LICENCE D'INFORMATIQUE

Sorbonne Université

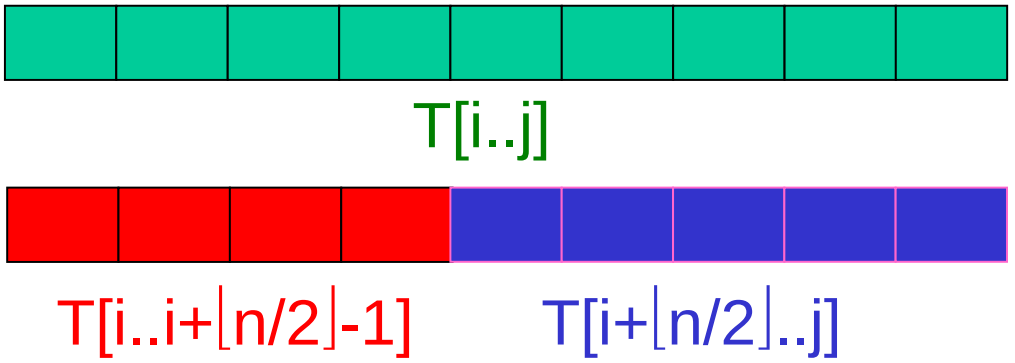
LU3IN003 – Algorithmique
Cours 2 : Programmation récursive I
Diviser pour régner

Année 2024-2025

Responsables et chargés de cours
Fanny Pascual
Olivier Spanjaard

Un premier exemple : le tri fusion

Division de T[i..j] en 2 sous-tableaux (j-i+1=n)

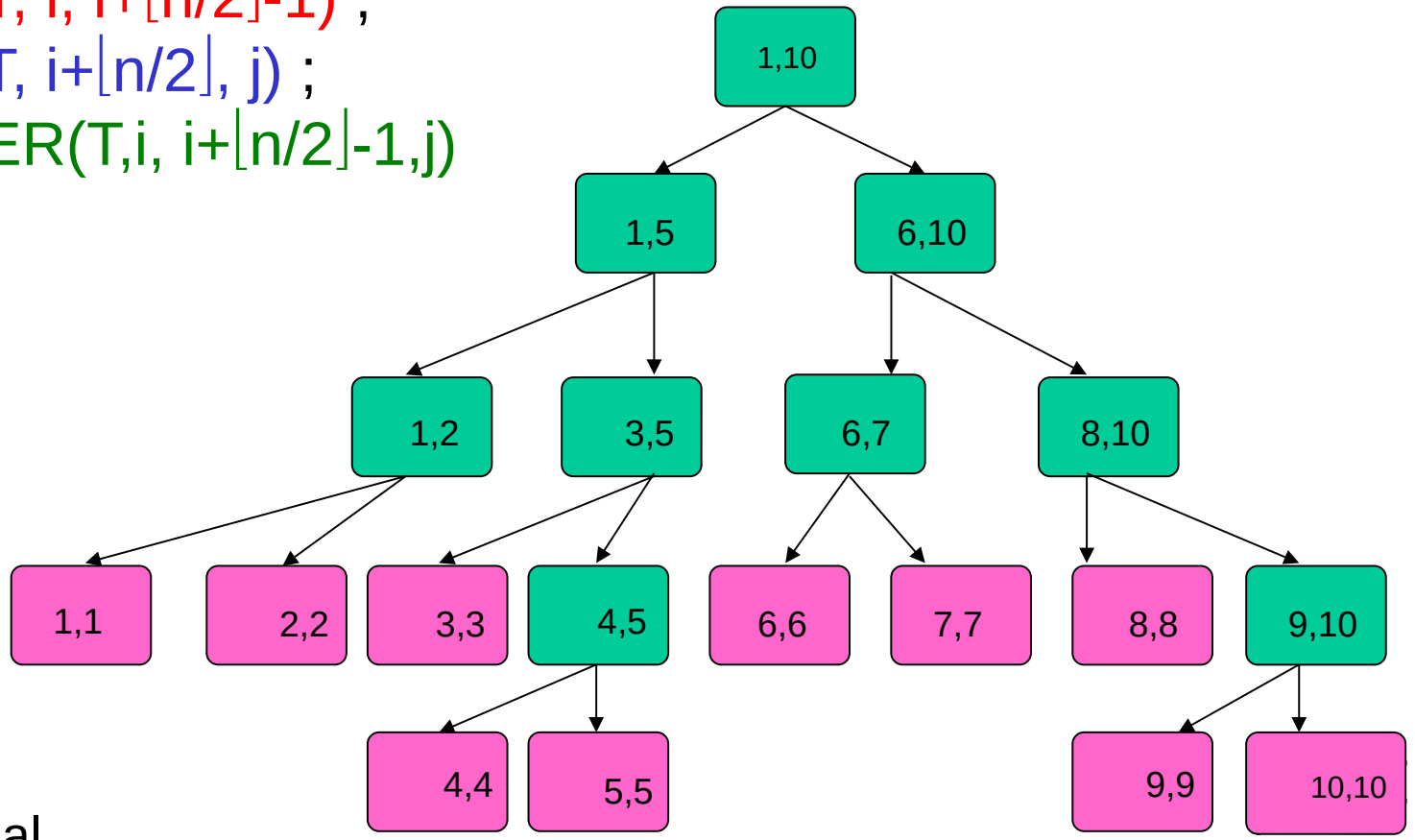


Appel récursif de TRI_FUSION(T, i, i+⌊n/2⌋-1) ;

