Introduction à l'algorithmique : Paradigmes

Serge Haddad

LSV, ENS Paris-Saclay & CNRS & Inria

L3

Plan

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 $T_1[i_1] < T_2[i_2]$ **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** $T[i] \leftarrow T_2[j]$; $i \leftarrow i + 1$;

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

D'où :

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

Complexité mémoire du tri fusion

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

Les tableaux auxiliaires occupent n cellules.

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

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

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

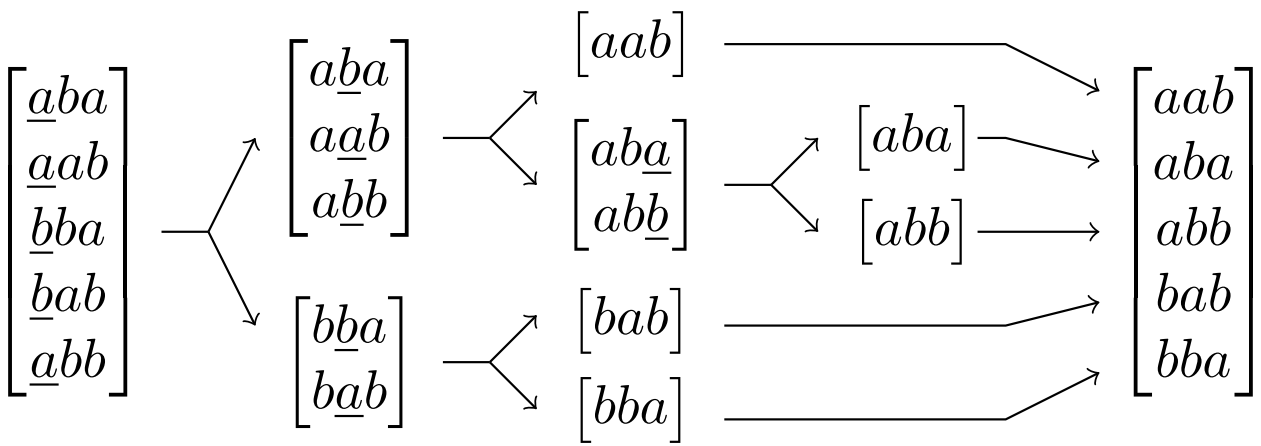
Tri par base : spécification

Σ est un alphabet ordonné.

Σ^k est ordonné lexicographiquement.

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

- Pour tout a, on insère les éléments de T commençant par a dans $T'[a]$.
- On trie récursivement les tableaux $T'[a]$ à partir de la deuxième lettre.
- On copie les tableaux $T'[a]$ dans T en suivant l'ordre de Σ.



Tri par base : implémentation

Tri(T, p, m)

If $p \leq 1$ **then return** ;

For $a \in \Sigma$ **do** $nb[a] \leftarrow 0$;

For i **from** 1 **to** p **do** $nb[T[i][m]] \leftarrow nb[T[i][m]] + 1$; $T'[T[i][m]][nb[T[i][m]] \leftarrow T[i]$;

If $m < k$ **then**

  For $a \in \Sigma$ **do** Tri$(T'[a], nb[a], m + 1)$;

$i \leftarrow 1$;

For $a \in \Sigma$ **do**

  For j **from** 1 **to** $nb[a]$ **do** $T[i] \leftarrow T'[a][j]$; $i \leftarrow i + 1$;

return

Tri$(T, n, 1)$

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

Observations.

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

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

Ce n'est pas un tri sur place.

Tri des NNI

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

Il y a approximativement 2^{26} français.

Tri par comparaison

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

Tri par base

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

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 (abcisse,ordonnée).
Créer le tableau Y d'indices de T trié selon les ordonnées.

Illustration.

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

Diviser pour régner (1)

Partitionner selon les coordonnées les points en deux sous-ensembles de taille approximativement égale.

Part(X, Y, n)

Partionnement 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** $X_d[i - mid] \leftarrow X[i]$;

Partionnement 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

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

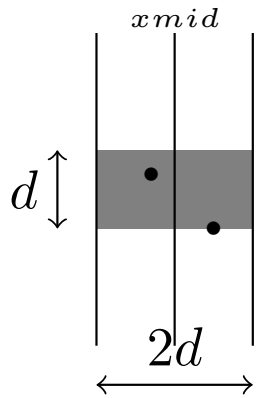
$d \leftarrow$ Short$(X_d, Y_d, \lceil \frac{n}{2} \rceil)$;

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

Prendre en compte les distances entre points gauches et droits.

