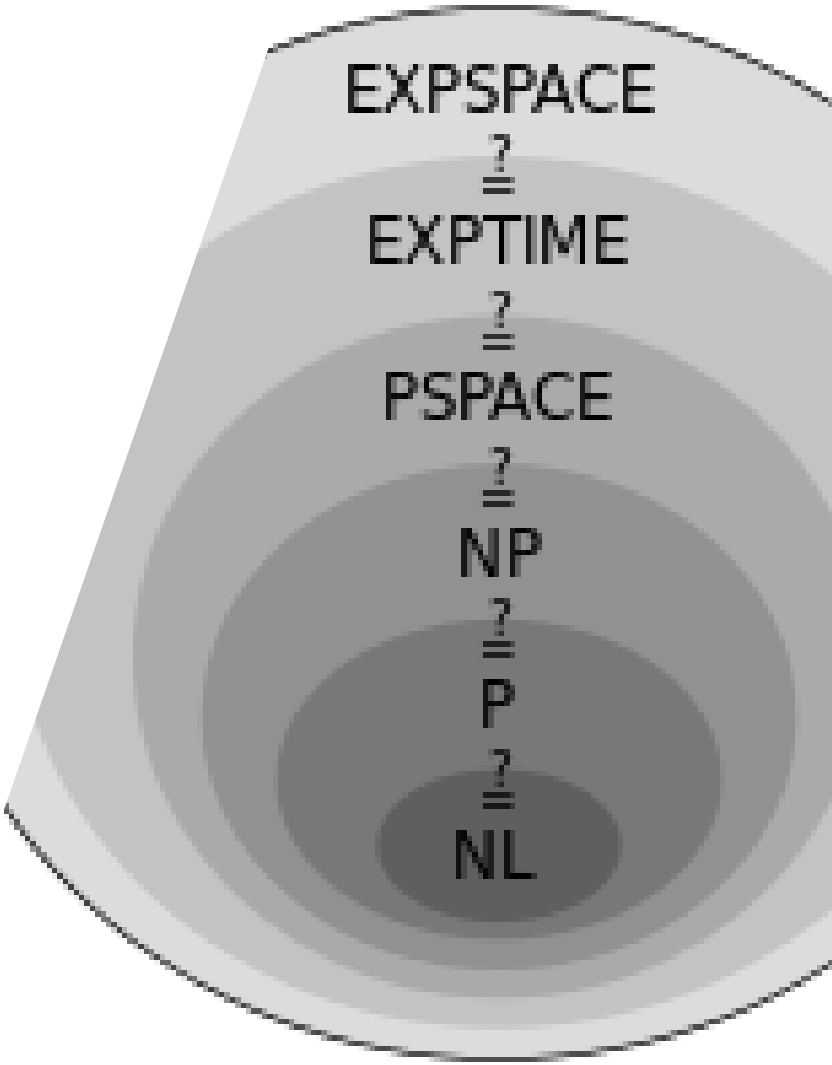
Chapitre 5:

Complexité

Algorithmique



2ème Année - Cycle Ingénieur

Algorithmique & Complexité

BENAZZOUZ 2024-2025

- La théorie de la **complexité algorithmique** s'intéresse à l'estimation de l'efficacité des algorithmes (en terme de mémoire et de temps d'exécution).

- Question : entre différents algorithmes réalisant une même tâche, quel est le plus rapide et dans quelles conditions ?
  - *exemple course de voiture !*

Pour qu'une analyse ne dépende pas de la vitesse d'exécution de la machine ni de la qualité du code produit par le compilateur, il faut utiliser comme unité de comparaison des « opérations élémentaires » en fonction de la taille des données en entrée.

Quelques exemples d'opérations élémentaires (atomiques):

- accès à une cellule mémoire ou affectation ;
- comparaison de valeurs ;
- opérations arithmétiques (sur valeurs à codage de taille fixe) ;
- opérations sur des pointeurs.

Exemple1 :

```
int Somme ( int N )
    {
        int acc = 0 , nb = 1 ;
         while ( nb < = N )
                {
                    acc = acc + nb;
                    nb = nb + 1 ;
                 }
         return acc ;
     }
```

Que fait cet

algorithme ?

Exemple1 :

```
int Somme ( int N )
   {
        int acc = 0 , nb = 1 ;
         while ( nb < = N )
                {
                     acc = acc + nb;
                     nb = nb + 1 ;
                 }
         return acc ;
    }
```

Cet algorithme demande:

- 2 affectations,
- N+1 comparaisons,
- N sommes et affectations,
- N sommes et affectations,

ce qui fait au total $5N+3$ opérations élémentaires.

