Algorithmique et structures de données

Chap 1. Notion d'algorithme

Déf : Un algorithme est une procédure de calcul parfaitement définie, c.à.d. une séquence d'étapes. L'algorithme accepte une ou plusieurs entrées (paramètres) et produit un résultat (sortie).
Ex : tri d'un tableau.
En entrée une séquence de nombres $\{a_1,...,a_n\}$ dans une structure de type liste ou tableau.
En sortie : une permutation $\{a_{\pi(1)},..., a_{\pi(n)}\}$ de l'entrée tel que la séquence soit triée (en ordre croissant ou décroissant).
L'objectif d'une étape est d'insérer le *i*-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit. C'est un "tri par insertion".

Soit le tableau suivant : T = [12, 7, 0, 5, 8]

0	*1*	*2*	*3*	*4*
12	7	0	5	8
7	12	0	5	8
0	7	12	5	8
0	5	7	12	8

0	*1*	*2*	*3*	*4*
7	12	0	5	8
0	7	12	5	8
0	5	7	12	8
0	5	7	8	12

```
  procédure tri_insertion(tableau T, entier n)
(1)  pour j de 2 à n  # equiv.for
       # mémoriser T[j] dans x = clé
(2)    x ← T[j]
      # décaler à droite les éléments de T[0]..T[i-1] > x (en partant de T[i-1])
(3)    i ← j-1
(4)    tant que i > 0 et T[i - 1] > x  # equiv. while
(5)        T[i+1] ← T[i]
(6)        i ← i - 1
      fin tant que
       # placer x dans le "trou" que ça a laissé
(7)   T[i+1] ← x
    fin pour
  fin procédure
```

trace : tableau [4, 5, 1]

E1 : j=2, T[j] =5, x= 5, i=1, T[i] =4 et la comparaison 5>4 =>T[2]=5

E2 : j=3, T[j]=1, x = 1, i=2, T[i]=5 et la comparaison 5>1 =>T[3] <- T[2] <-5 =>[4,5,5]

E3 :i=1, 1<4, T[2] <- T[1] <-4 => [4, 4, 5]

E4 : i=0, T[1]=1 => [1,4,5]

j=4 => stop.

Défs :

1. l'algorithme est stable (conserve l'ordre d'apparition d'éléments égaux),

2. Il est dit en place car n'utilisant pas de tableau auxiliaire, (il modifie directement la structure qu'il est en train de trier). Il peut recevoir les éléments à trier un par un sans perdre son efficacité.

Chap2. Notion de complexité algorithmique

Comme il existe sans doute plusieurs façons (algorithmes) de trier un tableau, il va nous falloir les comparer. Pour cela on défini la complexité algorithmique.

- Permet de savoir si un algorithme est plus efficace qu'un autre
- L'analyse de la complexité est indépendante des ressources physiques utilisées (processeur, temps d'accès mémoire...)

Pour comparer 2 algos, on mesure la complexité de chacun d'entre eux. Elle est définie comme le nombre d'étapes nécessaires pour résoudre le problème dans le pire des cas. Ce résultat permettra de choisir l'algo le plus efficace et de prévoir les ressources nécessaires. Les ressources critiques sont :

A. la mémoire requise (moins important car augmente plus vite que la vitesse des proc.)
B. le temps d'exécution (plus critique).

Le temps d'exécution dépendra de (i) l'algorithme (ii) de la taille des données (iii) de la cadence du processeur (notée en hertz = nbre d'opérations/seconde)

	cadence	mémoire	Signif.
1976	1 Mhz	8 ko	K =10^3
1984	8 Mhz	512 ko	
1992	33 Mhz	4 Mo	M=10^6
1998	400 Mhz	64 Mo	
2000	1 Ghz	512 Mo	
2007	3 Ghz	4 Go	G=10^9
2017	4 Ghz	8 -16 Go	

Entre 1975 et 2007 la vitesse a été multipliée par un facteur de 10^3 alors que la mémoire l'a été d'un facteur de 10^6. Les architectures actuelles font plafonner la vitesse pour des raisons de refroidissement et de temps de parcours du signal électronique pour exécuter l'instruction.

