Complexité d'un algorithme

Algorithmique

Le calcul de la complexité d'un algorithme permet de mesurer sa performance. Il existe deux types de complexité :

- La complexité spatiale : permet de quantifier l'utilisation de la mémoire
- La complexité temporelle : permet de quantifier la vitesse d'exécution

Complexité temporelle

L'objectif d'un calcul de complexité algorithmique temporelle est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème. Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le plus optimal.

Pour des données volumineuses la différence entre les durées d'exécution de deux algorithmes ayant la même finalité peut être de l'ordre de plusieurs jours !

Réaliser un calcul de complexité en temps revient à compter le nombre d'opérations élémentaires (affectation, calcul arithmétique ou logique, comparaison…) effectuées par l'algorithme.

Puisqu'il s'agit seulement de comparer des algorithmes, les règles de ce calcul doivent être indépendantes :

- Du langage de programmation utilisé ;
- Du processeur de l'ordinateur sur lequel sera exécuté le code ;
- De l'éventuel compilateur employé.

Par soucis de simplicité, on fera l'hypothèse que toutes les opérations élémentaires sont à égalité de coût, soit 1 « unité » de temps.

Exemple : a = b * 3 : 1 multiplication + 1 affectation = 2 « unités »

La complexité en temps d'un algorithme sera exprimée par une fonction, notée T (pour Time), qui dépend :

- De la taille des données passées en paramètres : plus ces données seront volumineuses, plus il faudra d'opérations élémentaires pour les traiter.
- On notera n le nombre de données à traiter.
- De la donnée en elle-même, de la façon dont sont réparties les différentes valeurs qui la constituent.

Par exemple, si on effectue une recherche séquentielle d'un élément dans une liste non triée, on parcourt un par un les éléments jusqu'à trouver, ou pas, celui recherché. Ce parcours peut s'arrêter dès le début si le premier élément est « le bon ». Mais on peut également être amené à parcourir la liste en entier si l'élément cherché est en dernière position, ou même n'y figure pas.

Cette remarque nous conduit à préciser un peu notre définition de la complexité en temps. En toute rigueur, on peut en effet distinguer deux formes de complexité en temps :

- La complexité dans le meilleur des cas : c'est la situation la plus favorable, par exemple : recherche d'un élément situé à la première position d'une liste
- La complexité dans le pire des cas : c'est la situation la plus défavorable, par exemple : recherche d'un élément dans une liste alors qu'il n'y figure pas

On calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

Ordre de grandeur

Pour comparer des algorithmes, il n'est pas nécessaire d'utiliser la fonction T, mais seulement l'ordre de grandeur asymptotique, noté O (« grand O »).

Une fonction T(n) est en O(f(n)) (« en grand O de f(n) ») si :

$$\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}, \forall\ n \geq n_0 : |T(n)| \leq c\ |f(n)|$$

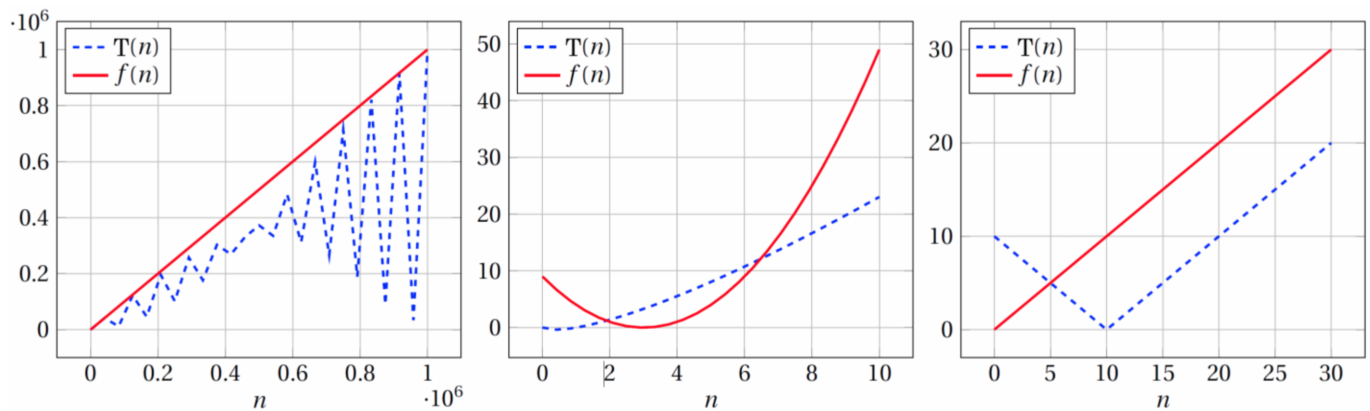
Autrement dit :

T(n) est en O(f(n)) s'il existe un seuil n0 à partir duquel la fonction T est toujours dominée par la fonction f, à une constante multiplicative fixée c près.

