Numérique et science informatique

Lycée hoche

année scolaire 2020-2021

Contents

0.1 Le rendu de monnaie 2
0.1.1 Recherche de la réponse optimale 2
0.1.2 Reconstitution des détails de la réponse optimale 4
0.2 Qu'est ce que la programmation dynamique? 5
0.3 Le problème du sac à dos 5
0.3.1 Description du problème 5
0.3.2 programmation 6
0.4 Plus longue séquence commune 6
0.4.1 introduction 6
0.4.2 Résolution du problème 7
0.4.3 programmation 7

0.1 Le rendu de monnaie

Fixons les notations. On suppose donné un système monétaire où les valeurs faciales des pièces (ou des billets) sont rangées en ordre décroissant. Par exemple, le système Euro pourra être décrit par la liste euros = [50, 20, 10, 5, 2, 1].
Pour payer une somme de 48 unités on pourrait bien sûr payer 48 pièces de 1, ou encore 3 pièces de 10, 3 pièces de 5, 1 pièce de 2 et 1 pièce de 1.

On cherche à payer la somme indiquée, en supposant qu'on a autant de pièces de chaque valeur que de besoin, en utilisant un nombre minimal de pièces.

L'algorithme glouton consiste à payer d'abord avec la plus grosse pièce possible : ici, il s'agit de 20, puisque $50 > 48$. Ayant donné 20,il reste 28 à payer, et on poursuit avec la même méthode. Finalement, on va payer 48 sous la forme $48 = 20 + 20 + 5 + 2 + 1$. On a eu besoin de 5 pièces.

Considérons un autre système monétaire (en fait c'est l'ancien système impérial britannique) représenté par la liste suivante de valeurs faciales : imperial = [30, 24, 12, 6, 3, 1]. Pour payer 48 l'algorithme glouton va répondre : $48 = 30 + 12 + 6$.

À la différence du système euro, pour lequel on peut démontrer que l'algorithme glouton donne toujours la réponse optimale, on constate que ce n'est pas le cas avec le système impérial, puisque on aurait pu se contenter de 2 pièces : $48 = 24 + 24$.

Rappelons comment peut se programmer l'algorithme glouton.

```
euros = [50, 20, 10, 5, 2, 1]
imperial = [30, 24, 12, 6, 3, 1]
def glouton(valeursFaciales, somme):
    i = 0 # index de la pièce qu'on va essayer
    p = len(valeursFaciales) # nombre de valeurs de pièces disponibles
    monnaie = [] # liste des pièces rendues
    while i<p and somme>0:
        if valeursFaciales[i]>somme:
            i += 1
        else:
            monnaie.append(valeursFaciales[i])
            somme -= valeursFaciales[i]
    if somme==0:
        return monnaie
    else:
        return None
```

L'appel glouton(euros, 48) renvoie la liste [20, 20, 5, 2, 1] ; l'appel glouton(imperial, 48) renvoie la liste [30, 12, 6]

0.1.1 Recherche de la réponse optimale

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 (l'exemple de rendre 48 avec le système impérial suffit à le prouver).

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 qu'il faut utiliser pour payer la somme s, on a simplement:

- $f(0) = 0$ et
- $f(s) = \min_{x<=s}(1 + f(s-x))$

le minimum étant calculé sur toutes les valeurs x d'une pièce.
Cette remarque permet d'écrire le programme récursif suivant :

```
from math import inf

def dynRecursif(valeursFaciales, somme):
    if somme < 0:
        return inf
    elif somme == 0:
       return 0
    mini = inf
    for x in valeursFaciales:
        if somme >= x:
            mini = min(mini, 1 + dynRecursif(valeursFaciales, somme - x))
    return mini
```

On a importé du module math la valeur spéciale inf qui représente l'infini : pour tout entier a, l'expression $a < inf$ vaut True.
Hélas, l'exécution de l'appel *dynRecursif*(euros, 48) est extrêmement lente. les mêmes calculs étant effectués de façon répétée. Une analyse du problème montrerait que la complexité est exponentielle.
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 $f(s-1), f(s-2), ..., f(3), f(2), f(1)$.

