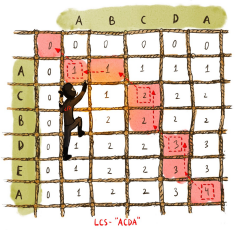
Programmation dynamique

Paradigme de conception
Procéder de récursion et de mémoisation
Propriété de sous-structure optimale

Suite Fibonacci
Coefficients binomiaux
Algorithme de Floyd Warshall



Auteur : Olivier Raynaud

Version de travail : rentrée 2020
Sensibilité :

Référence :

https://www.pinterest.fr/pin/495255290269224249/

Paradigme

❑ **Elément de définition**

Principe heuristique intuitive : La programmation dynamique est un paradigme de conception qui résout les problèmes en combinant les solutions de sous problems.

Ce concept a été introduit par Bellman, dans les années 50, pour résoudre des problème d'optimisation.

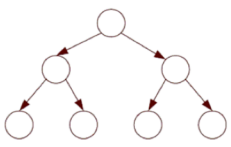
Richard Bellman
1920-1984

- La programmation dynamique se présente comme une adaptation de la méthode "diviser pour régner".
- Elle est applicable lorsque les sous-problèmes ne sont pas indépendants.

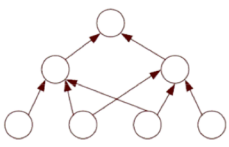
Paradigme de programmation dynamique

❑ Principe d'optimalité

Définition (Optimalité de Bellman) : Un problème possède la propriété de sous-structure optimale si une solution optimale contient la solution optimale des sous-problèmes.

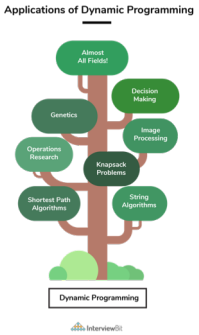


diviser et régner



programmation dynamique

Application



- Plus grande sous–séquence commune (LCS problem)
- Plus longue sous-sequence croissante commune
- Sac à dos (Knapsack problem)

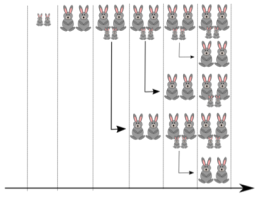
Cas d'étude Fibonacci

❑Suite de Fibonacci

Suite de Fibonacci : La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent.

Notée **F(n)** elle est définie par F(0) = 1; F(1) = 1 et F(n) = F(n-1) + F(n-2)

Leonardo Fibonacci
1170 – 1250 Pise



Wikipedia

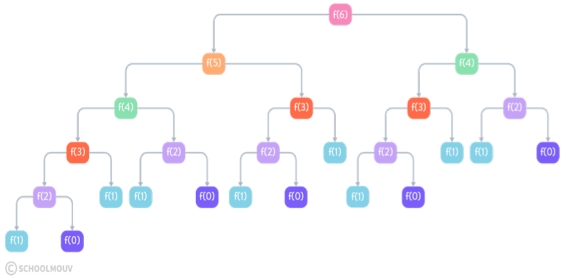
Cas d'étude Fibonacci

❑Algorithmique

Fibonacci(n)
si n=1 alors retourner 1;
si n=2 alors retourner 1;
sinon retourner *Fibonacci*(n-1) + *Fibonacci*(n-2);

Cas d'étude Fibonacci

❑Schéma d'exécution par approche récursive



Cas d'étude Fibonacci

❑Algorithmes de mémoisation et itérative

Fib(n) (version dite de mémoisation)
si n <=2 retourner 1;
si Fib(n) est dans la table retourner table[n];
sinon
table[n] ← Fib(n-1) + Fib(n-2);
retourner table[n];

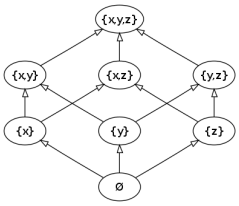
Fib(n) (version en programmation dynamique, itérative)
F[0] = 1; F[1] = 1;
Pour (i=2 à n) faire
F[i] = F[i-1] + F[i-2];
Retourner F[n];

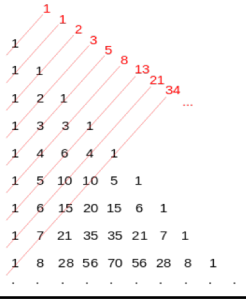
Ensemble des parties

❑*Triangle de Pascal (1623-1662)*

✓ **Le triangle** est une représentation des coefficients binomiaux

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$





Cas d'étude Coefficiens binomiaux

- Dénombrement des parties de taille k dans un ensemble de n éléments.

