Algorithmique

Alexandre Duret-Lutz
adl@lrde.epita.fr

7 décembre 2015

Quatrième partie IV

Principaux paradigmes

1. Diviser pour régner
   - Algorithmes récursifs
   - Principe de diviser pour régner
   - Ex. : Multiplication de Polynômes
   - Ex. FFT
   - Multiplication de matrices
2. Programmation dynamique
   - Distance de Levenshtein
   - Principes
   - Chaîne de Multiplications de matrices
   - Plus longue sous-séquence commune
3. Algorithmes gloutons
   - Principe
   - Distributeur de Monnaie
   - Propriétés gloutonnes
   - Le problème de la loutre
   - Codage de Huffman

Diviser pour régner

Algorithmes récursifs

To understand what is recursion you must first understand recursion.

Principe de la récursivité

- On sait gérer des cas simples. (Cas de bases)
- On sait passer d'un cas compliqué à un cas plus simple.
- On passe des cas compliqués aux cas simples de proche en proche.

Avantages

- Algorithmes concis
- et faciles à prouver

Inconvénient

- Une appel récursif est assez coûteux (temps & espace). On a souvent avantage à implémenter des versions non-récursives. Mais les compilateurs peuvent optimiser les recursions terminales : un appel récursif en fin de fonction est transformé en boucle.

Exemple de récursion terminale

Power(v, n)
1 if $n = 0$ then return 1
2 return $v \times$Power($v, n - 1$)

L'appel à Power n'est pas terminal : il y a une multiplication ensuite.

Power(v, n)
1 return Power'($v, n, 1$)

Power'(v, n, res)
1 if $n = 0$ then return res
2 return Power($v, n - 1, v \times res$)

Maintenant la récursion est terminale, elle peut être réécrite :

Power'(v, n, res)
1 if $n = 0$ then return res
2 $n \leftarrow n - 1$
3 $res \leftarrow v \times res$
4 goto 1

Optimisation des appels terminaux

(tail call optimization)
Pour un compilateur, au milieu d'une fonction *Titi*(...) toute instruction du type « return *Toto*(...) » (où *Toto* a la même signature que *Titi*) peut être remplacée par une mise à jour des arguments suivie d'un goto au début de la fonction en question. Cela dérécursive les recursions terminales, mais pas uniquement.

Cela est fait par GCC à partir de `-O2` ou si `-foptimize-sibling-calls` est spécifié.

Parmi les algorithmes vus, sont optimisables automatiquement :

- BinarySearch
- Heapify
- StochasticSelection
- Le dernier des deux appels récursifs du quick sort
  (on a donc intérêt à y traiter la plus grosse des deux partitions)

Diviser pour régner

Algorithmes en trois étapes :

- Diviser le problème en plusieurs sous-problèmes plus simples
- Régner, i.e., résoudre les sous-problèmes (récursivement ou non)
- Combiner les résultats obtenus pour chaque sous-problème en une solution globale.

MergeSort et QuickSort sont des algorithmes « diviser pour régner ».
(En anglais : *divide en conquer algorithms*.)
Dans le cas de QuickSort l'étape de combinaison est vide.

Le complexité est de la forme :

$$T(n) = D(n) + aT(n/b) + C(n)$$

On la résout généralement avec le théorème général.

Multiplication de Polynômes

$$P_1(x) = a_n x^n + \cdots + a_1 x + a_0$$
$$P_2(x) = b_n x^n + \cdots + b_1 x + b_0$$
$$P_3(x) = P_1(x)P_2(x) = c_{2n} x^{2n} + \cdots + c_1 x + c_0$$

On a $c_k = \sum_{i+j=k} a_i b_j$.

Problème : Multiplication de deux polynômes
Entrée : Les a_i et b_i
Sortie : Les c_i.

Méthode simple

$$
\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_0 b_1 + a_1 b_0 \\
&\vdots \\
c_n &= a_0 b_n + \cdots + a_n b_0 \\
c_{n+1} &= a_1 b_n + \cdots + a_n b_1 \\
&\vdots \\
c_{2n} &= a_n b_n
\end{aligned}
$$

Entre c_0 est c_n : $(n+2)(n+1)/2$ multiplications.
Entre c_{n+1} est c_{2n} : $(n+1)n/2$ multiplications.
Total $(n+3)(n+1)/2$ multiplications.
On a $(n+3)(n+1)/2 - (2n+1)$ additions.
Au final $\Theta(n^2)$ opérations.

Méthode « diviser pour régner » (Karatsuba 1/3)

Supposons $P_1(x) = q_1 x + r_1$ et $P_2(x) = q_2 x + r_2$.
Alors $P_3(x) = q_1 q_2 x^2 + (q_1 r_2 + r_1 q_2)x + r_1 r_2$.
Mais on peut aussi écrire
$P_3 = q_1 q_2 x^2 + (q_1 q_2 + r_1 r_2 - (r_1 - q_1)(r_2 - q_2))x + r_1 r_2$.
On fait alors 3 multiplications au lieu de 4.

Généralisons : si degré$(P) = 2^m - 1 = n - 1$, alors
$P_1(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ peut s'écrire
$P_1(x) = \underbrace{(a_{n-1}x^{n/2-1} + \cdots + a_{n/2})}_{Q_1(x)} x^{n/2} + \underbrace{(a_{n/2-1}x^{n/2-1} + \cdots + a_0)}_{R_1(x)}$.
On utilise donc une approche diviser pour régner.

$$P_3(x) = x^n Q_1(x)Q_2(x) + x^{n/2}(Q_1(x)Q_2(x) + R_1(x)R_2(x) - (R_1(x) - Q_1(x))(R_2(x) - Q_2(x)) + R_1(x)R_2(x)$$

Méthode « diviser pour régner » (Karatsuba 2/3)

Attend deux tableaux P_1 et P_2 de taille n (avec $n = 2^m$) représentant les coefficients de deux polynômes de degré $n-1$. Retourne un tableau de taille $2n-1$ contenant les coefficients du produit.

