

TD3 - Programmation dynamique

VERSION AVEC SOLUTION

La *programmation dynamique* est une technique algorithmique qui consiste à calculer la solution pour des sous-problèmes, et où la solution du sous-problème suivant dépend directement de un ou plusieurs sous-problèmes précédemment calculés.

Souvent, c'est le moyen de rendre itératif un algorithme récursif, en passant d'une complexité exponentielle à une complexité polynomiale.

Les sous-problèmes sont souvent stockés dans une tableau, ou bien, on peut aussi faire de la programmation dynamique sur des structures arborescentes ou des graphes orientés acycliques.

Exercice 1 (Rendu de monnaie).

On a une liste de dénominations de pièces et billets, par exemple $L = [1, 2, 5, 10, 20, 50, 100, 200, 500]$. Étant donné un entier n de monnaie à rendre en euros, on veut savoir quelle est la meilleure combinaison de pièces/billets (on souhaite minimiser le nombre total de pièces/billets à rendre).

On pourrait proposer un algorithme glouton pour ce problème : on choisit la plus grande dénomination d qui ne dépasse pas n , et on continue avec $n-d$ jusqu'à arriver à 0. Cependant, cet algorithme n'est pas optimal : par exemple pour $L = [5, 10, 20, 25]$ et $n=40$, cet algorithme va renvoyer 3 ($25 + 10 + 5$) alors que l'optimum est 2 ($20 + 20$). Il faut donc trouver une méthode plus exhaustive...

On note $s[i]$ la solution pour la somme i (plus petit nombre de pièces/billets dont la somme est i).

1. Exprimer $s[i]$ selon deux cas : (1) i est dans la liste L des dénominations, ou sinon (2), en fonction de $s[i - d]$ pour tous les d dans L et $d < i$.
2. En déduire un algorithme récursif `MonnaieRec(n,L)` pour calculer la valeur optimale.
3. Estimer de façon grossière la complexité de l'algorithme récursif en fonction de n et de la taille $|L|$ de L en analysant l'arbre des appels récursifs.
4. Le transformer en algorithme itératif `MonnaieProgDyn(n,L)` utilisant le principe de programmation dynamique.
5. Quelle est sa complexité approximative en temps en fonction de n et de la taille $|L|$ de L ? Comme $N = n + |L|$ est la taille de l'entrée, quelle est la complexité en fonction de N ?
6. Quelle modification peut-on faire pour aussi calculer la combinaison optimale (pas juste la valeur)?

Mise en pratique.

1. Coder une première solution récursive `MonnaieRec(n,L)` et tester (on s'intéresse tout d'abord au nombre optimal de pièces/billets, pas forcément à la combinaison exacte : par exemple pour $n=12$ la solution est 2 : un billet de 10 et une pièce de 2).

Jusqu'à quelle valeur cet algorithme est-il capable de calculer la solution rapidement ($<30\text{sec}$)? Vous pouvez tester avec 36 et la liste $L = [1, 2, 5, 10, 20, 50, 100, 200, 500]$ et trouver 4. Il est aussi intéressant de tester avec $n = 10$ et $L = [1, 5, 7]$ et de bien trouver 2.

2. Coder la version itérative `MonnaieProgDyn(n,L)` utilisant la programmation dynamique. Tester sur de grandes valeurs de n .
3. Modifier l'algorithme pour qu'il affiche également une solution (une combinaison optimale de pièces/billets). La solution pourra (par exemple) être stockée comme un dictionnaire dont les clés sont les éléments de L et les valeurs, le nombre de pièces/billets de chaque dénomination. Pour cela, il faut "backtracker" : partir de la valeur de la solution finale et déterminer pas à pas, à l'envers, quelles étapes ont été utilisées pour y arriver.

Solution.

