

Algorithmique et Approche Fonctionnelle

Cours 5 : Complexité et Algorithmes de Tri

13 Octobre 2008

Complexité

notion de complexité

l'analyse de la **complexité** d'un algorithme consiste à évaluer les ressources consommées par l'algorithme lors de l'exécution

deux critères d'évaluation

- le coût en **temps** (nombre d'opérations)
- le coût en **espace** (quantité de mémoire)

principes de base sur la complexité

- caractériser la quantité de ressources consommées en **fonction de la taille des données** sur lesquelles l'algorithme est appliqué
- évaluer le coût exact est difficile, on exprimera donc seulement un **ordre de grandeur**
- on s'intéresse aux opérations **les plus significatives**

si f est la fonction caractérisant *exactement* le coût d'un algorithme et n la taille des données

- on s'intéresse à la façon dont croît $f(n)$ lorsque n croît
- on va montrer que $f(n)$ ne croît pas plus vite qu'une autre fonction $g(n)$

Du point de vue mathématique, on dit que la fonction f est **dominée asymptotiquement** par la fonction g ce qui se note $f = O(g)$ et qui signifie :

$$f = O(g) \text{ ssi } \exists k, \exists n_0, \forall n, n > n_0 \Rightarrow f(n) \leq k.g(n)$$

Croissance	Ordre de grandeur
linéaire	$O(n)$
quadratique	$O(n^2)$
polynomiale	$O(n^p)$
exponentielle	$O(p^n)$
logarithmique	$O(\log(n))$

taille	Complexité						
	1	$\log_2(n)$	n	$n.\log_2(n)$	n^2	n^3	2^n
$n = 10^2$	1 μ s	6,6 μ s	0,1 ms	0,6 ms	10 ms	1 s	$4.10^{16} a$
$n = 10^3$	1 μ s	9,9 μ s	1 ms	9,9 ms	1 s	16,6 mn	∞
$n = 10^4$	1 μ s	13,3 μ s	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^5$	1 μ s	16,6 μ s	0,1 s	1,6 s	2,7 h	31,7 a	∞
$n = 10^6$	1 μ s	19,9 μ s	1 s	19,9 s	11,5 j	31700 a	∞

algorithmes de tri

Algorithmes de Tri

- le tri est une opération importante en Informatique
- il améliore l'efficacité de nombreuses opérations

nous allons étudier trois algorithmes de tri sur les listes :

- le tri par insertion
- le tri rapide
- le tri fusion

les algorithmes de tri ne doivent pas dépendre d'une relation d'ordre particulière, on doit pouvoir utiliser le même algorithme avec une relation d'ordre quelconque :

- par ordre croissant

```
# let pluspetit x y = x <= y;;
```

- par ordre décroissant

```
# let plusgrand x y = x >= y;;
```

- par un ordre plus "exotique"

```
let exotique x y =  
  match (x mod 2 = 0) , (y mod 2 = 0) with  
  (true , true) | (false, false) -> x <= y  
  | (true , _) -> true  
  | (_, true) -> false;;
```

Tri pas Insertion

principe

cet algorithme de tri suit de manière naturelle la structure récursive des listes

soit l une liste à trier :

- 1 si l est vide alors elle est déjà triée
- 2 sinon, l est de la forme $x::s$ et,
 - on trie **récurivement** la suite s et on obtient une liste triée s'
 - on **insert** x au bon endroit dans s' et on obtient une liste triée

insertion

- la fonction `insérer` permet d'insérer un élément x dans une liste l
- si la liste l est triée alors x est inséré au bon endroit
- on prend pour le moment `<=` comme relation d'ordre

```
# let rec inserer x l =  
  match l with  
  [] -> [x]  
  | y::s -> if x <= y then x::l else y::(inserer x s);;  
val inserer : 'a -> 'a list -> 'a list
```

```
# inserer 5 [3;7;10];;  
- : int list = [3; 5; 7; 10]
```

évaluation de inserer 5 [3;7;10]

```

[3; 7; 10] ≠ [], y = 3, s = [7; 10], 5 > 3 ⇒ 3::(inserer 5 [7;10])
[7; 10] ≠ [], y = 7, s = [10], 5 ≤ 7 ⇒ 3::5::[7;10]
⇒ 3::[5;7;10]
⇒ [3;5;7;10]

```

```

inserer 5 [3;7;10]

```

on ajoute une **relation d'ordre** comme argument à la fonction inserer

```

# let rec inserer rel_ordre x l =
  match l with
  [] -> [x]
  | y::s ->
    if rel_ordre x y then x::l
    else y::(inserer rel_ordre x s);;

val inserer :
('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>

```

```

# inserer pluspetit 5 [3; 7; 10];;

```

```

- : int list = [3; 5; 7; 10]

```

```

# inserer plusgrand 3 [5; 2; 1];;

```

```

- : int list = [5; 3; 2; 1]

```

```

# inserer exotique 3 [2; 4; 6; 1; 5];;

```

```

- : int list = [2; 4; 6; 1; 3; 5]

