Algorithmique avancée

Module : Algorithmes et structures de données

Douglas Teodoro

Haute Ecole Spécialisée
de Suisse occidentale

Fachhochschule Westschweiz

University of Applied Sciences and Arts
Western Switzerland

2019-2020

Sommaire

Objective

- Apprendre le concept de complexité algorithmique
- Analyser les temps d'exécution à l'aide d'outils de programmation
- Maîtriser le calcul de temps exécution d'un algorithme

Sommaire

INTRODUCTION À LA COMPLEXITÉ ALGORITHMIQUE

Temps d'exécution

Analyse de complexité en temps et en espace

Il existe en général **plusieurs algorithmes** possibles avec différents coûts en termes de :

temps d'exécution c'est-à-dire, le nombre d'opérations effectuées pour obtenir le résultat à partir des données

de taille mémoire la taille nécessaire pour stocker les différentes structures de données (variables) durant le calcul

L'**analyse de complexité** permet de mesurer l'efficacité d'un algorithme et de le comparer avec d'autre algorithmes résolvant le même problème

Notion de coût

On veut qu'un algorithme soit correct, mais aussi efficace, c'est-à-dire :

- rapide (en termes de temps d'exécution)
- économe en ressources (espace de stockage, mémoire utilisée)

Comment faire pour évaluer la qualité des algorithmes proposés ?

Notion de coût

On veut qu'un algorithme soit correct, mais aussi efficace, c'est-à-dire :

- rapide (en termes de temps d'exécution)
- économe en ressources (espace de stockage, mémoire utilisée)

Comment faire pour évaluer la qualité des algorithmes proposés ?

S1 : En mesurant le temps nécessaire à l'exécution d'un algorithme

Analyse de complexité en temps

Complexité en temps

On cherche une fonction $\mathbf{T}(\mathbf{n})$ représentant le temps d'exécution d'un algorithme en fonction de la taille de l'entrée n

Dépend de l'entrée

- algorithme de recherche : position de la clé de recherche
- algorithme de tri : si le tableau est déjà trié

On s'intéresse aux

- Meilleur des cas
- Cas moyen
- Pire de cas

Les différents temps d'exécution

Problème : étant donné un `tableau`, on demande `si l'une de ses entrées contient une valeur indiquée`

Algorithme : Recherche linéaire

Données :
A : un tableau de n nombres entiers
x : la valeur recherchée
Résultat : indice : entier // soit l'indice i pour lequel $A[i] = x$, soit la valeur spéciale -1

```
indice = -1
pour i ← 1 à n faire    // compare la valeur x avec chaque element du tableau
    si A[i] == x alors
        indice = i
retourner indice
```

Les différents temps d'exécution

Problème : étant donné un `tableau`, on demande `si l'une de ses entrées contient une valeur indiquée`

Algorithme : Recherche linéaire

Données :
A : un tableau de n nombres entiers
x : la valeur recherchée
Résultat : indice : entier `// soit l'indice i pour lequel` $A[i] = x$`, soit la valeur spéciale` -1

```
indice = −1
pour i ← 1 à n faire   // compare la valeur x avec chaque element du tableau
    si A[i] == x alors
        indice = i
retourner indice
```

Algorithme : Meilleure recherche linéaire

Données :
A : un tableau de n nombres entiers
x : la valeur recherchée
Résultat : indice : entier `// soit l'indice i pour lequel` $A[i] = x$`, soit la valeur spéciale` -1

```
pour i ← 1 à n faire   // compare la valeur x avec chaque element du tableau
    si A[i] == x alors
        retourner i
retourner −1
```

Exemple pratique - Tri d'un grand tableau

Problème : Dans une banque, on a besoin de trier un tableau de clientes avec 10 millions de registres (≈80 Mo)

Config A

instruction	$100'000 MIPS$
compilateur	langage machine
coût	$T(n) = 2n^2$

Config B

instruction	$10 MIPS$
compilateur	JAVA ou Python
coût	$T(n) = 50n\log_{10} n$

quel algorithme de tri finira plus vite : celui de config A ou de config B ?

