

Introduction à l'algorithmique : Paradigmes

Serge Haddad

LMF, ENS Paris-Saclay & CNRS

L3

- 1 Diviser pour régner
- 2 Programmation dynamique
- 3 Algorithmes gloutons

Plan

① Diviser pour régner

Programmation dynamique

Algorithmes gloutons

Diviser pour régner

Diviser pour régner est une stratégie de résolution de problèmes qui consiste à :

- ▶ Diviser le problème en sous-problèmes
- ▶ Résoudre les sous-problèmes
- ▶ Combiner les réponses aux sous-problèmes pour fournir la réponse finale

Le partitionnement doit viser à :

- ▶ résoudre le moins de sous-problèmes possibles
- ▶ équilibrer la taille des sous-problèmes
- ▶ s'assurer de l'indépendance des sous-problèmes

La combinaison ne doit pas entraîner un surcoût rédhibitoire.

Tri fusion

- Diviser le tableau en deux parties approximativement égales.
- Trier les sous-tableaux.
- Fusionner les sous-tableaux triés.

```
Tri( $T, n$ )  
If  $n = 1$  then return ;  
 $mid \leftarrow \lfloor \frac{n}{2} \rfloor$  ;  
For  $i$  from 1 to  $mid$  do  $T_1[i] \leftarrow T[i]$  ;  
For  $i$  from  $mid + 1$  to  $n$  do  $T_2[i - mid] \leftarrow T[i]$  ;  
Tri( $T_1, mid$ ) ;  
Tri( $T_2, n - mid$ ) ;  
Fusion( $T_1, mid, T_2, n - mid, T$ ) ;
```

Fusion

- Maintenir un indice par tableau
- Recopier la plus petite valeur et incrémenter l'indice correspondant
- Recopier la partie du dernier tableau restant

```
Fusion( $T_1, n_1, T_2, n_2, T$ )
```

```
 $i \leftarrow 1; i_1 \leftarrow 1; i_2 \leftarrow 1;$ 
```

```
While  $i_1 \leq n_1$  and  $i_2 \leq n_2$  do
```

```
  If  $T_1[i_1] \leq T_2[i_2]$  then  $T[i] \leftarrow T_1[i_1]; i_1 \leftarrow i_1 + 1;$ 
```

```
  Else  $T[i] \leftarrow T_2[i_2]; i_2 \leftarrow i_2 + 1;$ 
```

```
   $i \leftarrow i + 1;$ 
```

```
For  $j$  from  $i_1$  to  $n_1$  do  $T[i] \leftarrow T_1[j]; i \leftarrow i + 1;$ 
```

```
For  $j$  from  $i_2$  to  $n_2$  do  $T[i] \leftarrow T_2[j]; i \leftarrow i + 1;$ 
```

$$t_{\text{Fusion}}(n_1, n_2) = \Theta(n_1 + n_2) \quad t_{\text{Tri}}(n) = t_{\text{Tri}}(\lfloor \frac{n}{2} \rfloor) + t_{\text{Tri}}(\lceil \frac{n}{2} \rceil) + \Theta(n)$$

D'où :

$$t_{\text{Tri}}(n) = \Theta(n \log(n))$$

Analyse du tri fusion

Le tri fusion est un tri stable.

Son temps de calcul est indépendant des données (pas adaptatif).

Le tri fusion n'est pas un tri *en place*.

Les tableaux auxiliaires occupent n cellules.

En cumulant la place occupée par les appels récurifs, on obtient :

$$n + \frac{n}{2} + \frac{n}{4} + \dots \sim 2n$$

On peut se limiter à un unique tableau auxiliaire en alternant les copies d'un tableau à l'autre.

Tri fusion partiel

Toutes les modifications du tableau se font par échange.

```
Swap( $i, j$ )  $temp \leftarrow T[i]; T[i] \leftarrow T[j]; T[j] \leftarrow temp$ 
```

Le tri fusion partiel ordonne un *bloc* du tableau en utilisant un bloc disjoint de même taille comme zone de travail

```
TriP( $a, b, m$ )  $\{[a, a + m] \cap [b, b + m] = \emptyset\}$   
If  $m = 1$  then return ;  
 $mid \leftarrow \lfloor \frac{m}{2} \rfloor$  ;  
For  $i$  from  $0$  to  $m - 1$  do Swap( $a + i, b + i$ ) ;  
TriP( $b, a, mid$ ) ;  
TriP( $b + mid, a + mid, m - mid$ ) ;  
FusionP( $b, mid, b + mid, m - mid, a$ ) ;
```

Fusion partielle

Identique à la fusion mais procède par échange.

```
{[a, a + n_1[∩[b, b + n_2[ = [a, a + n_1[∩[c, c + n_1 + n_2[ = [c, c + n_1 + n_2[∩[b, b + n_2[ = ∅}
FusionP(a, n_1, b, n_2, c)
i ← c; i_1 ← a; i_2 ← b;
While i_1 < a + n_1 and i_2 < b + n_2 do
  If T[i_1] ≤ T[i_2] then Swap(i, i_1); i_1 ← i_1 + 1;
  Else Swap(i, i_2); i_2 ← i_2 + 1;
  i ← i + 1;
For j from i_1 to a + n_1 - 1 do Swap(i, j); i ← i + 1;
For j from i_2 to b + n_2 - 1 do Swap(i, j); i ← i + 1;
```

Comme précédemment :

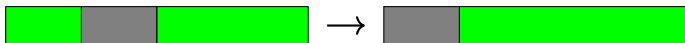
$$t_{\text{TriP}}(m) = \Theta(m \log(m))$$

Tri fusion sur place (1)

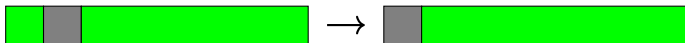
- Le tri fusion sur place trie la moitié droite du tableau où la zone de travail est la moitié gauche.



- Puis il trie le quart gauche du tableau et le fusionne avec la moitié droite où la zone de travail est le deuxième quart.



- Puis il trie le huitième gauche du tableau et le fusionne avec les trois quarts droits où la zone de travail est le deuxième huitième ...