Karatsuba(P_1, P_2, n)

1	if $n = 1$	$\Theta(1)$
2	return $[P_1[0] \times P_2[0]]$	
3	$R_1 \leftarrow P_1[0 : \frac{n}{2} - 1]$; $Q_1 \leftarrow P_1[\frac{n}{2} : n-1]$	$\Theta(n)$
4	$R_2 \leftarrow P_2[0 : \frac{n}{2} - 1]$; $Q_2 \leftarrow P_2[\frac{n}{2} : n-1]$	$\Theta(n)$
5	$T_1 \leftarrow$Karatsuba$(R_1, R_2, \frac{n}{2})$	$T(\frac{n}{2})$
6	$T_2 \leftarrow$Karatsuba$(Q_1, Q_2, \frac{n}{2})$	$T(\frac{n}{2})$
7	$T_3 \leftarrow Q_1 - R_1$; $T_4 \leftarrow R_2 - Q_1$	$\Theta(n)$
8	$T_5 \leftarrow$Karatsuba$(T_3, T_4, \frac{n}{2})$	$T(\frac{n}{2})$
9	$X[0 : n-2] \leftarrow T_2$; $X[n-1] \leftarrow 0$; $X[n : 2n-2] \leftarrow T1$	$\Theta(n)$
10	$X[\frac{n}{2} : \frac{3n}{2} - 2] \leftarrow X[\frac{n}{2} : \frac{3n}{2} - 2] + T_1 + T_2 + T_5$	$\Theta(n)$
11	return X	$\Theta(1)$

Méthode « diviser pour régner » (Karatsuba 3/3)

On a directement :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 3T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

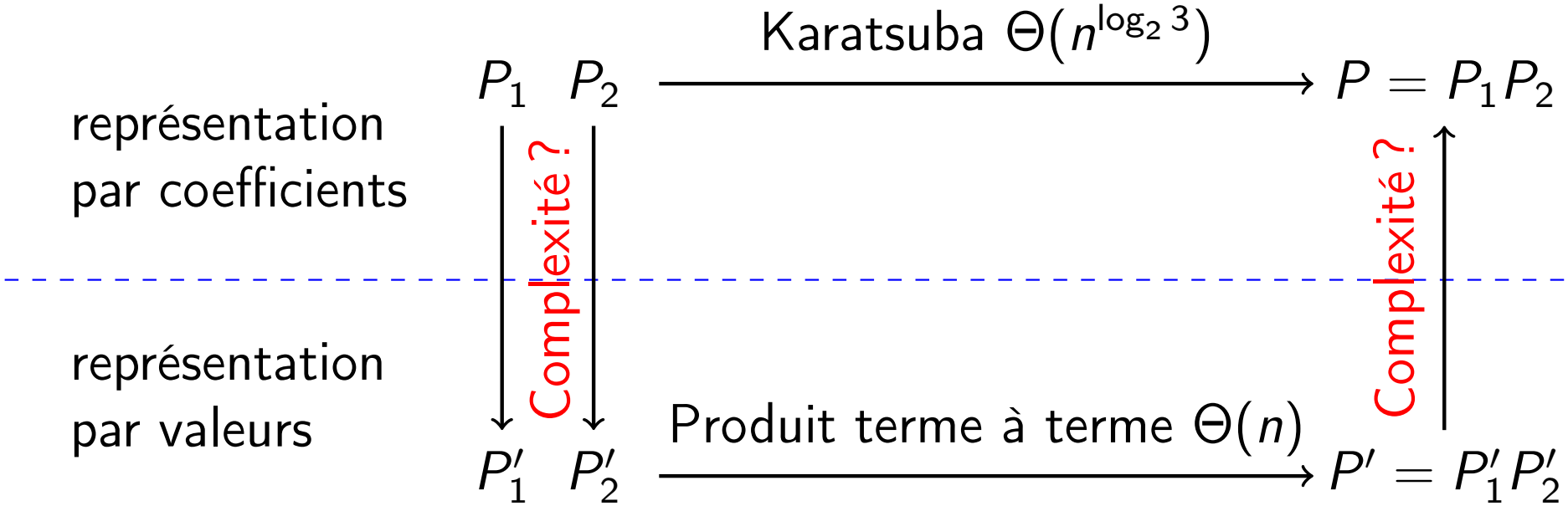
Par application du théorème général ($a = 3$, $b = 2$, $f(n) = \Theta(n)$ dominée par $n^{\log_2 3}$) on obtient :

$$T(n) = \Theta(n^{\log_2 3}) = \mathrm{O}(n^{1.59})$$

Autre approche de la multiplication de polynômes

- Un polynôme de degré $n-1$ est défini de manière unique par ses n coefficients. Il est aussi défini de manière unique par ses valeurs en n points différents (cf. unicité du polynôme de Lagrange).
- Multiplier deux polynômes représentés par leurs valeurs prises aux mêmes points est très simple : $(P \times Q)(x) = P(x) \times Q(x)$.

D'où l'idée de changer de représentation :



Évaluation d'un polynôme

Combien coûte l'évaluation d'un polynôme P de degré $n-1$ en un point x ?

$$\begin{aligned} v &= a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0 \\ &= ((\cdots((a_{n-1}x + a_{n-2})x + a_{n-3})x + \cdots)x + a_1)x + a_0 \end{aligned}$$

```
Eval(P, n, x)
1  xi = 1
2  v = P[0]
3  for i from 1 to n − 1 do
4      xi ← xi × x
5      v ← v + P[i] × xi
6  return v
```

$2n-2$ mult., $n-1$ additions.

```
EvalHorner(P, n, x)
1  xi = 1
2  v = P[n − 1]
3  for i from n − 2 to 0 do
4      v ← (v × x) + P[i]
5  return v
```

$n-1$ mult., $n-1$ additions.

Évaluation en n points

- Évaluer un polynôme de degré $n-1$ en un point coûte $\Theta(n)$.
  Il n'est pas possible de faire mieux. (Pourquoi ?)
- Évaluer un polynôme de degré $n-1$ en n points coûterait $\Theta(n^2)$ si l'on enchaîne les n évaluations.
  Avec une telle complexité, il est inutile de chercher à représenter un polynôme par valeurs pour pouvoir le multiplier plus vite : la conversion initiale est déjà plus chère que Karatsuba...

  Heureusement, en choisissant les points d'évaluation astucieusement, on peut faire (beaucoup) mieux que $\Theta(n^2)$.

L'astuce consiste évaluer le polynôme aux racines n-ièmes de l'unité :

$$\omega_n^k = e^{\frac{2\pi i k}{n}} \text{ pour } k \in [\![0, n[\![$$

Rappel : $\{\omega_4^k\}_k = \{1, i, -1, -i\}$.
C'est la base de la transformée de Fourier rapide (FFT).

FFT : Transformée de Fourier rapide (1/4)

Soit A un polynôme de degré $n-1$ tel que $n = 2^m$.
Notons a le tableau des coefficients de $A(X)$.

$$A(x) = a[n-1]x^{n-1} + a[n-2]x^{n-2} + \cdots + a[1]x + a[0]$$

La transformée de Fourier discrète (DFT) du tableau a est le tableau $b = \mathcal{F}(a)$ où

$$b[k] = A(\omega_n^k) = \sum_{j=0}^{n-1} a[j]\omega_n^{kj}$$

Un calcul naïf de $\mathcal{F}(A)$ demande $\Theta(n^2)$ opérations. Le choix des points d'évaluation nous permet de faire ces calculs plus rapidement avec une approche diviser pour régner.

