

Cours 2: Récursivité et programmation dynamique

- Récursivité, mémoïsation, programmation dynamique
- Distance minimale, diviser pour régner
- Sous-suites croissantes, mémoïsation
- Chemin dans un DAG, programmation dynamique
- Sac à dos, produits de matrices, alignements

Cours 2: Récursivité et programmation dynamique

- Récursivité, mémoïsation, programmation dynamique
- Distance minimale, diviser pour régner
- Sous-suites croissantes, mémoïsation
- Chemin dans un DAG, programmation dynamique
- Sac à dos, produits de matrices, alignements

Récurtivité, m emoisation, programmation dynamique

Donn ee: Une r ecurrence.

Probl eme: Calculer les premiers termes.

L'Exemple, **Fibonacci:** $f(0) = f(1) = 1, f(n) = f(n-1) + f(n-2)$.

Calcul r ecursif direct: nb d'appels r ecursifs exponentiel...

```
F:=proc(n) if (n=0 or n=1) return 1 else return f(n-1)+f(n-2);
```

(exercice: combien d'appels   $f(0)$ et $f(1)$?)

M emoisation:

- chaque fois qu'on a calcul  une valeur faire un m emo
- avant chaque calcul consulter les m emos.

```
F:=proc(n) if exists(M[n]) return M[n];  
           if (n=0 or n=1) M[n]:=1 else M[n]:=F(n-1)+F(n-2);  
           return M[n];
```

Programmation dynamique: prevoir les m emos (sous-probl emes) utiles et organiser les calculs it erativement.

```
M[0]:=0; M[1]:=0; for i=2..n do M[i+1]:=M[i]+M[i-1] od;
```

Cours 2: Récursivité et programmation dynamique

- Récursivité, mémoïsation, programmation dynamique
- Distance minimale, diviser pour régner
- Sous-suites croissantes, mémoïsation
- Chemin dans un DAG, programmation dynamique
- Sac à dos, produits de matrices, alignements

MIN DISTANCE: diviser pour régner

Donnée: E un ensemble de points

Problème: Déterminer $\min_{x,y \in E} d(x,y)$.

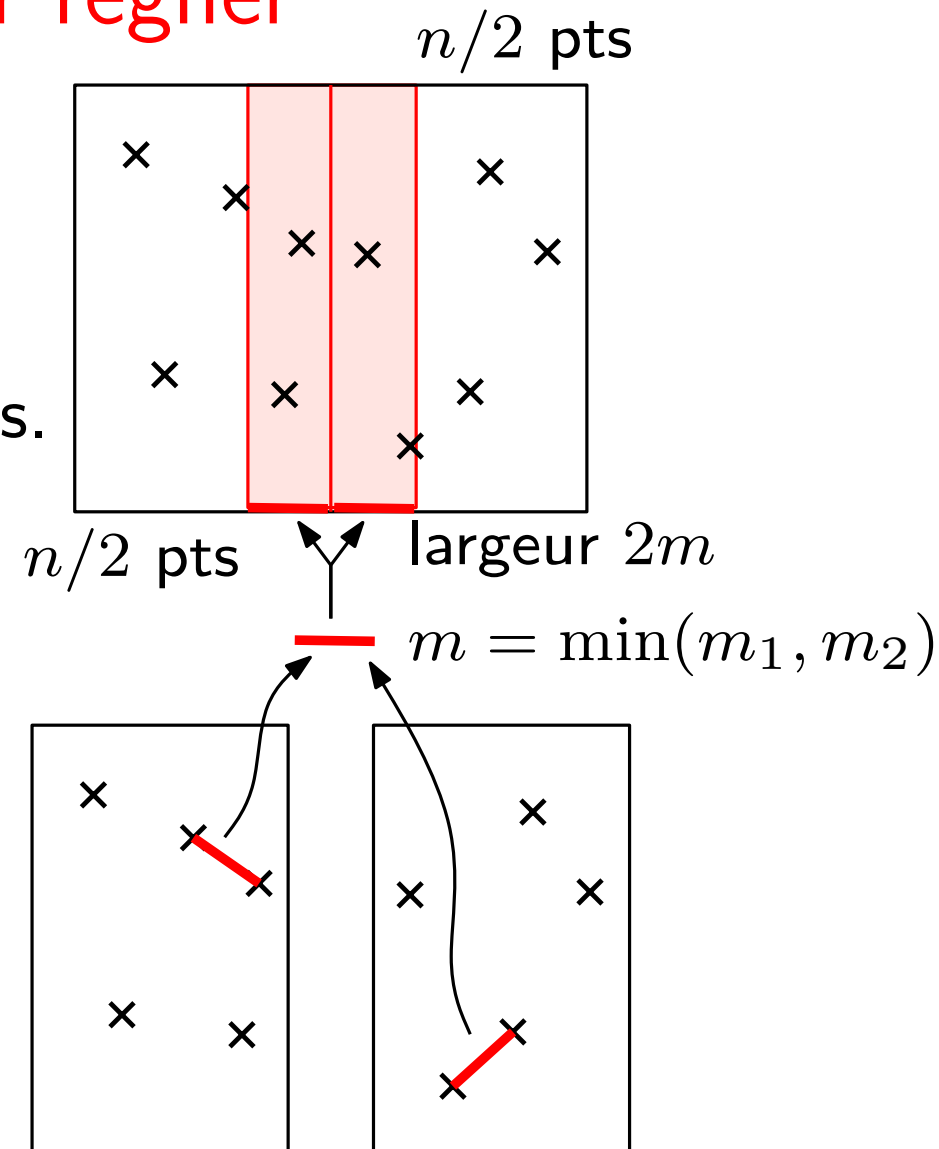
Algo exhaustif: calculer $O(n^2)$ distances.