On va donc créer un tableau conservant ces données, et calculer de façon systématique pour des valeurs croissantes de l'index

```
def dynMemoise(valeursFaciales, somme):
    f = [0] * (somme + 1)
    for s in range(1, somme + 1):
        f[s] = inf
        for x in valeursFaciales:
            if s >= x:
                f[s] = min(f[s], 1 + f[s - x])
    return f[somme]
```

Le calcul de f (48) va peut-être utiliser plusieurs fois la valeur de f (8), mais celle-ci n'aura cette fois été calculée qu'une seule fois,et aussitôt rangée en mémoire.
Cela ne fonctionne que parce que pour calculer f (48), par exemple, on a recours seulement aux valeurs de f (s) pour s $<$ 48.
Cette technique est habituellement appelée mémoïsation, fort laide tentative de traduction de l'anglais *memoization* mais qui est passée dans l'usage.
Cette fois l'appel *dynMemoise*(euros, 48) répond immédiatement 5 (l'algorithme glouton avait bien trouvé la réponse optimale) et *dynMemoise*(imperial, 48) renvoie 2, ce qui correspond à la solution optimale 48 = 24 + 24.
L'algorithme est de complexité tout à fait raisonnable : les deux boucles emboîtées correspondent à un temps d'exécution de l'ordre de somme * len(valeursFaciales).
En revanche, il a un coût en espace (ou en mémoire) : il faut réserver la place nécessaire pour ranger toutes les valeurs de f (n)

0.1.2 Reconstitution des détails de la réponse optimale

Si on remplace la dernière ligne de la fonction dynMemoise par return f, l'appel dynMemoise(imperial, 17) renvoie le tableau [0, 1, 2, 1, 2, 3, 1, 2, 3, 2, 3, 4, 1, 2, 3, 2, 3, 4].
Il y a une réponse optimale avec 4 pièces, comment la reconstituer ?

On a $f(17) = 4$, on cherche une pièce x telle que $f(17 - x) = 3$, on a le choix entre$x = 1, x = 3$ et $x = 12$. Choisissons $x = 12$.

On a $f(17) = 1 + f(5)$. On cherche maintenant une pièce x' telle que $f(5 - x') = 2$, on peut choisir $x' = 3$ ($x' = 1$ aurait également convenu). On a $f(17) = 1+f(5) = 1+1+f(2)$ et enfin $f(2) = 1+f(1)$. Une solution optimale est donc de payer $17 = 1 + 1 + 3 + 12$, avec 4 pièces.

On peut modifier la fonction précédente pour reconstituer une décomposition optimale : il suffit de détailler le calcul du minimum et en profiter pour mémoriser les valeurs notées x, x' etc. ci-dessus.

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

0.2 Qu'est ce que la programmation dynamique?

- On peut dire que la résolution algorithmique d'un problème relève de la programmation dynamique si
- le problème peut être résolu à partir de sous-problèmes similaires mais plus petits ; l'ensemble de ces sous-problèmes est discret, c'est-à-dire qu'on peut les indexer et ranger les résultats dans un tableau ;
- la solution optimale au problème posé s'obtient à partir des solutions optimales des sous-problèmes ;
- les sous-problèmes ne sont pas indépendants et un traitement récursif fait apparaître les mêmes sous-problèmes un grand nombre de fois

0.3 Le problème du sac à dos

0.3.1 Description du problème

On dispose de n objets de poids entiers strictement positifs $p_0, p_1, ..., p_{n-1}$ et auxquels on attache une valeur. On note $v_0, v_1, ...,_{n-1}$ les valeurs de ces objets,qui est également un entier strictement positif.

On dispose d'un sac qu'on ne doit pas surcharger : le poids total des objets qu'il contient ne doit pas dépasser un certain poids P .
On cherche à maximiser le total des valeurs des objets du sac.

Autrement dit, on cherche des nombres x_i valant 0 ou 1 tels que : $\sum_{i=0}^{n-1} x_i p_i =\leqslant P$ tout en maximisant $\sum_{i=0}^{n-1} x_i v_i$.

