

2. Programmation dynamique

Pierre Karpman

Lycée Champollion MP Tronc Commun

Adapté d'un cours de Bruno Grenet

<https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/>

Table des matières

1. Introduction
2. Premier exemple : plus longue sous-suite croissante
3. Qu'est-ce que la programmation dynamique ?
4. Deuxième exemple : la distance d'édition

Table des matières

1. Introduction

2. Premier exemple : plus longue sous-suite croissante

3. Qu'est-ce que la programmation dynamique ?

4. Deuxième exemple : la distance d'édition

Problème d'optimisation

Quelle est le *coût* de la *meilleure* solution à ma question (et une telle solution) ?

- ▶ À distinguer de problèmes de *décision/recherche*
 - ▶ Est-ce qu'il *existe* une solution (et quelle est une telle solution) ?

Caractéristiques d'un problème d'optimisation

- ▶ Généralement facile de trouver une (ou beaucoup de) solution *quelconque*
 - ▶ Mais la contrainte d'optimalité complique les choses...
- ▶ Généralement facile à résoudre « inefficacement »
 - ▶ On calcule toutes les solutions et garde la meilleure

Type de problème apparaissant couramment « dans la vraie vie »

Problèmes d'optimisation : quoi ?

Exemples

- ▶ Trouver un plus-court-chemin dans un graphe
 - ▶ Depuis un sommet : par ex. algorithme « de Dijkstra »
 - ▶ (Peut se voir comme de la) programmation dynamique!
 - ▶ Entre toutes les paires de sommets : par ex. algorithme « de Floyd-Warshall »
 - ▶ (Peut se voir comme de la) programmation dynamique!
- ▶ Problème du « voyageur de commerce » : trouver un cycle de longueur minimale passant par un ens. de points (par ex. dans le plan)
 - ▶ La semaine prochaine en TP!
- ▶ Problème du « sac-à-dos » : maximiser la valeur du contenu d'un sac avec contrainte de poids
- ▶ Problème des œufs mystérieux : étant donnés q œufs, trouver l'étage seuil *le plus vite possible*
- ▶ Problème du rendu de monnaie

Problèmes d'optimisation : comment ?

Quelles approches

- ▶ Recherche exhaustive
 - ▶ Marche (en principe) toujours
 - ▶ Toujours inefficace (asymptotiquement)
- ▶ Approche gloutonne
 - ▶ Ne marche généralement pas : donne souvent des résultats faux (pas optimaux)!
 - ▶ Toujours efficace
- ▶ Programmation dynamique!
 - ▶ Marche toujours
 - ▶ Parfois efficace, parfois non...

Prog dyn. : comment, pourquoi ?

- ▶ « Comme un glouton, mais pas à horizon borné »
 - ▶ Ne rate pas de solution, mais possiblement plus coûteux
 - ▶ Formulation souvent récursive
 - ▶ Souvent avec de la redondance dans les sous-cas... à exploiter!

Glouton (pas très dynamique)



Crédit : Arlid Landa

Martre des pins = glouton dynamique ?



