Partie 2. Techniques algorithmiques

8. Programmation dynamique

Bruno Grenet

Université Grenoble Alpes – IM²AG

L3 Mathématiques et Informatique

UE Algorithmique

https://membres-ljk.imag.fr/Bruno.Grenet/Algorithmique.html

Table des matières

1. Premier exemple : plus longue sous-suite croissante

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

3. Deuxième exemple : la distance d'édition

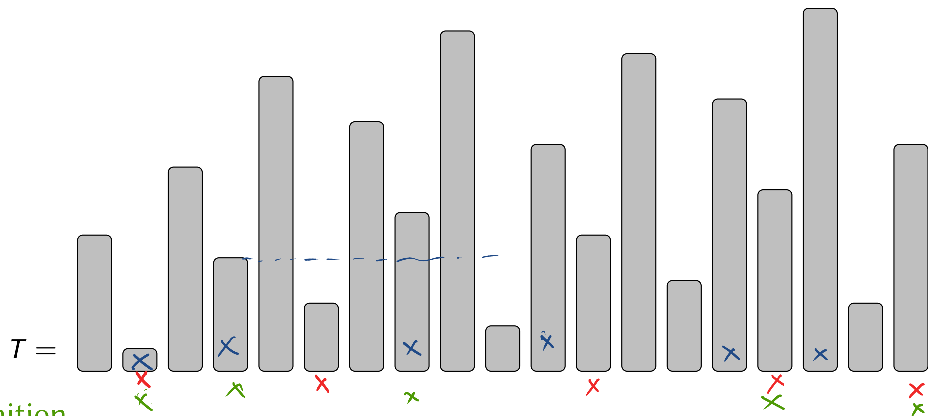
Table des matières

1. Premier exemple : plus longue sous-suite croissante

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

3. Deuxième exemple : la distance d'édition

Plus longue sous-suite croissante



Définition

PLSSC(T) : plus longue sous-suite croissante $T_{[i_1]} \leq T_{[i_2]} \leq \cdots \leq T_{[i_k]}$
avec $0 \leq i_1 < i_2 < \cdots < i_k < n$

Formalisation du problème

Entrée Un tableau 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) algorithme(s) ?

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

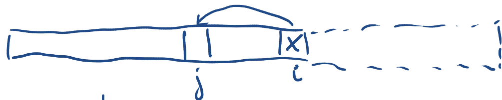
Formule récursive pour PLSSC

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

Lemme

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

Preuve



La longueur d'une PLSSC qui finit en case i est 1 + lgr d'une PLSSC qui finit en case j

$\ell_i = 1 + \ell_j$ où $\begin{cases} T_{[j]} \le T_{[i]} \\ \ell_j \text{ est la plus longue parmi les } \ell_k,\ k < i \end{cases}$

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. $L \leftarrow$ ALGOL(T, j)
6. Si $L > \max$: $\max \leftarrow L$;
7. Renvoyer $1 + \max$

Correction

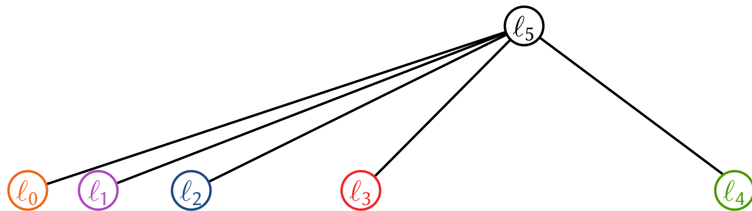
- Immédiat d'après la formule

Complexité

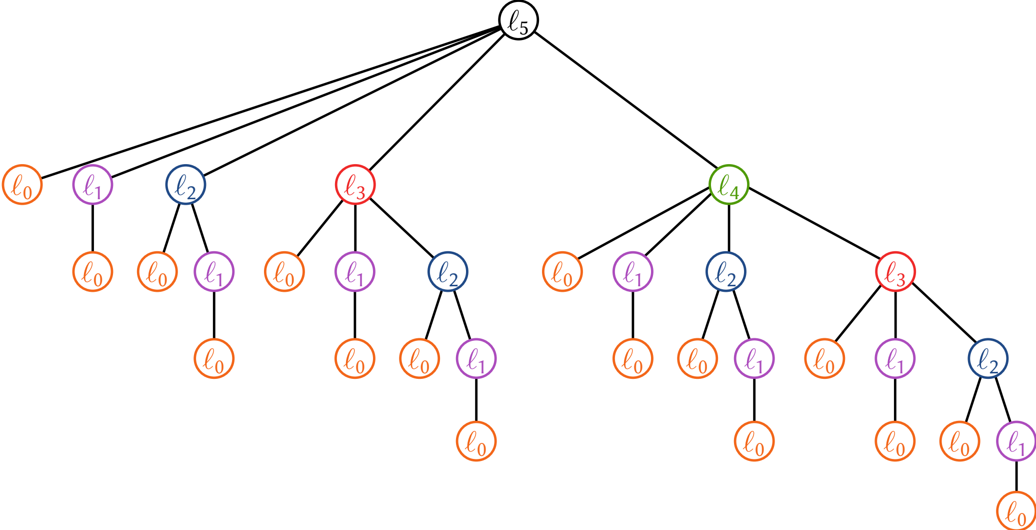
$t(i) \leq \sum_{j<i} t(j) \rightsquigarrow t(i) = O(2^i)$