Ce qui nous intéresse c'est un ordre de grandeur (cout exact, valeur asymptotique), donc : la complexité de cet algo est de l'ordre de N, ou encore linéaire en N.

Exemple2 : $Somme = \sum_{1}^{N} = 1 + 2 + 3 + \cdots + N$

Si on utilise les propriétés des séries numériques :

```
int Somme ( int N )
   {
        int acc = 0;

         acc = (N*(N+1))/2 ;

        return acc ;
    }
```

Cet algorithme demande : ?

- 2 affectations,
- 3 opérations arithmétiques

Même problème résolu en

temps constant (indépendant

des données) !

On distingue :

- la **complexité dans le pire des cas**, ou **complexité dans le cas le plus défavorable**, mesure la complexité (par exemple en temps ou en espace) d'un algorithme dans le pire des cas d'exécution possibles.
- la **complexité dans le meilleur des cas** correspond à la complexité (par exemple en temps) d'un algorithme dans le cas d'exécution le plus favorable possible.

Type de complexité algorithmique :

On considère désormais un algorithme dont le temps maximal d'exécution pour une donnée de taille n en entrée est noté T(n).

Chercher la complexité au pire – dans la situation la plus défavorable – c'est exactement exprimer T(n) en général en notation O (La notation grand O indique « l'ordre de grandeur » des fonctions.)

- **T(n) = O(1)**, temps constant : temps d'exécution indépendant de la taille des données à traiter(le cas de notre exemple2).

Type de complexité algorithmique :

- **T(n) = O(log(n))**, temps logarithmique (algo sous linéaire) : on rencontre généralement une telle complexité lorsque l'algorithme casse un gros problème en plusieurs petits, de sorte que la résolution d'un seul de ces problèmes conduit à la solution du problème initial.

Exemple : cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal *n*.

Type de complexité algorithmique :

- **T(n) = O(n)**, temps linéaire : cette complexité est généralement obtenue lorsqu'un travail en temps constant est effectué sur chaque donnée en entrée.

  Exemple : évaluation de la valeur d'une expression composée de *n* symboles.

- **T(n) = O(n.log(n))** (quasi- linéaire), l'algorithme scinde le problème en plusieurs sous problèmes plus petits qui sont résolus de manière indépendante. La résolution de l'ensemble de ces problèmes plus petits apporte la solution du problème initial.

  Exemple : les algorithmes optimaux de tri(heapsort).

Type de complexité algorithmique :

- **$T(n) = O(n^2)$**, temps quadratique : apparaît notamment lorsque l'algorithme envisage toutes les paires de données parmi les n entrées.

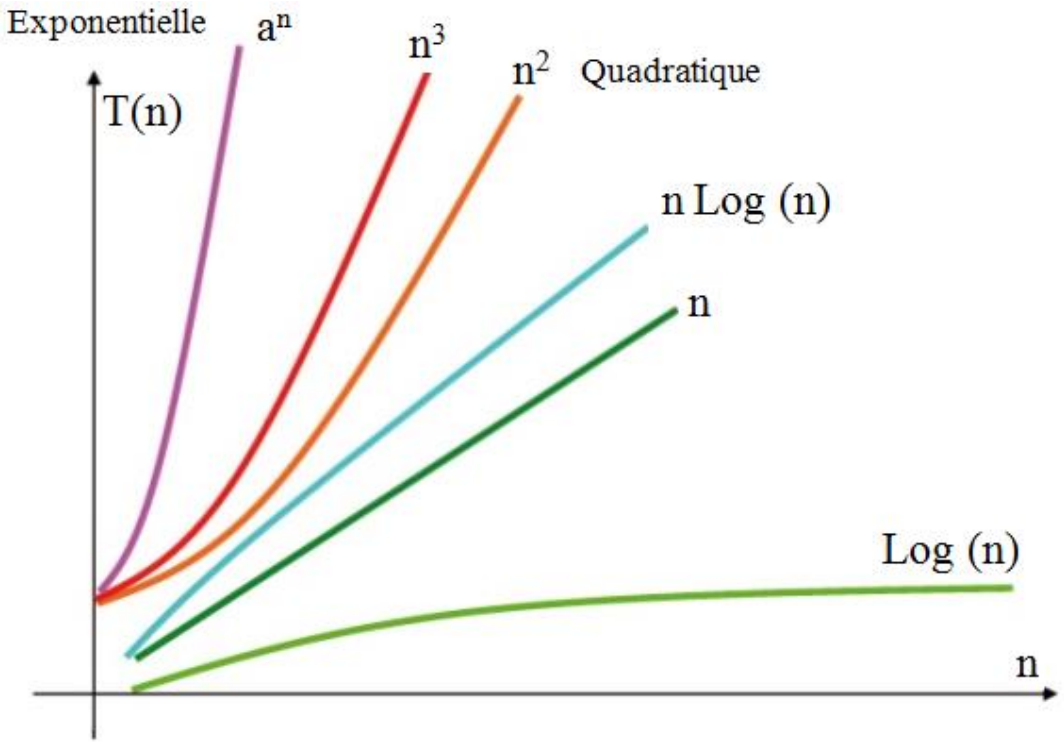
  Exemple : deux boucles imbriquées, tri naif

