LICENCE D'INFORMATIQUE

Sorbonne Université

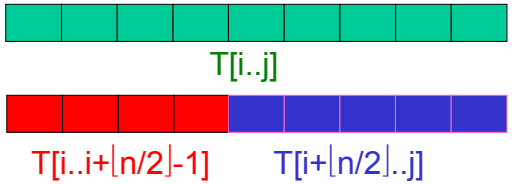
3I003 – Algorithmique
Cours 2 : Programmation récursive (diviser pour régner)

Année 2018-2019

Responsables et chargés de cours
Fanny Pascual
Olivier Spanjaard

Un exemple classique : tri fusion

Division de T[i..j] en 2 sous-tableaux

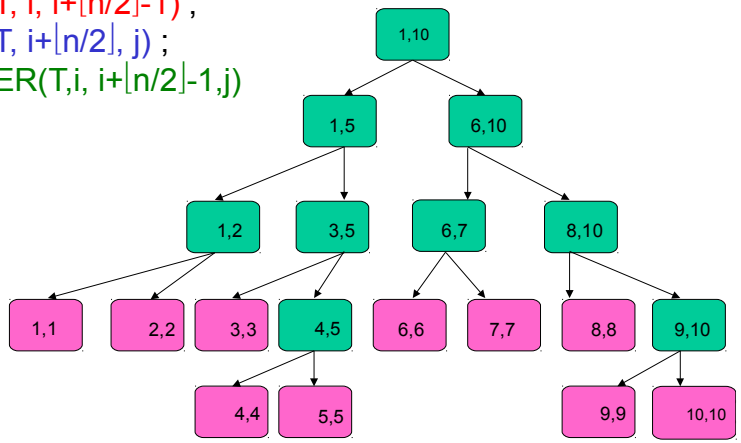


Appel récursif de TRI_FUSION(T, i, i+⌊n/2⌋-1) ;

Appel récursif de TRI_FUSION(T, i+⌊n/2⌋, j) ;

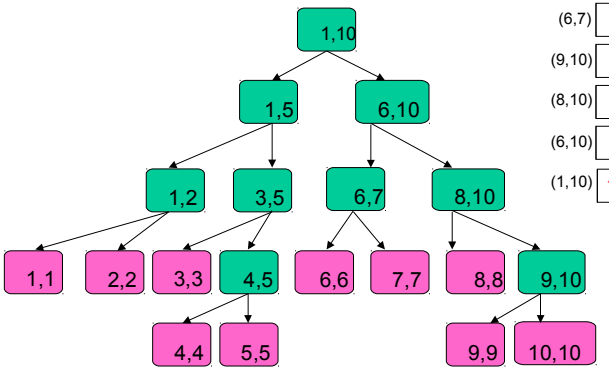
Construction de la solution par un algorithme d'interclassement sur place de T[i..i+⌊n/2⌋-1] et T[i+⌊n/2⌋..j].

```
Procédure TRI_FUSION(T, i, j);
n:=j-i+1;
Si n>1 alors
 TRI_FUSION(T, i, i+⌊n/2⌋-1) ;
 TRI_FUSION(T, i+⌊n/2⌋, j) ;
 INTERCLASSER(T,i, i+⌊n/2⌋-1,j)
Finsi.
```



appel terminal

appel non terminal (j>i)



	1	2	3	4	5	6	7	8	9	10
	12	3	6	1	7	2	10	15	3	1
(1,2)	3	12	6	1	7	2	10	15	3	1
(4,5)	3	12	6	1	7	2	10	15	3	1
(3,5)	3	12	1	6	7	2	10	15	3	1
(1,5)	1	3	6	7	12	2	10	15	3	1
(6,7)	1	3	6	7	12	2	10	15	3	1
(9,10)	1	3	6	7	12	2	10	15	1	3
(8,10)	1	3	6	7	12	2	10	1	3	15
(6,10)	1	3	6	7	12	1	2	3	10	15
(1,10)	1	1	2	3	3	6	7	10	12	15

L'ordre chronologique des appels à INTERCLASSER(T, i, i+⌊n/2⌋-1, j) est l'ordre des terminaisons des appels non terminaux :
(1,2), (4,5), (3,5), (1,5), (6,7), (9,10), (8,10), (6,10), (1,10)

Preuve de TRI_FUSION

Propriété: TRI_FUSION(T, i, j) se termine et trie le tableau T[i..j].

Preuve (récurrence forte sur la taille n=j-i+1)

- Cas de base : la propriété est vraie pour n=j-i+1=1.

- Etape d'induction : Soit T[i..j] un énoncé de taille n=j-i+1. Supposons la propriété vraie pour tous les énoncés de taille < n.

```
Procédure TRI_FUSION(T, i, j);
n:=j-i+1;
Si n>1 alors
 TRI_FUSION(T, i, i+⌊n/2⌋-1) ;
 TRI_FUSION(T, i+⌊n/2⌋, j) ;
 INTERCLASSER(T,i, i+⌊n/2⌋-1,j)
Finsi.
```

D'après l'induction ($\lfloor n/2 \rfloor < n$), après l'exécution de TRI_FUSION(T, i, i+⌊n/2⌋-1), T[i..i+⌊n/2⌋-1] est trié ;

D'après l'induction ($\lceil n/2 \rceil < n$), après l'exécution de TRI_FUSION(T, i+⌊n/2⌋, j), T[i+⌊n/2⌋..j] est trié ;

Après l'exécution d'INTERCLASSER(T,i,i+ ⌊n/2⌋ -1,j), le tableau T[i..j] est trié.