FFT : Transformée de Fourier rapide (2/4)

L'idée : séparer a en deux tableaux $Even(a)$ et $Odd(a)$ formés respectivement des $n/2$ valeurs d'indices pairs et impairs.

$$
\begin{aligned}
b[k] &= \sum_{j=0}^{n-1} a[j]\omega_n^{kj} \quad \text{posons } j = 2m \text{ ou } j = 2m+1 \\
&= \sum_{m=0}^{n/2-1} a[2m]\omega_n^{2km} + \sum_{m=0}^{n/2-1} a[2m+1]\omega_n^{2km+k} \\
&= \sum_{m=0}^{n/2-1} a[2m](\omega_n^2)^{km} + \omega_n^k \sum_{m=0}^{n/2-1} a[2m+1](\omega_n^2)^{km} \\
&= \sum_{m=0}^{n/2-1} Even(a)[m]\omega_{n/2}^{km} + \omega_n^k \sum_{m=0}^{n/2-1} Odd(a)[m]\omega_{n/2}^{km}
\end{aligned}
$$

FFT : Transformée de Fourier rapide (3/4)

On voit apparaître la récurrence :

$$\forall k \in [\![0, n[\![,\ \mathcal{F}(a)[k] = \sum_{j=0}^{n-1} a[j]\omega_n^{kj}$$

$$\forall k \in [\![0, n[\![,\ \mathcal{F}(a)[k] = \sum_{j=0}^{n/2-1} Even(a)[j]\omega_{n/2}^{kj} + \omega_n^k \sum_{j=0}^{n/2-1} Odd(a)[j]\omega_{n/2}^{km}$$

$$\forall k \in [\![0, n/2[\![,\ \mathcal{F}(a)[k] = \mathcal{F}(Even(a))[k] + \omega_n^k \mathcal{F}(Odd(a))[k]$$

Pour trouver les termes de $[\![n/2, n[\![$ on s'appuie sur la périodicité des racines de l'unité : $\omega_{n/2}^{(k+n/2)j} = \omega_{n/2}^{kj}$ et $\omega_n^{k+n/2} = -\omega_n^k$.

$$\forall k \in [\![0, n/2[\![,\ \begin{cases} \mathcal{F}(a)[k] = \mathcal{F}(Even(a))[k] + \omega_n^k \mathcal{F}(Odd(a))[k] \\ \mathcal{F}(a)[k + n/2] = \mathcal{F}(Even(a))[k] - \omega_n^k \mathcal{F}(Odd(a))[k] \end{cases}$$

FFT : Transformée de Fourier rapide (4/4)

Calcul de la transformée discrète d'un tableau A de n éléments (Cooley-Tukey, 1965).

FFT(A, n)

1 if $n = 1$ then return A
2 for k from 0 to $n/2$ do
3 $\quad E[k] \leftarrow A[2k]$
4 $\quad O[k] \leftarrow A[2k+1]$
5 $FE \leftarrow$ FFT$(E, n/2)$
6 $FO \leftarrow$ FFT$(O, n/2)$
7 for k from 0 to $n/2 - 1$ do
8 $\quad t \leftarrow e^{2i\pi k/n} \times FO[k]$ `//std::polar(1.,(2*k*M_PI)/n)`
9 $\quad B[k] \leftarrow FE[k] + t$
10 $\quad B[k + n/2] \leftarrow FE[k] - t$
11 return B

$$T(n) = 2T(n/2) + \Theta(n) \implies T(n) = \Theta(n \log n)$$

Transformée de Fourier inverse (1/2)

$b[k] = \sum_{j=0}^{n-1} a[j]\omega_n^{kj}$ peut être vu matriciellement comme $b = Wa$ où

$$W = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} = [\omega_n^{ij}]$$

Montrons que $W^{-1} = \frac{1}{n}[\omega_n^{-ij}]$ en considérant $[p_{ij}] = WW^{-1}$. On a $p_{ij} = \frac{1}{n}\sum_k \omega_n^{ik}\omega_n^{-kj} = \frac{1}{n}\sum_k \omega_n^{(i-j)k}$. Clairement $p_{ii} = 1$. Et si $i \neq j$, $\sum_{k=0}^{n}(\omega_n^{i-j})^k = \frac{(\omega_n^{i-j})^n - 1}{\omega_n^{i-j} - 1} = \frac{(\omega_n^n)^{i-j} - 1}{\omega_n^{i-j} - 1} = \frac{1^{i-j} - 1}{\omega_n^{i-j} - 1} = 0$. Donc $P = Id$.

On en déduit que

$$a[k] = \frac{1}{n}\sum_{j=0}^{n-1} b[j]\omega_n^{-kj}$$

Transformée de Fourier inverse (2/2)

Connaissant $b = \mathcal{F}(a)$ comment utiliser $\mathcal{F}$ pour retrouver a ?

$$b[k] = \sum_{j=0}^{n-1} a[j]\omega_n^{kj} \iff a[k] = \frac{1}{n}\sum_{j=0}^{n-1} b[j]\omega_n^{-kj}$$

$$\iff a[k] = \frac{1}{n}\overline{\left(\sum_{j=0}^{n-1} \overline{b[j]}\omega_n^{kj}\right)}$$

$$b = \mathcal{F}(a) \iff a = \frac{1}{n}\overline{\mathcal{F}(\overline{b})}$$

On peut donc réutiliser la FFT pour calculer son inverse. Le surcoût de $\Theta(n)$ opérations (calculs des conjugués et divisions par n) étant négligeable devant le coût en $\Theta(n \log n)$ de la FFT.

