La programmation dynamique

Programmation dynamique

- C'est une des plus vieilles techniques pour produire des algorithmes exacts plus efficaces que l'énumération exhaustive.

Programmation dynamique

- C'est une des plus vieilles techniques pour produire des algorithmes exacts plus efficaces que l'énumération exhaustive.

Principe (Bellman, 1949)

Composer une solution optimale du problème en combinant les solutions (optimales) de ses sous-problèmes.

Programmation dynamique

- C'est une des plus vieilles techniques pour produire des algorithmes exacts plus efficaces que l'énumération exhaustive.

Principe (Bellman, 1949)

Composer une solution optimale du problème en combinant les solutions (optimales) de ses sous-problèmes.

- En pratique :
  - Décomposer le problème en des sous-problèmes plus petits ;
  - Calculer les solutions optimales de tous ces sous-problèmes et les garder en mémoire.
  - Calculer la solution optimale à partir des solutions optimales des sous-problèmes

Plan

Suite de Fibonacci

- version récursive
- Version de la programmation dynamique

Un premier exemple : problème du stockage

Le plus court chemin dans un graphe

Récapitulatif

Codage des entiers

Plus précisément

Suite de Fibonacci

- version récursive
- Version de la programmation dynamique

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité
fonction Fib(n)

1. si (n=0) ou (n=1) alors retourner 1 ;
2. sinon retourner Fib(n-1) + Fib(n-2) ;

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité
fonction Fib(n)

1. si (n=0) ou (n=1) alors retourner 1 ;
2. sinon retourner Fib(n-1) + Fib(n-2) ;

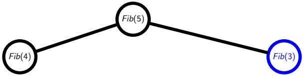
Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité
fonction Fib(n)

1. si (n=0) ou (n=1) alors retourner 1 ;
2. sinon retourner Fib(n-1) + Fib(n-2) ;



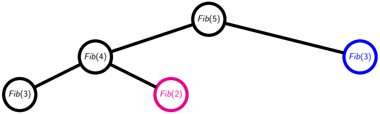
Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le *n*-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité
fonction Fib(n)

1. si (n=0) ou (n=1) alors retourner 1 ;
2. sinon retourner Fib(n-1) + Fib(n-2) ;



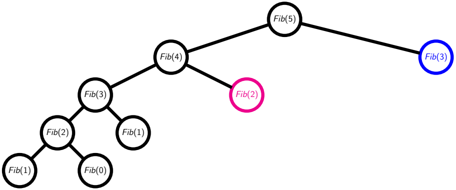
Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité
fonction Fib(n)

1. si (n=0) ou (n=1) alors retourner 1 ;
2. sinon retourner Fib(n-1) + Fib(n-2) ;



Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$

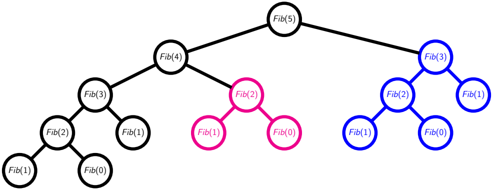
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité

fonction Fib(n)

1. si (n=0) ou (n=1) alors retourner 1 ;
2. sinon retourner Fib(n-1) + Fib(n-2) ;



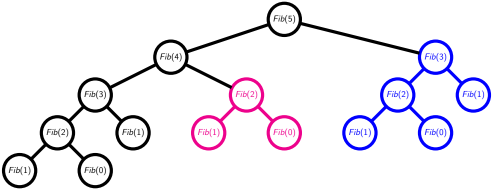
Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la récursivité
fonction Fib(n)

1. si (n=0) ou (n=1) alors retourner 1 ;
2. sinon retourner Fib(n-1) + Fib(n-2) ;



nb exponentiel d'appels récursifs : $\mathcal{O}(\phi^n)$ avec $\phi = (1+\sqrt{5})/2$.

