Récursivité[1] : Algorithmes et Complexité

Salem BENFERHAT

Centre de Recherche en Informatique de Lens (CRIL-CNRS)
email : benferhat@cril.fr

[1] Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Complexité des fonctions récursives : récurrence

Première approche

- Essayer de deviner une forme générale de la complexité temporelle d'une fonction
  - Par exemple, on calcule le coût de la fonction pour un certain nombre de valeurs des arguments.
  - On peut prendre des variables additionnelles qui sont incrémentées après chaque instruction élementaire. Ou bien prendre des variables qui calculent le nombre d'appels récursifs.
- Le vérifier pour des arguments quelconques (par récurrence)

Complexité : Tour de Hanoï

Nombre d'instructions

Dans la fonction de Hanoï :

- il y a une instruction élémentaire (dans chacun des cas du conditionnel) qui est une impression, plus
- deux appels récursifs (dans la version de base).

Nous allons donc compter le nombre d'appels récursifs grâce à une variable "nb_appels_recursifs" placée (et incrémentée) en première instruction d'une fonction récursive

Complexité : Tour de Hanoï

```
void hanoi(int nb_disques, dep,  intermidiare, dest)
{

 nb_appels_recursif ++;

 if(nb_disques==1)
  printf("Deplacer un disque de %d vers %d\n", dep, dest);
 else
 {
  hanoi(nb_disques-1,dep,dest,intermidiare);
  printf("Deplacer un disque de %d vers %d\n",dep,dest);
  hanoi(nb_disques-1,intermidiare,dep,dest);
 }
}
```

Complexité : Tour de Hanoï

nombre de disques (n)	Nombre d'appels récursifs T(n)
1	1
2	3
3	7
4	15
5	31
6	63
7	127
8	255

Exercice

Quelle est la relation entre T(n) et n?

Complexité : Tour de Hanoï

nombre de disques (n)	Nombre d'appels récursifs T(n)
1	1
2	3
3	7
4	15
5	31
6	63
7	127
8	255

Exercice

On remarque que :

$$T(n) = 2^n - 1.$$

Complexité : Tour de Hanoï

Remarques

- Calculer le nombre d'appels récursifs (ou d'instructions élémentaires) permet, dans certains cas, d'obtenir une idée sur la complexité temporelle d'une fonction.

Complexité : Tour de Hanoï

Remarques

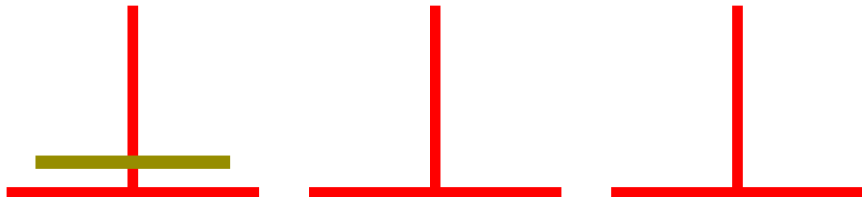
- Calculer le nombre d'appels récursifs (ou d'instructions élémentaires) permet, dans certains cas, d'obtenir une idée sur la complexité temporelle d'une fonction.
- Ca reste clairement insuffisant.

Complexité : Tour de Hanoï

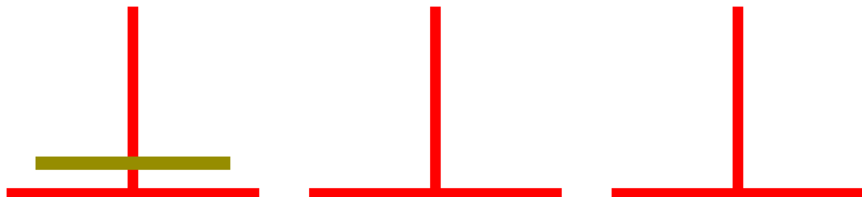
Remarques

- Calculer le nombre d'appels récursifs (ou d'instructions élémentaires) permet, dans certains cas, d'obtenir une idée sur la complexité temporelle d'une fonction.
- Ca reste clairement insuffisant.
- Il est important de regarder de près l'algorithme et déterminer l'équation qui relie T(n) en fonction de T(n-1), T(n-2) etc

Regardons de près l'algorithme de Hanoï N=1

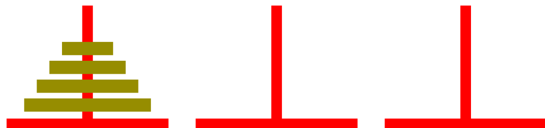


Regardons de près l'algorithme de Hanoï N=1

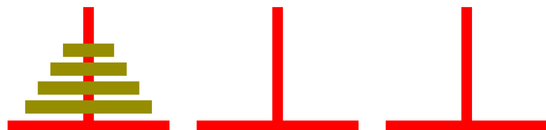


Déplacer le disque de 1 vers 3

Regardons de près l’algorithme de Hanoï (cas général)

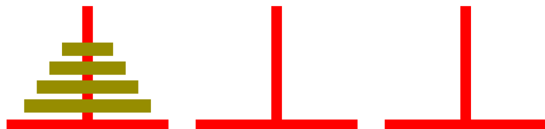


Regardons de près l'algorithme de Hanoï (cas général)

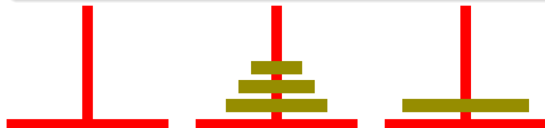


Déplacer $n-1$ disques de 1 vers 2 , puis 1 disque de 1 vers 3

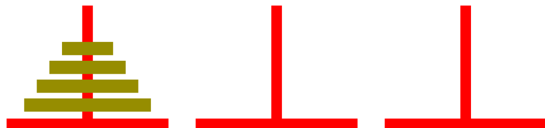
Regardons de près l'algorithme de Hanoï (cas général)



