Complexité algorithmique I

Introduction:

pour résoudre un problème donné, on peut avoir plusieurs méthodes, c'est-à-dire plusieurs algorithmes.

- la complixité des algorithmes consiste à évaluer leur efficacité.

Exemple:

Ecrire en py thon une fonction qui prend en argument une chaîne de caractères et détermine si le caractère 'a' est présent dans la chaîne (la fonction renvoie True ou False).

méthode 1:

```
def contient1(chaine):
    for i in range(len(chaine)):
        if chaine[i]=='a':
            return True
    return False
```

méthode 2

```
def contient2(chaine):
    k=0
    n=len(chaine)
    trouve = False
    while (trouve==False and k<n):
        if chaine[k]=='a':
            trouve=True
        k+=1
    return trouve
```

méthode 3

```
def contient3(chaine):
    return ('a' in chaine)
```

Donc comment peut-on désigner le meilleur code parmi ces solutions. un algorithme.

→

- la complixité d'un problème P est une mesure de la quantité de ressource nécessaires à la résolution du problème P

- cette mesure est basée sur une estimation du nombre d'opération de base effectuées par l'algo. en fonction de la taille des donnée en entrée de l'algorithme.

types de complexité:

l'évaluation de la complexité peut se faire à 2 niveaux:

> complexité temporelle: le nombre d'opérations élémentaires (affectation, comparaison, opérations arithmétiques...) effectuées par un algorithme.

> complexité spatiale (ou espace): le nombre d'emplacements mémoires occupés lors de l'éxécution d'un programme (algorithme)

→ Nous nous intéressons uniquement à la complexité temporelle.

Les types de la complexité temporelle :

on distingue deux formes de complexité en temps :

* complexité dans le meilleur des cas : temps d'exécution minimum, dans le cas le plus favorable.

Exemple : la recherche d'un élément situé à la première position d'une liste.

* complexité dans le pire des cas : temps d'exécution maximum dans le cas le plus défavorable.

Exemple : la recherche d'un élément dans une liste alors qu'il n'y figure pas

→ la plus souvent, on utilise la complexité au pire.

Calcule de la complexité algorithmique :

Principe :

La mesure de la complexité d'un algorithme consiste à évaluer le temps d'exécution de cet algo.

Dans l'évaluation de ce temps d'exécution (le coût), on sera amené à suivre les étapes suivantes :

- Déterminer les opérations élémentaires (OE) à prend en considération : T(OE)
- calculer le nombre d'instructions effectuées par les opérations composées (OC) : T(OC)
- Donc le coût de l'algorithme (complexité) est :

$$T(alg) = T(OE) + T(OC)$$

Le coût des instructions élémentaires :

→ on appelle opération élémentaire toute :

- affectation
- test de comparaison : ==, <, <=, >, >=, !=
- opération de lecture (input) et d'écriture (print)
- opération arithmétique : +, -, *, /, %, //

↳ le coût d'une opération élémentaire est égale à 1

coût d'un bloc d'instruction :

instructions 1 # coût T1(n)
instructions 2 # coût T2(n)

le coût totale est $T(n) = T1(n) + T2(n)$

Exemple :

- algo 1 :

```
s = n+1    # instruction 1
s = s/n    # instruction 2
```

coût(algo1) = coût(inst1) + coût(inst2)

= 2 + 2 = 4

affectation + addition

affectation + division.

- Coût d'une instruction conditionnelle :

```
if (condition):
    instruction1   # coût T1(n)
else:
    instruction2   # coût T2(n)
```

Le nombre d'opération est :

coût(algo) = coût(condition) + max(T1(n), T2(n))

Exemple :

1). fonction qui calcule $(-1)^n$

```
def puissance(n):
    if n%2 == 0:
        resultat = 1
    else:
        resultat = -1
    return resultat.
```

↳ - le test (condition) comporte une opération arithmétique et une comparaison.

- chaque alternative (càd if et else) possède une affectation, ainsi le max des coût des différentes alternatives est 1

donc : $T(n) = 2+1 = 3$ ← complexité

Remarque : on applique le même principe si on a if, elif ..., elif, else càd on prend le max des alternatives.

Exemple

```
def signe(x):
    if x>0:
        print("x est positive")
    elif x<0:
        print("x est neg")
    else:
        print("x est nul")
```

T(n) = 1 + 1 = 2

- coût d'une instruction répétitive (les boucle)

- le nombre d'opérations est égale à la multiplication de nombre de répétion par la somme du coût de chaque instruction du corps de la boucle.