Diviser pour régner: coût $C(n)$?

$$C(n) = 2C(n/2) + O(n \log n)$$

- Diviser l'espace de recherche en 2 parties. Tri en $O(n \log n)$
- Résoudre le sous-problème dans chaque partie. $2C(n/2)$
- Recoller les solutions partielles.

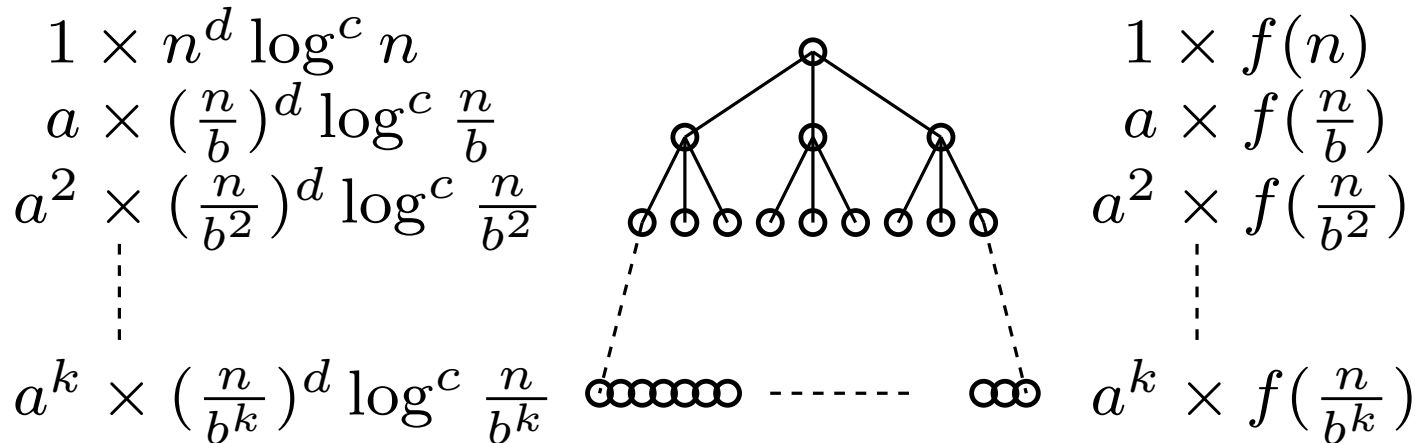
Tri vertical des pts de la bande, puis comparaison de chaque sommet aux k suivants (montrer que $k = 3$ suffit): $O(n \log n)$.



Analyse des récurrences diviser pour régner

Théorème. Soit $a > 0$, $b > 1$, et $d \geq 0$, et supposons que $T(n) = aT(\lceil n/b \rceil) + f(n)$ avec $f(n) = O(n^d \log^c n)$ alors

$$T(n) = \begin{cases} O(n^d \log^c n) & \text{si } d > \log_b a \\ O(n^d \log^{c+1} n) & \text{si } d = \log_b a \\ O(n^{\log_b a} \log^c n) & \text{si } d < \log_b a. \end{cases}$$



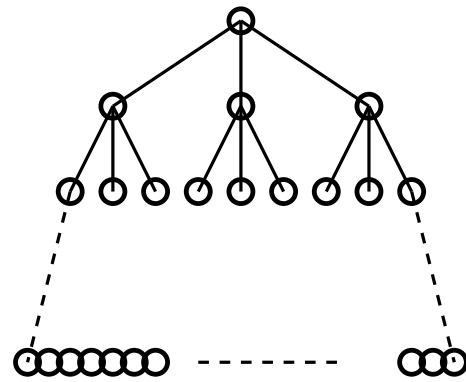
profondeur k tq $\frac{n}{b^k} \approx 1$, ie $k \approx \log_b n$.

