Programmation dynamique

Lundi 11 Avril 2016

Michael FRANÇOIS

francois@esiea.fr

Programmation dynamique

• Le concept de la programmation dynamique a été introduit au début des années 1950 par le mathématicien américain **Richard Ernest Bellman** (1920 -- 1984).

• À l'époque le terme « programmation » était vu plus comme *planification* ou même *ordonnancement*.

• La programmation dynamique s'applique généralement aux problèmes d'optimisation. Pour ce genre de problèmes, il peut y avoir de nombreuses solutions possibles.

• La programmation dynamique, comme la méthode diviser-pour-régner, résout des problèmes en combinant des solutions de sous-problèmes. Cependant :

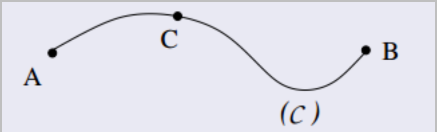
- un algorithme de type diviser-pour-régner partitionne le problème en sous-problèmes **indépendants**, qui sont résolus de manière récursive et dont les solutions sont combinées pour résoudre le problème initial.
- La programmation dynamique s'applique même lorsque les sous-problèmes se recoupent, c'est-à-dire lorsque des sous-problèmes ont des sous-problèmes en commun.

Remarque : lorsque les sous-problèmes se recoupent, un algorithme de type diviser-pour-régner fait plus de travail que nécessaire, car il doit résoudre plusieurs fois des sous-problèmes communs.

La programmation dynamique s'appuie sur un principe simple, appelé :

Principe d'optimalité de Bellman

Un chemin optimal est formé de sous-chemins optimaux : Si $(\mathcal{C})$ est un chemin optimal allant de A à B et si C appartient à $(\mathcal{C})$, alors les sous-chemins de $(\mathcal{C})$ allant de A à C et de C à B sont également optimaux.



Remarque : autrement dit, on peut déduire une ou la solution optimale d'un problème en combinant des solutions optimales d'une série de sous-problèmes. Les solutions des problèmes sont calculées de manière ascendante, c'est-à-dire qu'on débute par les solutions des sous-problèmes les plus petits pour ensuite déduire progressivement les solutions de l'ensemble.

Quelques problèmes classiques

Triangle de Pascal

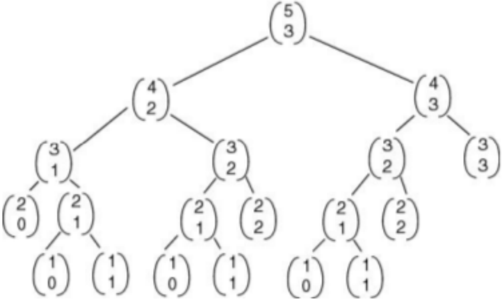
En mathématiques, le triangle de Pascal est une présentation des coefficients binomiaux dans un triangle. La construction du triangle est liée aux coefficients binomiaux selon la règle de Pascal qui s'énonce ainsi :
$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$ (formule du binôme)

- On veut calculer $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ (TRÈS CHER)
- Proposition :
  - $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ pour $0 < k < n$,
  - 1 sinon.
- Approche **Diviser-pour-régner** (récurrence intelligente) :

Fonction $C(n,k)$

- Si $k = 0$ ou $k = n$, alors retourner 1.
- Retourner $C(n-1,k-1) + C(n-1,k)$.

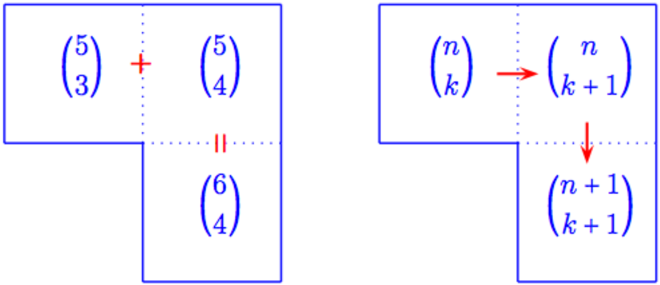
Arbre de calcul pour $C(5,3)$



NB : il y a 18 appels récursifs, et notamment des traitements similaires comme par exemple pour $\binom{3}{2}$, ou $\binom{1}{1}$, etc.

