HESTIM
ENGINEERING & BUSINESS SCHOOL

Module : Structures de données

Filière : Ingénierie Informatique & Intelligence Artificielle

Par : Youssef Ouassit

OBJECTIFS DU COURS

Objectifs :

- Comprendre et formaliser les structures de données fondamentales.
- Savoir choisir et justifier une structure adaptée à un problème.
- Évaluer la complexité (temps / mémoire) des opérations courantes.
- Implémenter et tester des structures (liste, pile, file, arbre, table de hachage, graphe).

Répartition & mode d’évaluation

Répartition du volume horaire :

- ✓ CM : 30%
- ✓ TD : 30%
- ✓ TP : 40%

Mode d'évaluation :

- ✓ TP : 40%
- ✓ Examen Final: 60%

Plan du module

- ✓ Complexité algorithmique
- ✓ Algorithmes de tri (revue) et recherche
- ✓ Pointeurs, Tableaux et Enregistrements
- ✓ Listes chainées
- ✓ Piles et files
- ✓ Arbres : Arbres binaires et arbres binaires de recherche
- ✓ Graphes
- ✓ Tables de Hachage

HESTIM
ENGINEERING & BUSINESS SCHOOL

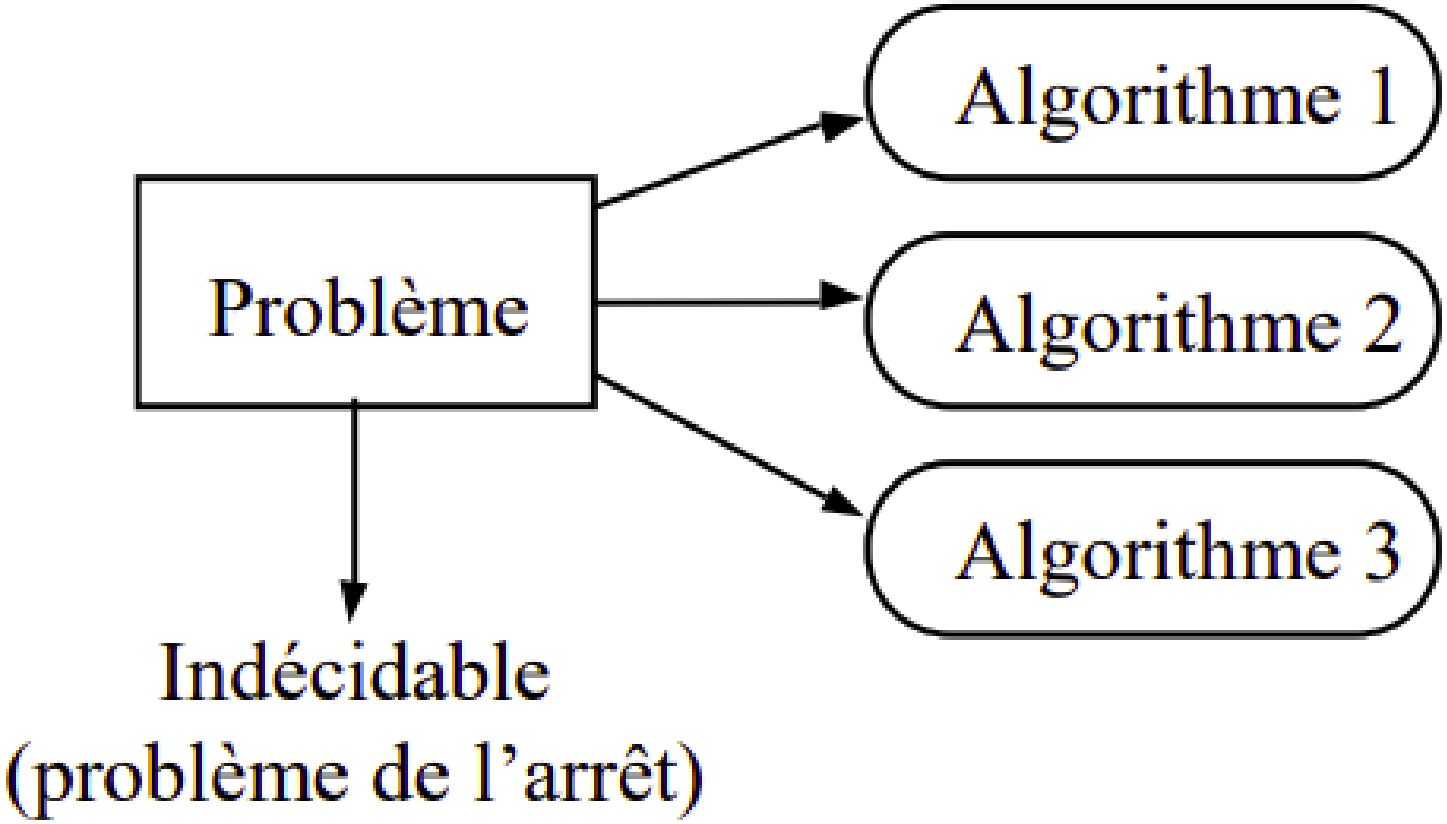
Complexité Algorithmique

Filière : Ingénierie Informatique & Intelligence Artificielle

Par : Youssef Ouassit

Plan

- Introduction
- Calcul du coût d’un algorithme
- Types de complexité
- Complexité asymptotique, Notation O
- Complexité des fonctions récursives
- Exercices


Problème
Algorithme 1
Algorithme 2
Algorithme 3
Indécidable
(problème de l'arrêt)

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
1  def fibIt(n):
2      a, b = 0, 1
3      for i in range(0, n):
4          a, b = b, a + b
5      return a
```

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
1  def fibRec(n):
2      if n==0 || n==1 :
3          return n
4      return fibRec(n-1) + fibRec(n-2)
```

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
F = [0]*50 #array to store fibonacci terms

def fibonacci_bottom_up(n):
    F[n] = 0
    F[1] = 1

    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
F = [-1]*50 #array to store fibonacci terms

def fibonacci_top_down(n):
  if (F[n] < 0):
    if (n==0):
      F[n] = 0
    elif (n == 1):
      F[n] = 1
    else:
      F[n] = fibonacci_top_down(n-1) + fibonacci_top_down(n-2)
  return F[n]
```

Quel est l'algorithme le plus efficace ?

Qu'est ce qui fait qu'un algorithme est plus efficace (critère de comparaison) ?

Comment comparer deux algorithmes ?

Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre!

La complexité d'un algorithme : est une mesure de la quantité de ressources nécessaires à son exécution. On distingue deux types de complexité :

- **Complexité temporelle :** mesure le temps de calcul consommé lors de l’exécution d’un programme
- **Complexité spatiale :** mesure le nombre d’emplacements mémoire occupés lors de l'exécution d'un programme

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

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

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;
```

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 mesuré 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 time

