⚠ À ne pas confondre prog dynamique et mémoïzat°.

906 Programmation dynamique : exemples et applications

faire du clavier

La programmation dynamique est un paradigme de programmation qui peut s'utiliser dans beaucoup de cas ; lorsque l'on a à faire des appels récursifs

bof bof bof

I Description du paradigme.

1| La suite de Fibonacci

On s'intéresse à $\begin{cases} F_{n+2} = F_{n+1} + F_n \\ F_0 = 0 \quad F_1 = 1 \end{cases}$ que l'on veut calculer

Algorithme naïf :

```
fibo (n)
  • si n=0 retourner 0
  • si n=1 retourner 1
  • sinon, retourner fibo(n-1) + fibo(n-2)
```

Défaut : cf Annexe 1

Pour calculer F_6, on fait 24 appels récursifs, et on va par exemple appeler 5 fois F_2 (en le recalculant à chaque fois) ; l'algorithme est exponentiel.

Solution : faire une approche ascendante plutôt que descendante.

```
fibis (n)
  F[0] = 0
  F[1] = 1
  pour k = 2 à n
    F[k] = F[k-1] + F[k-2]
  retourner F[n]
```

Cela donne un algorithme linéaire !

Préciser l'optimisat° mémoire.

2| Le paradigme : Important mais tard au changer l'intro

On a un problème dépendant d'une taille n.

On va chercher à lier ce problème à des sous-problèmes de taille inférieure (càd exprimer la solution de taille n en fonction des solutions de taille $k < n$) ; et on donne des solution de base (taille 0 ou 1).

On résout alors les sous-problèmes de la taille 0/1 à la taille n.

Efficace lorsque :

- les sous-problèmes utilisés sont nombreux et de taille comparable
- il y a de grandes chances de chevauchage avec un algorithme récursif.

Sous-problèmes courants : [P] p 167

- entrée de type $x_1, \dots, x_n$:
  - → sous-problème avec entrée $x_1, \dots, x_i$ $(i \leq n)$
  - → sous-problème avec entrée $x_i, \dots, x_j$ $(1 \leq i \leq j \leq n)$
- entrée arborescente :
  Sous-problème : un sous-arbre

Optimisation :

Lorsque c'est possible, afin de gagner en espace, on peut ne retenir les valeurs optimales que pour quelques derniers sous-problèmes calculés.

En faisant ça, on oublie l'entrée optimisant la solution. Il existe souvent et au cas par cas des algorithmes ne relevant pas de la programmation dynamique (que nous n'aborderons donc pas) permettant de retrouver cette entrée.

CACA

3] La mémoïzation [P] p 173 // 6.4

Problème du sac à dos

I l'est NP-complet

Entrée : n couples (poids, valeur) : $(p_1, v_1); \dots; (p_n, v_n)$; un poids W.

Sortie : $\max\limits_{a_1 p_1 + \dots + a_n p_n \leq W} a_1 v_1 + \dots + a_n v_n$

Sous-problème :

$$K(x) = \max_{a_1 p_1 + \dots + a_n p_n \leq x} a_1 v_1 + \dots + a_n v_n$$

Alors $K(x) = \max\limits_{\substack{1 \leq i \leq n \\ p_i \leq x}} (K(x - p_i) + v_i)$ et la sortie est $K(W)$.

Défaut :

Avec par exemple $(329, 60); (771, 150); (442, 30)$, et $W = 1337$

La programmation dynamique nous fait faire 1337 calculs, là où un algorithme récursif n'en fait que 21. (La complexité asymptotique de l'algorithme reste meilleure sur une entrée quelconque).

Solution : Cf Annexe 2

Il suffit de faire un algorithme récursif et de stocker chacun des résultats de sous-problèmes en mémoire : aucun des sous-problèmes n'est calculé deux fois ($\rightarrow$ meilleure complexité asymptotique) et on ne calcule que les sous-problèmes utiles ($\rightarrow$ meilleure complexité pratique).