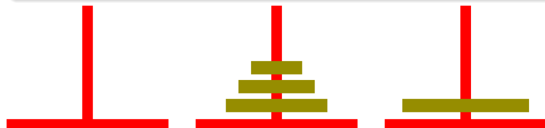
Déplacer $n-1$ disques de 1 vers 2 , puis 1 disque de 1 vers 3



Regardons de près l'algorithme de Hanoï (cas général)

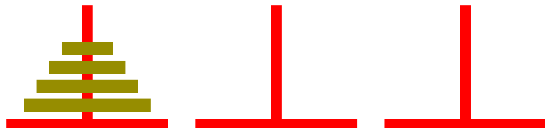


Déplacer $n - 1$ disques de 1 vers 2 , puis 1 disque de 1 vers 3

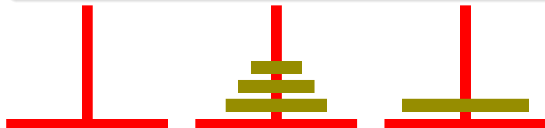


Déplacer $n - 1$ disques de 2 vers 3

Regardons de près l'algorithme de Hanoï (cas général)



Déplacer $n-1$ disques de 1 vers 2 , puis 1 disque de 1 vers 3



Déplacer $n-1$ disques de 2 vers 3

Regardons de près l’algorithme de Hanoï

Pour résumer

- $T(1) = 1$

Regardons de près l'algorithme de Hanoï

Pour résumer

- $T(1) = 1$
- Pour N différent de 1, il y a trois étapes consécutives :
  - Déplacer $N-1$ premiers disques de 1 vers 2, avec un complexité de $T(n-1)$

Regardons de près l'algorithme de Hanoï

Pour résumer

- $T(1) = 1$
- Pour N différent de 1, il y a trois étapes consécutives :
  - Déplacer $N-1$ premiers disques de 1 vers 2, avec un complexité de $T(n-1)$
  - Déplacer un disque de 1 vers 3, avec un complexité de 1

Regardons de près l'algorithme de Hanoï

Pour résumer

- $T(1) = 1$
- Pour N différent de 1, il y a trois étapes consécutives :
  - Déplacer $N - 1$ premiers disques de 1 vers 2, avec un complexité de $T(n - 1)$
  - Déplacer un disque de 1 vers 3, avec un complexité de 1
  - Déplacer les $N - 1$ disques de 2 vers 3, avec un complexité de $T(n - 1)$

Regardons de près l'algorithme de Hanoï

Pour résumer

- $T(1) = 1$
- Pour N différent de 1, il y a trois étapes consécutives :
  - Déplacer $N-1$ premiers disques de 1 vers 2, avec un complexité de $T(n-1)$
  - Déplacer un disque de 1 vers 3, avec un complexité de 1
  - Déplacer les $N-1$ disques de 2 vers 3, avec un complexité de $T(n-1)$
- Ce qui donne au final :

$$T(n) = 2.T(n-1) + 1.$$

Rappels : Somme d'une suite géométrique

Definition

Une suite géométrique est de la forme $\{T(0), T(1), ..., T(n)\}$ tels que :

$$T(i) = a.T(i-1) \text{ et } T(0) = b \neq 0$$

Rappels : Somme d’une suite géométrique

Definition

Une suite géométrique est de la forme $\{T(0), T(1), ..., T(n)\}$ tels que :

$$T(i) = a.T(i-1) \text{ et } T(0) = b \neq 0$$

Somme des premiers termes

$$\sum_{i=0}^{n} T(i) = T(0)\frac{1-a^{n+1}}{1-a} \quad (a \neq 1)$$

Rappels : Somme d'une suite géométrique

Exemple

$$\sum_{i=0}^{n} 2^i = T(0)\frac{1-2^{n+1}}{1-2} = 2^{n+1} - 1.$$

Rappels : Somme d'une suite géométrique

Exemple

$$\sum_{i=0}^{n} 2^i = T(0)\frac{1-2^{n+1}}{1-2} = 2^{n+1} - 1.$$

et

$$\sum_{i=0}^{n} 3^i = T(0)\frac{1-3^{n+1}}{1-3} = \frac{3^{n+1}-1}{2}.$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$T(n) = 2.T(n-1) + 1$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$\begin{aligned} T(n) &= 2.T(n-1) + 1 \\ &= 2.(2.T(n-2) + 1) + 1 \end{aligned}$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$\begin{aligned} T(n) &= 2.T(n-1)+1 \\ &= 2.(2.T(n-2)+1)+1 \\ &= 2^2.T(n-2)+(1+2) \end{aligned}$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$
\begin{aligned}
T(n) &= 2.T(n-1) + 1 \\
&= 2.(2.T(n-2) + 1) + 1 \\
&= 2^2.T(n-2) + (1+2) \\
&= 2^3.T(n-3) + (1+2+2^2)
\end{aligned}
$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$\begin{aligned}
T(n) &= 2.T(n-1)+1 \\
&= 2.(2.T(n-2)+1)+1 \\
&= 2^2.T(n-2)+(1+2) \\
&= 2^3.T(n-3)+(1+2+2^2) \\
&\quad . \\
&\quad .
\end{aligned}$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$\begin{array}{rcl} T(n) & = & 2.T(n-1)+1 \\ & = & 2.(2.T(n-2)+1)+1 \\ & = & 2^2.T(n-2)+(1+2) \\ & = & 2^3.T(n-3)+(1+2+2^2) \\ & & . \\ & & . \\ & = & 2^{n-1}.T(1)+(1+2+2^2+...+2^{n-2}) \end{array}$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$\begin{aligned}
T(n) &= 2.T(n-1)+1 \\
&= 2.(2.T(n-2)+1)+1 \\
&= 2^2.T(n-2)+(1+2) \\
&= 2^3.T(n-3)+(1+2+2^2) \\
& \quad . \\
& \quad . \\
&= 2^{n-1}.T(1)+(1+2+2^2+...+2^{n-2}) \\
&= 1+2+2^2+...+2^{n-2}+2^{n-1}
\end{aligned}$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$
\begin{aligned}
T(n) &= 2.T(n-1) + 1 \\
&= 2.(2.T(n-2) + 1) + 1 \\
&= 2^2.T(n-2) + (1 + 2) \\
&= 2^3.T(n-3) + (1 + 2 + 2^2) \\
& \quad . \\
& \quad . \\
&= 2^{n-1}.T(1) + (1 + 2 + 2^2 + \ldots + 2^{n-2}) \\
&= 1 + 2 + 2^2 + \ldots + 2^{n-2} + 2^{n-1} \\
&= 2^n - 1.
\end{aligned}
$$