- Lorsqu'il ne reste que le premier élément à trier, il est inséré en bonne position.

Tri fusion sur place (2)

Tri()

$mid \leftarrow \lfloor \frac{n}{2} \rfloor$;

TriP($n - mid + 1, n - 2 * mid + 1, mid$);

$m \leftarrow n - mid$;

While $m > 1$ **do**

$mid \leftarrow \lfloor \frac{m}{2} \rfloor$;

TriP($1, mid, mid$);

Fusion($mid, n - m$);

$m \leftarrow m - mid$;

$i \leftarrow 2$;

While $i \leq n$ **and** $T[i] < T[i - 1]$ **do** Swap($i, i - 1$); $i \leftarrow i + 1$;

Fusion (presque) sur place

$$\{2 * n_1 + n_2 \leq n\}$$

Fusion(n_1, n_2)

$i \leftarrow n - n_1 - n_2 + 1$; $i_1 \leftarrow 1$; $i_2 \leftarrow n - n_2 + 1$;

While $i_1 \leq n_1$ **and** $i_2 \leq n$ **do**

If $T[i_1] < T[i_2]$ **then** Swap(i, i_1); $i_1 \leftarrow i_1 + 1$;

Else Swap(i, i_2); $i_2 \leftarrow i_2 + 1$;

$i \leftarrow i + 1$;

For j **from** i_1 **to** n_1 **do** Swap(i, j); $i \leftarrow i + 1$;

Invariant de boucle.

- Les blocs bleus, verts et jaunes sont triés.
- Les items « jaunes » sont plus petits que les « bleus » et les « verts ».
- Les tailles des blocs bleus et marrons sont égales.

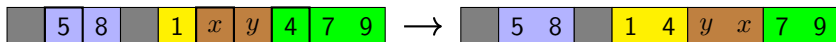
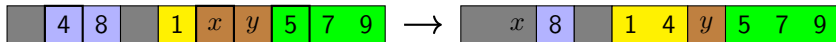
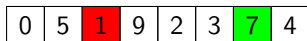


Illustration (1)

TriP(5, 1, 4)



TriP(7, 3, 1)



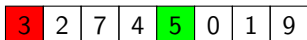
TriP(1, 5, 2)



TriP(8, 4, 1)



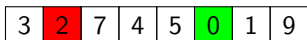
TriP(5, 1, 1)



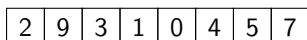
FusionP(1, 2, 3, 2, 5)



TriP(6, 2, 1)



Après TriP(5, 1, 4)



TriP(3, 7, 2)

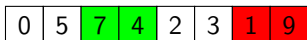
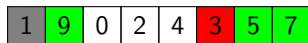
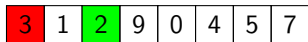


Illustration (2)

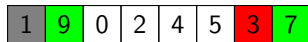
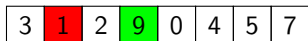
TriP(1, 3, 2)



TriP(3, 1, 1)



TriP(4, 2, 1)



Fusion(2, 4)

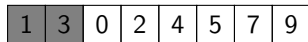
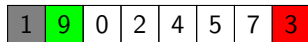
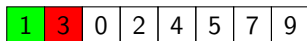
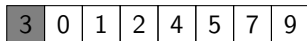
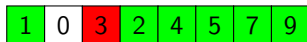
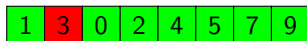


Illustration (3)

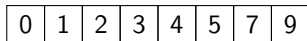
TriP(1, 2, 1)



Fusion(1, 6)



Insertion du premier élément



Complexité du tri fusion sur place

Soit $t_{\text{TriP}}(m) \leq cm \log(m)$.

Supposons sans perte de généralité $n = 2^k$.

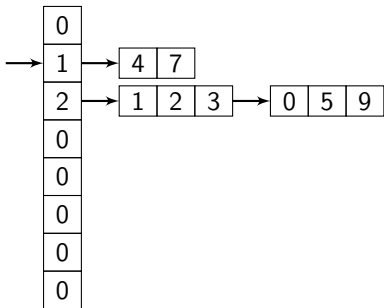
$$\begin{aligned} t_{\text{Tri}}(n) &\leq \sum_{i=1}^k t_{\text{TriP}}\left(\frac{n}{2^i}\right) + \sum_{i=1}^{k-1} dn + en \\ &\leq \sum_{i=1}^k c \frac{n}{2^i} \log(n) + d(k-1)n + en \\ &\leq (c + d + e)kn \\ &= (c + d + e)n \log(n) \end{aligned}$$

d , constante de Fusion

e , constante de l'insertion du dernier élément

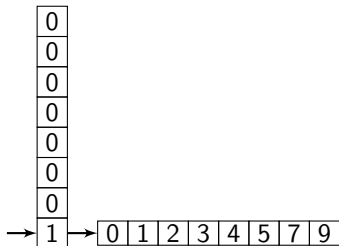
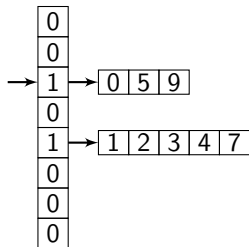
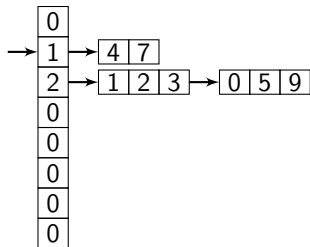
Tri fusion adaptatif (1)

Construction de séquences ordonnées



Tri fusion adaptatif (2)

Fusion de séquences ordonnées



Construction de séquences ordonnées

```
For  $i$  from 1 to  $n$  do  $TL[i] \leftarrow \text{CreerListeListe}(); Tnb[i] \leftarrow 0$   
 $Liste \leftarrow \text{CreerListeInt}(T[1]); \ell \leftarrow 1; new \leftarrow \top; B \leftarrow 1$   
For  $i$  from 2 to  $n$  do  
  If  $new$  then  $new \leftarrow \perp; up \leftarrow T[i-1] \leq T[i]$   
  If  $up$  and  $T[i-1] \leq T[i]$  then  $Liste \leftarrow Liste \cdot T[i]; \ell \leftarrow \ell + 1$   
  Else If not  $up$  and  $T[i-1] \geq T[i]$  then  $Liste \leftarrow T[i] \cdot Liste; \ell \leftarrow \ell + 1$   
  Else  
     $TL[\ell] \leftarrow TL[\ell] \cdot Liste; Tnb[\ell] \leftarrow Tnb[\ell] + 1$   
     $Liste \leftarrow \text{CreerListeInt}(T[i]); \ell \leftarrow 1; new \leftarrow \top; B \leftarrow B + 1$   
   $TL[\ell] \leftarrow TL[\ell] \cdot Liste; Tnb[\ell] \leftarrow Tnb[\ell] + 1$ 
```

Observation. La construction opère en temps linéaire.

