Université Assane SECK de Ziguinchor

Unité de Formation et de Recherche des Sciences et Technologies

Département d'Informatique

ANALYSE DES ALGORITHMES

Licence 1 Math - Physique – Informatique

Janvier 2021

©Youssou DIENG

ydieng@univ-zig.sn

Ce chapitre a pour but de vous familiariser avec la démarche que nous adopterons quand il s'agira de réfléchir à la conception et à l'analyse des algorithmes. Nous commencerons par étudier l'algorithme du tri par insertion, dont la problématique a été exposée au chapitre 1.

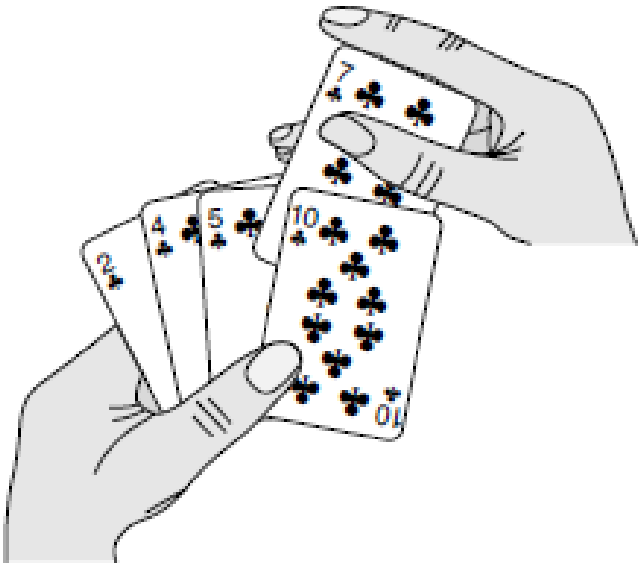
L'analyse permettra d'introduire une notation qui met en valeur la façon dont le temps d'exécution croît avec le nombre d'éléments à trier. Après avoir étudié le tri par insertion, nous présenterons le paradigme « diviser pour régner » pour la conception d'algorithmes. Nous nous servirons de cette technique pour développer l'algorithme du tri par fusion. Nous terminerons par l'analyse du temps d'exécution du tri par fusion.

1 - LE TRI PAR INSERTION

Le tri par insertion qui est notre premier algorithme, résout la problématique du tri exposée au chapitre 1 :

Entrée : Suite de n nombres « $a_1, a_2, \ldots, a_n$ ».

Sortie : permutation (réorganisation) « $a'_1, a'_2, \ldots, a'_n$ » de la suite donnée en entrée, de façon que « $a'_1 \leq a'_2 \leq \ldots \leq a'_n$ ».

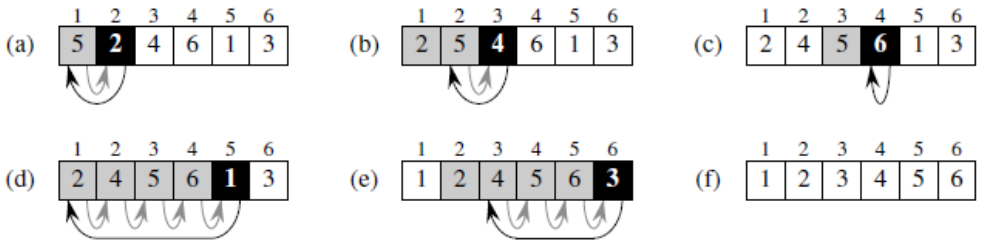


Notre algorithme pour le tri par insertion se présente sous la forme d'une procédure appelée TRI-INSERTION. Elle prend comme paramètre un tableau A[1 . . n] qui contient une séquence à trier, de longueur n. (Dans le code, le nombre n d'éléments de A est noté longueur[A]). Les nombres donnés en entrée sont triés sur place : ils sont réorganisés à l'intérieur du tableau A, avec tout au plus un nombre constant d'entre eux stocké à l'extérieur

du tableau à tout instant. Lorsque TRI-INSERTION se termine, le tableau d'entrée A contient la séquence de sortie triée.

1.1 - L'Algorithme

```
TRI-INSERTION(A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3           ▷ Insère A[j] dans la séquence triée A[1 . . j − 1].
4           i ← j − 1
5           tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                    i ← i − 1
8           A[i + 1] ← clé
```



Pour A = 5, 2, 4, 6,1, 3, les figures (a)–(e) illustrent les Itérations de la boucle pour des lignes lines 1–8 sur l'algorithme. À chaque itération, la case noire renferme la clé lue dans A[j] et cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5 sur l'algorithme). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8).