t0 = time( )*1000
tri_fusion(L)
t1 = time( )*1000
print(t1-t0)
```

EN PYTHON

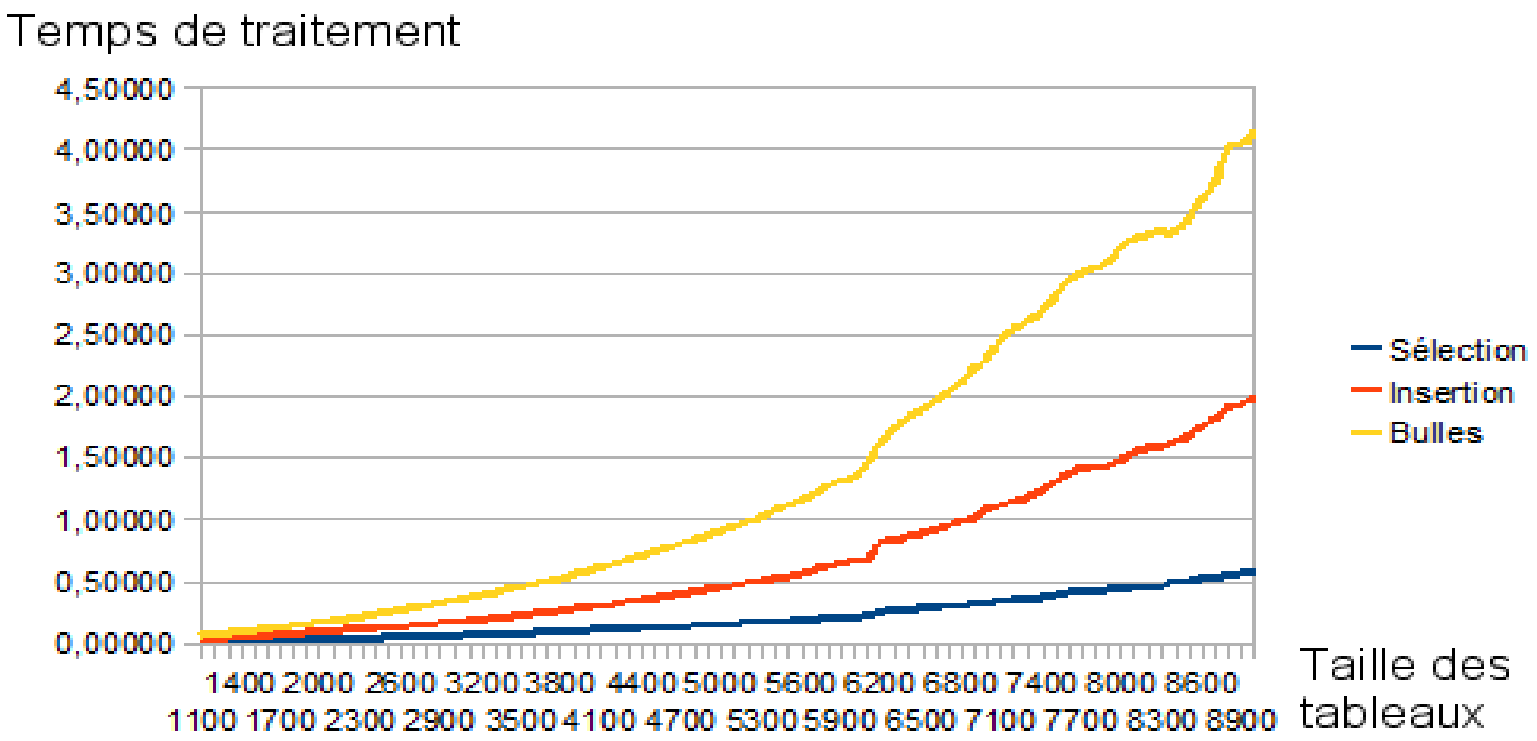
```
clock_gettime(CLOCK_MONOTONIC, &t0);
tri_fusion(L, n);
clock_gettime(CLOCK_MONOTONIC, &t1);

ms =  (t1.tv_sec - t0.tv_sec) * 1000LL +
      (t1.tv_nsec - t0.tv_nsec) / 1000000LL;
printf("%lld ms\n", ms);
```

EN Langage C

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 avec des tailles de données différents: beaucoup de 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 l’efficacité des algorithmes théoriquement
- Pas besoin d’implémentation
- Taille des exemplaires n'est pas une contrainte

Etapes de calcul de la complexité

1. Identifier le(s) paramètre(s) de complexité de l'algorithme

2. Calculer le coût en fonction du paramètre de la complexité

3. Calculer la complexité asymptotique

Paramètre de complexité (1/5)

- 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 (2/5)

- 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 (3/5)

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

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

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

Paramètre de complexité (4/5)

Exemple : table de multiplication d'un entier

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

Pas de paramètre de complexité

Paramètre de complexité (5/5)

- Exemple : produit de deux matrices

Paramètres de complexité :
l = Nombre de lignes de M
c = Nombre de colonnes de M
p= Nombre de de colonnes de N

n = max(l, p, c)

```
def produit(M, N):
    l = len(M)
    c = len(M[0])
    p = len(N[0])
    R = [[0]*l for I in range(p) ]
    for i in range(l):
        for j in range(p):
            for k in range(c):
                R[i][j] += M[i][k]*N[k][j]
    Return R
```

Un modèle pour calculer le coût d'un algorithme:

Règle 1 :

On appelle **opération de base**, ou **opération élémentaire**, toute :

- Affectation
- Test de comparaison : = =, <, ≤, ≥, !=
- Opération de lecture et écriture
- Opération arithmétique : +, -,*, /, %

Toute opération élémentaire a un coût qui égale à **1**, indépendamment

Un modèle pour calculer le coût d’un algorithme:

Règle 2 :

Le coût des instructions p et q en séquence est la somme des coûts de l’instruction p et de l’instruction q.

Un modèle pour calculer le coût d'un algorithme:

Règle 2 :

Exemple : Soit l'algorithme A