- pareil que la recherche exhaustive !
- pourquoi ?

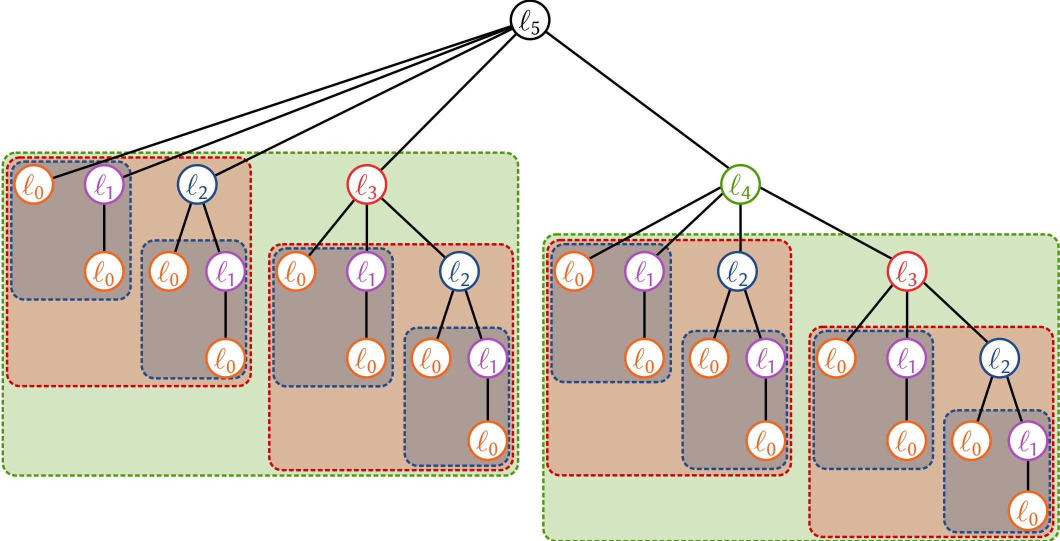
Arbre des appels récursifs



Arbre des appels récursifs



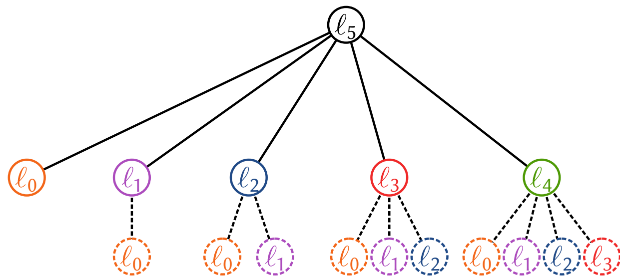
Arbre des appels récursifs



Mémoïsation : éviter les appels récursifs inutiles

MEMO(T, i, L) :

1. Si $i = 0$: Renvoyer 1
2. Si $L_{[i]} \neq$ None : Renvoyer $L_{[i]}$
3. max $\leftarrow 0$
4. Pour $j = 0$ à $i - 1$:
5. Si $T_{[j]} \leq T_{[i]}$:
6. $L_{[j]} \leftarrow$ MEMO(T, j, L)
7. Si $L_{[j]} >$ max : max $\leftarrow L_{[j]}$
8. Renvoyer 1 + max



Analyse de complexité

- $t(i) = \begin{cases} \sum_{j<i} t(j) & \text{si } L_{[i]} \text{ n'est pas encore connu} \\ O(1) & \text{sinon} \end{cases}$ $\rightsquigarrow$???
- Chaque $L_{[i]}$ est calculé une seule fois, en $O(i)$ opérations $\rightsquigarrow O(n^2)$

Coût total : $\Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta(n^2)$

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. $\quad L_{[i]} \leftarrow 1 + \max\{L_{[j]} : j < i \text{ et } T_{[j]} \leq T_{[i]}\}$
5. Renvoyer $\max_i(L_{[i]})$

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 $i - 1$:

$\quad$ Si $T_{[j]} \leq T_{[i]}$ et $L_{[j]} > \max$: $\max \leftarrow L_{[j]}$

$L_{[i]} \leftarrow 1 + \max$

$\Theta(i)$

total : $\Theta(\sum_i i) = \Theta(n^2)$

Obtenir la solution

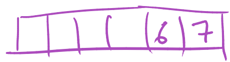
Comment calculer une PLSSC (en plus de sa longueur) ?