Analyse des récurrences diviser pour régner

Théorème. Soit $a > 0$, $b > 1$, et $d \geq 0$, et supposons que $T(n) = aT(\lceil n/b \rceil) + f(n)$ avec $f(n) = O(n^d \log^c n)$ alors

$$T(n) = \begin{cases} O(n^d \log^c n) & \text{si } d > \log_b a \\ O(n^d \log^{c+1} n) & \text{si } d = \log_b a \\ O(n^{\log_b a} \log^c n) & \text{si } d < \log_b a. \end{cases}$$

$$\begin{aligned} & 1 \times n^d \log^c n \\ & a \times \left(\frac{n}{b}\right)^d \log^c n \\ & a^2 \times \left(\frac{n}{b^2}\right)^d \log^c n \\ & \vdots \\ & a^k \times \left(\frac{n}{b^k}\right)^d \log^c n \end{aligned}$$



$$\begin{aligned} & 1 \times f(n) \\ & a \times f\left(\frac{n}{b}\right) \\ & a^2 \times f\left(\frac{n}{b^2}\right) \\ & \vdots \\ & a^k \times f\left(\frac{n}{b^k}\right) \end{aligned}$$

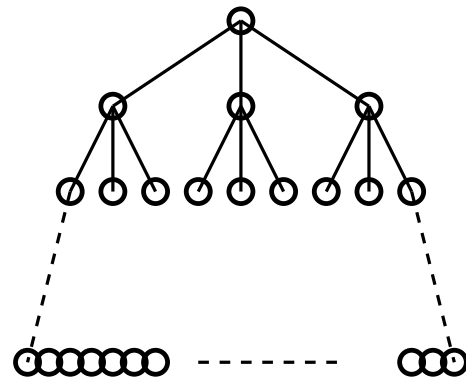
profondeur k tq $\frac{n}{b^k} \approx 1$, ie $k \approx \log_b n$.

Analyse des récurrences diviser pour régner

Théorème. Soit $a > 0$, $b > 1$, et $d \geq 0$, et supposons que $T(n) = aT(\lceil n/b \rceil) + f(n)$ avec $f(n) = O(n^d \log^c n)$ alors

$$T(n) = \begin{cases} O(n^d \log^c n) & \text{si } d > \log_b a \\ O(n^d \log^{c+1} n) & \text{si } d = \log_b a \\ O(n^{\log_b a} \log^c n) & \text{si } d < \log_b a. \end{cases}$$

$$\begin{aligned} & 1 \times n^d \log^c n \\ & a \times \left(\frac{n}{b}\right)^d \log^c n \\ & a^2 \times \left(\frac{n}{b^2}\right)^d \log^c n \\ & \vdots \\ & a^k \times \left(\frac{n}{b^k}\right)^d \log^c n \end{aligned}$$



profondeur k tq $\frac{n}{b^k} \approx 1$, ie $k \approx \log_b n$.

si $d > \log_b a$, la suite $\left(\frac{a}{b^d}\right)^k \searrow$
la série converge: coût $cte \cdot f(n)$

si $d = \log_b a$, alors $\left(\frac{a}{b^d}\right)^k = 1$

coût/niveau constant: $\log(n) \cdot f(n)$

si $d < \log_b a$, la suite $\left(\frac{a}{b^d}\right)^k \nearrow$
coût dominant en bas: $a^{\log_b n} f(cte)$

MIN DISTANCE: diviser pour régner

Théorème. Soit $a > 0$, $b > 1$, et $d \geq 0$, et supposons que $T(n) = aT(\lceil n/b \rceil) + f(n)$ avec $f(n) = O(n^d \log^c n)$ alors

$$T(n) = \begin{cases} O(n^d \log^c n) & \text{si } d > \log_b a \\ O(n^d \log^{c+1} n) & \text{si } d = \log_b a \\ O(n^{\log_b a} \log^c n) & \text{si } d < \log_b a. \end{cases}$$

La récurrence pour MinDistance:

$$C(n) = 2C(n/2) + O(n \log n)$$

d'où une complexité en $O(n \log^2 n)$

Remarque: une fois les tris horizontaux et verticaux faits au début, les opérations de division et recollement peuvent s'effectuer en $O(n)$

$$C(n) = 2C(n/2) + O(n) \quad \Rightarrow \text{coût final en } O(n \log n).$$

Cours 2: Récursivité et programmation dynamique

- Récursivité, mémoïsation, programmation dynamique
- Distance minimale, diviser pour régner
- **Sous-suites croissantes, mémoïsation**
- Chemin dans un DAG, programmation dynamique
- Sac à dos, produits de matrices, alignements

Plus longue sous-suite croissante

Donnée: un ensemble de points

Problème: trouver un chemin croissant utilisant un nb max de points

Trier horizontalement puis verticalement:

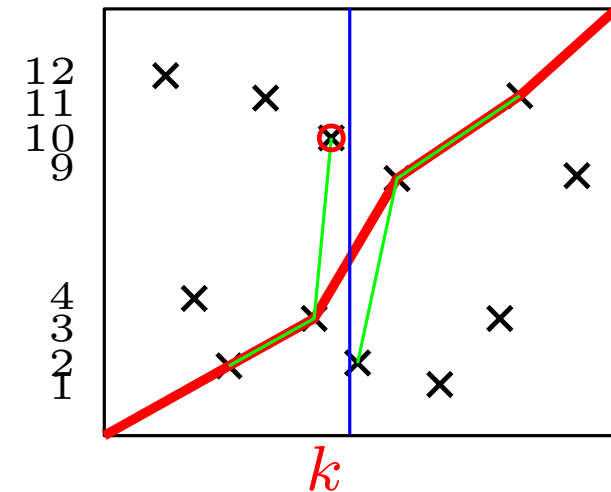
⇒ plus longue sous-suite croissante dans la liste des ordonnées

0, 12, 4, 2, 11, 3, 10, 2, 9, 1, 3, 11, 9, 13

Diviser pour régner ? Pas de recollement des solutions optimales !

