Étude d'algorithmes

Compléments de mathématiques- semaine 4

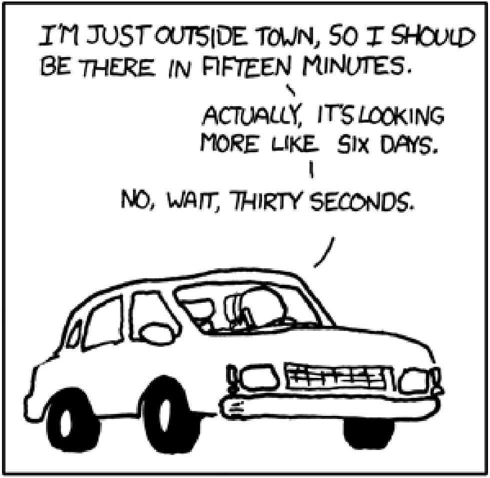
Guillaume CONNAN

IUT de Nantes - Département d'informatique

janvier 2016

Sommaire

Sommaire



THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v2

hQIOAykxJSA9FO+AEAf/QGAaWDSJHdvr3cy6c/xStJMz1DHmJfiG6Iw7ObGAzbIT
bufcEVR13nwnUngNNzpPEWHWm5oJltJOkAPkqypFOEOMKiebVyBldMOko2knnCR5
49jpiESoaIBhlwhS8u6Q4yQSKW6/Ch92giPJiLwIen79wLa9eUJwBjV2NXv7OOJN
knSPmOmbORZfYPfNYa3hAPaPIWnj7X7PP48uG2mKyAajx7BllEJAwTp9LLnOrERm
rAA6u5hpwRXzUK/zl44VOQF8CxgO96Ob2+m+dTkOSl4TqD/I8frOIw4a5lOp3aCn
AgiiSDyqNMWBA4RNhMsO83vQ1/oj4InwJLOh45JX3wf/fmgsIg126CsQkh8CI3dJ
BGZBeS3+XcPniZBiahMN1g9ANf9/+KHe/vN2HlxMz9aw4dT84Y6/kRJq6MpDOq/X

...

jiUvton//denRAhS5kunkzkSvYuQS12xIbbRdFp2XuGxzDJ8Z9kmm96kQXdXAp/8
McrpULj83CTCYxnX1W2zN9aC8m96mZ6qbftlQVLWIKpJJXgjJssUNXIOoMFW6KnL
yacE93YKFDQQWusUs7VKr4AfWaSN4Owoo3XdCZMPgoA3Bs7Fa9ZD5mfQj6OKTggb
2t+l27LpglSrJjal/LlbvzXhWbF+xHbvd3gMygJaBk2P4nnbyFQTtVV+dIA8OoWq
S2l8xhgsnhb6Sh5upDaasqZnJyQEHPlJh9Ae5dIjjLpx7W1sHHhO3Pcimaa8GFg6
ffAlS3MH9HDzVtIaOi2l1YqO
=C6AC
-----END PGP MESSAGE-----
```

RSA : calcul de x^n avec n de taille minimum 1024 bits.

RSA : calcul de x^n avec n de taille minimum 1024 bits.

```
In [1]: 2**1024 - 1
Out[1]: 179769313486231590772930519078902473361797697894230657273
430081157732675805500963132708477322407536021120113879871393357658
789768814416622492847430639474124377767893424865485276302219601246
094119453082952085005768838150682342462881473913110540827237163
350510684586298239947245938479716304835356329624224137215
```

RSA : calcul de x^n avec n de taille minimum 1024 bits.

```
In [1]: 2**1024 - 1
Out[1]: 1797693134862315907729305190789024733617976978942306572734300811577326758055009631327084773224075360211201138798713933576587897688144166224928474306394741243777678934248654852763022196012460941194530829520850057688381506823424628814739131105408272371633505106845862982399472459384797163048353563296242241372l5
```

Mon HP EliteBook doté d'un `i5` : 100 GFlops.

En combien de temps vais-je décoder le premier caractère de la clé de transmission ?

Algorithme naïf : $x^n = x \times x^{n-1}$.

```
def pow_naive(x,n) :
    assert n > 0, 'Exposant négatif'
    p = 1
    e = n
    while e > 0 :
        p *= x
        e -= 1
    return p
```

$$\frac{10^{308}}{10^{11}} = 10^{297}s$$

$$\frac{10^{308}}{10^{11}} = 10^{297}s = \frac{10^{297}}{3600 \times 24 \times 365.25 \times 100} \approx 3 \times 10^{287} \text{ siècles...}$$

Google

Sergey Brin Larry Page

Google

Sergey Brin   Larry Page

En 1996, Larry Page et Sergey Brin sont doctorants en informatique à Standford et réfléchissent à un algorithme de classement des pages du web qui deviendra deux ans plus tard *PageRank* et permettra à ces deux étudiants de fonder Google...

L'algo nécessite de travailler sur des matrices du type :

```
sage: trans
[  0 1/4 1/4 1/4 1/4   0   0   0   0   0   0   0]
[1/2   0 1/2   0   0   0   0   0   0   0   0   0]
[1/2   0   0 1/2   0   0   0   0   0   0   0   0]
[1/2 1/2   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0 1/3 1/3 1/3   0   0   0   0]
[1/2   0   0   0   0   0 1/2   0   0   0   0   0]
[  0   0   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0 1/2   0 1/2   0   0   0]
[  0   0   0   0 1/4   0   0   0   0 1/4 1/4 1/4]
[  0   0   0   0   0   0   0   0 1/2   0 1/2   0]
[  0   0   0   0   0   0   0   0 1/2   0   0 1/2]
[  0   0   0   0   0   0   0   0 1/2 1/2   0   0]
```

L'algo nécessite de travailler sur des matrices du type :

```
sage: trans
[  0 1/4 1/4 1/4 1/4   0   0   0   0   0   0   0]
[1/2   0 1/2   0   0   0   0   0   0   0   0   0]
[1/2   0   0 1/2   0   0   0   0   0   0   0   0]
[1/2 1/2   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0 1/3 1/3 1/3   0   0   0   0]
[1/2   0   0   0   0   0 1/2   0   0   0   0   0]
[  0   0   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0 1/2   0 1/2   0   0   0]
[  0   0   0   0 1/4   0   0   0   0 1/4 1/4 1/4]
[  0   0   0   0   0   0   0   0 1/2   0 1/2   0]
[  0   0   0   0   0   0   0   0 1/2   0   0 1/2]
[  0   0   0   0   0   0   0   0 1/2 1/2   0   0]
```

Sauf que ces matrices ont des tailles beaucoup, beaucoup plus grandes...Pauvre mémoire !

Un problème à notre portée

On dispose d'une liste de N nombres. Déterminez le nombre de triplets dont la somme est nulle.