- **$T(n) = O(n^3)$** temps cubique.

  Exemple : multiplication des matrices, parcours dans les graphes

- **$T(n) = O(2^n)$**, temps exponentiel : souvent le résultat de recherche brutale d'une solution.

- **$T(n) = O(n!)$** : complexité factorielle.



Graphe représentant la différence entre leurs croissances respectives lorsque n tend vers l'infini.

Nomenclature des classes de complexité

Notation	Type de complexité
$O(1)$	complexité constante (indépendante de la taille de la donnée)
$O(log(n))$	complexité logarithmique
$O(n)$	complexité linéaire
$O(n\,log(n))$	complexité quasi-linéaire
$O(n^2)$	complexité quadratique
$O(n^3)$	complexité cubique
$O(n^p)$	complexité polynomiale
$O(n\,log(n))$	complexité quasi-polynomiale
$O(2^n)$	complexité exponentielle
$O(n!)$	complexité factorielle

L'idée de base est donc qu'un algorithme en $O(n^a)$ est « meilleur » qu'un algorithme en $O(n^b)$, si $a < b$.

est-ce qu'en pratique $100 * n^2$ est « meilleur » que $5 * n^3$?

Exemples : (1)

Test de primalité ?

- Pour tester si *N est premier ?*
- On vérifie s'il est divisible par l'un des entiers compris au sens large entre 2 et N-1 (ou encore mieux $\sqrt{n}$).

Si la réponse est négative, alors *N* est premier, sinon il est composé.

complexité racinaire :

Exemples : (2)

```
x = n ;
      while ( x > 0 )
        {
             x = x – 1 ;
        }
```

O(n)

Exemples : (3)

```
x = n ;
    while ( x > 0 )
  {
        x = x / 2 ;
    }
```

$O(\log_2(n))$

Exemples : (4)

```
x = n ;
      while ( x > 0 ) {
            y = n ;
            while  ( y > 0 ) {
                  y = y – 1 ;
                        }
             x = x – 1 ;
       }
```

$O(n^2)$

Exemples : (5)

```
x = n ;
      while ( x > 0 ) {
          y = n ;
        while ( y > 0 ) {
           y = y / 2 ;
        }
          x = x – 1 ;
      }
```

$O(n.\log_2(n))$

Exemples : (6)

```
x = n ;
      while ( x > 0 ) {
        y = n ;
        while ( y > 0 ) {
            y = y – 1 ;
        }
        x = x / 2 ;
      }
```

$O(n.\log_2(n))$

Exemples : (7)

$O(\log_2(n))$

```
Iterative-Binary-Search(A, v, low, high)
   while (low <= high)
                mid=[(low + high)/2]
                if v = A[mid] return mid
                if v > A[mid]
                low = mid + 1
                else high= mid - 1
   return -1
```

Exemples : (8)

```
x = n ;
      while ( x > 0 ) {
        y = x ;
        while ( y > 0 ) {
            y = y – 1 ;
        }
        x = x / 2 ;
      }
```

O(???)

Exemples : (8)

```
x = n ;
    while ( x > 0 ) {
      y = x ;
      while ( y > 0 ) {
        y = y – 1 ;
      }
      x = x / 2 ;
    }
```

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots$$

$$n\left(\underbrace{1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots}\right)$$

Une **série géométrique infinie** converge vers 2.

T(n)= *n*·2=2*n*

O(n)