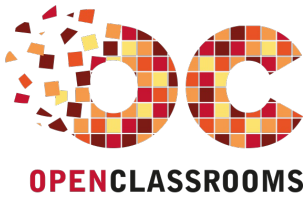
Introduction à la programmation dynamique

Par yoch



www.openclassrooms.com

Dernière mise à jour le 1/07/2012

Sommaire

Introduction à la programmation dynamique

Par  yoch

Mise à jour : 01/07/2012
Difficulté : Intermédiaire

La « **programmation dynamique** » est un paradigme de programmation, c'est-à-dire une façon particulière d'appréhender un problème algorithmique donné.

C'est une méthode utile pour obtenir une solution exacte à un problème algorithmique, là où une solution « classique » se trouve être trop complexe, c'est-à-dire trop peu efficace. On parle alors d'**optimisation combinatoire**.

L'efficacité de cette méthode repose sur le principe d'optimalité énoncé par le mathématicien Richard Bellman : *« toute politique optimale est composée de sous-politiques optimales »*.

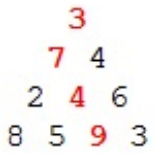
Cette méthode paraît souvent inaccessible au débutant (d'ailleurs, si vous n'avez rien compris à ce qui précède, ne craignez rien : c'est parfaitement normal), aussi je vous propose de la découvrir avec moi sur quelques problèmes concrets.

Sommaire du tutoriel :

Un exemple simple

Pour commencer, nous allons prendre un problème tiré d'ici :

Voici une pyramide de nombres. En partant du sommet, et en se dirigeant vers le bas à chaque étape, on doit réussir à maximiser le total des nombres traversés. Sur l'image d'exemple, ce maximum est 23 (le chemin est indiqué en rouge).



Quelle est la meilleure méthode pour trouver un tel chemin ?

Si l'on teste tous les chemins possibles, ça devient impraticable dès que la pyramide est un peu grande. En effet, pour une pyramide de hauteur N, il y a 2^{N-1} chemins possibles. Nous allons donc très vite nous retrouver bloqués...

Mais heureusement, il existe une manière de procéder infiniment meilleure (et très intuitive, dans notre cas) :

Chaque case de notre pyramide (sauf le sommet) possède un ou deux parents (parent = case du dessus permettant d'y parvenir). Les cases situées sur les arêtes (= cotés) n'ont qu'un parent, et les autres en ont deux, l'un à gauche et l'autre droite. A partir de là, et sachant que l'on parcourt la pyramide de haut en bas, il suffit de déterminer pour **chaque case de la pyramide** quel est le maximum possible, et nous trouvons alors facilement la réponse.

Analysons ensemble le tableau :

Code : Autre

```
LIGNE 1 (sommet) :
  - CASE 1 (3) : max = 3 (seule valeur possible)

LIGNE 2
  - CASE 1 (7) : max = 7 + 3 = 10
  - CASE 2 (4) : max = 4 + 3 = 7

LIGNE 3
  - CASE 1 (2) : max = 10 + 2 = 12
  - CASE 2 (4) : max = MAX(10+4, 7+4) = 10 + 4 = 14
  - CASE 3 (6) : max = 6 + 7 = 13

LIGNE 4 :
  - CASE 1 (8) : max = 12 + 8 = 20
  - CASE 2 (5) : max = MAX(12+5, 14+5) = 14 + 5 = 19
  - CASE 3 (9) : max = MAX(14+9, 13+9) = 14 + 9 = 23
  - CASE 4 (3) : max = 13 + 3 = 16
```

Il suffit maintenant de chercher le maximum pour la ligne 4, et le tour est joué ! 

Cette approche en informatique est appelée « programmation dynamique ». Elle se révèle extrêmement efficace pour résoudre certains problèmes courants (souvent des problèmes de combinatoire). Nous allons maintenant voir comment résoudre certains problèmes intéressants grâce à la programmation dynamique.

Problème de partition de nombres

Problème

Le problème est le suivant : on dispose d'un grand sac de monnaie (de billets, ou de ce que vous voulez). L'objectif est de diviser le sac en deux sacs plus petits contenant les sommes les plus rapprochées possibles.

L'approche la plus « simple » serait d'essayer toutes les combinaisons et d'en retenir la meilleure. Néanmoins, après un petit calcul, on s'aperçoit que cela n'est pas très réaliste. En effet, supposons que nous avons 100 pièces, il existe alors 2^{100} combinaisons possibles. Il faut donc obligatoirement trouver une meilleure solution.

Méthode

C'est là qu'intervient le concept de programmation dynamique. Pour connaître toutes les combinaisons possibles, il existe un autre moyen : traiter le problème « à l'envers ».