Meilleure idée $\Rightarrow$ triangle de Pascal

Schéma de calcul pour $\binom{6}{4}$



n \ k	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

NB : ici on évite de calculer plusieurs fois la même chose.

Remarque : en général, pour la construction de la solution globale du problème, les résultats partiels doivent être stockés au fur et à mesure. Cependant, l'exemple du triangle de Pascal montre qu'il n'est pas toujours nécessaire de les stocker tous.

Découpe de barres

• Une entreprise $\mathbb{E}$ achète des barres d'acier et les coupe en plus petits morceaux pour faire des barres plus courtes. Les morceaux sont ensuite vendues sur le marché.

• Voilà les tarifs que fait l'entreprise $\mathbb{E}$:

Longueur i (cm)	1	2	3	4	5	6	7	8	9	10
Prix P_i (€)	1	5	8	9	10	17	17	20	24	30

Chaque barre de longueur i cm rapporte à l'entreprise $\mathbb{E}$ P_i euros.

• Objectif de $\mathbb{E}$: connaître la meilleure façon de couper les barres, afin de maximiser le profit en vendant les morceaux.

Formulation du problème :

Soit une barre de longueur n cm et un tableau de référence de prix P_i pour $i = 1, \ldots, n$.
Quel est le revenu maximal R_n, lorsque l'on coupe la barre et on vend les morceaux correspondants ?

Si une solution optimale coupe la barre en k morceaux, pour $1 \leq k \leq n$, alors une découpe optimale de la barre en morceaux de longueur $i_1, \ldots, \imath_k$ est :

$$n = i_1 + \cdots + i_k$$

et le revenu maximal correspondant est de :

$$R_n = P_{i_1} + \cdots + P_{i_k}$$

Remarque : dans le cas où le prix P_n d'une barre de longueur n est assez élevé, une solution optimale consistera peut être à vendre la barre entière sans la couper.

Exemple : (barre de longueur 4 cm)

- Aucune découpe $\Longrightarrow R_4 = 9$ €
- Découpe de 1+3 $\Longrightarrow R_1 + R_3 = 9$ €
- Découpe de 2+2 $\Longrightarrow R_2 + R_2 = 10$ €
- Découpe de 3+1 $\Longrightarrow R_3 + R_1 = 9$ €
- Découpe de 1+1+2 $\Longrightarrow R_1 + R_1 + R_2 = 7$ €
- Découpe de 1+2+1 $\Longrightarrow R_1 + R_2 + R_1 = 7$ €
- Découpe de 2+1+1 $\Longrightarrow R_2 + R_1 + R_1 = 7$ €
- Découpe de 1+1+1+1 $\Longrightarrow R_1 + R_1 + R_1 + R_1 = 4$ €

Découpe optimale : deux morceaux de longueur 2 cm chacun.

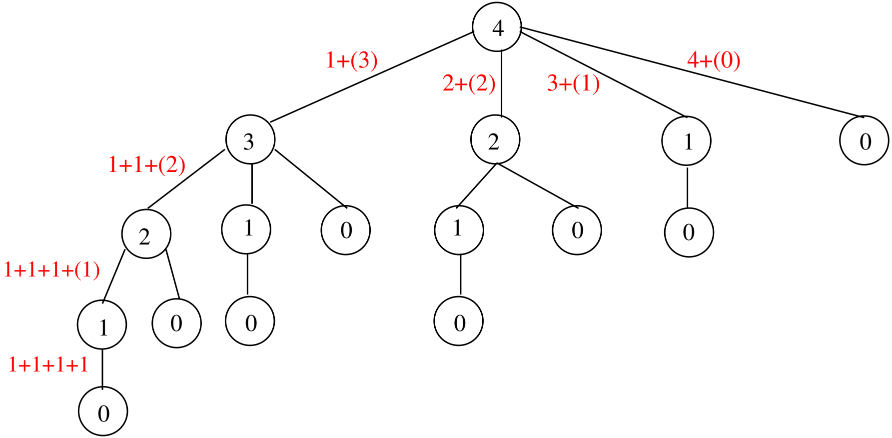
Remarque : il y a 2^{n-1} (ici 8) façons de découper la barre initiale.