$coût(\text{boucle for}) = \sum coût(x_i)$

$coût(\text{boucle while}) = \sum (coût(\text{comparaison}) + coût(x_i))$

Exemple :

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

$T(n) = 1 + \sum_{k=1}^{n} 2 = 1 + 2n$

2) i = 1
While i <= n :
    somme = somme + i
    i = i + 1

$T(n) = 1 + \sum_{k=1}^{n} 1+2+2$

$= 1 + 5n$

La notion de O :

Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs, et le degré de précision qu'ils requièrent est souvent inutile, donc on fait une approximation de ce temps de calcule, cette approximation est représenté par la notation $O()$.

Règles de calcule : simplification :

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

- Les constants multiplicatives valent 1.
- on annule les constantes additives.
- on ne retient que les terme dominants.

Exemple :

soit un algorithme de complexité $T(n) = 4n^3 - 5n^2 + n + 3$

- on remplace les constants multiplicatives par 1 :

$n^3 - n^2 + n + 3$

- on annule les constantes additives

$n^3 - n^2 + n$

- on garde le terme dominant

donc $T(n) = O(n^3)$

Les classes de complexité :

le tableau suivant récapitule les complexité de référence et qui sont rangés dans l'ordre croissant ↗

complexité	type de complexité
$O(1)$	constant
$O(\log(n))$	logarithmique
$O(n)$	linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(n^k), k>1$	polynomiale

Exemples :

1). fonction permet de renvoie le quotient et le rest de la division de a par b.

```
def devision(a,b)
    q= a//b
    r= a%b
    return(q,r)
```

- le nombre d'opération est 4

donc $T(n) = O(1)$

2). fonction qui permet de calculer le produit matriciel de deux matrices carrées de même taille n.

```
def produit(A,B):
    n=len(A)
    C=[n*[0] for i in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j]=A[i][k]*
                        B[k][j]
    return C
```

- le nombre d'opérations pour calculer C[i][j] est 2.
- le nombre d'interations de la bourcle sur k, i, et j est n.

donc $T(n) = 1+1+n+\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 2$

$= 1+n+2n^3$

alors $T(n) = O(n^3)$

2) complexité de l'algorithme de la recherche séquentiel et la recherche dichotomique d'un elément dans une liste.

- recherche séquentiel :

```
def rechercheSeq(L,x):
    n=len(L)
    for i in range(n):
        if L[i]==x:
            return True
    return False
```

- le nombre de fois d'exécution de la boucle for dans le pire des cas est n:

donc $T(n) = O(n)$

recherche dichotomique:

```
def RechercheDich(L,x):
    deb=0
    fin=len(L)-1
    while deb<=fin:
        m=(deb+fin)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            deb=m+1
        else:
            fin=m-1
    return False
```

- soit k le nombre de passages dans la boucle while.
- on divise le nombre d'éléments restants par 2 jusqu'à ce qu'il n'est rest qu'un (k division)

$((((n/2)/2)/2)\ldots/2)=1$

donc $n/2^k = 1$

ainsi $k = \log(n)$

donc $T(n) = \mathcal{O}(\log(n))$

complexité des algorithmes récursifs:

le calcule de la complexité des algorithmes récursifs se déduit d'une étude de suite.

Exemple:

la fonction factorielle qui calcule le factorielle d'un nombre n.

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

- on a 2 operations, une comparaison (le test) et une produit lors de l'appel récursif:

donc $T(n) = 2 + T(n-1)$

- si $n=0$: on a seule la comparaison a lieu.

donc $T(0) = 1$

donc on a une suite récurrente définit par: $\begin{cases} T(n) = 2 + T(n-1) \\ T(0) = 1 \end{cases}$

alors $T(n) = 1 + 2n = \mathcal{O}(n)$

Exercice sur la complexité

Exo 1

Quelle est la complexité du programme suivant :

```
def algo(n):
    res = 1
    for i in range(n):
        res = res + i
    for i in range(n)
        for j in range(i):
            res = res * (i+j)

    return res.
```

Exo 2

Soient les 3 fonctions suivantes permettant de calculer la valeur d'un polynome en un point x.

$P(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$

Les coefficients du polynômes sont stockés dans une liste

$A = [a_0, a_1, a_2, \dots, a_n]$

méthode 1 :

```
def f1(A,x):
    n = len(A)
    p = A[0]
    for i in range(1,n):
        p = p + A[i] * (x**i)

    return p.
```

méthode 2 : utilise le fait que $x^i = x^{i-1} * x$

```
def f2(A,x):
    n = len(A).
    p = A[0]
    q = 1
    for i in range(1,n):
        q = q * x
        p = p + A[i] * q.

    return p.
```

méthode 3 : (utilise l'algo de Horner)

$P(x) = (\dots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3}\dots)x + a_0.$

```
def f3(A,x):
    n = len(A)
    p = A[n-1]
    for i in range(n-1, 0, -1)
        p = p * x + A[i-1]

    return p.
```

Calculer la complixité dans chaque cas.