Regardons de près l'algorithme de Hanoï

Avec le constat des temps d'exécution de la fonction Hanoï, on peut aussi montrer que l'hypothèse $T(n) = 2^n - 1$ est vérifiée.

Regardons de près l'algorithme de Hanoï

Avec le constat des temps d'exécution de la fonction Hanoï, on peut aussi montrer que l'hypothèse $T(n) = 2^n - 1$ est vérifiée. On le fait par récurrence. On peut facilement vérifier que l'hypothèse est vérifiée pour le cas de base $N = 1$:

$$T(1) = 1$$

Regardons de près l'algorithme de Hanoï

Avec le constat des temps d'exécution de la fonction Hanoï, on peut aussi montrer que l'hypothèse $T(n) = 2^n - 1$ est vérifiée. On le fait par récurrence. On peut facilement vérifier que l'hypothèse est vérifiée pour le cas de base $N = 1$:

$$T(1) = 1$$

Supposons que l'hypothèse est vraie pour $n_0 > 1$.

Regardons de près l'algorithme de Hanoï

Avec le constat des temps d'exécution de la fonction Hanoï, on peut aussi montrer que l'hypothèse $T(n) = 2^n - 1$ est vérifiée. On le fait par récurrence. On peut facilement vérifier que l'hypothèse est vérifiée pour le cas de base $N = 1$:

$$T(1) = 1$$

Supposons que l'hypothèse est vraie pour $n_0 > 1$.
Montrons qu'elle reste vraie pour $n_0 + 1$.

$$T(n_0 + 1) = 2T(n_0) + 1 = 2 * (2^{n_0} - 1) + 1 = 2^{n_0+1} - 1.$$

Complexité de l’algorithme de Hanoï

Remarques

Nombre d’appels récursif n’est pas égal au nombre d’instructions élémentaires. Cependant, la complexité temporelle d’une fonction récursive est souvent (pas toujours) confondu avec le nombre d’appels récursifs.

Complexité de l'algorithme de Hanoï

Remarques

Cette démarche reste intéressante pour des situations relativement simples où il est possible de deviner la forme générale $T(n)$.
Ceci n'est pas toujours le cas avec toutes les fonctions récursives

Suite de Fibonnacci

Définition

- Pour $n = 0$, nous avons $Fib(0) = 0$
- Pour $n = 1$, nous avons $Fib(1) = 1$
- Pour $n = 2$, nous avons $Fib(2) = 1$
- Pour $n > 2$, nous avons :

$$Fib(n) = Fib(n-1) + Fib(n-2).$$

Complexité : Suite de Fibonnacci

```
2  int fibonnacci(int n)
3  {
4   if (n<2) return n;
5   else  return fibonnacci(n-1)+fibonnacci(n-2);
6  }
```

Suite de Fibonnacci

n	Fibonnacci(n)	Nombre d'appels récursifs(n)
1	1	1
2	1	3
3	2	5
4	3	9
5	5	15
6	8	25
7	13	41
8	21	67
9	34	109
10	55	177
11	89	287
12	144	465
13	233	753
14	377	1219
15	610	1973

Le nombre d'appels récursifs sur certains exemples ne permet pas de deviner la forme générale du nombre d'appels récursifs en fonction de l'entrée n.

Complexité : Suite de Fibonnacci

De même, à partir de

$$T(n) = T(n-1) + T(n-2).$$

Il est difficile de deviner la forme générale de T(n).
Nous ferons appel à la résolution des suites de récurrences.

Suites récurrentes

Suites récurrentes d'ordre 1

Suites récurrentes d'ordre 1

Définition

Elles sont de la forme :

$$T(n) = a.T(n-1) + b,$$

où a et b sont des constantes.

Suites récurrentes d'ordre 1

Définition

Elles sont de la forme :

$$T(n) = a.T(n-1) + b,$$

où a et b sont des constantes.

Remarque

Dans ce cours, on se limite aux cas où b est une constante. Mais de manière générale, b est une fonction $f(n)$ qui ne dépend que de n.