```

on utilise la fonction inserer pour réaliser un tri par insertion d'une liste

```

# let rec trier rel_ordre l =
  match l with
  [] -> []
  | x::s -> inserer rel_ordre x (trier rel_ordre s);;

val trier : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>

# trier pluspetit [6; 1; 9; 4; 3];;
- : int list = [1; 3; 4; 6; 9]

# trier plusgrand [6; 1; 9; 4; 3];;
- : int list = [9; 6; 4; 3; 1]

# trier exotique [4;1;7;3;10;4;2];;
- : int list = [2; 4; 4; 10; 1; 3; 7]

```

évaluation de trier (\leq) [6;4;1;5]

```

trier ( $\leq$ ) [6;4;1;5]
x = 6, s = [4;1;5]  $\Rightarrow$  inserer 6 (trier ( $\leq$ ) [4;1;5])
x = 4, s = [1;5]    $\Rightarrow$  inserer 6 (inserer 4 (trier ( $\leq$ ) [1;5]))
x = 1, s = [5]      $\Rightarrow$  ... (inserer 1 (trier ( $\leq$ ) [5]))
x = 5, s = []       $\Rightarrow$  ... (inserer 5 (trier ( $\leq$ ) []))
                    $\Rightarrow$  ... (inserer 5 [])
                    $\Rightarrow$  inserer 6 (inserer 4 (inserer 1 [5]))
                    $\Rightarrow$  inserer 6 (inserer 4 [1;5])
                    $\Rightarrow$  inserer 6 [1;4;5]
                    $\Rightarrow$  [1;4;5;6]

```

- on s'intéresse au **nombre de comparaisons** nécessaires pour trier une liste l de n éléments
- l'algorithme revient à :
 - trier une liste de longueur $n - 1$ (le reste de l)
 - + insérer le premier élément dans cette liste triée de $n - 1$ éléments
- ceci revient donc à :
 - insérer un élément dans une liste triée de $n - 1$ éléments
 - + insérer un élément dans une liste triée de $n - 2$ éléments
 - + ...
 - + insérer un élément dans une liste triée de 2 éléments
 - + insérer un élément dans une liste triée de 1 élément

l'insertion d'un élément p dans une liste l de n éléments oblige à comparer :

meilleur cas : une seule comparaison (p est plus petit que le premier élément de l)

cas le pire : n comparaisons (p est plus grand que tous les éléments de l)

en moyenne : $n/2$ comparaisons

coût total d'un tri par insertion :

meilleur cas : $1 + 1 + 1 + 1 + \dots + 1$ ($n - 1$ fois)
donc un coût en $O(n)$

cas le pire : $(n - 1) + (n - 2) + \dots + 1$, soit : $n(n - 1)/2$
donc un coût en $O(n^2)$

en moyenne : $(n - 1)/2 + (n - 2)/2 + \dots + 1/2$, soit : $n(n - 1)/4$
donc un coût en $O(n^2)$

Tri Rapide

soit une liste l à trier :

- 1 si l est vide alors elle est triée
- 2 sinon, choisir un élément p de la liste (le premier par exemple) nommé **le pivot**
- 3 **partager** l en deux listes g et d contenant les autres éléments de l qui sont plus petits (resp. plus grands) que la valeur du pivot p
- 4 **trier récursivement** g et d , on obtient deux listes g' et d' triées
- 5 on renvoie la liste $g'@[p]@d'$ (qui est bien triée)

la fonction suivante permet de **partager** une liste l en deux sous-listes g et d contenant les éléments de l plus petits (resp. plus grands) qu'une valeur donnée p

```
#let rec partage p l =
  match l with
  [] -> ([], [])
  |x::s -> let g,d = partage p s in
           if x<=p then (x::g , d) else (g , x::d) ;;
val partage : 'a -> 'a list -> 'a list * 'a list = <fun>
```

```
# partage 5 [1;9;7;3;2;4];;
- : int list * int list = ([1; 3; 2; 4], [9; 7])
```

évaluation de la fonction partage

évaluation de partage 5 [1;9;3;7]

```
partage 5 [1;9;3;7]
=> let g1,d1=partage 5 [9;3;7] in (1::g1,d1)
=> let g2,d2=partage 5 [3;7] in (g2 , 9::d2)
=> let g3,d3=partage 5 [7] in (3::g3 , d3)
=> let g4,d4=partage 5 [] in (g4 , 7::d4)
=> [],[]
=> let g4,d4=([],[]) in (g4 , 7::d4)
=> ([], [7])
=> let g3,d3=([],[7]) in (3::g3 , d3)
=> ([3] , [7])
=> let g2,d2=([3],[7]) in (g2 , 9::d2)
=> ([3] , [9;7])
=> let g1,d1=([3],[9;7]) in (1::g1,d1)
=> ([1;3] , [9;7])
```

tri rapide

```
# let rec tri_rapide l =
  match l with
  [] -> []
  | p::s -> let g , d = partage p s in
            (tri_rapide g@[p]@(tri_rapide d) ) ;;
val tri_rapide : 'a list -> 'a list = <fun>
```

```
# tri_rapide [5; 1; 9; 7; 3; 2; 4];;
- : int list = [1; 2; 3; 4; 5; 7; 9]
```

