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.

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

1

En toute rigueur, la complexité temporelle d'un algorithme est donnée par l'expression d'une suite u_n où

u_n est le nombre d'opérations, jugées significatives, effectuées.

n est la taille des données, ce qui peut aussi bien être la longueur d'une liste que la valeur d'un entier.

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 mène aux définitions suivantes :

Définition (Comparaisons asymptotiques des suites)

Soient u et v deux suites réelles. On note :

$u = O(v)$ s'il existe $A > 0$ et $n_0 \in \mathbb{N}$ tels que pour $n \geq n_0$, $|u_n| \leq A|v_n|$.

$u = \Omega(v)$ s'il existe $A > 0$ et $n_0 \in \mathbb{N}$ tels que pour $n \geq n_0$, $|u_n| \geq A|v_n|$.

$u = \Theta(v)$ s'il existe $A, B > 0$ et $n_0 \in \mathbb{N}$ tels que pour $n \geq n_0$, $A|v_n| \leq |u_n| \leq B|v_n|$.

Remarques :

* Ici les suites seront toutes à valeurs positives, on omettra donc les valeurs absolues.
* Essentiellement, $u = O(v)$ signifie que u ne croît pas significativement plus vite que v. De même, $u = \Theta(v)$ signifie que u et v croissent sensiblement à la même vitesse ; les rapports sont bornés.
* En informatique, quand on écrit $u = O(v)$, on sous-entend sauf mention explicite que cette majoration est plus ou moins optimale (ce qui signifie en général que $u = \Theta(v)$). Autrement dit, si la complexité de votre algorithme est de l'ordre de n^2 et que vous écrivez qu'il est en $O(2^n)$, c'est correct mathématiquement mais ça n'intéresse personne et **ne rapporte donc pas de points !**

Exercice 2

Déterminer l'ordre de grandeur asymptotique des suites à l'aide d'un $O(\,)$:

1. $u_n = 4n^2 + 5n + 1000$
2. $v_n = 2n^5 + 2^n$
3. $w_n = 7\ln(n) + 3\sqrt{n}$

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

Entraînement 1

Rappeler le code permettant de faire une division euclidienne. Quelle est sa complexité ?

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 Soit u la suite définie par $u_0 = 1$ et $u_{n+1} = \sum_{k=0}^{n} (nk+1)u_k$.

```python
def suite(n):
    L = [1]
    for k in range(n):
        u = 0
        for i in range(len(L)):
            u += (k*i + 1)*L[i]
        L.append(u)
    return L[-1]
```

Calculer la complexité temporelle et la complexité spatiale de cet algorithme.

Un exemple important : l'évaluation d'un polynôme.

On se donne un polynôme $P(X) = a_0 + a_1X + \cdots + a_nX^n$ et un réel x et l'on souhaite calculer $P(x)$.

On suppose que le polynôme P est stocké via la liste de ses coefficients : $P = [a_0, a_1, \ldots, a_n]$.

Exercice 5 Voici une première version naïve. Calculer sa complexité :

```python
def evaluation(P, x):
        n = len(P)
        s = 0
        for k in range(n):
                s += P[k]*puissance(x, k)
        return s
```

3

Exercice 6 Comment peut-on améliorer simplement la complexité ?

La relation $P(X) = a_0 + X(a_1 + a_2X + \cdots + a_nX^{n-1})$ permet le calcul le plus rapide, c'est l'**algorithme de Hörner**.

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

4

Exercice 7 Calculer sa complexité exacte en nombre de multiplications :

Voici une comparaison des temps de calcul : `tps(n)` donne le temps de calcul des trois algorithmes précédents pour le polynôme $1 + X + \cdots + X^{n-1}$ et $x = 2$ et `tps2`, de même mais seulement pour les deux derniers :

```
>>> tps(1000)
(0.23, 0.0, 0.0)
>>> tps(10000)
(50.80, 0.04, 0.04)
```

```
>>> tps2(100000)
(4.50, 3.41)
>>> tps2(200000)
(17.48, 13.36)
```

Ordre de grandeur des complexités

— $O(1)$: temps constant (ne dépend pas de la taille des données).