1. Si i est dans la liste L des dénominations, on a $s[i] = 1$.

Sinon, $s[i] = \min_{d \in L \text{ et } d < i} \{s[i - d] + 1\}$

2.

```
MonnaieRec(n,L):
Entrée : un entier n, une liste L d'entiers
Sortie : le nombre minimum de pièces/billets de L dont la somme est n

  * Pour chaque d dans L:
    - Si d == n:
      Retourner 1

  * min = n
  * Pour chaque d dans L tel que d < n:
    - val = MonnaieRec(n-d,L) + 1
    - Si val < min:
      min = val

  * Retourner min
```

3. On a un arbre des appels récursifs où chaque noeud a potentiellement $|L|$ enfants et la hauteur est au plus n , donc au plus $\sum_{i=0}^n |L|^i \leq |L|^{n+1}$ noeuds au total. (Chaque arête de l'arbre représente un appel récursif.) À chaque noeud, on fait $O(|L|)$ opérations, cela donne donc une complexité grossière en $O(|L|^{n+2})$ ce qui est fortement exponentiel.

4.

```
MonnaieProgDyn(n,L):
Entrée : un entier n, une liste L d'entiers
Sortie : le nombre minimum de pièces/billets de L dont la somme est n

  • Initialiser un tableau des valeurs optimales s[1..n]
  • Pour i allant de 1 à n:
    - Si i est dans L:
      s[i] = 1
    - Sinon:
      * min = +∞
      * Pour chaque d dans L tel que d < i:
        - val = s[i-d]
        - Si val < min + 1:
          min = val
      * s[i] = min
  • Retourner s[n]
```

5. $\Theta(n \times |L|)$ car pour chaque valeur de i on fait une boucle sur L . Si $N = n + |L|$ est la taille de l'entrée, cela donne $\Theta(N^2)$.
6. On peut faire du "backtracking", en se souvenant pour chaque $s[i]$ quelle était la dernière valeur de d pour laquelle $s[i] = s[i - d] + 1$. De fil en aiguille on peut ainsi reconstruire la solution optimale.

Voici le code Python de l'implémentation des deux algos :

```

1 import sys
2
3 L=[1,2,5,10,20,50,100,200,500]
4
5 def MonnaieRec(n,L):
6
7     #cas de base
8     if n in L:
9         return 1
10
11     dict_valeurs = {}
12
13     #calculs ré cursifs
14     for d in L:
15         if n-d > 0:
16             dict_valeurs[d] = MonnaieRec(n-d,L)
17
18     valeur_min = sys.maxsize
19     elt_min = 0
20
21     #trouver le minimum parmi les valeurs calculées
22     for d in dict_valeurs:
23         if dict_valeurs[d] < valeur_min:
24             valeur_min = dict_valeurs[d]
25             elt_min = d
26
27     return valeur_min + 1
28
29 print(MonnaieRec(36,L)) #ça commence à prendre du temps!
30
31 def MonnaieProgDyn(n,L):
32     #tableau de solutions
33     valeurs = (n+1)*[sys.maxsize]
34     valeurs[0] = 1
35
36     for i in range (1,n+1):
37         #si on peut rendre un seul billet /pièce
38         if i in L:
39             valeurs[i] = 1
40         else:
41             #on cherche la meilleure possibilité
42             #pour la dernière pièce/ billet
43             valeur_min = sys.maxsize
44             for d in L:
45                 if i-d > 0 and valeur_min > valeurs[i-d]:
46                     valeur_min = valeurs[i-d]
47             valeurs[i] = valeur_min + 1
48
49     return valeurs[i]
50
51 print(MonnaieProgDyn(176854,L))
52
53 def MonnaieProgDynAvecCombinaison(n,L):
54     valeurs = (n+1)*[sys.maxsize]
55     valeurs[0] = 1
56
57     for i in range (1,n+1):
58         if i in L:
59             valeurs[i] = 1
60         else:
61             valeur_min = sys.maxsize

```

```

62         for d in L:
63             if i-d > 0 and valeur_min > valeurs[i-d]:
64                 valeur_min = valeurs[i-d]
65             valeurs[i] = valeur_min + 1
66
67         # -- PARTIE NOUVELLE POUR LE BACKTRACKING --
68         # initialiser dico de solutions à 0
69         # pour chaque dénomination, on va compter
70         # le nombre qu'il faut
71         dict_solution = {}
72         for d in L:
73             dict_solution[d] = 0
74
75         #backtracking:
76         j = n
77         while j > 0:
78             if j in L:
79                 dict_solution[j] += 1
80                 j = 0
81             else:
82                 valeur_j = valeurs[j]
83                 #on cherche la piece / billet qui permet d'obtenir la valeur précédente
84                 for d in L:
85                     if j-d >= 0 and valeurs[j-d] == valeur_j - 1:
86                         j = j-d
87                         dict_solution[d] += 1
88                         break
89
90         print(dict_solution)
91         return valeurs[n]
92
93 print(MonnaieProgDynAvecCombinaison(655338,L))

```

Listing 1 – Rendu de monnaie

Exercice 2 (Répartition de médicaments).