Considérons la suite récurrente U(n) définie par :
U(1)=0;
U(n)=U(⌊n/2⌋)+U(⌈n/2⌉)+n

U(n) majore toute solution de (R)

Calculons U(n) dans le cas particulier où $n=2^k$.

On a U(n)=U(2^k).
Posons V(k)=U(2^k). Combien vaut V(k) ?
On a :V(0)=0 et V(k)=2V(k-1)+2^k .

2^0 V(k)=2^1V(k-1)+2^k
2^1V(k-1)=2^2V(k-2)+2^k
2^2V(k-2)=2^3V(k-3)+2^k
.....
2^{k-1}V(1)=2^kV(0)+2^k
2^kV(0)= 0

En sommant, il vient :
$V(k)=k2^k=n\log_2 n$

Cas général (n quelconque):
On peut montrer que $U(n)=\Theta(n\log_2 n)$

Complexité de TRI_FUSION

Hypothèses:
1. On compte le nombre de « comparaisons de 2 clés »;
2. INTERCLASSER(T,i,k,j) exécute au plus n=j-i+1 comparaisons de clés.

On note T(n) la fonction complexité de TRI_FUSION.

Il résulte de la structure de contrôle de TRI_FUSION que:
T(1)=0;
T(n)≤T(⌊n/2⌋)+T(⌈n/2⌉)+n (R)

```
Procédure TRI_FUSION(T, i, j);
n:=j-i+1;
Si n>1 alors
 TRI_FUSION(T, i, i+⌊n/2⌋-1) ;
 TRI_FUSION(T, i+⌊n/2⌋, j) ;
 INTERCLASSER(T,i,i+⌊n/2⌋-1,j)
Finsi.
```

Rappel de la séance précédente

- On s'est intéressé au comptage des lapins, via le calcul du $n^{ème}$ terme de la suite de Fibonacci :

$F_0 = 1$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$

- On a vu deux algorithmes pour ce problème :
  - Algorithme **`Fib1`** de complexité $O(2^{0.694n})$
  - Algorithme **`Fib2`** de complexité $O(n^2)$

Algorithme **Fib3** matriciel

On écrit $F_1=F_1$ et $F_2=F_0+F_1$ en matriciel :

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

où $F_0=0$ et $F_1=1$. De même,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Et plus généralement

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Calcul de F_n : élever la matrice à la puissance n.

Algorithme **Fib3** matriciel

- Le problème se réduit à calculer :

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

- Réalisable en *O(log$_2$ n)* produits matriciels (plus précisément, mises au carré) :

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{2^{2^{2\cdots}}}$$

Analyse de la complexité de **Fib3**

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \quad \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix} \quad \begin{pmatrix} 13 & 21 \\ 21 & 34 \end{pmatrix}$$

- A chaque itération, la mise au carré requiert :
  - 4 additions,
  - 8 multiplications.

  } On va s'intéresser aux nombres d'opérations élémentaires induites par les multiplications
- Initialement, chaque nombre dans la matrice tient sur *1* = *2^0* bit.
- A la *k^e* itération de **Fib3**, il tient sur *2^k* bits (les longueurs *sont doublées* à chaque itération).

Analyse de la complexité de **Fib3**

- A la *k^e* itération (mise au carré) de **Fib3**, une composante de la matrice tient sur *2^k* bits.
- Supposons que la multiplication de deux nombres de *n* bits requiert *O(n^a)* opérations élémentaires
- Le nombre d'opérations élémentaires induites par les multiplications au cours des *log$_2$n* itérations est :

$$\sum_{k=0}^{\log_2 n} 2^{ka} = \frac{2^{a(\log_2 n+1)}-1}{2^a-1} \in O(2^{a\log_2 n}) = O(n^a)$$

(somme des *(log$_2$n +1)* termes d'une suite géométrique de premier terme *1* et de raison *2^a*)

Analyse de la complexité de `Fib3`

En conclusion : pour savoir si **`Fib3`** est plus rapide que **`Fib2`**, il faut s'intéresser à la complexité de la multiplication de deux nombres de *n* bits. Si on peut le faire avec une complexité moindre que $O(n^2)$, alors **`Fib3`** est plus rapide.

Multiplication

Intuitivement plus difficile qu'une addition… Une analyse permet de *quantifier* cela.

```
[13]                    1   1   0   1
[11]                    1   0   1   1
                ---------------------------------
                        1   1   0   1
                    1   1   0   1
                0   0   0   0
            1   1   0   1
            -------------------------------------
[143]   1   0   0   0   1   1   1   1
```

Pour multiplier deux nombres de n bits : créer un tableau de n sommes intermédiaires, et les additionner. Chaque addition est en $O(n)$… un total en $O(n^2)$.

Il semble donc que l'addition est linéaire, alors que la multiplication est quadratique.