```python
def trois_sommes(xs):
    N = len(xs)
    cpt = 0
    for i in range(N):
        for j in range(i + 1, N):
            for k in range(j + 1, N):
                if xs[i] + xs[j] + xs[k] == 0:
                    cpt += 1
    return cpt
```

```
In [1]: %timeit trois_sommes([randint(-10000,10000) for k in
        ↪   range(100)])
10 loops, best of 3: 27.5 ms per loop

In [2]: %timeit trois_sommes([randint(-10000,10000) for k in
        ↪   range(200)])
1 loops, best of 3: 216 ms per loop

In [3]: %timeit trois_sommes([randint(-10000,10000) for k in
        ↪   range(400)])
1 loops, best of 3: 1.82 s per loop
```

```python
def trois_sommes_comp(xs):
    n = len(xs)
    return len([(i,j,k) for i in range(n) for j in range(i+1, n) for k
        ↪   in range(j+1,n) if xs[i] + xs[j] + xs[k] == 0 ])
```

```
In [24]: %timeit trois_sommes_comp([randint(-10000,10000) for k in
         ↪   range(100)])
10 loops, best of 3: 28.1 ms per loop

In [25]: %timeit trois_sommes_comp([randint(-10000,10000) for k in
         ↪   range(200)])
1 loops, best of 3: 222 ms per loop

In [26]: %timeit trois_sommes_comp([randint(-10000,10000) for k in
         ↪   range(400)])
1 loops, best of 3: 1.85 s per loop
```

```
from time import perf_counter
from math import log2

def temps(xs):
    debut = perf_counter()        # on déclenche le chrono
    trois_sommes(xs)              # on lance le calcul
    return perf_counter() - debut # on arrête le chrono quand c'est
            ↪   fini

# on fabrique une liste contenant les temps de calcul pour des
        ↪   longueurs de 100, 200, 400 et 800
t = [temps(range(100 * 2**k)) for k in range(4)]

# on forme la liste des ratios
ratio = [t[k + 1] / t[k] for k in range(3)]

# on applique la fonction log2 aux éléments de la liste des ratios
logratio = [log2(r) for r in ratio]
```

```
In [4]: ratio
Out[4]: [7.523860206447286, 9.118789882406599, 8.5098312160934]

In [5]: logratio
Out[5]: [2.940628541715559, 3.133284732580891, 3.128693841844642]
```

```
In [6]: temps(range(400))
Out[6]: 4.005484320001415
```

```
In [6]: temps(range(400))
Out[6]: 4.005484320001415
```

$4,00 = a \times 400^3$ donc $a \approx 6,25 \times 10^{-8}$
Donc pour $N = 1000$, on devrait avoir un temps de
$6,25 \times 10^{-8} \times 10^9 = 62,5$

```
In [6]: temps(range(400))
Out[6]: 4.005484320001415
```

$4,00 = a \times 400^3$ donc $a \approx 6,25 \times 10^{-8}$

Donc pour $N = 1000$, on devrait avoir un temps de $6,25 \times 10^{-8} \times 10^9 = 62,5$

```
In [7]: temps(range(1000))
Out[7]: 68.54615448799996
```

En C

```
$ gcc  -std=c99 -Wall -Wextra  -Werror -pedantic -O4 -o somm3
       ↪  Trois_Sommes.c
$ ./somm3
Temps en sec pour 100 : 0.00000
Temps en sec pour 200 : 0.000000
Temps en sec pour 400 : 0.020000
Temps en sec pour 800 : 0.090000
Temps en sec pour 1600 : 0.720000
Temps en sec pour 3200 : 5.760000
Temps en sec pour 6400 : 45.619999
Temps en sec pour 12800 : 360.839996
```

On a la même évolution en N^3 avec un rapport de 8 entre chaque doublement de taille

On a la même évolution en N^3 avec un rapport de 8 entre chaque doublement de taille mais la constante est bien meilleure :
$45,61 = a \times 6400^3$ d'où $a \approx 1,74 \times 10^{-10}$
$360,84 = a \times 12800^3$ d'où $a \approx 1.72 \times 10^{-10}$

```python
def trois_sommes(xs):
    N = len(xs)
    cpt = 0
    for i in range(N):
        for j in range(i + 1, N):
            for k in range(j + 1, N):
                if xs[i] + xs[j] + xs[k] == 0:
                    cpt += 1
    return cpt
```

```python
def trois_sommes(xs):
    N = len(xs)
    cpt = 0
    for i in range(N):
        xi = xs[i]
        for j in range(i + 1, N):
            sij = xi + xs[j]
            for k in range(j + 1, N):
                if sij + xs[k] == 0:
                    cpt += 1
    return cpt
```

```
In [28]:  xs = list(range(-50,51))
In [29]: %timeit trois_sommes(xs)
100 loops, best of 3: 12.9 ms per loop

In [30]:  xs = list(range(-100,101))
In [31]: %timeit trois_sommes(xs)
10 loops, best of 3: 94.4 ms per loop

In [32]:  xs = list(range(-200,201))
In [33]: %timeit trois_sommes(xs)
1 loops, best of 3: 851 ms per loop

