La complexité algorithmique

Problème

Enoncé :
Ecrire une fonction qui renvoie 1 si un nombre n appartient à une liste L ou 0 si non.

Plusieurs solutions sont possibles pour répondre à cet exercice.

- L'opérateur **in**
- La boucle **for**
- La boucle **while**
- Fonction prédéfinie **count**
- Fonction récursive
- Ect

Solutions proposés

(1)

```
def chercher(n, L):
    if n in L :
        return 1
    return 0
```

(2)

```
def chercher(n, L):
    for i in range(0, len(L)):
        if L[i]==n :
            return 1
    return 0
```

(3)

```
def chercher(n, L):
    i = 0
    while i<len(L) :
        if L[i]==n :
            return 1
        i=i+1
    return 0
```

(4

```
def chercher(n, L):
    if L= =[] :
        return 0
    if L[0]==n:
        return 1
    return chercher(n, L[1:])
```

(5)

```
def chercher(n, L):
    if L.count(n)>=1 :
        return 1
    return 0
```

(6)

```
def chercher(n, L):
    if L==[] :
        return 0
    if L[0]==n:
        return 1
    return chercher(n, L[1:])
```

Quelques questions

- Le code le plus courts: est-il meilleur ?

- Comment peut on désigner le meilleur code parmi les solutions proposées ?

Définitions

La **théorie de la complexité** est un domaine des mathématiques, et plus précisément de l'informatique théorique, qui étudie formellement la *quantité de ressources* (temps et/ou espace mémoire) nécessaire pour résoudre un *problème algorithmique* au moyen de l'exécution d'un algorithme.

Objectifs

Objectifs des calculs de complexité :

- pouvoir prévoir le temps d'exécution d'un algorithme

- pouvoir comparer deux algorithmes réalisant le même traitement

Exemples :

- si on lance le calcul de la factorielle de 100, combien de temps faudra t-il attendre le résultat?

- quel algorithme de tri vaut-il mieux utiliser pour retrier un tableau où on vient de changer un élément?

Complexité temporelle vs spatiale (1/2)

L'efficacité d'un algorithme peut être évalué en temps et en espace :

- Complexité en temps : évaluation du temps d'exécution de l'algorithme
- Complexité en espace : évaluation de l'espace mémoire occupé par l'exécution de l'algorithme

Règle (non officielle) de l'espace-temps informatique : pour gagner du temps de calcul, on doit utiliser davantage d'espace mémoire.

On s'intéresse essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)

Complexité temporelle vs spatiale (2/2)

Exemple : échange de deux valeurs entières

```
// échange des valeurs de deux variables
entier x, y, z;
... // initialisation de x et y
z ← x;
x ← y;
y ← z;
```

```
// échange des valeurs de deux variables
entier x, y;
... // initialisation de x et y
x ← y-x;
y ← y-x;
x ← y+x;
```

- la première méthode utilise une variable supplémentaire et réalise 3 affectations

- la deuxième méthode n'utilise que les deux variables dont on veut échanger les valeurs, mais réalise 3 affectations et 3 opérations

Méthodes

L'évaluation de la complexité peut se faire à plusieurs niveaux :

- au niveau de l'exécution du programme expérimentalement
- au niveau purement algorithmique, par l'analyse et le calcul