Crédit : Vince Smith

Table des matières

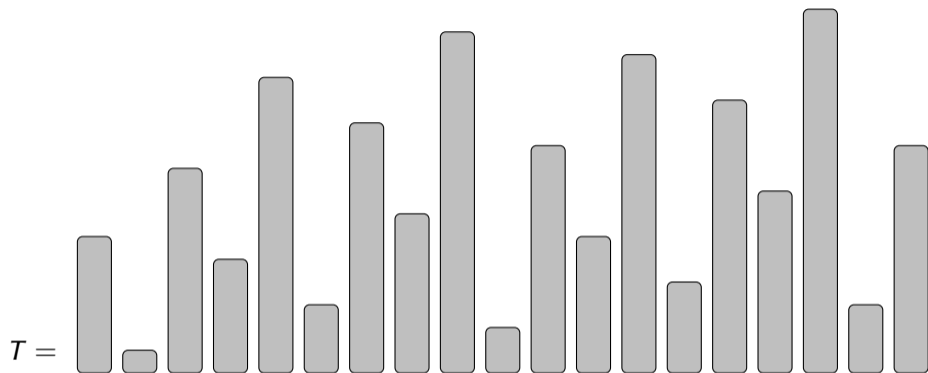
1. Introduction

2. Premier exemple : plus longue sous-suite croissante

3. Qu'est-ce que la programmation dynamique ?

4. Deuxième exemple : la distance d'édition

Plus longue sous-suite croissante



Définition

PLSSC(T): plus longue sous-suite croissante $T_{[i_1]} \leq T_{[i_2]} \leq \dots \leq T_{[i_k]}$

avec $0 \leq i_1 < i_2 < \dots < i_k < n$

Formalisation du problème

Entrée Un tableau/une liste T de n entiers

Sortie 1 La longueur d'une PLSSC de T

Sortie 2 Une PLSSC de T

Exemple précédent

Entrée $T = [6, 1, 9, 5, 13, 3, 11, 7, 15, 2, 10, 6, 14, 4, 12, 8, 16, 3, 10]$

Sortie 1 6

Sortie 2 $[6, 1, 9, 5, 13, 3, 11, 7, 15, 2, 10, 6, 14, 4, 12, 8, 16, 3, 10]$
ou $[6, 1, 9, 5, 13, 3, 11, 7, 15, 2, 10, 6, 14, 4, 12, 8, 16, 3, 10]$
ou ...

Quel(s) algorithm(e)s ?

- ▶ Recherche exhaustive : parcourir toutes les sous-suites $\rightsquigarrow O(2^n)$
- ▶ Algorithme de complexité polynomiale ?

Formule récursive pour PLSSC

- ▶ $\ell(T)$: longueur des PLSSC de T
- ▶ ℓ_i : longueur des PLSSC de T finissant en case $T_{[i]}$

Lemme

- ▶ $\ell(T) = \max_{0 \leq i < n} \ell_i$
- ▶ $\ell_i = \begin{cases} 1 & \text{si } i = 0 \\ 1 + \max\{\ell_j : j < i \text{ tels que } T_{[j]} \leq T_{[i]}\} & \text{pour } 1 \leq i < n \end{cases} \quad (\max(\emptyset) = 0)$

Preuve

Formule récursive pour PLSSC

- ▶ $\ell(T)$: longueur des PLSSC de T
- ▶ ℓ_i : longueur des PLSSC de T finissant en case $T_{[i]}$

Lemme

- ▶ $\ell(T) = \max_{0 \leq i < n} \ell_i$
- ▶ $\ell_i = \begin{cases} 1 & \text{si } i = 0 \\ 1 + \max\{\ell_j : j < i \text{ tels que } T_{[j]} \leq T_{[i]}\} & \text{pour } 1 \leq i < n \end{cases} \quad (\max(\emptyset) = 0)$

Preuve

Une PLSSC finit forcément, quelque part ; a taille au plus 1 si se termine à la première case ; admet forcément un préfixe strict qui est une PLSSC sinon (\leftarrow propriété de *sous-structure optimale*, cœur de la prog. dyn. !)

Traduction directe : algorithme récursif!

$$\ell_i = \begin{cases} 1 & \text{si } i = 0 \\ 1 + \max\{\ell_j : j < i \text{ tels que } T_{[j]} \leq T_{[i]}\} & \text{pour } 1 \leq i < n \end{cases}$$

ALGOL(T, i):

1. Si $i = 0$: Renvoyer 1
2. $\max \leftarrow 0$
3. Pour $j = 0$ à $i - 1$:
4. Si $T_{[j]} \leq T_{[i]}$:
5. $L \leftarrow \text{ALGOL}(T, j)$
6. Si $L > \max$: $\max \leftarrow L$;
7. Renvoyer $1 + \max$