Application au produit de polynômes

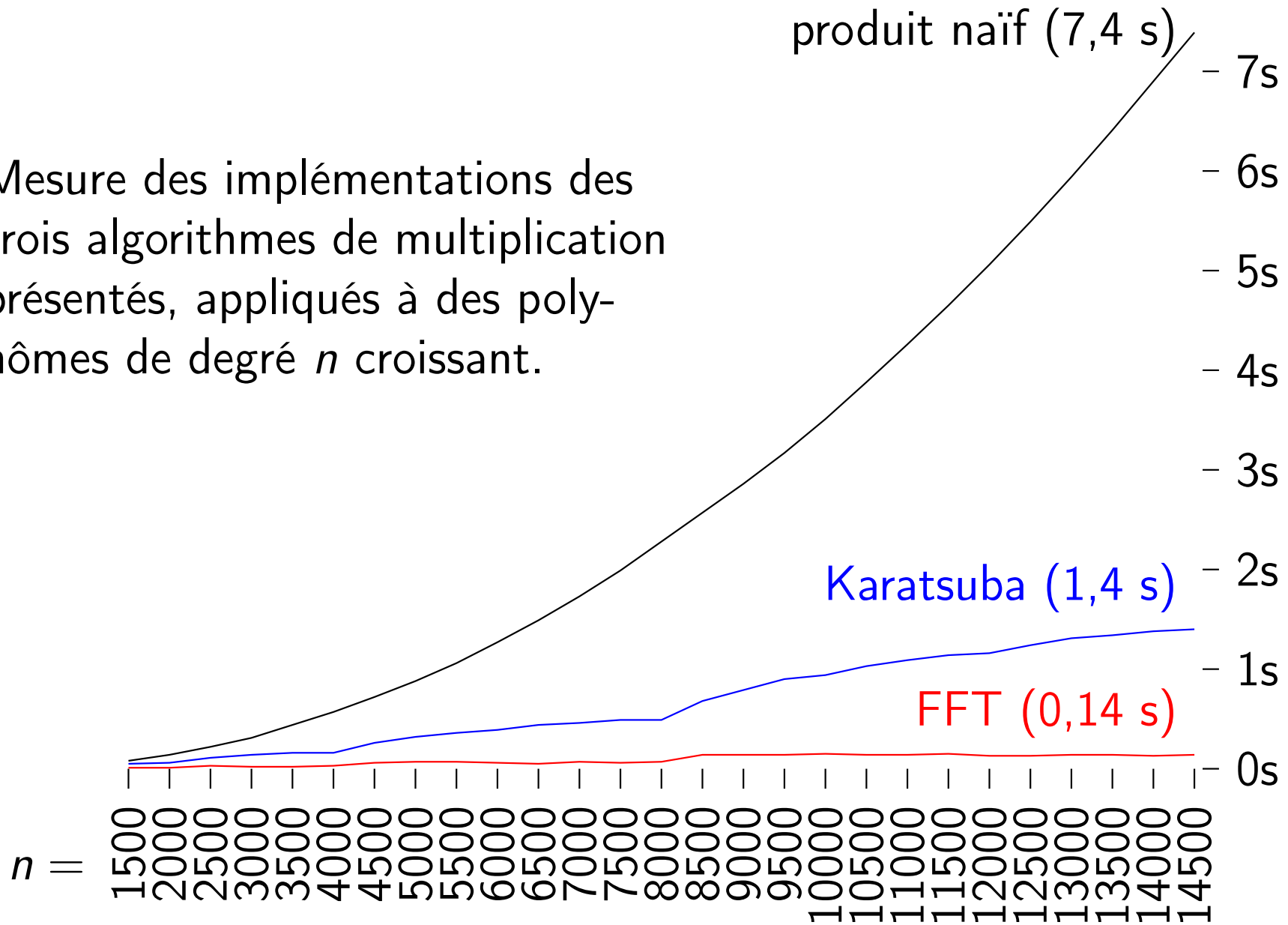
Attend deux tableaux P_1 et P_2 de taille n (avec $n = 2^m$) représentant les coefficients de deux polynômes de degré $n-1$. Retourne un tableau de taille $2n$ contenant les coefficients du produit.

	ProductFFT(P_1, P_2, n)	
1	$A[0:n-1] \leftarrow P_1; A[n:2n-1] \leftarrow 0$	$\Theta(n)$
2	$B[0:n-1] \leftarrow P_2; B[n:2n-1] \leftarrow 0$	$\Theta(n)$
3	$FA \leftarrow \mathrm{FFT}(A, 2n)$	$\Theta(n \log n)$
4	$FB \leftarrow \mathrm{FFT}(B, 2n)$	$\Theta(n \log n)$
5	for k in 0 to $2n-1$	$\Theta(n)$
6	$\quad FR[k] \leftarrow \overline{FA[k] \times FB[k]}$	$\Theta(n)$
7	$R \leftarrow \mathrm{FFT}(FR, 2n)$	$\Theta(n \log n)$
8	for k in 0 to $2n-1$	$\Theta(n)$
9	$\quad R[k] \leftarrow \overline{R[k]}/n$	$\Theta(n)$
10	return R	$\Theta(n)$

$$T(n) = \Theta(n \log n)$$

Comparaison des trois produits de polynômes

Mesure des implémentations des trois algorithmes de multiplication présentés, appliqués à des polynômes de degré n croissant.



Multiplication de matrices classique

Pour deux matrices A et B de taille $n \times n$, on calcule $C = A \times B$ avec

$$\forall i \in [\![1, n]\!], \forall j \in [\![1, n]\!], \quad c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

Quelle est la complexité du calcul de tous les coefficients de C ?

Multiplication de matrices récursive

Pour cet algorithme et le suivant, on suppose que n est une puissance de 2 (c'est-à-dire $n = 2^m$) ; il est toujours possible de s'y ramener en complétant les matrices avec des lignes et des colonnes remplies de 0. Découpons chacune de ces matrices en quatre blocs de taille $\frac{n}{2} \times \frac{n}{2}$:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}, \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

On a alors :

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Ceci suggère un algorithme récursif de calcul des coefficients de C. Quelle est sa complexité ?

Algorithme de Strassen

Comme dans l'algorithme précédent, on suppose les matrices découpées en quatre blocs de taille $\frac{n}{2} \times \frac{n}{2}$.
Posons

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \qquad M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$
$$M_3 = A_{1,1}(B_{1,2} - B_{2,2}) \qquad M_4 = A_{2,2}(B_{2,1} - B_{1,1})$$
$$M_5 = (A_{1,1} + A_{1,2})B_{2,2} \qquad M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$
$$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

on a alors

$$C_{1,1} = M_1 + M_4 - M_5 + M_7 \qquad C_{1,2} = M_3 + M_5$$
$$C_{2,1} = M_2 + M_4 \qquad C_{2,2} = M_1 - M_2 + M_3 + M_6$$

Ceci suggère un second algorithme récursif de calcul des coefficients de C. Quelle est sa complexité ?

Programmation dynamique

1. Diviser pour régner
   - Algorithmes récursifs
   - Principe de diviser pour régner
   - Ex. : Multiplication de Polynômes
   - Ex. FFT
   - Multiplication de matrices
2. Programmation dynamique
   - Distance de Levenshtein
   - Principes
   - Chaîne de Multiplications de matrices
   - Plus longue sous-séquence commune
3. Algorithmes gloutons
   - Principe
   - Distributeur de Monnaie
   - Propriétés gloutonnes
   - Le problème de la loutre
   - Codage de Huffman

Programmation dynamique

- Il s'agit encore une fois de définir le problème récursivement
- Mais cette fois-ci, les différents sous-problèmes partagent des sous-sous-problèmes.

- On veut éviter de recalculer chaque sous problème plusieurs fois
- On stocke ces résultats dans un tableau.

Distance de Levenshtein (1/4)

Distance d'édition entre deux chaînes (*string edit distance*). Si a, b sont deux lettres et u, v deux mots :

$$
\begin{aligned}
d_L(u, \varepsilon) &= |u| \\
d_L(\varepsilon, v) &= |v| \\
d_L(au, bv) &= \min\left(1 + d_L(u, bv),\ 1 + d_L(au, v),\ \delta_{a \neq b} + d_L(u, v)\right)
\end{aligned}
$$

Exemples :

$$
\begin{aligned}
d_L(\text{"}maks\text{"}, \text{"}make\text{"}) &= 1 \qquad && \text{1 remplacement} \\
d_L(\text{"}maks\text{"}, \text{"}emacs\text{"}) &= 2 \qquad && \text{1 insertion, 1 remplacement}
\end{aligned}
$$

Distance de Levenshtein (2/4)

Implémentation récursive directe :

```
StringEditDistance(u, v)
 1  if length(u) = 0 then
 2      return length(v)
 3  if length(v) = 0 then
 4      return length(u)
 5  if u[0] = v[0] then
 6      cost ← 0
 7  else
 8      cost ← 1
 9  d1 ←StringEditDistance(u[1..], v)
10  d2 ←StringEditDistance(u, v[1..])
11  d3 ←StringEditDistance(u[1..], v[1..])
12  return min(1 + d1, 1 + d2, cost + d3)
```

Notons $n = |uv|$.

$$T(n) = \Theta(1) + T(n-1) + T(n-1) + T(n-2)$$

$$T(n) \geq \Theta(1) + 2T(n-1)$$

$$T(n) = \Omega(2^n)$$

Distance de Levenshtein (3/4)

Exercice : représenter l'arbre correspondant au calcul récursif de StringEditDistance("bank","brake"). Repérer les nœuds identiques.