Par exemple, on peut considérer des objets de poids respectifs 1, 2, 5, 6, 7 et 9 et de valeurs 1, 6, 12, 18, 22, 28, et un sac dont le poids ne peut dépasser 11.
On peut choisir les objets de poids 1, 2 et 7, obtenant une valeur égale à 1 + 6 + 28 = 35. On peut aussi choisir les objets de poids 5 et 6, obtenant une valeur égale à 12 + 18 = 40.

Mais comment s'assurer qu'on ne peut pas mieux faire ?

Notons , pour $0 \leqslant i \leqslant n-1$ et $0 \leqslant p \leqslant P, V(i,p)$
la valeur maximale des objets qu'on peut ranger parmi ceux de numéros 0 à i sans dépasser le poids p.

Bien sûr, pour tout $i, V(i,0) = 0$ et $V(0,p) = \begin{cases} 0 & \text{si } p < p_0 \\ v_0 & \text{si } p \geqslant p_0 \end{cases}$

La clé de la résolution tient à l'équation récursive:
$V(i,p) = \begin{cases} V(i-1,p) & \text{si } p < p_i; \\ max(V(i-1,p), vi + V(i-1,p-pi)) & \text{si } p \geqslant pi \end{cases}$

On a distingué le cas où on renonce à utiliser l'objet numéro i : la valeur maximale est alors V (i, p). Si on utilise l'objet numéro i, on doit limiter à $p - p_i$ le poids consacré aux objets de numéros 0 à $i-1$ et ajouter sa valeur v_i à la valeur optimale du sous-problème

0.3.2 programmation

```
def sacADos1(poids, valeurs, poidsMaximum):
    assert(len(poids) == len(valeurs))
    n = len(poids)
    V = [[0]*(poidsMaximum + 1) for i in range(n)]
    for p in range(poids[0], poidsMaximum + 1):
        V[0][p] = valeurs[0]
    for i in range(1, n):
        for p in range(poids[i]):
            V[i][p] = V[i - 1][p]
        for p in range(poids[i], poidsMaximum + 1):
            V[i][p] = max(V[i - 1][p], valeurs[i] + V[i - 1][p - poids[i]])
    return V[n - 1][poidsMaximum]
```

L'imbrication des deux boucles permet d'assurer un temps d'exécution de l'ordre de $n \times P$. Le coût en mémoire est également de l'ordre de $n \times P$, puisqu'il faut réserver la place du tableau V. En fait, on pourrait consommer moins de mémoire. En effet on s'aperçoit que le calcul de la ligne V[i] du tableau utilise seulement les valeurs de la ligne précédent $V[i-1]$ et que la mise à jour de V[i][p] n'utilise que les valeurs de $V[i-1][k]$ pour $k \leqslant p$. Ainsi peut-on réécrire le programme de la façon suivante, en n'utilisant qu'une ligne de mémoire pour le tableau V.

```
def sacADos2(poids, valeurs, poidsMaximum):
    assert(len(poids) == len(valeurs))
    n = len(poids)
    L = [0] * poids[0] + [valeurs[0]] * (poidsMaximum + 1 - poids[0])
    for i in range(1, n):
        for p in range(poids[i], poidsMaximum + 1):
            L[p] = max(L[p], valeurs[i] + L[p - poids[i]])
    return L[poidsMaximum]
```

En ligne 4, on initialise la ligne correspondant à l'objet 0 : $V[0][p]$ est nul si $p < p_0$, et égal à v0 si $p \geqslant p_0$

0.4 Plus longue séquence commune

0.4.1 introduction

Le séquençage du génome a conduit à une collaboration fructueuse entre généticiens et informaticiens et au développement de nouveaux algorithmes.
En effet, on peut coder le génome avec les quatre lettres ACTG qui sont les initiales de quatre bases nucléiques : adénine, cytosine,thymine et guanine.

Du point de vue algorithmique, on considère donc des mots (très très longs) sur cet alphabet.
Un problème utile aux généticiens est celui de l'alignement de séquences, qui se décline en de nombreux sous-problèmes, dont plusieurs peuvent être abordés à l'aide de la programmation dynamique.