Fusion de séquences ordonnées

$ind \leftarrow 1$

For i **from** B **downto** 2 **do**

While $Tnb[ind] = 0$ **do** $ind \leftarrow ind + 1$

$L1 \leftarrow \text{Extraire}(TL[ind]); Tnb[ind] \leftarrow Tnb[ind] - 1; \ell \leftarrow ind$

While $Tnb[ind] = 0$ **do** $ind \leftarrow ind + 1$

$L2 \leftarrow \text{Extraire}(TL[ind]); Tnb[ind] \leftarrow Tnb[ind] - 1; \ell \leftarrow \ell + ind$

$L \leftarrow \text{Fusion}(L1, L2); TL[\ell] \leftarrow L \cdot TL[\ell]; Tnb[\ell] \leftarrow Tnb[\ell] + 1$

$L \leftarrow \text{Extraire}(TL[n])$

For i **from** 1 **to** n **do** $T[i] \leftarrow \text{Extraire}(L)$

Analyse de complexité (1)

Soit $n_1 \leq n_2 \dots \leq n_i$, la taille des i listes rangées dans TL au début d'une itération de fusion.

Soit $\Phi(n_1, n_2, \dots, n_i) = \sum_{j \leq i} n_j \log_{\frac{3}{2}}\left(\frac{n}{n_j}\right)$.

$$\begin{aligned} (n_1 + n_2) \log_{\frac{3}{2}}\left(\frac{n}{n_1 + n_2}\right) &= n_1 \log_{\frac{3}{2}}\left(\frac{n}{n_1 + n_2}\right) + n_2 \log_{\frac{3}{2}}\left(\frac{n}{n_1 + n_2}\right) \\ &\leq n_1 \log_{\frac{3}{2}}\left(\frac{n}{n_1}\right) + n_2 \log_{\frac{3}{2}}\left(\frac{n}{n_2}\right) \end{aligned}$$

D'où :

$$\boxed{\Phi(n_1 + n_2, \dots, n_i) \leq \Phi(n_1, n_2, \dots, n_i)}$$

Analyse de complexité (2)

Si $L2$ est obtenue par fusion de $L1'$ et $L2'$,

- ▶ Soit $|L1| = |L2|$;
- ▶ Soit $|L1| < |L2|$, $L1$ était présent au moment de la fusion ce qui implique $|L1| \geq \max(|L1'|, |L2'|)$.

D'où :

$$\boxed{2|L1| \geq |L2|}$$

$$\begin{aligned}(n_1 + n_2) \log_{\frac{3}{2}}\left(\frac{n}{n_1 + n_2}\right) &= n_1 \log_{\frac{3}{2}}\left(\frac{n}{n_1 + n_2}\right) + n_2 \log_{\frac{3}{2}}\left(\frac{n}{n_1 + n_2}\right) \\ &\leq n_1 \log_{\frac{3}{2}}\left(\frac{2n}{3n_2}\right) + n_2 \log_{\frac{3}{2}}\left(\frac{2n}{3n_2}\right) \\ &\leq n_1 \left(\log_{\frac{3}{2}}\left(\frac{n}{n_1}\right) - 1\right) + n_2 \left(\log_{\frac{3}{2}}\left(\frac{n}{n_2}\right) - 1\right)\end{aligned}$$

D'où :

$$\boxed{\Phi(n_1 + n_2, \dots, n_i) \leq \Phi(n_1, n_2, \dots, n_i) - (n_1 + n_2)}$$

Observation. Φ décroît du nombre de comparaisons effectuées.

Analyse de complexité (3)

- Avant les fusions : $\Phi(n_1, \dots, n_B) = \sum_{j \leq B} n_j \log_{\frac{3}{2}} \left(\frac{n}{n_j} \right)$.

$$\begin{aligned} \sum_{j \leq B} n_j \log_{\frac{3}{2}} \left(\frac{n}{n_j} \right) &= n \left(\sum_{j \leq B} \frac{n_j}{n} \log_{\frac{3}{2}} \left(\frac{n}{n_j} \right) \right) \\ &\leq n \log_{\frac{3}{2}} \left(\sum_{j \leq B} \frac{n_j}{n} \frac{n}{n_j} \right) \\ &= n \log_{\frac{3}{2}} (B) \end{aligned}$$

- Après les fusions $\Phi(n) = 0$.

D'où un nombre de comparaisons lors des fusions avec une liste fusionnée inférieur ou égal à $n \log_{\frac{3}{2}} (B)$.

Il y a au plus n comparaisons lors des fusions entre listes initiales.

D'où un nombre total de comparaisons en $O(n(\log(B) + 1))!$

Tri partiel par dénombrement

Σ est un alphabet ordonné. Σ^k est ordonné lexicographiquement.

T est un tableau de clés appartenant à Σ^k .

On veut trier partiellement T selon la valeur de la position $m \leq k$

```
TriP( $T, m$ )  
For  $a \in \Sigma$  do  $nb[a] \leftarrow 0$ ;  
For  $i$  from 1 to  $n$  do  $nb[T[i][m]] \leftarrow nb[T[i][m]] + 1$ ;  $T'[T[i][m]][nb[T[i][m]]] \leftarrow T[i]$ ;  
 $i \leftarrow i + 1$ ;  
For  $a \in \Sigma$  do  
  For  $j$  from 1 to  $nb[a]$  do  $T[i] \leftarrow T'[a][j]$ ;  $i \leftarrow i + 1$ ;
```

Observations.