Plus précisément

Suite de Fibonacci

- version récursive
- Version de la programmation dynamique

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :
fonction Fib-Dynamique (n)

1. F[0]=1 ; F[1]=1
2. pour i allant de 3 à n, faire F[i]=F[i-1] + F[i-2] ;
3. retourner $F[n]$

n	
$F[n]$	

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :
fonction Fib-Dynamique (n)

1. F[0]=1 ; F[1]=1
2. pour i allant de 3 à n, faire F[i]=F[i-1] + F[i-2] ;
3. retourner $F[n]$

n	0	1
$F[n]$	1	1

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :
fonction Fib-Dynamique (n)

1. F[0]=1 ; F[1]=1
2. pour i allant de 3 à n, faire F[i]=F[i-1] + F[i-2] ;
3. retourner $F[n]$

n	0	1	2
$F[n]$	1	1	2

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1, \; F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :
fonction Fib-Dynamique (n)

1. F[0]=1 ; F[1]=1
2. pour i allant de 3 à n, faire F[i]=F[i-1] + F[i-2] ;
3. retourner $F[n]$

n	0	1	2	3
$F[n]$	1	1	2	3

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :
fonction Fib-Dynamique (n)

1. F[0]=1 ; F[1]=1
2. pour i allant de 3 à n, faire F[i]=F[i-1] + F[i-2] ;
3. retourner $F[n]$

n	0	1	2	3	4
$F[n]$	1	1	2	3	5

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :
fonction Fib-Dynamique (n)

1. F[0]=1 ; F[1]=1
2. pour i allant de 3 à n, faire F[i]=F[i-1] + F[i-2] ;
3. retourner $F[n]$

n	0	1	2	3	4	5	...
$F[n]$	1	1	2	3	5	7	...

Suite de Fibonacci

Données : Un entier $t \in \mathbb{N}$
Objectif : Calculer le n-ième terme de la suite de Fibonacci

$$(F(0) = F(1) = 1,\ F(n) = F(n-1) + F(n-2)).$$

On peut le calculer en utilisant la programmation dynamique :
fonction Fib-Dynamique (n)

1. F[0]=1 ; F[1]=1
2. pour i allant de 3 à n, faire F[i]=F[i-1] + F[i-2] ;
3. retourner $F[n]$

n	0	1	2	3	4	5	...
$F[n]$	1	1	2	3	5	7	...

Complexité : n additions

Plan

Un premier exemple : problème du stockage.

Considérons n programmes $P_1, P_2, \ldots, P_n$ qui peuvent être stockés sur un disque dur de capacité D gigabytes.

- Chaque programme P_i a besoin s_i gigabytes pour être stocké.
- Tous les programmes ne peuvent pas être stockés sur le disque : $(\sum_{i=1}^{n} s_i > D)$

Objectif :

Concevoir un algorithme qui permet de maximiser **la quantité de données stockées** sur le disque.

Formulation sous-forme de problème d'optimisation

Données :

$\mathcal{P}$ un ensemble fini de programmes : $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$
v une valuation des éléments de $\mathcal{P}$: $v(P_i) = s_i$
Un entier D.

Solutions faisables :

Une solution faisable S est une partie de $\mathcal{P}$ telle que

$$v(S) = \sum_{e \in S} v(e) \leq D$$

Notons $\mathcal{F}$ l'ensemble des solutions faisables.

Objectif : Trouver $S \in \mathcal{F}$ qui maximise la quantité $v(S)$

Les algorithmes gloutons ne fonctionnent pas

1. Classer les programmes $P_1, P_2, \ldots, P_n$ en fonction de la taille du programme s_i : $s_1 \geq s_2 \geq \cdots \geq s_n$
2. Initialiser la recherche avec $S = \emptyset$
3. Pour $i = 1$ à n faire :

   Si $\sum_{P_j \in S} s_j + s_i \leq D$ alors $S \leftarrow S \cup \{P_i\}$
4. Retourner S

Les algorithmes gloutons ne fonctionnent pas

1. Classer les programmes $P_1, P_2, \ldots, P_n$ en fonction de la taille du programme s_i : $s_1 \geq s_2 \geq \cdots \geq s_n$
2. Initialiser la recherche avec $S = \emptyset$
3. Pour $i = 1$ à n faire :

   Si $\displaystyle\sum_{P_j \in S} s_j + s_i \leq D$ alors $S \leftarrow S \cup \{P_i\}$
4. Retourner S

Exemple : Considérons qu'on dispose 4 programmes de tailles 7, 5, 4, 1, et un disque dur de capacité $D = 10$.