Comme en général le temps d'exécution dépend de la taille des données à traiter par l'algorithme, on a pris l'habitude de calculer le temps d'exécution comme une fonction de cette taille.

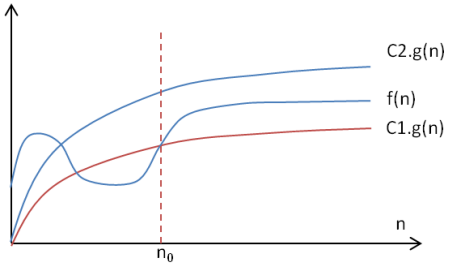
A.1. Notation

O(g(n)) = une borne approchée asymptotique

Etant donné une fonction g : N -> R, on défini l'ensemble *O* (g(n)) comme l'ensemble des fonctions qui peuvent être encodées entre $C_1g(n)$ et $C_2g(n)$:

$O\ (g(n)) = \{f(n) \mid \exists\ C_1, C_2, n_0>0, \forall n \geq n_0, 0 \leq C_1g(n) \leq f(n) \leq C_2g(n)\}$

Si $f \in O(g(n))$ on écrit : $f(n) = Og(n)$
Ce qui signifie : f(n) est égale à g(n) à un facteur constant près. g(n) est une borne approchée asymptotique pour f.



O(g(n)) = une borne supérieure asymptotique

$O(g(n)) = \{f(n) \mid \exists C, n_0>0, \forall n \geq n_0, 0 \leq f(n) \leq Cg(n)\}$. Si $f \in O(g(n))$ on écrit : $f(n) = Og(n)$

On a donc une borne sup. g(n) à une constante C près.

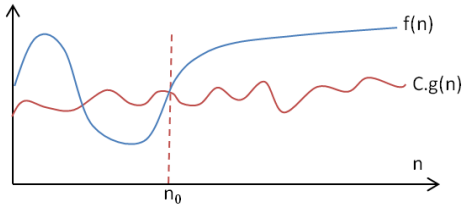
Ceci sert à estimer le temps maximum d'exécution

Ω(g(n)) = une borne inférieure asymptotique

$\Omega(g(n)) = \{f(n) \mid \exists C, n_0>0, \forall n \geq n_0, 0 \leq C\ g(n) \leq f(n)\}$

On a donc une borne inf. g(n) à une constante C près.

Ceci sert à estimer le temps minimum d'exécution



On ne considérera que le cas de machines à 1 proc. Sinon, il conviendrait d'inclure le temps de communication entre processeurs.

Complexité croissante 	**temps**	**Type complexité**	**Temps pour n = 50**
	$O(1)$	Constante	10 nanos
	$O(\log(n))$	logarithmique	20 nanos
	$O(n)$	linéaire	500 nanos
	...	...	
	$O(n^2)$	Quadratique	25 µs
	$O(2^{poly(n)})$	Exponentielle	130 jours
	$O(n!)$	Factorielle	10^{48} années

Exemple

Reprenons le cas du tri par insertion. Quelle est sa complexité ? En pratique, on calcule le nombre d'exécution de chaque ligne du code et on en déduit le temps d'exécution total.

ligne 1 : n-1 fois => C1(n-1) où les Ci représentent un cout abstrait...

ligne 2 : n-1 fois => C2(n-1)

ligne 3 : n-1 fois => C3(n-1)

ligne 4 : $\sum_{j=2}^{n} t_j$ => C4($\sum_{j=2}^{n} t_j$)

ligne 5 : $\sum_{j=2}^{n}(t_j - 1)$ => C5($\sum_{j=2}^{n}(t_j - 1)$)

ligne 6 : $\sum_{j=2}^{n}(t_j - 1)$ => C6($\sum_{j=2}^{n}(t_j - 1)$)
ligne 7 : n-1 => C7(n-1)