Suites récurrentes d’ordre 1

Cas particuliers

Suites récurrentes d'ordre 1

Cas particulier : b=0

Elles sont de la forme :

$$T(n) = a.T(n-1).$$

Suites récurrentes d'ordre 1

Cas particulier : b=0

Elles admettent la solution de la forme :

$$T(n) = a^n . T(0).$$

Suites récurrentes d'ordre 1

Cas particulier : b=0

Elles sont de la forme :

$$T(n) = a.T(n-1).$$

Indications :

Il suffit de développer $T(n) = a.T(n-1)$.
On trouve :

$$T(n) = a.T(n-1) = a^2.T(n-2) = a^3.T(n-3) = ... = a^n.T(0).$$

Suites récurrentes d'ordre 1

Cas particulier : b=0

Elles sont de la forme :

$$T(n) = a.T(n-1).$$

Indications :

Supposons que $T(n) = a^n.T(0)$ est vrai pour un certain n_0. Montrons que c'est le cas pour $n_0 + 1$.
Nous avons :

$$T(n_0 + 1) = a.T(n_0) = a.a^{n_0}.T(0) = a^{n_0+1}.T(0).$$

Suites récurrentes d'ordre 1

Cas particulier : a=1

Elles sont de la forme :

$$T(n) = T(n-1) + b$$

Cas particulier : a=1

Elles admettent la solution de la forme :

$$T(n) = n * b + T(0).$$

Suites récurrentes d'ordre 1

Cas particulier : a=1

Les équations $T(n) = T(n-1) + b$ admettent la solution de la forme :

$$T(n) = n * b + T(0).$$

Indications

$$T(n) \quad = T(n-1) + b$$

Suites récurrentes d'ordre 1

Cas particulier : a=1

Les équations $T(n) = T(n-1) + b$ admettent la solution de la forme :

$$T(n) = n * b + T(0).$$

Indications

$$\begin{aligned} T(n) &= T(n-1) + b \\ &= T(n-2) + 2b \\ &= T(n-3) + 3b \\ &\ldots \\ &= T(0) + n * b. \end{aligned}$$

Suites récurrentes d'ordre 1 : Exemple Factorielle

Cas particulier : a=1

Les équations $T(n) = T(n-1) + b$ admettent la solution de la forme :

$$T(n) = n * b + T(0).$$

Complexité de la fonction Factorielle

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

et

$$T(0) = 0.$$

Ce qui donne :

$$T(n) = 2.n \in O(n).$$

Suites récurrentes d’ordre 1

Cas général

Suites récurrentes d'ordre 1

Cas général ($a \neq 1$, $b \neq 0$)

Elles sont de la forme :

$$T(n) = aT(n-1) + b,$$

avec $a \neq 1$ et $b \neq 0$.

Solutions

Elles admettent la solution de la forme :

$$T(n) = a^n.T(0) + \frac{b}{a-1}.(a^n - 1).$$

Suites récurrentes d'ordre 1 : Tour de Hanoï

Cas général ($a \neq 1$, $b \neq 0$)

Les équations $T(n) = aT(n-1) + b$, admettent la solution de la forme :

$$T(n) = a^n . T(0) + \frac{b}{a-1} . (a^n - 1).$$

Tour de Hanoï

Le nombre d'instructions est : $T(n) = 2.T(n-1) + 1$ et T(1)=1

$$T(n) = a^{n-1} . T(1) + \frac{b}{a-1} . (a^{n-1} - 1).$$

Ce qui donne :

$$T(n) = 2^{n-1} + (2^{n-1} - 1) = 2^n - 1 \in O(2^n).$$

Suites récurrentes d'ordre 1

Ce qu'elles permettent de résoudre

Les suites récurrentes permettent de donner la complexité calculatoires de certaines fonctions récursives comme :

- Factiorielle de n
- Tour de Hanoï : $T(n) = 2.T(n-1) + 1$.

Suites récurrentes d'ordre 1

Ce qu'elles permettent de résoudre

Les suites récurrentes permettent de donner la complexité calculatoires de certaines fonctions récursives comme :

- Factiorielle de n
- Tour de Hanoï : $T(n) = 2.T(n-1) + 1$.

Ce qu'elles ne permettent pas de résoudre

Cependant, elles ne permettent pas de résoudre la complexité de certaines fonctions récursives comme celle de Fibonnacci:

$$\forall n \geq 2, T(n) = T(n-1) + T(n-2),$$

avec les valeurs de base T(0) et T(1).

Suites récurrentes d’ordre 2

Suites récurrentes d'ordre 2

Définition

Elles sont de la forme :

$$T(n) = a.T(n-1) + b.T(n-2) + c.$$

Cas particulier

c=0. On parle de relations de récurrence linéaires homogènes d'ordre 2 :

$$T(n) = a.T(n-1) + b.T(n-2).$$

Equation caractéristique

On définit :

$$r^2 - a.r - b$$

Equation caractéristique

On définit :

$$r^2 - a.r - b$$

Equation du second degré

Dans un premier temps, il s'agit de résoudre l'équation du second degré :

$$r^2 - a.r - b = 0.$$

Equation du second degré (rappel!)

$$a.x^2 + b.x + c = 0.$$

Définition du discriminant

Le discriminant, noté Δ, est défini par :

$$\Delta = b^2 - 4ac$$

Solutions réelles

- Si $\Delta > 0$ alors l'équation admet deux solutions réelles :

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a}; \qquad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

- Si $\Delta = 0$ alors l'équation admet une seul solution :

$$x = \frac{-b}{2a}.$$

Cas de $\Delta > 0$

L'équation admet deux solutions r_1 et r_2. La suite dans ce cas admet une solution de la forme :

$$T(n) = x.r_1^n + y.r_2^n.$$

Cas de $\Delta > 0$

L'équation admet deux solutions r_1 et r_2. La suite dans ce cas admet une solution de la forme :

$$T(n) = x.r_1^n + y.r_2^n.$$

Cas de $\Delta = 0$

L'équation admet une seule solution r_1. La suite dans ce cas admet une solution de la forme :

$$T(n) = (x + y).r_1^n.$$

Dans les deux cas, x et y peuvent être déterminée par les situations de bases.