...

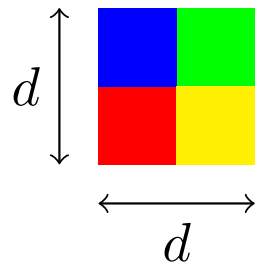
Si la distance entre un point *gauche* et un point *droit* est inférieure à d, alors :



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 $xmid - d \leq x \leq xmid + d$.
- 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.

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

Sélection

$i' \leftarrow 0$;

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

  If $T[Y[i]].x \geq xmid - d$ **and** $T[Y[i]].x \leq xmid + d$ **then**

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

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) ;

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

Plan

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 ≤ 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)

$Ready[k] \leftarrow$ **true** ;

If not $Ready[k-1]$ **then** Fibrec($k-1$) ;

$Fibval[k] \leftarrow Fibval[k-1] + Fibval[k-2]$;

return ;

Fib(n)

For i **from** 1 **to** 2 **do** $Ready[i] \leftarrow$ **true** ; $Fibval[i] \leftarrow 1$;

For i **from** 3 **to** n **do** $Ready[i] \leftarrow$ **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)

$new \leftarrow 1$; $old \leftarrow 1$;

For i **from** 3 **to** n **do**

  $temp \leftarrow new + old$;

  $old \leftarrow new$; $new \leftarrow temp$;

return new ;

Fib(n)

$new \leftarrow 1$; $old \leftarrow 1$;

For i **from** 3 **to** n **do**

  $new \leftarrow new + old$;

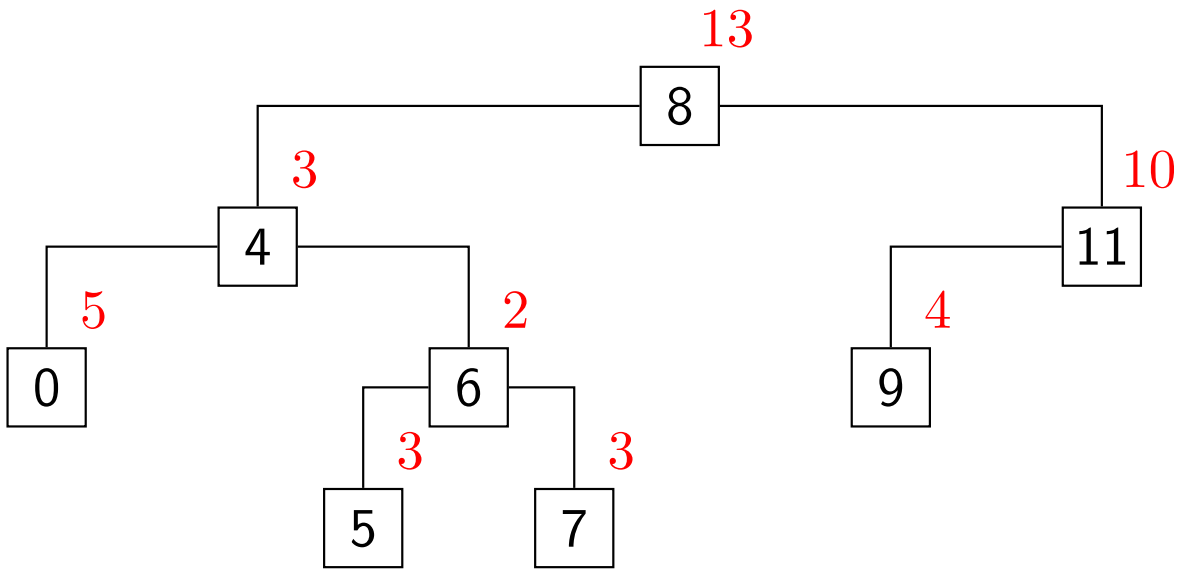
  $old \leftarrow new - old$;

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

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

Illustration.

$u =$ COMPUTER, $v =$ ORDINATEUR, $w =$ 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 $j < j'$.