La complexité est $\Theta(n + |\Sigma|)$.

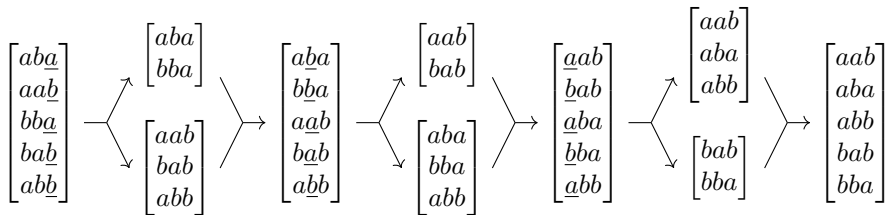
Ce n'est pas un tri sur place.

C'est un tri stable.

Tri par base

On itère le tri par dénombrement de la dernière à la première position :

For m **from** k **downto** 1 **do** TriP(T, m)



Observations.

$$t(n) = \Theta(k(n + |\Sigma|))$$

Si $|\Sigma| = \Theta(n)$ et $k = \Theta(1)$,

on a un algorithme en $\Theta(n)$ pour un espace de clés en $n^{\Theta(1)}$.

Tri des NNI

Le numéro d'identification national est constitué de 13 chiffres.

Il y a approximativement 2^{26} français.

Tri par comparaison

$$n \log(n) \approx 26 \cdot 2^{26}$$

Tri par base

$$k(n + |\Sigma|) \approx 13 \cdot 2^{26}$$

Pas de différence significative.

Plus proche paire de points

Problème. Déterminer une paire de points du plan la plus proche parmi un ensemble de n points.

Observation. L'algorithme naïf a une complexité $\Theta(n^2)$.

Etape préliminaire. (en $\Theta(n \log(n))$)

Créer le tableau X d'indices de T trié selon le couple (abscisse, ordonnée).

Créer le tableau Y d'indices de T trié selon les ordonnées.

Illustration.

$$T = \begin{bmatrix} (3.5, 0.7) \\ (2.5, 0.1) \\ (2.5, 0.9) \\ (4, 0.2) \\ (2, 1.5) \end{bmatrix} \quad X = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 1 \\ 4 \end{bmatrix} \quad Y = \begin{bmatrix} 2 \\ 4 \\ 1 \\ 3 \\ 5 \end{bmatrix}$$

Diviser pour régner (1)

Partitionner selon les coordonnées les points en deux sous-ensembles de tailles approximativement égales.

Part(X, Y, n)

Partitionnement de X trivial

$mid \leftarrow \lfloor \frac{n}{2} \rfloor$; $xmid \leftarrow T[X[mid]].x$; $ymid \leftarrow T[X[mid]].y$;

For i **from** 1 **to** mid **do** $X_g[i] \leftarrow X[i]$;

For i **from** $mid + 1$ **to** n **do** $X_d[i - mid] \leftarrow X[i]$;

Partitionnement de Y à l'aide des coordonnées

$i_g \leftarrow 1$; $i_d \leftarrow 1$

For i **from** 1 **to** n **do**

If $T[Y[i]].x < xmid$ **or** ($T[Y[i]].x = xmid$ **and** $T[Y[i]].y \leq ymid$) **then**

$Y_g[i_g] \leftarrow Y[i]$; $i_g \leftarrow i_g + 1$;

Else

$Y_d[i_d] \leftarrow Y[i]$; $i_d \leftarrow i_d + 1$;

return(X_g, Y_g, X_d, Y_d)

Diviser pour régner (2)

Résoudre les sous-problèmes.

Short(X, Y, n)

Cas de base $n \leq 2$

If $n = 1$ then return ∞

If $n = 2$ then return $\sqrt{(T[X[2]].x - T[X[1]].x)^2 + (T[X[2]].y - T[X[1]].y)^2}$;

d minimum des deux plus courtes distances

$(X_g, Y_g, X_d, Y_d) \leftarrow \text{Part}(X, Y, n)$;

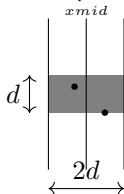
$d \leftarrow \text{Short}(X_d, Y_d, \lceil \frac{n}{2} \rceil)$;

If $n \geq 4$ then $d \leftarrow \min(d, \text{Short}(X_g, Y_g, \lfloor \frac{n}{2} \rfloor))$;

Prendre en compte les distances entre points gauches et droits.

...

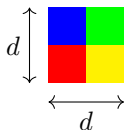
Si la distance entre un point *gauche* et un point *droit* est inférieure à d , alors :



Diviser pour régner (3)

Observation. Soit un ensemble de points de distance minimale d , il y a au plus 4 points dans un carré $d \times d$.

Preuve. Au plus un point par carré de couleur.



Pour diminuer la distance minimale :

- ▶ Sélectionner dans Y les points dans la bande $x_{mid} - d \leq x \leq x_{mid} + d$.
- ▶ Parcourir les points de la sélection Y' (selon les ordonnées).
- ▶ Calculer la distance d'un point à ses 7 successeurs.

Diviser pour régner (4)

Implémentation.

```
 $x_{mid} \leftarrow T[X[\lfloor \frac{n}{2} \rfloor]].x;$ 
```

Sélection

```
 $i' \leftarrow 0;$ 
```

```
For  $i$  from 1 to  $n$  do
```

```
  If  $T[Y[i]].x \geq x_{mid} - d$  and  $T[Y[i]].x \leq x_{mid} + d$  then
```

```
     $i' \leftarrow i' + 1;$      $Y'[i'] \leftarrow Y[i];$ 
```

Calcul des distances

```
For  $i$  from 1 to  $i' - 1$  do
```

```
  For  $j$  from  $i + 1$  to  $\min(i + 7, i')$  do
```

```
     $d' \leftarrow \sqrt{(T[Y'[j]].x - T[Y'[i]].x)^2 + (T[Y'[j]].y - T[Y'[i]].y)^2};$ 
```

```
    If  $d' < d$  then  $d \leftarrow d';$ 
```

```
return( $d$ );
```

$$t(n) = 2t\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow t(n) = \Theta(n \log(n))$$

Plan

Diviser pour régner

② Programmation dynamique

Algorithmes gloutons

Limites du diviser pour régner

Observations. Dans l'approche « diviser pour régner »,

- ▶ si les sous-problèmes ne sont pas indépendants
- ▶ alors cela induit des calculs redondants.