Complexité de la suite de Fibonnacci : Exercice

La suite est définie par :

$$T(n) = T(n-1) + T(n-2).$$

Complexité de la suite de Fibonnacci

L'équation caractéristique à résoudre est :

$$r^2 - r - 1 = 0.$$

Complexité de la suite de Fibonnacci

L'équation caractéristique à résoudre est :

$$r^2 - r - 1 = 0.$$

Calculer les racines?

Complexité de la suite de Fibonnacci

L'équation caractéristique à résoudre est :

$$r^2 - r - 1 = 0.$$

Calculons le discriminant :
$\Delta = (-1)^2 - 4(1)(-1) = 5 > 0$ ce qui donne deux solutions :

$$r_1 = \frac{1 + \sqrt{5}}{2} \qquad \text{et} \qquad r_2 = \frac{1 - \sqrt{5}}{2}$$

Complexité de la suite de Fibonnacci

La solution générale est égale à :

$$T(n) = x.\left(\frac{1+\sqrt{5}}{2}\right)^n + y.\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Trouver x et y avec les cas de base : T(0)=0 et T(1)=1?

Complexité de la suite de Fibonnacci

Trouvons les valeurs de x et y avec les cas de base : T(0) = 0, T(1)=1.

$$0 = x + y$$

et

$$1 = x.(\frac{1+\sqrt{5}}{2}) + y.(\frac{1-\sqrt{5}}{2})$$

Ce qui donne :

$$x = \frac{1}{\sqrt{5}} \quad y = -\frac{1}{\sqrt{5}}$$

En conclusion :

$$T(n) = \frac{1}{\sqrt{5}}(r_1^n - r_2^n).$$

Complexité de la suite de Fibonnacci

n	r_1^n	r_2^n
0	1.00000	1.000000000000000
10	122.99188	0.008130620915305
20	15127.00300	0.000066106996468
30	1860498.56625	0.000000537490928
40	228826219.85908	0.000000004370135
50	28143767399.14012	0.000000000035532
60	3461454915021.09961	0.000000000000289
70	425730853968392.56250	0.000000000000002
80	52361438894994816.00000	0.000000000000000
90	6440031906538371072.00000	0.000000000000000
100	792071643416902107136.00000	0.000000000000000

Complexité de la fonction ”Baguenaudière”

Exercice

Calculer le nombre d’appels récursifs de la fonction ”Baguenaudière”

Complexité de la fonction ”Baguenaudière”

Remarques

Les fonctions ”Remplir” et ”Vider” sont identiques si on ne regarde que le nombre d’appels réalisés. De ce fait la complexité de cette fonction récursives croisée est la même si on utilise une récursivité multiple.

Complexité de la fonction ”Remplir”

La suite est définie par :

$$T(n) = T(n-1) + 2T(n-2).$$

L’équation caractéristique à résoudre est :

$$r^2 - r - 2 = 0.$$

Complexité de la fonction ”Remplir”

Calculons le discriminant :
$\Delta = (-1)^2 - 4(1)(-2) = 9 > 0$ ce qui donne deux solutions :

$$r_1 = \frac{1+\sqrt{9}}{2} = 2 \qquad \text{et} \qquad r_2 = \frac{1-\sqrt{9}}{2} = -1$$

Complexité de la fonction ”Remplir”

La solution générale est égale à :

$$T(n) = x.(2)^n + y.(-1)^n$$

Trouvons les valeurs de x et y avec les cas de base : T(0) = 0, T(1)=1.

Complexité de la fonction ”Remplir”

Trouvons les valeurs de x et y avec les cas de base : T(0) = 0, T(1)=1.

$$0 = x + y$$

et

$$1 = x.(2) + y.(-1)$$

Ce qui donne :

$$x = \frac{1}{3} \quad y = -\frac{1}{3}$$

En conclusion :

$$T(n) = \frac{1}{3}(r_1^n - r_2^n).$$

Encore un résultat pour la complexité des fonctions récursives

Insuffisance des relations de récurrence linéaires

Les paramètres de certaines fonctions récursives n'évoluent pas toujours de manière linéaire. Certaines évoluent plus vite vers les cas de base, en passant par exemple de n vers n/b après chaque appel, où b est un réel strictement plus grand que 1.

Encore un résultat pour la complexité des fonctions récursives

Insuffisance des relations de récurrence linéaires

Les paramètres de certaines fonctions récursives n'évoluent pas toujours de manière linéaire. Certaines évoluent plus vite vers les cas de base, en passant par exemple de n vers n/b après chaque appel, où b est un réel strictement plus grand que 1.

Récurrences non linéaires

Elles sont de la forme :

$$T(n) = a.T\left(\frac{n}{b}\right) + n^c$$

avec $a \geq 1, b \geq 1, c > 0$.

Encore un résultat pour la complexité des fonctions récursives

Le théorème général concerne les équation de récurrence de la forme :

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Avec $a \geq 1$ et $b > 1$
Ici, intuitivement $T(n)$ contient le nombre d'instructions d'une fonction récursive tel que :

- a représente le nombre d'appels récursifs
- $\left(\frac{n}{b}\right)$ représente la taille des données au niveau de chaque appel récursif (c'est-à-dire la taille des sous-problèmes)
- $f(n)$ représente le nombre d'instructions qui restent dans la fonction récursive (on ne considère pas les appels récursifs).

