

Complexité

Informatique pour tous

En général, il y a plusieurs algorithmes différents pour résoudre le même problème.

```
def somme(n):  
    s = 0  
    for k in range(n):  
        s += k  
    return s
```

```
def somme(n):  
    return n*(n - 1)//2
```

Certains sont plus rapides que d'autres : on voudrait un critère de comparaison entre 2 algorithmes.

Définition

La **complexité** d'un algorithme est le nombre d'opérations élémentaires qu'il réalise, exprimé en fonction de la taille de l'entrée.

Définition

La **complexité** d'un algorithme est le nombre d'opérations élémentaires qu'il réalise, exprimé en fonction de la taille de l'entrée.

Exemples d'opérations élémentaires :

- opérations sur les nombres : +, - *
- comparaisons de nombres : ==, <=, <, !=
- sur les listes : L.append(e), L[i]...

```
def somme(n):  
    s = 0  
    for k in range(n):  
        s += k  
    return s
```

Complexité :

```
def somme(n):  
    s = 0  
    for k in range(n):  
        s += k  
    return s
```

Complexité : n additions.

```
def somme(n):  
    s = 0  
    for k in range(n):  
        s += k  
    return s
```

Complexité : n additions.

```
def somme(n):  
    return n*(n - 1)//2
```

Complexité :

```
def somme(n):  
    s = 0  
    for k in range(n):  
        s += k  
    return s
```

Complexité : n additions.

```
def somme(n):  
    return n*(n - 1)//2
```

Complexité : 3 opérations élémentaires (1 addtion, 1 multiplication, 1 division)

On s'intéresse souvent à l'**ordre de grandeur** de la complexité quand la taille de l'entrée est grande, en négligeant les constantes (souvent trop difficiles à calculer exactement).

Grand O

La notation suivante (« grand O ») permet d'estimer l'ordre de grandeur d'une complexité:

La notation suivante (« grand O ») permet d'estimer l'ordre de grandeur d'une complexité:

Définition

$f(n) = O(g(n)) \iff \exists A, f(n) \leq Ag(n), \text{ pour } n \text{ assez grand}$

La notation suivante (« grand O ») permet d'estimer l'ordre de grandeur d'une complexité:

Définition

$$f(n) = O(g(n)) \iff \exists A, f(n) \leq Ag(n), \text{ pour } n \text{ assez grand}$$

« $O(f(n))$ » signifie donc : « au plus une constante fois $f(n)$ ».

La notation suivante (« grand O ») permet d'estimer l'ordre de grandeur d'une complexité:

Définition

$$f(n) = O(g(n)) \iff \exists A, f(n) \leq Ag(n), \text{ pour } n \text{ assez grand}$$

« $O(f(n))$ » signifie donc : « au plus une constante fois $f(n)$ ».

Exemple: on dira qu'un algorithme de complexité $5 + 2n$ est en complexité $O(n)$.

En pratique : pour mettre une complexité sous la forme $O(\dots)$, **on conserve seulement le terme dominant (le plus grand), sans la constante.**

En pratique : pour mettre une complexité sous la forme $O(\dots)$, **on conserve seulement le terme dominant (le plus grand), sans la constante.**

Exemples :

- $18n^3 - n + 20 =$

En pratique : pour mettre une complexité sous la forme $O(\dots)$, **on conserve seulement le terme dominant (le plus grand), sans la constante.**

Exemples :

- $18n^3 - n + 20 = O(n^3)$

En pratique : pour mettre une complexité sous la forme $O(\dots)$, **on conserve seulement le terme dominant (le plus grand), sans la constante.**

Exemples :

- $18n^3 - n + 20 = O(n^3)$

- $n \ln(n) + 3n^2 =$

En pratique : pour mettre une complexité sous la forme $O(\dots)$, **on conserve seulement le terme dominant (le plus grand), sans la constante.**

Exemples :

- $18n^3 - n + 20 = O(n^3)$
- $n \ln(n) + 3n^2 = O(n^2)$

En pratique : pour mettre une complexité sous la forme $O(\dots)$, **on conserve seulement le terme dominant (le plus grand), sans la constante.**

Exemples :

- $18n^3 - n + 20 = O(n^3)$

- $n \ln(n) + 3n^2 = O(n^2)$

- $2^n + 25n^3 =$

En pratique : pour mettre une complexité sous la forme $O(\dots)$, **on conserve seulement le terme dominant (le plus grand), sans la constante.**