Coéfficients Binomiaux :

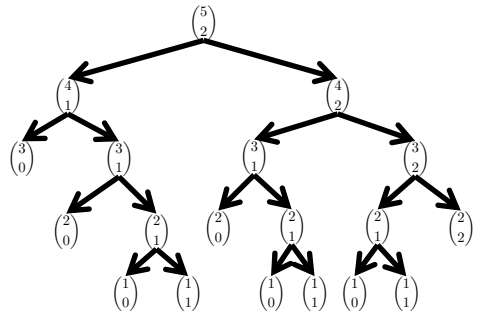
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad 0 < k < n$$

$$\binom{n}{k} = 1, \quad sinon$$

Binomial (n,k)
si (k=0 ou k=n) alors retourner 1;
sinon retourner Binomial(n-1,k-1) + Binomial(n-1,k)

Cas d'étude Coefficiens binomiaux

- Schéma d'exécution par approche récursive



Cas d'étude Coefficiens binomiaux

- Triangle de Pascal

k

n	0	1	2	3	4	5	6
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

Cas d'étude Coefficiens binomiaux

❑ Algorithme de programmation dynamique

```
Algorithm 1: Binomial()
    Données: n, k : entier ;
    Résultat: entier ;
    début
        Binomial[0..n, 0..k] : tableau d'entiers positifs (init à 0) ;
        Binomial[1, 0] ← 1 ; Binomial[1, 1] ← 1 ;

        pour (i = 2 à n) faire
            pour (j = 0 à min(i, k)) faire
                si (j = 0) ou (j = i) alors
                    Binomial[i, j] ← 1 ;
                sinon
                    Binomial[i, j] ← Binomial[i − 1, j − 1] + Binomial[i − 1, j] ;
                fin
            fin
        fin
        retourner Binomial[n, k] ;
    fin
```

Complexité : l'algorithme rempli la moitié du tableau de coté n et k, donc la complexité temporelle est de l'ordre de k.n .

Mise en oeuvre

❑ Le développement d'un algorithme de programmation dynamique peut être planifié dans une séquence de 4 étapes

Processus en 4 étapes :

1) Caractériser la structure d'une solution optimale;
2) Définir récursivement la valeur d'une solution optimale;
3) Calculer la valeur d'une solution optimale;
4) Calcul effectif de la solution optimale

Problème du plus court chemin

En théorie des graphes, le **problème** de **plus court chemin** est le **problème** algorithmique qui consiste à trouver un **chemin** d'un sommet à un autre de façon que la somme des poids des arcs de ce **chemin** soit minimale.

❑ Variante du problème pour toutes les paires de sommets

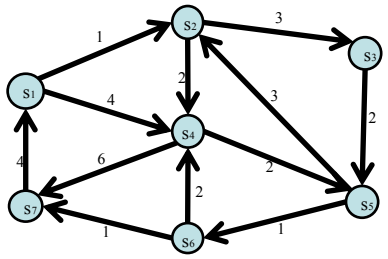
Problème :
Soit un graphe G=(S,A), S et A les ensembles de sommets et d'arcs de G. Le poids d'un arc a est un entier naturel noté l(a). La longueur d'un chemin est égale à la somme des longueurs des arcs qui le composent.

Question :

- Déterminer pour chaque couple de sommets (s_i,s_j), le plus court chemin, s'il existe, qui joint s_i à s_j .

Problème du plus court chemin

❑ Exemple d'instance



Graphe G = (S,A)

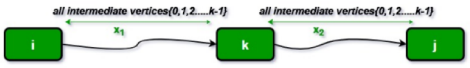
Problème du plus court chemin

❑ Algorithme de Floyd /Warshall

Etape 1 (Caractérisation de la structure d'une solution optimale) :
Si **f** est un chemin de longueur minimale joignant **x** à **y** et qui passe par **z**, alors il se décompose en deux chemins de longueur minimale. Le premier joint **x** à **z** et le second joint **z** à **y**.

On suppose les sommets numérotés : $s_1, s_2, \ldots s_n$ et pour tout k > 0, on considère la propriété P_k suivante:

- **$P_k(f)$**: Tous les sommets de f, autre que son origine et son extrémité, ont un indice strictement inférieur à k.