Méthode expérimentale

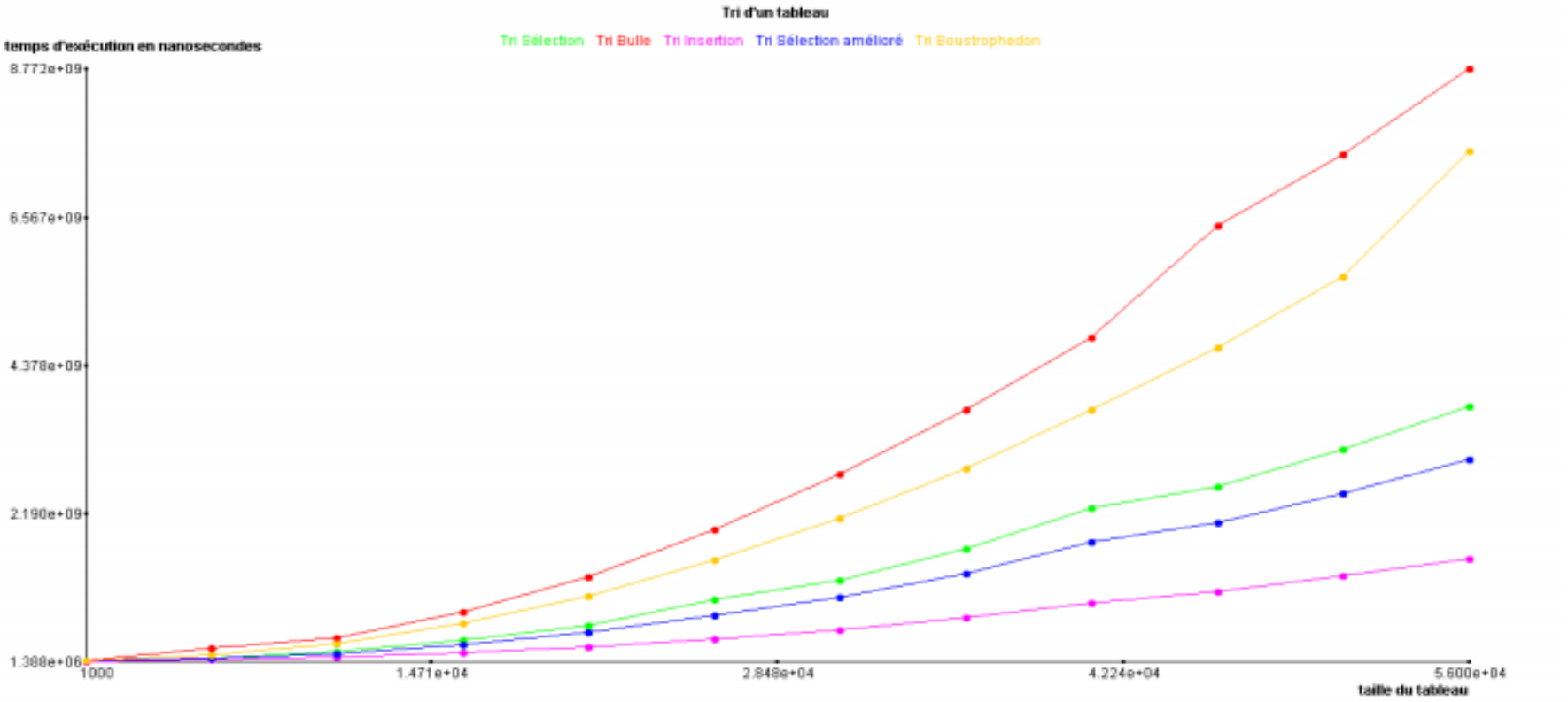
- La solution la plus intuitive pour choisir la meilleure solution consiste a mesurer le temps d'exécution exacte pour chaque algorithme.
- Pour ce faire, python offre le module time qui permet de calculer le temps d'exécution.

```
from time import clock

t0 = clock( )
tri_fusion(L)
t1 = clock( )
print(t1-t0)
```

Méthode expérimentale

Il est possible d'évaluer de façon expérimentale le temps d'exécution des programmes.



Méthode expérimentale

Inconvénients :

- Implémentation de l’algorithme
- Résultats dépends de l’environnements.
- Lancer plusieurs exécutions, qui peuvent prendre du temps

Méthode expérimentale

⇒**Environnement de simulation**

- Processeur Intel Core i5-4210U CPU 1,70GHz x 4
- Mémoire vive 8Go
- OS : Linux Ubuntu 14.04 LTS 64bits

⇒**Résultats de l'expérience**

N	10^2	10^3	10^4	10^5	10^6	10^7
Tri fusion	0,3ms	0,9ms	5,5ms	20ms	215ms	2,3s
Tri insertion	1,8ms	67ms	5,7ms	11mn	19h	66j

Méthode théorique

Avantages :

- Comparer les algorithmes théoriquement
- Pas besoin d'implémentation

Etapes de calcul de la complexité

1. Identifier le paramètre de complexité de l'algorithme
2. Calculer la complexité en fonction du paramètre de la complexité
3. Calculer la complexité asymptotique

Paramètre de complexité (1/4)

Le paramètre de la complexité est ce qui va faire varier le temps d'exécution de l'algorithme

Exemple : la calcul de la factorielle

```
def factoriel ( a ) :
        f=1
        for i in range(2, a ) :
                f = f*i
        return f
```

