Complexité des algorithmes

Introduction

L'étude de la complexité des algorithmes s'attache à mesurer leur efficacité. Lorsqu'on s'intéresse au temps d'exécution on parle de **complexité temporelle** et lorsqu'il s'agit de la mémoire utilisée, on parle de **complexité spatiale.**

I Méthodologie de l'évaluation de la complexité

Pour répondre de manière exacte à la question « quelle sera la durée d'exécution de cet algorithme ? », il faudrait connaître très précisément le temps d'exécution de chaque instruction en fonction du contexte (par exemple : le produit de 12×7 est certainement plus rapide que celui de deux nombres comportant plus de 1000 chiffres). Cela se révèle rapidement impossible et ça ne sert pas à grand chose. Un point de vue plus réaliste est de ne compter que les opérations dont on a déterminé *a priori* qu'elles domineraient le temps de calcul (par exemple les multiplications, ou les comparaisons).

Exercice 1 Pour l'algorithme suivant, déterminer le nombre de multiplications effectuées en fonction de n.

```
1 def puissance(x, n):
2     p = 1
3     for k in range(n):
4         p *= x
5     return p
```

1

En toute rigueur, la complexité temporelle d'un algorithme est donnée par une formule mathématique qui donne le nombre $f(n)$ d'opérations **élémentaires** effectuées en fonction de la taille n des données (qui est typiquement la longueur d'une liste).

Cependant ce point de vue n'est pas toujours praticable et n'est pas pertinent car d'une machine à l'autre on aura un coefficient de proportionnalité entre les temps d'exécution. Ce qui compte, c'est de savoir que le nombres d'opérations élémentaires ne s'accroît pas trop vite lorsque n devient grand.

Définition (Ordre de complexité)

On dit qu'un algorithme est d'une compléxité de l'ordre de $f(n)$ si il existe une constante positive K telle que, quelle que soit la taille n de l'entrée, le nombre d'opérations élémentaires est plus petit que $K \times f(n)$.
On dit alors que l'algorithme est en $\mathcal{O}(f(n))$

En pratique, on ne rencontrera qu'un petit nombre de complexités dont on peut faire la liste de la plus petite (algorithme rapide) à la plus grande (algorithme très lent) :

$\mathcal{O}(1)$: Complexité constante.

Le temps d'exécution est indépendant de n.

Exemples : accéder à un élément d'une liste de longueur n, ajouter un élément en fin de liste (méthode `.append()`).

$\mathcal{O}(\ln(n))$: Complexité logarithmique.

Le temps d'exécution est augmenté d'une quantité constante lorsque la taille de l'entrée est doublée.

Exemples : Recherche dichotomique dans une liste triée. Algorithme d'exponentiation rapide.

$\mathcal{O}(n)$: Complexité linéaire.

Le temps d'exécution est proportionnel à la taille de l'entrée.

Exemples : Calcul de la somme des éléments d'une liste. Recherche d'un élément dans une liste non triée (recherche séquentielle) dans le pire des cas (l'élément est en fin de liste).

$\mathcal{O}(n\ln(n))$: Complexité log-linéaire.

Le temps d'exécution n'est pas proportionnel à la taille de l'entrée mais la c'est à peine moins bien, on parle parfois de complexité quasi-linéaire.

Exemple : Le tri fusion.

$\mathcal{O}(n^2)$: Complexité quadratique.

Le temps d'exécution est multiplié par 4 lorsque la taille de l'entrée est doublée. C'est le cas des algorithmes qui sont construit avec deux boucles imbriquées.

Exemples : Le tri par sélection. Le tri par insertion.

$\mathcal{O}(n^k)$: Complexité polynomiale.

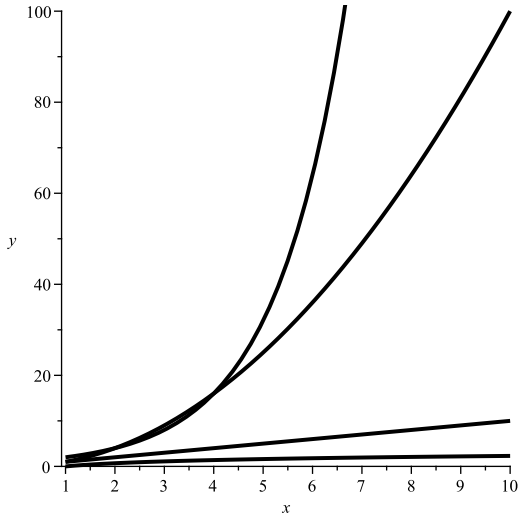
Le temps d'exécution est majorée par une expression polynomiale en n. Plus k est grand plus l'algorithme sera lent.

