Analyse d'algorithmes et programmation

- Introduction à l'analyse des algorithmes -

Kahina BOUCHAMA

Université de Versailles, Saint-Quentin en Yvelines

Février 2023

Table des matières

Qu'est ce qu'un algorithme

Un algorithme est une suite d'instructions bien définie qui prend en entrée une ou plusieurs données et qui retourne en sortie une ou plusieurs résultats. Les étapes de calcul de l'algorithme nous indiquent avec précision comment transformer une entrée en sortie.

Un bon algorithme doit vérifier ces critères:

- Lisible: l'algorithme doit être compréhensible même par un non-informaticien.
- Fini: l'algorithme ne doit pas boucler indéfiniment.
- Précis et non ambiguë: chaque élément de l'algorithme ne doit pas porter à confusion.
- Concis: un algorithme ne doit pas dépasser une page. Si c'est le cas, il faut décomposer le problème en plusieurs sous-problèmes.
- Structuré: un algorithme doit être composé de différentes parties facilement identifiables.

Un algorithme doit aussi être correcte et efficace. En effet:

- Un algorithme est dit **correct** ou encore **complet**, si pour chaque instance du problème à résoudre, l'algorithme se termine en produisant la bonne sortie.

  Au contraire, on dit qu'un algorithme n'est **pas correct**, s'il ne termine pas pour certaines instances, ou bien s'il produit une sortie erronée dans certains cas.

- Un algorithme est censé **résoudre le problème** pour lequel il a été conçu **de la manière la plus efficace possible**.

Notion d'efficacité:

L'efficacité d'un algorithme se mesure par plusieurs paramètres. Parmi ceux ci, on cite:

- La complexité en temps (durée du calcul);
- La complexité mémoire (consommation mémoire vive);
- ...

Pourquoi le tri ?

- Un algorithme de tri sert à construire à partir d'une liste de nombres $[a_0, \ldots, a_{(n-1)}]$ reçue en entrée et d'un opérateur de comparaison ($\leq$, $\geq$, ...), une permutation $[a'_0, \ldots, a'_{(n-1)}]$ de la liste, telle que:

$$a'_0 \leq a'_1 \leq \ldots \leq a'_{(n-1)}.$$

- Il existe plusieurs algorithmes de tri, avec différentes complexités, ce qui fait d'eux un très bon outil pour se familiariser avec l'analyse de la complexité.

Le principe du tri par insertion:

Ayant un tableau L, cet algorithme sépare L en deux parties: une partie triée et une partie non encore triée.

Le tri par insertion

Le principe du tri par insertion:

A chaque itération, un élément de la partie non triée est sélectionné puis inséré dans la partie triée, tout en préservant l'ordre. Cette opération est répétée jusqu'à ce que tous les éléments de la partie non triée soient traités.

Ses avantages:

- Très simple à implémenter;
- Présente de bonne performances lorsque le tableau à trier est de petite taille;
- C'est un algorithme de tri sur place (tout se fait au niveau du même tableau), donc l'espace requis est minimal.

Ses inconvénients:

- Sa complexité quadratique en la taille du tableau le rend moins pratique pour des tableaux de grande dimension.

Algorithme du tri par insertion

Algorithm 1 Tri-insertion

1: **Entrées:** Un tableau L à trier, de taille n.
2: **Sortie:** Le tableau L trié.

3: **for** $i = 1$ to $(n - 1)$ **do**
4: element = $L[i]$
5: $j = i - 1$
6: **while** $(j \geq 0)$ et $(L[j] >$ element$)$ **do**
7: $L[j + 1] = L[j]$
8: $j = j - 1$
9: **end while**
10: $L[j + 1]$= element
11: **end for**

Exemple:

Soit **L = [4,2,7,3,5,6]** le tableau à trier. On considère que l'élément L[0] est déjà trié. On représentera par la couleur bleue, la partie triée du tableau, et par la couleur **noire** la **partie non triée**. La case rouge correspond à l'élément en cours d'insertion.

- On compare $L[1] = 2$ aux éléments de la partie trié, ici ça sera avec $L[0] = 4$.

4	2	7	3	5	6

  Puisque $2 < 4$, on insère 2 avant la valeur 4. On passe après à l'insertion du prochain élément, qui est $L[2] = 7$.

2	4	7	3	5	6

- On compare $L[2] = 7$ aux éléments de la partie trié, ici ça sera avec les éléments $\{2, 4\}$. Ici on a $7 > 4$, donc le 7 est inséré après le 4. On remarque qu'à chaque itération la partie du tableau qui se trouve à gauche de l'élément à insérer est triée.

