UCLouvain

Algorithmique

Cours préparatoires au master en sciences informatiques

Vincent Branders

2022–2023

Complexité du tri

Résumé sur la complexité temporelle

- La complexité temporelle d'un algorithme mesure comment évolue le temps calcul du programme associé
  - lorsque la taille du problème augmente
  - en négligeant des constantes multiplicatives et des termes non-dominants
- Ces complexités s'expriment en $\mathcal{O}$ (borne supérieure), Ω (borne inférieure) ou Θ (borne exacte)
- La complexité dans le pire cas peut être différente de la complexité dans le meilleur cas
  - Exemple : $\Theta(1)$ meilleure cas, $\Theta(n^2)$ pire cas $\Rightarrow$ globalement $\mathcal{O}(n^2)$
- La complexité dans le pire cas est parfois égale à la complexité dans le meilleur cas
  - Exemple : $\Theta(n)$ meilleure cas, $\Theta(n)$ pire cas $\Rightarrow$ globalement $\Theta(n)$

Complexité spatiale

- Raisonnement analogue à celui utilisé pour la complexité temporelle
- On s'intéresse ici à l'espace utilisé, c'est-à-dire à ce qu'il faut mémoriser
- On s'intéresse aux termes dominants (analyse asymptotique)

Algorithm: arrayFirstNegative
Input: A, un tableau de n entiers ($n > 0$)
Output: L'indice du premier entier negatif dans A

```
for i ← 0 to n − 1 do
    if A[i] < 0 then
        return i                  // Indice du premier entier négatif
return −1                         // Aucun entier négatif dans A
```

La complexité spatiale de `arrayFirstNegative` est $\Theta(n)$

Quizz sur les notations $\mathcal{O}, \Theta, \Omega$

- $2^{n+1} \in \Theta(2^n)$?
  - $2^{n+1} = 2.2^n \in \Theta(2^n)$
- Est-il possible de définir 2 fonctions $f(n)$ et $g(n)$ telles que

$$f(n) < g(n), \forall n > 1 \text{ et } f(n) \in \Theta(g(n)) \text{ ?}$$

  - $f(n) = n$, $g(n) = 3n$
  - $f(n) = 2^n$, $g(n) = 2^{n+1}$
- $3^n + 2^n \in \Theta(5^n)$
  - $3^n + 2^n \notin \Theta(5^n)$
  - Terme dominant : $3^n \in \mathcal{O}(5^n)$ mais $3^n \notin \Omega(5^n) \implies 3^n \notin \Theta(5^n)$
  - Preuve : $3^n \notin \Omega(5^n)$ car $5^n \notin \mathcal{O}(3^n)$

    $$\lim_{n\to\infty} \frac{5^n}{3^n} = \lim_{n\to\infty} (\frac{5}{3})^n = \lim_{n\to\infty} 1.666^n = \infty$$
  - Il suffit de retenir : $a^n \gg b^n$ si $a > b$

$$n! \gg 3^n \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \sqrt{n} \gg \log n \gg 1$$

Prouver $3^n \notin \Omega(5^n)$

$3^n \in \Omega(5^n)$?si $5^n \in \mathcal{O}(3^n)$

Prouvons $5^n \in \mathcal{O}(3^n)$

Utilisons la définition de $\mathcal{O}$:

$f(n) \in \mathcal{O}\left(g\left(n\right)\right)$ si $\exists c > 0$, $\exists n_0 \geq 1$ tels que $f(n) \leq c.g\left(n\right)$, $\forall n \geq n_0$

Ici, $f(n) = 5^n$ et $g(n) = 3^n$, donc :

$5^n \in \mathcal{O}\left(3^n\right)$ si $\exists c > 0$, $\exists n_0 \geq 1$ tels que $5^n \leq c.3^n$, $\forall n \geq n_0$

Il faut chercher des valeurs de c et n_0 pour que ce soit vrai

Choisissons un c suffisament grand : x (par exemple $x = 10^9$ ou $x = 10^{100}$)

Il faut chercher la valeur de n_0 telle que : $5^n \leq x.3^n$, $\forall n \geq n_0$