Encore un résultat pour la complexité des fonctions récursives

Les solutions du théorème général

- Si $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ pour une constante $\epsilon > 0$ alors

$$T(n) = O\left(n^{\log_b a}\right)$$

- Si $f(n) = O\left(n^{\log_b a}\log^k n\right)$ alors

$$T(n) = O\left(n^{\log_b a}\log^{k+1} n\right)$$

- Si $f(n) = O\left(n^{\log_b(a)+\epsilon}\right)$ et que $af\left(\frac{n}{b}\right) \leq cf(n)$ pour t c ¡ 1 et n suffisamment grand, alors

$$T\left(n\right) = O\left(f\left(n\right)\right)$$

Récurrences non linéaires

Elles sont de la forme :

$$T(n) = a.T\left(\frac{n}{b}\right) + n^c$$

avec $a \geq 1, b \geq 1, c > 0$.

Récurrences non linéaires

Elles sont de la forme :

$$T(n) = a.T\left(\frac{n}{b}\right) + n^c$$

avec $a \geq 1, b \geq 1, c > 0$.

Solutions (pour info!)

Elles admettent la solution suivante :

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

La fonction puissance

```
double puissance_n(double n, int p)
{
 if (p == 0)
  return(1);
 else return(n*puissance_n(n,p-1));
}
```

La fonction puissance plus efficace

```
double puissance_rapide(double n, int p)
 {
 double resultat;
 if(p == 0)
  return(1);
 else
  if (p % 2==0)
  {
   resultat = puissance_rapide(n,p/2);
   return (resultat * resultat);
  }
  else return(n*puissance_rapide(n,p-1));
 }
```

Récursivité terminale

On dit qu'un fonction est récursive terminale, si tout appel récursif est de la forme retourner f(...) (... ne contient pas d'appels récursifs). Typiquement, une récursivité terminal contient **en pratique** un seul appel récursif.

Récursivité terminale

On dit qu'un fonction est récursive terminale, si tout appel récursif est de la forme retourner f(...) (... ne contient pas d'appels récursifs). Typiquement, une récursivité terminal contient **en pratique** un seul appel récursif.

Récursivité terminale

On dit qu'un fonction est récursive terminale, si tout appel récursif est de la forme retourner f(...) (. . . ne contient pas d'appels récursifs). Typiquement, une récursivité terminal contient **en pratique** un seul appel récursif.

Des compilateurs qui optimisent le code remplace la récursivité terminale.

Exercice : Puissance

La fonction puissance plus efficace

```
double puissance_rapide(double n, int p)
 {
 double resultat;
 if(p == 0)
  return(1);
 else
  if (p % 2==0)
  {
   resultat = puissance_rapide(n,p/2);
   return (resultat * resultat);
  }
  else return(n*puissance_rapide(n,p-1));
 }
```

Une autre écriture

```
double puissance_rapide(double n, int p)
 {
 double resultat;
 if(p == 0)
  return(1);
 else
  resultat = puissance_rapide(n,p/2);
  if (p % 2==0)
  {
   return (resultat * resultat);
  }
  else return(n*resultat * resultat);
 }
```

Complexité de la fonction puissance

$$T(p) = T(p/2) + 4$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= T(p/2) + 4 \\
&= \left(T\left(\frac{p}{2^2}\right) + 4\right) + 4
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= T(p/2) + 4 \\
&= \left(T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= T\left(\frac{p}{2^2}\right) + 2.4
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= T(p/2) + 4 \\
&= \left(T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= T\left(\frac{p}{2^2}\right) + 2.4 \\
&= T\left(\frac{p}{2^3}\right) + 3.4
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= T(p/2) + 4 \\
&= \left(T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= T\left(\frac{p}{2^2}\right) + 2.4 \\
&= T\left(\frac{p}{2^3}\right) + 3.4 \\
&\vdots
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= T(p/2) + 4 \\
&= \left(T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= T\left(\frac{p}{2^2}\right) + 2.4 \\
&= T\left(\frac{p}{2^3}\right) + 3.4 \\
&. \\
&. \\
&= T\left(\frac{p}{2^m}\right) + m.4
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= T(p/2) + 4 \\
&= \left(T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= T\left(\frac{p}{2^2}\right) + 2.4 \\
&= T\left(\frac{p}{2^3}\right) + 3.4 \\
&. \\
&. \\
&= T\left(\frac{p}{2^m}\right) + m.4
\end{aligned}
$$

Question?

Que vaut m?

Réponse

m est tel que : $T\left(\frac{p}{2^m}\right) = T(1) = 1$, c'est à dire :

$$\frac{p}{2^m} = 1.$$

C'est-à-dire : $p = 2^m$.
Appliquons le *log* base (2), ce qui donne:

$$m = log_2(p).$$

Donc : $T(p) \in O(log_2(p))$.

Exercice

Fonction puissance

Ecrire une fonction récursive qui calcule n^p où n et p sont des entiers en utilisant les propriété suivantes :

- Si $p = 0$ alors $n^p = 1$
- Si p est pair alors $n^p = n^{(p/2)} * n^{(p/2)}$
- Si p est impair alors $n^p = n * n^{(p-1)}$

La fonction puissance

```
double puissance_rapide(double n, int p)
 {
 double resultat;
 if(p == 0)
  return(1);
 else
  if (p % 2==0)
  {
   resultat = puissance_rapide(n,p/2);
   return (resultat * resultat);
  }
  else return(n*puissance_rapide(n,p-1));
 }
```

Une autre écriture

```
double puis_lente(double n, int p)
 {
 if(p == 0)
  return(1);
 else
  if (p % 2==0)
  {
   return (puis_lente(n,p/2) * puis_lente(n,p/2));
  }
  else return(n*puis_lente(n,p/2) * puis_lente(n,p/2))
 }
```

Complexité de la fonction puissance

$$T(p) = 2 * T(p/2) + 4$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * \left(2 * T\left(\frac{p}{2^2}\right) + 4\right) + 4
\end{aligned}
$$