Doit-on en conclure que **`Fib3`** ne fait pas mieux que **`Fib2`** ?

Multiplication égypto-franco-russe

- Il existe d'autres façons de multiplier !
- La paternité de la méthode ci-dessous varie selon les sources (d'où le titre du transparent) :

11	13
5	26
~~2~~	~~52~~
1	104
	143

Répéter
Diviser par 2 le nb de gauche
Multiplier par 2 le nb de droite
Jusqu'à ce que nb de gauche = 1
Retourner la somme des nbs de la colonne de gauche qui sont face à des nbs impairs

Réécrit en binaire...

1011	1101
101	11010
~~10~~	~~110100~~
1	1101000
	10001111

Multiplier par 2 un nombre binaire : insérer un bit 0 en fin de représentation

Diviser par 2 un nombre binaire : supprimer le dernier bit de la représentation

Hum... ça ne rappelle pas quelque chose ?

Comparaison des deux méthodes

Les deux méthodes réalisent en fait les mêmes calculs !
→ même complexité

```
1011          1 1 0 1
101           1 1 0 1 0
10          1 1 0 1 0 0
1           1 1 0 1 0 0 0
-----------------------------
            1 0 0 0 1 1 1 1
```

```
[13]          1   1   0   1
[11]          1   0   1   1
-----------------------------
              1   1   0   1
          1   1   0   1   0
      0   0   0   0
  1   1   0   1   0   0   0
-----------------------------
[143] 1   0   0   0   1   1   1   1
```

Une multiplication récursive

```
fonction multiplier(x,y)
Entrée: Deux entiers x et y sur n bits
Sortie: Leur produit
si x = 1 retourner y
z = multiplier(x/2,y) (division entière)
si x est pair retourner 2z
sinon retourner y+2z
```

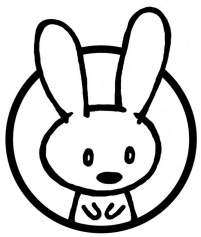
Analyse de cet algorithme en TD

Une multiplication récursive

Remarque importante : cet algorithme récursif peut en fait se voir comme une autre formulation de la multiplication égypto-franco-russe...

MAIS c'est formateur de le voir en exercice.

Néanmoins, cela ne fait pas avancer le comptage des lapins, car **`Fib3`** n'est toujours pas de meilleure complexité que **`Fib2`**...



Résumé des épisodes précédents

- On s'est intéressé au comptage des lapins, via le calcul du $n^{ème}$ terme de la suite de Fibonacci :
  - algorithme **`Fib1`** de complexité $O(2^{0.694n})$
  - algorithme **`Fib2`** de complexité $O(n^2)$
  - Algorithme **`Fib3`** matriciel :

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

La complexité de **`Fib3`** est équivalente à la complexité de multiplication de deux entiers de n bits. On cherche donc à concevoir un algorithme de multiplication de complexité $O(n^{\alpha})$ avec $\alpha < 2$.

Diviser pour régner

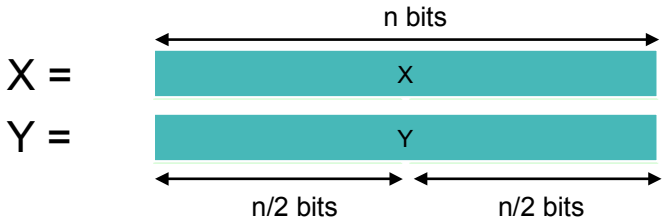
Pour concevoir des algorithmes plus rapides :

DIVISER un problème en sous-pbs plus petits

RESOUDRE les sous-problèmes récursivement

FUSIONNER les réponses aux sous-pbs afin d'obtenir la réponse au problème de départ

Application à la multiplication

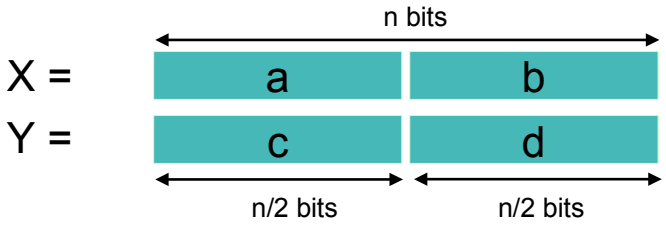


$X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$

$X \times Y = ac\,2^{n} + (ad + bc)\,2^{n/2} + bd$

Pour simplifier la présentation, mais sans perte de généralité, on suppose que n est une puissance de 2.