En fusionnant les nœuds identiques, on peut obtenir la grille suivante, où par exemple on voit que la distance entre "ban" et "bra" est 2. Chaque entrée de cette grille se calcule à partir des voisins nord, nord-ouest, et ouest.

	ε	b	r	a	k	e
ε	0	1	2	3	4	5
b	1	0	1	2	3	4
a	2	1	1	1	2	4
n	3	2	2	2	2	3
k	4	3	3	3	3	3

Les lignes de ce tableau peuvent être calculées en écrasant la précédente.

Distance de Levenshtein (4/4)

$\Theta(|u| \cdot |v|)$ opérations

```
StringEditDistance(u, v)
 1  for i ← 0 to length(u) do
 2      D[i] ← i
 3  for j ← 1 to length(v) do
 4      old ← D[0]
 5      D[0] ← j
 6      for i ← 1 to length(u) do
 7          if u[i − 1] = v[j − 1] then
 8              cost ← 0
 9          else
10              cost ← 1
11          tmp ← min(1 + D[i − 1], 1 + D[i], cost + old)
12          old ← D[i]
13          D[i] ← tmp
14  return tmp
```

Programmation dynamique : principes

Le mot « programmation » n'est pas celui qu'on imagine. Il faut le prendre dans le sens « plannification », « optimisation » : on cherche un meilleur chemin parmi des choix possibles.

L'approche s'applique aux problèmes qui ont des sous-structures optimales. C'est-à-dire que l'on peut construire une solution optimale à partir de solutions optimales pour des sous-problèmes.

- Casser le problème en sous-problèmes plus petits.
- Trouver des solutions optimales pour ces sous-problèmes, par la même méthode, récursivement.
- Utiliser les solutions optimales de chaque sous-problème pour trouver une solution optimale du problème original.

À chaque étape, mémoizer le résultat, c'est-à-dire sauvegarder le résultat correspondant à chaque entrée, pour ne pas le recalculer.

Chaîne de multiplications de matrices 1/10

Soient n matrices $A_1, A_2, \ldots A_n$ dont on veut calculer le produit $A_1 \cdot A_2 \cdots A_n$.

On peut évaluer cette expression après l'avoir parenthésée pour lever toute ambiguïté sur l'ordre des multiplications.

La multiplication de matrices étant associative, le résultat est indépendant du parenthésage.

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1(A_2(A_3A_4))) \\ &= (A_1((A_2A_3)A_4)) \\ &= ((A_1A_2)(A_3A_4)) \\ &= ((A_1(A_2A_3))A_4) \\ &= (((A_1A_2)A_3)A_4) \end{aligned}$$

Chaîne de multiplications de matrices 2/10

Le parenthésage peut avoir un impact sur le coût du produit. Le produit d'une matrice de taille $p \times q$ par une matrice de taille $q \times r$ génère une matrice de taille $p \times r$ après pqr multiplications.

Soient trois matrices A_1, A_2 et A_3 de dimensions 10×100, 100×5 et 5×50.

Combien de multiplications pour

- $((A_1 A_2) A_3)$?
- $(A_1 (A_2 A_3))$?

Moralité ?

Chaîne de multiplications de matrices 3/10

Problème Soient $A_1, A_2, \ldots A_n$, n matrices de dimensions $p_{i-1} \times p_i$. Comment choisir le parenthésage de $A_1 \cdot A_2 \cdots A_n$ pour minimiser le nombre de multiplications scalaires ?

Solution bovine On teste tous les cas possibles. Soit $P(n)$ le nombre de parenthésages possibles pour n matrices.

$$P(n) = \begin{cases} 1 & \text{si } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{sinon} \end{cases}$$

$P(n)$ désigne aussi le nombre d'arbres binaires de n feuilles. On peut montrer que

$$P(n+1) = \underbrace{\frac{1}{n+1} C_{2n}^n}_{n^{\text{e}} \text{ nombre de Catalan} = C_{2n}^n - C_{2n}^{n-1}} = \frac{4^n}{n^{3/2}\sqrt{\pi}}(1 + \mathrm{O}(1/n))$$

Nombre de parenthésages exponentiel en n.

Chaîne de multiplications de matrices 4/10

Sous-structure optimale

Nous notons $A_{i..j}$ la matrice résultant de l'évaluation du produit $A_i \cdot A_{i+1} \cdots A_j$. Un parenthésage optimal de $A_1 \cdot A_2 \cdots A_n$ sépare le produit entre A_k et A_{k+1} pour un certain k.

Dans notre solution optimale on commence donc par calculer les matrices $A_{1..k}$ et $A_{k+1..n}$ puis on les multiplie pour obtenir $A_{1..n}$.

Le coût du calcul est la somme des coûts des calculs des matrices $A_{1..k}$ et $A_{k+1..n}$ et de leur produit.

Le parenthésage du sous-produit $A_1 \cdots A_k$ (et celui de $A_{k+1} \cdots A_n$) doit être optimal : sinon, on le remplace par un parenthésage plus économique, et on obtient un parenthésage global plus efficace que... le parenthésage optimal !

Une solution optimale à une instance du problème de multiplication d'une suite de matrices utilise donc uniquement des solutions optimales aux instances des sous-problèmes.

Chaîne de multiplications de matrices 5/10

Définition récursive d'une solution optimale

- Sous-problème : coût minimum d'un parenthésage de $A_i \cdot A_{i+1} \cdots A_j$
- On note $m[i,j]$ le nombre minimum de multiplications scalaires pour $A_{i..j}$.
- $\forall i,\ m[i,i] = 0$ car $A_{i..i} = A_i$.
- Considérons $i < j$, et supposons le produit $A_i \cdot A_{i+1} \cdots A_j$ coupé entre A_k et A_{k+1}. Alors $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$. On doit choisir k pour minimiser $m[i,j]$.

$$m[i,j] = \begin{cases} 0 & \text{si } i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{si } i < j \end{cases}$$

On note $s[i,j]$ le k minimum pour $A_i \cdot A_{i+1} \cdots A_j$.