L'indice j indique la « carte courante » en cours d'insertion dans la main. Au début de chaque itération de la boucle pour, indicée par j, le sous-tableau composé des éléments A[1 . . j − 1] correspond aux cartes qui sont déjà dans la main. Les éléments A[j + 1. . n] correspondent aux cartes qui sont encore sur la table et les éléments A[1 . . j − 1] sont les éléments qui occupaient initialement les positions 1 à j − 1, mais qui depuis ont été triés.

1.2 - Invariants de boucle et validité du tri par insertion

L'invariant de boucle est tel que au début de chaque itération de la boucle pour des lignes 1 à 8 le sous-tableau A[1 . . j − 1] est composé des éléments qui occupaient initialement les positions

A[1 . . j−1] qui sont maintenant triés. Ces éléments nous aident à comprendre pourquoi un algorithme est correct.

1.3 - Correction d'un algorithme

Montrer la correction d'un algorithme revient à montrer trois choses, concernant un invariant de boucle :

1) ***Initialisation :*** L'invariant de boucle est vérifié avant la première itération de la boucle.
2) ***Conservation :*** Si l'invariant de boucle est vérifié avant une itération de la boucle, montrer qu'il le reste vérifié avant l'itération suivante.
3) ***Terminaison :*** Une fois que la boucle s'arrête, l'invariant fournit une propriété utile qui aide à montrer la validité de l'algorithme.

IL faut remarquer que si les deux premières propriétés sont vérifiées, alors l'invariant est vrai avant chaque itération de la boucle. Il existe une très forte ressemblance avec la récurrence mathématique dans laquelle, pour démontrer qu'une propriété est vraie, vous montrez que :

1) la propriété est vraie pour une valeur initiale,
2) si elle est vraie pour le rang N, alors elle l'est pour le rang N + 1 (phase inductive).

1.1.3.1 - Initialisation : Cas du Tri par insertion

On veut montrer que l'invariant est vérifié avant la première itération de la boucle, c'est-à-dire pour j = 2. En effet pour j = 2, le sous-tableau A[1 . . j−1] se compose uniquement de l'élément A[1] qui est, en fait, l'élément originel de A[1]. Ce sous-tableau est trié (c'est une trivialité). L'invariant de boucle est donc vérifié avant la première itération de la boucle.

1.1.3.2 - Conservation : Cas du Tri par insertion

On veut montrer que chaque itération conserve l'invariant. En effet, supposons qu'avant l'itération j, l'invariant de boucle est vérifié. Montrons que juste après la jieme itération l'invariant de boucle reste vérifié.

On sait que le corps de la boucle **pour** fonctionne en déplaçant A[j−1], A[j−2], A[j−3], etc. d'une position vers la droite; jusqu'à ce qu'on trouve la bonne position pour A[j] (lignes 4–7), auquel cas on insère la valeur de A[j] (ligne 8). Donc, la bonne position pour A[j] est l'entrée j-k+1 tel que $A[j-k] \leq A[j] \leq A[j-k+1]$. Le tableau A[1...j-1] est donc trié après cette opération et le sous-tableau A[1 . . j − 1] est composé des éléments qui occupaient initialement les positions A[1 . . j−1]. L'invariant est donc conservé après chaque itération.

1.1.3.3 - Terminaison : Cas du Tri par insertion

La boucle pour prend fin quand j dépasse n (c'est-à-dire quand j = n+1). En substituant n+1 à j dans la formulation de l'invariant de boucle, on a que le sous-tableau A[1 . . n] se compose des éléments qui appartenaient originellement à A[1 . . n] mais qui ont été triés depuis lors. Vu que le sous-tableau A[1 . . n] n'est autre que le tableau complet, alors le tableau tout entier est trié, et donc l'algorithme est correct.

2 - ANALYSER UN ALGORITHME

Analyser un algorithme consiste à prévoir les ressources nécessaires à cet algorithme. Parfois, les ressources à prévoir sont :

- la mémoire,
- la largeur de bande d'une communication ou
- le processeur (mais, le plus souvent, c'est le temps de calcul qui nous intéresse).