Exemple : Le calcul du produit de deux matrices de taille n est en $\mathcal{O}(n^3)$

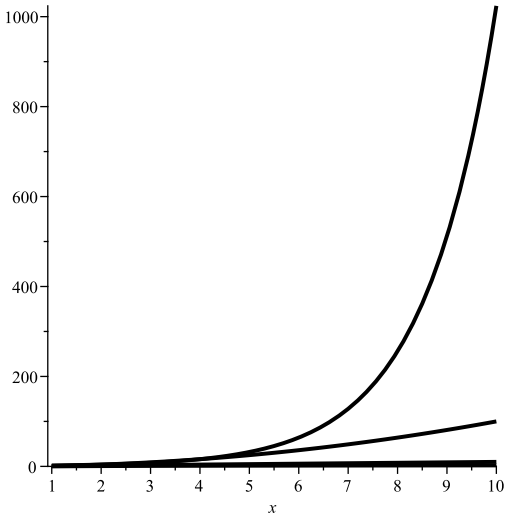
$\mathcal{O}(k^n)$: Complexité exponentielle.

Le temps d'exécution croit très rapidement. Ces algorithmes sont impraticables sauf pour des données de petites tailles. Pour résoudre certains problèmes, on ne sait parfois pas faire mieux.

Exemple : Problème du voyageur de commerce.



graphe de $\ln(x), x, x^2, 2^x$



idem mais en changeant l'échelle

D'un point de vue pratique, pour un processeur capable d'effectuer un million d'instructions élémentaires par seconde :

	n	$n\log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n=10$	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	4 s
$n=30$	<1 s	<1 s	<1 s	<1 s	<1 s	18 m	10^{25} a
$n=50$	<1 s	<1 s	<1 s	<1 s	11 m	36 a	∞
$n=10^2$	<1 s	<1 s	<1 s	1s	12.9 a	10^{17} a	∞
$n=10^3$	<1 s	<1 s	1s	18 m	∞	∞	∞
$n=10^4$	<1 s	<1 s	2 m	12 h	∞	∞	∞
$n=10^5$	<1 s	2 s	3 h	32 a	∞	∞	∞
$n=10^6$	1s	20s	12 j	31710 a	∞	∞	∞

- Notations: ∞ = le temps dépasse 10^{25} années, s= seconde, m= minute, h = heure, a = an.

Estimation empirique de la complexité en temps :

— Si un algorithme s'exécute en temps proportionnel à n alors, lorsque l'on multiplie la taille des données par 10, le temps de calcul est également multiplié par 10.

— Pour un algorithme quadratique, le temps de calcul est multiplié par 100.

— Pour un algorithme exponentiel, il est élevé à la puissance 10.

— Pour un algorithme logarithmique, on ajoute un temps constant au temps de calcul.

On remarquera que le facteur a n'intervient pas dans cette analyse ce qui explique que l'on donne les complexités avec un O.

Tout ceci permet souvent de mesurer empiriquement une complexité. Pour vérifier une estimation de complexité, on peut diviser le temps d'exécution par la complexité conjecturée et vérifier que le quotient évolue peu en fonction de la taille.

Exercice 2

On considère la fonction suivante :

```
def enum_binaire(n):
    L =[[]]
    for k in range (n):
        M = []
        for B in L:
            B0 = B + [0]
            B1 = B + [1]
            M.append(B0)
            M.append(B1)
        L = M
    return L
```

À l'aide d'une démarche empirique estimer sa complexité temporelle et spatiale.
Justifier ensuite, par une analyse du code, les résultats obtenus.