L'algorithme retourne $S = \{P_1, P_4\}$
alors que la solution optimale est $\{P_2, P_3, P_4\}$.

Algorithme « force brute »

- Toutes les solutions sont des parties de l'ens. des programmes.
  Le nombre de solutions est au plus $2^{|\mathcal{P}|}$.

- L'algorithme « force brute » est l'algorithme suivant :
  1. pour toute partie S de l'ensemble $\mathcal{P}$
     - 1.1 tester si S est une solution faisable
     - 1.2 Si oui, conserver S si
       $v(S) = \max\{v(S') : S' \text{ solution déjà testée }\}$
  2. retourner la solution conservée ;

- La complexité de l'algorithme « force brute » est en $\mathcal{O}(2^{|\mathcal{P}|})$ opérations.

Exemple

Considérons l'exemple où 4 programmes sont de tailles 4, 7, 1, 5 et où $D = 10$.

- Pour les deux solutions $S_1 = \{5\}$ et $S_1 = \{1, 4\}$, la quantité des données stockées est identique.

Exemple

Considérons l'exemple où 4 programmes sont de tailles 4, 7, 1, 5 et où $D = 10$.

- Pour les deux solutions $S_1 = \{5\}$ et $S_1 = \{1, 4\}$, la quantité des données stockées est identique.

- $S[q]$: ensemble des solutions faisables S telles que $v(S) = q$

q	0	1	2	3	4	5	6	7	8	9	10
$S[q]$	$\emptyset$										

- Les ensembles $\{7, 5\}$, $\{7, 4\} \dots$ ne sont pas des solutions faisables.

Exemple

Considérons l'exemple où 4 programmes sont de tailles 4, 7, 1, 5 et où $D = 10$.

- Pour les deux solutions $S_1 = \{5\}$ et $S_1 = \{1, 4\}$, la quantité des données stockées est identique.

- $S[q]$: ensemble des solutions faisables S telles que $v(S) = q$

q	0	1	2	3	4	5	6	7	8	9	10
$S[q]$	$\emptyset$	$\{1\}$									

- Les ensembles $\{7, 5\}$, $\{7, 4\} \dots$ ne sont pas des solutions faisables.

Exemple

Considérons l'exemple où 4 programmes sont de tailles 4, 7, 1, 5 et où $D = 10$.

- Pour les deux solutions $S_1 = \{5\}$ et $S_1 = \{1, 4\}$, la quantité des données stockées est identique.

- $S[q]$: ensemble des solutions faisables S telles que $v(S) = q$

q	0	1	2	3	4	5	6	7	8	9	10
$S[q]$	$\emptyset$	$\{1\}$			$\{4\}$						

- Les ensembles $\{7, 5\}$, $\{7, 4\}$ ... ne sont pas des solutions faisables.

Exemple

Considérons l'exemple où 4 programmes sont de tailles 4, 7, 1, 5 et où $D = 10$.

- Pour les deux solutions $S_1 = \{5\}$ et $S_1 = \{1, 4\}$, la quantité des données stockées est identique.

- $S[q]$: ensemble des solutions faisables S telles que $v(S) = q$

q	0	1	2	3	4	5	6	7	8	9	10
$S[q]$	$\emptyset$	$\{1\}$			$\{4\}$	$\{5\}$ et $\{4, 1\}$					

- Les ensembles $\{7, 5\}$, $\{7, 4\}$ ... ne sont pas des solutions faisables.

Exemple

Considérons l'exemple où 4 programmes sont de tailles 4, 7, 1, 5 et où $D = 10$.

- Pour les deux solutions $S_1 = \{5\}$ et $S_1 = \{1, 4\}$, la quantité des données stockées est identique.

- $S[q]$: ensemble des solutions faisables S telles que $v(S) = q$

q	0	1	2	3	4	5	6	7	8	9	10
$S[q]$	$\emptyset$	$\{1\}$			$\{4\}$	$\{5\}$ et $\{4, 1\}$	$\{5, 1\}$				

- Les ensembles $\{7, 5\}$, $\{7, 4\}$ ... ne sont pas des solutions faisables.