Exemple pratique - Tri d'un grand tableau

Problème : Dans une banque, on a besoin de trier un tableau de clientes avec 10 millions de registres (≈80 Mo)

Config A

instruction	$100'000 MIPS$
compilateur	langage machine
coût	$T(n) = 2n^2$

Config B

instruction	$10 MIPS$
compilateur	JAVA ou Python
coût	$T(n) = 50n\log_{10} n$

quel algorithme de tri finira plus vite : celui de config A ou de config B ?

$$\texttt{temps d'exécution (s)} = \frac{\texttt{coût d'algorithme}}{\texttt{nombres d'instructions par second}}$$

Measurer le temps d'exécution

Utiliser des **outils de programmation** pour chronométrer des morceaux de code :

- Python `time` or `timeit`
- Java `System.nanoTime()` or `Instant.now()`
- PyCharm `Run >> Profile`
- NetBeans `Profile >> Profile File`

Measurer le temps d'exécution

Utiliser des **outils de programmation** pour chronométrer des morceaux de code :

Python	`time` or `timeit`
Java	`System.nanoTime()` or `Instant.now()`
PyCharm	`Run >> Profile`
NetBeans	`Profile >> Profile File`

- Presque tout algorithme s'exécutant sur de petites données sera rapide et sera lent sur des grandes données
- Il ne suffit pas de procéder à une seule exécution de cet algorithme par taille de données
- Il ne suffit pas de faire des moyennes sur n'importe quelles instances : il faudrait également les choisir aléatoirement

Implementation en Python

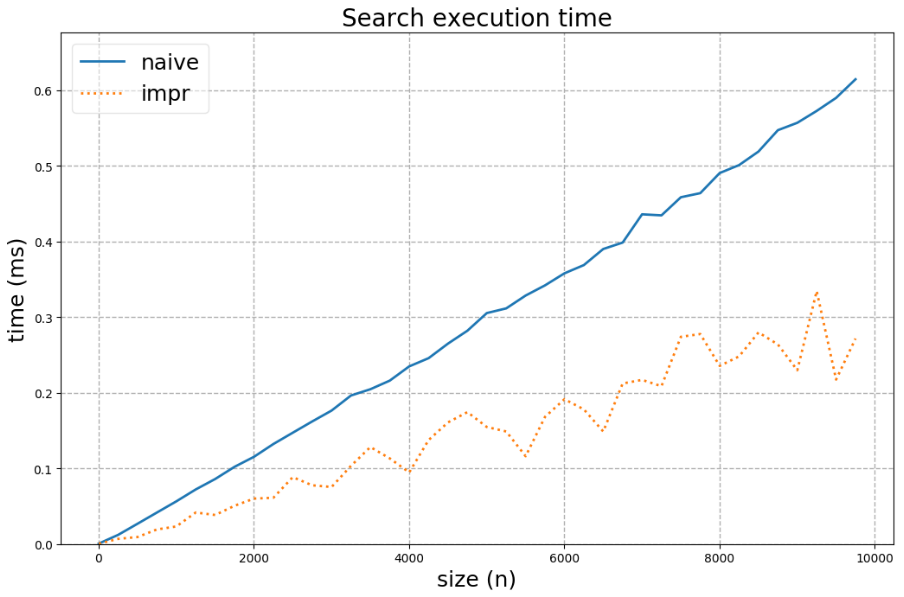
```python
def linear_naive(A: list, x: int):
  """
  Compare x with each element of A
  and return the index if found
  :param A: list of integers
  :param x: search key
  :return: index of x
  """
  r: int = -1
  for i in range(len(A)):
    if A[i] == x:
      r = i
  return r

if __name__ == "__main__":
  A = [300, 33, 947, 14, 459, 937]
  x = 14
  print("index of x in A:")
  print(linear_naive(A, x))
```