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

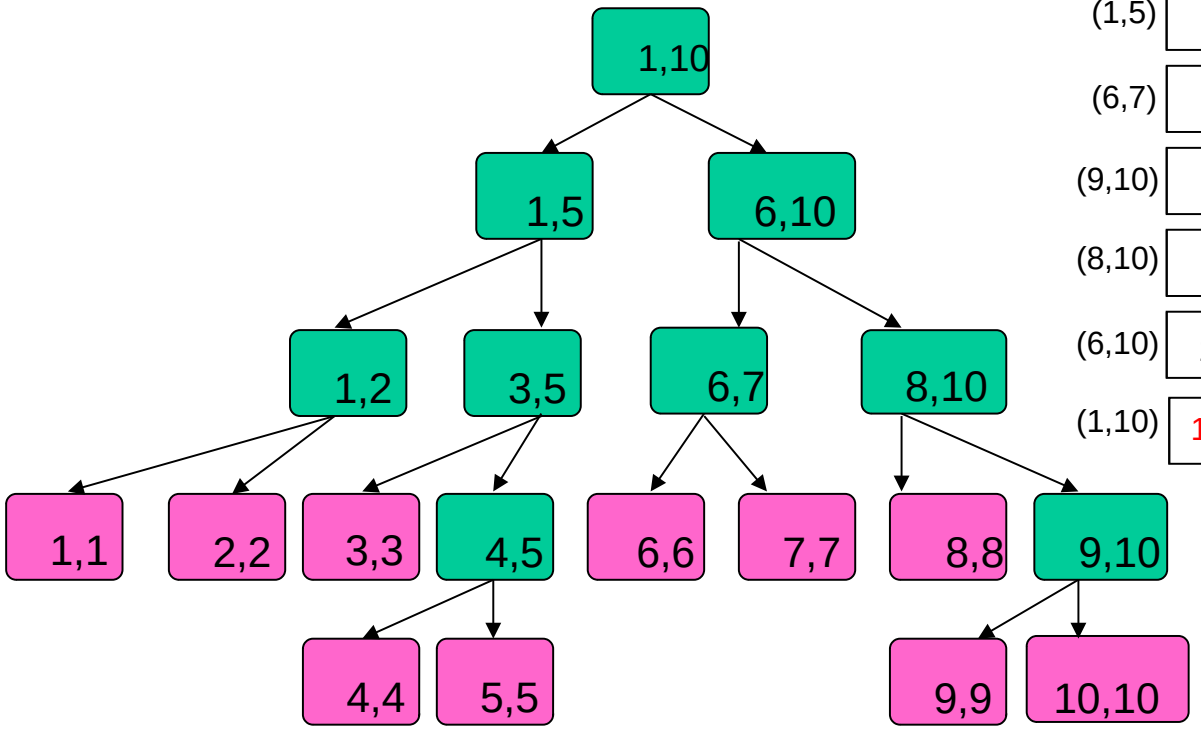
Construction de la solution par un algorithme d'interclassement de T[i..i+⌊n/2⌋-1] et T[i+⌊n/2⌋..j].

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.



i , i appel terminal

i , j 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+$\lfloor n/2 \rfloor$-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 ;

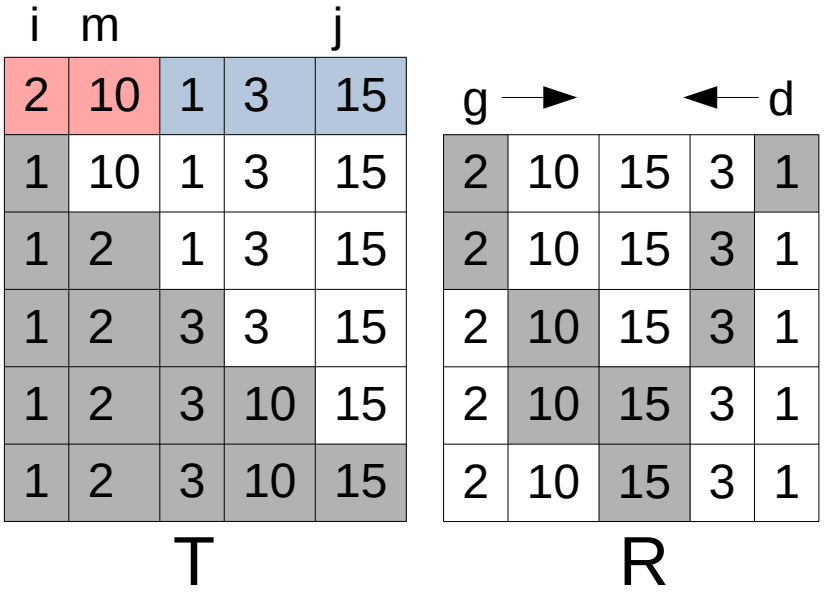
D'après l'induction ($\lceil n/2 \rceil < n$), après l'exécution de TRI_FUSION(T, i+⌊n/2⌋, j), T[i+⌊n/2⌋..j] est trié ;

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

Complexité du tri fusion

La complexité du tri fusion dépend en particulier de la complexité de l'algorithme d'interclassement.

```
Procédure INTERCLASSER(T, i, m, j)
Pour k de i à m faire R[k]:=T[k]
Pour k de m+1 à j faire R[k]:=T[j+m+1-k]
g:=i ; d:=j
Pour k de i à j faire
      Si R[g]<R[d]
            alors T[k]:=R[g] ; g:=g+1
            sinon T[k]:=R[d] ; d:=d-1
```



La complexité de la procédure d'interclassement est en O(j-i+1), autrement dit en O(n).

Pour en déduire la complexité du tri fusion proprement dit, on va se servir du **théorème maître**. Avant cela, on va revenir sur la calcul du $n^{ème}$ terme de la suite de Fibonacci.

Quiz : Fonction récursive

Que calcule la fonction récursive Fun ci-dessous ?

```
Fun(x,y)
    si y=0 alors
        retourner 1
    sinon
        si y%2=0 alors
            retourner Fun(x,y//2)*Fun(x,y//2)
        sinon
            retourner x*Fun(x,y//2)*Fun(x,y//2)
```

A) $x*y$
B) $x+y$
C) 2^{x+y}
D) x^y

Rappel de la séance précédente

- On s'est intéressé au 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 $F_0=1$ 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 :

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

- Soit *A* une matrice.
  On peut élever *A* à la puissance *n* à l'aide de la relation de récurrence suivante (en calculant une seule fois $A^{n/2}$ à chaque appel récursif) :

$$A^n = \begin{cases} A & \text{si } n = 1 \\ A^{n/2} * A^{n/2} & \text{si } n \text{ pair} \\ A^{n/2} * A^{n/2} * A & \text{si } n \text{ impair} \end{cases}$$

  Le nombre d'appel récursif pour arriver au cas de base n=1 est en $\Theta(\log_2 n)$.

Analyse de la complexité de **`Fib3`**

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} \quad \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix} \quad \begin{pmatrix} 3 & 5 \\ 5 & 8 \end{pmatrix}$$

- A chaque itération, le produit $A^{n/2} * A^{n/2}$ requiert :
  - 4 additions scalaires,
  - 8 multiplications scalaires.

  } On va s'intéresser aux nombres d'opérations élémentaires induites par les multiplications

- Initialement, chaque composante de la matrice *A* tient sur $1 = 2^0$ bit (la matrice est constituée de 0 et 1).
- Lors du k^e produit de matrices, il tient sur 2^k bits (les longueurs *sont doublées* à chaque produit).