C'est en analysant plusieurs algorithmes susceptibles de résoudre le problème, on arrive aisément à identifier le plus efficace.

2.1 - Modèle de calcul

Pour pouvoir analyser un algorithme, il faut avoir un modèle de la technologie qui sera employée, un modèle pour les ressources de cette technologie et pour leurs coûts. Le modèle de calcul considéré est générique est basé sur une machine à accès aléatoire (RAM) à processeur unique.

1.2.1.1 - Modèle RAM

Dans le modèle RAM, les instructions sont exécutées l'une après l'autre, sans opérations simultanées. Les instructions du modèle RAM sont :

- arithmétique (addition, soustraction, multiplication, division, modulo, partie entière, partie entière supérieure),
- transfert de données (lecture, stockage, copie) et
- instructions de contrôle (branchement conditionnel et inconditionnel, appel de sous-routine et sortie de sous-routine).

Chacune de ces instructions a un temps d'exécution constant. On supposera qu'il existe une limite à la taille de chaque mot de données. Ainsi, pour des entrées de taille n, nous supposerons que les entiers sont représentés par $c \lg n$ bits pour une certaine constante $c \geq 1$.

Nous choisissons $c \geq 1$ pour que chaque mot puisse contenir la valeur n. Nous pourrons ainsi indexer les divers éléments de l'entrée et c doit être une constante pour éviter que la taille du mot augmente de manière arbitraire.

2.2 - Analyse du tri par insertion

La durée d'exécution de TRI-INSERTION dépend de l'entrée. Ainsi, le tri d'un millier de nombres prend plus de temps que le tri de trois nombres. TRI-INSERTION peut demander des temps différents pour trier deux entrées de même taille, selon qu'elles sont déjà plus ou moins triées partiellement. En général, le temps d'exécution d'un algorithme croît avec la taille de l'entrée. On a donc pris l'habitude d'exprimer le temps d'exécution d'un algorithme en fonction de la taille de son entrée. Il faut juste définir plus précisément ce que signifient « temps d'exécution » et « taille de l'entrée ».

1.2.2.1 - Taille de l'entrée

La Taille de l'entrée est très dépendante du problème étudié. Pour de nombreux problèmes, tels que le tri ou le calcul de transformées de Fourier discrètes c'est: le nombre d'éléments constituant l'entrée (par exemple la longueur n du tableau à trier). Pour beaucoup d'autres problèmes, comme la multiplication de deux entiers, la meilleure mesure de la taille de l'entrée est le nombre total de bits nécessaire à la représentation de l'entrée dans la notation binaire habituelle.

Parfois, il est plus approprié de décrire la taille de l'entrée avec deux nombres au lieu d'un seul. Par exemple, si l'entrée d'un algorithme est un graphe, la taille de l'entrée peut être décrite par le nombre de sommets et le nombre d'arcs.

1.2.2.2 - Temps d'exécution

Le temps d'exécution est le nombre **d'opérations élémentaires**, ou « **étapes** », exécutées. Il est commode de définir la notion d'étape de façon qu'elle soit le plus indépendante possible de la machine. Pour le moment, nous adopterons le point de vue suivant. L'exécution de chaque ligne de pseudo code demande un temps constant. Deux lignes différentes peuvent prendre des temps différents, mais chaque exécution de la i-ème ligne prend un temps ci, ci étant une constante. Ce point de vue est compatible avec le modèle RAM et reflète la manière dont le pseudo code serait implémenté sur la plupart des ordinateurs réels.

Nous commencerons par présenter la procédure TRI-INSERTION avec le « coût » temporel de chaque instruction et le nombre de fois que chaque instruction est exécutée.

En effet, pour tout j = 2, 3, . . . , n, où n = longueur[A], soit tj le nombre de fois que le test de la boucle tant que, en ligne 5, est exécuté pour cette valeur de j. Quand une boucle pour ou tant que se termine normalement (c'est-à-dire, suite au test effectué dans l'en tête de la boucle), le test est exécuté une fois de plus que le corps de la boucle. On suppose que les commentaires ne sont pas des instructions exécutables et qu'ils consomment donc un temps nul.

Le temps d'exécution est la somme des temps d'exécution de chaque instruction exécutée. Une instruction qui demande c_i étapes et qui est exécutée n fois compte pour c_in dans le temps d'exécution total.