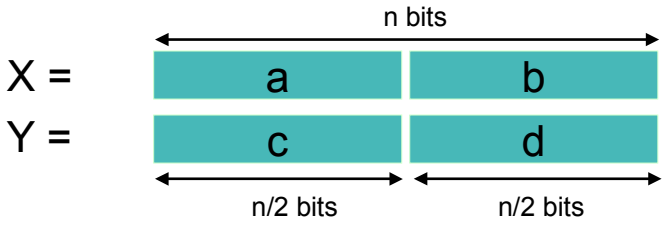
Application à la multiplication



$X \times Y = ac\,2^{n} + (ad + bc)\,2^{n/2} + bd$

```
fonction mult(x,y)
Entrée: Deux entiers x et y sur n bits
Sortie: Leur produit
si n = 1 retourner xy
sinon partitionner x en a,b et y en c,d
      retourner mult(a,c)2^n +
            (mult(a,d)+ mult(b,c))2^(n/2) + mult(b,d)
```

Ce qui donne en décimal



$X = a\,10^{n/2} + b \qquad Y = c\,10^{n/2} + d$

$X \times Y = ac\,10^{n} + (ad + bc)\,10^{n/2} + bd$

Exemple (en décimal)

12345678 * 21394276

1234*2139 1234*4276 5678*2139 5678*4276

12*21 12*39 34*21 34*39

1*2 1*1 2*2 2*1

2 1 4 2

Ainsi : $12*21 = 2*10^2 + (1 + 4)10^1 + 2 = 252$

X = a b

Y = c d

$X \times Y = ac\ 10^n + (ad + bc)\ 10^{n/2} + bd$

Exemple (en décimal)

12345678 * 21394276

1234*2139 1234*4276 5678*2139 5678*4276

252 468 714 1326

$*10^4 + *10^2 + *10^2 + *1$ = 2639526

X = a b

Y = c d

$X \times Y = ac\ 10^n + (ad + bc)\ 10^{n/2} + bd$

Exemple (en décimal)

12345678 * 21394276

2639526 5276584 12145242 24279128

$*10^8 + *10^4 + *10^4 + *1$

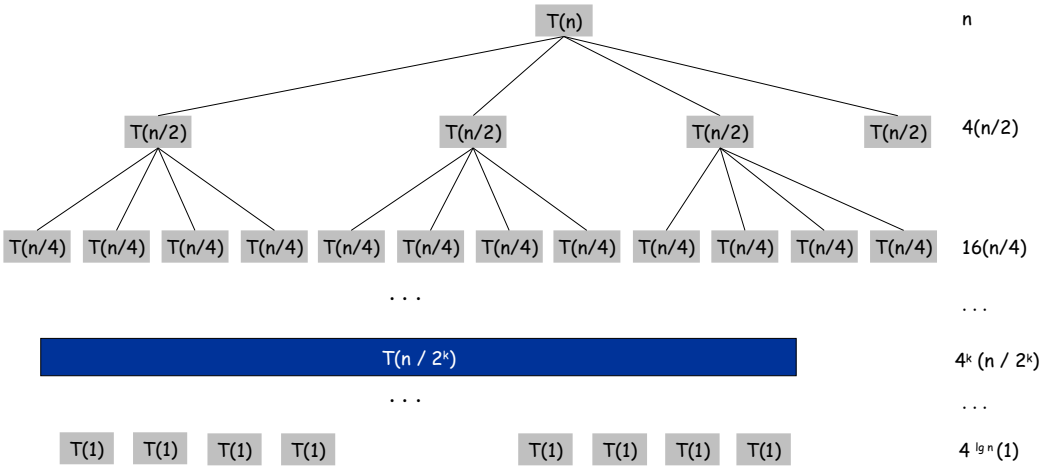
= 264126842539128

X = a b

Y = c d

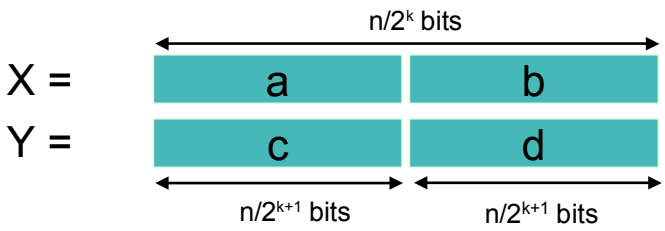
$X \times Y = ac\ 10^n + (ad + bc)\ 10^{n/2} + bd$

Arbre des appels récursifs



A chaque niveau k de l'arbre, il y a 4^k nœuds.