Vérifions déjà quand est-ce que $5^n \leq x.3^n$:

$$\frac{5^n}{3^n} \leq x$$

$$\left(\frac{5}{3}\right)^n \leq x$$

$$\log_{\frac{5}{3}} \left(\frac{5}{3}\right)^n \leq \log_{\frac{5}{3}} x$$

$$n \leq \log_{\frac{5}{3}} x$$

Donc $5^n \leq x.3^n$ est vrai seulement si $n \leq \log_{\frac{5}{3}} x$

La définition de $\mathcal{O}$ n'est pas respectée $\implies 5^n \notin \mathcal{O}(3^n)$

$\implies 3^n \notin \Omega(5^n)$

Question

- Les algorithmes `Magnifique` et `Splendide` résolvent le même problème
- La complexité temporelle de `Magnifique` est en $\Theta(n)$ dans tous les cas (avec n la taille du problème)
- La complexité temporelle de `Splendide` est en $\Theta(n^2)$ dans tous les cas

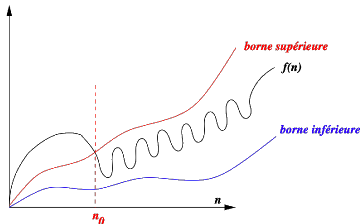
Est-il possible qu'un programme Python implémentant `Splendide` s'exécute plus vite qu'un programme Python implémentant `Magnifique`? Toujours, jamais, parfois?

Réponses

- Cela dépend du programmeur !
- Cela dépend de tous les autres facteurs (machine, OS, charge du réseau, *etc*) qui influencent le temps d'exécution

De la théorie à la pratique

- Et en supposant que les programmes soient écrits en ne rajoutant pas d'instructions modifiant la complexité des algorithmes associés et que les autres facteurs ne jouent pas significativement (même machine, même langage, réseau "stable",...)?
  - Oui, c'est parfois possible pour $n < n_0$



- Quel est alors l'intérêt d'implémenter un algorithme en $\Theta(n)$ plutôt qu'en $\Theta(n^2)$ (en pratique on ne connaît pas avec précision n_0) ?
  - Cet algorithme finira toujours par être meilleur lorsque la taille du problème augmente !

Complexité avec boucles imbriquées


```
Algorithm: equalPair
Input: A un tableau de n + 1 entiers (n > 0)
Output: Une paire d'éléments égaux dans A, ∅ sinon
for i ← 0 to n do
    current ← A[i]
    for j ← i + 1 to n do
        if A[j] == current then
            return (current, A[j])          // Paire trouvée
return ∅                                    // Aucune paire d'éléments égaux
```

- La boucle externe est exécutée $\mathcal{O}(n)$ fois
- La boucle interne est exécutée au plus $n + (n-1) + (n-2) + \ldots 1$ fois
  - $\sum_{k=0}^{n-1} n - k = \sum_{k=1}^{n} k = 1 + 2 + \ldots n = \frac{n(n+1)}{2} \in \Theta(n^2)$
- Toutes les autres opérations s'exécutent en temps constant $\Theta(1)$

La complexité temporelle est globalement en $\mathcal{O}(n^2)$

- Meilleur cas : $\Theta(1)$
- Pire cas : $\Theta(n^2)$

Complexité temporelle avec boucles


```
...
for i ← 1 to n do
    ...
    for j ← 1 to n do
        ...
    ...
...
```

```
...
for i ← 1 to n do
    ...
...
for j ← 1 to n do
    ...
...
```

- Les opérations se multiplient
- $\Theta(n) \times \Theta(n) \Rightarrow \Theta(n^2)$ itérations

- Les opérations s'additionnent
- $\Theta(n) + \Theta(n) \Rightarrow \Theta(n)$ itérations

Attention

La complexité globale dépend aussi du nombre et du coût des opérations effectuées à l'intérieur (...) et à l'extérieur (...) de chaque boucle

Principales classes d'algorithmes

Rappel des relations

$$n! \gg 3^n \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \sqrt{n} \gg \log n \gg 1$$

- Temps constant $\mathcal{O}(1) = \Theta(1)$
  - addition de deux nombres
  - `print("Hello World!")`
  - $f(n) = \min(n, 100)$
- Fonction logarithmique $\mathcal{O}(\log(n))$
  - problème divisé par 2 (ou 3 ou 4 ou …) à chaque étape
  - recherche dichotomique
- Fonction linéaire $\mathcal{O}(n)$
  - énumération de chaque élément d'une collection de taille n
  - nombre constant d'opérations par élément
- Fonction super-linéaire $\mathcal{O}(n \log n)$
  - algorithme de tri par fusion