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.

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

Non, car il existe peut-être un algorithme de meilleure complexité pour multiplier deux entiers de n bits !

Diviser pour régner

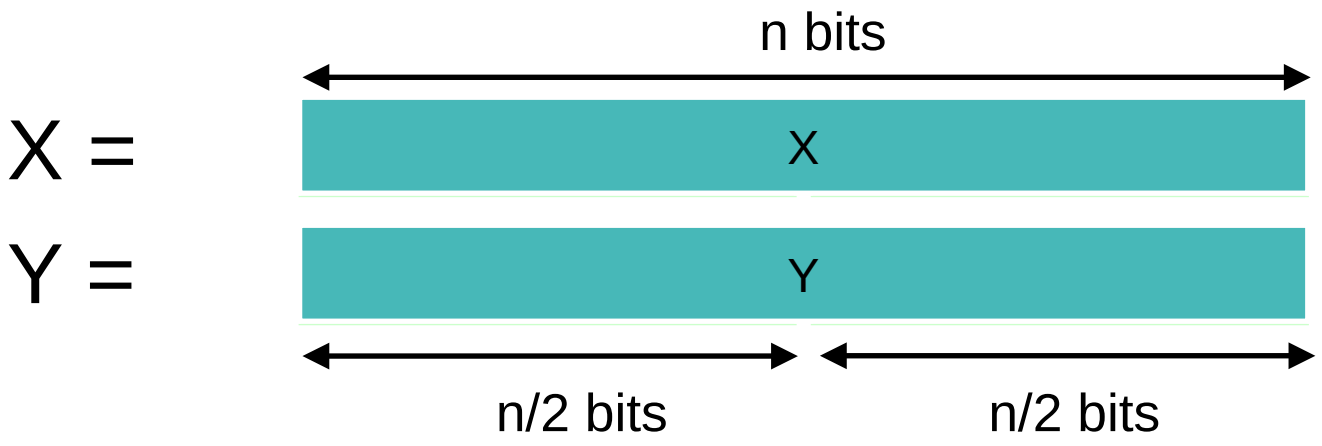
Pour concevoir des algorithmes plus rapides :