Le paramètre de complexité est la valeur de a
n = a

Paramètre de complexité (2/4)

Exemple : multiplication des éléments d'une liste par un nombre

```
def multiplication(L,x):
        for i in range(len(L)):
            L[i]=L[i]*x
```

Le paramètre de complexité est la valeur de len(L)
n = len(L)

Paramètre de complexité (3/4)

Exemple : somme de la i ème ligne d'une matrice

```
def somme(M, i ):
        s=0
        for j in len(M[i]):
                s = s + M[i][j]
        return s
```

Le paramètre de complexité est la valeur de len(M[0])
n = len(M[0])

Paramètre de complexité (4/4)

Exemple : table de multiplication d'un entier

```
def table1(n):
        for i in range(11):
                print(i * n)
```

Pas de paramètre de complexité

Calculer la complexité

Calculer le coût d'un algorithme

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

1. 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 du coût d'un algorithme.
2. Le coût des instructions p et q en séquence est la somme des coûts de l'instruction p et de l'instruction q.
3. 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'expression b.
4. Le coût d'une boucle **for** : le coût 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 du corps de la boucle pour chaque valeur de i.
5. Le cas des boucles **while** est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori. On peut majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison et ainsi majorer le coût de l'exécution de la boucle.

Calculer le coût d'un algorithme

Blocs d'instructions	Coût
X = a + b	
A = 0 B = 1 X = A + B	
A = X + Y + Z	
if A == B : pirnt("Egalité") else : c = A – B print(c)	
S = 0 for i in range(1, 11) : print("Donner A : ") A = input() S = S + A	

Blocs d'nstructions	Coût
for i in range(1, 11) : print("table de : " , i) for j in range(1, 11) : print(i, " x ", j , " = " , i*j)	
for i in range(1, 5) : for j in range(1, i + 1) : print("X", end= '') print("")	

Calculer le coût d'un algorithme

Programme	Coût

def table1(n):
for i in range(1,11):
print(i * n)	

def table2(n):
for i in range(n):
print(i * i)	

def table3(n):
for i in range(n):
for j in range(n):
print(i * j, end=" ")
print()	

Programme	Coût
def somme(n) : i=1 s=0 while i <= n : s=s+i i=i+1 return s	
def affiche(n) : i=1 while i <= n : print(i) i=i+2	

Nuances de la complexité temporelles

Lorsque, pour une valeur donnée du paramètre de complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer :

- La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable.
- La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (en pratique, cette complexité n'est pas très utile).
- La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution.

Nuances de la complexité temporelles

Exemple :

```
def rechercherElement(L , x) :
        i=0
        taille =  len(L)
        while i < taille :
                if L[i]==x :
                    return True
                i = i + 1
        return False
```

La compléxité au pire :
5n+3
La compléxité au mieux:
n+8
La compléxité au moyen:
3n+3

Complexité asymptotique : La notation O

Ce qui nous intéresse n'est pas un temps précis d'exécution mais un ordre de grandeur de ce temps d'exécution en fonction de la taille des données.

Première approximation : Considérer la complexité au pire des cas

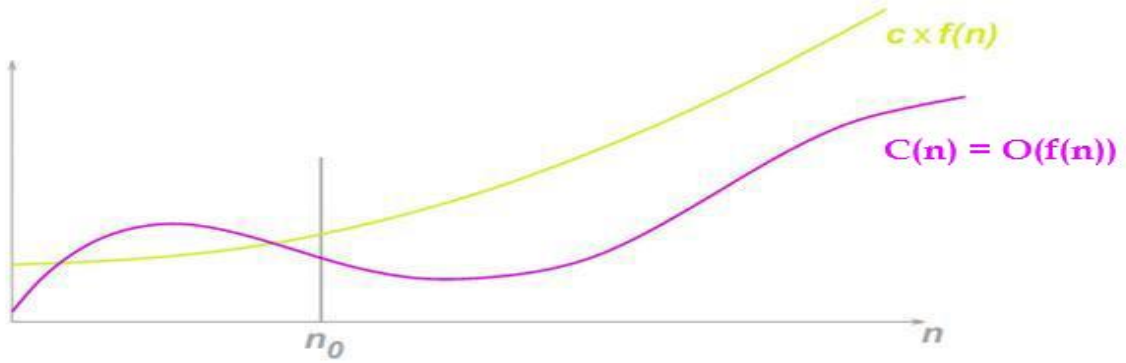
Deuxième approximation : Considérer le calcule asymptotique de la complexité, s'intéresser juste au grandes quantités des données.