Une ONG dispose d'un stock de denrées (médicaments) en quantité n . Quand l'ONG distribue une quantité i de denrées à un endroit, l'expérience montre que cela sauve $S[i]$ personnes. Le tableau S a été établi par des experts et mis à disposition à l'ONG. On cherche à calculer le nombre maximal de personnes que l'on pourra sauver avec le stock en le découpant en sous-parties. Par exemple, avec le tableau $S = [1, 3, 2]$ (c'est-à-dire $S[1] = 1$, $S[2] = 3$, $S[3] = 2$) et $n = 3$, on peut sauver :

- soit $1 + 1 + 1 = 3$ personnes en coupant le stock en 3 parties de taille 1 ;
- ou bien, 2 personnes en laissant le stock en une seule partie de taille 3 ;
- ou bien, $1 + 3 = 4$ personnes en coupant le stock en 1 partie de taille 1 et 1 partie de taille 2.

1. On note $m[i]$ la meilleure solution (nombre de personnes sauvées) pour un stock de taille i . Exprimer $m[i]$ récursivement, en fonction des valeurs $m[j]$ avec $j < i$, en utilisant aussi S lorsque nécessaire.
2. En déduire un algorithme récursif `RepartitionRec(n,S)` pour calculer la valeur optimale pour une valeur de n et un tableau $S[1..n]$ donné.
3. Quelle est la classe de complexité (approximative) en temps de cet algorithme ?
4. Donner un algorithme itératif `RepartitionProgDyn(n,S)` en utilisant le principe de la programmation dynamique.
5. Quelle est sa complexité asymptotique en temps ?

Solution.

1. $m[i] = \max_{0 \leq j < i} \{m[i-j] + S[j]\}$ (on prend une partie de taille j)

Autre formulation possible : $m[i] = \max \left\{ S[i], \max_{0 < j < i} \{m[j] + S[i-j]\} \right\}$ (soit on prend tout le stock, soit on prend une partie de taille $i-j$)

2. Avec la première formulation :

```
RepartitionRec(n,S):  
Entrée : un entier n, une liste S d'entiers  
Sortie : le nombre maximum de personnes qui peuvent être sauvées avec  
le stock de taille n  
  
  * max = S[n]  
  * Pour chaque j < n:  
    - val = RepartitionRec(n-j)+S[j]  
    - Si val > max:  
      max = val  
  
  * Retourner max
```

Avec la deuxième formulation :

```
RepartitionRec(n,S):  
Entrée : un entier n, une liste S d'entiers  
Sortie : le nombre maximum de personnes qui peuvent être sauvées avec  
le stock de taille n  
  
  * max = S[n]  
  * Pour chaque j < n:  
    - val = RepartitionRec(j)+S[n-j]  
    - Si val > max:  
      max = val  
  
  * Retourner max
```

3. Complexité exponentielle, car on a un arbre des appels récursifs où chaque noeud a potentiellement n enfants et la hauteur est d'ordre n , donc n^n noeuds au total. (Chaque arête représente un appel récursif.)
4. Première formulation :

```
RepartitionProgDyn(n,S):  
Entrée : un entier n, une liste S d'entiers  
Sortie : le nombre maximum de personnes qui peuvent être sauvées avec  
le stock de taille n  
  
  • Initialiser un tableau des valeurs optimales m[1..n] rempli de 0  
  • Pour i allant de 1 à n:  
    * max = S[i]  
    * Pour j allant de 1 à i-1:  
      - val = m[i-j] + S[j]  
      - Si val > max:  
        max = val  
    * m[i] = max  
  
  • Retourner m[n]
```

5. $\Theta(n^2)$ car pour chaque valeur de i entre 1 et n on fait une boucle sur $1\dots n$.

Exercice 3 (Problème du sac à dos (knapsack)).