Prenons un cas concret.
Dans un sac de pièces, nous avons :

Code : Autre

```
PIÈCES : 5, 9, 3, 8, 2, 5
```

Nous allons rechercher une sous-liste de pièces optimale dont la somme se rapprochera le plus possible de la moitié de la somme de toutes les pièces. Raisonnons :

1. Avec la première pièce, quel est l'assemblage possible ?
2. Et avec les deux premières pièces, quels sont les assemblages possibles ? Quelle est la meilleure solution ?
3. Et ainsi de suite jusqu'à *n* pièces...

Code : Autre

```
ÉTAPE 1. avec 5 :
 - COMBINAISONS CONNUES : néant.
 - PIÈCE À ÉVALUER : 5.
 - NOUVELLES COMBINAISONS : 5.

ÉTAPE 2. avec 5 et 9 :
 - COMBINAISONS CONNUES : 5.
 - PIÈCE À ÉVALUER : 9.
 - NOUVELLES COMBINAISONS : 9,  14 (9+5).

ÉTAPE 3. avec 5, 9 et 3 :
 - COMBINAISONS CONNUES : 5, 9, 14.
 - PIÈCE À ÉVALUER : 3.
 - NOUVELLES COMBINAISONS : 3,  8 (3+5), 12 (3+9)...

ÉTAPE 4. avec 5, 9, 3 et 8 :
 - COMBINAISONS CONNUES : 3, 5, 8, 9, 12, 14.
 - PIÈCE À ÉVALUER : 8.
 - NOUVELLES COMBINAISONS : 11 (3+8),  13 (5+8), 16 (8+8)...

On a trouvé 16, alors on s'arrête !
```

En pratique, on peut connaître les assemblages possibles avec deux pièces à partir des assemblages possibles avec une pièce, et ainsi de suite. C'est ce que l'on appelle une relation de récurrence : la solution du problème sur *n* instances est récurrente à la solution du problème sur *n-1* instances.

On constate que, pour travailler, le programme a besoin de connaître les données déjà trouvées. C'est le fondement même de la programmation dynamique : on élimine les recalculs en mémorisant ce qui a déjà été calculé.

C'est ce qui fait l'efficacité de cette méthode, mais c'est aussi son point faible : **la programmation dynamique requiert de la mémoire et ne sera pas toujours envisageable à cause de cela.**

Et concrètement, on fait comment ?

Mise en oeuvre

Passons maintenant à l'implémentation de tout cela.

Nous allons devoir créer une matrice booléenne M de dimensions nombrePieces * (miSommeGlobale+1), que nous appellerons M[i][j].

Dans le cas où sommeGlobale est paire, miSommeGlobale vaut sommeGlobale/2. Dans le cas inverse, on prendra (sommeGlobale-1)/2.

Code : Autre

```
DEFINIT sommePieces, miSommeGlobale
sommePieces = somme(listePieces)

SI sommePieces EST PAIRE
   miSommeGlobale = sommePieces/2
SINON
   miSommeGlobale = (sommePieces-1)/2

DEFINIT M[nombrePieces][miSommeGlobale+1]
```

Puis on remplit la première ligne comme ceci :

Code : Autre

```
M[0][0] = VRAI
POUR j=1 TANT QUE j < miSommeGlobale
   SI piece[0] EGALE j
       M[0][j] = VRAI
   SINON
       M[0][j] = FAUX
   incrémente j
```

Ce qui nous donne :

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
1re pièce	V	F	F	F	F	V	F	F	F	F	F	F	F	F	F	F	F

Cette ligne représente les combinaisons possibles avec la première pièce.
On peut avoir soit 0 avec zéro pièces, soit 5 avec une pièce.

Ensuite, nous allons remplir les autres lignes du tableau (ligne par ligne) en suivant la condition suivante :

Code : Autre

```
M[i][j] EST VRAI SI
   - M[i-1][j] EST VRAI
   OU
   - piece[i] <= j ET M[i-1][j-piece[i]] EST VRAI
```

Ce qui nous donne bien :

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
1re pièce	V	F	F	F	F	V	F	F	F	F	F	F	F	F	F	F	F
2 premières pièces	V	F	F	F	F	V	F	F	F	V	F	F	F	F	V	F	F
3 premières pièces	V	F	F	V	F	V	F	F	V	V	F	F	V	F	V	F	F
4 premières pièces	V	F	F	V	F	V	F	F	V	V	F	V	V	V	V	F	V

Chaque ligne représente les combinaisons possibles avec les i premières pièces.