- on s'intéresse toujours au nombre de comparaisons
- celles-ci ont lieu lors du partage de la liste par rapport au pivot

- 1 le coût du partage d'une liste de n éléments est n comparaisons
- 2 le coût du tri de la liste 1 de n éléments est donc de
 - le coût du partage de s en (g, d) (i.e. $n - 1$ comparaisons)
 - + le coût du tri de g
 - + le coût du tri de d

ce coût dépend des **longueurs de g et d**

dans le **pire des cas** l'une des deux listes est vide à chaque partage

```

tri_rapide [1;2;3;4]
⇒ (tri_rapide [])@[1]@(tri_rapide [2;3;4])
⇒ [1]@(tri_rapide [2;3;4])
⇒ [1]@(tri_rapide [])@[2]@(tri_rapide [3;4])
⇒ [1]@[2]@(tri_rapide [3;4])
⇒ [1]@[2]@(tri_rapide [])@[3]@(tri_rapide [4])
⇒ [1]@[2]@[3]@(tri_rapide [])@[4]@(tri_rapide [])
⇒ [1]@[2]@[3]@[4]

```

- le nombre de comparaisons est de

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$$

- la complexité au pire est donc en $O(n^2)$

dans le **meilleur cas** les listes g et d sont toujours d'égale longueur

```

tri_rapide [4;2;6;3;5;1;7]
⇒ (tri_rapide [2;3;1])@[4]@(tri_rapide [6;5;7])
⇒ (tri_rapide [1])@[2]@(tri_rapide [3]) @[4]@
  (tri_rapide [5])@[6]@(tri_rapide [7])
⇒ [1]@[2]@[3]@[4]@[5]@[6]@[7]

```

- le nombre de comparaisons est de l'ordre de :

$$(n - 1) + 2 * [(n - 1)/2] + 4 * [(n - 1)/4] + \dots$$

- la profondeur de l'arbre est de l'ordre de $\log_2(n)$
- la complexité dans le meilleur des cas est donc en $O(n \log_2(n))$

dans le **cas en moyenne** c'est aussi $O(n \log_2(n))$

Tri Fusion

soit une liste l à trier :

- ① si la liste est **vide**, ou si elle a **un** élément, alors elle est triée
- ② sinon, **partager** la liste en deux sous-listes $l1$ et $l2$ *plus petites*
- ③ **trier récursivement** $l1$ et $l2$, on obtient deux listes triées $t1$ et $t2$
- ④ **fusionner** les deux listes $t1$ et $t2$

la fonction `couper` coupe une liste l en deux sous-listes plus petites que l dont les longueurs diffèrent au plus d'un

```
# let couper l =
  let rec couper_rec (l1,l2) l =
    match l with
    | [] -> l1,l2
    | x::l -> couper_rec (x::l2,l1) l
  in
  couper_rec ([],[]) l

val couper : 'a list -> 'a list * 'a list = <fun>

# couper [4;1;8;5;10;4];;
- : int list * int list = ([4; 5; 1], [10; 8; 4])
```

évaluation de la fonction `couper`

évaluation de `couper [3;1;5;7]`

```
couper [3;1;5;7]
=> couper_rec ([],[]) [3;1;5;7]
=> couper_rec (3::[],[]) [1;5;7]
=> couper_rec (1::[],[3]) [5;7]
=> couper_rec (5::[3],[1]) [7]
=> couper_rec (7::[1],[5;3]) []
=> ([7;1],[5;3])
```

fusion de deux listes

la fonction `fusion` fusionne deux listes triées

```
#let rec fusion l1 l2 =
  match l1,l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x::s1 , y::s2 ->
    if x<=y then x::(fusion s1 l2)
    else y::(fusion l1 s2);;

val fusion : 'a list -> 'a list -> 'a list = <fun>
```

évaluation de fusion [1;3;5] [2;5;6]

```

fusion [1;3;5] [2;6]
⇒ 1::(fusion [3;5] [2;6])
⇒ 1::2::(fusion [3;5] [6])
⇒ 1::2::3::(fusion [5] [6])
⇒ 1::2::3::5::(fusion [] [6])
⇒ 1::2::3::5::6(fusion [] [])
⇒ 1::2::3::5::6 []
⇒ ...
⇒ [1; 2; 3; 5; 6]

```

finalement, l'algorithme du tri fusion est défini par la fonction suivante

```

#let rec tri l =
  match l with
  [] | [_] -> l
  | _ -> let l1,l2 = couper l in fusion (tri l1) (tri l2);;
val tri : 'a list -> 'a list = <fun>

```

complexité du tri fusion

le tri fusion est un tri optimal, sa complexité dans le meilleur, le pire et le cas moyen est $O(n \log_2(n))$