On dispose d'un container avec une capacité (volume) limitée (CAPACITE), et on veut charger des objets dans le container. Chaque objet a un volume et une utilité donnés. Ceux-ci sont listés comme des couples (volume,utilité). Les volumes sont en m^3 , et l'utilité est un entier (qui représente par exemple des milliers d'euros, ou des milliers de vies sauvées). Par exemple :

$A = [(1,3), (2,4), (4,5), (8,8), (9,10), (6,6), (12,15)]$

Ici, le premier objet a un volume de $1m^3$, et une utilité de 3. Chaque objet ne peut être choisi qu'une seule fois au maximum.

On définit un tableau à 2 dimensions T tel que $T[i][j]$ contient la meilleure utilité totale pour un volume total d'au plus j , qui utilise des objets parmi les i premiers objets de A .

Par exemple, $T[0][j]=0$ pour tout j (car on veut une solution qui utilise 0 objets) et $T[i][0]=0$ pour tout i (car le volume total est de 0). L'utilité de la solution optimale se retrouvera dans la valeur de $T[\text{len}(A)][\text{CAPACITE}]$ puisqu'on aura droit au budget total et à tous les objets.

1. Donner une formule récursive permettant de calculer $T[i][j]$. Pour cela, considérer deux cas : soit on ne sélectionne pas l'objet i et on se ramène à $T[i-1][j]$, soit on le sélectionne et on se ramène aussi à un cas précédent (lequel?).
2. Écrire l'algorithme de programmation dynamique basé sur cette formule.
3. Quelle est sa complexité asymptotique ?

Mise en pratique.

1. Codez une fonction `SacADos(A,budget)` utilisant la programmation dynamique. Tester avec $A = [(1,3), (2,4), (4,5), (8,8), (9,10), (6,6), (12,15)]$.
2. Modifier le code pour qu'il affiche une solution optimale (pas seulement la valeur de l'utilité maximum, mais aussi les objets à sélectionner), son coût total et son utilité totale. Il faut "backtracker".

Solution.

1. Si $i=0$ ou $j=0$ on a $T[i][j]=0$ (en supposant que les indices des objets commencent à 1, ainsi l'indice $i=0$ correspond à aucun objet autorisé).

Sinon, avec v le volume de l'objet i et u son utilité, on a :

$$T[i][j] = \begin{cases} \max(T[i-1][j], T[i-1][j-v]+u), & \text{si } v \leq j \\ T[i-1][j], & \text{sinon.} \end{cases}$$

2. voir code ci-dessous. On crée le tableau à double entrée T puis on fait deux boucles imbriquées pour le remplir à l'aide de la formule récursive ci-dessus.
3. Complexité : on a $\text{CAPACITE} \times \text{len}(A)$ cases de tableau à remplir. Pour chacune, on fait $O(1)$ opérations. Donc au total, la complexité est $O(\text{CAPACITE} \times \text{len}(A))$.

```
1 import sys
2
3 A=[(1,3),(2,4),(4,5),(8,8),(9,10),(6,6),(12,15)]
4
5 def SacADos(A,budget):
6
7     # initialiser T
8     T = (len(A)+1)*[[]]
```

```

9     for i in range(len(T)):
10         T[i] = (budget+1)*[0]
11
12     #calculer les valeurs de T
13     for i in range (1,len(A)+1):
14         for j in range(1,budget+1):
15             T[i][j] = T[i-1][j]
16             v = A[i-1][0]
17             u = A[i-1][1]
18             if v <= j and T[i][j] < T[i-1][j-v]+u:
19                 T[i][j] = T[i-1][j-v]+u
20
21     return T[len(A)][budget]
22
23 print(SacADos(A,15))
24
25 #avec le backtrack pour calculer la solution
26 def SacADosAvecSolution(A,budget):
27
28     # initialiser T
29     T = (len(A)+1)*[[]]
30     for i in range(len(T)):
31         T[i] = (budget+1)*[0]
32
33     #calculer les valeurs de T
34     for i in range (1,len(A)+1):
35         for j in range(1,budget+1):
36             T[i][j] = T[i-1][j]
37             v = A[i-1][0]
38             u = A[i-1][1]
39             if v <= j and T[i][j] < T[i-1][j-v]+u:
40                 T[i][j] = T[i-1][j-v]+u
41
42     # -- PARTIE NOUVELLE -- BACKTRACKING --
43     objets = []
44     i = len(A)
45     j = budget
46     #on "backtracke"
47     while T[i][j] != 0:
48         if T[i-1][j] == T[i][j]: #l'objet i n'a pas été choisi
49             i = i-1
50         else: #le ième objet (indice i-1) avait été choisi
51             objets.append(i-1)
52             c = A[i-1][0] #cout du ième objet
53             j = j-c
54             i = i-1
55
56     #mise en forme et calcul du cout et utilité totales
57     objets.reverse()
58     cout = 0
59     utilite = 0
60     for i in objets:
61         cout += A[i][0]
62         utilite += A[i][1]
63
64     print("Une solution optimale est: "+str(objets)+"\npour un cout de: "+str(cout)+" et une
        utilité de: "+str(utilite))
65
66     return T[len(A)][budget]
67
68 print(SacADosAvecSolution(A,13))