Correction

- ▶ Immédiat d'après la formule

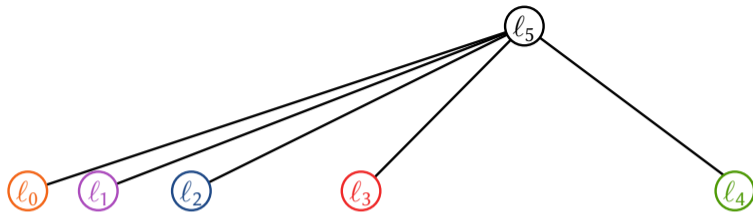
Complexité

Pire cas (quel cas?):

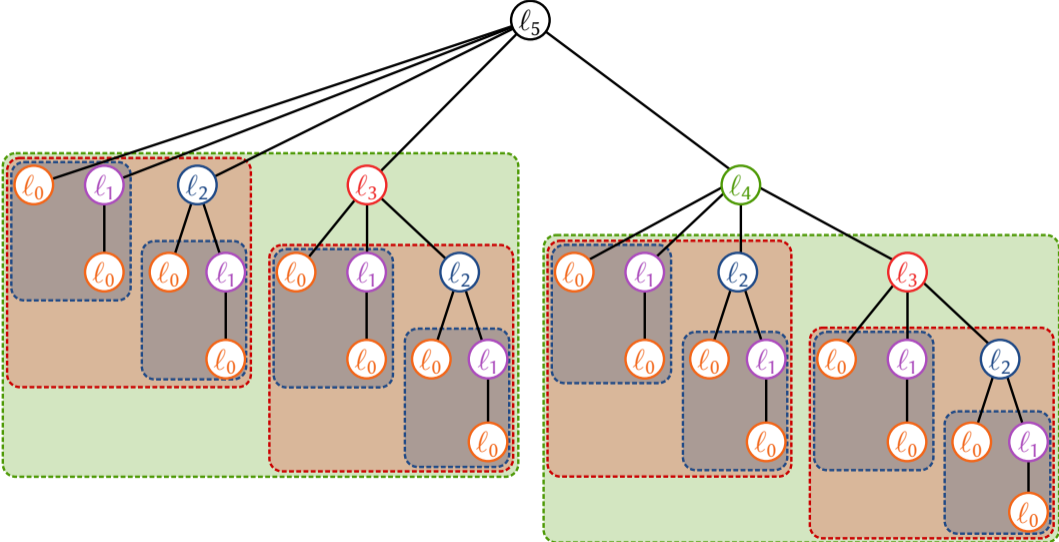
$$t(i) = \sum_{j < i} [t(j) + O(1)] \rightsquigarrow t(i) = \Theta(2^i)$$

- ▶ pareil que la recherche exhaustive!
- ▶ pourquoi?

Arbre des appels récursifs



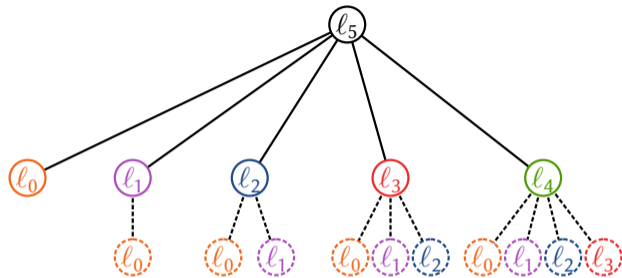
Arbre des appels récursifs



Mémoïsation : éviter les appels récursifs inutiles

MEMO(T, i, L):

1. Si $i = 0$: Renvoyer 1
2. Si i in $L[i]$: Renvoyer $L[i]$
3. $\max \leftarrow 0$
4. Pour $j = 0$ à $i - 1$:
5. Si $T[j] \leq T[i]$:
6. $L[j] \leftarrow \text{MEMO}(T, j, L)$
7. Si $L[j] > \max$: $\max \leftarrow L[j]$
8. Renvoyer $1 + \max$



Analyse de complexité

- ▶ $t(i) = \begin{cases} \sum_{j < i} [t(j) + O(1)] & \text{si } L[i] \text{ n'est pas encore connu} \\ O(1) & \text{sinon} \end{cases} \rightsquigarrow ???$
- ▶ Chaque $L[i]$ est calculé une seule fois, en $O(i)$ opérations *supplémentaires* $\rightsquigarrow O(n^2)$
 - ▶ Le calcul d'un $L[j]$ stocke tous les $L[<j]$
 - ▶ La récursion fait qu'on calcule $L[0]$ avant $L[1]$ avant ...