Dans ce cas précis, on s'arrête à la ligne 4 car la solution idéale a été trouvée.

Sinon, on continuera à remplir le tableau jusqu'à la dernière ligne, et nous connaîtrons de ce fait la solution optimale : la valeur maximale de la dernière ligne du tableau représente l'une des deux sous-sommes optimales (la plus petite), l'autre sous-somme optimale étant alors la somme globale moins cette valeur.

Code : Autre

```
DEFINIT sousListe1, sousListe2, ecart
i = nombrePieces, j = miSommeGlobale + 1

TANT QUE M[i][j] EST FAUX
```

```
   décrémente j

sousListe1 = j
sousListe2 = sommePieces - j
ecart = sommePieces-(j*2)
```

Et si je veux récupérer la sous-liste en question, y a-t-il un moyen ?

Eh bien en fait, si tu as gardé le tableau ci-dessus, je pense qu'il doit y avoir un moyen. 

Récupérer la sous-liste

Nous allons récupérer la sous-liste obtenue en suivant le procédé à l'envers.

Analysons le tableau :

- En M[3][16], nous avons le résultat optimal. Or le quatrième nombre (piece[3]) est 8.
- Passons alors en M[2][16-8] ; à nouveau, le résultat est optimal (la case au-dessus est *false*). De plus, nous savons que le troisième nombre (piece[2]) est 3.
- On passe à M[1][8-3] ; le résultat n'est pas optimal.
- Puis on passe à M[0][5] ; le résultat est optimal, donc on ajoute le premier nombre (piece[0]) à notre sous-liste.
- On retrouve bien notre sous-liste optimale : 8, 3 et 5.

Algorithme

Une fois que l'on a trouvé la sous-liste optimale, on fait :

Code : Autre

```
TANT QUE j > 0
   TANT QUE i > 0 ET M[i-1][j] EST VRAI
      décrémente i
   j = j - piece[i]
   SI j > 0
      Ajoute-a-petite-sous-liste ( piece[i] )
   décrémente i
```

Problème du sac à dos

Problème

J'ai un sac à dos de capacité maximale k (poids). J'ai *n* objets de poids et de valeurs diverses. Je souhaite remplir ledit sac de façon à cumuler un maximum de valeurs. Quel est le bénéfice maximum que je peux faire ?

Ce problème est un grand classique en algorithmique et il existe diverses méthodes pour le résoudre.
Je vais vous présenter ici la méthode dite par programmation dynamique.

Mais auparavant, prenons un cas concret :

Code : Autre

```
MAX CONTENANCE : 12

- OBJET : A| B|  C| D|  E|  F|  G| H
- POIDS : 2| 3|  5| 2|  4|  6|  3| 1
- VALUE : 5| 8| 14| 6| 13| 17| 10| 4
```

Analyse

C'est là que la fameuse règle « toute politique optimale est composée de sous-politiques optimales » prend tout son sens. Nous allons l'appliquer à nouveau et traiter le problème depuis le bas.

On a :

Code : Autre

```
ÉTAPE 1 : pour l'objet A
 - COMBINAISONS OPTIMALES CONNUES :
 - OBJET À ÉVALUER : A
 - NOUVELLES COMBINAISONS OPTIMALES :
    - je peux avoir un bénéfice de 5 pour un poids de 2 (A).

ÉTAPE 2 : pour les objets A et B
 - COMBINAISONS OPTIMALES CONNUES :
    - je peux avoir un bénéfice de 5 pour un poids de 2 (A).
 - OBJET À ÉVALUER : B
 - NOUVELLES COMBINAISONS OPTIMALES :
    - je peux avoir un bénéfice de 8 pour un poids de 3 (B).
    - je peux avoir un bénéfice de 13 pour un poids de 5 (A+B).

ÉTAPE 3 : pour les objets A, B et C
 - COMBINAISONS OPTIMALES CONNUES :
    - je peux avoir un bénéfice de 5 pour un poids de 2 (A).
    - je peux avoir un bénéfice de 8 pour un poids de 3 (B).
    - je peux avoir un bénéfice de 13 pour un poids de 5 (A+B).
 - OBJET À ÉVALUER : C
 - NOUVELLES COMBINAISONS OPTIMALES :
    - je peux avoir un bénéfice de 14 pour un poids de 5 (C).
    - je peux avoir un bénéfice de 19 pour un poids de 7 (A+C).
    - je peux avoir un bénéfice de 22 pour un poids de 8 (B+C).
    - je peux avoir un bénéfice de 27 pour un poids de 10 (A+B+C).

ÉTAPE 4 : pour les objets A, B ,C et D
 - COMBINAISONS OPTIMALES CONNUES :
    - je peux avoir un bénéfice de 5 pour un poids de 2 (A).
    - je peux avoir un bénéfice de 8 pour un poids de 3 (B).
    - je peux avoir un bénéfice de 14 pour un poids de 5 (C).
    - je peux avoir un bénéfice de 19 pour un poids de 7 (A+C).
    - je peux avoir un bénéfice de 22 pour un poids de 8 (B+C).
    - je peux avoir un bénéfice de 27 pour un poids de 10 (A+B+C).
 - OBJET À ÉVALUER : D
 - NOUVELLES COMBINAISONS OPTIMALES :
    - je peux avoir un bénéfice de 6 pour un poids de 2 (D).
    - je peux avoir un bénéfice de 11 pour un poids de 4 (A+D).
    - je peux avoir un bénéfice de 20 pour un poids de 7 (C+D).
    - je peux avoir un bénéfice de 25 pour un poids de 9 (A+C+D).
    - je peux avoir un bénéfice de 28 pour un poids de 10 (B+C+D).
    - je peux avoir un bénéfice de 33 pour un poids de 12 (A+B+C+D).


Et ainsi de suite...
```