Chaîne de multiplications de matrices 6/10

BestCost(p, i, j)
1 if $i = j$ then return 0
2 $m[i, j] \leftarrow +\infty$
3 for $k \leftarrow i$ to $j - 1$ do
4 $q \leftarrow$BestCost(p, i, k) + BestCost$(p, k + 1, j) + p_{i-1} p_k p_j$
5 if $q < m[i, j]$ then $m[i, j] \leftarrow q$
6 return $m[i, j]$

Si $n = j - i + 1$, on a

$$T(n) = \Theta(1) + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$$

$$= 2 \sum_{k=1}^{n-1} T(k) + n = \Omega(2^n)$$

Chaîne de multiplications de matrices 7/10

Le nombre de sous problème $A_{i..j}$ avec $i \geq j$ est restreint. Il y a $n(n+1)/2 = \Theta(n^2)$ possibilités.
L'algorithme BestCost rencontre chaque sous-problème un nombre exponentiel de fois d'où sa complexité.
Cette propriété, dite des sous-problèmes superposés, est celle qui permet de faire de la programmation dynamique.

$$m[i,j] = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{si } i < j \end{cases}$$

On résout le problème diagonale par diagonale :

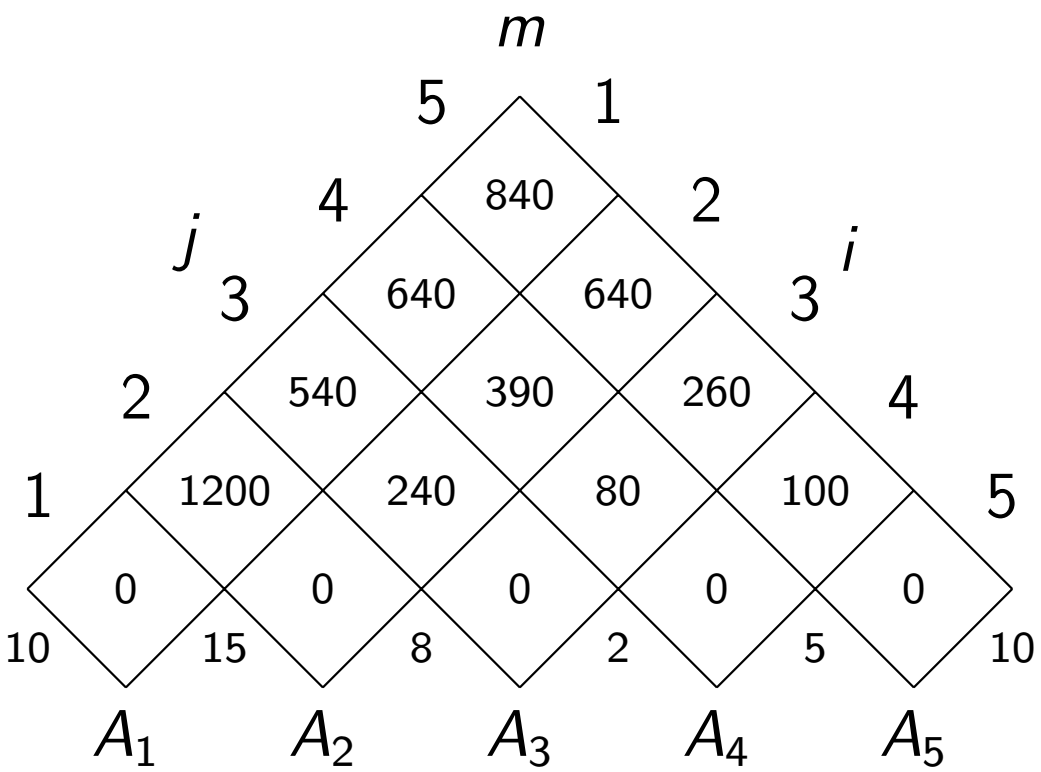
- On calcule $m[i,j]$ pour $i = j$, facile c'est 0.
- On calcule $m[i,j]$ pour $i - j = 1$ en fonction des précédents.
- On calcule $m[i,j]$ pour $i - j = 2$ en fonction des précédents.
- etc.

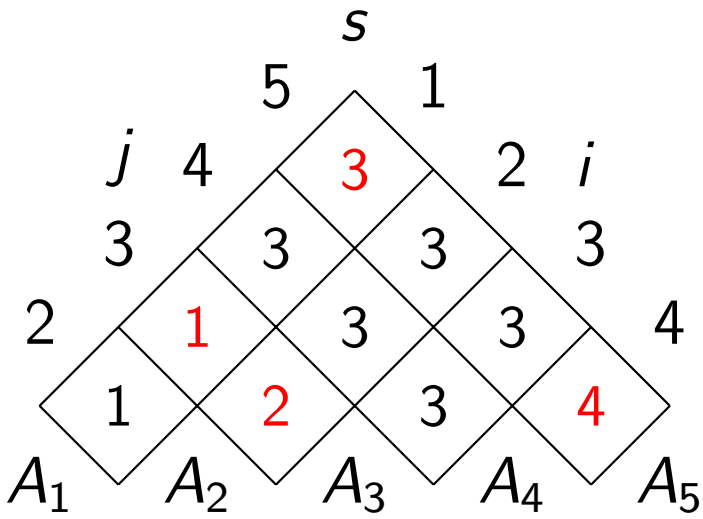
Chaîne de multiplications de matrices 8/10

MatrixChain(p)

```
 1  n ← length(p) − 1
 2  for i ← 1 to n do m[i, i] ← 0
 3  for l ← 2 to n do
 4      for i ← 1 to n − l + 1 do
 5          j ← i + l − 1
 6          m[i, j] ← +∞
 7          for k ← 1 to j − 1 do
 8              q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
 9              if q < m[i, j] then
10                  m[i, j] ← q
11                  s[i, j] ← k
12  return (m, s)
```

Chaîne de multiplications de matrices 9/10





A_1	10×15
A_2	15×8
A_3	8×2
A_4	2×5
A_5	5×10

$$m[i,j] = \begin{cases} 0 \text{ si } i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} \end{cases}$$

Meilleur parenthésage : $((A_1(A_2A_3))(A_4A_5))$

Chaîne de multiplications de matrices 10/10

L'algorithme MatrixChain possède trois boucles imbriquées qui font chacune au plus n itérations.

$$T(n) = \mathrm{O}(n^3)$$

Un décompte plus précis permet de montrer $T(n) = \Theta(n^3)$.

Plus longue sous-séquence commune (1/4)

Problème

Entrée : deux chaînes, p.ex. `"loutre"` et `"troupe"`.

Sortie : une plus longue sous-séquence commune, `"oue"` ou `"tre"`.

Approche bovine

Il y a 2^n sous-séquences dans une séquence de n lettres. Une telle aproche demande donc un temps exponentiel en la taille des chaînes.