Complexité asymptotique : La notation O

Soit C(n) une fonction qui désigne le temps de calcul d'un algorithme A.

On dit que C(n) est en grand O de f(n) : C(n) = O(f (n)) si et seulement

si : ∃(n0; c) telle que C(n) <= c*f (n) pour tous n >= n0



La notation O(f (n)) est aussi appelée Notation de Landau : (Complexité asymptotique) (i.e.quand n → ∞)

Notation O

Exemple 1 :

Si C(n) = 3n + 6 alors C(n) = O(n)

Démonstration :

En effet, pour n >= 2, on a 3n + 6 <= 9n ; la quantité 3n + 6 est donc bornée, à partir d'un certain rang, par le produit de n et d'une constante.

3 *2 + 6 <= 9 *2

3 *3 + 6 <= 9 *3

3 *4 + 6 <= 9 *4

3 *5 + 6 <= 9 *5

Notation O

Exemple 2 :

Si $C(n) = n^2 + 3n$ alors $T(n) = O(n^2)$

Démonstration :

En effet, pour $n >= 3$, on a $n^2 + 3n <= 2n^2$; la quantité $n^2 + 3n$ est donc bornée, à partir d'un certain rang, par le produit de n^2 et d'une constante.

Notation O : Simplification

On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :

- On oublie les constantes multiplicatives (elles valent 1) ;
- On annule les constantes additives ;
- On ne retient que les termes dominants ;

Soit un algorithme effectuant $C(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

=> On remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$

=> On annule les constantes additives : $n^3 - n^2 + n + 0$

=> On garde le terme de plus haut degré : n^3

Donc : $C(n) = O(n^3)$:

Notation O : Propriétés

$c * O(f(n)) = O(f(n))$

$O(f(n)) + O(g(n)) = O(f(n) + g(n))$

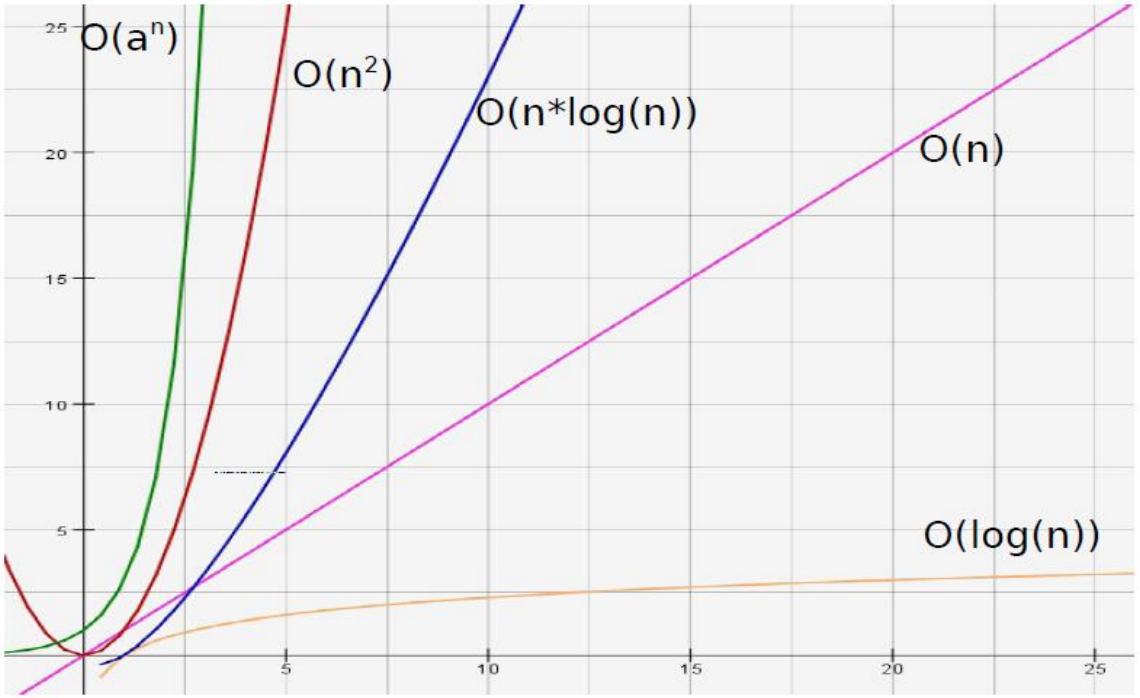
$O(f(n)) * O(g(n)) = O(f(n) * g(n))$

Classes de la complexité

On voit souvent apparaitre les complexités suivantes :