Illustration. Suite de Fibonacci : $u_n = u_{n-1} + u_{n-2}$ avec $u_1 = u_2 = 1$.

```
Fib(n)
If  $n \leq 2$  then return(1);
return(Fib( $n-1$ ) + Fib( $n-2$ ));
```

Le temps de calcul est proportionnel à la valeur retournée : un temps exponentiel !

Comment y remédier ?

Mémoïsation

Eviter les appels inutiles par mémorisation des valeurs retournées.

```
Fibrec(k)
```

```
  Ready[k] ← true;
```

```
  If not Ready[k - 1] then Fibrec(k - 1); (ici le test renvoie toujours true)
```

```
  Fibval[k] ← Fibval[k - 1] + Fibval[k - 2];
```

```
  return;
```

```
Fib(n)
```

```
  For i from 1 to 2 do Ready[i] ← true; Fibval[i] ← 1;
```

```
  For i from 3 to n do Ready[i] ← false;
```

```
  If n > 2 then Fibrec(n);
```

```
  return Fibval[n];
```

Pourquoi pas une approche « bottom up » ?

Programmation Dynamique

Le programmation dynamique consiste à :

- ▶ résoudre les sous-problèmes dans un ordre approprié
- ▶ de telle sorte que les sous-problèmes nécessaires à la résolution d'un sous-problème soient déjà résolus.

```
Fib( $n$ )  
 $Fibval[1] \leftarrow 1; Fibval[2] \leftarrow 1;$   
For  $i$  from 3 to  $n$  do  $Fibval[i] \leftarrow Fibval[i - 1] + Fibval[i - 2];$   
return  $Fibval[n];$ 
```

Une contrainte : le graphe des dépendances doit être acyclique.

Gestion de la mémoire

L'ordre choisi vise à minimiser les sous-problèmes nécessaires dans le futur et donc la mémoire.

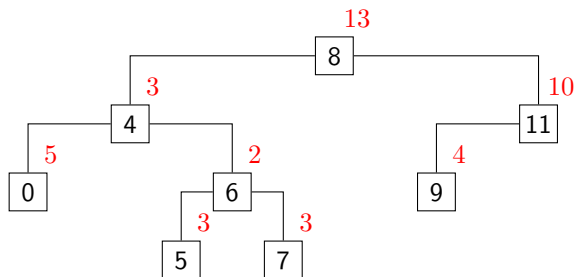
```
Fib(n)  
new ← 1; old ← 1;  
For i from 3 to n do  
    temp ← new + old;  
    old ← new; new ← temp;  
return new;
```

```
Fib(n)  
new ← 1; old ← 1;  
For i from 3 to n do  
    new ← new + old;  
    old ← new - old;  
return new;
```

Arbre de recherche optimal

On associe à chaque clé i une fréquence d'accès f_i .

On cherche un arbre binaire de recherche \mathcal{A} qui minimise $F_{\mathcal{A}} = \sum_i f_i \cdot h_{\mathcal{A}}(i)$
où $h_{\mathcal{A}}(i)$ est la hauteur de i dans \mathcal{A} .



$$F_{\mathcal{A}} = 0 \cdot 13 + 1 \cdot 13 + 2 \cdot 11 + 3 \cdot 6 = 53$$

Sous-problèmes

Supposons les clés données par T , un tableau ordonné de n clés.

Une approche consiste à :

- ▶ déterminer la clé r qui sera la racine ;
- ▶ une fois connue le coût d'un arbre optimal pour $T[1, r - 1]$;
- ▶ et le coût d'un arbre optimal pour $T[r + 1, n]$.

Les sous-problèmes sont donc les $\frac{n(n+1)}{2}$ segments $[i, j]$ de clés contiguës.

Un ordre partiel approprié se fonde sur la longueur du segment.

Observation.

Le coût d'un arbre optimal pour $T[i, j]$ doit être augmenté de $\sum_{k=i}^j f_k$ lorsqu'il est inséré en sous-arbre.

Calcul de l'arbre optimal : les variables

L'instance du problème ($1 \leq i \leq n$)

- $F[i]$, la fréquence d'accès à la clé i (f_i)

Les variables associées au sous-problème des clés de i à j ($1 \leq i \leq j \leq n$)

- $W[i, j]$, la somme des fréquences des clés de i à j ($\sum_{k=i}^j f_k$)
Par convention, $W[i, i-1] = 0$.
- $opt[i, j]$, le coût d'un arbre optimal pour le sous-ensemble des clés de i à j
Par convention, $opt[i, i-1] = 0$.
- $root[i, j]$, la racine d'un arbre optimal pour le sous-ensemble des clés de i à j

Calcul de l'arbre optimal

```
For  $i$  from 1 to  $n + 1$  do Arbre optimal à 0 ou 1 sommet
   $opt[i, i] \leftarrow 0$ ;  $root[i, i] \leftarrow i$ ;  $opt[i, i - 1] \leftarrow 0$ ;  $W[i, i - 1] \leftarrow 0$ ;
For  $i$  from 1 to  $n$  do Calcul fréquences cumulées
  For  $j$  from  $i$  to  $n$  do
     $W[i, j] \leftarrow W[i, j - 1] + F[j]$ ;
For  $\ell$  from 2 to  $n$  do Par longueur croissante
  For  $i$  from 1 to  $n - \ell + 1$  do Par début croissant
     $j \leftarrow i + \ell - 1$ ;  $opt[i, j] \leftarrow \infty$ ;
    For  $k$  from  $i$  to  $j$  do Résolution pour  $[i, j]$ 
       $o \leftarrow W[i, k - 1] + W[k + 1, j] + opt[i, k - 1] + opt[k + 1, j]$ ;
      If  $o < opt[i, j]$  then  $opt[i, j] \leftarrow o$ ;  $root[i, j] \leftarrow k$ ;
```

Observation.