TRI-INSERTION (A)		*coût*	*fois*
1	**pour** $j \leftarrow 2$ à *longueur*[A]	c_1	n
2	**faire** *clé* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Insère $A[j]$ dans la suite triée $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**tant que** $i > 0$ et $A[i] >$ *clé*	c_5	$\sum_{j=2}^{n} t_j$
6	**faire** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$ *clé*	c_8	$n-1$

Pour calculer le temps d'exécution $T(n)$, on additionne les produits des colonnes ***coût*** et ***fois***, ce qui donne:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) .$$

Le cas le plus favorable: Ce cas se présente lorsque tableau est déjà trié. En effet, pour tout $j = 2, 3, \ldots, n$, on trouve alors que $A[i]$ *clé* en ligne 5 quand i a sa valeur initiale de $j - 1$. Donc $tj = 1$ pour $j = 2, 3, \ldots, n$, et le temps d'exécution associé au cas optimal est:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Ce temps d'exécution est sous la forme an + b, *a et b sont des constantes dépendant des coûts c_i des instructions.* Le temps d'exécution *T(n) est donc une* ***fonction linéaire.***

Cas le plus défavorable: ce cas se présente lorsque le tableau est trié dans l'ordre décroissant. On doit comparer chaque élément $A[j]$ avec chaque élément du sous-tableau trié $A[1 . . j - 1]$, et donc $tj = j$ pour $j = 2, 3, . . . , n$. Si l'on remarque que:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \text{et} \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Alors on a :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Le temps d'exécution T(n) est donc sous la forme $an^2 + bn + c$, a, b et c étant des constantes qui dépendent des coûts ci des instructions ; T(n) est donc une fonction quadratique de n.

En résumé: Nous nous sommes intéressés aux deux cas suivants:

- **le cas le plus favorable** (le tableau en entrée est déjà trié)
- **le cas le plus défavorable** (le tableau en entrée est trié en sens inverse).

Dans la suite de ce cours, nous allons nous limiter uniquement sur le temps d'exécution du cas le plus défavorable. C'est le temps d'exécution maximal pour une entrée de taille n. En effet, le temps d'exécution du cas le plus défavorable est une borne supérieure du temps d'exécution d'une entrée quelconque. Donc connaître cette valeur nous permettra d'avoir la certitude que l'algorithme ne mettra jamais plus de temps que cette limite. Pour certains algorithmes, le cas le plus défavorable survient assez souvent. Par exemple, prenons les cas d'une recherche d'information dans une base de données. Si cette information n'existe pas dans la base le cas le plus défavorable se présentera souvent pour l'algorithme de recherche. Il n'est pas rare que le « cas moyen » soit presque aussi mauvais que le cas le plus défavorable.

1.2.2.3 - Ordre de grandeur

Des hypothèses simplificatrices ont été utilisées pour faciliter notre analyse de la procédure TRI-INSERTION.

1. On a d'abord ignoré le coût réel de chaque instruction en employant les constantes c_i pour représenter ces coûts.

2. On a ensuite observé que même ces constantes nous donnent plus de détails que nécessaire. En effet, le temps d'exécution du cas le plus défavorable est $an^2 + bn + c$, a, b et c étant des constantes qui dépendent des coûts c_i des instructions. Nous avons donc ignoré non seulement les coûts réels des instructions, mais aussi les coûts abstraits c_i.

Nous allons à présent utiliser une simplification supplémentaire. Ce qui nous intéresse vraiment, c'est le taux de croissance, ou ordre de grandeur, du temps d'exécution. On ne considérera donc que le terme dominant d'une formule (par exemple an^2), puisque les termes d'ordre inférieur sont moins significatifs pour n grand. On ignorera également le coefficient constant du terme dominant, les facteurs constants sont moins importants que l'ordre de grandeur pour ce qui est de la détermination de l'efficacité du calcul pour les entrées volumineuses. On écrira donc que le tri par insertion, par exemple, a dans le cas le plus défavorable un temps d'exécution de $\Theta(n^2)$ (prononcer « théta de n-deux »).

En résumé…

On considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur. Compte tenu des facteurs constants et des termes d'ordre inférieur, cette évaluation peut être erronée pour des entrées de faible volume. En revanche, pour des entrées de taille assez grande, un algorithme en $\Theta(n^2)$, par exemple, s'exécute plus rapidement, dans le cas le plus défavorable, qu'un algorithme en $\Theta(n^3)$.