de les calculer plusieurs fois. Ici, le calcul de $f(s)$ utilise le calcul de $f(s-x)$ pour chaque valeur de x. Autrement dit, $f(s)$ n'utilise au plus que les valeurs de $f(s-1)$, $f(s-2)$, $\ldots$, $f(3)$, $f(2)$, $f(1)$. On va donc créer un tableau conservant ces données, et calculer de façon systématique pour des valeurs croissantes de l'index. Cette technique est habituellement appelée mémoïsation.

7. Ecrire une fonction `rendu_memoise(systeme,somme)` qui prends en argument une liste `systeme` de pièces (euros ici), un entier `somme` à rendre et qui retourne le `mini` de pièces et billets à rendre c'est-à-dire `f[somme]`. Cette fonction commencera par la création d'un tableau (une liste) `f` contenant `somme+1` zéros. La tester pour le système euros avec `somme=16`. On pourra s'aider du code 2 en annexes.

8. Remplacer la dernière ligne de votre fonction afin de retourner `f` au lieu de `f[somme]`. Que retourne alors cette version ? Vérifier que :

```
>>> rendu_memoise(euros,16)
>>> [0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 1, 2, 2, 3, 3, 2, 3]
```

9. Que représente le dernier élément de cette liste ?

10. On a $f(16) = 3$, on cherche une pièce x (appartenant au système monétaire) telle que $f(16-x) = 2$. Quelles sont les possibilités pour x ? Poursuivre les étapes d'analyse.

11. Dresser l'arbre de cet algorithme. Vérifier que le nombre minimum de pièces à rendre est bien égal à 3 dans le cas d'une somme à rendre de 16.

12. Commenter la fonction `rendu_memoise_reconstitue` en annexe 2. L'implémenter et la tester pour le système euros avec `somme=16`. Vérifier que le résultat affiché est en accord avec votre arbre.

Le problème du sac à dos

Un voleur pénètre dans la maison d'un physicien. Les objets les plus intéressants sont un porte-clés de haute valeur sentimentale, un livre de physique d'un célèbre auteur, des parchemins contenant de nombreux schémas de physique de qualité ainsi qu'une calculatrice standard. Comme il doit rester léger pour escalader la façade de l'immeuble, il n'a pris qu'un petit sac à dos pouvant contenir au maximum 8 kg. Le tableau suivant donne le poids et la valeur :

objet	poids (kg)	valeur (x100 €)
porte-clés	1	15
livre de physique	5	10
parchemins de schémas	3	9
calculatrice	4	5

Le voleur s'interroge sur les objets qu'il doit mettre dans son sac. On a trois stratégies :

— Le voleur est « glouton » : l'algorithme consiste à prendre en priorité les objets de plus grande valeur. Quels objets prendra-t-il s'il suit cette méthode ? Le choix est-il optimal ?

— Le voleur a le temps : le voleur décide de calculer toutes les combinaisons d'objet possibles. Il lui faut 5 s pour analyser chaque possibilité. Combien de temps lui faudra-t-il ?

— Le voleur est informaticien : il utilise un algorithme de programmation dynamique pour calculer la solution optimale. Traiter la suite du problème...

Notations

On considère un sac à dos dont la capacité en poids est notée P. On considère N objets dont on note $p_0, \ldots, p_{N-1}$ les poids et $v_0, \ldots, v_{N-1}$ les valeurs. On supposera que les poids et valeurs sont des entiers strictement positifs. Ces données seront implémentées sous la forme de tableaux : la liste `poids` et la liste `valeurs`. Ainsi le but est de choisir un sous-ensemble X de $[0, N[$ tel que $\sum_i \in X p_i \leq P$ et pour lequel la valeur totale $\sum_{i \in X} v_i$ soit maximale.

Programmation dynamique du problème

Pour tout P et tout $i \in [0, N[$, on pose $V(i, P)$ la valeur maximale des objets qu'on peut mettre dans un sac de capacité P en choisissant uniquement des objets parmi les i premiers (soit dans $[0, i[$). Si $P \leq 0$, nous convenons que $V(i, P) = 0$.

13. Que vaut $V(i, P)$ lorsque $P = 0$ ou $i = 0$?

14. Démontrer que pour tout P entier positif et $i \in [0, N[$:

$$V(i+1, P) = \max(V(i, P), V(i, P - p_i) + v_i)$$

15. A partir du pseudo-code en annexe 3, écrire `SacADos(poids, valeurs, capacite)` la fonction qui prends en argument une liste `poids` des poids des objets, une liste `valeurs` des valeurs des objets, un entier `capacite` correspondant au poids P maximal transportable par le sac et retournant la valeur maximale `V(N-1, P)`. Vérifier que dans notre exemple la fonction retourne 29. Faire ce code sur papier et remplir le tableau à la main.

16. Que représente les lignes du tableau `V` ?

17. Le programme précédent ne calcule que la valeur maximale des objets qu'on peut mettre dans le sa à dos, et non la liste de ces objets. Une méthode consiste à retrouver cette liste après coup en lisant le tableau `V` obtenu. L'idée est que si `V[i][p]=V[i-1][p-p_{i-1}]+v_i` c'est que l'objet `i-1` a été pris. Programmer `solutionSac(V, poids, valeurs)` la fonction prenant en entrée le tableau `V` rempli et renvoyant la liste des numéros des objets à prendre.

Annexes

Algorithm 1: pseudo-code 1

```
def rendu_recursif(systeme,somme):
    if somme<0 then
        return infini
    else if somme=0 then
        return 0
    minimum ← infini
    for x ∈ systeme do
        if somme >=0 then
            minimum ← min(minimum, 1 + rendu_recursif(systeme, somme − x))
    return minimum
```

Algorithm 2: code 2

```python
def rendu_memoise_reconstitue(systeme, somme):
        f = [0] * (somme + 1)
        g = [0] * (somme + 1)
        for s in range(1, somme + 1):
                f[s] = inf
                for x in systeme:
                        if s >= x:
                                if 1 + f[s - x] < f[s]:
                                        f[s] = 1 + f[s - x] # MAJ du minimum
                                        g[s] = s - x        # on retient d'où
                                         ↪ l'on vient
        monnaie = []
        while somme > 0:
                monnaie.append(somme - g[somme])
                somme = g[somme]
        return monnaie
```

Algorithm 3: pseudo-code 3

```
def SacADos(poids,valeurs,capacite):
    N ← taille(poids)
    V ← 0[N][capacite + 1]
    for i ← 1 to N + 1 do
        for p ← 1 to capacite + 1 do
            if poids[i-1]<=p then
                V[i][p] ← max(V[i − 1][p], valeurs[i − 1] + V[i − 1][p − poids[i − 1]])
            else
                V[i][p] = V[i − 1][p]
    return V[N][capacite] (A MODIFIER PAR V POUR 17.)
```

Bon courage et bon travail ! ☺