Complexité de la fonction puissance

$$\begin{aligned} T(p) &= 2 * T(p/2) + 4 \\ &= 2 * \left(2 * T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\ &= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \end{aligned}$$

Complexité de la fonction puissance

$$\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * (2 * T\left(\frac{p}{2^2}\right) + 4) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2)
\end{aligned}$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * (2 * T\left(\frac{p}{2^2}\right) + 4) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2) \\
&.
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * (2 * T\left(\frac{p}{2^2}\right) + 4) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2) \\
&. \\
&= 2^m T\left(\frac{p}{2^m}\right) + 4(1 + 2 + 2^2 + \ldots + 2^{m-1})
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * (2 * T\left(\frac{p}{2^2}\right) + 4) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2) \\
&. \\
&= 2^m T\left(\frac{p}{2^m}\right) + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 2^m * 4 + 4(1 + 2 + 2^2 + ... + 2^{m-1})
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * \left(2 * T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2) \\
&. \\
&= 2^m T\left(\frac{p}{2^m}\right) + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 2^m * 4 + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 4(1 + 2 + 2^2 + ... + 2^{m-1} + 2^m)
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * (2 * T\left(\frac{p}{2^2}\right) + 4) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2) \\
&. \\
&= 2^m T\left(\frac{p}{2^m}\right) + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 2^m * 4 + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 4(1 + 2 + 2^2 + ... + 2^{m-1} + 2^m) \\
&= 4.(2^{m+1} - 1)
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * \left(2 * T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1+2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1+2+2^2) \\
&. \\
&= 2^m T\left(\frac{p}{2^m}\right) + 4(1+2+2^2+\ldots+2^{m-1}) \\
&= 2^m * 4 + 4(1+2+2^2+\ldots+2^{m-1}) \\
&= 4(1+2+2^2+\ldots+2^{m-1}+2^m) \\
&= 4.(2^{m+1}-1) \\
&= 8.2^m - 4
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * \left(2 * T\left(\frac{p}{2^2}\right) + 4\right) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2) \\
&. \\
&= 2^m T\left(\frac{p}{2^m}\right) + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 2^m * 4 + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 4(1 + 2 + 2^2 + ... + 2^{m-1} + 2^m) \\
&= 4.(2^{m+1} - 1) \\
&= 8.2^m - 4 \\
&= 8.2^{log_2(p)} - 4
\end{aligned}
$$

Complexité de la fonction puissance

$$
\begin{aligned}
T(p) &= 2 * T(p/2) + 4 \\
&= 2 * (2 * T\left(\frac{p}{2^2}\right) + 4) + 4 \\
&= 2^2 * T\left(\frac{p}{2^2}\right) + 4(1 + 2) \\
&= 2^3 * T\left(\frac{p}{2^3}\right) + 4(1 + 2 + 2^2) \\
&. \\
&= 2^m T\left(\frac{p}{2^m}\right) + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 2^m * 4 + 4(1 + 2 + 2^2 + ... + 2^{m-1}) \\
&= 4(1 + 2 + 2^2 + ... + 2^{m-1} + 2^m) \\
&= 4.(2^{m+1} - 1) \\
&= 8.2^m - 4 \\
&= 8.2^{log_2(p)} - 4 \\
&= 8.p - 4 \in O(p)
\end{aligned}
$$

Exercice : algorithme de tri lent

Considérons l'algorithme de tri suivant.

- On divise le tableau en trois sous-tableaux :
  T_1 d'indices $\{0, .., \frac{N}{3} - 1\}$, T_2 d'indices $\{\frac{N}{3}, .., \frac{2.N}{3} - 1\}$ et T_2 d'indices $\{\frac{2.N}{3}, .., N - 1\}$.
- On procède à trois tris (le même algorithme)
  - Trier le sous-tableau $T_1 \cup T_2$
  - Trier le sous-tableau $T_2 \cup T_3$
  - Trier le sous-tableau $T_1 \cup T_2$ (de nouveau)

  Lorsque le tableau est de taille 3, on le trie comme on le veut (en $O(1)$ bien sûr).
  - Ecrire la fonction récursive qui réalise ce tri.
  - Evaluer sa complexité.

Exercice : Tri

L'idée est de montrer que le résultat est juste par récurrence (intuitivement).
Lorsque le tableau contient moins de 3 éléments, il est facile de vérifier qu'il est trié (c'est la condition d'arrêt de la fonction récursive).

Exercice : Tri

- Supposons qu'un tableau T [0,n-1] contient n éléments et il est trié (par notre méthode). C'est-à-dire on fait l'hypothèse notre algorithme fonctionnement parfaitement pour des tailles inférieurs ou égales à n.

Exercice : Tri

- Supposons qu'un tableau T [0,n-1] contient n éléments et il est trié (par notre méthode). C'est-à-dire on fait l'hypothèse notre algorithme fonctionnement parfaitement pour des tailles inférieurs ou égales à n.
- Le but est de vérifier que T[0,n] reste trié. C'est-à-dire que notre algorithme fonctionne également pour la taille $n + 1$.

Exercice : Tri

- La première étape trie les sous-tableaux T1=[0,..,(n+1)/3-1], T2=[(n+1)/3,..,2(n+1)/3-1]. A ce niveau là, on obtient $T_1 \cup T_2$ trié, et en particulier tout élément de T_1 est plus petit que tout élément de T_2.