Calculs en un nœud $T(n/2^k)$

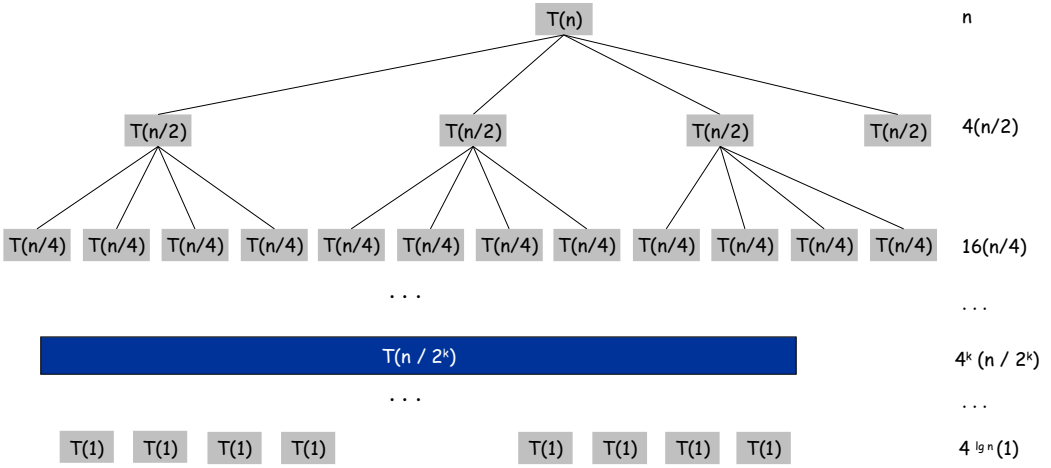


$$X \times Y = ac\ 2^{n/2^k} + (ad + bc)\ 2^{n/2^{k+1}} + bd$$

A un nœud $T(n/2^k)$, on réalise une multiplication par $2^{n/2^k}$ ($n/2^k$ décalages vers la gauche), une autre par $2^{n/2^{k+1}}$, et trois additions avec des nombres comportant au plus $n/2^{k-1}$ bits, soit :

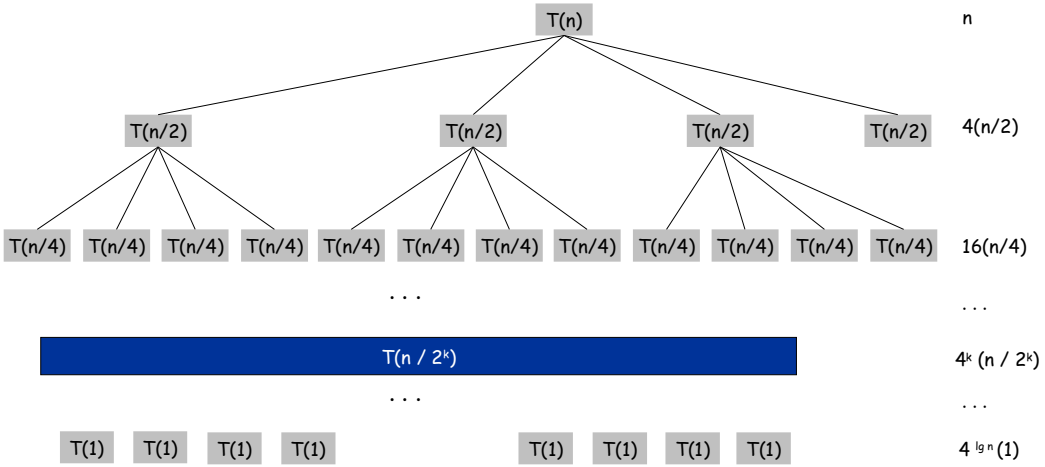
$O(n/2^k)$ opérations élémentaires

Arbre des appels récursifs



$$T(n) = \sum_{k=0}^{\log_2 n} 4^k \left(\frac{n}{2^k}\right) = \sum_{k=0}^{\log_2 n} n\left(\frac{4}{2}\right)^k = n\left(2^{\log_2 n + 1} - 1\right) = n(2n-1)$$

Arbre des appels récursifs



Complexité en $O(n^2)$

Question

- Soient deux nombres complexes a+bi et c+di
- Leur produit : (a+bi)(c+di) = [ac-bd] + [ad+bc]i
- Entrée : a, b, c, d
- Sortie : ac-bd, ad+bc

Si la multiplication de deux nombres coûte 1€ et leur addition coûte 1 centime, quelle est la façon la moins coûteuse d'obtenir la sortie à partir de l'entrée ?

Pouvez-vous faire mieux que 4.02€ ?

La solution de Gauss à 3.05€

Entrée : a,b,c,d

Sortie : ac-bd, ad+bc

1 centime : $P_1 = a + b$

1 centime : $P_2 = c + d$

1€ : $P_3 = P_1P_2 = ac + ad + bc + bd$

1€ : $P_4 = ac$

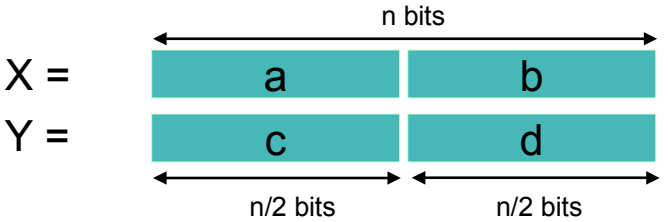
1€ : $P_5 = bd$

1 centime : $P_6 = P_4 - P_5 = ac - bd$

2 centimes : $P_7 = P_3 - P_4 - P_5 = bc + ad$

Carl Friedrich Gauss
1777-1855

`mult` gaussifiée (Karatsuba, 1962)