Version itérative : calcul des ℓ_i par i croissant

PLSSC-VALEUR(T) :

1. $L \leftarrow$ tableau de taille n
2. $L_{[0]} \leftarrow 1$
3. Pour $i = 1$ à $n - 1$:
4. $L_{[i]} \leftarrow 1 + \max\{L_{[j]} : j < i \text{ et } T_{[j]} \leq T_{[i]}\}$
5. Renvoyer $\max_i(L_{[i]})$

- ▶ On choisit un ordre de calcul garantissant que tous les $L_{[<i]}$ sont connus au moment de calculer $L_{[i]}$
- ▶ Plus besoin de récursion

Correction

- ▶ $L_{[i]}$ calculé exactement avec la formule $\rightarrow L_{[i]} = \ell_i$
- ▶ $\ell(T) = \max_i \ell_i$

Complexité


4. $\Leftrightarrow \max \leftarrow 0$
Pour $j = 0$ à $i - 1$:
Si $T_{[j]} \leq T_{[i]}$ et $L_{[j]} > \max$: $\max \leftarrow L_{[j]}$
 $L_{[i]} \leftarrow 1 + \max$

Obtenir la solution

Comment calculer une PLSSC (en plus de sa longueur) ?

- ▶ Retenir les indices des max
 - ▶ Économe en stockage
- ▶ Reconstruire *a posteriori*
 - ▶ Coût linéaire en la taille de la solution : efficace
- ▶ Fonctionne grâce à la propriété de sous-structure optimale

Exemple

$T =$ 0 2 9 3 5 8 1 4 6 7
 $L =$ 

Remarque

- ▶ Même principe que pour le calcul d'un plus-court-chemin avec « Dijkstra »

Algorithme PLSSC avec reconstruction

PLSSC⁺(T):

1. $L, P \leftarrow$ tableaux de taille n
2. $L_{[0]} \leftarrow 1$
3. Pour $i = 1$ à $n - 1$:
4. $L_{[i]} \leftarrow 1 + \max\{L_{[j]} : j < i \text{ et } T_{[j]} \leq T_{[i]}\}$
5. $P_{[i]} \leftarrow$ indice du max précédent
6. Renvoyer $\max_i(L_{[i]})$, l'indice du maximum et P

PLSSC-SOLUTION(T, ℓ, i, P):

1. $S \leftarrow$ tableau de taille ℓ
2. $S_{[\ell-1]} \leftarrow T_{[i]}$
3. Pour $j = \ell - 2$ à 0 :
4. $i \leftarrow P_{[i]}$
5. $S_{[j]} \leftarrow T_{[i]}$
6. Renvoyer S

Lemme

L'algorithme PLSSC-SOLUTION reconstruit une PLSSC de T en temps $O(n)$.

Table des matières

1. Introduction
2. Premier exemple : plus longue sous-suite croissante
3. Qu'est-ce que la programmation dynamique ?
4. Deuxième exemple : la distance d'édition

Idée générale

Programmation dynamique : récursion sans répétition pour la résolution de problèmes structurés

Ingrédients

- ▶ Formule **réursive** pour la valeur optimale
 - ▶ en fonction des valeurs optimales des sous-problèmes
 - ▶ sous-problèmes possiblement nombreux et *non disjoints*
- ▶ Algorithme **récuratif avec mémoïsation** ou **itératif** pour la valeur optimale
 - ▶ en commençant par les plus petits sous-problèmes
 - ▶ approche « *bottom-up* »
- ▶ Reconstruction de la solution ***a posteriori***
 - ▶ ajout d'informations à l'algorithme pour la valeur
 - ▶ algorithme de reconstruction indépendant (exploitant la structure des solutions)

« **Diviser pour régner** »

Sous-problèmes *disjoints*, approche « *top-down* »

Ingrédient 1: Formule récursive

Partie la plus importante (et difficile) !