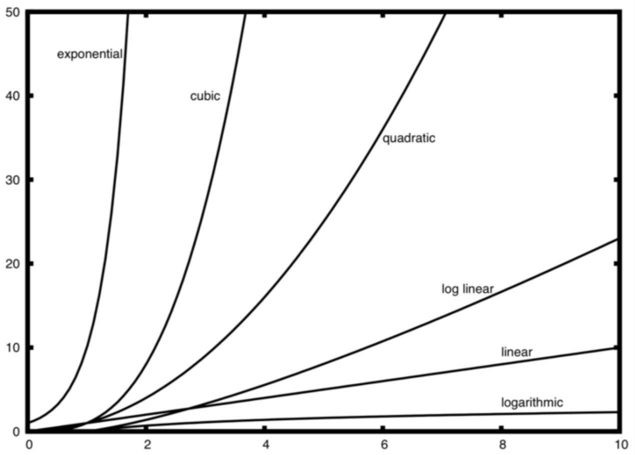
```python
def linear_impr(A: list, x: int):
  """
  Compare x with each element of A
  and return the index if found
  :param A: list of integers
  :param x: search key
  :return: index of x
  """
  for i in range(len(A)):
    if A[i] == x:
      return i
  return -1

if __name__ == "__main__":
  A = [300, 33, 947, 14, 459, 937]
  x = 14
  print("index of x in A:")
  print(linear_impr(A, x))
```

Mesurer les différents temps d'exécution



Complexité - Fonctions de temps d'exécution



T(n)	Nom
1	Constant
$\log n$	Logarithmique
n	Lineaire
$n \log n$	Log lineaire
n^2	Quadratique
n^3	Cubique
2^n	Exponentielle

Fonctions de temps d'exécution - Révision

T(n)	`n = 10`	`n = 100`	`n = 1000`
1			
$\log n$			
n			
$n \log n$			
n^2			
n^3			
2^n			

Temps d’exécution des structures de base

Coûts d'affectation et de calculs

affectation coût constant c

calcul arithmétiques coût constant c

calcul booléen coût constant c

Exemple

Algorithme : Affection, test et opérations

```
a = 1          // constante c1
a > 1          // constante c2
b = a * 2      // constante c3
```

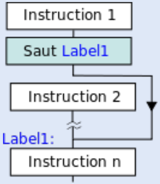
Coûts des instructions en séquence

séquence coût total est la somme des coûts de chaque instruction 1 .. k de la séquence

Exemple

Algorithme : Séquence 1-3

```
a = 1          // constante c1
a > 1          // constante c2
b = a * 2      // constante c3
```



$$T(n) = c_1 + c_2 + c_3$$

Coûts des instructions conditionnelles

`si ... alors` coût total est le coût du branche plus le coût d'évaluation de la condition

`si ... sinon` coût total est le coût de l'une des branches plus le coût d'évaluation de la condition

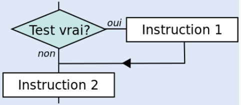
Exemple

Algorithme : Si Alors

```
si a > 1 alors
    b = a * 2
```

Algorithme : Si Sinon

```
si a == b alors
    b = a * 2
sinon
    a = b * 2
```



$T(n) = c_1 + c_2$

$T(n) = c_1 + max(c_2, c_4)$

Coûts des boucles

`tant que` ou `pour` coût total est la somme des coûts de chaque itération

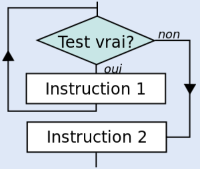
Exemple

Algorithme : Tant que

```
i = 0
tant que i < n faire
    ...
    i = i + 1
```

Algorithme : Pour

```
pour i ← 1 à n faire
    ...
```



$T(n) =$
$c1 + c_2(n+1) + (c_3 + c_4)n$

$T(n) = c_1(n+1) + c_2(n)$

PyCharm - semaine_2

`exercice_1.py`

PyCharm - Exercice 1

Objective : voir en pratique le temps d’exécution d’un algorithme en utilisant la méthode `time` de Python

1. Dans le fichier `exercice_1.py`, implementer la méthode `sum_of_n` pour calculer la somme des `n` premiers nombres entiers
    - La méthode doit retourner la somme mais aussi le temps nécessaire à son calcul

2. Augmenter la taille du paramètre d’entrée `n` ($10\times$, $100\times$, $1000\times$, etc.) et relancer le programme pour voir comment le temps d’exécution change en fonction de la taille de l’entrée

3. Est-il possible d’améliorer ?

Sommaire

L'ORDRE DE GRANDEUR ET LA NOTATION ASYMPTOTIQUE

Ordre de grandeur

Analyse de complexité en temps

Comment faire pour évaluer la qualité des algorithmes proposés ?