```
S=1        #Instruction 1
S=S*n      #Instruction 2
S=S-n      #Instruction 3
```

$T_A = T_{\text{instruction 1}} + T_{\text{instruction 2}} + T_{\text{instruction 3}} = 1 + 2 + 2 = 5$

Un modèle pour calculer le coût d'un algorithme:

Règle 2 :

Exemple : Soit l'algorithme A

```
A=5
B=A+5
print("S=", A+B)
```

$T_A = T_{\text{instruction 1}} + T_{\text{instruction 2}} + T_{\text{instruction 3}} = 1 + 2 + 3 = 6$

Un modèle pour calculer le coût d'un algorithme:

Règle 3 :

Le coût d'un test **if test : p else: q** est inférieur ou égal au maximum des coûts des instructions *p* et q, plus le temps d'évaluation de l'expression **test**.

$$T_A <= T_{test} + \max(T_q , T_{test})$$

Un modèle pour calculer le coût d'un algorithme:

Règle 3 :

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

$$T_A <= T_{test} + Max(T_{block\ 1} , T_{block\ 2}) = 2 + Max(2 , 4) = 6$$

Un modèle pour calculer le coût d'un algorithme:

Règle 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.

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
for i in range(a,b):
    Q
```

$$T_A = \sum_{i=a}^{b-1} T_Q$$

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
for i in range(10):
    a = int(input())
    print(a*5)
```

$$T_A = \sum_{i=0}^{9} 5 = 50$$

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
S=0
for i in range(n+1):
    S = S + 1
```

$$\mathbf{T}(\boldsymbol{n}) = \mathbf{1} + \sum_{\boldsymbol{i}=\mathbf{0}}^{\boldsymbol{n}} \boldsymbol{T_q} = \mathbf{1} + \sum_{\boldsymbol{i}=\mathbf{0}}^{\boldsymbol{n}} \mathbf{2} \quad = \mathbf{1} + \mathbf{2}(\boldsymbol{n}+\mathbf{1}) = \mathbf{2}\boldsymbol{n} + \mathbf{3}$$

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
for i in range(n):
    for j in range(i):
        S = i+j
        print(S)
```

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

Un modèle pour calculer le coût d’un algorithme:

Règle 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.

Un modèle pour calculer le coût d'un algorithme:

Règle 5 :

```
while test:
    Q
```

$$T_A = \sum(T_{test} + T_Q)$$

Un modèle pour calculer le coût d'un algorithme:

Règle 5 : Exemple

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

$$T_A = 1 + \sum_{i=1}^{n+1} 1 + \sum_{i=1}^{n} 4 = 1 + n + 1 + 4n$$

Un modèle pour calculer le coût d’un algorithme:

Règle 6 : Exemple

Lorsqu’une fonction ou une procédure est appelée, le cout cet appel est le nombre total d’opérations élémentaires et composées engendrées par l’appel de cette fonction

Le coût d'un algorithme

Exercice : Calculer le coût des instructions suivantes

Programme	Coût
`for i in range(1,11):` `  print(i * n)`	
`for i in range(n):` `  print(i * i)`	

`for i in range(n):`
`  for j in range(n):`
`  print(i * j)`
`  print()`	

Programme	Coût

`i=1`
`s=0`
`while i <= n :`
`  s=s+i`
`  i=i+1`
`return s`	

`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) :
        for i in range(0, len(L)):
                if L[i]==x :
                        return True
        return False
```

Le coût au pire :

2n+1

Le coût au mieux:

2

Nuances de la complexité temporelles

La complexité en moyenne :

La complexité en moyenne d'un algorithme est une mesure qui tient compte de la performance d'un algorithme sur un ensemble de cas d'entrée possibles, en prenant en considération les probabilités d'apparition de ces différents cas.

