Chapitre 1
Complexité algorithmique

2ème année Licence Informatique

Plan

1. Introduction
2. Notion de complexité
   - Définition de la complexité d'un algorithme
   - Types de complexité
3. Complexité et notation O
   - La notation O
   - Classes de complexité les plus usuelles
   - Comparaison de temps d'exécution
4. Comment mesurer la complexité d'un algorithme
   - Le coût des instructions élémentaires/composées
   - Evaluer la compléxité d'un algorithme
   - Des exemples de calculs de complexité
   - Le module timeit
5. Différentes nuances de complexité

Introduction

Exercice

Ecrire en python une fonction qui prend en argument une chaîne de caractères et détermine si le caractère 'a' est présent dans la chaîne (on retourne soit True soit False).

- Analysons plusieurs solutions
  1. Première solution

```
def contienta1(chaine) :
    k = 0
    N = len(chaine)
    result = False
    while (result == False and k < N)
        if chaine[k] == 'a' :
            result = True
        k=k+1
    return result
```

Introduction

2. Deuxième solution :

```
def contienta2(chaine) :
    for i in range(len(chaine))
        if chaine[i] == 'a' :
            return True
    return False
```

3. Troisième solution :

```
def contienta3(chaine) :
    n = chaine.count('a')
    return bool(n)
```

4. Quatrième solution :

```
def contienta4(chaine) :
    return ('a' in chaine)
```

- Quelques questions :
  1. que remarquez vous concernant cet exercice ?
  2. Le code le plus court ! Est-il meilleur ?
  3. Comment peut-on désigner le meilleur code parmi ces quatre solutions ?

Définition de la complexité d'un algorithme

Définition

- La complexité d'un problème mathématique P est une mesure de la quantité de ressources necessaires à la résolution du problème P.
- Cette mesure est basée sur une estimation du nombre d'opérations de base effectuées par l'algorithme en fonction de la taille des données en entrée de l'algorithme.
- Généralement, pour le même problème P, on peut trouver plusieurs algorithmes $(Alg_1, Alg_2, .., Alg_n)$.
- L'objectif de la complexité est d'évaluer le coût d'exécution de chaque algorithme afin de choisir le meilleur.

.

Problème :

Comment évaluer le coût d'exécution d'un algorithme donné ?

Types de complexité

En fonction des ressources utilisées pour évaluer la complexité d'un algorithme, on sera amené à dsitinguer deux types de complexité : complexité temporelle et complexité spatiale.

Complexité temporelle (en temps)

La complexité temporelle d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme.

Complexité spatiale (en espace)

La complexité en mémoire donne le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

Remarque

Dans ce cours, nous nous intéressons uniquement à la complexité temporelle.

La notation O

Définition

- Soit T(n) une fonction qui désigne le temps de calcul d'un algorithme A.
- On dit que T(n) est en grand O de f(n) : $T(n) = O(f(n))$ si et seulement si :$\exists(n_0, c)$ telle que $T(n) <= c * f(n)\ \forall n >= n0$

.

Exemples

1. Exemple 1 :
   si $T(n) = 3n + 6$ alors $T(n) = O(n)$
   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.
2. Exemple 2 :
   si $T(n) = n^2 + 3n$ 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.

Classes de complexité les plus usuelles

On voit souvent apparaître 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 (si le paramètre double, le temps double).**
$O(nlog(n))$	**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**

Comparaison de temps d'exécution

- Prenons un ordinateur (très rapide), dont la vitesse du microprocesseur est $1GHz$ (càd il exécute un milliard d'opérations par seconde), et imaginons des algorithmes qui effectuent un traitement donné dans un temps $T(N)$
- Nous donnons ci-dessous tableau des ordres de grandeur des temps d'exécution que l'on peut espérer pour N éléments suivant la complexité de l'algorithme.

.