Complexité	Nom courant	Description
$O(1)$	constante	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare !
$O(log(n))$	logarithmique	augmentation très faible du temps d'exécution quand le paramètre croit.
$O(n)$	linéaire	augmentation linéraire du temps d'exécution quand le paramètre croit.
$O(nlog(n))$	quasi-linéaire	augmentation un peu supérieure à O(n)
$O(n^2)$	quadratique	quand le paramètre double, le temps d'exécution est multiplié par 4. Exemple : algorithmes avec deux boucles imbriquées.
$O(n^k)$	polynômiale	ici, n^k est le terme de plus haut degré d'un polynôme en n ; il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(k^n)$	exponentielle	quand le paramètre double, le temps d'exécution est élevé à la puissance k avec $k > 1$.
$O(n!)$	factorielle	asymptotiquement équivalente à n^n

Classes de la complexité



Exercices

```
def factoriel(n):
  f=1
  for i in range(2, n+1):
            f=f*i
  return f
```

Complexité linéaire : O(n)

Exercices

```
def minimum(L):
  min = L[0]
  For i in range(0, len(L)):
            If L[i] < min :
                      min = L[i]
  return min
```

Complexité linéaire : O(n)

Exercices

```
def table_multiplication(n):
  for i in range(1, 11):
            print(n,"x",i,"=",n*i)
```

Complexité constante : O(1)

Exercices

```
def ProduitMatriciel(A,B) :
    n=len(A)
    prod = []
    for i in range(n) :
            prod.append [[0]*n])
    for i in range(n) :
           for j in range(len(B[0])) :
                 s = 0
                 for k in range(len(B)) :
                       s= s + A[i][k] * B[k][i]
                 prod[i].append(s)
     return prod
```

Complexité polynomiale : $O(n^3)$

Exercices

```
def tri_bulle(L) :
    for i in range(len(L)-1, 0, -1 ) :
        for j in range(0, i+1):
            if L[j]>L[j+1] :
                L[j],L[j+1] = L[j+1], L[j]
    return L
```

Paramètre de complexité : n=len(L)

C(n) $= \sum_{i=1}^{n-1} \sum_{j=0}^{i} 3$

$= \sum_{i=1}^{n-1} 3(i+1)$

$= 3 \sum_{i=1}^{n-1} (i+1)$

$= 3 \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1$

= 3x((n-1)(n-2)/2) + (n-1)

= $3n^2$-2n-1

Complexité quadratique: $O(n^2)$

Exercices

```
def tri_selection(T,n) :
    for i in range(n-1) :
        posmin=i
        for j in range(i+1,n) :
            if T[j]<T[posmin] :
            posmin=j
        T[i],T[posmin]=T[posmin],T[i]
    return T
```

Paramètre de complexité : n=len(T)

C(n) $= \sum_{i=1}^{n-1}(3 + (\sum_{j=i+1}^{n-1} 2))$

$= \sum_{i=1}^{n-1} 3 + \sum_{i=1}^{n-1}(\sum_{j=1}^{n-1} 2)$

$= 3(n-1) + \sum_{i=1}^{n-1}(\sum_{j=1}^{n-i-1} 2)$

$= 3(n-1) + 2\sum_{i=1}^{n-1}(n-i-1)$

$= 3(n-1) + 2(n(n-1)-(n-1)-(n-1) = 2n^2 - 3n - 1$

Complexité quadratique: $O(n^2)$

Exercices

```
def tri_insertion(T) :
    for i in range(1,len(T)) :
        k=i
        while k>0 and T[k]<T[k-1] :
            T[k], T[k-1] = T[k-1], T[k]
            k = k – 1
```

Paramètre de complexité : n=len(T)

C(n) $= \sum_{i=1}^{n-1}(1 + (\sum_{j=1}^{i} 6))$

$= \sum_{i=1}^{n-1}(1 + 6i)$

= (n-1) + 6((n-1)(n-2)/2)

Complexité quadratique: O(n²)