- Plus généralement, le revenu maximal est donné par la relation :
  - $R_n = \max(P_n, R_1 + R_{n-1}, R_2 + R_{n-2}, \ldots, R_{n-1} + R_1)$
    où P_n correspond à la vente de la barre entière et les autres $n-1$ arguments correspondent au revenu maximal obtenu en effectuant un découpage en morceaux de longueur i et $n-i$.

- Version récursive (plus simple) :
  - $R_n = \max_{1 \leq i \leq n}(P_i + R_{n-i})$
    ici une découpe se compose d'un premier morceau de longueur i coupé à partir de la gauche et d'un reste de longueur $n-i$. On ne peut couper ensuite que le reste et pas le premier morceau.

Implémentation descendante récursive naïve

```
--------------------------------------------------------------------
fonction COUPER_BARRE(P, n)
/*P[1..n] contient les tarifs et n la longueur initiale de la barre*/
DEBUT
   Si n=0 alors Retourner 0 /*pas de revenu possible*/
   R <-- -1 /*Initialisation du revenu maximal R à -1*/
   Pour i=1 à n faire
      R <-- max(R, P[i] + COUPER_BARRE(P, n-i))
   fin pour
   Retourner R
FIN
--------------------------------------------------------------------
```



L'arbre récursif montrant les appels récursifs induits par un appel à `COUPER_BARRE(P, n)` (ici $n = 4$).

L'ordre des appels récursifs est le suivant : 3 2 1 0 0 1 0 0 2 1 0 0 1 0 0

Complexité : $O(2^n)$

Exemple de code C :

```c
#include <stdio.h>
#include <stdlib.h>


int COUPER_BARRE(int P[11], int n)
{
  printf("%d\n", n);
  int i,R,x;
  if (n==0) {return 0;}
  R=-1;
  for (i=1; i<=n; i++)
  {
    x = P[i] + COUPER_BARRE(P, n-i);
    if (x>R) {R=x;}
  }
  return R;
}

int main(int argc, char ** argv)
{
  int P[11] = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30};
  printf("R = %d\n", COUPER_BARRE(P, atoi(argv[1])));

  return 0;
}
```

Exécution pour n=4 :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 4
4
3
2
1
0
0
1
0
0
2
1
0
0
1
0
0
R = 10
```

On voit bien les 15 appels récursifs. Le premier correspond au premier appel de la fonction `COUPER_BARRE` dans le `main`.

Exécutions pour n=2 et n=0 :

```
$ ./EXEC 2
2
1
0
0
R = 5
$ ./EXEC 0
0
R = 0
```

Optimisation du découpage des barres en utilisant cette fois-ci la programmation dynamique

- On a pu observer l'inefficacité d'une solution récursive naïve, à cause de la répétition de la résolution des mêmes sous-problèmes.
- Cette fois-ci, chaque sous-problème ne sera résolu qu'une seule fois et en sauvegardant sa solution, qui pourra ainsi être réutilisée plus tard.
- Programmation dynamique : utilisation d'avantage de mémoire, mais moins de temps de calcul :
  - une solution en $O(2^n)$ peut se transformer en une solution en $O(n^2)$
- Il existe généralement deux façons équivalentes d'utiliser la programmation dynamique :
  1. Approche **descendante avec mémoïsation**
  2. Approche **ascendante**

1. Approche descendante avec mémoïsation :

• **[WIKI]** En informatique, la **mémoïsation** est une technique d'optimisation de code consistant à réduire le temps d'exécution d'une fonction en mémorisant ses résultats d'une fois sur l'autre.

• Ici donc, la procédure est écrite de manière récursive comme la précédente mais en sauvegardant au fur et à mesure les résultats de chaque sous-problème dans un tableau.

• On dit que la procédure récursive a été mémoïsée ; elle "se souvient" des résultats qu'elle avait précédemment calculés.

Procédure descendante COUPER_BARRE avec ajout de la mémoïsation :