Il faut raffiner les sous-problèmes en fonction du point final:

$f(k)$ = longueur max d'une ss-suite croissante terminant par (x_k, y_k) .



Plus longue sous-suite croissante: mémoïsation

Donnée: un ensemble de points

Problème: trouver un chemin croissant utilisant un nb max de points

Trier horizontalement puis verticalement:

⇒ plus longue sous-suite croissante dans la liste des ordonnées

0, 12, 4, 2, 11, 3, 10, 2, 9, 1, 3, 11, 9, 13

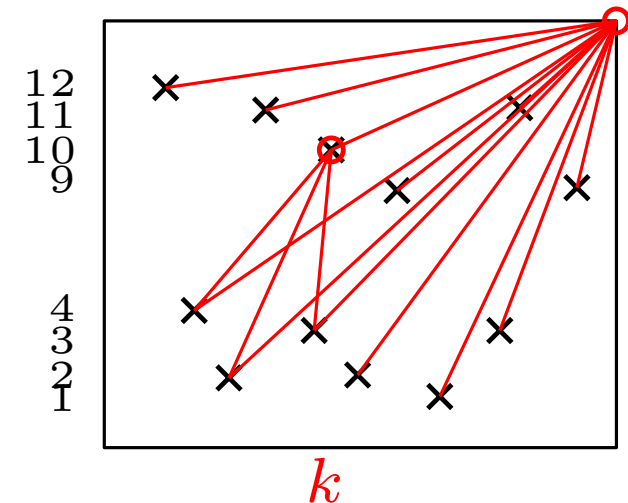
Il faut raffiner les sous-problèmes en fonction du point final:

$f(k)$ = longueur max d'une ss-suite croissante terminant par (x_k, y_k) .

$$\Rightarrow f(k) = 1 + \max\{f(i), i < k, y_i \leq y_k\}$$

Exploiter directement cette récurrence en partant de $f(n)$?

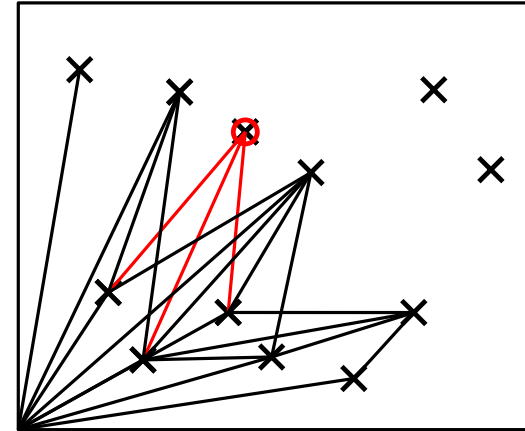
Ok mais il faut mémoïser sinon Fibonacci strikes again.



Plus longue sous-suite croissante: prog. dyn ?

Donnée: un ensemble de points

Problème: trouver un chemin croissant utilisant un nb max de points



$f(k)$ = longueur max d'une ss-suite croissante terminant par (x_k, y_k) .
 $\Rightarrow f(k) = 1 + \max\{f(i), i < k, y_i \leq y_k\}$

Programmation dynamique: identifier les sous-problèmes utiles et organiser les calculs itérativement...

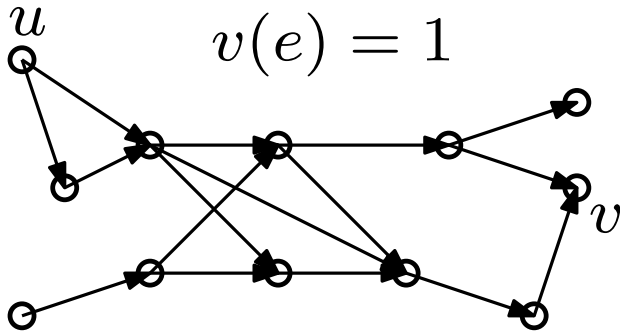
Ici on voit qu'on peut faire les calculs par ordre croissants de k :
en effet $f(k)$ ne dépend que de $f(i)$ avec $i < k$.

Graphe des dépendances:

\Rightarrow recherche d'un plus long chemin dans un graphe orienté sans cycle.