Complexité en moyenne = Σ (Complexité temporelle du cas i * Probabilité du cas i) pour tous les cas i

$$C_A^{moy}(n) = \sum_{d \in \mathcal{D}_{A,n}} p(d).C_A(d)$$

Nuances de la complexité temporelles

La complexité en moyenne : Exemple

```
def rechercherElement(L , x) :
        for i in range(0, len(L)):
                if L[i]==x :
                        return True
        return False
```

- **Complexité moyenne :** Soit q la probabilité que X est dans le tableau T, alors :
  - La probabilité que X se trouve dans la i^{ème} case est $\frac{q}{n}$ puisqu'il peut se trouver dans n'importe quelle case avec la même probabilité
  - La probabilité que X ne soit pas dans le tableau T est (1 - q)
- Le cout dépend des nombre de comparaisons effectuées :
  - Si X est dans la i^{ème} case de T alors on effectuera « i » comparaisons
  - Si X n'est pas dans T alors on effectuera « n » comparaisons
- Donc : $T(\mathbf{n}) = \sum_{\mathbf{i=1}}^{\mathbf{n}} \frac{\mathbf{q}}{\mathbf{n}}\,\mathbf{i} + \mathbf{n}(\mathbf{1}-\mathbf{q}) = \frac{\mathbf{q}}{\mathbf{n}}\frac{\mathbf{n(n+1)}}{\mathbf{2}} + \mathbf{n}(\mathbf{1}-\mathbf{q}) = \mathbf{q}\frac{\mathbf{(n+1)}}{\mathbf{2}} + \mathbf{n}(\mathbf{1}-\mathbf{q})$
- Si on est sur que X appartient au tableau T (q=1) alors : $T(\mathbf{n}) = \frac{\mathbf{n+1}}{\mathbf{2}}$

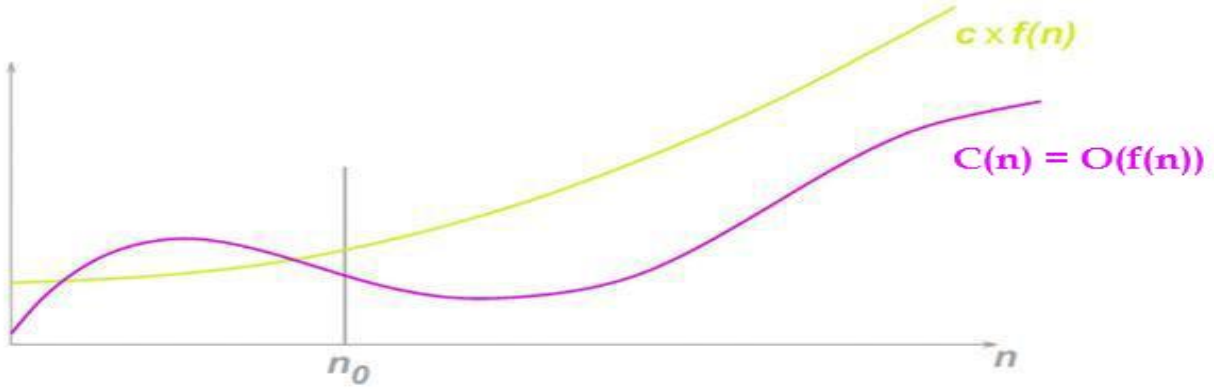
Complexité asymptotique : La notation O

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 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 : ∃(n0; c) telle que T(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 $T(n) = 3n + 6$ alors $T(n) = O(n)$

<u>Démonstration :</u>

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

<u>Démonstration :</u>

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.

La complexité asymptotique — 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 $T(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

- => On remplace les constantes multiplicatives par 1 : $n^3 - n^2 + n + 3$
- => On annule les constantes additives : $n^3 - n^2 + n$
- => On garde le terme de plus haut degré : n^3
- Donc : $T(n) = O(n^3)$:

Notation O : Propriétés

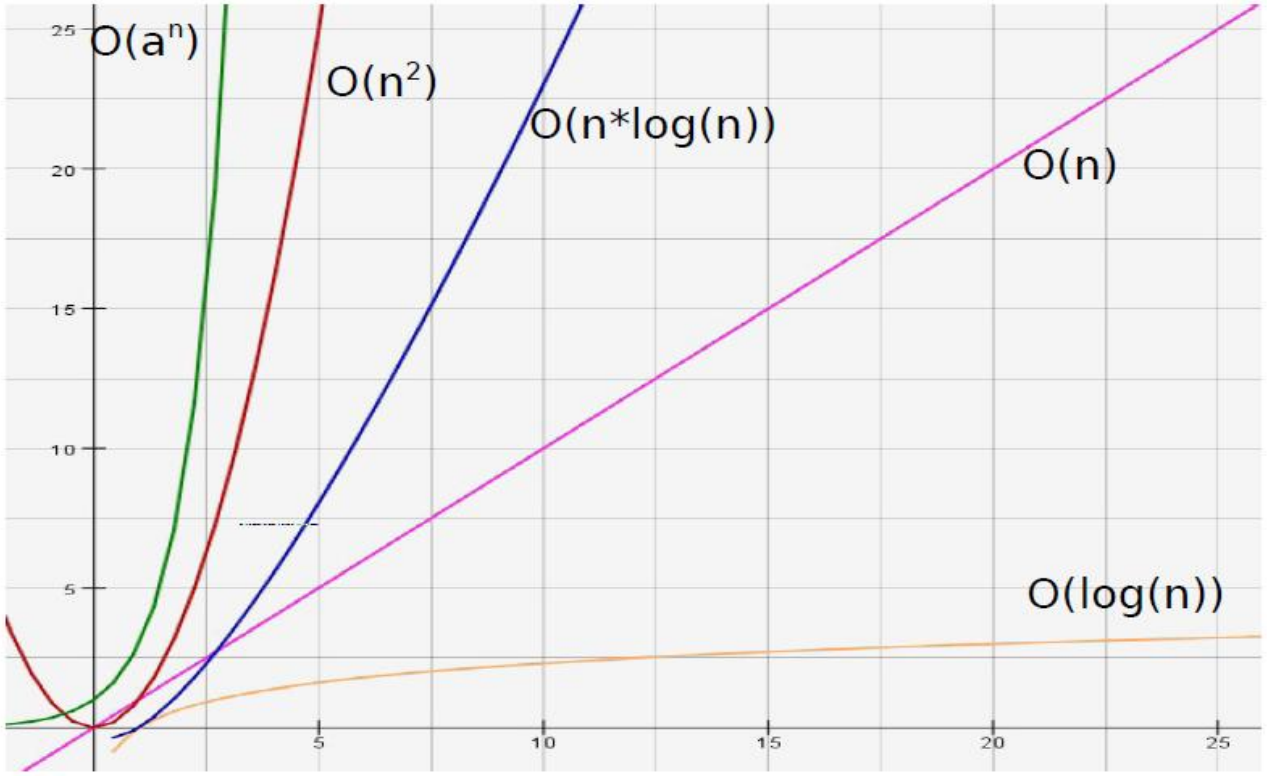
- $c * O(f) = O(f)$
- $O(f) + O(g) = O(f + g) = O(\max(f, g))$
- $O(f) * O(g) = O(f * g)$
- $f = O(g)$ et $g = O(h)$ alors $f = O(h)$ Transivité

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):                    # n est le paramètre de complexité
    f=1                              # O(1)
    for i in range(2, n+1):          # n-1 boucle donc : O(n)
        f=f*i                        # O(1)
    return f                         # O(1)
```

Complexité linéaire : O(1) + O(n)*O(1) + O(1) = O(n)

Exemples

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

n = len(L) est le paramètre de complexité

O(1)

n boucles donc : O(n)

O(1)

O(1)

O(1)

Complexité linéaire : O(n)

Exemples

```
def table_multiplication(n):          # pas de paramètre de complexité
    for i in range(1, 11):            # 10 boucle donc : O(1)
        print(n,"x",i,"=",n*i)        # O(1)
```

Complexité constante : O(1)

Exemples

Somme des éléments de la diagonale d'une matrice carrée :

```
def  sommeMat1(A):
        s=0
        n = len(A)
        for i in range(n):
                for j in range(n):
                        if i == j : s=s+A[i][j]
        return s
def  sommeMat2(A):
        n = len(A)
        s=0
        for i in range(n):
                s = s + A[i][i]
        return s
```

Complexité quadratique : $O(n^2)$

Complexité linéaire : $O(n)$

Exemples

```
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(n))

Exemples

```
def rechercheDoublans(T) :
    for i in range(0, len(T)-1):
        for j in rangr(i+1, len(T) ):
            if T[i]==T[j] :
                return True
    return False
```

Complexité logarithmique : $O(n^2)$

Au pire des cas: pas de doublons ➔Comparer tous les éléments entre eux

$$\left(\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 2\right) + 1 = 1 + 2\sum_{i=0}^{n-2}(n-1-i) = \mathbf{O}(\boldsymbol{n^2})$$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

La récursivité est le fait pour une méthode de s'appeler elle même. On parle alors de méthode récursive.