```
---------------------------------------------------------------------
fonction COUPER_BARRE_MEMO(P, n)
DEBUT
   Soit r[0...n] un nouveau tableau contenant les revenus
   Pour i=0 à n faire
      r[i] <-- -1 /*Initialisation de tous les revenus à -1*/
   fin pour
   Retourner COUPER_BARRE_MEMO_REC(P, n, r)
FIN

fonction COUPER_BARRE_MEMO_REC(P, n, r)
DEBUT
   Si r[n]>=0 Retourner r[n] /*si la valeur est déjà connue on la renvoie*/
   Si n=0 alors R <-- 0
   Sinon R <-- -1
   Pour i=1 à n faire
       R <-- max(R, P[i] + COUPER_BARRE_MEMO_REC(P, n-i, r))
   fin Pour
   r[n] <-- R /*Sauvegarde du résultat correspondant au sous-problème*/
   Retourner R
FIN
---------------------------------------------------------------------
```

Complexité en $O(n^2)$

Code C :

```c
#include <stdio.h>
#include <stdlib.h>

int COUPER_BARRE_MEMO_REC(int P[], int n, int r[n+1])
{
  int i, R, x;

  if (r[n] >= 0) {return r[n];}
  else if (n==0) {R=0;}
  else
  {
     printf("%d \n", n); R=-1;
     for (i=1; i<=n; i++)
     {
       x = P[i] + COUPER_BARRE_MEMO_REC(P, n-i, r);
       if (x>R) {R=x;}
     }
  }
  r[n] = R; return R;
}

int COUPER_BARRE_MEMO(int P[], int n)
{
  int i, r[n+1];
  for (i=0; i<=n; i++) {r[i] = -1;}
  return COUPER_BARRE_MEMO_REC(P, n, r);
}

int main(int argc, char ** argv)
{
  int P[11] = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30};
  printf("R = %d\n", COUPER_BARRE_MEMO(P, atoi(argv[1])));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 4
4
3
2
1
R = 10
$ ./EXEC 10
10
9
8
7
6
5
4
3
2
1
R = 30
$ ./EXEC 1
1
R = 1
$ _
```

Remarque : on voit bien ici, que le calcul est effectué qu'une seule fois pour chaque valeur de *n*, d'où une grande économie de temps pour cette version d'algorithme.

2. Approche ascendante :

• Dans cette approche, on résout les sous-problèmes par ordre de taille, en commençant par le plus petit.

• Chaque sous-problème est résolu qu'une seule fois, et quand il est rencontré pour la première fois, ses sous-problèmes correspondants ont déjà été tous résolus.

• Cette approche est souvent légèrement meilleure, car les appels de procédure génèrent moins de traitements.

La version ascendante est plus simple :

```
-------------------------------------------------------------------
fonction COUPER_BARRE_ASCENDANTE(P, n)
DEBUT
   Soit r[0...n] un nouveau tableau contenant les revenus
   r[0] <-- 0 /*Une barre de longueur 0 ne rapporte rien*/
   Pour j=1 à n faire
      R <-- -1
      Pour k=1 à j faire
         R <-- max(R, P[k] + r[j-k])
      fin pour
      r[j] <-- R /*Mémorisation de R pour le sous-problème de long. j*/
   fin Pour
   Retourner r[n] /*Valeur optimale pour une barre de longueur n*/
FIN
-------------------------------------------------------------------
```

Complexité en $O(n^2)$

Code C :