Les optimisations mémoire de la programmation dynamique ne sont cependant plus appliquables ($\rightarrow$ moins bonne complexité spatiale).

Le choix mémoïzation/programmation dynamique doit donc se faire au cas par cas.

II Algorithmique des graphes [C] p 639 // 25.2

1] Algorithme de Floyd-Warshall [P] p 177 // 6.6

Entrée : graphe pondéré $G = (S, A)$ sans cycle de poids négatif $\quad l : A \rightarrow \mathbb{Z}$ $\quad S = \{1, \dots, n\}$

Sortie : $\forall u, t \in S$, $d(u,t)$ = distance minimale des chemins entre u et t

Sous-problème : $D_k(u,t)$ = distance du plus court chemin entre u et t ne passant que par $\{1, \dots, k\}$ en points intermédiaires

Initialisation : $D_0(u,u) = 0$; $\quad D_0(u,t) = \begin{cases} l(u,t) & \text{si } (u,t) \in A \\ \infty & \text{sinon} \end{cases}$

Récursion : $D_k(u,t) = \min(D_{k-1}(u,t),\ D_{k-1}(u,k) + D_{k-1}(k,t))$

Solution : $d(u,t) = D_n(u,t)$

Optimisation : le calcul de D_k ne requiert que D_{k-1} et pas les D_{k-i}, $i \geq 2$. On peut donc n'utiliser que deux tableaux au lieu de n.

Complexité : temporelle $O(|S|^3)$, spatiale $O(|S|^2)$

Applications :

En changeant les opérations min et + utilisées en récursion, et en modifiant légèrement les données, on a avec le même algorithme la clôture transitive d'une relation ou l'expression régulière d'un automate fini.

2] Algorithme de Bellman-Ford [C] p 603 // 24.1

Entrée : graphe pondéré $G = (S, A)$; $S = \{1, \dots, n\}$, $l : A \rightarrow \mathbb{Z}$

Sortie : $\forall t \in S$, $d(t)$ = distance du plus court chemin entre 1 et t

DÉVELOPPEMENT 1 : présentation de l'algorithme en $O(|S||A|)$ temporel et $O(|S|)$ spatial

3] Voyageur de commerce [P] p 178 // 6.6

Entrée : graphe pondéré $G=(S,A)$. $S=\{1,\dots,n\}$ $\ell : A \to \mathbb{Z}$

Sortie : longueur du plus court chemin allant de 1 à 1 et passant une unique fois par chaque sommet.

$\longrightarrow$ Le problème est NP-complet ; solution naïve en $O(n!)$

Sous-problème : $\forall E \subseteq S$, $1 \in E$, $\forall j \in E \setminus \{1\}$

$C(E,j)$ = longueur du plus petit chemin $1 \to \dots \to j$ passant une unique fois par chaque sommet de E et pas de $S \setminus E$

Initialisation : $\forall j \neq 1$, $C(\{1,j\},j) = \begin{cases} \ell(1,j) & \text{si } (1,j) \in A \\ \infty & \text{sinon} \end{cases}$

Récursion : à faire avec des valeurs croissantes de $|E|$

$$C(E,j) = \min_{\substack{i \neq j \\ i \neq 1}} \left(C(E \setminus \{j\}, i) + \ell(i,j) \right)$$

Solution : $\min_{1 < i \leq n} C(S,i) + \ell(i,j)$

Complexité : temporelle et spatiale : $O(n^2 2^n)$

III Algorithmique du texte $\to$ [P] pour moins de détails

1] Distance de Levenstein [Cro] p225 // 7.1

def : Soient x et y deux mots définis sur un alphabet Σ.
Soient $\mathrm{Del} : \Sigma \to \overline{\mathbb{N}}$, $\mathrm{Ins} : \Sigma \to \overline{\mathbb{N}}$, $\mathrm{Sub} : \Sigma^2 \to \overline{\mathbb{N}}$ des fonctions définissant un coût d'insertion, de suppression (Del) et de substitution.
La distance de Levenstein entre x et y est le coût minimum d'une suite d'opérations Del, Ins et/ou Sub pour passer de x à y.