Cet algorithme peut être étendu pour prendre en compte les fréquences des recherches infructueuses entre $T[i]$ et $T[i + 1]$, avant $T[1]$ et après $T[n]$.

Plus longue sous-séquence commune

Soit u un mot de longueur m et v un mot de longueur n .

w un mot de longueur p est une sous-séquence commune de u et v si :

- ▶ Il existe α une fonction strictement croissante de $\{1, \dots, p\}$ dans $\{1, \dots, m\}$ telle que pour tout i , $w[i] = u[\alpha(i)]$;
- ▶ Il existe β une fonction strictement croissante de $\{1, \dots, p\}$ dans $\{1, \dots, n\}$ telle que pour tout i , $w[i] = v[\beta(i)]$

Illustration.

$u = \text{COMPUTER}$, $v = \text{ORDINATEUR}$, $w = \text{OTER}$

- ▶ $\alpha = \langle 2, 6, 7, 8 \rangle$;
- ▶ $\beta = \langle 1, 7, 8, 10 \rangle$.

Objectif. Calcul d'une plus longue sous-séquence commune (PLSC).

Sous-problèmes

En raisonnant sur la dernière lettre appariée, une plus longue sous-séquence commune de u et v est :

- ▶ soit une plus longue sous-séquence commune de $u[1, m - 1]$ et v ;
- ▶ soit une plus longue sous-séquence commune de u et $v[1, n - 1]$;
- ▶ soit une plus longue sous-séquence commune de $u[1, m - 1]$ et $v[1, n - 1]$ étendue par $u[m] = v[n]$.

Les sous-problèmes sont donc les $(m + 1)(n + 1)$ sous-mots $u[1, i]$ et $v[1, j]$.

Un ordre partiel approprié peut être défini par $(i, j) < (i', j')$ si :

- ▶ $i < i'$ et $j \leq j'$;
- ▶ ou $i \leq i'$ et $j < j'$.

Relation entre sous-problèmes

Soit le sous-problème défini par $u[1, i]$ et $v[1, j]$.

Si $u[i] = v[j]$ alors il existe une PLSC qui apparie $u[i]$ et $v[j]$.

Preuve.

Soit w une PLSC.

- ▶ Si w n'apparie ni $u[i]$ ni $v[j]$ alors on peut l'étendre par $u[i] = v[j]$ d'où une contradiction.
- ▶ Supposons que w apparie $u[i]$ et $v[k]$ pour $k < j$ alors w est aussi obtenue en appariant $u[i]$ et $v[j]$.

D'où en notant $\ell[i, j]$ la longueur de la PSLC de $u[1, i]$ et $v[1, j]$:

- ▶ Si $u[i] = v[j]$ alors $\ell[i, j] = \ell[i - 1, j - 1] + 1$;
- ▶ Sinon $\ell[i, j] = \max(\ell[i - 1, j], \ell[i, j - 1])$.

Calcul d'une PLSC

Calcul des longueurs et préparation pour la PLSC

```
For  $i$  from 0 to  $m$  do  $\ell[i, 0] \leftarrow 0$ ; For  $j$  from 1 to  $n$  do  $\ell[0, j] \leftarrow 0$ ;  
For  $i$  from 1 to  $m$  do  
  For  $j$  from 1 to  $n$  do  
    If  $u[i] = v[j]$  then  $\ell[i, j] \leftarrow \ell[i - 1, j - 1] + 1$ ;  
    Else  
       $\ell[i, j] \leftarrow \ell[i - 1, j]$ ;  
      If  $\ell[i, j] < \ell[i, j - 1]$  then  $\ell[i, j] \leftarrow \ell[i, j - 1]$ ;
```

Ecriture de la PLSC

```
 $i \leftarrow m$ ;  $j \leftarrow n$ ;  $k \leftarrow \ell[i, j]$ ;  
While  $k > 0$  do  
  If  $\ell[i, j] = \ell[i - 1, j]$  then  $i \leftarrow i - 1$ ;  
  Else If  $\ell[i, j] = \ell[i, j - 1]$  then  $j \leftarrow j - 1$ ;  
  Else  $w[k] \leftarrow u[i]$ ;  $i \leftarrow i - 1$ ;  $j \leftarrow j - 1$ ;  $k \leftarrow k - 1$ ;
```

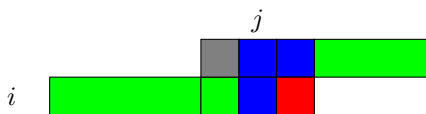
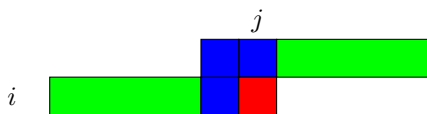
Considérations de mémoire

La mémoire utilisée est en $\Theta(mn)$.

Si on cherche uniquement à connaître la longueur, on peut économiser la mémoire.

Le problème (i, j) ne dépend que des problèmes $(i-1, j-1)$, $(i-1, j)$ et $(i, j-1)$.

Il suffit de conserver $\ell[i, 1], \dots, \ell[i, j-1], \ell[i-1, j-1], \ell[i-1, j], \dots, \ell[i-1, n]$ qu'on notera $\ell[1], \dots, \ell[j-1], old\ell, \ell[j], \dots, \ell[n]$.



Calcul des longueurs

```
Length( $u, m, v, n, \ell$ )  
For  $j$  from 0 to  $n$  do  $\ell[j] \leftarrow 0$ ;  
For  $i$  from 1 to  $m$  do  
   $old\ell \leftarrow 0$ ;  
  For  $j$  from 1 to  $n$  do  
     $temp \leftarrow \ell[j]$ ;  
    If  $u[i] = v[j]$  then  $\ell[j] \leftarrow old\ell + 1$ ;  
    Else If  $\ell[j] < \ell[j - 1]$  then  $\ell[j] \leftarrow \ell[j - 1]$ ;  
     $old\ell \leftarrow temp$ ;
```

Observations

- A la fin de l'algorithme pour tout j , $\ell[j] = |PLSC(u, v[1, j])|$.
- En permutant si nécessaire u et v , la mémoire utilisée est en $\Theta(\min(m, n))$.