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

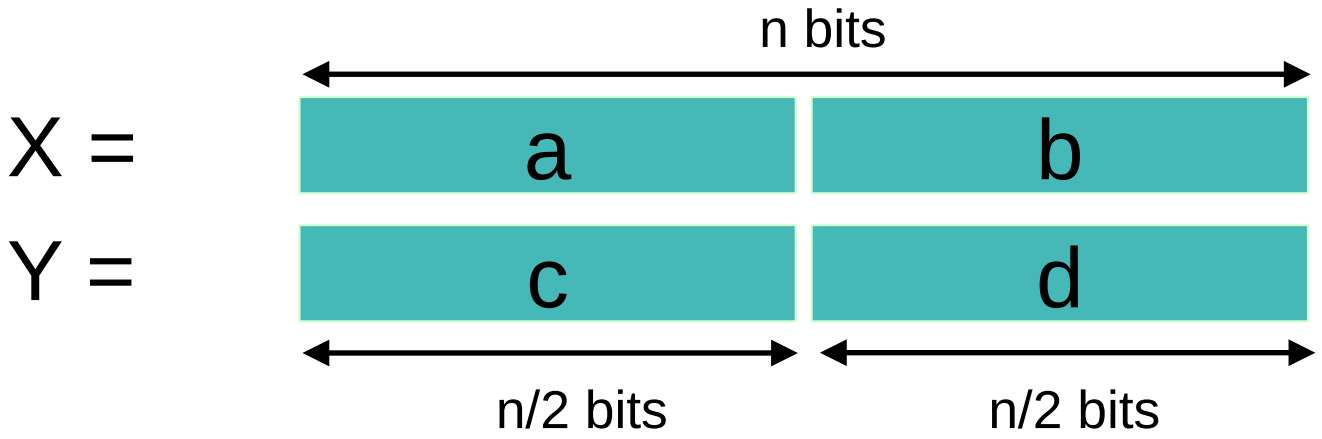


$$X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$$

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

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

Application à la multiplication

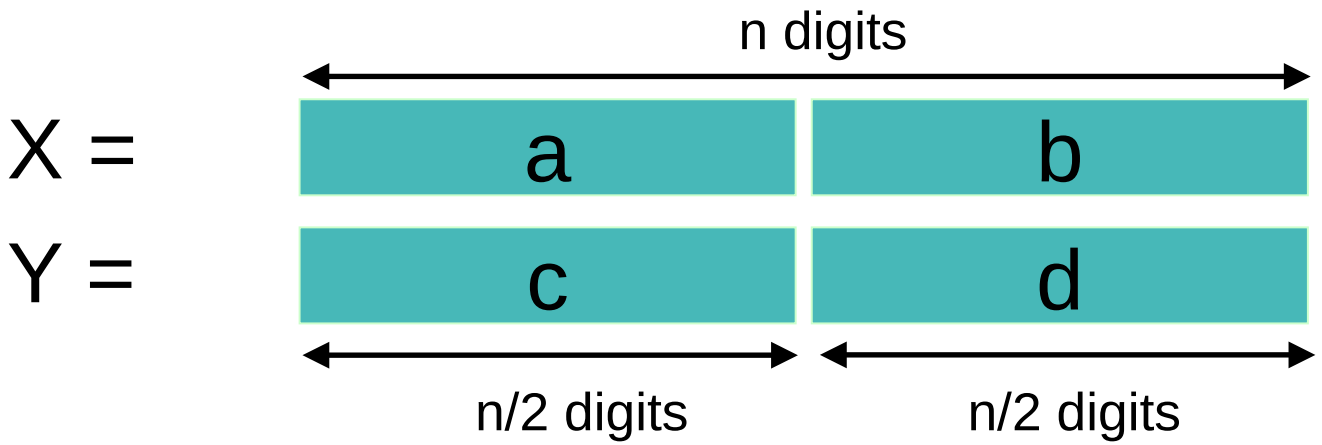


$$X \times Y = ac\ 2^{n} + (ad + bc)\ 2^{n/2} + bd$$

```
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



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

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

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

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

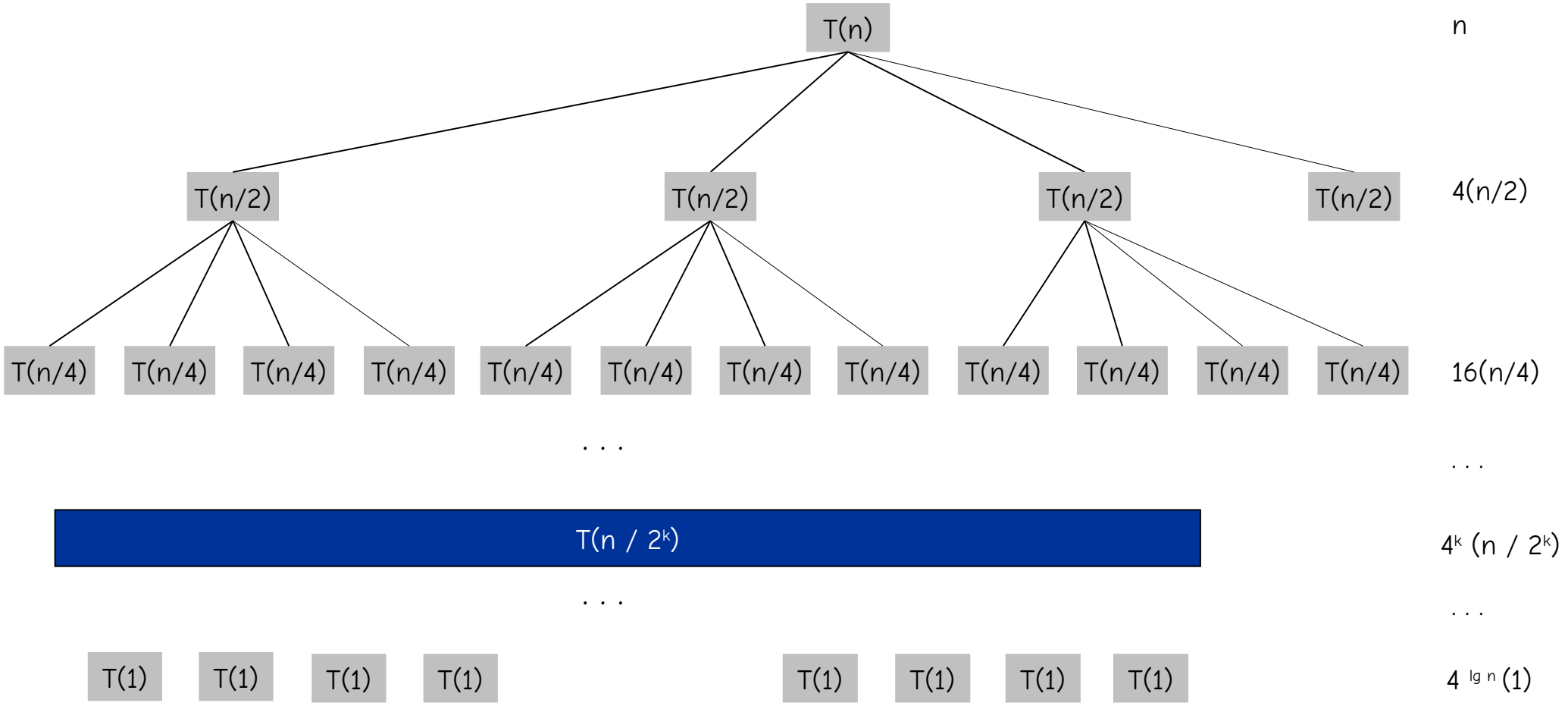
= 264126842539128

X = | a | b |