Exo 3 (mine 2017)

On considère la fonction suivante :

```
def egale(L1,L2):
    if len(L1) != len(L2):
        return False

    for i in range(len(L1)):
        if L1[i] != L2[i]:
            return False

    return True
```

Que peut-on dire de la complexité de cette fonction.

• Complexité des algorithmes de tri :

- Tri par selection :

```
def triSelection(L):
    n=len(L)
    for i in range(0,n-1):
        m=i
        for j in range(i+1,n):
            if L[j]<L[m]:
                m=j
        L[i],L[m]=L[m],L[i]
    return L
```

$$C(N) = 1 + \sum_{i=0}^{n-2}\left(1+\sum_{j=i+1}^{n-1} 2 + 2\right)$$

$$= 1 + \sum_{i=0}^{n-2}\left(3+n-i-1\right)$$

$$= 1 + 2(n-1) + n(n-1) - \frac{(n-1)(n-2)}{2}$$

alors la complexité est : $O(N^2)$

Tri à Bulles :

```
def triBulles(L):
    n=len(L)
    while n!=0:
        for i in range(0,n-1):
            if L[i]>L[i+1]:
                L[i],L[i+1]=L[i+1],L[i]
        n=n-1
    return L
```

2 boucle imbriquées :

la complexité est : $O(N^2)$

Tri par insertion :

```
def triInsertion(L):
    n=len(L)
    for i in range(1,n):
        x=L[i]
        j=i
        while j>0 and L[j-1]>x:
            L[j]=L[j-1]
            j=j-1
        L[j]=x
    return L
```

la complexité est $O(N^2)$

Tri rapide :

```
def TriRapide(L):
    n=len(L)
    if n<=1:
        return L
    else:
        p=L[0]
        A=[]
        B=[]
        for i in range(1,n):
            if L[i]<p:
                A.append(L[i])
            else:
                B.append(L[i])
        return TriRapid(A)+[p]+TriRapid(B)
```

- la boucle for fait N-1 comparaison
- dans le pire des cas une des parties (soit A soit B) est vide et l'autre contient N-1 éléments

donc $C(N) = 3 + N - 1 + C(N-1)$

donc $C(N-1) = C(N-2) + N + 1$

⋮

$C(2) = C(N) + 4$

$C(1) = C(0) + 3$

$$\sum_{k=1}^{N} C(k) = \sum_{k=0}^{N-1} C(k) + \sum_{k=3}^{2+N} k$$

alors $C(N) = -6 + \frac{(N+2)(N+3)}{2}$

donc la complexité dans le pire des cas est: $O(N^2)$

- dans le meilleur des cas la liste est coupée en deux moitiés égale

donc $C(N) = 3 + N - 1 + 2C(N/2)$

alors $C(N) = N.\log(N)$

donc la complexité est: $O(N.\log(N))$

Tri par fusion

```
def Fusion(L):
    n = len(L):
    if n <= 1:
        return L
    else:
        L1 = Tri Fusion(L[ :n//2])
        L2 = Tri Fusion(L[n//2: ])
    return fusion(L1, L2)
```

soit : $C(N)$: coût de la fct Tri Fusion

et $f(N)$: coût de la fct Fusion

pour trier une liste de longueur N on a :

$C(N) = 2.C(N/2) + f(N)$

- dans le meilleur des cas la fonction fusion n'examine que les éléments de l'un des deux listes car ils sont tous plus petits que ceux de l'autre.

dans ce cas $f(N) = N/2$

donc $C(N) = \frac{1}{2} N.\log(N)$

alors la complexité est : $O(N.\log(N))$

- dans le pire des cas, tous les éléments sont examinés par la fonction fusion

dans ce cas $f(N) = N - 1$

donc $C(N) = N.\log(N)$

alors la complexité est:

$O(N.\log(N))$

Corrigé des exo sur la complexité

Exo1

$$T(n) = 1 + \sum_{i=0}^{n-1} 2 + \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 3$$

$$= 1 + 2n + \sum_{i=0}^{n-1} 3i$$

$$= 1 + 2n + \frac{3n(n-1)}{2}$$

donc $T(n) = O(n^2)$ quadratique

Exo2

- méthode 1 :

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

donc $T(n) = O(n^2)$ quadratique

- méthode 2 :

$$T(n) = 3 + \sum_{i=1}^{n-1} 5$$

$$= 3 + 5(n-1)$$

donc $T(n) = O(n)$ linéaire

méthode 3 :

$$T(n) = 3 + \sum_{i=1}^{n-1} 4$$

$$= 3 + 4(n-1)$$

donc $T(n) = O(n)$

Exo3

$$T(n) = 1 + \sum_{i=0}^{n-1} 1$$

$$= 1 + n$$

donc $T(n) = O(n)$. linéaire

avec $n = \text{len}(L1) \neq \text{len}(L2)$.