Complexité	**Temps d'exécution en fonction du nombre d'opérations**					
N	**5**	**10**	**15**	**20**	**100**	**1000**
$log\ N$	$3.10^{-9}s$	$4.10^{-9}s$	$5.10^{-9}s$	$7.10^{-9}s$	$10^{-8}s$	$3.10^{-9}s$
$2N$	$10^{-8}s$	$2.10^{-8}s$	$3.10^{-8}s$	$4.10^{-8}s$	$2.10^{-7}s$	$2.10^{-6}s$
$N\ log\ N$	$1,2.10^{-8}s$	$3.10^{-8}s$	$6.10^{-8}s$	$10^{-7}s$	$7.10^{-7}s$	$10^{-5}s$
N^2	$2,5.10^{-8}s$	$10^{-7}s$	$2,3.10^{-7}s$	$4.10^{-7}s$	$10^{-5}s$	$10^{-3}s$
N^5	$3.10^{-6}s$	$10^{-4}s$	$7,2.10^{-4}s$	$3.10^{-3}s$	$10s$	$11jours$
2^N	$3,2.10^{-8}s$	$10^{-6}s$	$3,3.10^{-5}s$	$10^{-3}s$	$4.10^{13}ans$	$3.10^{284}ans$
$N!$	$1,2.10^{-7}s$	$4.10^{-3}s$	$23minutes$	$77ans$	$3.10^{141}s$	$10^{500}s$
N^N	$3.10^{-6}s$	$10s$	$13ans$	$3.10^{9}ans$	$3.10^{183}ans$	$10^{300}s$

TAB.: Ordres de grandeur des temps d'exécution

Le coût des instructions élémentaires

Opération élémentaire

- On appelle opération de base, ou opération élémentaire, toute :
  1. Affectation ;
  2. Test de comparaison $:==, <, <=, >=, !=$;
  3. Opération de lecture (input) et écriture (print) ;
  4. Opération arithmétique $:+, -, *, /, \%$;
  5. Opération d'incrémentation :a=a+1,(pas 1) a=a+n (pas n) ;
  6. Opération de décrémentation :a=a-1,(pas 1) a=a-n (pas n) ;
- Le coût d'une opération élémentaire est égale à 1.

- **Exemple 1 :** Que vaut le coût de l'algorithme A

$somme = n + 1$ #instr1
$somme = somme * n$#instr2
$somme = somme/2$#instr3

Coût(A)=Coût($inst1$)+Coût($instr2$)+Coût($instr3$)=3

Le coût des instructions composées

Opération composée

- On appelle opération composée, toute instruction contenant :

1. L'exécution d'un instruction conditionnelle : Si P est une instruction conditionnelle du type Si b alors $Q1$ sinon $Q2$ finsi, le nombre d'opérations est :

   $$\text{Coût(P)} <= \text{Coût(test)} + \max(\text{Coût(Q1)}, \text{Coût(Q2)})$$

2. L'exécution d'une boucle : le temps d'une boucle est égal à la multiplication de nombre de répétition par la somme du coût de chaque instruction xi du corps de la boucle ;

   $$\text{Coût(boucle for)} = \sum \text{Coût(xi)}$$

   $$\text{Coût(boucle while)} = \sum(\text{Coût(comparaison)} + \text{Coût(xi)})$$

3. L'appel d'une fonction : Lorsqu'une fonction ou une procédure est appelée, le coût de cette fonction ou procédure est le nombre total d'opérations élémentaires engendrées par l'appel de cette fonction.