S2 : En analysant l'algorithme pour determiner sont temps d'exécution

La **mesure pratique** du temps d'exécution du code déjà programmé est utile, *mais* :

1. cette technique est utile lorsqu'on a déjà écrit le code correspondant à un algorithme
2. les mesures obtenues ne sont valides que pour une certaine machine à un moment donné dans un état bien précis

Coût d'un algorithme

Coût au pire

Le coût d'un algorithme A est **fonction** de la taille des données :

$$T_A(n) = max(cout(d)) \text{ pour toutes les données } d \text{ de taille } n$$

- Suppose d'avoir fixé la notion de taille
- Maximum = garantie quelles que soient les conditions d'utilisation
- Concrètement : majorer le coût + exhiber un cas défavorable

Coût au mieux

$$T_A^{min}(n) = min(cout(d)) \text{ pour toutes les données } d \text{ de taille } n$$

- Correspond au cas le plus favorable

Complexité d'un algorithme

Complexité d'un algorithme

On appelle **complexité d'un algorithme** une fonction de référence (logarithme, polynome, exponentielle...) comparable à son coût

La **complexité** est une prédiction du temps d'exécution d'un algorithme

- elle est une approximation (le temps dépend de l'architecture de la machine)
- on s'intéresse au passage à l'échelle des algorithmes (plus qu'à une mesure précise du temps d'exécution)

Ce qui est important c'est

- l'ordre de grandeur
- de pouvoir comparer les algorithmes

Simplification du coût → Complexité

Une fois trouvé la fonction de coût, on aura aussi recours aux simplifications suivantes :

1. on oublie les constantes multiplicatives (elles valent 1)
2. on annule les constantes additives
3. on ne retient que les termes dominants

Exemple (simplification)

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

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$

Simplification du coût $\longrightarrow$ Complexité

Une fois trouvé la fonction de coût, on aura aussi recours aux simplifications suivantes :

1. on oublie les constantes multiplicatives (elles valent 1)
2. on annule les constantes additives
3. on ne retient que les termes dominants

Exemple (simplification)

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

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$

Simplification du coût → Complexité

Une fois trouvé la fonction de coût, on aura aussi recours aux simplifications suivantes :

1. on oublie les constantes multiplicatives (elles valent 1)
2. on annule les constantes additives
3. on ne retient que les termes dominants

Exemple (simplification)

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

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : n^3 et on a donc

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

Exemple I - Analyse de la recherche linéaire

Algorithme : Recherche linéaire

Données : A : tableau ; n : entier ; x : entier
Résultat : indice : entier

```
indice = −1
pour i ← 1 à n faire
    si A[i] == x alors
        indice = i
retourner indice
```

$$\begin{aligned}\mathbf{T(n)} &= c_1 + c_2(n+1) + c_3 n + c_4 n + c_5 \\ &= a\mathbf{n} + b \\ &= O(n)\end{aligned}$$

Exemple I - Analyse de la recherche linéaire

Algorithme : Recherche linéaire

Données : A : tableau ; n : entier ; x : entier
Résultat : indice : entier

indice = -1
pour $i \leftarrow 1$ **à** n **faire**
    si $A[i] == x$ **alors**
        indice = i
retourner indice

Algorithme : Meilleure recherche

Données : A : tableau ; n : entier ; x : entier
Résultat : indice : entier

pour $i \leftarrow 1$ **à** n **faire**
    si $A[i] == x$ **alors**
        retourner i
retourner -1

$$\begin{aligned}\mathbf{T}(\mathbf{n}) &= c_1 + c_2(n+1) + c_3 n + c_4 n + c_5 \\ &= a\mathbf{n} + b \\ &= \mathcal{O}(n)\end{aligned}$$

$$\begin{aligned}\mathbf{T}(\mathbf{n}) &= c_1(n+1) + c_2 n + c_3 + c_4 \\ &= a\mathbf{n} + b \\ &= \mathcal{O}(n)\end{aligned}$$

Exemple II - Recherche dans des tableaux

Étant donnée les tableaux `A` et `B` de taille `n`

Problème : Les tableaux `A` et `B` contiennent-ils un élément commun ?