Exemples :

- $18n^3 - n + 20 = O(n^3)$
- $n \ln(n) + 3n^2 = O(n^2)$
- $2^n + 25n^3 = O(2^n)$

Exemples

On considère que `print` est une opération élémentaire.

```
for i in range(n):  
    print("Hello World")
```

Complexité :

Exemples

On considère que `print` est une opération élémentaire.

```
for i in range(n):  
    print("Hello World")
```

Complexité : n .

```
for i in range(n):  
    for j in range(p):  
        print("Hello World")
```

Complexité :

Exemples

On considère que `print` est une opération élémentaire.

```
for i in range(n):  
    print("Hello World")
```

Complexité : n .

```
for i in range(n):  
    for j in range(p):  
        print("Hello World")
```

Complexité : np .

Exemples

```
for i in range(n):  
    for j in range(i):  
        print("Hello World")
```

Complexité :

Exemples

```
for i in range(n):  
    for j in range(i):  
        print("Hello World")
```

Complexité : $0 + 1 + 2 + 3 + \dots + n - 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$.

```
for i in range(n):  
    print("Hello World")  
for i in range(n):  
    print("Hello World")
```

Complexité :

Exemples

```
for i in range(n):  
    for j in range(i):  
        print("Hello World")
```

Complexité : $0 + 1 + 2 + 3 + \dots + n - 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$.

```
for i in range(n):  
    print("Hello World")  
for i in range(n):  
    print("Hello World")
```

Complexité : $2n = O(n)$.

Question

Écrire un algorithme pour calculer le nombre de diviseurs d'un entier n .
Quelle est sa complexité?

Question

Écrire un algorithme pour calculer le nombre de diviseurs d'un entier n .
Quelle est sa complexité?

```
nb_div = 0
for d in range(2, n + 1):
    if n % d == 0:
        nb_div += 1
```

Question

Écrire un algorithme pour calculer le nombre de diviseurs d'un entier n .
Quelle est sa complexité?

```
nb_div = 0
for d in range(2, n + 1):
    if n % d == 0:
        nb_div += 1
```

- 1 On effectue n fois les opérations $n \% d == 0$ et $nb_div = nb_div + 1$.

Question

Écrire un algorithme pour calculer le nombre de diviseurs d'un entier n .
Quelle est sa complexité?

```
nb_div = 0
for d in range(2, n + 1):
    if n % d == 0:
        nb_div += 1
```

- 1 On effectue n fois les opérations $n \% d == 0$ et $nb_div = nb_div + 1$.
- 2 On effectue une fois les opérations $nb_div = 0$ et $n+1$.

Question

Écrire un algorithme pour calculer le nombre de diviseurs d'un entier n .
Quelle est sa complexité?

```
nb_div = 0
for d in range(2, n + 1):
    if n % d == 0:
        nb_div += 1
```

- 1 On effectue n fois les opérations $n \% d == 0$ et $nb_div = nb_div + 1$.
- 2 On effectue une fois les opérations $nb_div = 0$ et $n+1$.
- 3 Au total, il y a $2 + 2n$ opérations, c'est à dire $O(n)$.

Nombre de diviseurs

Si d divise n alors $\frac{n}{d}$ divise aussi n .

On peut donc compter deux fois les diviseurs jusqu'à \sqrt{n} .

Nombre de diviseurs

Si d divise n alors $\frac{n}{d}$ divise aussi n .

On peut donc compter deux fois les diviseurs jusqu'à \sqrt{n} .

```
nb_div = 0
for d in range(2, int(n**.5)):
    if n % d == 0:
        nb_div += 2
if d*d == n:
    nb_div += 1
```

Question

Écrire une fonction premier déterminant si un entier est premier.

Question

Écrire une fonction `premier` déterminant si un entier est premier.

```
def premier(n):  
    for d in range(2, int(n**.5) + 1):  
        if n % d == 0:  
            return False  
    return True
```

Complexité?

Question

Écrire une fonction `premier` déterminant si un entier est premier.

```
def premier(n):  
    for d in range(2, int(n**.5) + 1):  
        if n % d == 0:  
            return False  
    return True
```

Complexité? ça dépend... si n est pair, même grand, `premier(n)` s'arrête pour $d = 2$.