Le coût des instructions composées

- **Exemple 2 :** Que vaut le coût de l'algorithme B

```
if i%2==0 :
        n=i//2
else :
        i=i+1
        n=i//2
```

Coût(B)=Coût($i\%2 == 0$)+ max(Coût($n = i//2$),Coût($i = i + 1, n = i//2$))
=1+max(1,2)
=3

- **Exemple 3 :** Que vaut le coût de l'algorithme C

```
while i <= n :
        somme=somme+i
        i=i+1
```

Coût(C)=Coût($i <= n$)+Coût($somme = somme + i$)+Coût($i = i + 1$)
=n+n+n
=3n

Comment mesurer la compléxité d'un algorithme

Principe pour calculer la complexité d'un algorithme

- La mesure de complexité d'un algorithme consiste à évaluer le temps d'exécution de cet algorithme.
- Dans l'évaluation de ce temps d'exécution (coût), on sera amené à suivre les étapes suivantes :
  1. Déterminer les opérations élémentaires (OE) à prendre en considération : coût T(OE).
  2. Calculer le nombre d'instructions effectuées par les opérations composées (OC) : coût T(OC).
  3. Préciser les instructions à négliger.
  4. Le coût de l'algorithme T(Alg) est la somme des deux coûts T(OE) et T(OC).

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

.

Des exemples de calculs de complexité

Exemple 1 : la fonction append

- Le code ci-dessous consiste à programmer la fonction append
```
def AjoutFin(L,a) :
    L=L+[a]
    return L
```
- Exemple
$>>> L = [1, 2]$
$>>> AjoutFin(L, 3)$
$[1, 2, 3]$
- Le nombre d'opérations est : 3 (concaténation, affectation et return) ;
- Quelque soit la taille de liste, le nombre d'opérations est constant ;
- Temps de calcul est constant
- Complexité : $O(1)$

Exemple 2 : la fonction insert

- Le code ci-dessous consiste à programmer la fonction insert :
```
def AjoutElement(L,a,i) :
    L[i :i]=[a]
    return L
```
- Exemple
$>>> L = [1, 2]$
$>>> AjoutElement(L, 3, 1)$
$[1, 3, 2]$
- Le nombre d'opérations est : 2 (affectation et return) ;
- Quelque soit la taille de liste, le nombre d'opérations est constant ;
- Temps de calcul est constant
- Complexité : $O(1)$

Des exemples de calculs de complexité

Exemple 3 : boucle simple

```
for i in range(n) :
  print("Bonjour")#une instruction s
```

- Dans la boucle for,on a une seule instruction : print
- Temps de calcul de print : T_s est constant $T_s = C$
- Nombre de fois d'exécution de cette instruction est n
- Le nombre total d'opérations est $n * 1 = n$
- Temps de calcul total : $T(n) = n * T_s$
- Complexité : $O(n)$

.

Exemple 4 : remplir un tableau

```
def RemplirTab(T) :
   for i in range(len(T)) :
      print("T[",i,"]=",end="")
      T[i]=int(input())
```

- Le paramètre de complexité est la taille du tableau d'entrée T.
- On fixant i on exécute 3 opérations : print, input et affectation
- Nombre de fois d'exécution de ces 3 opérations est : $len(n)$
- Le nombre total d'opérations est $3 * len(n)$
- Complexité : $O(len(T))$

.

Des exemples de calculs de complexité

Ex. 5 : remplir une matrice

```
def RemplirMat(M,n,p) :
  for i in range(n) :
    for j in range(p) :
      print("T[",i,"][",j,"]=",end="")
      T[i][j]=int(input())
```

- Coût pour saisir une valeur est 3
- Le nombre d'itération de la boucle sur j pour i fixé égal à p
- Le nombre total pour lire toutes les valeurs pour i fixé égal à $3p$
- Le nombre total d'opérations est $\underbrace{p + p + \ldots + p}_{n \text{ fois}} = nxp$
- Complexité : $O(np)$

.

Ex. 6 : produit matriciel

```
def ProduitMatriciel(A,B) :
  prod = [[0]*len(A) for i in range(len(A))]
  for i in range(len(A)) :
    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
```

- On suppose que A et B sont deux matrices carrées d'ordre n ($len(A) = len(B) = n$)
- Coût pour calculer une valeur de s est 2 (produit et affectation)
- Le nombre d'itération de la boucle sur k pour j fixé égal à n
- Le nombre d'itération de la boucle sur j pour i fixé égal à n
- Le nombre total d'opérations est $2 + n(n(2 + n(2))$
- Complexité : $O(n^3)$

.