Ex : * Si pour $a,b \in \Sigma$, $\mathrm{Del}(a) = \mathrm{Ins}(a) = \infty$, $\mathrm{Sub}(a,a) = 0$, $\mathrm{Sub}(a,b) = 1$, et x et y ont la même taille, on retrouve la distance de Hamming : le nombre de positions ayant une lettre différente entre x et y.

* Si $\mathrm{Del}(a) = \mathrm{Ins}(a) = \mathrm{Sub}(a,b) = 1$, $\mathrm{Sub}(a,a) = 0$, calcul de la distance entre MARIO et MALADE, introduction du graphe d'édition (cf Annexe 3)

2] Calcul pratique de la distance de Levenstein [Cro] p 232 // 7.2

Entrée : $x, y \in \Sigma^*$ de tailles m et n ; les fonctions Del ; Ins ; Sub

Sortie : $\mathrm{Lev}(x,y)$

Sous-problème : Soit $T[i,j] = \mathrm{Lev}(x[0,\dots,i], y[0,\dots,j])$ $\substack{-1 \leq i \leq m-1 \\ -1 \leq j \leq n-1}$

Initialisation : $T[-1,-1] = 0$

$T[i,-1] = T[i-1,-1] + \mathrm{Del}(x[i])$

$T[-1,j] = T[-1,j-1] + \mathrm{Ins}(y[j])$

Récursion : $T[i,j] = \min \begin{pmatrix} T[i-1,j-1] + \mathrm{Sub}(x[i], y[j]) \\ T[i-1,j] + \mathrm{Del}(x[i]) \\ T[i,j-1] + \mathrm{Ins}(y[j]) \end{pmatrix}$

Solution : $\mathrm{Lev}(x,y) = T[m-1, n-1]$

Complexité : temporelle : $O(mn)$
spatiale : $O(mn)$
$O(\min(m,n))$ si optimisé

DÉVELOPPEMENT 2 : Présentation, correction, complétude de l'algorithme ; lien avec PLSSC

3] Plus longue sous-suite commune [Cro] p 243 // 7.3

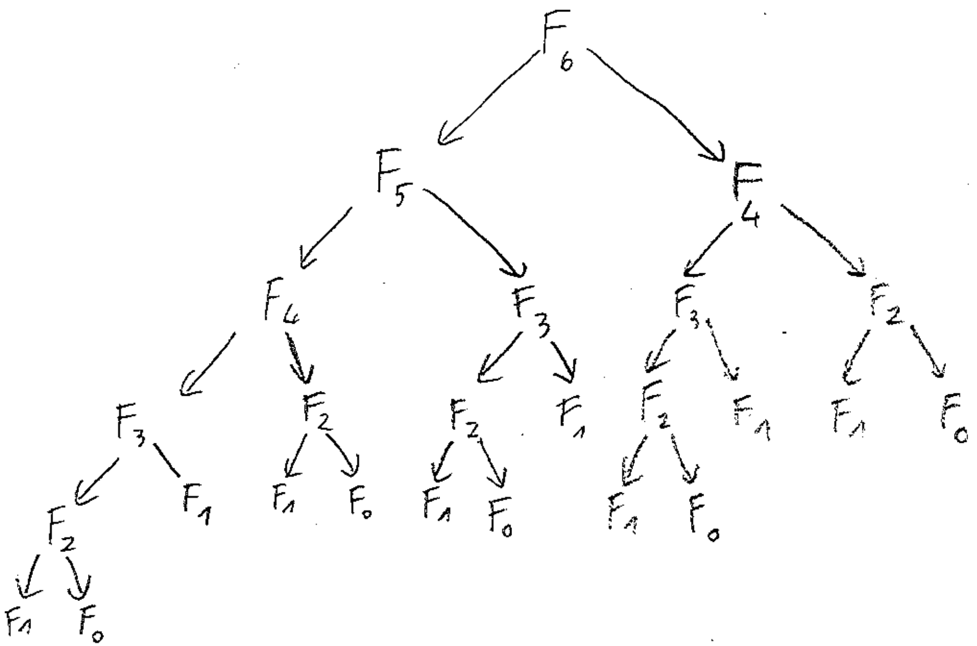
Si $\mathrm{Del} = \mathrm{Ins} = 1$, $\mathrm{Sub}(a,a) = 0$ et $\mathrm{Sub}(a,b) = \infty$,
Alors la longueur de la plus longue sous-suite commune entre x et y est $\frac{1}{2}\left(|x| + |y| - \mathrm{Lev}(x,y)\right)$
On peut aussi faire un algorithme avec la même création de sous-problème.