Une généralisation: chemins dans un DAG

DAG = Directed Acyclic Graph = graphe orienté sans cycle



$$v(e) = 1$$

Donnée: un DAG, une valuation des arêtes

Problème. Déterminer un chemin de poids maximal entre 2 sommets u et v :

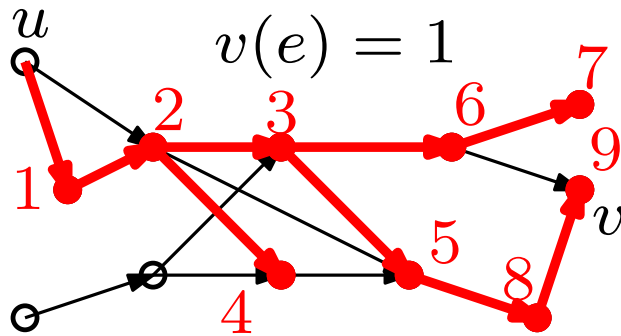
$$p(u, v) = \max_{u, e_1, w_1, \dots, w_{k-1}, e_k, v} p(e_1) + \dots + p(e_k)$$

Exemple: plus longs ou plus courts chemins dans un DAG

- plus longue sous-séquence maximale

Une généralisation: chemins dans un DAG

DAG = Directed Acyclic Graph = graphe orienté sans cycle



Donnée: un DAG, une valuation des arêtes

Problème. Déterminer un chemin de poids maximal entre 2 sommets u et v :

$$p(u, v) = \max_{u, e_1, w_1, \dots, w_{k-1}, e_k, v} p(e_1) + \dots + p(e_k)$$

Récurrence: un chemin arrivant en x finit par une arête (y, x)

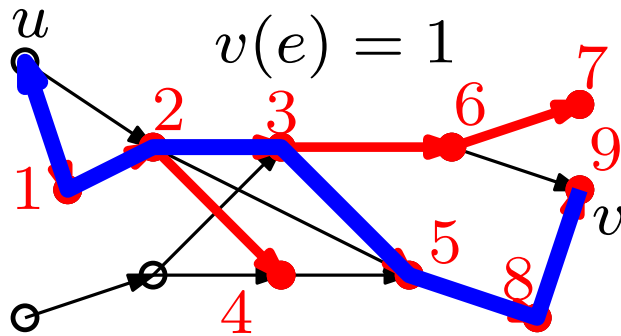
$$\text{donc si } u \neq x, \text{ alors } p(u, x) = \max_{(y,x) \in E} (p(u, y) + p(y, x))$$

Algorithme: • Ordonner u, x_1, x_2, \dots les sommets accessibles de sorte que les arêtes soient croissantes (tri topologique, INF431).

- Calculer les $p(u, x_i)$ dans l'ordre des $i \nearrow$, et pour chaque i marquer l'arête qui donne le max.
- Remonter les arêtes marquées depuis v .

Une généralisation: chemins dans un DAG

DAG = Directed Acyclic Graph = graphe orienté sans cycle



Donnée: un DAG, une valuation des arêtes

Problème. Déterminer un chemin de poids maximal entre 2 sommets u et v :

$$p(u, v) = \max_{u, e_1, w_1, \dots, w_{k-1}, e_k, v} p(e_1) + \dots + p(e_k)$$

Récurrence: un chemin arrivant en x finit par une arête (y, x)

$$\text{donc si } u \neq x, \text{ alors } p(u, x) = \max_{(y, x) \in E} (p(u, y) + p(y, x))$$

Algorithme: • Ordonner u, x_1, x_2, \dots les sommets accessibles de sorte que les arêtes soient croissantes (tri topologique, INF431).

- Calculer les $p(u, x_i)$ dans l'ordre des $i \nearrow$, et pour chaque i marquer l'arête qui donne le max.
- Remonter les arêtes marquées depuis v .

Cours 2: Récursivité et programmation dynamique

- Distance minimale, diviser pour régner
- Analyse des récurrences diviser pour régner
- Sous-suite croissante, chemins dans un DAG
- Principe de la programmation dynamique
- Sac à dos, produits de matrices, alignements

Mémoïsation ou programmation dynamique

Les chemins dans un DAG permettent une modélisation générique des problèmes avec une structure récursive:

- les sommets représentent les sous-problèmes
- les arêtes les dépendances entre problèmes et sous-problèmes: un arc de x à y s'il faut connaître la solution de x pour trouver celle de y .

Si le DAG est un arbre: pas de réutilisation des solutions, l'efficacité de la récursion dépend uniquement du nb de sous-problèmes à traiter.

Sinon il faut mémoïser ou utiliser la **programmation dynamique**:

- résoudre toutes les instances du problème par ordre croissant de taille
- stocker les instances pouvant encore servir à une étape ultérieure.