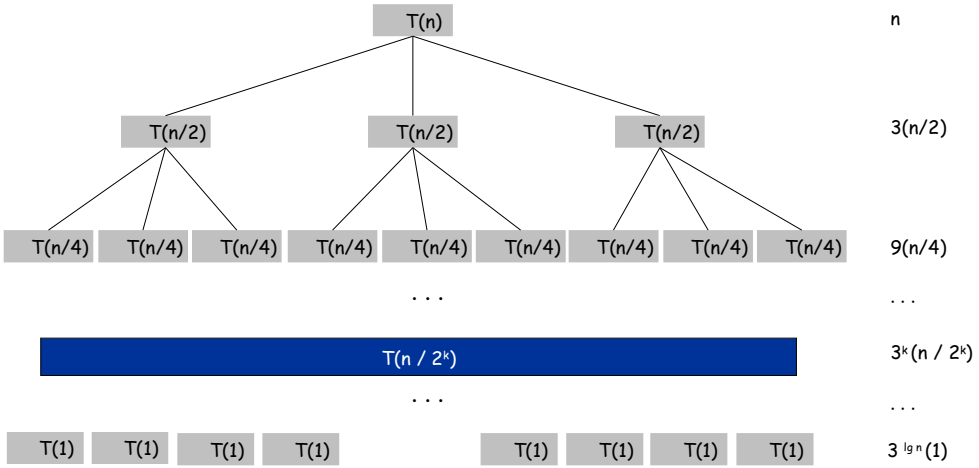


$X \times Y = ac\ 2^n + (ad + bc)\ 2^{n/2} + bd$

Anatolii Alexeevich
Karatsuba,
1937-2008

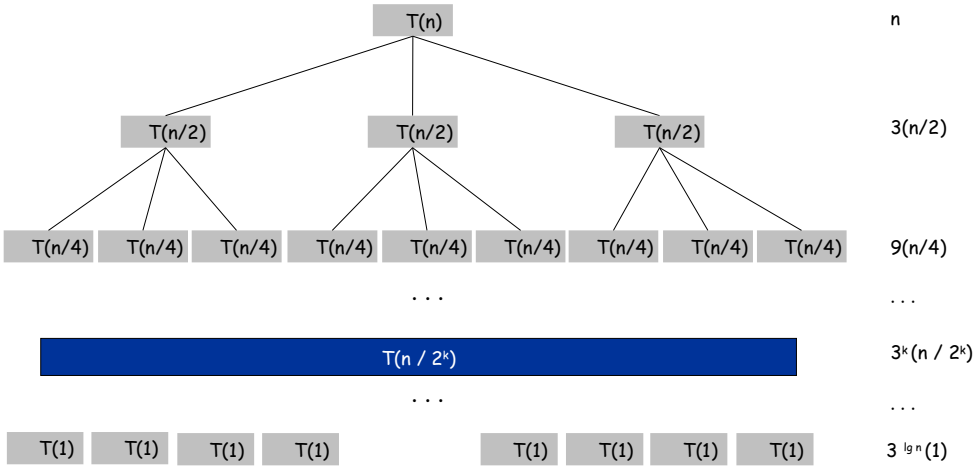
```
fonction mult2(x,y)
Entrée: Deux entiers x et y sur n bits
Sortie: Leur produit
si n = 1 retourner xy
sinon
    partitionner x en a,b et y en c,d
    P3= mult(a+b,c+d); P4= mult(a,c); P5= mult(b,d)
    retourner P4.2^n + (P3-P4-P5).2^(n/2) + P5
```

`mult2` : arbre des appels récursifs



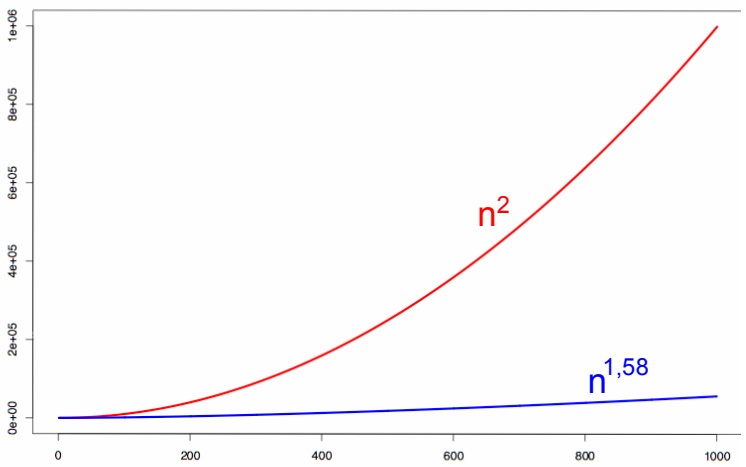
$$T(n) = \sum_{k=0}^{\log_2 n} n\left(\frac{3}{2}\right)^k = n\,\frac{\left(\frac{3}{2}\right)^{1+\log_2 n} - 1}{\frac{3}{2} - 1} = 2n\left(\left(\frac{3}{2}\right)^{\log_2 n}\frac{3}{2} - 1\right) = 2n\left(n^{\log_2 \frac{3}{2}}\frac{3}{2} - 1\right) = 3n^{\log_2 3} - 2n$$

`mult2` : arbre des appels récursifs



Complexité : $3n^{\log_2 3} - 2n \in O\left(n^{1,58}\right)$ car $\log_2 3 = 1,58$

$O(n^{1,58}) << O(n^2)$



Grâce à **`mult2`**, on a donc un algorithme de complexité $O(n^{1.58})$ pour compter les lapins !

Bref : diviser pour régner

DIVISER le problème en *a* sous-pbs de taille n/b

RESOUDRE les sous-problèmes récursivement

FUSIONNER les réponses aux sous-pbs en $O(n^d)$ afin d'obtenir la réponse au problème de départ

A partir de la connaissance des valeurs des paramètres a, b et d, le théorème maître permet de déterminer « automatiquement » la complexité d'une méthode de type diviser pour régner.

