cours_4

February 13, 2020

0.1 Efficience d'un algorithme

0.1.1 Taille d'un problème

L'efficacité d'un algorithme dépend généralement la taille des données en entrée:

- Taille d'un tableau, d'une liste, d'un fichier
- Longueur d'une chaîne de caractères
- Quantité d'informations (nombres de bits) pour coder une information

Il est donc pertinent d'étudier l'efficience d'un algorithme par rapport à la taille des données en entrée.

0.1.2 Cas où *n* est "petit"

Pour des problèmes de taille petite:

- L'architecture de la machine est prépondérante
- Les "caches" (pré-calculs) sont utilisés implicitement pour accélérer les calculs.

Il est difficile de comparer deux algorithmes dans ce cas là.
=> Mesure fine du temps d'éxecution (ex.: module *timeit*).

0.1.3 Cas où *n* est grand

Pour des problèmes de grande taille:

- Les "caches" sont insuffisants pour accélérer les calculs.
- On doit l'évolution de *Ops* en fonction de *n*:
  - Nombre de cases parcourues
  - Nombre d'échanges pour un tri en place
  - Nombre de comparaisons

Comme nos algorithmes ont:

- des branchements conditionnels `Si <condition> alors <instructions alors> Sinon <instructions sinon>`
- des boucles "Tant que" `Tant que <condition> faire <instructions>`

le nombre d'opérations élémentaires Ops(n) varie même pour n fixé:

- Suivant si l'on effectue $<$ *condition* $>$ $+$ $<$ *instructions alors* $>$ **ou** $<$ *condition* $>$ $+$ $<$ *instructions sinon* $>$
- Suivant le nombre d'itérations de $<$ *condition* $>$ $+$ $<$ *instructions* $>$

On parlera donc de:

- Borne sup : "pire des cas", ie trouver une configuration de la donnée qui maximise *Ops(n)*.
- Borne inf : "meilleur des cas", ie trouver une configuration de la donnée qui minimise *Ops(n)*.

Pour *n* fixé:

$$Meilleur <= Raliste <= Pire$$

On regarde ainsi l'évolution de ses bornes (cet encadrement) en fonction de *n*.
=> Les notations de *Landau*.
Elles vont nous permettrent de comparer les algorithmes.

0.2 Notations de Landau

0.2.1 Notation 'grand o' pour la borne sup

$$\mathcal{O}(f(n)) := \{g : \mathbb{N} \to \mathbb{R}^* | \exists c \in \mathbb{R}^*, n_0 \in \mathbb{N} \backslash \forall n \geq n_0, g(n) \leq c\ f(n)\}$$

$Ops(n) \in \mathcal{O}(n)$ dire "Ops(n) est en grand o de n"
$\Rightarrow$ Borne supérieure asymptotique sur nombre d'opérations (au coef. mult. c près)

$$\text{Ex : } Ops(n) = 10n^2 - 5n + 2 \qquad \begin{array}{l} \text{— } Ops(n) \in \mathcal{O}(n^2) \\ \text{— } Ops(n) \notin \mathcal{O}(n) \\ \text{— } Ops(n) \in \mathcal{O}(n^3) \end{array}$$

Grand O

0.2.2 Notation 'grand omega' pour la borne inf

$$\Omega(f(n)) := \{g : \mathbb{N} \to \mathbb{R}^* | \exists c \in \mathbb{R}^*, n_0 \in \mathbb{N} \backslash \forall n \geq n_0, g(n) \geq c\ f(n)\}$$

$Ops(n) \in \Omega(n)$ dire "Ops(n) est en grand omega de n"
$\Rightarrow$ Borne inférieure asymptotique sur nombre d'opérations (au coef. mult. c près)

$$\text{Ex : } Ops(n) = 10n^2 - 5n + 2 \qquad \begin{array}{l} \text{— } Ops(n) \in \Omega(n^2) \\ \text{— } Ops(n) \in \Omega(n) \\ \text{— } Ops(n) \notin \Omega(n^3) \end{array}$$