[P] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani, Algorithms

[C] Cormen-Leiserson-Rivest-Stein, Algorithmique

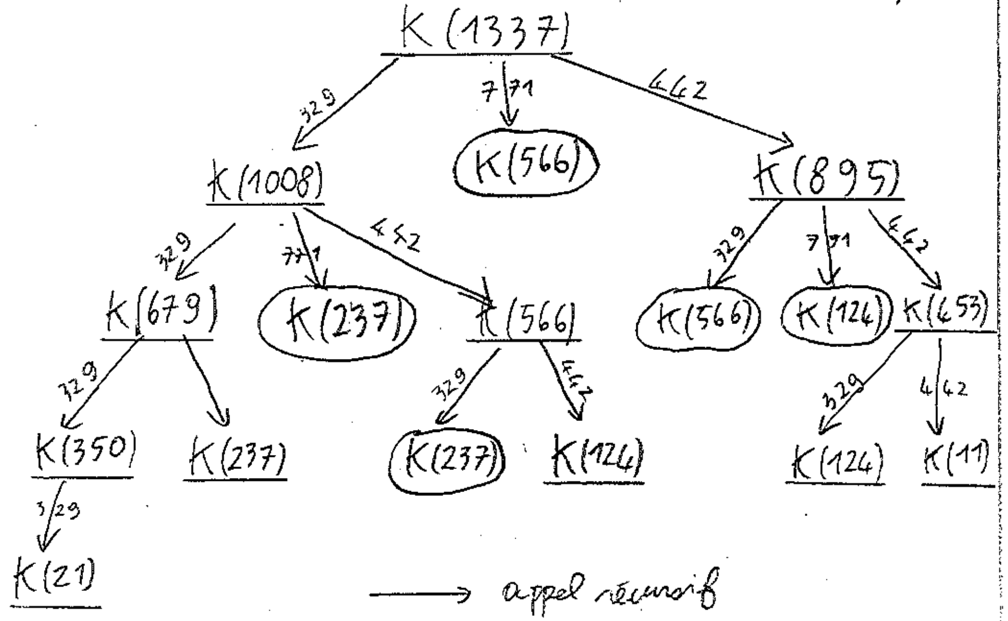
[Cro] M. Crochemore, C. Hancart, T. Lecroq, Algorithmique du texte

Annexe 1 : arbre de fibo (6)



Annexe 2 : calcul de K(1337)

en mémoïzant (de gauche à droite)

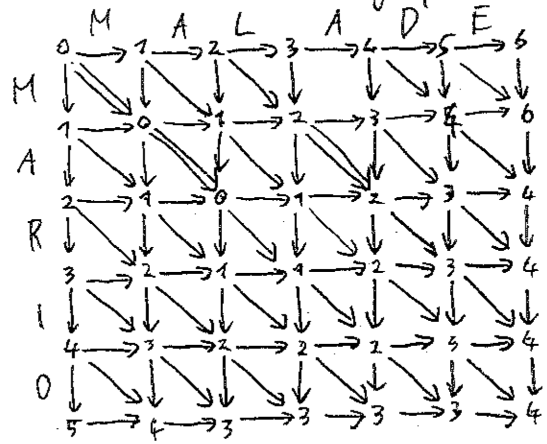


→ appel récursif

K(42) calcul fait par la machine

K(7) calcul déjà effectué, donc mémorisé.