Algorithme : Recherche d'élément commun

pour $i \leftarrow 1$ **à** n **faire**
    pour $j \leftarrow 1$ **à** n **faire**
        si $A[i] == B[j]$ **alors**
            retourner True
retourner False

Exemple II - Recherche dans des tableaux

Étant donnée les tableaux A et B de taille n

Problème : Les tableaux A et B contiennent-ils un élément commun ?

Algorithme : Recherche d'élément commun

```
pour i ← 1 à n faire
    pour j ← 1 à n faire
        si A[i] == B[j] alors
            retourner True
retourner False
```

coût	fois
c_1	$n+1$
c_2	$n*(n+1)$
c_3	$n*n$
c_4	1
c_5	1

Exemple II - Recherche dans des tableaux

Étant donnée les tableaux `A` et `B` de taille `n`

Problème : Les tableaux `A` et `B` contiennent-ils un élément commun ?

Algorithme : Recherche d'élément commun

```
pour i ← 1 à n faire
    pour j ← 1 à n faire
        si A[i] == B[j] alors
            retourner True
retourner False
```

coût	fois
c_1	$n + 1$
c_2	$n * (n + 1)$
c_3	$n * n$
c_4	1
c_5	1

$$\begin{aligned} \mathbf{T(n)} &= c_1(n+1) + c_2 n(n+1) + c_3 n^2 + c_4 + c_5 \\ &= a\mathbf{n^2} + bn + c \\ &= O(n^2) \end{aligned}$$

Classes de complexité

Complexité	**Type**
$\mathcal{O}(1)$	accéder au premier élément d'un ensemble de données
$\mathcal{O}(\log n)$	couper un ensemble en deux puis chacun en deux, etc.
$\mathcal{O}(n)$	parcourir un ensemble de n données
$\mathcal{O}(n \log n)$	couper répétitivement un ensemble en deux et parcourir chacune des parties
$\mathcal{O}(n^2)$	parcourir un ensemble de données une fois par élément d'un autre ensemble de même taille (tris par comparaison)
$\mathcal{O}(2^n)$	résolution par recherche exhaustive du Rubik's Cube
$\mathcal{O}(n!)$	résolution par recherche exhaustive du probleème du voyageur de commerce

Classes de complexité

Supposons un processeurs que fait 10^6 opérations par seconde

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10^2	$6.6\mu s$	0.1ms	0.6ms	10ms	1s	4×106 ans
10^3	$9.9\mu s$	1 ms	10 ms	1 s	16.6 min	
10^4	$13.3\mu s$	10 ms	0.1 s	1.6 min	11.6 j	
10^5	$16.6\mu s$	0.1 s	1.6 s	2.7 h	317 ans	
10^6	$19.9\mu s$	1 s	19.9 s	11.6 j	106 ans	
10^7	$23.3\mu s$	10 s	3.9 min	3.17 ans		
10^8	$26.6\mu s$	1.6 min	44.3 min	317 ans		
10^9	$29.9\mu s$	16.6min	8.3 h	31709 ans		

QCM1 - Temps de deux boucles

Étant donnée les tableaux `A` et `B` (longueur `n`) et `x` (un entier)

Problème : Est-ce que le tableau `A` ou le tableau `B` contient `x` ?

Algorithme : Deux boucles

pour $i \leftarrow 1$ **à** n **faire**
    si $A[i] == x$ **alors**
        retourner True
pour $i \leftarrow 1$ **à** n **faire**
    si $B[i] == x$ **alors**
        retourner True
retourner False

Question : Quel est le temps d'exécution ?

A) $\mathcal{O}(1)$
B) $\mathcal{O}(\log n)$
C) $\mathcal{O}(n)$
D) $\mathcal{O}(n^2)$

QCM1 - Temps de deux boucles

Étant donnée les tableaux `A` et `B` (longueur `n`) et `x` (un entier)

Problème : Est-ce que le tableau `A` ou le tableau `B` contient `x` ?