Diviser pour régner (encore !)

On note \tilde{u} le mot miroir de u et ε le mot vide.

ℓ et $\tilde{\ell}$ sont des tableaux indicés de 0 à n , initialisés à 0.

PLSC(u, m, v, n)

If $n = 0$ **then return**(ε);

If $m = 1$ **then**

For j **from** 1 **to** n **do if** $u[1] = v[j]$ **then return**(u);

return(ε);

$mid \leftarrow \lfloor \frac{m}{2} \rfloor$;

Détermination des longueurs des PLSC pour $(u[1, mid], v)$ et $(\tilde{u}[1, m - mid], \tilde{v})$

$\text{Length}(u[1, mid], mid, v, n, \ell)$; $\text{Length}(\tilde{u}[1, m - mid], m - mid, \tilde{v}, n, \tilde{\ell})$;

Recherche de la meilleure partition

$max\ell \leftarrow -\infty$;

For j **from** 0 **to** n **do if** $\ell[j] + \tilde{\ell}[n - j] > max\ell$ **then** $max\ell \leftarrow \ell[j] + \tilde{\ell}[n - j]$; $k \leftarrow j$;

Appels récurifs et concaténation

If $k > 0$ **then** $w_1 \leftarrow \text{PLSC}(u[1, mid], mid, v[1, k], k)$ **else** $w_1 \leftarrow \varepsilon$;

If $k < n$ **then** $w_2 \leftarrow \text{PLSC}(u[mid + 1, m], m - mid, v[k + 1, n], n - k)$ **else** $w_2 \leftarrow \varepsilon$;

return $\text{CONCAT}(w_1, w_2)$;

Illustration (1)

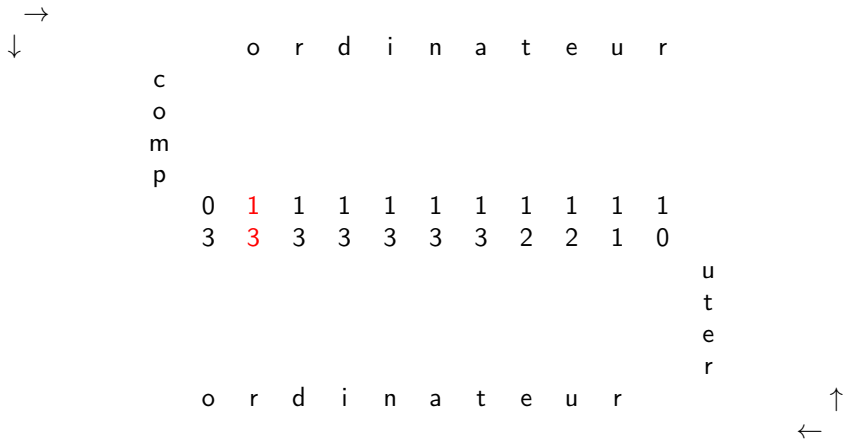
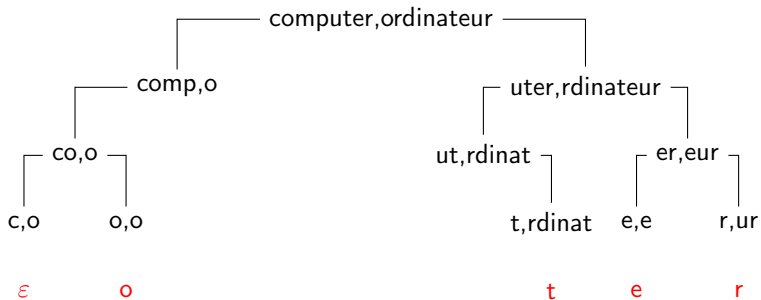


Illustration (2)



Analyse de complexité

Espace

ℓ et $\tilde{\ell}$ peuvent être des variables globales, d'où $\Theta(n)$.

Les variables locales vérifient $|w_1| \leq m$ et $|w_2| \leq m$.

A chaque appel récursif m est divisé par deux, d'où $\Theta(m)$.

En sommant, on obtient $\Theta(m + n)$.

Temps

$$T(m, n) \leq \max_{k \leq n} (T(\frac{m}{2}, k) + T(\frac{m}{2}, n - k)) + Cmn$$

$$T(i, n) \leq Cn \text{ pour } i \in \{0, 1\}$$

$$T(m, i) \leq Cm \text{ pour } i \in \{0, 1\}$$

Par induction pour $m \geq 1$ and $n \geq 1$,

$$T(m, n) \leq 2Cmn$$

Plan

Diviser pour régner

Programmation dynamique

3 Algorithmes gloutons

Optimisation discrète

Un problème générique.

- ▶ Soit E , un ensemble fini ;
- ▶ $Adm \subseteq 2^E$, un ensemble de sous-ensembles admissibles tel que le test $E' \in Adm ?$ se fasse en temps polynomial ;
- ▶ r , une fonction « récompense » de E dans \mathbb{N} .
- ▶ On cherche $E' \in Adm$ tel que $r(E') = \sum_{e \in E'} r(e)$ soit maximale.

Quelques exemples.

- ▶ Clique maximale dans un graphe ;
- ▶ Somme de sous-ensembles d'entiers inférieure à un seuil ;
- ▶ Arbre couvrant de poids maximal ;

Certains problèmes sont NP-complets (voir le cours de complexité) et d'autres admettent des **algorithmes en temps polynomial**.

Famille libre maximale

Soit un ensemble de vecteurs $\{\mathbf{v}_i\}_{i \leq n}$.

On cherche $I \subseteq \{1, \dots, n\}$ de taille maximale telle que $\{\mathbf{v}_i\}_{i \in I}$ soit une famille libre.

Un algorithme en temps polynomial.

```
 $I \leftarrow \emptyset;$   
For  $i$  from 1 to  $n$  do  
  If  $\{\mathbf{v}_j\}_{j \in I} \cup \{\mathbf{v}_i\}$  est une famille libre then  $I \leftarrow I \cup \{i\};$   
Return( $I$ );
```

Invariant de boucle.