⇒ prog dyn quand on peut facilement itérer sur les sous-problèmes:

- ss-pbs indexés par des uplets d'entiers
⇒ se ramène au remplissage d'un tableau par boucles imbriquées

Cours 2: Récursivité et programmation dynamique

- Distance minimale, diviser pour régner
- Analyse des récurrences diviser pour régner
- Sous-suite croissante, chemins dans un DAG
- Principe de la programmation dynamique
- Sac à dos, produits de matrices, alignements

Le problème du sac à dos $S(P)$

Donnée: les poids p_1, \dots, p_n de n objets et leurs valeurs v_1, \dots, v_n

Problème: trouver le sous-ensemble de poids $\leq P$ de valeur maximale

Hypothèse: on suppose les poids entiers et P pas trop grand

On utilise la programmation dynamique:

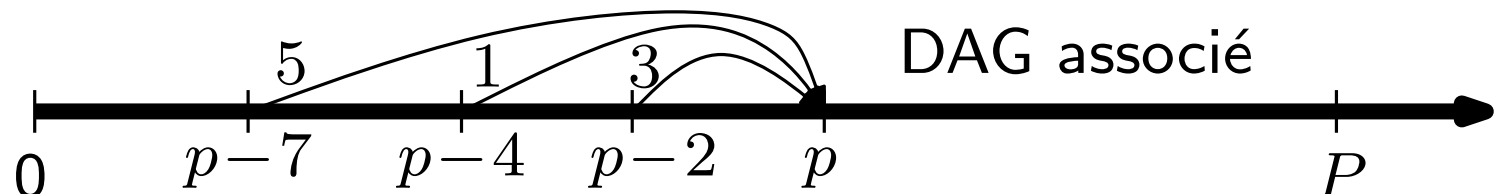
- on peut considérer les sous-problèmes $S(p)$ pour $p < P$

$$S(p) = \max_{i, p_i \leq p} (v_i + S(p - p_i))$$

ok si on peut prendre le même objet plusieurs fois

Exemple: objets

p_i	2	4	7
v_i	3	1	5



Le problème du sac à dos $S(P)$

Donnée: les poids p_1, \dots, p_n de n objets et leur valeur v_1, \dots, v_n

Chaque objet ne peut être pris qu'une fois

- On considère $S(k, p)$, le meilleur sac de poids $\leq p$ utilisant uniquement des objets parmi $1, \dots, k$

$$S(k, p) = \max \begin{cases} S(k-1, p) & \text{dans tous les cas} \\ v_k + S(k-1, p - p_k) & \text{si } p_k \leq p \end{cases}$$

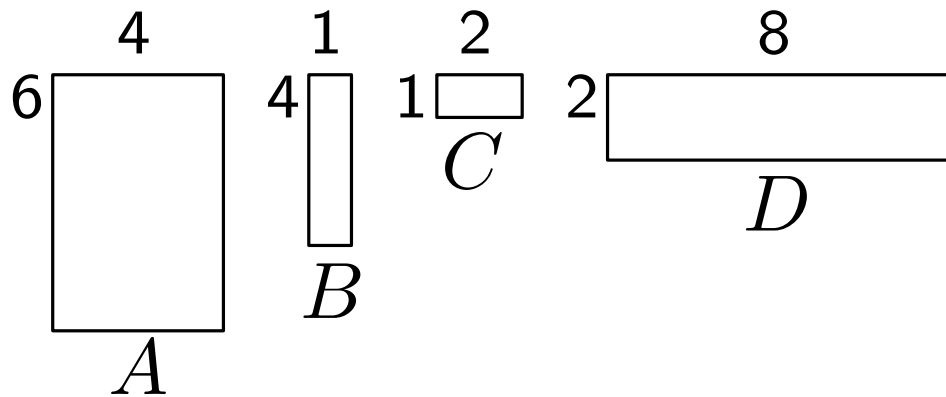
Exemple: objets

p_i	2	4	7
v_i	3	1	5

p	0	1	2	3	4	5	6	7	
$S(0, p)$	0	0	0	0	0	0	0	0	
$S(1, p)$	0	0	3	3	3	3	3	3	+3
$S(2, p)$	0	0	3	3	3	3	4	4	+1
$S(3, p)$	0	0	3	3	3	3	4	5	+5

Produit de matrices en chaîne

On calcule des produits de matrices dont les tailles sont fixées.



$$A \cdot (B \cdot (C \cdot D)) = 240$$

$$6 \cdot 4 \cdot 8 \quad 4 \cdot 1 \cdot 8 \quad 1 \cdot 2 \cdot 8$$

$$(A \cdot (B \cdot C)) \cdot D = 152$$

$$6 \cdot 4 \cdot 2 \quad 4 \cdot 1 \cdot 2 \quad 6 \cdot 2 \cdot 8$$

$$(A \cdot B) \cdot (C \cdot D) = 88$$

$$6 \cdot 4 \cdot 1 \quad 6 \cdot 1 \cdot 8 \quad 1 \cdot 2 \cdot 8$$

L'ordre des produits joue sur la complexité: $M((k, \ell), (\ell, m)) \approx k\ell m$,

On veut un algorithme qui trouve le meilleur parenthésage.

Algorithme glouton: faire d'abord le produit le moins coûteux.

sur l'exemple on obtient $(A \cdot (B \cdot C)) \cdot D$, non optimal...

Produit de matrices en chaîne

Donnée: les tailles $m_i \times m_{i+1}$ des matrices M_i pour $i = 1 \dots, n$

Problème: trouver le meilleur parenthésage du produit $M_1 \cdots M_n$.