Et si tu nous disais comment tu expliques ça au programme ?

C'est vrai que la récurrence est ici moins évidente à trouver.  Voici comment nous allons procéder.

Mise en oeuvre

Nous allons créer une matrice d'entiers M de dimensions nombreObjets * (maxContenance+1), que nous appellerons M[i][j].

Puis on remplit la première ligne comme ceci :

Code : Autre

```
POUR j=0 TANT QUE j < maxContenance
   SI poidsObjet[0] > j
       M[0][j] = 0
   SINON
       M[0][j] = valeurObjet[0]
   incrémente j
```

Ce qui nous donne :

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5

Chaque case du tableau représente le bénéfice maximum possible pour les i premiers objets avec un poids j.

Ensuite, nous allons remplir les autres lignes du tableau (ligne par ligne) ainsi :

Code : Autre

```
SI poidsObjet[i] > j
    M[i][j] = M[i-1][j]
SINON
    M[i][j] = maximum ( M[i-1][j], M[i-1][j-
poidsObjet[i]] + valeurObjet[i] )
```

Ce qui nous donne au final :

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0	0	5	8	8	13	13	13	13	13	13	13	13
3 premiers objets	0	0	5	8	8	14	14	19	22	22	27	27	27
4 premiers objets	0	0	6	8	11	14	14	20	22	25	28	28	33
5 premiers objets	0	0	6	8	13	14	19	21	24	27	28	33	35
6 premiers objets	0	0	6	8	13	14	19	21	24	27	30	33	36
7 premiers objets	0	0	6	10	13	16	19	23	24	29	31	34	37
8 premiers objets	0	4	6	10	14	17	20	23	27	29	33	35	38

Le plus grand bénéfice possible se trouve dans la dernière case du tableau (en bas à droite). Celui-ci est donc de 38.

Le poids nécessaire pour faire un tel bénéfice se retrouve en reculant horizontalement depuis la dernière case du tableau (en bas à droite) tant que le bénéfice reste le même. Donc ici le poids nécessaire est 12 (car on ne peut pas reculer).

Récupérer la liste d'objets

Le principe est le même que plus haut. Tout d'abord, on récupère dans la dernière ligne le poids minimal nécessaire pour faire le bénéfice optimal :

Code : Autre

```
TANT QUE M[i][j] EGALE M[i][j-1]
   décrémente j
```

Et puis, de là, on récupère les objets :

Code : Autre

```
TANT QUE j > 0
   TANT QUE i > 0 ET M[i][j] EGALE M[i-1][j]
      décrémente i
   j = j - PoidsObjet[i]
   SI j > 0
      Ajoute-objet ( Objet[i] )
   décrémente i
```

Voilà, ce (court) tuto touche à sa fin. J'espère que vous avez bien compris et qu'il vous aura donné matière à réflexion.

Pour résumer :

Citation : candide

1°) La programmation dynamique est simplement le fait de construire un tableau dont chaque ligne se déduira d'une ou de plusieurs lignes précédentes.
2°) La programmation dynamique a ses limites (la taille des instances du problème).

Cette approche permettra (sous certaines conditions) de résoudre nombre de problèmes « difficiles ».
Exemples de problèmes que l'on peut résoudre par programmation dynamique (en anglais) :
http://people.csail.mit.edu/bdean/6.046/dp/

Remerciements à candide pour sa relecture et ses commentaires pertinents.

Ce tutoriel a été corrigé par les zCorrecteurs.