Exemple : Le calcul de la factorielle de N.

N != N*(N-1)*(N-2)*…*2*1 , on peut écrire ainsi N != N*(N-1)!

Pour calculer N! il faut savoir calculer (N-1) ! et pour calculer (N-1) ! Il faut savoir calculer (N-2) ! et ainsi jusqu'à 1! qui est égal à 1 et qui permet à la récursivité de s'arrêter après une série d'auto appels.

-> Il est impératif donc de prévoir une condition d'arrêt à la récursion sinon le programme ne s'arrête jamais!

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

Récursivité simple :

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

Récursivité multiple :

```
def combinaison(n, p) :
        if p==0 or n==p :
                return 1
        return combiaison(n-1, p) + combiaison(n-1, p-1)
```

Récursivité mutuelle :

$$pair(n) = \begin{cases} vrai & si\ n = 0 \\ impair(n-1) & sinon \end{cases}$$

$$impair(n) = \begin{cases} vrai & si\ n = 0 \\ pair(n-1) & sinon \end{cases}$$

```
def pair(n) :
        if n==0 :
                return True
        return impair(n-1)
```

```
def impair(n) :
        if n==0 :
                return True
        return pair(n-1)
```

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

Récursivité imbriquée : La fonction Ackerman

$$A(m, n) = \begin{cases} n+1 & si\ m = 0 \\ A(m-1, 1) & si\ m > 0\ et\ n = 0 \\ A\big(n-1, A(m, n-1)\big) & sinon \end{cases}$$

```
def  Ackerman(m, n) :
        if m==0 :
                return n+1
        if m>0 or n==0:
                return Ackerman(m-1, 1)
        return Ackerman(n-A, Ackerman(m, n-1))
```

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

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

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n-1) + 5, & sinon \end{cases}$$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n-1) + 5, & sinon \end{cases}$$

La complexité d'un algorithme récursif se fait par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.

T(n) = T(n-1) + b
= T(n-2) + 2*b
= T(n-3) + 3*b
= T(n-4) + 4*b
= ....
= T(n-i) + i*b
= T(n-i+1) + (i+1)*b
...
= T(n-n) + n*b = T(0) + n*b = 5n + 2 *=> donc complexité en O(n)*

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

```
def  puissance(x, n) :
        if n==0 :
                return 1
        return x*puissance(x, n-1)
```

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n-1) + 5, & sinon \end{cases}$$

$T(n) = 5n + 2$ donc complexité en $O(n)$

Complexité des fonctions récursives

```
def longueur(ch):
    if ch=="":
        return 0
    return 1+longueur(ch[1:])
```

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n-1) + 4, & sinon \end{cases}$$

$T(n) = 4n + 2$ donc complexité en $O(n)$

Complexité des fonctions récursives

```
def RechDichotomiqueRec(T, x, debut, fin) :
    m=(debut+fin)//2
    if T[m]==x :
        return m
    if debut>fin:
        return -1
    if T[m]<x :
            return RechDichotomiqueRec(T,x,m+1,fin)
    else :
            return RechDichotomiqueRec(T,x,debut,m-1)
```

$$T(n) = \begin{cases} 5, & n = 1 \\ T(n/2) + 9, & sinon \end{cases}$$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

$$T(n) = \begin{cases} 5, & n = 1 \\ T(n/2) + 9, & sinon \end{cases}$$

La complexité d'un algorithme récursif se fait par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.

$$\begin{aligned} T(n) &= T(n/2) + b \\ &= T(n/4) + 2*b \\ &= T(n/8) + 3*b \\ &= T(n/16) + 4*b \\ &= .... \\ &= T(n/2^i) + i*b \\ &... \\ &= T(1) + \log(n)*b \quad = \quad 9\log(n) + 5 \end{aligned}$$

=> donc complexité en O(log(n))