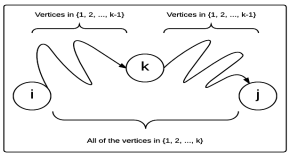


Problème du plus court chemin

❑ Algorithme de Floyd /Warshall

Etape 2 (définition récursive d'une solution optimale) :

$$\forall (s_i, s_j), \quad \delta_1(s_i, s_j) = l(s_i, s_j) \quad ou \quad \infty.$$
$$\delta_{k+1}(s_i, s_j) = min[\delta_k(s_i, s_j), \delta_k(s_i, s_k) + \delta_k(s_k, s_j)]$$



https://brilliant.org/wiki/floyd-warshall-algorithm/

❑**Etape 3** Calcul de la valeur de la solution optimale

Algorithme de Floyd /Warshall

Algorithm 2: *PlusCourtsChemins*()

Données: G : un graphe orienté valué ;
Résultat: $\delta[1..n, 1..n]$ matrice carrée des distances les plus courtes ;
début
  Initialisation de $\delta[]$ pour k valant 1 ;
  pour *(*$k = 2$ *à* n*)* **faire**
    pour *(*$i = 1$ *à* n*)* **faire**
      pour *(*$j = 1$ *à* n*)* **faire**
        $\delta[i,j] \leftarrow min(\delta[i,j]\ ,\ \delta[i,k]\ +\ \delta[k,j]))$;
      fin
    fin
  fin
  retourner $\delta[]$;
fin

Phase d'initialisation pour k=1

j

i	1	2	3	4	5	6	7
1	0	1		4			
2		0	3	2			
3			0		2		
4				0	2		6
5		3			0	1	
6				2		0	1
7	4						0

$$\forall (s_i, s_j), \quad \delta_1(s_i, s_j) = l(s_i, s_j) \quad ou \quad \infty.$$

Matrice finale correspondant à $\delta_k(s_i, s_j)$

j

i	1	2	3	4	5	6	7
1	0	1	4	3	5	6	7
2	10	0	3	2	4	5	6
3	8	5	0	5	2	3	4
4	8	5	8	0	2	3	4
5	6	3	6	3	0	1	2
6	5	6	9	2	4	0	1
7	4	5	8	7	9	10	0

$$\delta_{k+1}(s_i, s_j) = min[\delta_k(s_i, s_j), \delta_k(s_i, s_k) + \delta_k(s_k, s_j)]$$

Problème du plus court chemin

❑ **Etape 4** Calcul effecif de la solution optimale

Algorithme de Floyd /Warshall

Algorithm 3: *PlusCourtsChemins*()
Données: G : un graphe orienté valué ;
Résultat: $suivant[1..n, 1..n]$ matrice carrée des chemins effectifs les plus courts
début
  $\delta[1..n, 1..n]$ matrice carrée des distances les plus courtes ;
  Initialisation de $\delta[]$ pour k valant 1 ;
  pour *(k = 2 à n)* **faire**
    pour *(i = 1 à n)* **faire**
      pour *(j = 1 à n)* **faire**
        si $(\delta[i,j] > \delta[i,k] + \delta[k,j])$ **alors**
          $\delta[i,j] \leftarrow \delta[i,k] + \delta[k,j]$;
          $suivant[i,j] \leftarrow suivant[i,k]$;
        fin
      fin
    fin
  fin
  retourner *suivant*[] ;
fin

Problème du plus court chemin

Matrice « suivant[,] »

j

i	1	2	3	4	5	6	7
1	1	2	2	2	2	2	2
2	4	2	3	4	4	4	4
3	5	5	3	5	5	5	5
4	5	5	5	4	5	5	5
5	6	2	2	6	5	6	6
6	7	7	7	4	4	6	7
7	1	1	1	1	1	1	7

Problème du plus court chemin

❑ **Exemple d'algorithme pour exploiter la matrice "suivant[,]**

Algorithm 4: *ImpressionPlusCourtsChemins*()
Données: $suivant[1..n, 1..n]$: matrice carrée des chemins effectifs ; i, j : entier ;
Résultat: Impression à l'écran du plus court chemin entre i et j :
début
  $k \leftarrow i$;
  tant que *(k ≠ j)* **faire**
    écrire(k,") ;
    $k \leftarrow suivant[k,j]$;
  fin
fin

Pour résumer

❑ *Quelques points essentiels de l'exposé*

✓ Nous avons introduit le principe d'optimalité de Bellman:

"toute solution optimale s'appuie elle-même sur des sous-problèmes résolus localement de façon optimale"

✓ Nous avons décrit un processus en 4 étapes pour la conception d'algorithmes de programmation dynamique:

- ✓ Structure d'une solution optimale;
- ✓ Définition récursive d'une solution optimale;
- ✓ Calcul de la valeur de la solution optimale;
- ✓ Calcul effectif de la solution optimale;

✓ Nous avons mis en oeuvre ces principes pour les problèmes du calcul des nombres de Fibonacci, du calcul des coefficients binomiaux et pour le calcul des chemins minimaux dans un graphe orienté valué.

Bibliographie

- [HMU] **Introduction to Automata Theory, Langage and Computation** *J.E. Hopcroft, R. Motwani, J.D. Ullman* Edition Adison Wesley 2001;
- [CLM] **Introduction à l'algorithmique,** *T. Cormen, C. Leiserson, R. Rivest, Edition Dunod 2010;*
- [Wikipedia] **Multiples références dans le texte.**