Théorème maître

- Les algorithmes de type « diviser pour régner » résolvent *a* sous-pbs de taille *n/b* et combinent ensuite ces réponses en un temps $O(n^d)$, pour *a,b,d > 0*
- Leur temps d'exécution *T(n)* peut donc s'écrire :

$$T(n) = aT(n/b) + O(n^d)$$

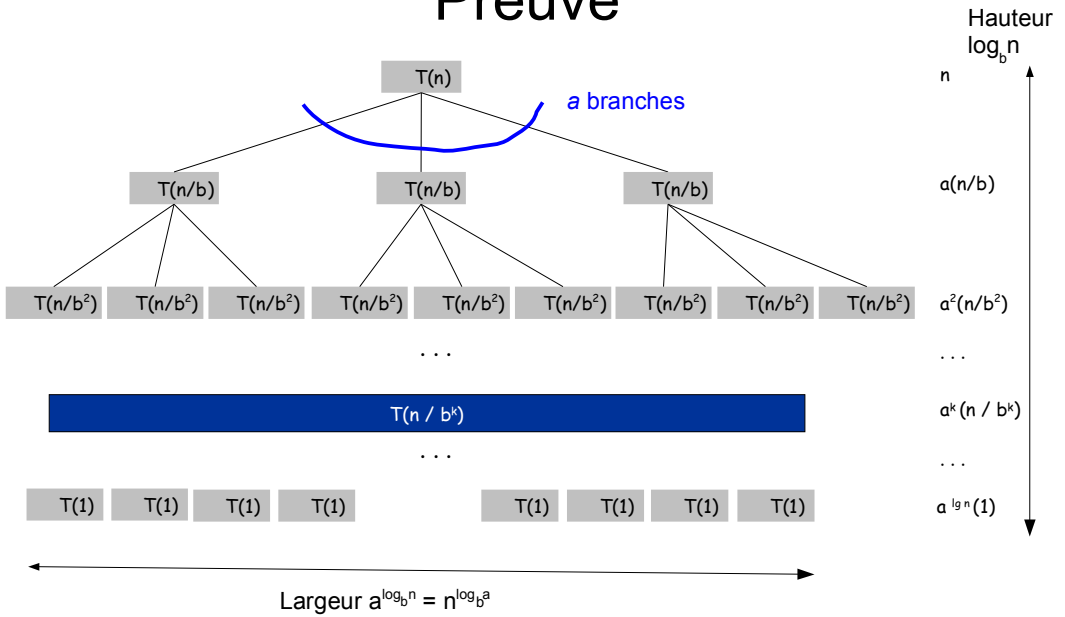
(« n/b » signifiant ici partie entière inférieure ou supérieure de n/b)

- Le terme général est alors :

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

Ce théorème permet de déterminer la complexité de la plupart des algorithmes de type « diviser pour régner ».

Preuve



Le niveau k est composé de a^k sous-problèmes, chacun de taille n/b^k

Preuve

- Sans perte de généralité, on suppose que n est une puissance de b.
- Le niveau k est composé de a^k sous-problèmes, chacun de taille n/b^k. Le travail total réalisé à ce niveau est :

$$a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k$$

- Comme k varie de 0 (racine) à $\log_b n$ (feuilles), ces nombres forment une suite géométrique de raison a/b^d
- Trois cas :
  - La raison est inférieure à 1 : la suite est décroissante et la somme des termes est du même ordre de grandeur que le premier terme, soit $O(n^d)$
  - La raison est supérieure à 1 : la suite est croissante et la somme des termes est du même ordre de grandeur que le dernier terme, soit $O(n^{\log_b a})$:

$$n^d(\frac{a}{b^d})^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d} \right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}$$

  - La raison est exactement 1 : dans ce cas les O(log n) termes de la suite sont égaux à $O(n^d)$

Le théorème « marche » bien

- Pour $T(n) = aT(n/b) + O(n^d)$, le terme général est :

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

- Algorithme **`mult`** : $T(n) = 4T(n/2) + O(n)$

  $a=4, b=2, d=1$ et donc $1<2=\log_2 4 \rightarrow O(n^{\log 4})=O(n^2)$
- Algorithme **`mult2`** : $T(n) = 3T(n/2) + O(n)$

  $a=3, b=2, d=1$ et donc $1<1{,}58\approx\log_2 4 \rightarrow O(n^{\log 4})=O(n^{1,58})$
- Algorithme TRI_FUSION : $T(n) = 2T(n/2) + O(n)$

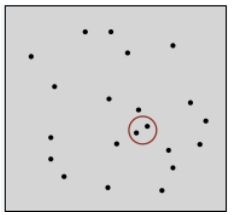
  $a=2, b=2, d=1$ et donc $1=1=\log_2 2 \rightarrow O(n \log n)$

Paire de points les plus proches

Etant donné n points dans un plan, trouver une paire de points les plus proches au sens de la distance euclidienne

Applications : vision, systèmes d'informations géographiques, contrôle aérien, modélisation moléculaire...