Complexité des fonctions récursives

```
def fibonacci(n):
    if n <= 1 :
        return n
return fibonacci(n - 1) + fibonacci(n - 2)
```

$$T(n) = \begin{cases} 2, & n = 0 \\ 2, & n = 1 \\ T(n-1) + T(n-2) + 5, & sinon \end{cases}$$

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

Exercices

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

Complexité quadratique: $O(n^2)$

Paramètre de complexité : n=len(L)

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} 6i + 1 \\
&= 6 \sum_{i=1}^{n-1} i + 1 \\
&= 6(n-1)(n)/2 + 1 \\
&= 3n^2 - 3n + 1
\end{aligned}
$$

Exercices

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

Complexité quadratique: $O(n^2)$

Paramètre de complexité : n=len(T)

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n-2}\left(3 + \left(\sum_{j=i+1}^{n-1} 2\right)\right) \\
&= \sum_{i=0}^{n-2} 3 + \sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} 2\right) \\
&= 3(n-1) + \sum_{i=0}^{n-2}\left(\sum_{j=1}^{n-i-1} 2\right) \\
&= 3(n-1) + 2\sum_{i=0}^{n-2}(n-i-1) \\
&= 3(n-1) + 2\left(\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1\right) \\
&= 3(n-1) + 2(n(n-1) - (n-1)(n)/2 - (n-1)) = n^2 - 5n - 2
\end{aligned}
$$

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
```

Complexité quadratique: $O(n^2)$

Paramètre de complexité : n=len(T)

$T(n) = \sum_{i=1}^{n-1}(1 + 10i)$

$T(n) = (n-1) + 10((n-1)(n)/2)$

$T(n) = 5n^2 - 4n - 6$

Complexité : Manipulation de base d'une liste

liste est une **liste** de taille N

Instruction	Exemple	Complexité
Récupérer la longueur	len(liste)	
Ajouter un élément à la fin	liste.append(5)	
Accéder à un élément	liste[20]	
Remplacer un élément	liste[30]=100	
Parcourir une liste	for a in liste : print(a)	

Complexité : Manipulation de base d'une liste

liste est une liste de taille N

Instruction	Exemple	Complexité
Supprimer le dernier élément	liste.pop(-1)	
Supprimer le premier élément	liste.pop(0)	
Comparer deux listes	if liste == liste2 : print("listes égales")	
Faire une copie	copie = liste.copy()	
Trouver le min	min(liste)	
Trouver le max	max(liste)	
Vérifier si un élément est dans la liste	if 40 in liste : print("OK")	

Complexité : Manipulation de base d'un ensemble

E est un ensemble de taille N

Instruction	Exemple	Complexité
Récupérer la longueur	len(E)	
Ajouter un élément à la fin	E.add(5)	
Supprimer une élément	E.remove(20)	
Parcourir un ensemble	for a in E : print(a)	
Comparer deux ensembles	if E==autre : print("Ensembles égaux")	

Complexité : Manipulation de base d’un ensemble

E est un ensemble de taille N

Instruction	Exemple	Complexité
Faire une copie	copie = E.copy()	
Trouver le min	min(E)	
Trouver le max	max(E)	
Vérifier si un élément est dans l’ensemble	if 40 in E : print("OK")	

Complexité : Manipulation de base d'un dictionnaire

D est un dictionnaire de taille N

Instruction	Exemple	Complexité
Récupérer la longueur	len(D)	
Ajouter un élément à la fin	D[8] = 20	
Accéder à un élément si on connaît la clé	D[8]	
Parcourir une liste	for key, value in liste.items() : print(key,'=',value)	

Complexité : Manipulation de base d'un dictionnaire

D est un dictionnaire de taille N

Instruction	Exemple	Complexité
Supprimer un élément	D.pop(key)	
Comparer deux dictionnaires	if D == D2 : print("dictio égales")	
Faire une copie	copie = D.copy()	
Trouver le min	min(D)	
Trouver le max	max(D)	
Vérifier si une clé est dans la liste	if 40 in D.keys() : print("OK")	
Vérifier si une vlauer est dans la liste	if 40 in D.values() : print("OK")	

HESTIM
ENGINEERING & BUSINESS SCHOOL

Algorithmes de Tri et de Recherche

Filière : Ingénierie Informatique & Intelligence Artificielle

Par : Youssef Ouassit