```c
#include <stdio.h>
#include <stdlib.h>

int COUPER_BARRE_ASCENDANTE(int P[], int n)
{
  int j, k, R, x, r[n+1];
  r[0]=0;
  for (j=1; j<=n; j++)
  {
    R=-1;
    for (k=1; k<=j; k++)
    {
      x = P[k] + r[j-k];
      if (x>R) {R=x;}
    }
    r[j]=R;
  }
  return r[n];
}

int main(int argc, char ** argv)
{
  int P[11] = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30};
  printf("R = %d\n", COUPER_BARRE_ASCENDANTE(P, atoi(argv[1])));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 4
R = 10
$ ./EXEC 5
R = 13
$ ./EXEC 8
R = 22
$ ./EXEC 10
R = 30
$ ./EXEC
Segmentation fault (core dumped)
$
```

Reconstruction d'une solution :

• Jusqu'à là, la solution optimale retournée (*i.e.* R) correspondait au profit maximal réalisable pour la découpe d'une barre de longueur n.

• Maintenant on veut obtenir également la liste des tailles des morceaux, afin de procéder aux découpes.

• Pour cela, il suffit d'enregistrer le choix ayant conduit à la solution optimale pour chaque sous-problème.

```
--------------------------------------------------------------------------------
fonction COUPER_BARRE_ASCENDANTE_2(P, n)
DEBUT
   Soit r[0...n] et m[0...n] de nouveaux tableaux
   r[0] <-- 0 /*Une barre de longueur 0 ne rapporte rien*/
   Pour j=1 à n faire
      R <-- -1
      Pour k=1 à j faire
         x <-- P[k] + r[j-k]
         Si R < x alors
            R <-- x
            m[j] <-- k /*stockage de la taille optimale k du 1er morceau
                        à couper pour un sous-problème de taille j*/
         fin Si
      fin pour
      r[j] <-- R /*Mémorisation de R pour le sous-problème de long. j*/
   fin Pour
   Retourner (r,m)
FIN

fonction AFFICHAGE_SOLUTION(P, n)
DEBUT
   (r,m) <-- COUPER_BARRE_ASCENDANTE_2(P, n)
   Tant que n > 0 faire
       afficher m[n]
       n <-- n - m[n]
   fin Tant que
FIN
--------------------------------------------------------------------------------
```

Code C :

```c
#include <stdio.h>
#include <stdlib.h>

void COUPER_BARRE_ASCENDANTE_2(int P[], int n, int r[n+1], int m[n+1])
{
  int j, k, R, x;
  for (j=1; j<=n; j++)
  {
    R=-1;
    for (k=1; k<=j; k++)
    {
      x = P[k] + r[j-k];
      if (x>R) {R=x; m[j]=k;}
    }
    r[j]=R;
  }
}

void AFFICHAGE_SOLUTION(int P[], int n)
{
  int i, m[n+1], r[n+1];
  r[0]=0;
  for (i=0; i<=n; i++) {m[i] = 0;}
  COUPER_BARRE_ASCENDANTE_2(P, n, r, m);
  i=n;
  printf("Pour une barre de longueur %d, voilà le découpage optimal :\n", n);
  while (i>0)
  {
    printf("%d ", m[i]);
    i = i-m[i];
  }printf("\nR = %d\n", r[n]);
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 4
Pour une barre de longueur 4, voilà le découpage optimal :
2 2
R = 10
$ ./EXEC 5
Pour une barre de longueur 5, voilà le découpage optimal :
2 3
R = 13
$ ./EXEC 10
Pour une barre de longueur 10, voilà le découpage optimal :
10
R = 30
$ ./EXEC 7
Pour une barre de longueur 7, voilà le découpage optimal :
1 6
R = 18
$
```

Bibliographie

- R. ERRA, "Cours d'informatique", 1A-S2 2013-2014 ESIEA.
- Cormen, Leiserson, Rivest, Stein "Algorithmique", 3ème éd. DUNOD.
- Olivier Bournez, "Cours 5 : programmation dynamique", LIX, École polytechnique, 2011-2012.