Nous nous intéressons ici à la recherche de la plus longue sous-chaîne commune. Étant donné deux textes x et y sur l'alphabet A, C, T, G, on cherche le texte z le plus longueur possible qui soit à la fois extrait de x et de y. Dire que z est extrait de x signifie que l'on peut obtenir z en effaçant des lettres de x. En particulier, les lettres qui figurent dans z figurent, dans le même ordre dans x.

Prenons l'exemple de x = AT AT ACAGGT CA et y = GACT ACACGACT , pour lesquels une plus longue sous-chaîne commune est z = AT ACACA.

```
A  T  A     T  A  C  A  G  G  T  C     A
      A     T  A  C  A              C     A
   G  A  C  T  A  C  A              C  G  A  C  T
```

Notons bien qu'il n'y a pas unicité ! ATAACAT est également une plus longue sous-chaîne commune, de même longueur 7,comme le schéma suivant le montre :

```
A     T  A  T  A  C     A  G  G  T  C  A
A     T  A     A  C     A           T
G  A  C  T  A  C  A  C  G  A  C     T
```

plscc(ATATACAGGTCA, GACTACACGACT) = ATAACAT et
λ(ATATACAGGTCA, GACTACACGACT) = 7.

0.4.2 Résolution du problème

Cherchons une solution récursive au problème.
Notons $X_m = \{x_1x_2...x_m\}$ une chaîne dans laquelle $x_1x_2...x_m$ représentent des caractères de l'alphabet $\{G, T, A, C\}$.
Notons $Y_n = \{y_1y_2...y_n\}$ une deuxième chaîne.
Nous cherchons une PLSC de X_n et Y_m.

Si $x_m = y_m$ alors il reste à trouver une PLSC de X_{m-1} et Y_{n-1} où X_{m-1} désigne la chaîne déduite de X_m en supprimant le dernier caractère de la chaîne.
De même pour Y_{n-1}.
La concaténation de $x_m = y_n$ à cette PLSC nous donne alors une PLSC de X_n et Y_m.

Si $x_m \neq y_m$, nous devons alors étudier et résoudre deux sous problèmes:
Trouver une PLSC de X_{m-1} et Y_n, et trouver une PLSC de X_n et Y_{n-1}.
La La plus grande des deux PLSC est une PLSC de X_n et Y_m.
Les cas étudiés précédemment couvrent toutes les possibilités,l' une , quelconque , des solutions optimales des sous problèmes apparaît obligatoirement dans une PLSC de X_n et Y_m.

Signalons les chevauchements possibles des sous-problèmes dans le problème de la PLSC.

Notons $l[i, j]$, la longueur d'une PLSC des séquences X_i et Y_j.Si $i = 0$ ou $j = 0$, l'une des séquences a une longueur nulle, et donc la PLSC est de longueur nulle.
La structure optimale du problème de la PLSC nous conduit aux relations:

$$l[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ l[i-1, j-1] + 1 & \text{si } i, j > 0 \text{ et } x_i = y_j \\ max(l[i, j-1], l[i-1, j]) & \text{si } i, j > 0 \text{ et } x_i \neq y_j \end{cases}$$