Exemple:

2	4	7	3	5	6

On passe alors à l'insertion de l'élément $L[3] = 3$.

- $3 < 7$ et $3 < 4$, mais $3 > 2$. On insère alors 3 avant $\{4, 7\}$, et après 2.

2	3	4	7	5	6

- $5 < 7$, mais $5 > 4$. On insère alors 5 après 4.

2	3	4	5	7	6

- $6 < 7$, mais $6 > 5$. On insère alors 6 après 5.

2	3	4	5	6	7

Le tableau est maintenant trié.

Analyse de l'algorithme du tri par insertion

Critère de justesse

Il s'agit de prouver qu'à la fin de l'exécution de l'algorithme, le tri par insertion a effectué correctement la tâche pour laquelle il a été sollicité: trier correctement le tableau en entrée.

En algorithmique, on peut prouver mathématiquement qu'un algorithme est correct (valide). La technique utilisé est basée sur la définition d'un invariant de boucle.

invariant de boucle

Un invariant de boucle est une propriété qui est vraie avant et après chaque itération.

Voici les trois étapes pour prouver qu'une propriété est un invariant de boucle:

- **Initialisation:** il faut démontrer que l'invariant de boucle est vrai avant la première itération de boucle.
- **Conservation:** il faut prouver que si l'invariant de boucle est vrai avant une itération, alors il le reste également après.
- **Terminaison:** il faut prouver qu'une fois la boucle terminée, l'invariant est toujours vérifié, ce qui traduit le fait que la boucle réalise bien la tâche souhaitée.

Justesse du tri par insertion

Prouvons que la propriété "le sous tableau $L[0, ..., i-1]$ est trié" est un invariant de boucle pour l'algorithme du tri par insertion.

- initialisation: pour $i = 1$, on a $L[0, ..., i-1] = L[0]$, un seul élément est considéré trié, alors la propriété est vraie;
- conservation: la boucle extérieure déplace les éléments $L[i-1], L[i-2], ..$ d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour l'élément en cours d'insertion $L[i]$. Après ces décalages, on insère $L[i]$ à la bonne position. Finalement, une itération de la boucle extérieure permet de passer d'un tableau $L[0, ..., i]$ non-trié à un tableau trié.
- terminaison la boucle extérieure prend fin lorsque $i > (n-1)$, c'est-à-dire $i = n$. En substituant n à i dans l'invariant de boucle défini, on obtient que le sous-tableau $L[0, ..., n-1]$ est trié. Ce dernier correspond au tableau en entier. D'où, la justesse de l'algorithme du tri par insertion.

Le temps d'exécution d'un algorithme

- La durée d'exécution d'un algorithme dépend de la taille de son entrée:
  Données de petites taille $\Longrightarrow$ petit temps d'exécution.
- La taille des entrées dépend du problème à traiter (le nombre d'éléments à traiter, le nombre total de bits nécessaires à la représentation de l'entrée, ...).
- Il est impératif de décrire de manière claire la taille des entrées, avant de procéder à l'analyse de la complexité d'un algorithme.

Remarque: La complexité d'un algorithme peut aussi dépendre de la structure particulière de l'entrée et pas seulement de sa taille. On peut avoir des temps d'exécution très variés sur des entrées ayant des valeurs différentes, mais de même taille.

La complexité temporelle

Le temps d'exécution d'un algorithme pour une entrée donnée est le nombre d'opérations élémentaires exécutées.

Pour analyser le temps d'exécution de l'algorithme de tri par insertion, on va considérer que:

- Le tableau à trier est de taille n;
- Chaque ligne i de l'algorithme a un temps d'exécution constant c_i, on notera par t_i le nombre de fois que cette ligne est exécutée;
- Lorsqu'une boucle se termine, la condition de la boucle est exécutée une fois de plus que les instructions internes de la boucle.

Complexité du tri par insertion

- Le temps d'exécution d'un algorithme est la somme des temps d'exécution de ses instructions.
- Une instruction i qui a un coût c_i et qui est exécutée t_i fois, compte pour $c_i \times t_i$ dans le temps d'exécution total.