In [34]:  xs = list(range(-400,401))
In [35]: %timeit trois_sommes(xs)
1 loops, best of 3: 7.04 s per loop
```

Loi de Brooks

Adding manpower to a late software project makes it later » a result of the fact that the expected advantage from splitting work among N programmers is $O(N)$, but the complexity and communications cost associated with coordinating and then merging their work is $O(N^2)$

Définition 1 (« Grand O »)

Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. On dit que f est un « grand O » de g et on note $f = O(g)$ ou $f(n) = O(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \leqslant C|g(n)|$ pour tout $n \in \mathbb{N}$.

Définition 2 (« Grand Oméga »)

Soit f et g deux fonctions de $\mathbb{R}$ dans lui-même. On dit que f est un « grand Oméga » de g et on note $f = \Omega(g)$ ou $f(n) = \Omega(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \geqslant C|g(n)|$ pour tout $n \in \mathbb{N}^*$.

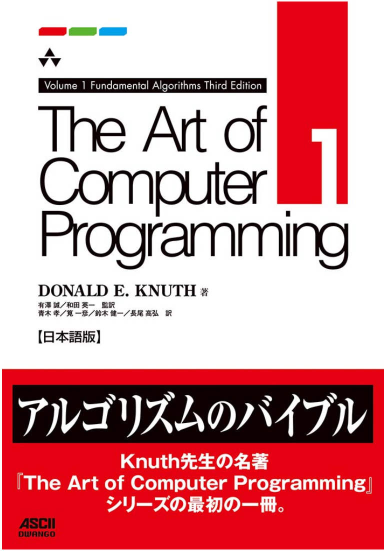
Définition 3 (« Grand Théta »)

$$f = \Theta(g) \iff f = O(g) \quad \wedge \quad f = \Omega(g)$$

coût \ n	100	1000	10^6	10^9
$\log_2(n)$ logarithmique	≈ 7	≈ 10	≈ 20	≈ 30
$n\log_2(n)$ linéarithmique	≈ 665	$\approx 10^4$	$\approx 2 \cdot 10^7$	$\approx 3 \cdot 10^{10}$
n^2 quadratique	10^4	10^6	10^{12}	10^{18}
n^3 cubique	10^6	10^9	10^{18}	10^{27}
2^n exponentiel	$\approx 10^{30}$	$> 10^{300}$	$> 10^{10^5}$	$> 10^{10^8}$

coût \ n	100	1000	10^6	10^9
$\log_2(n)$ logarithmique	≈ 7	≈ 10	≈ 20	≈ 30
$n\log_2(n)$ linéarithmique	≈ 665	$\approx 10^4$	$\approx 2 \cdot 10^7$	$\approx 3 \cdot 10^{10}$
n^2 quadratique	10^4	10^6	10^{12}	10^{18}
n^3 cubique	10^6	10^9	10^{18}	10^{27}
2^n exponentiel	$\approx 10^{30}$	$> 10^{300}$	$> 10^{10^5}$	$> 10^{10^8}$

Gardez en tête que l'âge de l'Univers est environ de 10^{18} secondes...


Volume 1 Fundamental Algorithms Third Edition
The Art of Computer Programming
1
DONALD E. KNUTH 著
有澤 誠／和田 英一　監訳
青木 孝／筧 一彦／鈴木 健一／長尾 高弘　訳
【日本語版】
アルゴリズムのバイブル
Knuth先生の名著
『The Art of Computer Programming』
シリーズの最初の一冊。
ASCII
DWANGO