Principales classes d'algorithmes

Rappel des relations

$$n! \gg 3^n \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \sqrt{n} \gg \log n \gg 1$$

- Fonction quadratique $\mathcal{O}\left(n^2\right)$
  - énumération des paires d'éléments d'une collection de taille n
  - 2 boucles imbriquées, chacune $\mathcal{O}\left(n\right)$ itérations
- Fonction cubique $\mathcal{O}\left(n^3\right)$
  - énumération des triplets d'éléments d'une collection de taille n
  - 3 boucles imbriquées, chacune $\mathcal{O}\left(n\right)$ itérations
- Fonction exponentielle $\mathcal{O}\left(2^n\right)$
  - énumération des 2^n sous-ensembles d'une collection de taille n

Par exemple, avec $n = 5$ et l'ensemble : $\{1, 2, 3, 4, 5\}$

- sous-ensembles : $\{3\}$ ou $\{2, 4\}$ ou $\{1, 2, 5\}$ ou ...
- code : 00100 ou 01010 ou 11001 ou ...

Globalement, $2^5 = 32$ codes possibles sur 5 bits

Le problème du tri

Input: Une collection de n éléments que l'on peut ordonner deux à deux ($\leq$)
Output: La collection triée en ordre croissant : $a_1 \leq a_2 \leq \dots a_n$

- 10; 4; 3; 0; 67; 4; -27; 5 $\Rightarrow$ -27; 0; 3; 4; 4; 5; 10; 67
- `b`; `d`; `a`; `c` $\Rightarrow$ `a`; `b`; `c`; `d` (selon $\leq$, l'ordre alphabétique)

Tri à bulles : exemple

Principe général

- Inverser toute paire d'éléments adjacents qui sont désordonnés
- Un plus grand élément est comme une bulle remontant vers la fin
- Un plus petit élément est comme une bulle descendant vers le début

5	1	12	-5	16	unsorted
5	1	12	-5	16	5 > 1, swap
1	5	12	-5	16	5 < 12, ok
1	5	12	-5	16	12 > -5, swap
1	5	-5	12	16	12 < 16, ok
1	5	-5	12	16	1 < 5, ok
1	5	-5	12	16	5 > -5, swap
1	-5	5	12	16	5 < 12, ok
1	-5	5	12	16	1 > -5, swap
-5	1	5	12	16	1 < 5, ok
-5	1	5	12	16	-5 < 1, ok
-5	1	5	12	16	sorted

Algorithme du tri à bulles

Algorithm: BubbleSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant

for $i \leftarrow n - 1$ **down to** 0 **do**
  for $j \leftarrow 1$ **to** i **do**
    if $A[j-1] > A[j]$ **then**
      swap($A[j], A[j-1]$)

return A

Complexité temporelle du tri à bulles


```
Algorithm: BubbleSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant
for i ← n − 1 down to 0 do                    // Θ(n) itérations
    for j ← 1 to i do                          // O(n) itérations
        if A[j − 1] > A[j] then                // Θ(1)
            swap(A[j], A[j − 1])               // Θ(1)
return A                                       // Θ(1)
```

- Nombre exact d'itérations :

$$(n-1)+(n-2)+\ldots 1=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2} \in \Theta\left(n^2\right)$$

- $\Theta\left(n^2\right)$ comparaisons, $\mathcal{O}\left(n^2\right)$ échanges
  - Pire cas : tableau en ordre décroissant $\Rightarrow \Theta(n^2)$ swaps
  - Meilleur cas : tableau trié en ordre croissant $\Rightarrow 0$ swaps
- Complexité dans tous les cas : $\Theta(n^2)$

Qualités attendues d'un algorithme de tri

	BubbleSort
❶ Minimiser la complexité temporelle globale	❶ $\Theta(n^2)$
❷ Minimiser le nombre de comparaisons	❷ $\Theta(n^2)$
❸ Minimiser le nombre d'échanges	❸ $\mathcal{O}(n^2)$
❹ Stabilité : clés égales non réordonnées	❹ Ok
❺ Tri en place : espace additionnel $\Theta(1)$	❺ Ok
❻ Tri adaptatif : meilleur complexité si (quasi) déjà trié	❻ Non