Y = | c | d |

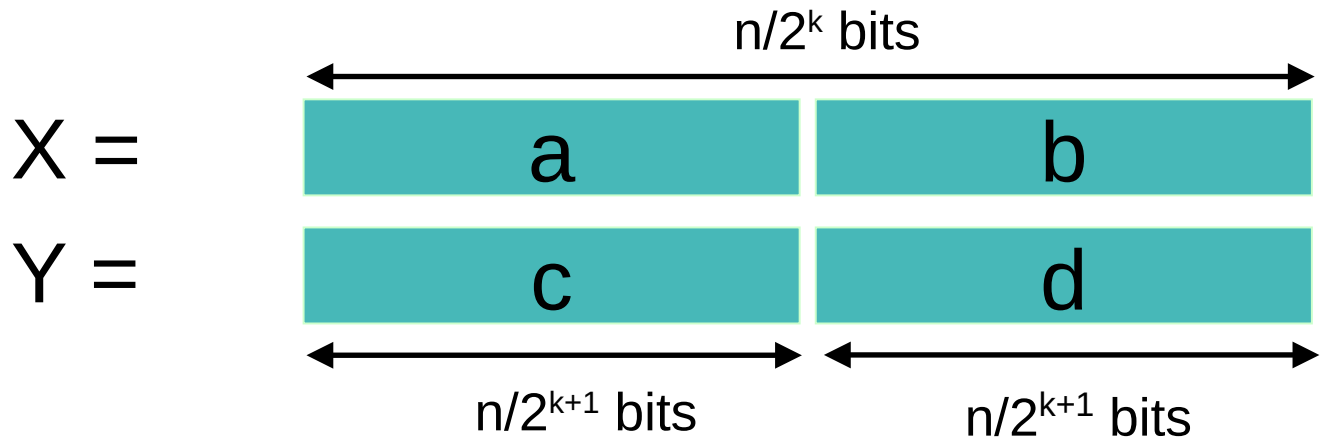
$X \times Y = ac\ 10^n + (ad + bc)\ 10^{n/2} + bd$

Arbre des appels récursifs



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

Calculs en un nœud $T(n/2^k)$

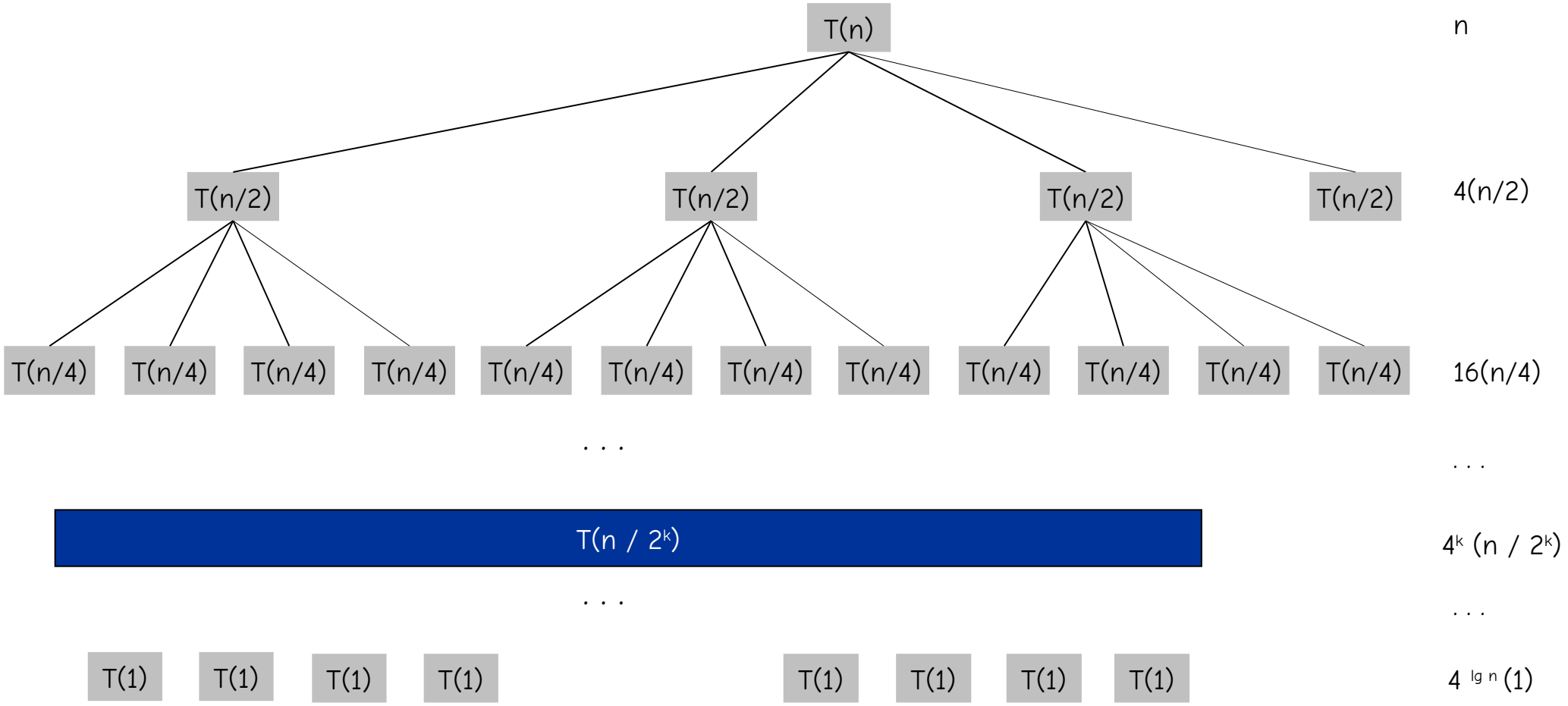


$$X \times Y = ac\ 2^{n/2^k} + (ad + bc)\ 2^{n/2^{k+1}} + bd$$

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 :

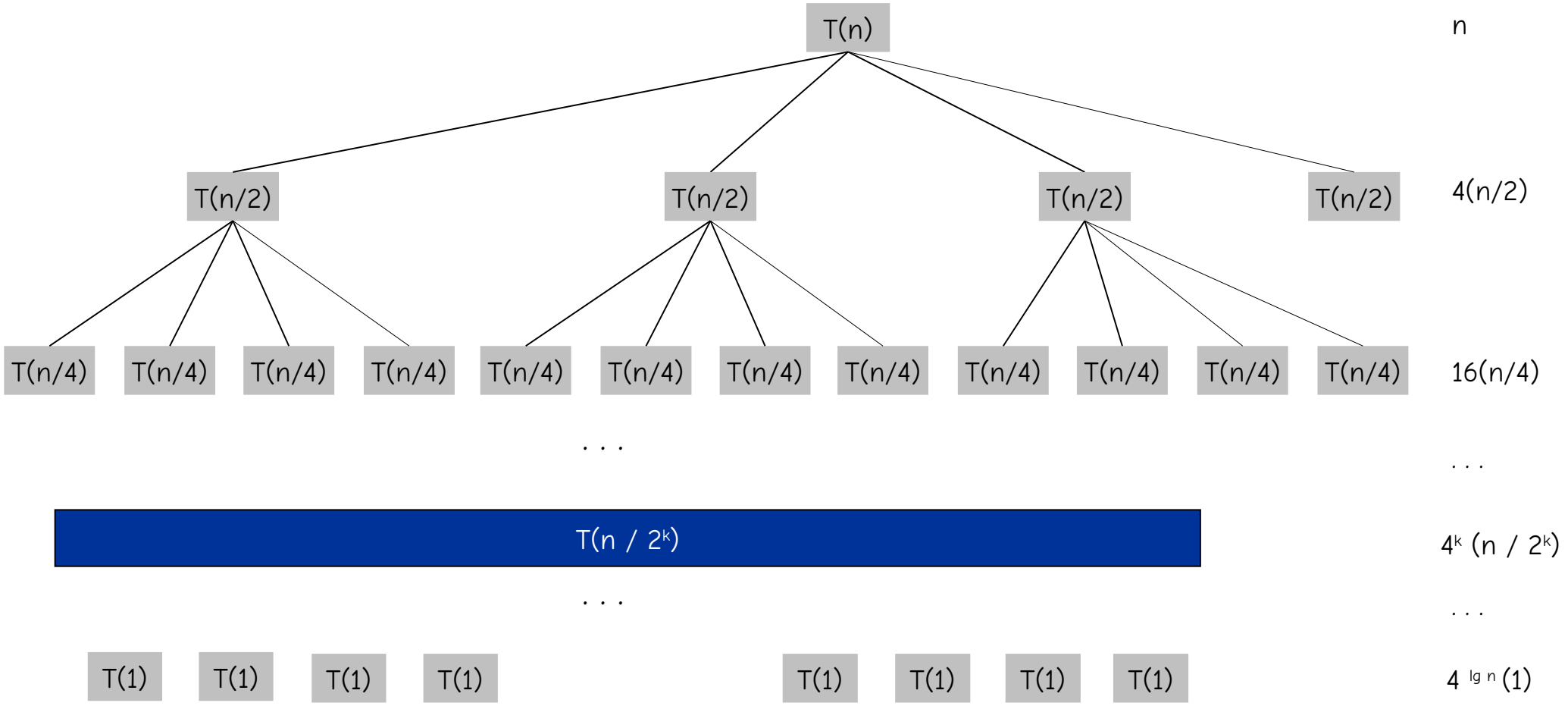
$O(n/2^k)$ opérations élémentaires

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 $O(n^2)$

Quiz : Diviser pour régner

Laquelle de ces étapes ne fait pas partie de la stratégie diviser pour régner ?

A) Diviser le problème en sous-problèmes.
B) Trier les sous-problèmes.
C) Résoudre les sous-problèmes.