Exemple

Considérons l'exemple où 4 programmes sont de tailles 4, 7, 1, 5 et où $D = 10$.

- Pour les deux solutions $S_1 = \{5\}$ et $S_1 = \{1, 4\}$, la quantité des données stockées est identique.

- $S[q]$: ensemble des solutions faisables S telles que $v(S) = q$

q	0	1	2	3	4	5	6	7	8	9	10
$S[q]$	$\emptyset$	$\{1\}$			$\{4\}$	$\{5\}$ et $\{4, 1\}$	$\{5, 1\}$	$\{7\}$	$\{7, 1\}$	$\{5, 4\}$	$\{5, 4, 1\}$

- Les ensembles $\{7, 5\}$, $\{7, 4\} \ldots$ ne sont pas des solutions faisables.

Concept

L'algorithme proposé est basé sur concept de la programmation dynamique.

Concept

L'algorithme proposé est basé sur concept de la programmation dynamique.

- Décomposer le problème en des sous-problèmes plus petits ;
  - Pour cela, on va résoudre dans cet ordre les problèmes
    - $\mathcal{P}_1 = \{4\}$, $\mathcal{P}_2 = \{4, 7\}$, $\mathcal{P}_3 = \{4, 7, 1\}$, $\mathcal{P}_3 = \{4, 7, 1, 5\}$,

Concept

L'algorithme proposé est basé sur concept de la programmation dynamique.

- Décomposer le problème en des sous-problèmes plus petits ;

  Considérer les problèmes $\mathcal{P}_i = \{s_1, \ldots, s_i\}$, $i = 1, \ldots, |\mathcal{P}|$

Concept

L'algorithme proposé est basé sur concept de la programmation dynamique.

- Décomposer le problème en des sous-problèmes plus petits ;

  Considérer les problèmes $\mathcal{P}_i = \{s_1, \ldots, s_i\}$, $i = 1, \ldots, |\mathcal{P}|$

- Calculer les solutions faisables de tous ces sous-problèmes.

  Pour le sous-problème $\mathcal{P}_1 = \{4\}$,

q	0	1	2	3	4	5	6	7	8	9	10
	$\emptyset$				$\{4\}$						

  Pour le sous-problème $\mathcal{P}_2 = \{4, 7\}$,

q	0	1	2	3	4	5	6	7	8	9	10
	$\emptyset$				$\{4\}$			$\{7\}$			

  Pour le sous-problème $\mathcal{P}_3 = \{4, 7, 1\}$,

q	0	1	2	3	4	5	6	7	8	9	10
	$\emptyset$	$\{1\}$			$\{4\}$	$\{4, 1\}$		$\{7\}$	$\{7, 1\}$		

Concept

L'algorithme proposé est basé sur concept de la programmation dynamique.

- Décomposer le problème en des sous-problèmes plus petits ;

  Considérer les problèmes $\mathcal{P}_i = \{s_1, \dots, s_i\}$, $i = 1, \dots, |\mathcal{P}|$

- Calculer les solutions faisables de tous ces sous-problèmes.

  Calculer les tableaux T_i qui stockent toutes les solutions faisables de $\mathcal{P}_i$.

Concept

L'algorithme proposé est basé sur concept de la programmation dynamique.

- Décomposer le problème en des sous-problèmes plus petits ;

  Considérer les problèmes $\mathcal{P}_i = \{s_1, \ldots, s_i\}$, $i = 1, \ldots, |\mathcal{P}|$

- Calculer les solutions faisables de tous ces sous-problèmes.

  Calculer les tableaux T_i qui stockent toutes les solutions faisables de $\mathcal{P}_i$.

- Calculer les solutions faisables à partir des solutions faisables des sous-problèmes

  Trouver la relation entre les tableaux T_i et T_{i+1}

Principe et Notation

- T_i : le tableau des sommes distinctes de tous les sous-ensembles de $\mathcal{P}_i = \{s_1, \ldots, s_i\}$.

  $T_i[j] = 1$ si et seulement si il existe une solution faisable S telle que $v(S) = j$.

