

# Complexité d'un algorithme

Alexandre Benoit

BCPST

# I Coût d'un algorithme

# Un premier exemple

Pour traiter un même problème, il existe souvent plusieurs algorithmes. Pour décider quel algorithme choisir, l'un des critères est celui du temps d'exécution, qu'on appelle aussi **coût de l'algorithme**.

## Exemple

```
def evaluate(P, x):  
    S = 0  
    n = len(P)  
    for i in range(n):  
        S = S + P[i]*x**i  
    return S
```

- À quoi sert cet algorithme ?  
à évaluer un polynôme

- Combien de produits et de sommes effectue cet algorithme ?

effectuer puissance i coûte  $i$  produits. On effectue donc à chaque étape de la boucle  $i + 1$  produits et une somme.

En tout donc  
 $\frac{(n+1)(n+2)}{2}$  produits et  
 $n$  sommes

## Exemple

```
def horner(P, x):  
    resultat = 0  
    n = len(P)  
    for i in range(n-1, -1, -1):  
        resultat = x * resultat + P[i]  
    return resultat
```

- À quoi sert cet algorithme ?  
à évaluer un polynôme
- Combien de produits et de sommes effectue cet algorithme ? Il effectue 1 produits et 1 somme à chaque itération.  
L'algorithme effectue donc  $n$  produits et  $n$  sommes

- Si le premier algorithme avec comme entrée un polynôme de degré  $n$  s'exécute en 10 secondes, en combien de secondes s'exécutera-t-il avec en entrée un polynôme de degré  $100n$  ?

Avec un degré  $100n$ , le premier algorithme effectuera  $\frac{(100n + 1) \times (100n + 2)}{2}$  produits. C'est à dire environ 10000 fois plus de produits qu'avec le polynôme d'origine.

Si on considère que le produit prend plus de temps, on peut estimer que l'algorithme s'exécutera en environ 10000 fois plus de temps. C'est à dire en environ 28 heures.

- Même question avec le second algorithme.

Avec un degré  $100n$ , le second algorithme effectuera  $100n$  produits. C'est-à-dire 100 fois plus de produits qu'avec un polynôme de degré  $n$ . On peut estimer que l'algorithme s'exécutera en environ 100 fois plus de temps. C'est-à-dire en moins de 2 minutes.

# Déterminer le coût d'un algorithme

Pour déterminer le coût d'un algorithme, on se fonde en général sur le **modèle de complexité** suivant :

- Une affectation, une comparaison ou l'évaluation d'une opération arithmétique ayant en général un faible temps d'exécution, celui-ci sera considéré comme l'unité de mesure d'un algorithme.
- Le coût d'un test `if b: p else: q` est inférieur ou égal au maximum des coûts des instructions  $p$  et  $q$ , plus le temps d'évaluation de l'instruction  $b$
- Le coût d'une boucle `for i in iterable : p` est égal au nombre d'éléments de l'itérable multiplié par le coût de l'instruction  $p$  si ce dernier ne dépend pas de la valeur de  $i$ .  
Quand le coût du corps de la boucle dépend de la valeur de  $i$ , le coût total de la boucle est la somme des coûts de corps de la boucle pour chaque valeur de  $i$ .
- Pour les boucles `while`, on doit majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison.

## II Applications : Complexités de certains algorithmes

## Exemple 1 : Une premier test

Quel est le coût de l'algorithme suivant ?

```
def test_de_presence(L)
    if 2 in L:
        print ("Il y est")
    else:
        print ("Il n'y est pas")
```

Dans le pire cas, 2 n'est pas dans  $L$ . Dans ce cas, on doit effectuer  $len(L)$  tests. La complexité est alors de  $len(L) + 1$  opérations.