Algorithme : Deux boucles

```
pour i ← 1 à n faire
    si A[i] == x alors
        retourner True
pour i ← 1 à n faire
    si B[i] == x alors
        retourner True
retourner False
```

Question : Quel est le temps d'exécution ?

A) $\mathcal{O}(1)$
B) $\mathcal{O}(\log n)$
C) $\mathcal{O}(n)$
D) $\mathcal{O}(n^2)$

QCM2 - Temps de deux boucles imbriquées

Étant donnée A (tableau de longueur n)

Problème : Est-ce que le tableau A contient des doublons ?

Algorithme : Deux boucles imbriquées II

```
pour i ← 1 à n faire
    pour j ← i + 1 à n faire
        si A[i] == A[j] alors
            retourner True
retourner False
```

Question : Quel est le temps d'exécution ?

A) $\mathcal{O}(1)$

B) $\mathcal{O}(\log n)$

C) $\mathcal{O}(n)$

D) $\mathcal{O}(n^2)$

QCM2 - Temps de deux boucles imbriquées

Étant donnée A (tableau de longueur n)

Problème : Est-ce que le tableau A contient des doublons ?

Algorithme : Deux boucles imbriquées II

pour $i \leftarrow 1$ **à** n **faire**
  pour $j \leftarrow i+1$ **à** n **faire**
    si $A[i] == A[j]$ **alors**
      retourner True
retourner False

Question : Quel est le temps d'exécution ?

A) $\mathcal{O}(1)$

B) $\mathcal{O}(\log n)$

C) $\mathcal{O}(n)$

D) $\mathcal{O}(n^2)$

PyCharm - semaine_2

`exercice_2.py`

PyCharm - Exercice 2

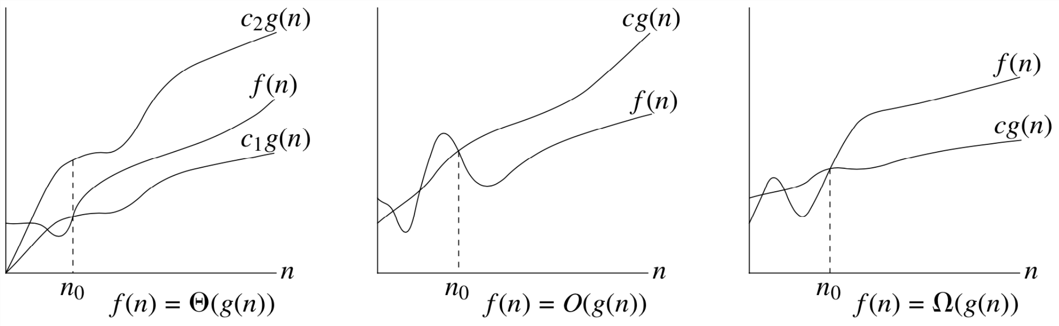
Objective : voir en pratique le temps d'exécution pour quelques complexités algorithmiques

1. Pour chaque fonction non implémentée dans le fichier `exercice_2.py`, l'implémenter de manière à respecter en fonction de la taille d'entrée n la complexité respective
   1.1 `comp_constant`
   1.2 `comp_lineaire`
   1.3 `comp_quadratique`
   1.4 `comp_cubique`
   - L'algorithme n'a pas à résoudre un problème
2. Augmenter la taille du paramètre d'entrée `n` (5×, 25×, 50×, etc.) et relancer le programme
3. **Bonus** : Quel est la complexité de la fonction `comp_x` ?

Notation asymptotique

Notation asymptotique

Les notations asymptotiques représentent la croissance d'une fonction lorsque son argument tend vers l'infini de façon asymptotique



Borne supérieure asymptotique

Notation O

Pour une fonction donnée $g(n)$ on note $O(g(n))$ (« grand O de g de n » ou « o de g de n ») l'ensemble de fonctions :

$$\begin{aligned} O(g(n)) = \{f(n) : & \text{ il existe des constantes positives } c \text{ et } n_0 \\ & \text{telles que } 0 \leq f(n) \leq cg(n) \text{ pour tout } n \geq n_0\} \end{aligned}$$