Pour l'algorithme du tri par insertion, on a:

Lignes	Coûts	Nombre de répétitions
for $i = 1$ to $(n1)$ do	c_1	n
element = $L[i]$	c_2	$n-1$
$j = i1$	c_3	$n-1$
while $(j \geq 0)$ et $(L[j] > element)$ do	c_4	$\sum_{i=1}^{n-1} t_i$
$L[j+1] = L[j]$	c_5	$\sum_{i=1}^{n-1} (t_i - 1)$
$j = j1$	c_6	$\sum_{i=1}^{n-1} (t_i - 1)$
$L[j+1]$ = element	c_7	$n-1$

Complexité du tri par insertion

Soit $T(n)$ la complexité du tri par insertion en fonction de n. Nous avons:

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{i=1}^{n-1} t_i + c_5 \cdot \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7 \cdot (n-1)$$

En analysant la formule ci-dessus, on s'intéressera à deux cas:

1. Le tableau L en entrée est déjà trié (par ordre croissant);
2. Le tableau L est trié par ordre décroissant.

Analyse du premier cas:

L est déjà triée par ordre croissant. C'est le cas le plus favorable. En effet:

- $\forall i,\ i = \overline{1, n}$, $L[j] <$clé pour $j = i - 1$. Donc, on n'exécute jamais la deuxième boucle, alors $t_i = 1, \forall i$. On obtient les simplifications suivantes:

$$
\begin{aligned}
T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{i=1}^{n-1} 1 + c_5 \cdot \sum_{t=1}^{n-1} (1-1) + c_6 \sum_{t=1}^{n-1} (1-1) + c_7 \cdot (n-1) \\
&= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\
&= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7)
\end{aligned}
$$

- Finalement, $T(n)$ est de la forme $a.n + b$.

On dit alors que la complexité est **linéaire en** n.

Analyse du deuxième cas:

L est triée par ordre décroissant. C'est le cas le plus défavorable. En effet:

- $\forall i,\ i = \overline{1,n}$, on compare chaque élément $L[i]$ avec chaque élément de $L[0,\ldots,i-1]$. Donc, $t_i = i, \forall i$. De plus:

$$\sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} - n = \frac{n^2-n}{2}$$

$$\sum_{i=1}^{n-1} (i-1) = \frac{n^2-n}{2} - (n-1) = \frac{n^2-3n+2}{2}$$

On obtient alors:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \frac{n^2-n}{2} + c_5 \frac{n^2-3n+2}{2} + c_6 \frac{n^2-3n+2}{2} + c_7 \cdot (n-1) \\ &= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 - \frac{c_4}{2} - \frac{3c_5}{2} - \frac{3c_6}{2} + c_7)n - (c_2 + c_3 - c_5 - c_6 + c_7) \end{aligned}$$

- Finalement, $T(n)$ est de la forme $a.n^2 + b.n + c$.

On dit alors que la complexité est **quadratique en** n.

Analyse du cas défavorable et du cas moyen

- En analyse de complexité, on s'intéresse souvent au temps maximal qui mettra l'algorithme renvoyer un résultat sur une entrée de taille n. On appelle cela le cas le plus défavorable ;

- Le cas le plus défavorable est plus intéressant que ce qu'on appelle le cas moyen, car il fournit une borne supérieure du temps d'exécution pour une entrée quelconque ;

- Pour certains algorithmes, le cas le plus défavorable peut arriver souvent, exemple: recherche d'un élément non existant dans un tableau ;

- Il arrive que le cas moyen et aussi mauvais que le cas le plus défavorable. Exemple : tri par insertion appliqué sur un tableau où la moitié des éléments de $L[0, \ldots, i-1]$ seront inférieurs à $L[i]$ et l'autre moitié lui seront supérieurs . Par conséquent, $t_i = i/2$ en moyenne. Dans ce cas, la complexité en temps $T(n)$ sera à nouveau une fonction quadratique de n.

Ordre de grandeur du temps d'exécution

Pour analyser la complexité du tri par insertion, nous avons fait recours à quelques simplifications:

- Les coûts réels remplacés par les constantes (c_i);
- Les coûts abstraits repris sous une forme ($a.n + b$) ou bien ($a.n^2 + b.n + c$).

L'ordre de grandeur du temps d'exécution permet d'avoir une idée du temps d'exécution lorsque n est vraiment grand.

Pour le tri par insertion, on a:

- Pour $(n \gg 0)$, on a $(a.n^2 \gg b.n)$. On dit alors que $(a.n^2)$ est le terme dominant. On va donc le garder;
- Pour $(n \gg 0)$, on a $(n^2 \gg a)$. Nous garderons alors que (n^2) et on écrira que le temps d'exécution du tri par insertion dans le cas le plus défavorable est de $\Theta(n^2)$.