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**
    $\ell[i,j] \leftarrow \ell[i-1,j]$;
    If $\ell[i,j] < \ell[i,j-1]$ **then** $\ell[i,j] \leftarrow \ell[i,j-1]$;
    If $\ell[i,j] \leq \ell[i-1,j-1]$ **and** $u[i] = v[j]$ **then** $\ell[i,j] \leftarrow \ell[i-1,j-1]+1$;

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 $w[k] \leftarrow u[i]$; $i \leftarrow i-1$; $j \leftarrow j-1$; $k \leftarrow k-1$;

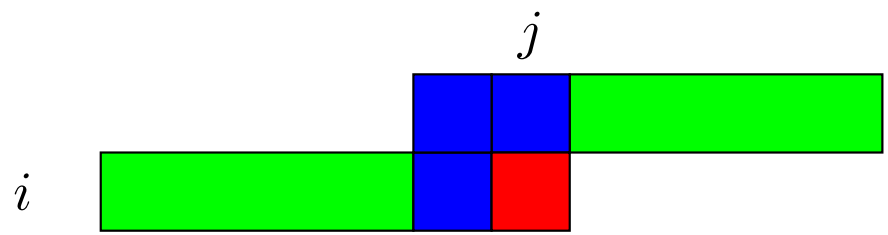
Considérations de mémoire

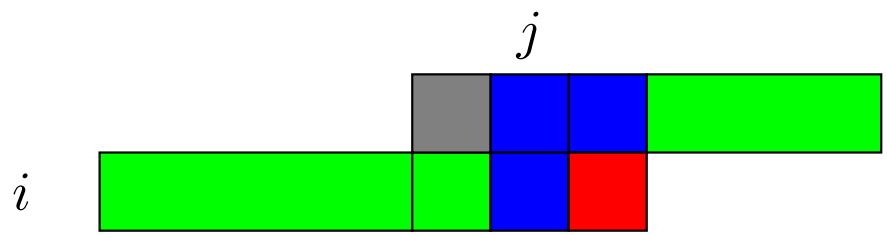
La mémoire utilisée est en $\Theta(mn)$.

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 $\ell[i, 1], \ldots, \ell[i, j-1], \ell[i-1, j-1], \ell[i-1, j], \ldots, \ell[i-1, n]$ qu'on notera $\ell[1], \ldots, \ell[j-1], old\ell, \ell[j], \ldots, \ell[n]$.





Calcul des longueurs

Length(u, m, v, n, ℓ)

For j **from** 1 **to** n **do** $\ell[j] \leftarrow 0$;

For i **from** 1 **to** m **do**

  $oldl \leftarrow 0$;

  For j **from** 1 **to** n **do**

    $temp \leftarrow \ell[j]$;

    If $j > 1$ **and** $\ell[j] < \ell[j-1]$ **then** $\ell[j] \leftarrow \ell[j-1]$;

    If $\ell[j] \leq oldl$ **and** $u[i] = v[j]$ **then** $\ell[j] \leftarrow oldl + 1$;

    $oldl \leftarrow temp$;

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.

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) ;

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

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

Length$(u[1, mid], mid, v, n, \ell)$; 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 récursifs et concaténation

$w_1 \leftarrow$ PLSC$(u[1, mid], mid, v[1, k], k)$;

$w_2 \leftarrow$ PLSC$(u[mid + 1, m], m - mid, v[k + 1, n], n - k)$;

return CONCAT(w_1, w_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

Optimisation discrète

Un problème générique.

- Soit E, un ensemble ;
- $Adm \subseteq 2^E$, 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 ;
- Arbre couvrant de poids minimal ;

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 $I \subseteq \{1,\ldots,n\}$ 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.

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

- Si i est ajouté à I alors $\mathbf{v}_i = \sum_{j\in I^*} \alpha_j \mathbf{v}_i$ 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 $E' \uplus \{e\} \in Adm$.

Par analogie, les ensembles admissibles sont appelés *indépendants*.

Observation. Les ensembles indépendants maximaux ont la même taille.

Un algorithme générique.

Trier E par récompense décroissante ;
$F \leftarrow E$; $E' \leftarrow \emptyset$;
Repeat
  $e \leftarrow$ 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.

> E' peut être complété en un ensemble indépendant 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 indépendant de récompense maximale de F
ssi $F^* \cup E'$ est indépendant de récompense maximale de E.

• $E' \cup \{e\} \notin Adm$. e ne peut appartenir à un ensemble indépendant de récompense maximale de F.

• $E' \cup \{e\} \in Adm$. Soit F^*, indépendant 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 indépendant.

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

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

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.

Quelques applications

Construction d'un arbre couvrant de poids minimal.

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

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

Un algorithme fondé sur ce matroïde : Kruskal
(voir la deuxième partie de ce cours)

Ordonnancement de tâches.

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

Le matroïde est l'ensemble des tâches
dont les sous-ensembles indépendants sont ordonnançables sans pénalité.

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.

$sol \leftarrow solmin$;
While Extend$(sol) \neq \emptyset$ **do**
  $rew \leftarrow -\infty$;
  For $sol' \in$ 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.