Des exemples de calculs de complexité

Exemple 7 Tri par sélection

```
def TriSelection(T,n) :
  for i in range(n-1) :
    Posmin=i
    for j in range(i+1,n) :
      if T[j]<T[Posmin] :
        Posmin=j
    #Permutation
    T[i],T[Posmin]=T[Posmin],T[i]
```

- Coût des échanges : le tri pas sélection sur n nombres fait $n-1$ échanges, ce quit fait $3(n-1)$ affectations
- Coût des recherches de maximum :
  - On recherche le maximum parmi n éléments : au plus $4(n-1)$ opérations. (c'est le nombre d'itération de la boucle sur j pour i fixé égal à $n-1$)
  - On recherche le maximum parmi $n-1$ éléments : au plus $4(n-2)$ opérations. (c'est le nombre d'itération de la boucle sur j pour i fixé égal à $n-2$)
  - On recherche ensuite le maximum parmi $n-2$ éléments : $4(n-3)$ tests.
  - ...
- Le nombre total de tests est :$4(1+2+3+...+(n-2)+(n-1)) = 4\frac{(n-1)n}{2}$
- Le nombre total d'opérations est :$3(n-1)+4\frac{(n-1)n}{2}$
- Donc, la complexité est quadratique $O(n^2)$

Des exemples de calculs de complexité

Ex. 8 : recherche séquentielle

```
def Recherche_Seq(T, n, x) :
  i=0
  while i<n and T[i] !=x :
      i=i+1
  if i<n :
      return 1
  else :
      return 0
```

- Le nombre d'opérations avant while est 1 (une affectation)
- La boucle while contient 2 instructions : incrémentation et affectation
- Le nombre de fois d'exécution de ces 2 instructions est n
- Le nombre d'opérations après while est 1 (un test)
- Le nombre d'opérations total est $1 + n * (1 + 1) + 1$
- Complexité : $O(n)$

.

Ex. 9 : Recherche dichotomique

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

- la complexité de chaque itération de while $O(1)$
- Nombre de fois maximum d'exécution de la boucle while est m (càd la complexité est m)
- Au pire, on divise l'intervalle de recherche par 2 à chaque étape.
- Donc : $2^{m-1} <= n <= 2^m$
- on déduit que $m <= log_2(n) + 1$
- Complexité : $O(log_2(n))$

.

Des exemples de calculs de complexité

Exemple 10 Recherche d'un mot dans une chaîne de caractères

Le principe est le même que pour la recherche d'un caractère dans une chaîne mais ici il nécessaire d'ajouter une boucle "while" pour tester tous les caractères du mot.

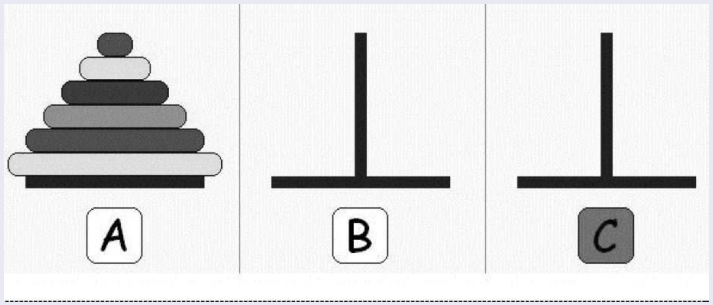
```
def searchWord(mot,texte) :
    if len(mot)>len(texte) :
        return False
    for i in range(1+len(texte)-len(mot)) :
        j=0
        while j < len(mot) and mot[j] == texte[i + j] :
            j=j+1
        if j == len(mot) :
            return i
    return None
```