Sous-structure optimale

Notons $X = x_1 x_2 \cdots x_m$ et $Y = y_1 y_2 \cdots y_n$ les chaînes d'entrée, et $Z = z_1 z_2 \cdots z_k$ une PLSC. Notons de plus X_i le i^{e}préfixe de X, c.-à-d. $X_i = x_1 x_2 \cdots x_i$. On a :

- $x_m = y_n \implies z_k = x_m = y_n$ et Z_{k-1} est une PLSC de X_{m-1} et Y_{n-1}
- $x_m \neq y_n \wedge z_k \neq x_m \implies Z$ est une PLSC de X_{m-1} et Y
- $x_m \neq y_n \wedge z_k \neq y_n \implies Z$ est une PLSC de X et Y_{n-1}

Plus longue sous-séquence commune (2/4)

Définition récursive de la longueur :

$$C[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ C[i-1,j-1]+1 & \text{si } i,j > 0 \text{ et } x_i = y_i \\ \max(C[i,j-1], C[i-1,j]) & \text{si } i,j > 0 \text{ et } x_i \neq y_i \end{cases}$$

On voit tout de suite que la procédure pour calculer la longueur maximale sera en $\Theta(mn)$: il suffit de remplir le tableau ligne par ligne jusqu'à obtenir $C[m,n]$.

Pour produire la PLSC il faut retenir les choix qui ont été faits, comme dans MatrixChain. On utilise un tableau B pour cela.

Plus longue sous-séquence commune (3/4)

```
LCS(X, Y)
 1  m ← length(X)
 2  n ← length(Y)
 3  for i ← 0 to m do C[i, 0] ← 0
 4  for j ← 1 to n do C[0, j] ← 0
 5  for i ← 1 to m do
 6      for j ← 1 to n do
 7          if x_i = y_j then
 8              C[i, j] ← C[i − 1, j − 1] + 1
 9              B[i, j] ← " ↖ "
10          else if C[i − 1, j] ≥ C[i, j − 1] then
11              C[i, j] ← C[i − 1, j]
12              B[i, j] ← " ↑ "
13          else
14              C[i, j] ← C[i, j − 1]
15              B[i, j] ← " ← "
16  return (B, C)
```

Plus longue sous-séquence commune (4/4)

		L	O	U	T	R	E
	0	0	0	0	0	0	0
T	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	← 1
R	0	↑ 0	↑ 0	↑ 0	↑ 1	↖ 2	← 2
O	0	↑ 0	↖ 1	← 1	↑ 1	↑ 2	↑ 2
U	0	↑ 0	↑ 1	↖ 2	← 2	↑ 2	↑ 2
P	0	↑ 0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2
E	0	↑ 0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3

Algorithmes gloutons

Algorithmes gloutons

Résolvent un problème d'optimisation, comme la programmation dynamique. Leur méthode est complètement différente.

Un algorithme glouton résoud un problème pas à pas en faisant localement le choix qu'il estime le meilleur. Il espère ainsi trouver la meilleure solution globale.

P.ex. un algorithme glouton du voyageur de commerce visite toujours la ville non-visitée la plus proche de la dernière ville visitée.

Ingrédients :

- un ensemble de candidats pour créer la solution
- une fonction `select`, pour choisir le meilleur candidat à ajouter à la solution
- une fonction `feasible` vérifie si un ensemble de candidats est faisable
- une fonction `is_solution` indique quand une solution a été

Principe d'un algorithme glouton

Notons C l'ensemble des candidats et S une tentative de solution.

```
1  while not is_solution(S) and C ≠ ∅ do
2      x ← select(C, S)
3      C ← C \ {x}
4      if feasible(S ∪ {x}) then
5          S ← S ∪ {x}
6  return S
```

Rendu de Monnaie

La monnaie : $C = \{2, 2, 1, 1, .50, .50, .20, .20, .10, , .10\}$.
Soit $v = 1.90€$ à rendre avec le moins de pièces possible.

L'algorithme glouton :

- $is\_solution(S) = \top \iff v = \sum_{i \in S} i$
- $select(C, S) = \max C$
- $feasible(S) = \top \iff v \geq \sum_{i \in S} i$

La solution trouvée : $S = \{1, .50, .20, .20\}$.

Cette solution est-elle optimale ?
Ici, oui. En général, cela dépend de C.
Par exemple avec $C = \{1, .50, .30, .30, .30, .5, .5, .5\}$ la solution gloutonne serait $\{1, .50, .30, .5, .5\}$ et non $\{1, .30, .30, .30\}$.

Propriétés gloutonnes

Propriété du choix glouton : on peut arriver à une solution globalement optimale en effectuant un choix localement optimal.

Le choix peut dépendre des choix faits jusque là, mais pas des choix qui seront faits ensuite (e.g. solutions des sous-problèmes).

Dans le rendu de monnaie à partir des pièces $v_1 \geq v_2 \geq \cdots \geq v_n$ on montre que pour tout montant v tel que $v_i < v \leq v_{i+1}$, il n'existe pas de solution optimale qui n'utilise pas v_{i+1}.

Sous-structure optimale : une solution optimale du problème contient la solution optimale de sous-problèmes.

Dans le rendu de monnaie, si $S \cup \{x\}$ est une solution optimale pour rendre v à partir de C, alors S est une solution optimale pour rendre $v - \mathit{valeur}(x)$ à partir de $C \setminus \{x\}$.

Le problème de la loutre (1/2)

Une loutre est au bord d'une rivère en train de pêcher des poissons pour sa petite famille, les poissons ne pèsent pas tous le même poids et ont des valeurs caloriques différentes.
La loutre veut maximiser le nombre de calories qu'elle va rapporter sachant qu'elle ne peut porter qu'au plus 10 kilos de poisson.

- Si la loutre sait découper un poisson, elle peut appliquer une stratégie gloutonne. Elle choisit les poissons en commençant par celui dont le ratio calories/kilos est le plus élevé ; elle découpe le dernier poisson pour ne pas dépasser 10kg.
- Sans découpage une stratégie gloutonne n'est pas applicable.

P.ex. 3 poissons :

2kg	6000kcal
4kg	10000kcal
6kg	12000kcal

La solution gloutonne utilise un poisson de 2kg et un de 4kg, alors que la solution optimale est 4kg+6kg.

Le problème de la loutre (2/2) : Prog. Dyn.

Pour la loutre sans découpage, le problème a bien la propriété de sous-structure optimale (si on retire un poisson de $x$kg de la solution et du problème, et si la loutre ne peut porter que $(10 - x)$kg alors la nouvelle solution est optimale) mais il n'a pas la propriété du choix glouton.

On peut résoudre le problème de la loutre (sans découpage) par la programmation dynamique... Notons w_i et c_i les poids et calories des différents poissons. On note $S_{k,w}$ les calories de la solution optimale pour les k premiers poissons jusqu'à concurrence du poids w. On a

$$S_{k,w} = \begin{cases} 0 & \text{si } k = 0 \text{ ou } w = 0 \\ S_{k-1,w} & \text{si } k > 0 \text{ et } 0 < w < w_k \\ \max(S_{k-1,w}, c_k + S_{k-1,w-w_k}) & \text{si} k > 0 \text{ et } w_k \leq w \end{cases}$$

D'où un algorithme faisant $1 + k$ passes sur un tableau $S[0..w]$.