D) Fusionner les résultats.

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 ?

Peut-on 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€ : $P_4 = ac$

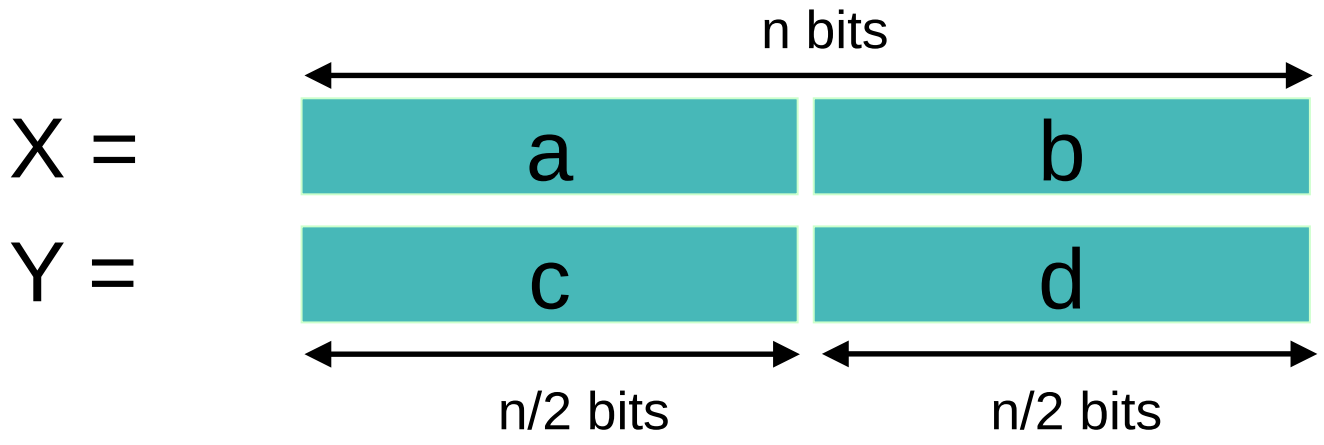
1€ : $P_5 = bd$

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)

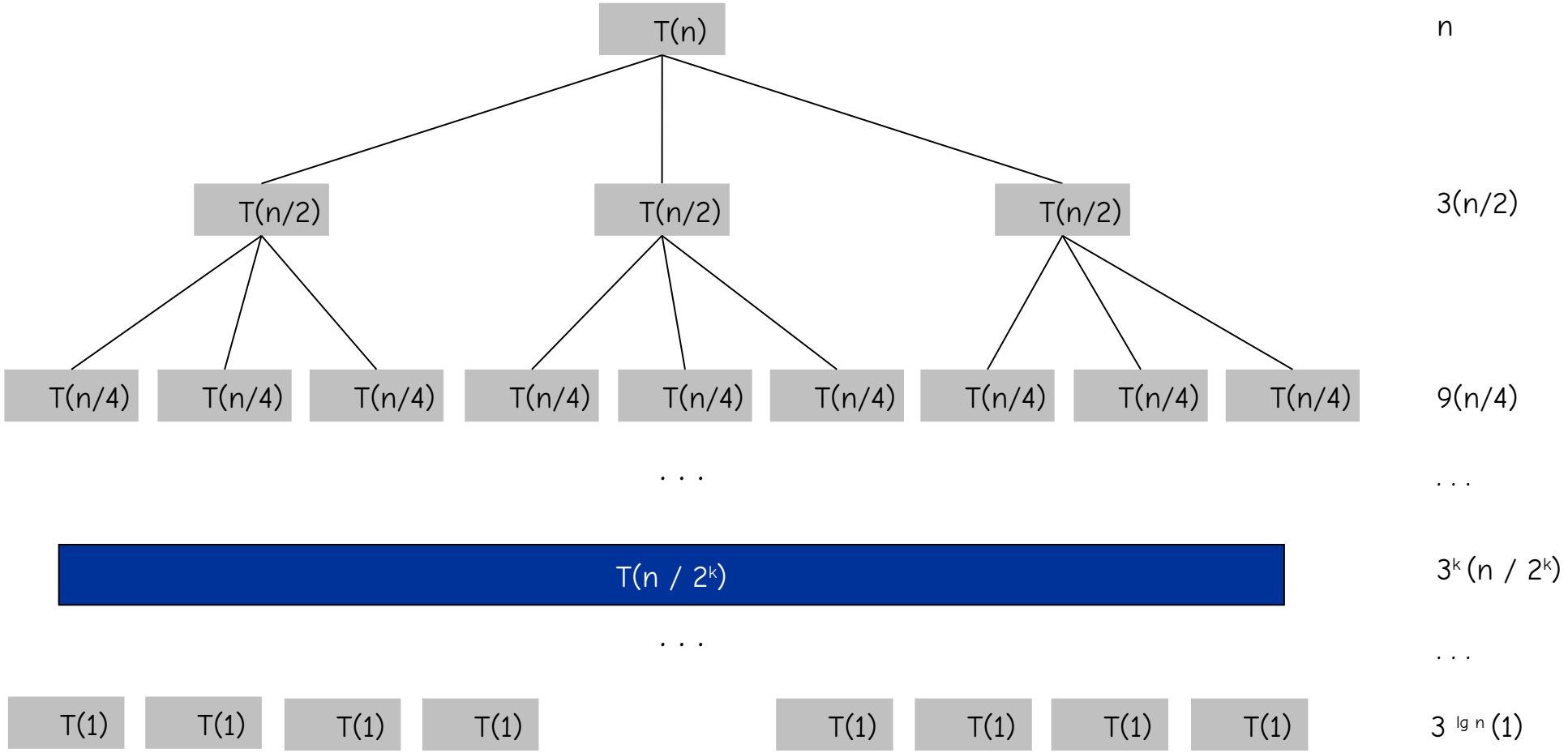


$$X \times Y = ac\ 2^{n} + (ad + bc)\ 2^{n/2} + bd$$

Anatolii Alexeevich Karatsuba, 1937-2008

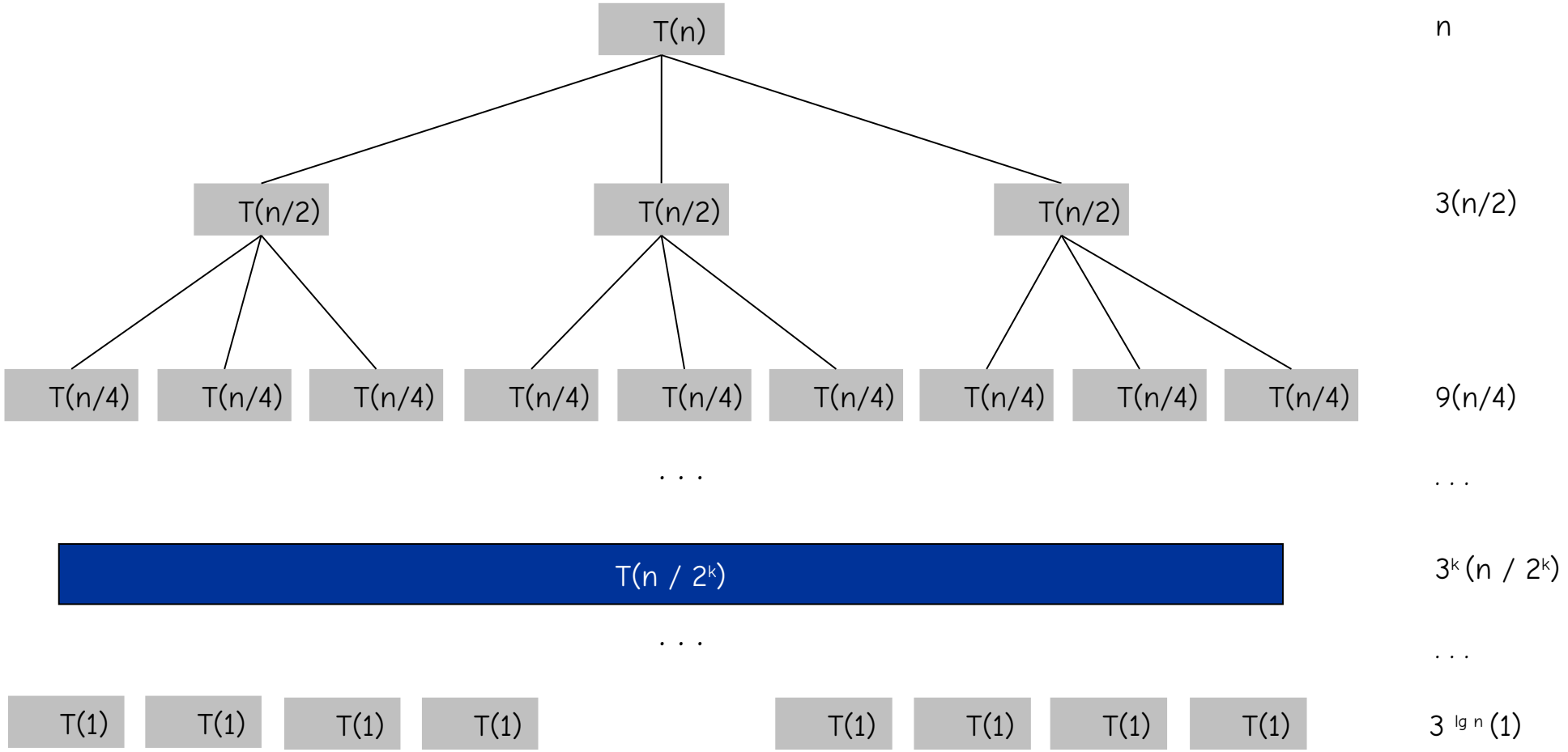
```
fonction mult2(x,y)
Entrée: Deux entiers x et y sur n bits
Sortie: Leur produit
si n = 1 retourner xy
sinon
    partitionner x en a,b et y en c,d
    P3=mult2(a+b,c+d); P4=mult2(a,c); P5=mult2(b,d)
    retourner P4.2^n + (P3-P4-P5).2^(n/2) + P5
```

mult2 : arbre des appels récursifs