```python
def trois_sommes(xs):
    N = len(xs)
    cpt = 0
    for i in range(N):
        xi = xs[i]
        for j in range(i + 1, N):
            sij = xi + xs[j]
            for k in range(j + 1, N):
                if sij + xs[k] == 0:
                    cpt += 1
    return cpt
```

OPÉRATION	FRÉQUENCE
Déclaration de la fonction et du paramètre (l. 1)	2
Déclaration de N, cpt et i (l. 2, 3 et 4)	3
Affectation de N, cpt et i (l. 2, 3 et 4)	3
Déclaration de xi (l. 5)	N
Affectation de xi (l. 5)	N
Accès à xs[i] (l. 5)	N
Déclaration de j (l.6)	N
Calcul de l'incrément de i (l. 6)	N
Affectation de j (l.6)	N

OPÉRATION	FRÉQUENCE
Déclaration de sij (l. 7)	S_1
Affectation de sij (l. 7)	S_1
Accès à xs[j] (l.7)	S_1
Somme (l.7)	S_1
Déclaration de k (l.8)	S_1
Incrément de j (l. 8)	S_1
Affectation de k (l.8)	S_1
Accès à x[k] (l.9)	S_2
Calcul de la somme (l.9)	S_2
Comparaison à 0 (l.9)	S_2
Incrément de cpt (l.9)	entre 0 et S_2
Affectation de la valeur de retour (l.11)	1

Calcul

Que valent S_1 et S_2 ?

$$S_1 = \sum_{i=0}^{N-1} N - (i + 1)$$

$$S_1 = \sum_{i=0}^{N-1} N - (i+1) = \sum_{i'=0}^{N-1} i' = \frac{N(N-1)}{2}$$

$$S_1 = \sum_{i=0}^{N-1} N - (i+1) = \sum_{i'=0}^{N-1} i' = \frac{N(N-1)}{2} \qquad (\text{avec } i' = N - (i+1))$$

$$S_2 = \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} N - (j+1)$$

$$
\begin{aligned}
S_2 &= \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} N - (j+1) \\
&= \sum_{i=0}^{N-1} \sum_{j'=0}^{N-(i+2)} j' \qquad (j' = N - (j+1))
\end{aligned}
$$

$$
\begin{aligned}
S_2 &= \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} N-(j+1) \\
&= \sum_{i=0}^{N-1} \sum_{j'=0}^{N-(i+2)} j' \qquad (j' = N-(j+1)) \\
&= \sum_{i=0}^{N-2} \frac{(N-(i+2))(N-(i+1))}{2}
\end{aligned}
$$

$$\begin{aligned}
S_2 &= \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} N-(j+1) \\
&= \sum_{i=0}^{N-1} \sum_{j'=0}^{N-(i+2)} j' \qquad (j' = N-(j+1)) \\
&= \sum_{i=0}^{N-2} \frac{(N-(i+2))(N-(i+1))}{2} \\
&= \sum_{i'=1}^{N-2} \frac{i'(i'+1)}{2} \qquad (i' = N-(i+2))
\end{aligned}$$

$$
\begin{aligned}
S_2 &= \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} N-(j+1) \\
&= \sum_{i=0}^{N-1} \sum_{j'=0}^{N-(i+2)} j' \qquad (j' = N-(j+1)) \\
&= \sum_{i=0}^{N-2} \frac{(N-(i+2))(N-(i+1))}{2} \\
&= \sum_{i'=1}^{N-2} \frac{i'(i'+1)}{2} \qquad (i' = N-(i+2)) \\
&= \frac{1}{2}\left(\sum_{i=1}^{N-2} i'^2 + \sum_{i=1}^{N-2} i'\right)
\end{aligned}
$$

$$
\begin{aligned}
S_2 &= \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} N-(j+1) \\
&= \sum_{i=0}^{N-1} \sum_{j'=0}^{N-(i+2)} j' \qquad (j' = N-(j+1)) \\
&= \sum_{i=0}^{N-2} \frac{(N-(i+2))(N-(i+1))}{2} \\
&= \sum_{i'=1}^{N-2} \frac{i'(i'+1)}{2} \qquad (i' = N-(i+2)) \\
&= \frac{1}{2}\left(\sum_{i=1}^{N-2} i'^2 + \sum_{i=1}^{N-2} i'\right) \\
&= \frac{1}{2}\left(\frac{(N-2)(2N-3)(N-1)}{6} + \frac{(N-2)(N-1)}{2}\right)
\end{aligned}
$$

$$
\begin{aligned}
S_2 &= \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} N-(j+1) \\
&= \sum_{i=0}^{N-1} \sum_{j'=0}^{N-(i+2)} j' \qquad (j' = N-(j+1)) \\
&= \sum_{i=0}^{N-2} \frac{(N-(i+2))(N-(i+1))}{2} \\
&= \sum_{i'=1}^{N-2} \frac{i'(i'+1)}{2} \qquad (i' = N-(i+2)) \\
&= \frac{1}{2}\left(\sum_{i=1}^{N-2} i'^2 + \sum_{i=1}^{N-2} i'\right) \\
&= \frac{1}{2}\left(\frac{(N-2)(2N-3)(N-1)}{6} + \frac{(N-2)(N-1)}{2}\right) \\
&= \frac{N(N-1)(N-2)}{6}
\end{aligned}
$$

Notons a le temps constant d'affectation, Notons a le temps constant d'affectation, d le temps constant de déclaration, Notons a le temps constant d'affectation, d le temps constant de déclaration,x le temps constant d'accès à une cellule, Notons a le temps constant d'affectation, d le temps constant de déclaration,x le temps constant d'accès à une cellule, s le temps constant d'une somme,

Notons a le temps constant d'affectation, d le temps constant de déclaration,x le temps constant d'accès à une cellule, s le temps constant d'une somme, c le temps constant d'une comparaison.

Notons a le temps constant d'affectation, d le temps constant de déclaration,x le temps constant d'accès à une cellule, s le temps constant d'une somme, c le temps constant d'une comparaison.
Le temps d'exécution vérifie :

$$\tau(N) \leqslant (2d + 3d + 3a + a) + (d + a + x + d + s + a)N + (d + a + x + s + d + s + a)S_1 + (x + s + c + s)S_2 \quad (1)$$

Notons a le temps constant d'affectation, d le temps constant de déclaration,x le temps constant d'accès à une cellule, s le temps constant d'une somme, c le temps constant d'une comparaison.
Le temps d'exécution vérifie :

$$\begin{aligned}\tau(N) \leqslant (2d + 3d + 3a + a) + (d + a + x + d + s + a)N + \\ (d + a + x + s + d + s + a)S_1 + (x + s + c + s)S_2 \quad (1)\end{aligned}$$

Or $S_1 \sim N^2$ et $S_2 \sim N^3$ quand N est « grand ».

Notons a le temps constant d'affectation, d le temps constant de déclaration,x le temps constant d'accès à une cellule, s le temps constant d'une somme, c le temps constant d'une comparaison.
Le temps d'exécution vérifie :

$$\begin{aligned}\tau(N) \leqslant (2d+3d+3a+a)+(d+a+x+d+s+a)N+\\ (d+a+x+s+d+s+a)S_1+(x+s+c+s)S_2 \quad (1)\end{aligned}$$

Or $S_1 \sim N^2$ et $S_2 \sim N^3$ quand N est « grand ».
Finalement...

Notons a le temps constant d'affectation, d le temps constant de déclaration,x le temps constant d'accès à une cellule, s le temps constant d'une somme, c le temps constant d'une comparaison.
Le temps d'exécution vérifie :

$$\begin{aligned}\tau(N) \leqslant (2d+3d+3a+a)+(d+a+x+d+s+a)N+ \\ (d+a+x+s+d+s+a)S_1+(x+s+c+s)S_2 \quad (1)\end{aligned}$$

Or $S_1 \sim N^2$ et $S_2 \sim N^3$ quand N est « grand ».
Finalement...

$$\tau(N) = O(N^3)$$

Sommaire

Addition de deux entiers de n chiffres : $O(n)$

Addition de deux entiers de n chiffres : $O(n)$
Multiplier un entier de n chiffres par un entier de 1 chiffre : $O(n)$

Addition de deux entiers de n chiffres : $O(n)$
Multiplier un entier de n chiffres par un entier de 1 chiffre : $O(n)$
Algorithme de l'école primaire pour multiplier deux nombres de n chiffres :

Addition de deux entiers de n chiffres : $O(n)$
Multiplier un entier de n chiffres par un entier de 1 chiffre : $O(n)$
Algorithme de l'école primaire pour multiplier deux nombres de n chiffres : n multiplications d'un nombre de n chiffres par un nombre de 1 chiffre puis une addition des n nombres obtenus :

Addition de deux entiers de n chiffres : $O(n)$
Multiplier un entier de n chiffres par un entier de 1 chiffre : $O(n)$
Algorithme de l'école primaire pour multiplier deux nombres de n chiffres : n multiplications d'un nombre de n chiffres par un nombre de 1 chiffre puis une addition des n nombres obtenus :$O(n^2)$

$O(n^2)$: multiplication de nombres deux fois plus petits $\longrightarrow$ quatre fois plus rapide ?

$$xy = (10^m x_1 + x_2)(10^m y_1 + y_2) = 10^{2m} x_1 y_1 + 10^m (x_2 y_1 + x_1 y_2) + x_2 y_2$$