  - Pour le sous-problème $\mathcal{P}_3 = \{4, 7, 1\}$,

q	0	1	2	3	4	5	6	7	8	9	10
$T_3[q]$	1	1	0	0	1	1	0	1	1	0	0
	$\emptyset$	$\{1\}$			$\{4\}$	$\{4, 1\}$		$\{7\}$	$\{7, 1\}$		

- Si le problème $\mathcal{P}_i$ a une solution faisable S alors le problème $\mathcal{P}_{i+1}$ a
  - la solution faisable S
  - et aussi la solution faisable $S \cup \{s_{i+1}\}$ si $v(S) + v(s_{i+1}) \leq D$.

Algorithme dynamique

$$\text{Entrée : } \begin{cases} \mathcal{P} : \text{ un ens. de programmes : } \mathcal{P} = \{P_1, P_2, \dots, P_n\} \\ v : \text{ une valuation des éléments de } \mathcal{P}, v(P_i) = s_i \\ D : \text{Un entier} \end{cases}$$

Sortie : un entier

1. pour j allant de 1 à D faire $T_0[j] \leftarrow 0$
2. $T_0[0] \leftarrow 1$ // la solution $S = \emptyset$ est une solution faisable
3. pour i allant de 1 à $|\mathcal{P}|$ faire
   3.1 pour j allant de 1 à D faire
      3.1.1 si $T_{i-1}[j] == 1$ alors $\begin{cases} T_i[j] \leftarrow 1 \\ T_i[j + s_i] \leftarrow 1 : \text{ si } j + s_i \leq D \end{cases}$
4. retourner la plus grande valeur j telle que $T_n[j] == 1$

Algorithme dynamique

$$\text{Entrée :} \begin{cases} \mathcal{P} : \text{ un ens. de programmes : } \mathcal{P} = \{P_1, P_2, \dots, P_n\} \\ v : \text{ une valuation des éléments de } \mathcal{P}, v(P_i) = s_i \\ D : \text{Un entier} \end{cases}$$

Sortie : un entier

1. pour j allant de 1 à D faire $T_0[j] \leftarrow 0$
2. $T_0[0] \leftarrow 1$ // la solution $S = \emptyset$ est une solution faisable
3. pour i allant de 1 à $|\mathcal{P}|$ faire
   3.1 pour j allant de 1 à D faire
      3.1.1 si $T_{i-1}[j] == 1$ alors $\begin{cases} T_i[j] \leftarrow 1 \\ T_i[j + s_i] \leftarrow 1 : \text{ si } j + s_i \leq D \end{cases}$
4. retourner la plus grande valeur j telle que $T_n[j] == 1$

Complexité : $\mathcal{O}(D \cdot |\mathcal{P}|)$

Remarque : Construction de la solution.

On peut construire en même temps la solution.

Entrée : $\begin{cases} T_i \text{ un ens. de tableaux :} \\ \mathcal{P} : \text{ un ens. de programmes : } \mathcal{P} = \{P_1, P_2, \ldots, P_n\} \\ v : \text{ une valuation des éléments de } \mathcal{P}, v(P_i) = s_i \\ D : \text{Un entier} \end{cases}$

Sortie : un ensemble d'entiers S

1. Trouver la valeur j telle que $j = argmax\{k : T_{|\mathcal{P}|}[k] = 1\}$
2. $S \rightarrow \emptyset$
3. pour i allant de $|\mathcal{P}|$ à 1 faire
   3.1 Si $T_{i-1}[j - v(s_i)] == 1$ alors $\begin{cases} S \leftarrow S \cup \{s_i\}; \\ j \leftarrow j - v(s_i); \end{cases}$
4. retourner S

Plan

Plus court chemin dans un graphe

Données :

$G = (V, E)$: un graphe orienté où chaque arc possède une longueur non-négative.

Objectif : Calculer la longueur plus court chemin entre toutes les paires de sommets.

Notation : On suppose que

- $V = \{1, \ldots, n\}$.
- G est donné sous forme de matrice $L[1...n, 1...n]$:
  - s'il n'y a pas d'arc allant de i à j, alors $L[i, j] = \infty$
  - sinon $L[i, j]$ correspond à la longueur de l'arc (i, j)

Principe

L'algorithme de Floyd-Warshall construit une matrice D_n qui donne la longueur du plus court chemin entre chaque paire de sommets.