Algorithme naïf : tester toutes les paires de points en $\Theta(n^2)$

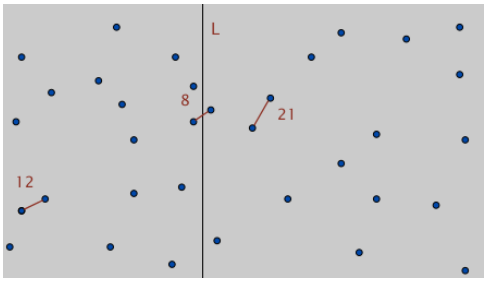


Paire de points les plus proches

Diviser Tracer une droite verticale L de façon à obtenir n/2 points dans chaque sous-région

Régner Trouver la paire de points les plus proches dans chaque sous-région

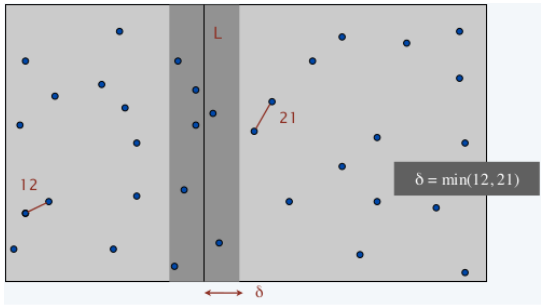
Combiner Trouver la paire de points les plus proches avec un point dans chaque sous-région
Retourner la meilleure des trois solutions trouvées



Paire de points les plus proches

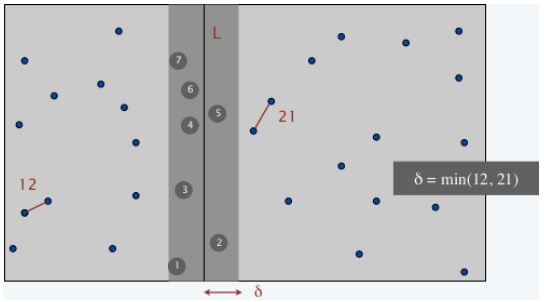
Comment trouver efficacement la paire de points les plus proches avec un point dans chaque sous-région ?

Remarque On peut se contenter d'examiner seulement les points de distance ≤ δ de la droite L, où $\delta = \min(d_{min}(regG), d_{min}(regD))$



Paire de points les plus proches

Trier les points dans la bande de largeur de largeur 2δ selon leurs ordonnées



Observation examen des distances que pour 8 positions de la liste triée

Paire de points les plus proches

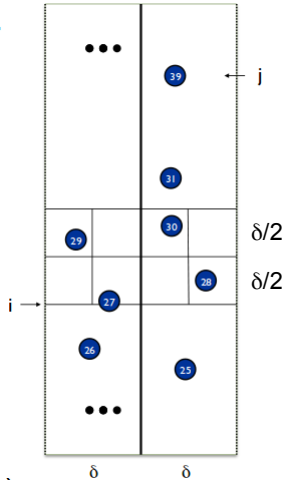
Soit s_i le point de la bande de largeur 2δ de i^e plus petite ordonnée

Propriété Si $|i-j| \geq 8$, alors la distance entre s_i et s_j est $\geq \delta$

Preuve (arguments)

1 seul point dans chaque carré

Deux points séparés par au moins deux bandes horizontales sont distants d'au moins $2(\delta/2)$



Paire de points les plus proches

PairePlusProche(p_1,…,p_n)

Si n ≤ 3 alors retourner distmin entre les points (∞ si n=1)
Sinon calculer la ligne de séparation L

1. $d_{min}(regG) \leftarrow$ PairePlusProche(points regG)
2. $d_{min}(regD) \leftarrow$ PairePlusProche(points regD)
3. $\delta = \min(d_{min}(regG), d_{min}(regD))$
4. Supprimer tous les points à distance supérieure à δ de L
5. Examiner les points dans l'ordre de leurs ordonnées croissantes et comparer la distance entre le point courant et les 8 suivants. Si une de ces distances est ≤ δ, mettre à jour δ
6. Retourner (δ)

Paire de points les plus proches

On dispose de deux listes des points triés par ordre d'abscisses croissantes et d'ordonnées croissantes (calculées une fois pour toute en O(n log n))

PairePlusProche($p_1,\ldots,p_n$)

Si $n \leq 3$ alors retourner distmin entre les points (∞ si n=1)
Sinon calculer la ligne de séparation L

1. d_{min}(regG) ← PairePlusProche(points regG)
2. d_{min}(regD) ← PairePlusProche(points regD)

← 2T(n/2)

3. $\delta = \min(d_{min}(regG), d_{min}(regD))$
4. Supprimer tous les points à distance supérieure à δ de L
5. Examiner les points dans l'ordre de leurs ordonnées croissantes et comparer la distance entre le point courant et les 8 suivants. Si une de ces distances est $\leq \delta$, mettre à jour δ

← O(n)

6. Retourner (δ)

T(n) fonction complexité pire-cas de PairePlusProche($p_1,\ldots,p_n$)

Paire de points les plus proches

$T(1)=O(1)$;
$T(n)=T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+O(n)$

La complexité de l'algorithme diviser pour régner pour le calcul d'une paire de points les plus proches est $O(n \log n)$