```
Fonction MUL(x:entier ,y: entier ):entier
Si n==1 Alors
  | Retourner x.y
Sinon
  | m ← ⌊n/2⌋
  | x1 ← ⌊x/10^m⌋
  | x2 ← x mod 10^m
  | y1 ← ⌊y/10^m⌋
  | y2 ← y mod 10^m
  | a ← MUL(x1,y1,m)
  | b ← MUL(x2,y1,m)
  | c ← MUL(x1,y2,m)
  | d ← MUL(x2,y2,m)
  | Retourner 10^(2m) a + 10^m (b + c) + d
FinSi
```

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant.

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant. L'addition finale est en λn

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant. L'addition finale est en λn donc le temps d'exécution est défini par :

$$T(n) = 4T(\lfloor n/2 \rfloor) + \lambda n \qquad T(1) = 1$$

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant. L'addition finale est en λn donc le temps d'exécution est défini par :

$$T(n) = 4T(\lfloor n/2 \rfloor) + \lambda n \qquad T(1) = 1$$

$$n = 2^k$$

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant. L'addition finale est en λn donc le temps d'exécution est défini par :

$$T(n) = 4T(\lfloor n/2 \rfloor) + \lambda n \qquad T(1) = 1$$

$$n = 2^k \qquad T(n) = T(2^k) = x_k$$

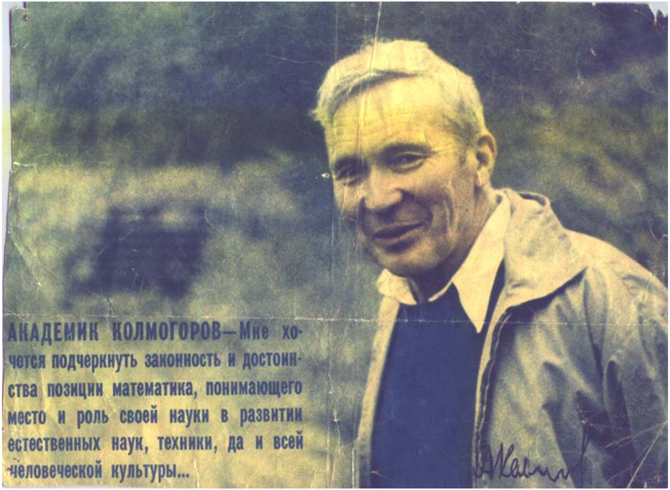
Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant. L'addition finale est en λn donc le temps d'exécution est défini par :

$$T(n) = 4T(\lfloor n/2 \rfloor) + \lambda n \qquad T(1) = 1$$

$$n = 2^k \qquad T(n) = T(2^k) = x_k$$

$$x_k = 4x_{k-1} + \lambda 2^k \qquad x_0 = 1$$

$$
\begin{aligned}
x_k &= 4(4x_{k-2} + \lambda' 2^{k-1}) + \lambda 2^k \\
&= 4^k x_0 + \sum_{i=1}^{k} \Lambda_k 2^k \\
&= 4^k + k\Lambda_k 2^k \\
&= n^2 + \Lambda_k n \log n \\
&= \Theta(n^2)
\end{aligned}
$$


АКАДЕМИК КОЛМОГОРОВ—Мне хочется подчеркнуть законность и достоинства позиции математика, понимающего место и роль своей науки в развитии естественных наук, техники, да и всей человеческой культуры...


Конференция памяти
Анатолия Алексеевича
Карацубы
по теории чисел
и приложениям
$M(n) = O(n^{\log_2 3})$
Москва, МИАН им. В.А. Стеклова
31 января 2014 года

$$bc + ad = ac + bd - (a-b)(c-d)$$

Recherche

En quoi cela simplifie le problème ? Quelle est alors la nouvelle complexité ?

Sommaire

Sun Zi (544–496 av. J.-C.)

Sun Zi (544–496 av. J.-C.)

Le commandement du grand nombre est le même pour le petit nombre, ce n'est qu'une question de division en groupes.

in « L'art de la guerre » de Sun Zi (VIe siècle avant JC)

On considère un problème de taille n qu'on découpe en a sous-problèmes de taille n/b avec a et b des entiers.

On considère un problème de taille n qu'on découpe en a sous-problèmes de taille n/b avec a et b des entiers. Le coût de l'algorithme est alors :

$$\begin{cases} T(1) = 1 \\ T(n) = a \times T(n/b) + \text{Reconstruction(n)} \end{cases}$$

On considère un problème de taille n qu'on découpe en a sous-problèmes de taille n/b avec a et b des entiers. Le coût de l'algorithme est alors :

$$\begin{cases} T(1) = 1 \\ T(n) = a \times T(n/b) + \text{Reconstruction(n)} \end{cases}$$

En général la reconstruction est de l'ordre de $c \times n^{\alpha}$.

On considère un problème de taille n qu'on découpe en a sous-problèmes de taille n/b avec a et b des entiers. Le coût de l'algorithme est alors :

$$\begin{cases} T(1) = 1 \\ T(n) = a \times T(n/b) + \text{Reconstruction(n)} \end{cases}$$

En général la reconstruction est de l'ordre de $c \times n^{\alpha}$.
Par exemple, pour l'algorithme de КАРАЦУБА nous avions

$$T(n) = 3T(\lfloor n/2 \rfloor) + \Theta(n)$$

On considère un problème de taille n qu'on découpe en a sous-problèmes de taille n/b avec a et b des entiers. Le coût de l'algorithme est alors :

$$\begin{cases} T(1) = 1 \\ T(n) = a \times T(n/b) + \text{Reconstruction(n)} \end{cases}$$

En général la reconstruction est de l'ordre de $c \times n^{\alpha}$.
Par exemple, pour l'algorithme de КАРАЦУБА nous avions