Étapes

1. Spécification précise du problème
2. Formule récursive
 - ▶ Par une Analyse de cas

Exemple de PLSSC

1. Définition des ℓ_i et pas seulement $\ell(T)$
 2. Expression de ℓ_i en fonction des $\ell_j, j < i$
- ▶ Plusieurs cas à considérer

Ingrédient 2 : Algorithme pour la valeur

Partie plutôt facile... mais attention quand même !

Étapes

- ▶ choix d'une structure de données (*très souvent un tableau ou un dictionnaire*)
 - ▶ en fonction de si les sous-problèmes sont indexés par des entiers ou non
- ▶ écriture effective de l'algorithme
 - ▶ mémoïsation : assez direct
 - ▶ itératif : ordre de calcul si tableau multi-dimensionnel cf. ex. suivants
- ▶ analyse de complexité
 - ▶ mémoïsation : raisonnement global
 - ▶ itératif : assez classique

Exemple : PLSSC

- ▶ Tableau L
- ▶ Ordre croissant

Ingrédient 3 : Reconstruction

Partie de difficulté très variable !

Étapes

- ▶ ajout d'informations supplémentaires à l'algorithme pour la valeur
- ▶ *redescente* depuis la solution générale vers les instances petites

Exemple : PLSSC

- ▶ tableau P , indice i du maximum
- ▶ descente depuis $T_{[i]}$, en suivant P

En pratique, étape pas toujours effectuée

Problématique de la mémoire

Exemple de PLSSC

- ▶ Tableau P et L de taille n
 - ▶ Complexité *en espace* $O(n)$
 - ▶ Si $n = 2^{25}$: env. 1Mo de mémoire

En général

- ▶ La programmation dynamique est **gourmande en mémoire**
 - ▶ Parfois le *réactif limitant*
 - ▶ Si l'on n'a plus de mémoire le calcul échoue... il ne « suffit pas » d'attendre plus longtemps
 - ▶ Exemple : prochain TP
- ▶ Utile de limiter l'espace mémoire quand c'est possible
 - ▶ En général uniquement sans reconstruction
 - ▶ Formulation itérative plus adaptée

→ exemple suivant

Conclusion sur la programmation dynamique

Comparaison avec « diviser pour régner »

- ▶ Similitude : expression récursive de la solution
- ▶ Différences :
 - ▶ nombre de sous-problèmes : $O(1)$ en général en DPR
 - ▶ taille des sous-problèmes : division de la taille par une constante en DPR
 - ▶ écriture de l'algorithme directe en DPR
- ▶ Remarque : choix du type d'algorithme en fonction de la formule récursive

Comparaison avec « glouton »

- ▶ Similitude : résolution en se ramenant à un/des sous-problème
- ▶ Différences :
 - ▶ souvent *plusieurs* sous-problèmes traités
 - ▶ on résout *vraiment* les sous-problèmes
 - ▶ vision lointaine plutôt que « myope »

Dans quels contextes

- ▶ Souvent pour résoudre des problèmes d'optimisation (ex. précédent ; suivant...)
- ▶ Mais pas que (ex. : calcul des coefficients binomiaux (cf. TD ?))

Table des matières

1. Introduction
2. Premier exemple : plus longue sous-suite croissante
3. Qu'est-ce que la programmation dynamique ?
4. Deuxième exemple : la distance d'édition

Corrigeons les erreurs

À quelle distance se trouve-t-on du mot correct ?