Codage de Huffman (1/6)

- Le codage de Huffman est une méthode de compression de texte.
- Chaque caractère est représenté de manière optimale par une chaîne binaire, appelé un code.
- Pour économiser de l'espace, on utilise des codes de longueur variables.
- Ce code est basé sur une propriété statistique : plus un caractère est fréquent, plus son code est de longueur petite.
- Aucun code n'est préfixe d'un autre code : on parle de codage préfixe. (La décompression est aisée : on lit bit à bit jusqu'à reconnaître un mot du code.)

Codage de Huffman (2/6)

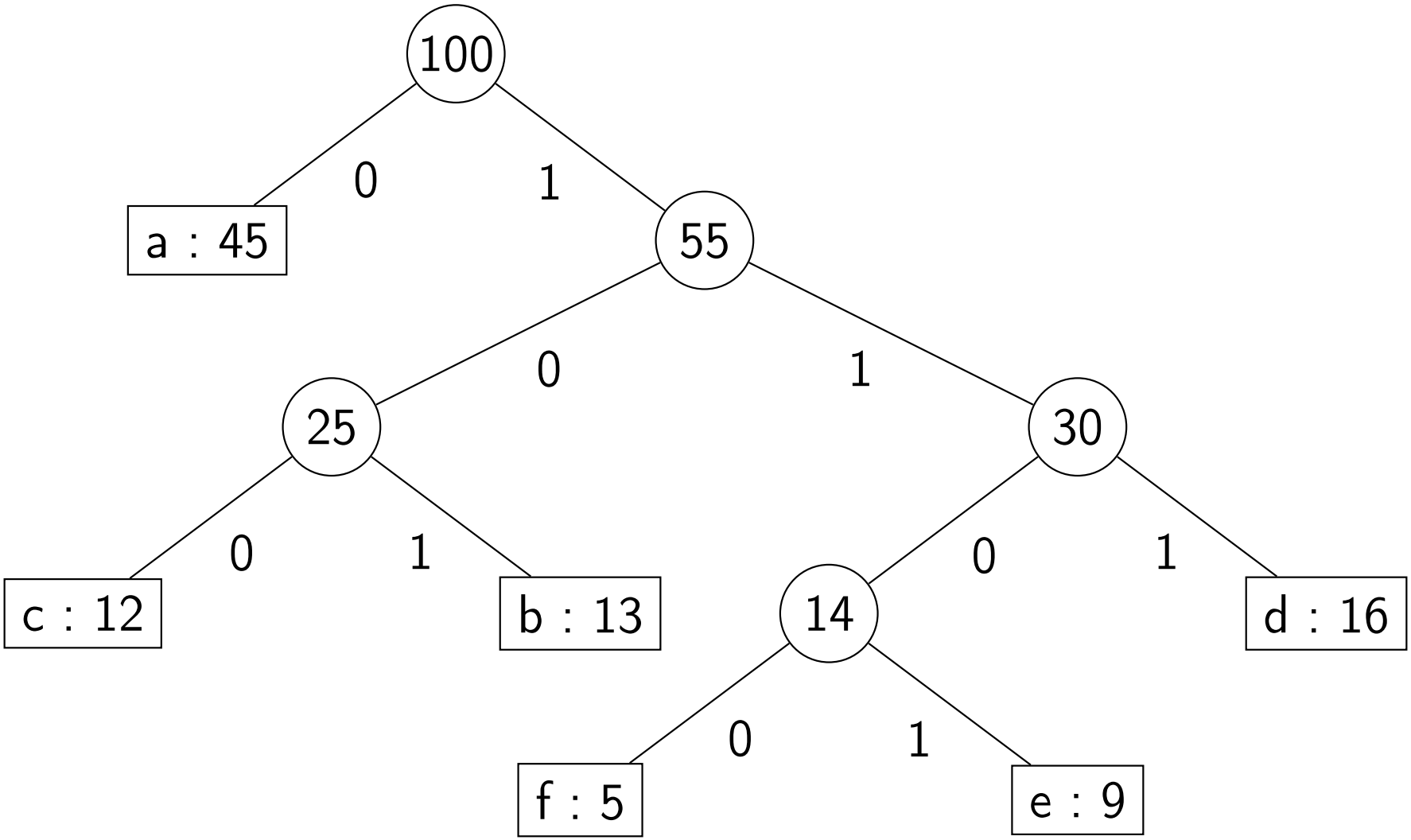
lettre	a	b	c	d	e	f
fréquence	45%	13%	12%	16%	9%	5%
code de longueur fixe	000	001	010	011	100	101
code de longueur variable	0	101	100	111	1101	1100

Un fichier composé de 100000 caractères avec les fréquences du tableau occupe 300000 bits avec les codes de longueur fixe et 224000 bits avec le code de longueur variable. Économie : 25%.

Codage de Huffman (3/6)

- Un arbre binaire peut être utilisé pour représenter n'importe quel codage binaire.
- Les arêtes de l'arbre sont étiquetées par des 0 (gauche) et 1 (droite).
- Le code de chaque lettre est donc un chemin dans l'arbre.
- Chaque feuille correspond à une lettre donnée.
- Décoder un caractère se fait en suivant un chemin de la racine à une feuille, le codage est l'opération inverse.
- On ajoute des cumuls de fréquences sur les nœuds.

Codage de Huffman (4/6)



Exemple de décodage : 110001001101 = *face*.

Codage de Huffman (5/6)

Un codage préfixe optimal est forcément représenté par un arbre binaire complet (sinon cela signifie que préfixes ne sont pas utilisés). L'arbre possède alors $|C|$ feuilles (le nombre de lettres) et $|C| - 1$ nœuds internes.

L'algorithme glouton de Huffman part d'un ensemble de $|C|$ feuilles et effectue un ensemble de $|C| - 1$ fusions pour construire l'arbre final.

```
Huffman(C)
 2   n ← |C|
 3   F ← C
 3   for i ← 1 to n − 1 do
 6       z ← NewNode()
 7       LeftChild(z) ← x ← ExtractMin(F)
 7       RightChild(z) ← y ← ExtractMin(F)
 7       freq(z) ← freq(x) + freq(y)
 9       Insert(F, z)
16   return ExtractMin(F)
```

Codage de Huffman (6/6)

Si la file de priorité F est implémentée par un tas.
$T(n) = \mathrm{O}(n) + (n-1)\mathrm{O}(\log n) = \mathrm{O}(n \log n)$

On peut montrer que

- Le codage de Huffman est un codage préfixe optimal. (Il n'existe pas de codage préfixe donnant de meilleure réduction.)
- La compression maximale accessible à partir d'un codage de caractères peut être obtenue avec un codage préfixe.
- Le codage de Huffman est donc un codage de caractère optimal.

Cela ne veut pas dire qu'il n'existe pas de meilleures compressions : simplement elles ne sont pas basées sur des codages de caractères.