Différentes notions de complexité

On distingue:

- **Complexité dans le pire des cas:** le plus grand nombre possible d'opérations réalisées.
- **Complexité dans le meilleur des cas:** le plus petit nombre possible d'opérations réalisées.
- **Complexité en moyenne:** le nombre moyen d'opérations réalisées.

Si on ne précise pas de quelle complexité on parle, il s'agit de la complexité dans le pire des cas.

Question

Écrire une fonction `premier` déterminant si un entier est premier.

```
def premier(n):  
    for d in range(2, int(n**.5) + 1):  
        if n % d == 0:  
            return False  
    return True
```

Complexité dans le pire des cas :

Question

Écrire une fonction `premier` déterminant si un entier est premier.

```
def premier(n):  
    for d in range(2, int(n**.5) + 1):  
        if n % d == 0:  
            return False  
    return True
```

Complexité dans le pire des cas : $O(\sqrt{n})$, si n est premier.

Question

Écrire une fonction `premier` déterminant si un entier est premier.

```
def premier(n):  
    for d in range(2, n):  
        if n % d == 0:  
            return False  
    return True
```

Complexité dans le meilleur des cas :

Question

Écrire une fonction `premier` déterminant si un entier est premier.

```
def premier(n):  
    for d in range(2, n):  
        if n % d == 0:  
            return False  
    return True
```

Complexité dans le meilleur des cas : $O(1)$, si n est pair.

Question

Écrire une fonction `premier` déterminant si un entier est premier.

```
def premier(n):  
    for d in range(2, n):  
        if n % d == 0:  
            return False  
    return True
```

Complexité en moyenne: ?? (difficile à calculer).

Question

Écrire une fonction `tous_premiers` telle que `tous_premiers(n)` renvoie la liste des nombres premiers entre 1 et n .

Question

Écrire une fonction `tous_premiers` telle que `tous_premiers(n)` renvoie la liste des nombres premiers entre 1 et n .

```
def tous_premiers(n):  
    L = []  
    for i in range(2, n):  
        if premier(i):  
            L.append(i)  
    return L
```

Complexité dans le pire des cas?

Question

Écrire une fonction `appartient` déterminant si un élément appartient à une liste.

Complexité dans le pire cas?

Question

Écrire une fonction `appartient` déterminant si un élément appartient à une liste.

Complexité dans le meilleur cas?

Question

Écrire une fonction `maximum` renvoyant le maximum d'une liste.

Complexité dans le pire cas?

Question

Écrire une fonction `maximum` renvoyant le maximum d'une liste.

Complexité dans le meilleur cas?

Complexités typiques:

- 1 $O(1)$ (constante): instantané.
- 2 $O(\ln(n))$ (logarithmique): très rapide.
- 3 $O(n)$ (linéaire): rapide.
- 4 $O(n^a)$, $a > 1$ (polynomiale): assez lent.
- 5 $O(a^n)$, $a > 1$ (exponentielle): très lent.

Ordres de grandeur

Temps d'exécutions pour un processeur à 1 Ghz (10^9 opérations élémentaires par seconde), en fonction du nombre n d'opérations élémentaires et de la complexité:

Complexité	$n = 10^2$	$n = 10^4$	$n = 10^6$
$\ln(n)$	7 ns	13 ns	20 ns
n	100 ns	10^{-5} s	1 ms
n^2	10 μ s	100 ms	17 min
n^3	1 ms	17 s	32 ans
2^n	10^{13} ans

Algorithme d'Euclide

```
def euclide(a, b):  
    while b != 0:  
        a, b = b, a%b  
    return a
```

Algorithme d'Euclide

```
def euclide(a, b):  
    while b != 0:  
        a, b = b, a%b  
    return a
```

Soient a_k et b_k , $k \geq 0$, la suite des valeurs prises par a et b .

On peut montrer (et on admet) que $b_k \leq \frac{b_{k-2}}{2}$, $\forall k \geq 2$. Alors:

$$b_{2k} \leq \frac{b_{2k-2}}{2} \leq \dots \leq \frac{b_0}{2^k}$$

Donc quand $\frac{b_0}{2^k} < 1$, $b_{2k} = 0$ et la boucle s'arrête. Or:

$$\frac{b_0}{2^k} < 1 \iff k > \log_2(b_0)$$

Donc il y a au plus $2 \log_2(b)$ itérations du while: pgcd est $O(\log(b))$.