Pour comparer des algorithmes:

- On dit qu'un algorithme est plus efficace qu'un autre si l'ordre de grandeur de son temps d'exécution est inférieur que celui de l'autre.

- ce n'est pas toujours vrai, pour des entrées de petite taille.

- On considérera alors qu'**un algorithme en $\Theta(n)$ est plus efficace qu'un autre en $\Theta(n^2)$.**

Performances asymptotiques

Lorsqu'on souhaite étudier le comportement d'un algorithme face à des données de grandes tailles, seul l'ordre de grandeur du temps d'exécution est significatif. On parle alors d'*étude des performances asymptotiques de l'algorithme.*

En gros, on s'intéresse à la manière dont augmente le temps d'exécution lorsque la taille des données augmente indéfiniment.

Formellement, on définit les notations Θ et O, comme suit:

Notation Θ

Pour une fonction donnée $g(n)$, on a:

$$\Theta(g(n)) = \left\{ f(n) \ / \ \exists(c_1, c_2, n_0) \text{ positifs, tels que: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n); \forall n \geq n0 \right\}$$

$f(n) = \Theta(g(n))$ signifie que $f(n) \in \Theta(g(n))$.

Exemple: $3n^2 - 4n = \Theta(n^2), 3n^3 \neq \Theta(n^2)$.

Notation O

Quand on ne dispose que d'une borne supérieure asymptotique pour une fonction $g(n)$, on utilise la notation O, définie par:

$$O(g(n)) = \left\{ f(n) \ / \ \exists(c,\ n_0) \text{ positifs, tels que: } \quad 0 \leq f(n) \leq c_g(n); \forall n \geq n0 \right\}$$

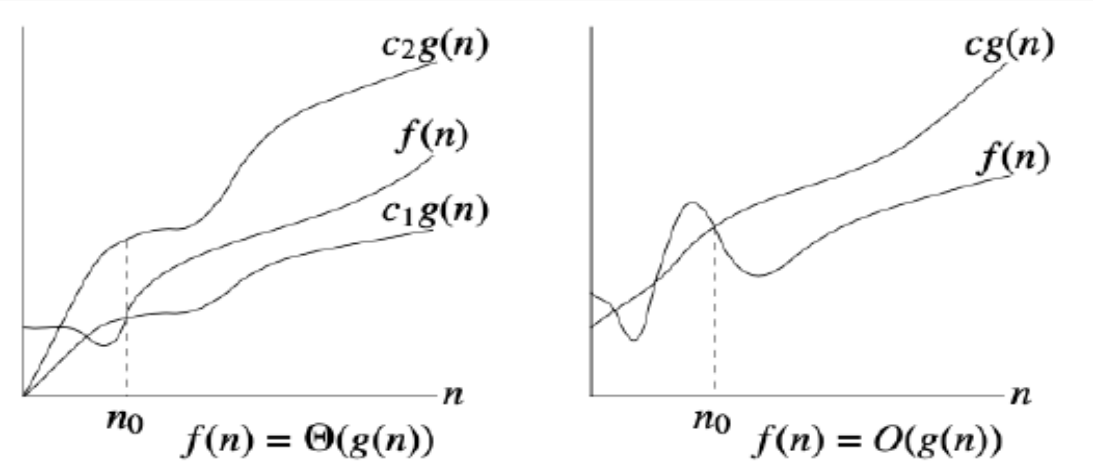

Figure: Les notations asymptotiques

Notons que $\Theta(g(n)) \subseteq O(g(n))$, ce qui signifie que que la notation Θ est plus forte. D'où,

$$f \in \Theta(g(n)) \Longrightarrow f \in O(g(n)).$$

Exemple: $f(n) = an + b = O(n^2), \ \ pour \ \ a > 0.$

Classes de complexité

Soit $f(n)$ la complexité de l'algorithme, où n est la taille de l'entrée. Alors:

- Si $f(n) = O(1)$, la complexité est dite constante.
- Si $f(n) = O(logn)$, la complexité est dite logarithmique.
- Si $f(n) = O(n)$, la complexité est dite linéaire.
- Si $f(n) = O(n^2)$, la complexité est dite quadratique.
- Si $f(n) = O(n^k)$, la complexité est dite polynomiale.
- Si $f(n) = O(a^n)$, la complexité est dite exponentielle.
- Si $f(n) = O(nlogn)$, la complexité est dite quasi-linéaire.