```

Exercice 4 (Plus grand sous-tableau croissant).

Étant donné un tableau de N entiers, on cherche le plus grand sous-tableau d'éléments croissants (pas forcément consécutifs). Par exemple, pour le tableau $[1, 5, 3, 1, 8]$ des sous-tableaux croissants sont (entre autres) $[1, 8]$, $[1, 5, 8]$, ou encore $[1, 3, 8]$. La taille maximum d'un sous-tableau croissant de TAB est donc 3.

Pour le résoudre on va déterminer la solution optimale de façon incrémentale.

On suppose qu'on a un tableau d'entiers $TAB[1...N]$. Soit $d[i]$ la taille d'un plus long sous-tableau de TAB croissant dont le dernier élément est $TAB[i]$.

1. Exprimer $d[i]$ en fonction des valeurs $d[j]$ avec $j < i$.
2. En déduire un algorithme récursif $STCRec(TAB, N)$ pour calculer *la taille* d'un plus grand sous-tableau de TAB croissant.
3. Quelle est sa complexité asymptotique en fonction de N ?
4. Donner un algorithme itératif $STCProgDyn(TAB, N)$ utilisant la technique de la programmation dynamique.
5. Quelle est sa complexité en temps ?
6. Quelle modification peut-on faire pour aussi calculer le plus long sous-tableau croissant (pas juste sa taille) ?

Solution.

1.

$$d[i] = \begin{cases} \max_{j < i \text{ et } TAB[j] \leq TAB[i]} \{d[j] + 1\} & \text{s'il existe } j \text{ avec } j < i \text{ et } TAB[j] \leq TAB[i] \\ d[i] = 1, & \text{sinon.} \end{cases} \quad (1)$$

2.

`tailleMaxSousTableauCroissantRec(T):` taille d'un plus long sous-tableau croissant du tableau T d'entiers

Entrée : Un tableau d'entiers $TAB[1...N]$

Sortie : taille d'un plus long sous-tableau croissant de T

- Si $N == 1$:
Retourner 1
- Sinon :
 - ★ $max = 1$
 - ★ Pour i de 1 à $N - 1$:
 - $n = \text{tailleMaxSousTableauCroissantRec}(TAB[1...i]) + 1$
 - Si $TAB[i] \leq TAB[N]$ et $n > max$:
 - $max = n$
 - ★ Retourner max

3.

`tailleMaxSousTableauCroissant(TAB)`: taille d'un plus long sous-tableau croissant du tableau TAB d'entiers

Entrée : Un tableau d'entiers $TAB[1...N]$

Sortie : taille d'un plus long sous-tableau croissant de T

- Soit $d[1...N] = [1, ..., 1]$ un tableau rempli de 1
- Pour i de 1 à N :
 - ★ $max = 1$
 - ★ Pour j de 1 à $i - 1$:
 - $n = d[j] + 1$
 - Si $TAB[j] \leq TAB[i]$ et $n > max$:
 - $max = n$
 - ★ $d[i] = max$
- Retourner $\text{maximum}(d[1], ..., d[N])$

4. $\Theta(N^2)$

5. Il faut, en plus du calcul de $d[i]$, aussi conserver l'indice $p[i]$ qui précède $TAB[i]$ dans un plus long sous-tableau croissant qui finit avec $TAB[i]$, et “backtracker” (revenir sur ses pas) pour reconstruire la solution.