Tri à bulles adaptatif


```
Algorithm: BubbleSort2
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant
for i ← n − 1 down to 0 do
    swapped ← FALSE
    for j ← 1 to i do
        if A[j − 1] > A[j] then
            swap(A[j], A[j − 1])
            swapped ← TRUE
    if not swapped then
        break
return A
```


- Complexité dans le meilleur cas (quasi) déjà trié : $\Theta(n)$
- Complexité dans tous les cas : $\mathcal{O}(n^2)$

Tri à bulles pas si adaptatif que ça

Algorithm: BubbleSort2
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant

```
for i ← n − 1 down to 0 do
    swapped ← FALSE
    for j ← 1 to i do
        if A[j − 1] > A[j] then
            swap(A[j], A[j − 1])
            swapped ← TRUE
    if not swapped then
        break
return A
```

- Les petites valeurs descendent lentement
- Complexité en $\mathcal{O}(n^2)$

Résumé

- Un algorithme de tri intuitif, stable mais peu efficace
- Même avec l'amélioration de la version `BubbleSort2`
  - $\mathcal{O}(n^2)$ comparaisons
  - $\mathcal{O}(n^2)$ échanges
  - Complexité temporelle pour tous les cas : $\mathcal{O}(n^2)$
- Asymétrie entre valeurs élevées (qui remontent vite) et valeurs faibles (qui descendent lentement)

Tri par sélection : exemple

5	1	12	-5	16	2	12	14
5	1	12	-5	16	2	12	14
-5	1	12	5	16	2	12	14
-5	1	12	5	16	2	12	14
-5	1	2	5	16	12	12	14
-5	1	2	5	16	12	12	14
-5	1	2	5	12	16	12	14
-5	1	2	5	12	12	16	14

Principe général

- Sélectionner de façon répétée le plus petit élément restant et le mettre au début

Algorithme du tri par sélection

Algorithm: SelectionSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant

```
for i ← 0 to n − 1 do
    min ← i
    // Recherche du min dans le tableau restant
    for j ← i + 1 to n − 1 do
        if A[j] < A[min] then
            min ← j
    if min ≠ i then
        swap(A[i], A[min])
return A
```

Complexité temporelle du tri par sélection

Algorithm: SelectionSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant

```
for i ← 0 to n − 1 do                                   // Θ(n) itérations
    min ← i                                             // Θ(1)
    // Recherche du min dans le tableau restant
    for j ← i + 1 to n − 1 do                           // O(n) itérations
        if A[j] < A[min] then                           // Θ(1)
            min ← j                                     // Θ(1)
    if min ≠ i then                                     // Θ(1)
        swap(A[i], A[min])                              // Θ(1)
return A
```

- Nombre exact d'itérations : $\frac{n(n-1)}{2} \in \Theta\left(n^2\right)$
- $\Theta\left(n^2\right)$ comparaisons, $\mathcal{O}\left(n\right)$ échanges
  - Pire cas : tableau en ordre décroissant $\Rightarrow \Theta(n)$ swaps
  - Meilleur cas : tableau trié en ordre croissant $\Rightarrow 0$ swaps
- Complexité dans tous les cas : $\Theta(n^2)$

Le tri par sélection n'est pas adaptatif

Propriété du tri par sélection

Quel que soit l'ordre initial, on effectuera $\Theta\left(n^2\right)$ comparaisons

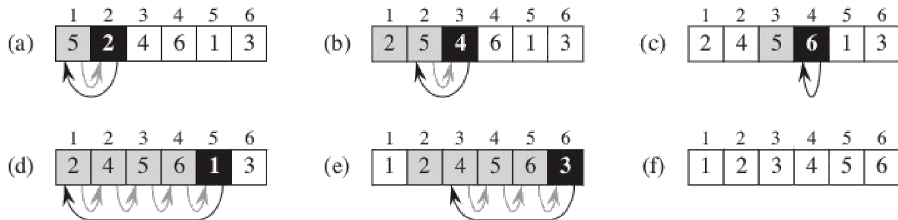
https://www.toptal.com/developers/sorting-algorithms/selection-sort