Comme on considère le pire des cas c.à.d. que les valeurs de tj sont identiques au temps j, la clé x est toujours plus petite que tous les éléments à sa gauche :

On a donc : (C1 + C2 + C3 + C7)(n-1)+ C4($\sum_{j=2}^{n} j$)+(C5+C6)($\sum_{j=2}^{n}(j - 1)$)

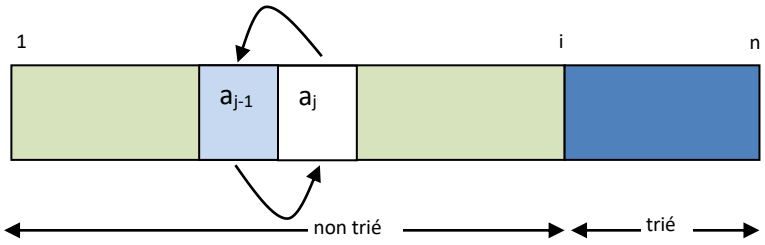
Sachant que : $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$ on a : $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$
de plus : $\sum_{j=2}^{n-1} j = \sum_{2}^{n}(j - 1) = \frac{n(n-1)}{2}$

on a : (C1 + C2 + C3 + C7)(n-1) + C4($\frac{n(n+1)}{2} - 1$) + (C5+C6)($\frac{n(n-1)}{2}$)
On néglige les couts abstraits (Ci) de chaque instruction et on ne retient que les termes dominants.
O(Cn + C $\frac{n(n+1)}{2}$) = O(n²)

Dans le pire des cas (le tableau est entièrement trié à l'envers), il effectue n²/2 opérations de comparaisons et d'affectations. Dans le meilleur des cas (le tableau est trié) il en effectue seulement n (les lignes 4,5,6 et 7 disparaissent !).
La complexité du tri par insertion est donc quadratique : O(n²). Ce qui n'est pas très efficace. Cependant, si le tableau d'entrée est petit et presque trié, l'algo. reste linéaire en n O(n) et reste le plus rapide de tous les algos de tri !

Tri par échange ou sélection : le tri à bulle O(n²).

Un tableau tab[ ] est en mémoire centrale. Admettons qu'une partie du tableau soit déjà triée. On parcourt plusieurs fois la partie non triée en réordonnant les couples (a_j, a_{j-1}) non classés en inversant leurs rangs au besoin (si $a_{j-1} > a_j$) dans la partie non triée. Quand on atteint la frontière trié/non trié on recommence au début de la partie non triée.



On aura donc 2 boucles imbriquées pour parcourir le tableau :

pour i variant de n à 1 faire
pour j variant de 2 à i faire
.....

```
Algorithme Tri_a_Bulles
i , j , n, temp = Entiers naturels
Entrée : Tab  Tableau d'Entiers naturels de 1 à n éléments
Sortie  : Tab  Tableau d'Entiers naturels de 1 à n éléments
début
pour i de n jusquà 2 faire        // recommence une sous-suite (a1, a2, ... , ai)
    pour j de 2 jusquà i faire    // échange des couples non classés de la sous-suite
    si Tab[ j-1 ] > Tab[ j ] alors // aj-1et aj non ordonnés
       temp ← Tab[ j-1 ] ;
       Tab[ j-1 ] ← Tab[ j ] ;
       Tab[ j ]  ← temp           //on échange les positions de aj-1et aj
    Finsi
   finpour
finpour
Fin Tri_a_Bulles
```

Complexité du tri à bulles est égale à la somme des n termes suivants (i = n, i = n-1, ...)
C = (n-2) + 1 + ([n-1] – 2) + 1 + ... + 1 + 0 = (n-1) + (n-2) + ... + 1 = n(n-1)/2 , car c'est la somme des n-1 premiers entiers. La complexité en temps est donc bien de l'ordre de n^2.