$$T(n) = 3T(\lfloor n/2 \rfloor) + \Theta(n)$$

$a = 3$, $b = 2$, $\alpha = 1$ et c quelconque avec n pair.

$$T(n) = aT(n/b) + cn^{\alpha}$$

$$
\begin{aligned}
T(n) &= aT(n/b) + cn^\alpha \\
&= a^2T(n/b^2) + ac(n/b)^\alpha + cn^\alpha
\end{aligned}
$$

$$\begin{aligned} T(n) &= aT(n/b) + cn^{\alpha} \\ &= a^2 T(n/b^2) + ac(n/b)^{\alpha} + cn^{\alpha} \\ &= \cdots \end{aligned}$$

$$\begin{aligned}
T(n) &= aT(n/b) + cn^{\alpha} \\
&= a^2 T(n/b^2) + ac(n/b)^{\alpha} + cn^{\alpha} \\
&= \cdots \\
&= a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i c(n/b^i)^{\alpha}
\end{aligned}$$

$$
\begin{array}{rcl}
T(n) & = & aT(n/b) + cn^{\alpha} \\
& = & a^2 T(n/b^2) + ac(n/b)^{\alpha} + cn^{\alpha} \\
& = & \cdots \\
& = & a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i c(n/b^i)^{\alpha}
\end{array}
$$

$n = b^{k_n}$

$$\begin{aligned} T(n) &= aT(n/b) + cn^\alpha \\ &= a^2 T(n/b^2) + ac(n/b)^\alpha + cn^\alpha \\ &= \cdots \\ &= a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i c(n/b^i)^\alpha \end{aligned}$$

$n = b^{k_n}$

$$T(n) = a^{k_n} + cn^\alpha \sum_{i=0}^{k_n-1} (a/b^\alpha)^i$$

$$T(n) = a^{k_n} + cn^{\alpha} \sum_{i=0}^{k_n - 1} (a/b^{\alpha})^i$$

$$T(n) = a^{k_n} + cn^{\alpha} \sum_{i=0}^{k_n - 1} (a/b^{\alpha})^i$$

1. $\boxed{a > b^{\alpha}}$ alors la somme géométrique est équivalente au premier terme négligé à une constante multiplicative et additive près $\left(\frac{1}{a/b^{\alpha}-1}\left(\frac{a}{b^{\alpha}}\right)^{k_n} - \frac{1}{a/b^{\alpha}-1}\right)$ à savoir $(a/b^{\alpha})^{k_n}$ et $T(n) = O(a^{k_n})$ soit $\boxed{T(n) = O(n^{\log_b(a)})}$

$$T(n) = a^{k_n} + cn^{\alpha} \sum_{i=0}^{k_n-1} (a/b^{\alpha})^i$$

1. $\boxed{a > b^{\alpha}}$ alors la somme géométrique est équivalente au premier terme négligé à une constante multiplicative et additive près $\left(\frac{1}{a/b^{\alpha}-1}\left(\frac{a}{b^{\alpha}}\right)^{k_n} - \frac{1}{a/b^{\alpha}-1}\right)$ à savoir $(a/b^{\alpha})^{k_n}$ et $T(n) = O(a^{k_n})$ soit $\boxed{T(n) = O(n^{\log_b(a)})}$
2. $\boxed{a = b^{\alpha}}$ alors $\boxed{T(n) = O(n^{\alpha} \log_b(n))}$

$$T(n) = a^{k_n} + cn^\alpha \sum_{i=0}^{k_n-1} (a/b^\alpha)^i$$

1. $\boxed{a > b^\alpha}$ alors la somme géométrique est équivalente au premier terme négligé à une constante multiplicative et additive près ($\frac{1}{a/b^\alpha - 1}\left(\frac{a}{b^\alpha}\right)^{k_n} - \frac{1}{a/b^\alpha - 1}$) à savoir $(a/b^\alpha)^{k_n}$ et $T(n) = O(a^{k_n})$ soit $\boxed{T(n) = O(n^{\log_b(a)})}$
2. $\boxed{a = b^\alpha}$ alors $\boxed{T(n) = O(n^\alpha \log_b(n))}$
3. $\boxed{a < b^\alpha}$ alors la série géométrique est convergente : $S(n) \approx n^{\log_b(a)} + cn^\alpha/(1 - a/b^\alpha)$ or $a < b^\alpha$ donc $\log_b(a) < \alpha$. Finalement $\boxed{T(n) = O(n^\alpha)}$

En fait, on peut démontrer (cf Cormen pp. 86-97) que les O sont des Θ et considérer des fonctions de reconstruction plus générales..
Pour revenir à l'algorithme de КАРАЦУБА , $a = 3$ et $b^\alpha = 2$ donc $T(n) = \Theta(n^{\log_2 3})$.

Sommaire

Sommaire

Expérience informatique

- k actionnaires

- k actionnaires
- $N = 2^n$ étages

- k actionnaires
- $N = 2^n$ étages
- RDC = 0

- k actionnaires
- $N = 2^n$ étages
- RDC = 0
- Il existe un étage fatal

- k actionnaires
- $N = 2^n$ étages
- RDC = 0
- Il existe un étage fatal
- RDC non fatal

- k actionnaires
- $N = 2^n$ étages
- RDC = 0
- Il existe un étage fatal
- RDC non fatal
- Minimiser le nombre d'essais

- Première idée : on commence au rez-de-chaussée et on progresse d'un étage.

- Première idée : on commence au rez-de-chaussée et on progresse d'un étage.
- Combien d'essais au pire ?

- Première idée : on commence au rez-de-chaussée et on progresse d'un étage.
- Combien d'essais au pire ?
- En moyenne

Sommaire