On écrit $f(n) = O(g(n))$ s'il existe des constantes positives n_0 et c telles que, à droite de n_0, la valeur de $f(n)$ soit toujours inférieure ou égale à $cg(n)$

On dit que $g(n)$ est une borne supérieure asymptotique pour $f(n)$

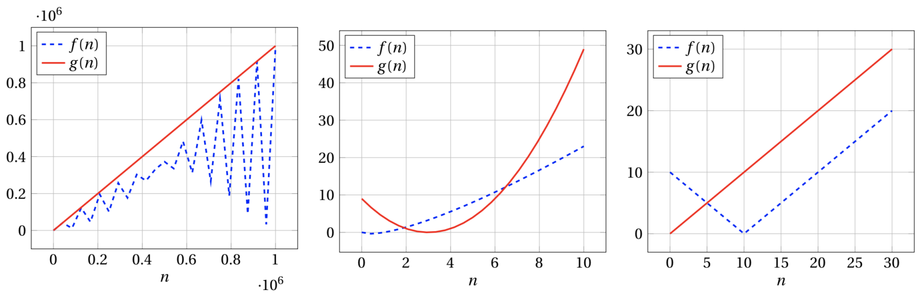
Borne supérieure asymptotique

Notation O

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

Vite dit : $f(n)$ est dépassée par $g(n)$ à partir d'une certaine taille de données

Exemple



Borne supérieure asymptotique

Démontrer que $n^2 = \mathcal{O}(10^{-5}n^3)$ en utilisant la definition de la notation $\mathcal{O}$:

$$0 \leq f(n) \leq cg(n)$$
$$n \geq n_o$$
$$c > 0; n_o \geq 0$$

1. $f(n) \leftarrow n^2$

Borne supérieure asymptotique

Démontrer que $n^2 = \mathcal{O}(10^{-5}n^3)$ en utilisant la definition de la notation $\mathcal{O}$:

$$0 \leq f(n) \leq cg(n)$$
$$n \geq n_o$$
$$c > 0; n_o \geq 0$$

1. $f(n) \leftarrow n^2$
2. $g(n) \leftarrow 10^{-5}n^3$

Borne supérieure asymptotique

Démontrer que $n^2 = \mathcal{O}(10^{-5}n^3)$ en utilisant la definition de la notation $\mathcal{O}$:

$$0 \leq f(n) \leq cg(n)$$
$$n \geq n_o$$
$$c > 0; n_o \geq 0$$

1. $f(n) \leftarrow n^2$
2. $g(n) \leftarrow 10^{-5}n^3$
3. $f(n) \leq cg(n)$

Borne supérieure asymptotique

Démontrer que $n^2 = \mathcal{O}(10^{-5}n^3)$ en utilisant la definition de la notation $\mathcal{O}$:

$$0 \leq f(n) \leq cg(n)$$
$$n \geq n_o$$
$$c > 0; n_o \geq 0$$

1. $f(n) \leftarrow n^2$
2. $g(n) \leftarrow 10^{-5}n^3$
3. $f(n) \leq cg(n)$
4. $n^2 \leq c10^{-5}n^3$

Borne supérieure asymptotique

Démontrer que $n^2 = \mathcal{O}(10^{-5}n^3)$ en utilisant la definition de la notation $\mathcal{O}$:

$$
\begin{gathered}
0 \leq f(n) \leq cg(n) \\
n \geq n_o \\
c > 0; n_o \geq 0
\end{gathered}
$$

1. $f(n) \leftarrow n^2$
2. $g(n) \leftarrow 10^{-5}n^3$
3. $f(n) \leq cg(n)$
4. $n^2 \leq c10^{-5}n^3$
5. $cn \geq 10^5$

En fixant $c = 1$ et replaçant dans 5, on a $n_o = 10^5$

Borne supérieure asymptotique

Démontrer que $n^2 = \mathcal{O}(10^{-5}n^3)$ en utilisant la definition de la notation $\mathcal{O}$:

$$0 \leq f(n) \leq cg(n)$$
$$n \geq n_o$$
$$c > 0; n_o \geq 0$$

1. $f(n) \leftarrow n^2$
2. $g(n) \leftarrow 10^{-5}n^3$
3. $f(n) \leq cg(n)$
4. $n^2 \leq c10^{-5}n^3$
5. $cn \geq 10^5$