HAGORRYTNE

ALGORITHME

La distance d'édition

Définition

La distance d'édition entre deux mots A et B est le nombre minimal de désaccords dans un alignement de A et B

HA GORRYT NE
ALGO RITHME

Définition du problème

Entrée Deux mots A et B sur un alphabet
(mot : chaîne de caractère ou tableau de caractères ou ...)

Sortie 1 La distance d'édition entre A et B

Sortie 2 Un alignement optimal de A et B

Utilité

- ▶ Orthographe :
 - ▶ Correcteur orthographique
 - ▶ Reconnaissance optique de caractères
 - ▶ Linguistique (proximité de langues)
- ▶ Bioinformatique :
 - ▶ similarité de séquences ADN
 - ▶ similarité d'arbres phylogénétiques
- ▶ ...

Formalisation et formule récursive

Distances entre préfixes

- ▶ $\Delta_{i,j}$: distance entre $A_{[0,i[}$ et $B_{[0,j[}$ où
 - ▶ $A_{[0,i[} = A_{[0}A_{[1} \cdots A_{[i-1}$
 - ▶ $B_{[0,j[} = B_{[0}B_{[1} \cdots B_{[j-1}$
- ▶ On veut $\Delta_{m,n}$ où $m = \#A$ et $n = \#B$

$$\Delta_{i,0} = i$$

$$\Delta_{0,j} = j$$

Lemme

$$\Delta_{i,j} = \min \begin{cases} \Delta_{i-1,j} + 1 \\ \Delta_{i,j-1} + 1 \\ \Delta_{i-1,j-1} + \delta \text{ où } \delta = 1 \text{ si } A_{[i-1} \neq B_{[j-1}, 0 \text{ sinon} \end{cases}$$

Preuve : alignements possibles

$A_{[0,i-1[}$	$A_{[i-1}$
$B_{[0,j[}$	-

$A_{[0,i[}$	-
$B_{[0,j-1[}$	$B_{[j-1}$

$A_{[0,i-1[}$	$A_{[i-1}$
$B_{[0,j-1[}$	$B_{[j-1}$

Algorithme : exemple

	H	A	G	O	R	R	Y	T	N	E	
	0	1	2	3	4	5	6	7	8	9	10
A	1										
L	2										
G	3										
O	4										
R	5										
I	6										
T	7										
H	8										
M	9										
E	10										

Algorithmme : exemple

		H	A	G	O	R	R	Y	T	N	E
	0	1	2	3	4	5	6	7	8	9	10
A	1	1	1	2	3	4	5	6	7	8	9
L	2	2	2	2	3	4	5	6	7	8	9
G	3	3	3	2	3	4	5	6	7	8	9
O	4	4	4	3	2	3	4	5	6	7	8
R	5	5	5	4	3	2	3	4	5	6	7
I	6	6	6	5	4	3	3	4	5	6	7
T	7	7	7	6	5	4	4	4	4	5	6
H	8	7	8	7	6	5	5	5	5	5	6
M	9	8	8	8	7	6	6	6	6	6	6
E	10	9	9	9	8	7	7	7	7	7	6

L'algorithme (itératif)

EDITION(A, B):

1. $(m, n) \leftarrow$ tailles de A et B
2. $\Delta \leftarrow$ tableau de dimensions $(m + 1) \times (n + 1)$
3. Pour $i = 0$ à m : $\Delta_{[i,0]} \leftarrow i$
4. Pour $j = 0$ à n : $\Delta_{[0,j]} \leftarrow j$
5. Pour $i = 1$ à m :
6. Pour $j = 1$ à n :
7. $\delta \leftarrow \begin{cases} 0 & \text{si } A_{[i-1]} = B_{[j-1]} \\ 1 & \text{sinon} \end{cases}$
8. $\Delta_{[i,j]} \leftarrow \min(\Delta_{[i-1,j]} + 1, \Delta_{[i,j-1]} + 1, \Delta_{[i-1,j-1]} + \delta)$
9. Renvoyer $\Delta_{[m,n]}$

Lemme

L'algorithme EDITION renvoie la distance entre A et B en temps $O(mn)$ et espace $O(mn)$.