Il existe I^* tel que $\{\mathbf{v}_j\}_{j \in I}$ peut être complétée en une famille libre maximale $\{\mathbf{v}_j\}_{j \in I^*}$ par des vecteurs de $\{\mathbf{v}_i, \dots, \mathbf{v}_n\}$.

- Si i n'est pas ajouté à I alors $i \notin I^*$.
- Si i est ajouté à I alors $\mathbf{v}_i = \sum_{j \in I^*} \alpha_j \mathbf{v}_j$ avec $\alpha_k \neq 0$ pour un $k \in I^* \setminus I$.
- D'ou $\{\mathbf{v}_j\}_{j \in I^* \setminus \{k\} \cup \{i\}}$ est une famille libre maximale.

Matroïde

(E, Adm) est un matroïde si :

- ▶ Adm est non vide et clos par inclusion :
 $E' \in Adm \wedge E'' \subseteq E' \Rightarrow E'' \in Adm$;
- ▶ Si $E', E'' \in Adm$ et $|E'| < |E''|$,
il existe $e \in E'' \setminus E'$ tel que $E' \uplus \{e\} \in Adm$.

Observation. Les ensembles admissibles maximaux ont la même taille.

Un algorithme générique.

```
Trier  $E$  par récompense décroissante ;  
 $F \leftarrow E$  ;  $E' \leftarrow \emptyset$  ;  
Repeat  
   $e \leftarrow \text{Extraire}(F)$  ;  
  If  $E' \cup \{e\} \in Adm$  then  $E' \leftarrow E' \cup \{e\}$  ;  
Until  $F = \emptyset$  ;  
Return( $E'$ ) ;
```

Preuve de l'algorithme

Invariant de boucle.

Il existe E^* tel que E' peut être complété en un ensemble admissible de récompense maximale E^* par des éléments de F .

Soit le matroïde (F, Adm') défini par $F' \in Adm'$ si $F' \cup E' \in Adm$.

Par hypothèse d'induction,

F^* est un ensemble admissible de récompense maximale de F
ssi $F^* \cup E'$ est admissible de récompense maximale de E .

- $E' \cup \{e\} \notin Adm$. e ne peut appartenir à un ensemble admissible de récompense maximale de F .
- $E' \cup \{e\} \in Adm$. Soit F^* , admissible de récompense maximale de F .

Cas 1 $\{e\} \in F^*$. immédiat.

Cas 2 $\{e\} \notin F^*$. Si $|F^*| > 1$, il existe $f_1 \in F^*$ tel que $\{e, f_1\}$ soit admissible.

En itérant, on obtient $F^+ = \{e, f_1, \dots, f_k\}$ admissible tel que :

$$F^* = \{f_0, \dots, f_k\}$$

Puisque $r(e) \geq r(f_0)$, F^+ a une récompense maximale dans F

et donc $E' \cup F^+$ a une récompense maximale dans E .

Arbre couvrant de poids maximal

Soit un graphe connexe $G = (V, E)$ dont les arêtes sont pondérées.
On cherche un arbre couvrant de poids maximal.

Le matroïde est l'ensemble des arêtes
dont les sous-ensembles admissibles forment des graphes acycliques.

Preuve

Soit E_1, E_2 des ensembles formant des graphes acycliques avec $|E_1| < |E_2|$.

Il existe une décomposition $G_1 = \biguplus_{j \leq m} T_{1,j}$ et $G_2 = \biguplus_{j \leq n} T_{2,j}$
telle que $T_{i,j} = (V_{i,j}, E_{i,j})$ soit un arbre et $E_i = \biguplus E_{i,j}$.

Cas 1 $\exists \{u, v\} \in E_2 \wedge \{u, v\} \cap \biguplus_j V_{1,j} = \emptyset$. Alors $E_1 \cup \{\{u, v\}\}$ convient.

Cas 2 $\exists j \exists \{u, v\} \in E_2 \wedge u \in V_{1,j} \wedge v \notin V_{1,j}$. Alors $E_1 \cup \{\{u, v\}\}$ convient.

Cas 3 $\forall \{u, v\} \in E_2 \exists j \{u, v\} \subseteq V_{1,j}$.

Alors $\forall j' \leq n \exists j \leq m V_{2,j'} \subseteq V_{1,j}$ ce qui implique $|E_2| \leq |E_1|$. (pourquoi?)

Observation

L'algorithme de Kruskal est une adaptation à l'arbre couvrant de poids minimal.

Ordonnement de tâches

Soit un ensemble de tâches I , de même durée avec date d'échéance d_i et pénalité r_i si non respect. On cherche un ordonnancement qui minimise les pénalités.

Observation

Ceci équivaut à maximiser les pénalités des tâches qui respectent leur échéance.

Le matroïde est l'ensemble des tâches

dont les sous-ensembles admissibles sont ordonnançables sans pénalité.

Preuve

Soit $J = i_1 \dots i_m$ et $J' = j_1 \dots j_n$ des sous-ensembles « ordonnancés » avec $m < n$. Soit j_k la dernière tâche de J' qui n'appartient pas à J .

On ordonnance $J \cup \{j_k\}$ comme suit :

- ▶ On ordonnance d'abord $J \setminus \{j_{k+1}, \dots, j_n\}$ selon l'ordre de J ;
- ▶ Puis on place $\{j_k, \dots, j_n\}$.

Toutes les tâches de cet ordonnancement respectent leur échéance. (pourquoi ?)

Comment tester efficacement si un sous-ensemble est ordonnançable ?

Généralisation : algorithmes gloutons

Un problème générique : Un espace de solutions

- ▶ structuré par une relation d'extension ;
- ▶ contenant une solution minimale ;
- ▶ telle que la récompense des solutions soit croissante par extension.

Un algorithme glouton générique.

```
sol ← solmin ;  
While Extend(sol) ≠ ∅ do  
  rew ←  $-\infty$  ;  
  For sol' ∈ Extend(sol) do  
    If  $r(sol') > rew$  then  $rew \leftarrow r(sol')$  ;  $sol \leftarrow sol'$  ;  
Return(sol) ;
```

Selon les problèmes, les algorithmes gloutons fournissent :

- ▶ la solution optimale ;
- ▶ une solution approchée avec garantie d'approximation.