Exemples :

$T1(n)=7=O(1)$
$T2(n)=12n+5=O(n)$
$T3(n)=4n^2+2n+6=O(n^2)$
$T4(n)=2+(n-1)\times 5=O(n)$

Exemple visuel de la différence entre la complexité et l'ordre de grandeur de diverses fonctions



Ordre de grandeur d'une fonction récursive

Une des fonctions récursives la plus simple est celle qui permet de calculer n! (la factorielle de n), c'est-à-dire $2\times 3\times \cdots \times n$

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n*factorielle(n-1)
```

Il y a un test pour commencer (sur n). Si le test est vrai, on s'arrête, sinon on effectue une opération (un produit) en faisant appel à la même fonction. Ainsi, si n est non nul, il y a 2 opérations élémentaires + le nombre d'opérations élémentaires de la fonction au rang n – 1.

Si T_n représente la complexité de la fonction factorielle(n) alors :

$$T_n=T_{n-1} + 2,\ \ T_0=1$$

La complexité est donc donnée par une suite arithmético-géométrique. C'est toujours le cas pour les fonctions récursives.

On peut alors montrer que $T_n=2n+1=O(n)$

Classes de complexité

Ordre de complexité	Exemples	Type de complexité
O(1)	Ici la complexité ne dépend pas des données. Accès à une cellule d'un tableau.	Constante
$O(\log_k(n))$	Algorithme divisant le problème par une constante k. $O(\log_2(n))$ pour la recherche dichotomique (k=2).	Logarithmique
O(n)	Parcours de liste.	Linéaire
$O(n.\log_k(n))$	Algorithme divisant le problème en nombre de sous-problèmes constants (k), dont les résultats sont réutilisés par recombinaison (ex : Tri fusion).	Quasi-linéaire
$O(n^2)$	Algorithme traitant généralement de couples de données (boucles imbriquées). Parcours d'une matrice de pixels.	Quadratique
$O(n^3)$	Algorithme naïf de multiplication matricielle.	Cubique
$O(2^n)$	Algorithme en force brute pour le problème du sac à dos	Exponentielle
O(n!)	Problème du voyageur de commerce avec une approche naïve	Factorielle

Exemple d'évolution du temps d'exécution en fonction de la complexité de l'algorithme et pour plusieurs tailles de paquets de données.

	1	$\log_2(n)$	**n**	$n.\log_2(n)$	n^2	n^3	2^n
$n = 10^2$	1µs	6µs	0.1ms	0.6ms	10ms	1s	4×10^{16}a
$n = 10^3$	1µs	10µs	1ms	10ms	1s	16.6min	∞
$n = 10^4$	1µs	13µs	10ms	0.1s	100s	11,5j	∞
$n = 10^5$	1µs	17µs	0.1s	1.6s	2.7h	32a	∞
$n = 10^6$	1µs	20µs	1s	19.9s	11,5j	32×10^3a	∞

Exemple de calcul de complexité T_n pour l'algorithme non-récursif de calcul de la factorielle implémenté en python.

```
def factorielle(n):
    fact = 1
    i = 2
    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

```
affectation : 1
affectation : 1

itérations : au plus n − 1
   comparaison : 1
   multiplication + affectation : 2
   addition + affectation : 2
```

T_n = 1+1+ (n-1).(1+2+2) = 2 + 5n - 5 = 5n – 3

Cette fonction a donc une complexité en O(n). À noter que, comme vu précédemment, sa variante récursive est aussi en O(n).

Exercice 1

Donner la complexité de la fonction suivante :

```
def fonction_2(n):
    for i in range(n):
        print("Bonjour!")
```

Exercice 2

Montrer que la complexité de la fonction suivante est égale à 5.

```
def maFonction(n):
    if n%3 == 0:
        p = n/3 + 2
    else:
        p = n*2 + 1
    return p
```

Exercice 3

Calculer la complexité de la fonction somme définie dans ce programme :

```
def somme(n):
    s = 0
    for i in range(n+1):
        s += i
    return s

print(somme(100))
```

Exercice 4

Calculer la complexité de la fonction mystere du programme suivant :

```
def mystere(n):
    m = 0
    for i in range(n):
        for j in range(i):
            m += i+j
    return m

print(mystere(100))
```

Exercice 5

Déterminer la complexité de la fonction suivante :

```
def fibo(n):
    if n<2:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Exercice 6

Calculer la complexité de la fonction suivante :

```
def triSelection(l):
    for i in range(len(l)-1):
        indMini=i
        for j in range(i+1,len(l)):
            if l[j]<l[indMini]:
                indMini=j
        l[i],l[indMini]=l[indMini],l[i]
```

Vidéo de Rachid Guerraoui au sujet de la complexité : https://www.youtube.com/watch?v=exaHKrP6RsA
Exercices, en partie inspirés de https://www.mathweb.fr/euclide/2019/03/19/initiation-a-la-complexite-algorithmique/
Document issu de https://info.blaisepascal.fr/nsi-complexite-dun-algorithme sous licence CC-BY-SA.