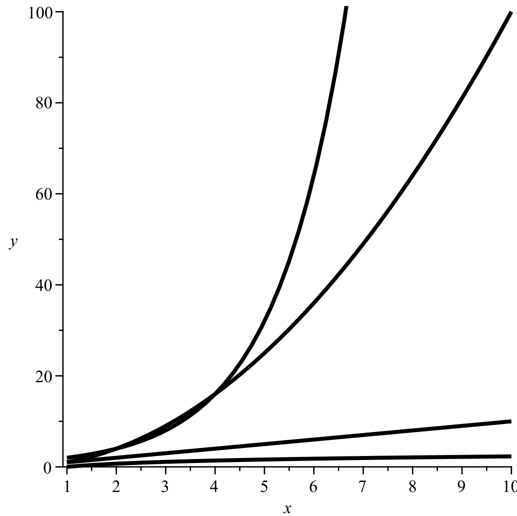
— $O(\ln n)$: complexité **logarithmique**. Temps de calcul quasi instantané.

— $O(n)$: complexité **linéaire**. Temps de calcul très rapide.

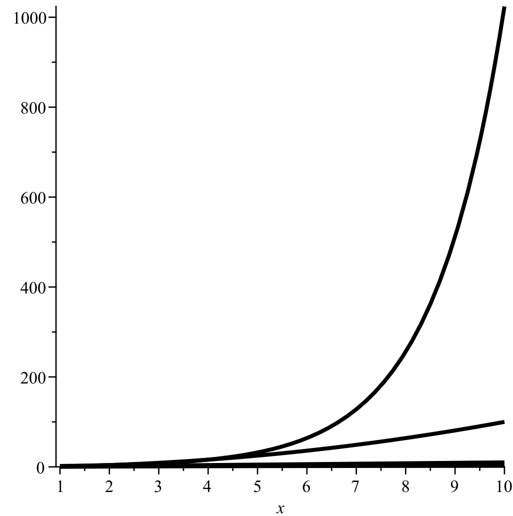
— $O(n^2)$: complexité **quadratique**. Temps de calcul satisfaisant.

— $O(n^k)$: complexité **polynomiale**. Temps de calcul correct dépendant de la valeur de k.

— $O(2^n)$: complexité **exponentielle**. Temps de calcul rédhibitoire. L'algorithme ne peut tourner que sur des données de petite taille !



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, disons an 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.

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.

III Complexité dans le pire / meilleur cas

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

```
1 def distincts(L):
2     n = len(L)
3     for i in range(n-1):
4         for j in range(i+1, n):
5             if L[i] == L[j]:
6                 return False
7     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 8

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

Exercice 9

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**).

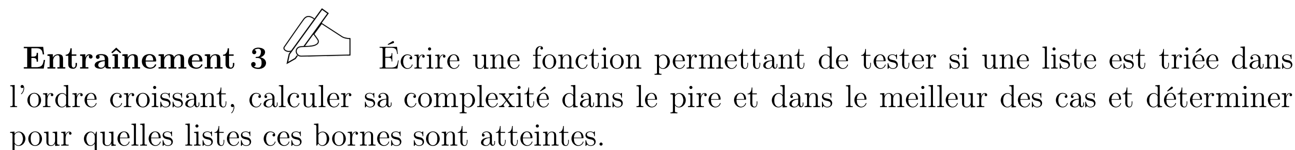
```python
def est_perm(L):
    n = len(L)
    Vu = [False]*n
    for x in L:
        if Vu[x]:
            return False
        Vu[x] = True
    return True
```

Quelle est sa complexité en temps et en espace ?

Entraînement 2 Calculer la complexité de l'algorithme suivant :

```python
def est_parfait2(n):
    ''' Déterminer si n >= 2 est parfait '''
    S = 1
    k = 2
    while k**2 < n:
```