Tri par sélection : en résumé

	`BubbleSort`
1. Complexité temporelle globale	1. $\Theta(n^2)$
2. Nombre de comparaisons	2. $\Theta(n^2)$
3. Nombre d'échanges	3. $\mathcal{O}(n)$
4. Stabilité : clés égales non réordonnées	4. Non
5. Tri en place : espace additionnel $\Theta(1)$	5. Ok
6. Tri adaptatif	6. Non

En résumé : un algorithme peu efficace, $\Theta(n^2)$ dans tous les cas, mais marginalement intéressant si on veut minimiser le nombre d'échanges

Tri par insertion : exemple



- la clé courante ($A[i]$)
- la partie du sous-tableau déjà trié avec lequel la clé est comparée avant insertion

Algorithme du tri par insertion

Algorithm: InsertionSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant

for $i \leftarrow 1$ **to** $n-1$ **do**
  $key \leftarrow A[i]$
  $j \leftarrow i-1$
  // Insertion de $A[i]$ dans le tableau trié $A[0 \ldots i-1]$
  while $j \geq 0$ **and** $A[j] > key$ **do**
    $A[j+1] \leftarrow A[j]$ // Déplacement d'une position
    $j \leftarrow j-1$
  $A[j+1] \leftarrow key$
return A

Complexité temporelle du tri par sélection


```
Algorithm: InsertionSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant
for i ← 1 to n − 1 do                          // Θ(n) itérations
    key ← A[i]                                 // Θ(1)
    j ← i − 1                                  // Θ(1)
    while j ≥ 0 and A[j] > key do              // O(n) itérations
        A[j + 1] ← A[j]                        // Θ(1)
        j ← j − 1                              // Θ(1)
    A[j + 1] ← key                             // Θ(1)
return A                                       // Θ(1)
```

- Nombre effectif d'itérations : $\mathcal{O}(n^2)$ (2 conditions d'arrêt du while)
- $\mathcal{O}\left(n^2\right)$ comparaisons, $\mathcal{O}\left(n^2\right)$ échanges
  - Pire cas : décroissant $\Rightarrow \Theta(n^2)$ comp. + swaps
  - Meilleur cas : croissant $\Rightarrow \Theta(n)$ comp. + swaps
- Complexité dans tous les cas : $\mathcal{O}(n^2)$

Tri par insertion : en résumé

	BubbleSort
❶ Complexité temporelle globale	❶ $\mathcal{O}(n^2)$
❷ Nombre de comparaisons	❷ $\mathcal{O}(n^2)$
❸ Nombre d'échanges	❸ $\mathcal{O}(n^2)$
❹ Stabilité : clés égales non réordonnées	❹ Ok
❺ Tri en place : espace additionnel $\Theta(1)$	❺ Ok
❻ Tri adaptatif	❻ Ok

En résumé : un algorithme stable et adaptatif mais qui reste quadratique $\mathcal{O}(n^2)$

Exercices sur le tri :

https://inginious.org/course/prepa-master/Algo_8_bulle

https://inginious.org/course/prepa-master/Algo_9_selection

https://inginious.org/course/prepa-master/Algo_10_insertion

https://inginious.org/course/prepa-master/Algo_11_stable_search

https://inginious.org/course/prepa-master/Algo_12_Pluvio_median

https://inginious.org/course/prepa-master/Algo_13_Pluvio_mode

Résumé

- Le problème du tri est omniprésent en informatique
- Il vaut mieux éviter le `BubbleSort` qui est en $\Theta(n^2)$
  - Même Barak Obama est au courant !
- `SelectionSort` n'est pas vraiment meilleur, sauf pour minimiser le nombre d'échanges ($\mathcal{O}(n)$)
- `InsertionSort` stable et adaptatif mais reste quadratique $\mathcal{O}(n^2)$
  - son caractère adaptatif est un plus
  - utile comme algorithme de base dans des algorithmes plus avancés
- Il existe des algorithmes plus efficaces qui permettent de trier en $\mathcal{O}(n \log n)$, voire $\mathcal{O}(n)$ sous certaines hypothèses …
- Démos complémentaires :

  https://www.toptal.com/developers/sorting-algorithms