## Exemple 2 : Un second test

Quel est le coût de l'algorithme suivant ?

```
def test_de_presence(a, L)
    if a == 0:
        print ("a doit etre non nul")
    else:
        L.remove(a)
```

La commande `L.remove(a)` retire la première occurrence de l'élément  $a$  dans  $L$

Dans tous les cas, le coût du test est : 1 opération.

Si  $a \neq 0$ , et que  $a \notin L$ , on doit parcourir toute la liste pour chercher  $a$  ce qui fait  $len(L)$  opérations.

Si  $a \in L$ , on doit décaler tous les éléments se situant après  $a$ , ce qui coûte  $len(L)$  opérations.

On a donc  $len(L) + 1$  opérations.

## Exemple 3 : Une première boucle

Quel est le coût de l'algorithme suivant ?

```
for i in range(11):  
    print(i * i)
```

Cet algorithme effectue 11 multiplications. Il effectue donc 11 opérations.

## Exemple 4 : Une seconde boucle

Quel est le coût de l'algorithme suivant ?

```
for i in range(n):  
    print(i * i)
```

Cet algorithme effectue  $n$  multiplications. Il effectue donc  $n$  opérations.

## Exemple 5 : Une troisième boucle

Quel est le coût de l'algorithme suivant ?

```
for i in range(n):  
    for j in range(n):  
        print(i * j)
```

Cet algorithme construit une table de multiplications pour tous les entiers de 1 à  $n$  en donnant tous leurs multiples jusqu'au  $n$ -ième. Il comprend deux boucles imbriquées, chacune effectuant  $n$  répétitions de son corps; le corps de la boucle interne ne comporte qu'une multiplication. La complexité est ici  $n^2$  opérations.

## Exemple 5 : Une dernière boucle

Quel est le coût de l'algorithme suivant ?

```
for i in range(n):  
    for j in range(i):  
        print(i * j)
```

Cet algorithme construit une table de multiplications pour tous les entiers de 1 à  $n$  en donnant tous leurs multiples jusqu'au  $n$ -ième ; en prenant en compte la commutativité de la multiplication (On affichera  $10 \times 5$  mais pas  $5 \times 10$ ).

Il comprend deux boucles imbriquées, la première effectuant  $n$  répétitions de son corps. La boucle interne dépend du compteur de la boucle externe. Le corps de la boucle interne ne comporte qu'une multiplication. Le nombre d'opérations sera donc :

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

### III Complexité et équivalences entre les suites

## Définition d'une classe de complexité

Dans la plupart des cas, on a pas besoin d'aller jusqu'au niveau de détails des exemples précédents.

On sait que

$$n(n+1) \underset{n \rightarrow \infty}{\sim} n^2,$$

donc les complexités des algorithmes 4 et 5 sont du même « ordre ».

### Définition

On dit qu'un algorithme a une complexité  $g(n)$  **de l'ordre** de  $f(n)$  s'il existe des constantes strictement positives  $c$  et  $c'$  telles que

$$cf(n) \leq g(n) \leq c'f(n).$$

Ordres de grandeur des temps d'exécution d'un problème de taille  $10^6$  sur un ordinateur à un milliard d'opérations par seconde.

Ordre	Nom	Temps pour $n = 10^6$
1	Temps constant	$1ns$
$\log n$	logarithmique	$10ns$
$n$	linéaire	$1ms$
$n^2$	quadratique	$1/4h$
$n^k$	polynomiale	30 ans si $k = 3$
$2^n$	exponentielle	$> 10^{300000}$ milliards d'années



## IV Différentes nuances de complexité

- Complexité au pire : Notre modèle de complexité
- Complexité dans le meilleurs cas : Modèle de complexité peu utile
- Complexité en moyenne : à partir d'une répartition probabiliste des tailles de données, tente d'évaluer le temps moyen que l'on peut attendre de l'évaluation d'un algorithme sur une donnée d'une certaine taille.
- Complexité en espace : On ne s'occupe plus du temps d'exécution mais de la mémoire utile pour exécuter un algorithme.