En fixant $c = 1$ et replaçant dans 5, on a $n_o = 10^5$
Pour $n \geq 10^5 \rightarrow n^2 \leq c10^{-5}n^3$

Exercice 3 - Notation asymptotique

Question : En utilisant la définition de la borne supérieure asymptotique O, prouver que la fonction

$$f(n) = 5n + 10$$

est en $O(n)$

Sommaire

Conclusion

- La complexité d'un algorithme nous permet de le classer dans les choix qu'on va réaliser pour résoudre un problème donné

- Si on a le choix entre un algorithme A en $\mathcal{O}(2^n)$ et un algorithme B en $\mathcal{O}(\log n)$, il n'y a généralement pas besoin d'hésiter : on choisira B pour résoudre notre problème (si seulement le temps de calcul nous intéresse)

- L'analyse de complexité est un critère fiable pour les comparer mais il est toujours nécessaire d'effectuer des analyses expérimentales avant de choisir le "meilleur" algorithme

La prochaine fois

- Schémas récursifs
- Comparaison des approches
- Preuves et corrections

Référence

Algorithmes Notions de base : Pages 11 - 21 Cormen, http://hesge.scholarvox.com

Cyberlearn : 19_HES-SO-GE_633-1 ALGORITHMES ET STRUCTURES DES DONNéES

http://cyberlearn.hes-so.ch

Extra

Contribution par terme

Exemple

Soit une fonction polynomial

$$T(n) = a_k n^k + a_{(k-1)} n^{(k-1)} + \cdots + a_2 n^2 + a_1 n + a_0$$

le terme "intéressant" est n^k car c'est le terme qui croît le plus vite

On dit que l'expression $T(n)$ est de l'ordre de n^k

$$T(n) = a_k n^k + \cdots + a_0 = O(n^k)$$

Contribution par terme

Soit le polynôme $n^2 + n + 1$.

Si $n = 10$,

- la contribution du terme quadratique (n^2) est :

$$\frac{n^2}{n^2 + n + 1} = \frac{10^2}{10^2 + 10 + 1} = 0.90$$

- et la contribution du terme linéaire (n) est :

$$\frac{n}{n^2 + n + 1} = \frac{10}{10^2 + 10 + 1} = 0.09$$

Donc, la fonction $n^2 + n + 1$ est de l'ordre $O(n^2)$

QCM 3 - Contribution par terme

Question : En sachant qu'une fonction exponentielle est de la forme k^n (k une constante donnée), quel terme croît plus vite pour la fonction :

$$T(n) = 1000n^2 + 10n^3 + 2^n + 1024$$

A) n^2

B) n^3

C) 2^n

D) 1

QCM 3 - Contribution par terme

Question : En sachant qu'une fonction exponentielle est de la forme k^n (k une constante donnée), quel terme croît plus vite pour la fonction :

$$T(n) = 1000n^2 + 10n^3 + 2^n + 1024$$

A) n^2
B) n^3
C) 2^n
D) 1

Exemple (n=50)

$$\frac{2^n}{1000n^2 + 10n^3 + 2^n + 1024} = \frac{2^{50}}{1000 \times 50^2 + 10 \times 50^3 + 2^{50} + 1024} \approx 0.99$$

Exercice 3 - Notation asymptotique : Résolution

Question : En utilisant la définition de la borne supérieure asymptotique O, prouver que la fonction

$$f(n) = 5n + 10$$

est en $O(n)$

1. notre but est de trouver une constante c et un seuil n_0 à partir duquel $f(n) \leq cg(n)$
$$5n + 10 \leq cn$$
2. en résolvant l'inéquation
$$10 \leq (c - 5)n$$
pour $c = 6$
3. on en déduit donc que $5n + 10 \leq 6n$ à partir du seuil $n_0 = 10$

Remarque : on ne demande pas d'optimisation (le plus petit c ou n_0 qui fonctionne), juste de donner des valeurs qui fonctionnent : $c = 10$ et $n_0 = 2$ sont donc aussi acceptables