Exercices

```
def RechDichotomique(T,x) :
    inf=0
    sup=len(T)-1
    while inf<=sup :
        m=(inf+sup)//2
        if T[m]==x :
            return m
        if T[m]<x :
            inf=m+1
        else :
            sup=m-1
    return -1
```

Complexité logarithmique : $O(\log_2(n))$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

```
def  factoriel(n) :
        if n==0 or n==1 :
                return 1
        return n*factoriel(n-1)
```

Le paramètre de complexité : n

C(0)=2

C(1)=2

C(n)=4 +1+ C(n-1) = 5 + C(n-1)

La résolution de la suite récurrente nous donne :

5n+2 = O(n)

Complexité des fonctions récursives

Equation : $c(n) = c(n-1) + b$
Solution : $c(n) = c(0) + b*n = O(n)$
Exemples : factorielle, recherche séquentielle récursive dans un tableau

Equation : $c(n) = a*c(n-1) + b$, $a \neq 1$
Solution : $c(n) = a^n*(c(0) - b/(1-a)) + b/(1-a) = O(a^n)$
Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif

Equation : $c(n) = c(n-1) + a*n + b$
Solution : $c(n) = c(0) + a*n*(n+1)/2 + n*b = O(n^2)$.
Exemples : traitement en coût linéaire avant l'appel récursif, tri à bulle

Equation : $c(n) = c(n/2) + b$
Solution : $c(n) = c(1) + b*\log_2(n) = O(\log(n))$
Exemples : élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive

Complexité des fonctions récursives

Equation : $c(n) = a*c(n/2) + b$, $a \neq 1$

Solution : $c(n) = n^{log2(a)} * (c(1) - b/(1-a)) + b/(1-a) = O(n^{log2(a)})$

Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique

Equation : $c(n) = c(n/2) + a*n + b$

Solution : $c(n) = O(n)$

Exemples : traitement linéaire avant l'appel récursif dichotomique

Equation : $c(n) = 2*c(n/2) + a*n + b$

Solution : $c(n) = O(n*log(n))$

Exemples : traitement linéaire avant double appel récursif dichotomique, tri fusion

Diviser pour régner : Théorème maître

En général, les algorithmes suivant le paradigme *diviser pour régner* partagent un problème de taille n en deux sous-problèmes de tailles respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$ et conduisent à une relation de récurrence de la forme :

$$\boxed{c_n = ac_{\lfloor n/2 \rfloor} + bc_{\lceil n/2 \rceil} + d_n} \qquad \text{avec } a + b \geqslant 1,$$

d_n représentant le coût du partage et de la recomposition du problème et c_n le coût total.

Théorème. — *Lorsque* $a + b > 1$, *la suite* $(d_n)_{n \in \mathbb{N}}$ *croissante et* $d_n = \Theta(n^k)$, *on a :*

$$\boxed{\begin{array}{l} \text{si } \log(a+b) < k,\ c_n = \Theta(n^k) ; \\ \text{si } \log(a+b) = k,\ c_n = \Theta(n^k \log n) ; \\ \text{si } \log(a+b) > k,\ c_n = \Theta(n^{\log(a+b)}). \end{array}}$$

Nous utiliserons désormais ce résultat pour évaluer rapidement le coût d'une méthode suivant le principe *diviser pour régner*.

Exercices

```
def fusionner(A, B):
    R = []
    while A!=[] and B!=[]:
        if A[0] < B[0] :
            R.append(A.pop(0))
        else :
            R.append(B.pop(0))
    return R + A + B

def tri_fusion(L):
    n = len(L)
    if n==0 or n==1:
        return L
    m=(n-1)//2
    return fusionner(tri _fusion(L[:m+1]),  tri _fusion(L[m+1:]))
```

Complexité quasi logarithmique : $O(n \log_2(n))$

Exercices

```
def partition(L, debut, fin):
    pivot = L[debut]
    ipivot = debut
    for i in range(debut+1, fin+1):
        if L[i]<=pivot :
            a = L.pop(i)
            L.insert(debut,a)
            ipivot = ipivot + 1
    return ipivot

def quick(L, debut, fin):
    if debut<fin :
        ipivot = partition(L, debut, fin)
        quick(L,debut,ipivot-1)
        quick(L,ipivot+1, fin)
```

Complexité quasi-log au mieux : O(n log(n))

Complexité quadratique au pire : $O(n^2)$