Remarque: Dans un parenthésage optimal, toute paire de parenthèses entoure un parenthésage optimal du facteur qu'elle délimite.

On considère les sous-problèmes "parenthésage optimal des facteurs"

$$c(i, j) = \text{coût min du calcul } M_i \cdots M_j.$$

Récurrence: $c(i, j)$ est donné par le meilleur découpage de la forme $M_i \cdots M_j = (M_i \cdots M_k)(M_{k+1} \cdots M_j)$ avec les 2 facteurs optimaux

$$c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

avec $c(i, i) = 0$, et $c(i, i + 1) = m_i m_{i+1} m_{i+2}$.

Produit de matrices en chaîne: prog dyn

Donnée: les tailles $m_i \times m_{i+1}$ des matrices M_i pour $i = 1 \dots, n$

Problème: trouver le meilleur parenthésage du produit $M_1 \cdots M_n$.

Remarque: Dans un parenthésage optimal, toute paire de parenthèses entoure un parenthésage optimal du facteur qu'elle délimite.

On considère les sous-problèmes "parenthésage optimal des facteurs"

$$c(i, j) = \text{coût min du calcul } M_i \cdots M_j.$$

Récurrence: $c(i, j)$ est donné par le meilleur découpage de la forme $M_i \cdots M_j = (M_i \cdots M_k)(M_{k+1} \cdots M_j)$ avec les 2 facteurs optimaux

$$c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

avec $c(i, i) = 0$, et $c(i, i + 1) = m_i m_{i+1} m_{i+2}$.

Produit de matrices en chaîne: prog dyn

Donnée:

$M_1 M_2 M_3 M_4$

i	1	2	3	4	5
m_i	6	4	1	2	8

$i \backslash j$	1	2	3	4
1	0	24		
2		0	8	
3			0	16
4				0

$$c(1, 3) = \min_k ($$

1: $0 + 8 + 6 \cdot 4 \cdot 2 = 56$

2: $24 + 0 + 6 \cdot 1 \cdot 2 = 36$

$$)$$

Algorithme: • Remplir la table des coûts $c(i, j)$ en commençant par les produits de 2 matrices, puis 3, etc, en appliquant la formule

$$c(i, i) = 0 \text{ et } c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

Produit de matrices en chaîne: prog dyn

Donnée:

$M_1 M_2 M_3 M_4$

i	1	2	3	4	5
m_i	6	4	1	2	8

$i \setminus j$	1	2	3	4
1	0	24	36	
2		0	8	
3			0	16
4				0

$$c(2, 4) = \min_k ($$

2: $0 + 16 + 4 \cdot 1 \cdot 8 = 48$

3: $8 + 0 + 4 \cdot 2 \cdot 8 = 72$

$$)$$

Algorithme: • Remplir la table des coûts $c(i, j)$ en commençant par les produits de 2 matrices, puis 3, etc, en appliquant la formule

$$c(i, i) = 0 \text{ et } c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

Produit de matrices en chaîne: prog dyn

Donnée:

$M_1 M_2 M_3 M_4$

i	1	2	3	4	5
m_i	6	4	1	2	8

$i \setminus j$	1	2	3	4
1	0	24	36	
2		0	8	48
3			0	16
4				0

$$c(1, 4) = \min_k ($$

$$1: 0 + 48 + 6 \cdot 4 \cdot 8 = 240$$

$$2: 24 + 16 + 6 \cdot 1 \cdot 8 = 88$$

$$3: 36 + 0 + 6 \cdot 2 \cdot 8 = 132$$

)

Algorithme: • Remplir la table des coûts $c(i, j)$ en commençant par les produits de 2 matrices, puis 3, etc, en appliquant la formule

$$c(i, i) = 0 \text{ et } c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

Produit de matrices en chaîne: prog dyn

Donnée:

$M_1 M_2 M_3 M_4$

i	1	2	3	4	5
m_i	6	4	1	2	8

$i \setminus j$	1	2	3	4
1	0	24	36	88
2		0	8	48
3			0	16
4				0

$c(1, 4) = \min_k ($

1: $0 + 48 + 6 \cdot 4 \cdot 8 = 240$

2: $24 + 16 + 6 \cdot 1 \cdot 8 = 88$

3: $36 + 0 + 6 \cdot 2 \cdot 8 = 132$

)

Algorithme: • Remplir la table des coûts $c(i, j)$ en commençant par les produits de 2 matrices, puis 3, etc, en appliquant la formule

$$c(i, i) = 0 \text{ et } c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

Produit de matrices en chaîne: prog dyn

Donnée:

$M_1 M_2 M_3 M_4$

i	1	2	3	4	5
m_i	6	4	1	2	8

$i \setminus j$	1	2	3	4
1	0	24	36	88
2		0	8	48
3			0	16
4				0

$c(1, 4) = \min_k ($

1: $0 + 48 + 6 \cdot 4 \cdot 8 = 240$

2: $24 + 16 + 6 \cdot 1 \cdot 8 = 88$

3: $36 + 0 + 6 \cdot 2 \cdot 8 = 132$

)

Algorithme: • Remplir la table des coûts $c(i, j)$ en commençant par les produits de 2 matrices, puis 3, etc, en appliquant la formule

$$c(i, i) = 0 \text{ et } c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

Complexité: espace n^2 , temps n^3 .