- Le programme va chercher le mot à toutes les places possibles et va tester tous les caractères du mot
- Le nombre d'opérations total est de l'ordre de $m(1 + n - m)$ où m est la longueur du mot et n la longueur du texte
- En particulier le maximum est atteint pour $m = \frac{n}{2}$ et et on obtient $\frac{n^2+2n}{4}$
- Donc la complexité de l'algorithme est $O(n^2)$

Des exemples de calculs de complexité

Exemple. 11 les tours de Hanoï

```
def hanoi(n,a=1,b=2,c=3) :
  if (n > 0) :
      hanoi(n-1,a,c,b)
      print("Déplace ",a,"sur",c)
      hanoi(n-1,b,a,c)
```



- L'opération élémentaire étant le mouvement d'un plateau, pour déplacer une tour de taille $n >= 1$, il faut déplacer deux tours de taille $n-1$ et un plateau (déplacer une tour vide, c'est ne rien faire).
- La complexité vérifie donc $T(n) = 2T(n-1)+1$ avec $T(0) = 0$
- On trouve $T(n) = 2^n - 1$
- Donc la complexité est éxponentielle $O(2^n)$

Des exemples de calculs de complexité

Résolution numérique de l'équation $f(x) = 0$

Exemple. 12 : Méthode de dichotomie

```
def dich_solve(f, n, a = 0, b = 2) :
    while (b-a)/2 >= 10**(-n) :# ϵ = 10 * *(−n)
    ###### Tant que précision non atteinte
        c=(a+b)/2. # Prendre le milieu de [a, b]
        if f(c)==0 :
            return c
        if f(a)xf(c)>0 :
            a, b = c, b # dichotomie à droite
        else :
            a, b = a, c # dichotomie à gauche
    return (a+b)/2
```

- Il y a une k-ième itération si et seulement si $\frac{b-a}{2^k-1} > \epsilon$, c'est-à-dire $k < ln_2(\frac{b-a}{\epsilon})$
- La boucle sera donc exécutée exactement $[ln_2(\frac{b-a}{\epsilon}] - 1$ fois
- Pour avoir p bits(décimales) significatifs, on prend $\epsilon = \frac{1}{2^p}$
- Cette méthode approche une solution avec p bits significatifs en p étapes
- La complexité est : $O(p)$

Des exemples de calculs de complexité

Résolution numérique de l'équation $f(x) = 0$

Exemple. 13 : Méthode de Newton

```
#######Défintion de la fonction derivee
def df(f,x0,h) :
    y=(f(x0+h)-f(x0))/h
    return y
#######Définition de la fontion newton
def newton_solve(f, n, x0) :
    xn=x0
    h=10**(-n)
    while abs(f(xn)/df(f,xn,h)) >= 10**(-n) :
    ###### Tant que précision non atteinte
        xn=xn - f(xn)/df(f,xn,h)
    return xn
```

- On peut montrer que, sous certaines hypothèses, on a $|x_{n+1} - s| \leq C|x_n - s|^2$ (il suffit de voir *http : //fr.wikipedia.org/wiki/M%C3%A9thode_de_Newton*)
- A chaque itération, on double le nombre décimale exactes, on dit que la convergence est **quadratique**
- La convergence de la méthode de **Newton** est donc bine plus rapide que la **dichotomie**
- Si on veut p décimales exactes, on va avoir une complexité : $O(log_2(p))$

Des exemples de calculs de complexité

- **Exercice 1 :** Que vaut la complexité de l'algorithme suivant :

```
i=1
while i <= n :
        i=i*2
```

- **Exercice 2 :** Le code ci-dessous consiste à programmer la fonction remove. Analyser sa complexité ?

```
def SupprimerElement(L,a) :
        i=0
        if a in L :
                while L[i] !=a :
                        i=i+1 :
                L[i :i+1]=[]
        return L
```