```
inf ← Étage inférieur

sup ← Étage supérieur

milieu ← (inf + sup)/2
Si estFatal(milieu) Alors
  | ChercherEntre(inf,milieu)
Sinon
  | ChercherEntre(milieu, sup)
FinSi
```

```
Fonction ChercherEntre( inf, sup : Entiers ) : Entier
{ pré-condition: il existe au moins un étage fatal entre inf et sup }
{ invariant: le plus petit étage fatal est entre inf et sup }
{ post-condition: la valeur retrournée est le plus petit étage fatal }
Si sup == inf Alors
 | Retourner sup
Sinon
 | milieu ← (inf + sup)/2
 | Si estFatal(milieu) Alors
 |  | ChercherEntre(inf,milieu)
 | Sinon
 |  | ChercherEntre(milieu, sup)
 | FinSi
FinSi
```

```
{ pré-cond.: il existe un étage fatal entre inf et sup, inf n'est pas fatal et inf
< sup }
{ invariant: le plus petit étage fatal est entre inf (non compris) et sup }
{ post-condition: la valeur retournée est le plus petit étage fatal }
inf,sup ← 0, N
TantQue sup > inf + 1 Faire
  { le plus petit étage fatal est entre inf (non compris) et sup et sup > inf +
1 }
  milieu ← (inf + sup)/2
  Si estFatal(milieu) Alors
    sup = milieu
  Sinon
    inf = milieu
  FinSi
FinTantQue
{ le plus petit étage fatal est entre inf (non compris) et sup et sup = inf + 1 }
Retourner sup
{ la valeur retournée est le plus petit étage fatal }
```

1. étude de la *terminaison* de l'algorithme : est-ce que la fonction renvoie effectivement une valeur ?

1. étude de la *terminaison* de l'algorithme : est-ce que la fonction renvoie effectivement une valeur ?
2. étude le la *correction* de l'algorithme : est-ce que la fonction renvoie la valeur attendue ?

1. étude de la *terminaison* de l'algorithme : est-ce que la fonction renvoie effectivement une valeur ?
2. étude le la *correction* de l'algorithme : est-ce que la fonction renvoie la valeur attendue ?
3. étude de la *complexité* de l'algorithme : peut-on estimer la vitesse d'exécution de cet algorithme ?

- TERMINAISON :

- TERMINAISON :

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i}$

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i} \quad \ell_n = 1$
- CORRECTION :

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i} \quad \ell_n = 1$
- CORRECTION :

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i} \quad \ell_n = 1$
- CORRECTION :l'invariant est vérifié à chaque itération
- COMPLEXITÉ :

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i} \quad \ell_n = 1$
- CORRECTION :l'invariant est vérifié à chaque itération
- COMPLEXITÉ :

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i} \quad \ell_n = 1$
- CORRECTION :l'invariant est vérifié à chaque itération
- COMPLEXITÉ : il suffit donc de compter combien de lancers ont été effectués. La réponse est dans l'étude faite pour prouver la terminaison : c'est à l'étape n que l'on atteint la condition de sortie de l'algorithme.

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i} \quad \ell_n = 1$
- CORRECTION :l'invariant est vérifié à chaque itération
- COMPLEXITÉ : il suffit donc de compter combien de lancers ont été effectués. La réponse est dans l'étude faite pour prouver la terminaison : c'est à l'étape n que l'on atteint la condition de sortie de l'algorithme.

  Que vaut n ?

- TERMINAISON : $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i} \quad \ell_n = 1$
- CORRECTION :l'invariant est vérifié à chaque itération
- COMPLEXITÉ : il suffit donc de compter combien de lancers ont été effectués. La réponse est dans l'étude faite pour prouver la terminaison : c'est à l'étape n que l'on atteint la condition de sortie de l'algorithme.

  Que vaut n ?

  On a posé au départ que le nombre d'étages était une puissance de 2 : $N = 2^n$. Ainsi, $n = \log_2 N$.

Sommaire

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2.

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2. Il va falloir chercher :

- un nombre

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2. Il va falloir chercher :

- un nombre

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2. Il va falloir chercher :

- un nombre (un étage)
- dans un tableau

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2. Il va falloir chercher :

- un nombre (un étage)
- dans un tableau

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2. Il va falloir chercher :

- un nombre (un étage)
- dans un tableau(un immeuble)
- de 2^{10} nombres

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2. Il va falloir chercher :

- un nombre (un étage)
- dans un tableau(un immeuble)
- de 2^{10} nombres

Cherchons une approximation de $x^2 - 2 = 0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2. Il va falloir chercher :

- un nombre (un étage)
- dans un tableau(un immeuble)
- de 2^{10} nombres (étages)

1	$1+2^{-10}$	$1+2\times 2^{-10}$	$1+3\times 2^{-10}$	$\cdots$	$1+2^{10}\times 2^{-10}$

1	$1+2^{-10}$	$1+2\times 2^{-10}$	$1+3\times 2^{-10}$	$\cdots$	$1+2^{10}\times 2^{-10}$

Notre fonction booléenne « estFatal » est alors le test $x \mapsto x * x <= 2$ et l'on va chercher une cellule de ce tableau par dichotomie comme on cherchait un étage dans un immeuble.

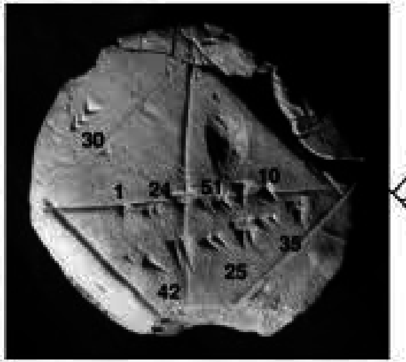
```
def racineDicho(prec):
    cpt = 0
    inf = 1
    sup = 2
    while (sup - inf > prec):
        m = inf + (sup - inf) / 2
        cpt += 1
        if m*m <= 2:
            inf = m
        else:
            sup = m
    return sup,cpt
```

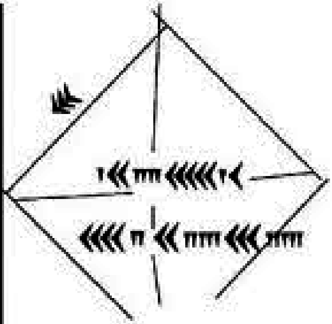
```
In [1]: racineDicho(2**(-10))
Out[1]: (1.4150390625, 10)