0.4.3 programmation

Calcul d'une plus grande séquence commune

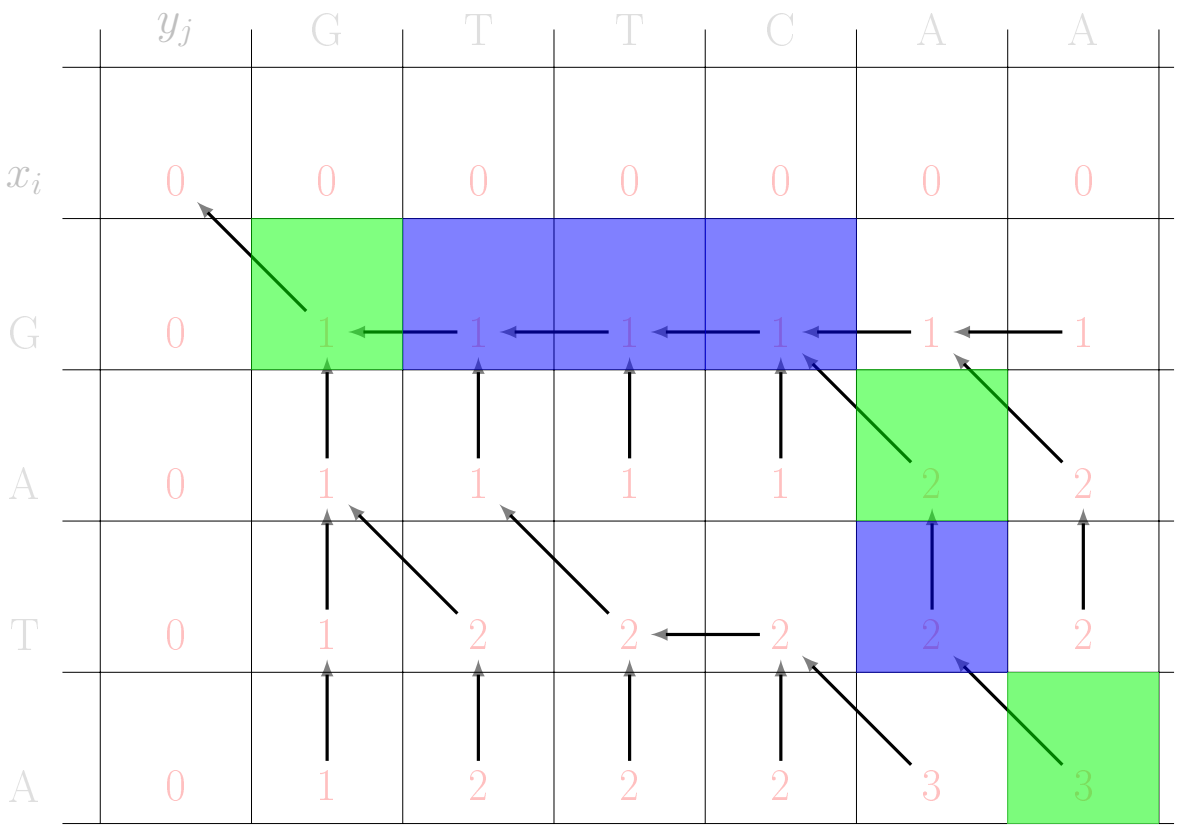
La fonction LONGUEUR-PLSC prend pour arguments deux séquences $X_m = \{x_1x_2...x_m\}$ et $Y_n = \{y_1y_2...y_n\}$.
Elle stocke les valeurs l[i,j] dans un tableau SOL[0..m,0..n] dont les éléments sont calculés dans l'ordre des lignes (de gauche à droite).
Pour obtenir la plus longue séquence commune , on utilise un tableau TAB[1..m,1..n] :

Chaque valeur TAB[i,j] "pointe" vers l'élément de tableau qui correspond à la solution optimale du sous-problème choisie lors du calcul de l[i,j]..

Algorithme 1 : LONGUEUR-PLSC(X,Y)

```
fonction PLSC(X,Y)
m=X.longueur
n=Y.Longueur
TAB=[1..m,1..n]
SOL=[0,..m,0..n]
pour i allant de 1 à m faire
  | SOL[i,0]=0
fin
pour j allant de 0 à n faire
  | SOL[0,j]=0
fin
pour i allant de 1 à m faire
  | pour j allant de 1 à n faire
  |   | si xi == yj alors
  |   |   | SOL[i,j]= SOL[i-1,j-1] + 1 ;
  |   |   | TAB[i,j]="Diag-gauche"
  |   | sinon si SOL[i-1,j] >= SOL[i,j-1] alors
  |   |   | SOL[i,j] = SOL[i-1,j] ;
  |   |   | TAB[i,j]="haut"
  |   | sinon
  |   |   | SOL[i,j] = SOL[i,j-1] ;
  |   |   | TAB[i,j]="gauche"
  | fin
fin
retourner SOL , TAB
```

la fonction retourne les tableaux SOL et TAB


y_j
G
T
T
C
A
A
x_i
0 0 0 0 0 0 0
G 0 1 1 1 1 1 1
A 0 1 1 1 1 2 2
T 0 1 2 2 2 2 2
A 0 1 2 2 2 3 3