```
            if n%k == 0:
                S += k + n//k
            k += 1
        if k**2 == n:
            S += k
        return S == n
```



Entraînement 3 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.

IV exponentiation rapide

Il est possible d'améliorer considérablement la complexité de l'algorithme de calcul des puissances d'un nombre. On remarque que :

$$x^8 = \left(\left(x^2\right)^2\right)^2$$

Ce qui donne 3 multiplications au lieu de 7!!

D'une manière générale, si $n = 2^k$ alors $x^n = x^{2^k}$ peut se calculer en faisant k multiplications (des élévations au carré). On passe donc de n multiplications à $\ln_2(n)$ multiplications ce qui est un gain considérable.

Lorsque la puissance n'est pas une puissance de deux, on peut encore utiliser une idée similaire :

$$x^{11} = x^{10}x = \left(x^5\right)^2 x = \left(x^4 x\right)^2 x = \left(\left(x^2\right)^2 x\right)^2 x = x^8 x^2 x$$

Combien faut-il alors de multiplications pour calculer x^{11} ?

Exercice 10 Faire la même chose avec x^{19} puis x^{31} et x^{32}.

Pour calculer x^n, on va donc calculer successivement $x, x^2, x^4, \ldots, x^{2^k}, \ldots$ par des élévations successives au carré et à chaque étape, on utilisera cette puissance ou pas suivant la parité de l'exposant qu'il reste à calculer.

Calcul de x^{11}

nb d'itérations	0	1	2	3	4
n	11	5	2	1	0
p	1	x	$x.x^2$	$x.x^2$	$x.x^2.x^8$
q	x	x^2	x^4	x^8	x^{16}

Calcul de x^{19}

nb d'itérations	0	1	2	3	4	5
n	19	9	4	2	1	0
p	1	x	$x.x^2$	$x.x^2$	$x.x^2$	$x.x^2.x^{16}$
q	x	x^2	x^4	x^8	x^{16}	x^{32}

```
def exp_rap(x, n):
    p = 1
    q = x
    while n != 0:
        if n % 2 == 1:
            p *= q
        q = q**2
        n = n // 2
    return p
```

Pour étudier la complexité, on commence par une approche empirique

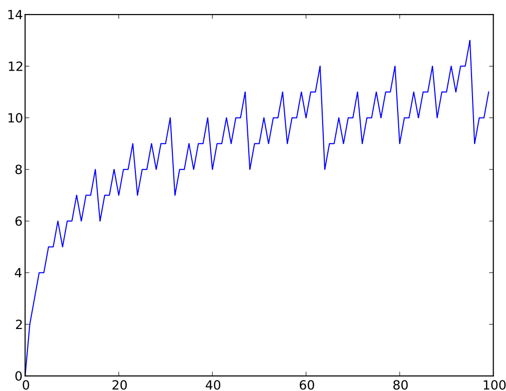
```
def complexite(n):
    compteur = 0
    p = 1
    q = 1
    while n != 0:
        if n % 2 == 1:
            p *= q
            compteur += 1
        q = q**2
        compteur += 1
        n = n // 2
    return compteur
```

```
>>> complexite(10)
6
>>> complexite(1000)
16
```

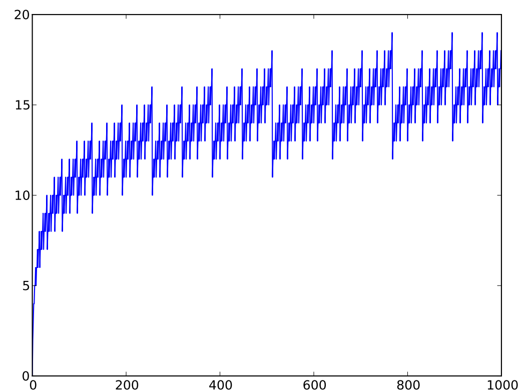
```
import matplotlib.pyplot as plt

def dessin(n):
    x = range(n)
    y = [complexite(k) for k in x]
    plt.plot(x, y)
    plt.show()
```

`dessin(100)`



`dessin(1000)`



Exercice 11 Démontrer que la complexité est logarithmique en nombre de multiplications.

Exercice 12

Modifier le code de l'exponentiation rapide pour calculer le reste de a^n modulo m.

Entraînement 4

On considère la suite de Fibonacci : $u_0 = 0, \quad u_1 = 1, \quad u_{n+2} = u_{n+1} + u_n$.

On pose $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Montrer par récurrence que pour tout $n \in \mathbb{N}$,

$$A^{n+1} = \begin{pmatrix} u_n & u_{n+1} \\ u_{n+1} & u_{n+2} \end{pmatrix}$$

En déduire un algorithme très rapide pour calculer les nombres de Fibonacci.