II Calculs de complexité en Python

Opérations sur les types usuels

- **Flottants :** toutes les opérations de base sur les flottants sont en temps constant (sauf élever à une puissance entière, temps logarithmique en la puissance).
- **Entiers :** toutes les opérations de base (sauf élever à une puissance entière) sont en temps constant si les entiers ont une taille raisonnable (jusqu'à 10^{20} environ). Sinon c'est plus compliqué.
- **Listes :**
  - `t[i] = x` ou `x = t[i]` : temps constant.
  - `t.append(x)` : temps constant.
  - `t = u + v` : temps proportionnel à `len(u) + len(v)`.
  - `u = t[ : ]` (copie) : temps proportionnel à `len(t)`.
  - `u = t` : temps constant (mais ce n'est pas une copie, bien sûr).
  - `x in t` (qui renvoie `True` si l'un des éléments de `t` vaut `x`, `False` sinon) : temps proportionnel à `len(t)` (dans le pire des cas).

Boucles `for`

Le nombre d'opérations effectué au total dans une boucle `for` est la somme des nombres d'opérations à chaque itération. On veillera à bien distinguer les boucles successives des boucles imbriquées.

Boucles `while`

Le cas des boucles `while` est similaire à celui des boucles `for`, sauf qu'il est plus délicat de déterminer combien de fois on passe dans la boucle. Notez qu'on a le même problème avec une boucle `for` contenant un `return` ou un `break`.

Recommandation

Dans les questions de devoir, lorsqu'il est demandé de déterminer la complexité de l'algorithme que vous proposez, il est préférable de n'utiliser que des opérations élémentaires.

Exercice 3

Calculer la complexité de l'algorithme suivant :

```
def estParfait(n):
    S = 0
    for k in range(1, n):
        if n%k == 0:
            S += k
    return S == n
```

2

La complexité spatiale

Lorsque l'on veux évaluer la complexité spatiale d'un algorithme tout se passe de la même façon mais on compte cette fois la quantité de mémoire de travail utilisée par l'algorithme (sans compter la taille des données ni celle du résultat). Dans les exemples précédents, la taille de la mémoire occupée est de 1 ou 2 variables, ce qui ne pose aucun problème. Mais parfois la mémoire utilisée peut être importante.

Exercice 4 Rappeler l'algorithme de tri par insertion. Déterminer les complexités temporelle et spatiale.

Exercice 5

`t1` et `t2` sont deux tableaux de nombres de même longueur n. On donne la fonction suivante :

```
def somme_produits(t1, t2):
    somme = 0
    for x in t1:
        for y in t2:
            somme = somme + x*y
    return somme
```

Déterminer la complexité de cette fonction.

Proposer une fonction de complexité moindre qui fournit le même résultat.

III Complexité dans le pire / meilleur cas

Voici une fonction permettant de tester si les éléments d'une liste sont deux à deux distincts.

```
def distincts(L):
    n = len(L)
    for i in range(n-1):
        for j in range(i+1, n):
            if L[i] == L[j]:
                return False
    return True
```

L'algorithme peut s'arrêter très vite ou pas. On parle dans ce cas de complexité dans le **meilleur** ou le **pire** des cas. On tente en général de calculer les deux et de déterminer quelles sont les données qui les réalisent.

Exercice 6

Calculer la complexité dans le pire et dans le meilleur des cas.

Exercice 7

Voici une fonction permettant de tester si les éléments d'une liste sont deux à deux distincts lorsque la liste à tester est de longueur n et ne contient que des entiers de $[\![0, n-1]\!]$ (technique de **mémoïzation**).

```
def est_permutation(L):
    n = len(L)
    Vu = [False]*n
    for x in L:
        if Vu[x]:
            return False
        Vu[x] = True
    return True
```

Quelle est sa complexité en temps et en espace ?

Entraînement 1

Écrire une fonction permettant de tester si une liste est triée dans l'ordre croissant, calculer sa complexité dans le pire et dans le meilleur des cas et déterminer pour quelles listes ces bornes sont atteintes.

Remarques :

- Par défaut, c'esst la complexité dans le pire des cas qu'il faut estimer afin d'avoir une borne pour le temps d'exécution.

- On peut aussi estimer la compexité moyenne. C'est souvent assez difficile et ce n'est pas au programme.

Exercice 8

Rappeler le code pour la recherche dichotomique. Justifier que la complexité est logarithmique.