Exercice : Tri

- La première étape trie les sous-tableaux T1=[0,..,(n+1)/3-1], T2=[(n+1)/3,..,2(n+1)/3-1]. A ce niveau là, on obtient $T_1 \cup T_2$ trié, et en particulier tout élément de T_1 est plus petit que tout élément de T_2.
- La deuxième étape trie les sous-tableaux T2=[(n+1)/3,..,2(n+1)/3-1] et T3=[2(n+1)/3,..,n]. Après cette étape, tout élément de $T3$ est au moins aussi grand que tout élément de T2 (et de T1). Donc les élément de $T3$ sont triés et sont les plus grands éléments du tableau. Pas besoin de les tester de nouveau!

Exercice : Tri

- La première étape trie les sous-tableaux T1=[0,..,(n+1)/3-1], T2=[(n+1)/3,..,2(n+1)/3-1]. A ce niveau là, on obtient $T_1 \cup T_2$ trié, et en particulier tout élément de T_1 est plus petit que tout élément de T_2.
- La deuxième étape trie les sous-tableaux T2=[(n+1)/3,..,2(n+1)/3-1] et T3=[2(n+1)/3,..,n]. Après cette étape, tout élément de $T3$ est au moins aussi grand que tout élément de T2 (et de T1). Donc les élément de $T3$ sont triés et sont les plus grands éléments du tableau. Pas besoin de les tester de nouveau!
- Il suffit maintenant de trier T_1 et T_2 ce qui est fait en troisième et dernière étape.

Récursivité : Tri

```
int max (int a, int b)
{
 if (a>=b) return a;
  else return b;
}

int min (int a, int b)
{
 if (a<=b) return a;
 else return b;
}
```

Récursivité : Tri

```
void trier_2_els(int A[], int N, int i)
{
 int a=A[i],b=A[i+1];
 A[i]=max(a,b);
 A[i+1]=min(a,b);
}
```

Récursivité : Tri

```c
void trier_3_els(int A[], int N, int i)
{
 int a=A[i],b=A[i+1],c=A[i+2];
 if (a >= max(b,c))
 {
  A[i]=a; A[i+1]=max(b,c); A[i+2]=min(b,c);
 }
 else
 {
  if (a <=min(b,c))
  {
  A[i+2]=a; A[i]=max(b,c); A[i+1]=min(b,c);
  }
  else
  {
  A[i+1]=a; A[i]=max(b,c); A[i+2]=min(b,c);
  }
 }
}
```

Récursivité : Tri

```
void trier(int A[], int N, int i)
{
 int entier;
 if (N==2) trier_2_elets (A,N,i);
 else
 if (N==3)  trier_3_elets (A,N,i);
 else
 if (N!=1)
 {
  entier=floor(N/3);
  trier (A, 2*entier, i);
  trier (A, N-entier, i+entier);
  trier (A, 2*entier, i);
 }
}
```

Complexité de la fonction tri

$$
\begin{aligned}
T(n) &= 3.T(2/3\ n) + 4 \\
&= 3.\ (3.\ T((2/3)^2\ n) + 4) + 4 \\
&= 3^2.\ T((2/3)^2\ n) + 4.(1+3) \\
&= 3^3.\ T((2/3)^3\ n) + 4.(1+3+3^2) \\
&. \\
&. \\
&= 3^m.\ T((2/3)^m\ n) + 4.(1+3+3^2+...+3^{m-1}))
\end{aligned}
$$

Complexité de la fonction tri

$$
\begin{aligned}
T(n) &= 3.T(2/3\ n) + 4 \\
&= 3.\ (3.\ T((2/3)^2\ n) + 4) + 4 \\
&= 3^2.\ T((2/3)^2\ n) + 4.(1+3) \\
&= 3^3.\ T((2/3)^3\ n) + 4.(1+3+3^2) \\
&. \\
&. \\
&= 3^m.\ T((2/3)^m\ n) + 4.(1+3+3^2+...+3^{m-1}))
\end{aligned}
$$

Question?

Que vaut m?

Complexité de la fonction tri

Réponse

m est tel que : T($(\frac{2}{3})^m$. n) = T(1), c'est à dire :

$$\left(\frac{2}{3}\right)^m .n = 1.$$

C'est-à-dire :

$$\left(\frac{3}{2}\right)^m = n$$

Appliquons le *log* base (3/2), ce qui donne:

$$log_{(\frac{3}{2})}\left(\frac{3}{2}\right)^m = log_{(\frac{3}{2})}(n)$$

Suite

Après simplification :

$$m.log_{(\frac{3}{2})}\left(\frac{3}{2}\right) = log_{(\frac{3}{2})}(n)$$

(car $log_a(b^x) = x.log_a(b)$)
Ce qui donne :

$$m = log_{(\frac{3}{2})}(n)$$

Complexité de la fonction tri

Suite

$$
\begin{aligned}
T(n) &= 3^m.\ T((2/3)^m\ n) + 4.(1+3+3^2+...+3^{m-1}) \\
&= 3^m.\ T((2/3)^m\ n) + 4.\frac{3^m}{2} \\
&= 3^{log_{1.5}(n)} + 4.\frac{3^{log_{1.5}(n)}}{2} \\
&= 3.\ 3^{log_{1.5}(n)}
\end{aligned}
$$

Complexité de la fonction tri

Suite

Comme :

$$log_{(\frac{3}{2})}(n) = \frac{log_3(n)}{log_3(\frac{3}{2})}$$

De même :

$$log_3(\frac{3}{2}) = 1 - log_3(2) = 0.37$$

De ce fait :

$$\begin{aligned} 3^{log_{1.5}(n)} &= 3^{\frac{log_3(n)}{log_3(\frac{3}{2})}} \\ &= n^{\frac{1}{log_3(\frac{3}{2})}} \\ &= n^{2.7} \end{aligned}$$

Complexité de la fonction tri

Suite

Au final la complexité de l'algorithme de tri :

$$T(n) \in O(n^{2.7})$$

plus coûteux que les algorithmes de tri standard!!!!