In [2]: racineDicho(2**(-15))
Out[2]: (1.414215087890625, 15)

In [3]: racineDicho(2**(-20))
Out[3]: (1.4142141342163086, 20)

In [4]: racineDicho(2**(-30))
Out[4]: (1.4142135623842478, 30)

In [5]: racineDicho(2**(-50))
Out[5]: (1.4142135623730958, 50)
```

Sommaire

Sommaire


ΗΡωΝ
ΟΦΙΛΟCΟ
ΦΟ


30
1
24
51
10
35
25
42



« Puisque alors les 720 n'ont pas le côté exprimable, nous prendrons le côté avec une très petite différence ainsi. Puisque le carré le plus voisin de 720 est 729 et il a 27 comme côté, divise les 720 par le 27 : il en résulte 26 et deux tiers. Ajoute les 27 : il en résulte 53 et deux tiers. De ceux-ci la moitié : il en résulte 26 2' 3'. Le côté approché de 720 sera donc 26 2' 3'. En effet 26 2' 3' par eux-mêmes : il en résulte 720 36', de sorte que la différence est une 36e part d'unité. Et si nous voulons que la différence se produise par une part plus petite que le 36', au lieu de 729, nous placerons les 720 et 36' maintenant trouvés et, en faisant les mêmes choses, nous trouverons la différence qui en résulte inférieure, de beaucoup, au 36'. »

Héron d'Alexandrie, Metrica, tome I, 8

- Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ et vice-versa.

- Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ et vice-versa.
- La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.

- Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ et vice-versa.
- La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.
- On peut montrer que c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).

```
Fonction heron_rec( a,fo : Réels  n : Entier ) : Réel
Si n = 0 Alors
 |  Retourner 0
Sinon
 |  Retourner heron_rec(a,(fo + a / fo) / 2, n - 1)
FinSi
Fonction heron_it( a,fo : Réels  n : Entier ) : Réel
app ← 0
Pour k de 0 à n - 1 Faire
 |  app ← (app + a / app) / 2
FinPour
Retourner app
```

Sommaire

Théorème de la limite monotone

Toute suite croissante majorée (ou décroissante minorée) converge.

Théorème de la limite monotone

Toute suite croissante majorée (ou décroissante minorée) converge.

Théorème de la limite monotone

Toute suite croissante majorée (ou décroissante minorée) converge.

Théorème light du point fixe

Soit I un intervalle fermé de $\mathbb{R}$, soit f une fonction continue de I vers I et soit (r_n) une suite d'éléments de I telle que $r_{n+1} = f(r_n)$ pour tout entier naturel n. SI (r_n) est convergente ALORS sa limite est UN point fixe de f appartenant à I.

$$r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2} \qquad r_0 = 1$$

$$r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2} \qquad r_0 = 1$$

1. pour tout entier naturel non nul n, on a $r_n \geqslant \sqrt{2}$;

$$r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2} \qquad r_0 = 1$$

1. pour tout entier naturel non nul n, on a $r_n \geqslant \sqrt{2}$;
2. la suite est décroissante ;

$$r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2} \qquad r_0 = 1$$

1. pour tout entier naturel non nul n, on a $r_n \geqslant \sqrt{2}$;
2. la suite est décroissante ;
3. $\sqrt{2}$ est l'unique point fixe positif de f.

Sommaire

Définition : ordre d'une suite

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$.

Définition : ordre d'une suite

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n\to+\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$.

Définition : ordre d'une suite

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$.
On dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

Définition : ordre d'une suite

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$.
On dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

$$\left| r_{n+1} - \sqrt{2} \right| = \left| \frac{(r_n - \sqrt{2})^2}{2r_n} \right|$$

Définition : ordre d'une suite

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$.
On dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

$$\left|r_{n+1} - \sqrt{2}\right| = \left|\frac{(r_n - \sqrt{2})^2}{2r_n}\right|$$

$$\left|r_{n+1} - \sqrt{2}\right| \leqslant \left|(r_n - \sqrt{2})^2\right|$$

Définition : ordre d'une suite

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$.
On dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

$$\left| r_{n+1} - \sqrt{2} \right| = \left| \frac{(r_n - \sqrt{2})^2}{2r_n} \right|$$

$$\left| r_{n+1} - \sqrt{2} \right| \leqslant \left| (r_n - \sqrt{2})^2 \right|$$

$$d_n = -\log_{10} |r_n - \sqrt{2}|$$

Définition : ordre d'une suite

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$.
On dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

$$\left| r_{n+1} - \sqrt{2} \right| = \left| \frac{(r_n - \sqrt{2})^2}{2 r_n} \right|$$

$$\left| r_{n+1} - \sqrt{2} \right| \leqslant \left| (r_n - \sqrt{2})^2 \right|$$

$$d_n = -\log_{10} |r_n - \sqrt{2}| \qquad d_{n+1} \geqslant 2 d_n$$