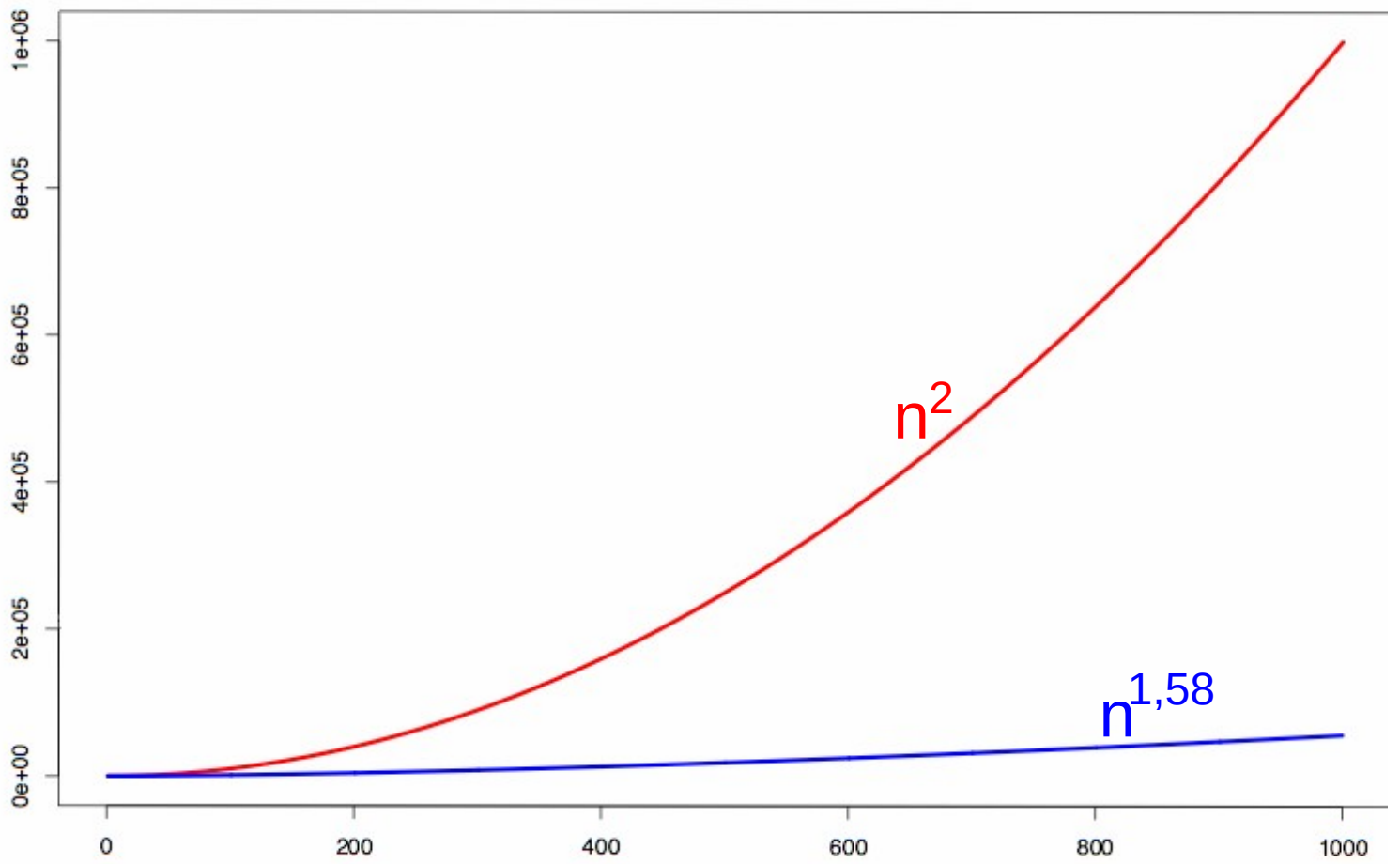
$$T(n)=\sum_{k=0}^{\log_2 n} n\left(\frac{3}{2}\right)^k = n\frac{\left(\frac{3}{2}\right)^{1+\log_2 n}-1}{\frac{3}{2}-1} = 2n\left(\left(\frac{3}{2}\right)^{\log_2 n}\frac{3}{2}-1\right) = 2n\left(n^{\log_2 \frac{3}{2}}\frac{3}{2}-1\right) = 3n^{\log_2 3}-2n$$

`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 calculer F_n !

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 :

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

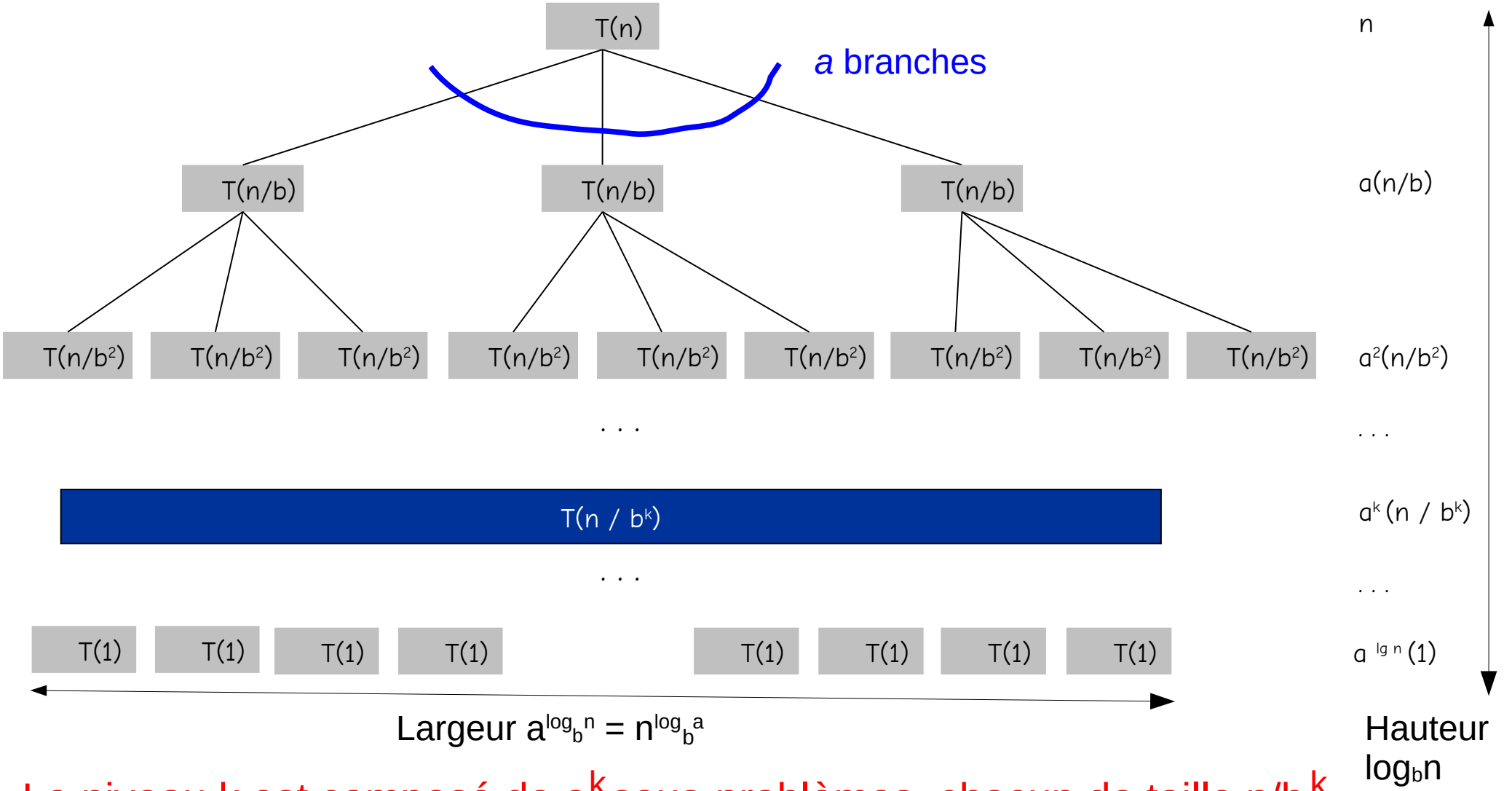
(« n/b » signifiant ici partie entière inférieure ou supérieure de n/b)

- Le terme général est alors :

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

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 :

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

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

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

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

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

Quiz : Théorème maître

On considère un algorithme de complexité :

$$T(n) = 2*T(n/3) + T(n/2) + O(n)$$

Peut-on utiliser le théorème maître pour calculer T(n) ?

A) Oui
B) Non
C) Ca dépend du problème

Quiz : Diviser pour régner

En calculant la complexité d'un algorithme diviser pour régner, on aboutit à la formule $T(n) = T(n/2)+\Theta(1)$ et $T(0) = 1$. Quelle est la complexité $T(n)$ de cet algorithme ?

A) $\Theta(\log n)$
B) $\Theta(n)$
C) $\Theta(n^2)$
D) $\Theta(2^n)$