- Retenir les indices des max
- Reconstruire *a posteriori*

Exemple

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

$L =$ 1 2 3 3 4 5 2 4 5 6



→ 0, 2, 3, 4, 6, 7

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. $\quad L_{[i]} \leftarrow 1 + \max\{L_{[j]} : j < i \text{ et } T_{[j]} \leq T_{[i]}\}$
5. $\quad P_{[i]} \leftarrow$ 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. $\quad i \leftarrow P_{[i]}$
5. $\quad S_{[j]} \leftarrow T_{[i]}$
6. Renvoyer S

Lemme

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

Table des matières

1. Premier exemple : plus longue sous-suite croissante

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

3. Deuxième exemple : la distance d'édition

Idée générale

Programmation dynamique = récursion sans répétition

Ingrédients

- Formule **récursive** pour la valeur optimale
  - en fonction des valeurs de sous-problèmes
  - sous-problèmes possiblement nombreux et non disjoints
- Algorithme **récursif 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

« 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

Exemple de PLSSC

1. Définition des ℓ_i et pas seulement $\ell(T)$
2. Expression de ℓ_i en fonction des ℓ_j, $j < i$

En pratique, étape souvent (très) guidée

Ingrédient 2 : Algorithme pour la valeur

Partie plutôt facile… mais attention quand même !

Étapes

- choix d'une structure de données (*très* souvent un tableau)
- 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

En pratique, étape souvent non guidée

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*
- Utile de limiter l'espace mémoire quand c'est possible
  - En général uniquement sans reconstruction
  - Formulation itérative plus adaptée

$\rightarrow$ 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

D'où vient ce nom ?

Programmation : planification, ordonnancement

Dynamique : « *it's impossible to use the word dynamic in a pejorative sense* »

- Bellman (1940) : travaux en optimisation mathématique
- Origine en fait peu claire : référence à la programmation linéaire, et/ou problèmes de financements (*cf* Wikipédia)

Table des matières

1. Premier exemple : plus longue sous-suite croissante

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

3. Deuxième exemple : la distance d'édition

Corrigeons les erreurs

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

HAGORRYTNE

L I H M

distance 6

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

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

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

$\Delta_{i-1,j} + 1$

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

$\Delta_{i,j-1} + 1$

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

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

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	1	1	2	3	4	5	6	7	8	9
L	2	2	2	2	. - - -						
G	3										
O	4										
R	5										
I	6										
T	7										
H	8										
M	9										
E	10										

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	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. $\quad C_{[0]} \leftarrow i$
6. $\quad$ Pour $j = 1$ à n :
7. $\quad\quad \delta \leftarrow 0$ si $A_{[i-1]} = B_{[j-1]}$, 1 sinon
8. $\quad\quad C_{[j]} \leftarrow \min(P_{[j]} + 1, C_{[j-1]} + 1, P_{[j-1]} + \delta)$
9. $\quad$ 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

HA_GORRYT_NE

_ALGOR_ITHME

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. $\quad\delta \leftarrow 0$ si $A_{[i-1]} = B_{[j-1]}$, 1 sinon
4. $\quad$Si $\Delta_{[i,j]} = \Delta_{[i-1,j-1]} + \delta$:
5. $\quad\quad(i, j) \leftarrow (i-1, j-1)$ $\qquad A_{[i-1]} \| B_{[j-1]}$
6. $\quad$Sinon si $\Delta_{[i,j]} = \Delta_{[i-1,j]} + 1$:
7. $\quad\quad$Insérer « _ » en $j^{\text{ème}}$ position dans B; $i \leftarrow i - 1$ $\qquad A_{[i-1]} \| _$
8. $\quad$Sinon ($\Delta_{[i,j]} = \Delta_{[i,j-1]} + 1$) :
9. $\quad\quad$Insérer « _ » en $i^{\text{ème}}$ position dans A; $j \leftarrow j - 1$ $\qquad _ \| B_{[j-1]}$
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 (2023 !) sur le sujet