Grand Omega

0.2.3 Notation 'grand theta'

$$\Theta(f(n)) \sim \Omega(f(n)) \bigcap \mathcal{O}(f(n))$$

$$\begin{aligned}\Theta(f(n)) := \quad & \{g : \mathbb{N} \to \mathbb{R}^*| \quad && \exists (c_1, c_2) \in \mathbb{R}^* \times \mathbb{R}^*, n_0 \in \mathbb{N} \\ & \text{tel que } \forall n \geq n_0, && c_1 f(n) \leq g(n) \leq c_2 f(n)\}\end{aligned}$$

$Ops(n) \in \Theta(n)$ dire "Ops(n) est en grand theta de n"
$\Rightarrow Ops(n)$ est encadrée par $f(n)$ "de l'ordre exact de"

$$\text{Ex : } Ops(n) = 10n^2 - 5n + 2 \qquad \begin{array}{l} \text{— } Ops(n) \in \Theta(n^2) \\ \text{— } Ops(n) \notin \Theta(n) \\ \text{— } Ops(n) \notin \Theta(n^3) \end{array}$$

Grand Theta

0.2.4 Echelle de comparaison (voir code)

notation	grandeur au plus
O(1)	module majoré par une constante
O(log(n))	logarithmique
O(n)	linéaire
O(n log(n))	quasi-linéaire
O(n^2)	quadratique
O(n^c)	polynomial d'ordre c
O(c^n)	exponentiel

0.2.5 Applications aux algorithmes déjà rencontrés

- Calcul du min/max: $\Theta(n)$
- Recherche d'un élément: $\Omega(1)$ et $O(n)$, en moyenne : $n/2$
- Tri par sélection: $\Theta(n^2)$
- Tri par insertion: $\Omega(n)$ et $O(n^2)$, en moyenne : $n^2/4$

0.3 Stratégie Diviser pour Régner

- Simplifier un problème en tâches plus simples
- Fusionner les résultats

0.3.1 Tri par fusion (merge sort)

Principe:

- T est un tableau de taille n
- Partitionner T en deux parties égales (taille n/2) -> Gauche, Droite
- Trier récursivement **Gauche** et **Droite** (parallèle possible)
- Fusionner **Gauche** et **Droite** en T

Tri Fusion: partitionnement

[ ]:
```python
def partitionnement (T):
    mil = len(T) // 2
    return T[:mil],  T[mil:]
```

[9]:
```python
# remarque
T = [1, 2, 3, 4]
G, D = T[:2], T[2:]
D[0] = 1000
T
```

[9]:
```
[1, 2, 3, 4]
```

Tri Fusion: Fusion Principe:

- Parcourir simultanément **Gauche** et **Droite**
- Recopier l'élement minimal entre les deux valeurs courantes
- Augmenter l'indice sur le tableau qui contenait ce minimum.
- Continuer jusqu'a avoir recopier **Gauche** et **Droite**

Algorithme 5: Fusion

Données : $G[1..g]$ et $D[1..d]$ deux tableaux *triés*, $T[1..g+d]$
Résultat : $T[1..g+d]$ est trié et contient les éléments de G et D

$i_t \leftarrow 1$; $i_g \leftarrow 1$; $i_d \leftarrow 1$;
tant que $i_g \leq g$ *et* $i_d \leq d$ **faire**
  si $G[i_g] \leq D[i_d]$ **alors**
    $T[i_t] \leftarrow G[i_g]; i_g \leftarrow i_g + 1$
  sinon
    $T[i_t] \leftarrow D[i_d]; i_d \leftarrow i_d + 1$
  fin
  $i_t \leftarrow i_t + 1;$
fin
tant que $i_g \leq g$ **faire** $T[i_t] \leftarrow G[i_g]; i_t \leftarrow i_t + 1; i_g \leftarrow i_g + 1;$
tant que $i_d \leq d$ **faire** $T[i_t] \leftarrow D[i_d]; i_t \leftarrow i_t + 1; i_d \leftarrow i_d + 1;$

Fusion

Voir codes:

- Algorithme
- Vérification

Tri Fusion: Preuves

- Terminaison: Fin de la récursivité
- Correction de l'algorithme = Correction de la fusion
- Efficience: Ops(n) = NbComp(n) $\in \Theta$(n log n)

Tri Fusion: Caractéristiques

- Tri par comparaison
- Tri stable
- Tri qui n'est pas en place $\ldots$ > Utilisation de la mémoire: $\Theta(n)$
- Donc un tri extrêmement efficace