Annexe 3 : Calcul de la distance entre MARIO et MALADE à l'aide du graphe d'édition.



Autres développements possible :

→ Floyd-Warshall (déjà dans la leçon)

→ PLSSC

→ CYK (décide si un mot est engendré par une grammaire sous forme de Chomsky)

sources variées : Carton, Hopcroft-Ullman, Floyd-Beigel

→ ABR optimaux (Cormen)

Autres algorithmes présentable :

classique et simple — Parenthésage de matrices → [P] ou [C]

Sac à dos sans répétition → [P]

Exercices de la section "programmation dynamique" du Cormen

Présenter B-F ≈ prog. dynamique, c'est pas classique (c'est pas fait ds les bouquins) → Du coup c'est original !

[Cormen p603 // §24.1]

Il faut bien expliquer et si on le met ds le plan, il faut écrire l'éq. récursive et le proposer en développement

Algorithme de Bellman-Ford

Problème : plus court chemin dans un graphe

Entrée : $G=(S,A)$ un graphe ; $S=\{1,\dots,n\}$; $\ell : A \to \mathbb{Z}$ une distance

Sortie : la longueur du plus court chemin de 1 à t, et ce pour tout sommet t, qu'on note $d(t)$

On utilise un raisonnement de programmation dynamique

⚠ taille = nb de sommets
longueur = somme des distances

Sous-problème : $D_k(t)$ = longueur du plus court chemin de taille $\leq k$ de 1 à t

Initialisation : $D_0(t) = \begin{cases} 0 & \text{si } t=1 \\ \infty & \text{sinon} \end{cases}$

Récursion : $D_k(t) = \min\left(D_{k-1}(t), \min\limits_{(v,t)\in A}\left(D_{k-1}(v)+\ell(v,t)\right)\right)$

⚠ À l'argumentaire !

Solution : * Si le graphe n'admet pas de cycle de négatif, ∃ UN chemin de taille minimale qui n'a pas de boucle, et donc est de taille au plus $n-1$. $d(t) = D_{n-1}(t)$ $(=D_n(t))$

* Si le graphe admet un cycle de longueur négative, plus la taille d'un chemin bouclant sur ce cycle sera importante, plus court (strictement) sera le chemin : certains points t vont donc vérifier $D_{n-1}(t) > D_n(t)$

ça demande plus d'explicat°s

⇒ si $\exists t$, $D_n(t) \neq D_{n-1}(t)$, renvoyer "il y a un cycle négatif"
sinon, renvoyer les D_{n-1}

<u>Complexité</u> temporelle naïve : $O(|S|^2 . A)$
intelligente : $O(|S| . |A|)$
spatiale : $O(|S|^2)$

naïf / intelligente : lorsque l'on fait pour tout sommet t le min sur les (u,t), on ne lit au final qu'une seule fois chaque arrête.

<u>Optimisation</u> : l'algorithme suivant renvoie le même résultat en $O(|S|.|A|)$ temporel, $O(|S|)$ spatial

Bellman-Ford $((S,A), \ell)$, $n := |S|$;
- D := tableau de taille n rempli avec des ∞ (indexé de 1 à n)
- $D[1] := 0$
- Pour $k = 1$ à $n-1$ faire
  - Pour $(u,v) \in A$ faire
    - $D[v] := \min(D[v], D[u] + \ell(u,v))$
- Pour $(u,v) \in A$ faire
  - Si $D[v] > D[u] + \ell(u,v)$
    - Renvoyer "il y a un cycle négatif" et s'arrêter
- <u>Renvoyer</u> D.