Version efficace en mémoire

- Pour remplir la ligne i , on n'a besoin que des lignes i et $i - 1$

EDITIONMINMEMOIRE(A, B):

1. $(m, n) \leftarrow$ tailles de A et B
2. $P, C \leftarrow$ tableaux de dimension $n + 1$ *(lignes précédente et courante)*
3. Pour $j = 0$ à n : $P_{[j]} \leftarrow j$
4. Pour $i = 1$ à m :
5. $C_{[0]} \leftarrow i$
6. Pour $j = 1$ à n :
7. $\delta \leftarrow 0$ si $A_{[i-1]} = B_{[j-1]}$, 1 sinon
8. $C_{[j]} \leftarrow \min(P_{[j]} + 1, C_{[j-1]} + 1, P_{[j-1]} + \delta)$
9. Pour $j = 0$ à n : $P_{[j]} \leftarrow C_{[j]}$ *(courante \rightarrow précédente)*
10. Renvoyer $C_{[n]}$

Lemme

EDITIONMINMEMOIRE calcule la distance entre A et B en temps $O(mn)$ et espace $O(n)$.

Reconstruction : exemple

	H	A	G	O	R	R	Y	T	N	E	
	0	1	2	3	4	5	6	7	8	9	10
A	1	1	1	2	3	4	5	6	7	8	9
L	2	2	2	2	3	4	5	6	7	8	9
G	3	3	3	2	3	4	5	6	7	8	9
O	4	4	4	3	2	3	4	5	6	7	8
R	5	5	5	4	3	2	3	4	5	6	7
I	6	6	6	5	4	3	3	4	5	6	7
T	7	7	7	6	5	4	4	4	4	5	6
H	8	7	8	7	6	5	5	5	5	5	6
M	9	8	8	8	7	6	6	6	6	6	6
E	10	9	9	9	8	7	7	7	7	7	6

Algorithme de reconstruction

ALIGNEMENT(A, B, Δ):

1. $(i, j) \leftarrow$ tailles de A et B
2. Tant que $i > 0$ et $j > 0$:
3. $\delta \leftarrow 0$ si $A_{[i-1]} = B_{[j-1]}$, 1 sinon
4. Si $\Delta_{[i,j]} = \Delta_{[i-1,j-1]} + \delta$:
5. $(i, j) \leftarrow (i - 1, j - 1)$ $A_{[i-1]} \parallel B_{[j-1]}$
6. Sinon si $\Delta_{[i,j]} = \Delta_{[i-1,j]} + 1$:
7. Insérer « _ » en $j^{\text{ème}}$ position dans B ; $i \leftarrow i - 1$ $A_{[i-1]} \parallel _$
8. Sinon ($\Delta_{[i,j]} = \Delta_{[i,j-1]} + 1$):
9. Insérer « _ » en $i^{\text{ème}}$ position dans A ; $j \leftarrow j - 1$ $_ \parallel B_{[j-1]}$
10. Insérer j symboles « _ » en tête de A ou i symboles « _ » en tête de B
11. Renvoyer A et B

Lemme

L'algorithme ALIGNEMENT *aligne* les mots A et B de manière optimale, en temps $O(m + n)$.

Conclusion

Théorème

Soit A et B deux mots de tailles respectives m et n .

- ▶ La distance entre A et B peut être calculée en temps $O(mn)$ et espace $O(\min(m, n))$
- ▶ Un alignement de A et B peut être calculé temps et espace $O(mn)$

Peut-on faire mieux qu'un temps $O(mn)$?

- ▶ un peu... $\rightarrow O(mn / \log \max(m, n))$

Peut-on faire beaucoup mieux que $O(mn)$?

- ▶ sans doute pas (si on veut le résultat exact) \rightarrow borne inférieure *conditionnelle*

- ▶ Problème très important en pratique... et en théorie !
- ▶ Beaucoup de résultats très récents sur le sujet