On trouve le parenthésage optimal à l'aide d'une table des meilleurs k .

$i \setminus j$	1	2	3	4
1	*	1	2	2
2		*	2	2
3			*	3
4				*

Produit de matrices en chaîne: prog dyn

Donnée:

	M_1	M_2	M_3	M_4	
i	1	2	3	4	5
m_i	6	4	1	2	8

$i \setminus j$	1	2	3	4
1	0	24	36	88
2		0	8	48
3			0	16
4				0

$$c(1, 4) = \min_k ($$

$$1: 0 + 48 + 6 \cdot 4 \cdot 8 = 240$$

$$2: 24 + 16 + 6 \cdot 1 \cdot 8 = 88$$

$$3: 36 + 0 + 6 \cdot 2 \cdot 8 = 132$$

)

Algorithme: • Remplir la table des coûts $c(i, j)$ en commençant par les produits de 2 matrices, puis 3, etc, en appliquant la formule

$$c(i, i) = 0 \text{ et } c(i, j) = \min_{i \leq k < j} \left(c(i, k) + c(k + 1, j) + m_i m_{k+1} m_{j+1} \right).$$

Complexité: espace n^2 , temps n^3 .

On trouve le parenthésage optimal à l'aide d'une table des meilleurs k .

$i \setminus j$	1	2	3	4
1	*	1	2	2
2		*	2	2
3			*	3
4				*

$M_1 M_2 M_3 M_4$

$(M_1 M_2)(M_3 M_4)$

Complexité:
temps linéaire

Distance d'édition et alignement de séquences

Donnée: Deux mots u, v sur l'alphabet $\mathcal{A} = \{A, C, G, T\}$ et une fonction de similarité des lettres $\sigma : (\mathcal{A} \cup \{-\})^2 \rightarrow \mathbb{R}$.

Problème: trouver l'**alignement** (U, V) qui maximise $\sum_i \sigma(U_i, V_i)$.

$u = GATGCAT$

$v = ATCGAT$

U	G	A	T	$-$	G	C	A	T
V	$-$	A	T	C	G	$-$	A	T

$\sigma(x, x) = 2$

$\sigma(x, -) = -1$

$\sigma(x, y) = -\infty$

$\sigma(U, V) = -1 + 2 + 2 - 1 + 2 - 1 + 2 + 2$

Hypothèses: • $\sigma(x, y)$ d'autant plus grand que x et y se ressemblent.

• pour tout x , $\sigma(x, x) > 0 > \sigma(-, x)$.

Récurrence: pour $S_{i,j}$ = meilleur alignement de $u_1 \dots u_i$ avec $v_1 \dots v_j$.

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + \sigma(u_i, v_j) & \text{si } i > 0 \text{ et } j > 0 \\ S_{i-1,j} + \sigma(u_i, -) & \text{si } i > 0 \\ S_{i,j-1} + \sigma(-, v_j) & \text{si } j > 0 \end{cases}$$

Distance d'édition et alignement de séquences

On veut calculer $S_{n,n}$ à partir de $S_{0,0} = 0$ et de la récurrence

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + \sigma(u_i, v_j) & \text{si } i > 0 \text{ et } j > 0 \\ S_{i-1,j} + \sigma(u_i, -) & \text{si } i > 0 \\ S_{i,j-1} + \sigma(-, v_j) & \text{si } j > 0 \end{cases}$$

Pour calculer les $S_{i,j}$ avec $i + j = k$ on a besoin que des $S_{i,j}$ avec $i + j = k - 1$ ou $k - 2 \Rightarrow$ on peut se contenter d'un espace linéaire.

Mais dans ce cas on ne veut pas garder la table des meilleurs choix.

On peut trouver le meilleur alignement en espace linéaire par une approche **diviser pour régner**.

- Soit $i_0 = |u|/2$, on calcule le score $a_j = S_{i_0,j}$ pour tout j ainsi que $b_j = S_{i_0,j}^*$ score du meilleur alignement de la 2ème moitié de u avec $v_{j+1} \dots v_m$ pour tout j : **espace linéaire**.
- On choisit le $k = \operatorname{argmax}_j (a_j + b_j)$
on recommence dans les 2 moitiés séparément.

Cours 2: Récursivité et programmation dynamique

- Distance minimale, diviser pour régner
- Analyse des récurrences diviser pour régner
- Sous-suite croissante, chemins dans un DAG
- Principe de la programmation dynamique
- Sac à dos, produits de matrices, alignements

⇒ 3 idées pour des problèmes avec une structure récursive:
diviser pour régner, mémorisation, programmation dynamique