Terminaison : ni while, ni appel récursif.

Correction : On note $D^0_{(k)}$ la valeur de D au début de la boucle "Pour $k = 1$ à $n-1$",
et $D^{(u,v)}_{(k)}$ sa valeur le calcul pour l'arc (u,v)

Alors on a toujours $d(t) \leq D^0_{(k)}(t) \leq D_k(t)$
↳ de l'algorithme non optimisé.

1ère inégalité : il suffit de montrer que $D^{i}_{(k)}(t)$ correspond toujours à la longueur d'un chemin de 1 à t.
$\rightarrow$ récurrence immédiate

2e inégalité : on procède aussi par récurrence (sur k)
Init : les $D^{0}_{(k)}(t)$ sont égaux aux $D_{k}(t)$

Hérédité : Si tous les $D^{i}_{(k-1)}(t)$ sont plus petits que les $D_{k-1}(t)$

$$\text{Alors } D^{0}_{(k)}(t) = \min\left(D^{0}_{(k-1)}(t), \min_{(v,t)\in A}\left(D^{(v,t)}_{(k-1)}(v) + l(v,t)\right)\right)$$

$$\leq \min\left(D_{k-1}(t), \min_{(v,t)\in A}\left(D_{k-1}(v) + l(v,t)\right)\right)$$

$$\leq D_{k}(t)$$

On remarque alors que pour tout (v,w), $D^{(v,w)}_{(k)}(t) \leq D^{0}_{(k)}(t)$ par propriété du min, ce qui clôt la récurrence.

$\rightarrow$ 19/12 $\longrightarrow$ Ne pas jouer avec les cycles de poids négatifs !
[Du coup, le préciser qd on annonce la dist]
$\hookrightarrow$ ⚠ Beaucoup d'imprécisions !
↳ LA plus petite distance d'UN plus court chemin
↳ graphe ORIENTÉ
↳ Taille = Nombre d'arêtes
↳ Calcul de complexité AVANT l'algo ??!??
↳ Preuve de correction : on va montrer un INVARIANT

Question : Modifier l'algo pr trouver un plus court chemin
(À chaque fois, il faut se souvenir du nouveau père !)
↳ On stocke le PÈRE, et jamais tout le chemin
↳ Explicitemt : Il faut ajouter ceci pour trouver une solution au pb

Questions sur le plan :

- Voyageur de commerce : pdp : "difficulté spécifique"
  ↳ Expliquer cette difficulté ? "bof"

En fait, c'est similaire à d'autres pbs, comme le sac à dos.

- Est-ce qu'il y a d'autres pbs NP-complets que VdC
  ↳ Sac à dos → Pourquoi alors l'annoncer ĉ pseudo-polynômial ? (⚠ à la taille de l'entrée !) (quand c'est des nbres)

Il est NP-complet

Il faut raisonner sur la taille du nbre et non la valeur

- Est-ce qu'on peut comparer approche gloutonne et progᵐ dynamique ?
  ↳ Qui ex : sac à dos → les deux
  ↳ Ex. Dijkstra et B.F → lequel on choisit ?
  ça dépend de l'implémentatᵒ mais Dijkstra (i.e. glouton) est mieux (pas de PD)

Dijkstra est très restrictif quand même

⚠ Rappeler qu'on est sur des pbs d'optimisation

- Quels sont les dvpts proposés ? (Expliquer ce qu'on va faire ! et ce qu'on va admettre)

Rq : C'est bien de donner un cadre ds l'intro

⚠ À ne pas juste lister des exemples

Rq : Ça manque d'exemples où on peut constater l'optimisatᵒ possible en mémoire → C'est dommage → Il faut le faire ! (Regarder ds les exos !)

Pb NP-complets → les (mettre à part) séparer ?

Rq : Quand on énonce une complexité "il est polynômial en ..."
⚠ Classe de complexité "en la taille"