1. On initialise D_0 à L ;
2. Après l'itération k, D_k donne la longueur du plus court chemin lorsque l'on utilise que les sommets dans $\{1, \ldots, k\}$ comme sommets intermédiaires (ou éventuellement aucun sommet intermédiaire).

Définition : D_k est la matrice D après l'itération k.

Récurrence

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$$

- Dans une séquence optimale de décisions, ou de choix,
  chaque sous-séquence doit aussi être optimale.

- En effet, si $(1,4)$, $(4,2)$ est le chemin le plus court entre 1 et 2, alors
  $(1,4)$ est le chemin le plus court entre 1 et 4
  et $(4,2)$ est le chemin le plus court entre 4 et 2.

- **Remarque** : cela ne marche pas pour les chemins les plus longs.

Algorithme de Floyd-Warshall

Entrée : L, la matrice du graphe G où $L[i,j]$ correspond à la longueur de l'arc (i,j).
Sortie : une matrice $n \times n$

- $D_0 \leftarrow L$;
- Pour k allant de 1 à n faire
  - Pour i allant de 1 à n faire
    - Pour j allant de 1 à n faire
      $D_k[i,j] \leftarrow min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$
- Retourner D_n.

Complexité : $\mathcal{O}(n^3)$ opérations

Plan

Récapitulatif

- Calcul de la suite de Fibonacci
  - Version récursive : $\mathcal{O}(\phi^n)$ opérations
  - Programmation dynamique : $\mathcal{O}(n)$ opérations

- Problème du stockage.
  - Algorithme « force brute » : $\mathcal{O}(2^{|\mathcal{P}|})$ opérations
  - Programmation dynamique : $\mathcal{O}(D \cdot |\mathcal{P}|)$ opérations

- Le plus court chemin dans un graphe
  - Programmation dynamique : $\mathcal{O}(n^3)$ opérations

Plan

Remarque : Codage des entiers

- Pour coder les données des instances du problème du stockage,

Données :
$\mathcal{P}$: un ensemble fini de programmes. ;
v : une valuation des éléments de $\mathcal{P}$;
D : un entier.

on a besoin de coder $(n+1)$ entiers.

- En informatique, les données (les entiers) sont codées par des 0 et des 1.

Combien faut-il de « bits » pour coder un entier i ?

Remarque : Codage des entiers

- Un entier i s'écrit en base b en utilisant des b chiffres allant de 0 à $b-1$:

  i s'écrit $c_n ... c_2 c_1 c_0$ en base b ssi $i = c_n b^n + ... + c_2 b^2 + c_1 b^1 + c_0 b^0$

- Par exemple :
  - $2406 = 2 \cdot 10^3 + 4 \cdot 10^2 + 0 \cdot 10^1 + 6 \cdot 10^0$ (en base 10)
  - $1001 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ (en base 2)

- Un entier i entre 0 et $b^n - 1$ peut être coder en n chiffres, ou en
  - $\lceil \log_b(i+1) \rceil$ chiffres
  - $\lceil \log_2(i+1) \rceil$ bits si $b = 2$

On peut le prouver par récurrence.

Le problème du stockage

L'instance du problème est codé avec

- $\mathcal{O}((n+1)D)$ bits si l'entier en codé **en unaire**,
- $\mathcal{O}((n+1)log_2 D)$ bits si l'entier en codé **en unaire**,

L'algorithme basé sur la programmation dynamique se réalise en $\mathcal{O}(D \cdot |\mathcal{P}|)$ opérations, c'est-à-dire

- en temps polynomial si l'entier en codé **en unaire**
- en temps exponentiel si l'entier en codé **en binaire**

car $\mathcal{O}(D) = \mathcal{O}(2^{log_2 D})$

Remarque : l'algorithme s'exécute en temps polynomial si les entiers sont « petits ».

Aujourd'hui

- Programmation dynamique
- Exemples : Suite de Fibonacci, problème du stockage, Le plus court chemin dans un graphe
- Le codage des entiers

La semaine prochaine :

introduction à la complexité et à la définition de problèmes difficiles.