Algorithmique et Approche Fonctionnelle de la Programmation

Cours 5 : Complexité et Algorithmes de Tri

12 octobre 2009

Louis Mandel
Université Paris-Sud 11
louis.mandel@lri.fr

D'après des transparents de Sylvain Conchon

Complexité

Notion de complexité

- L'analyse de la complexité d'un algorithme consiste à évaluer les ressources consommées par l'algorithme lors de son 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 consommée 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 le plus significatives

Ordre de grandeur

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 qui se note $f = O(g)$ et qui signifie :

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

Exemples

Croissance	Ordre de grandeur
constante	$O(1)$
logarithmique	$O(log(n))$
linéaire	$O(n)$
quadratique	$O(n^2)$
polynomiale	$O(n^p)$
exponentielle	$O(p^n)$

Exemples

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

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

Relation d'ordre

Les algorithmes de tri ne doivent as 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 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 par 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 :: l'` et,
     - ▷ on trie récursivement la liste `l'` et on obtient une liste triée `tl'`
     - ▷ on insère `x` au bon endroit dans `tl'` et on obtient une liste triée

Insertion

- La fonction `inserer` permet d'insérer un élément `x` dans une liste `l`
- Si la liste est triée alors `x` est inséré au bon endroit pour obtenir une nouvelle liste qui est toujours triée
- Pour le moment, on prend <= comme relation d'ordre

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

Évaluation de la fonction `inserer`

Évaluation de `inserer 5 [3; 7; 10]` ?

Fonction d'ordre supérieur : insertion paramétrée

- On ajoute une relation d'ordre comme paramètre de la fonction `inserer`

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

Insertion paramétrée : exemples

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

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

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

Trier une liste

- On utilise la fonction `inserer` pour réaliser un tri par insertion d'une liste

```
# let rec trier rel_ordre l =
    match l with
    | [] -> []
    | x::l' ->
      inserer rel_ordre x (trier rel_ordre l')
  ;;
val trier : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

Exemples

```
# let _ = trier pluspetit [ 4; 1; 7; 3; 10; 4; 2 ];;
- : int list = [1; 2; 3; 4; 4; 7; 10]
```

```
# let _ = trier plusgrand [ 4; 1; 7; 3; 10; 4; 2 ];;
- : int list = [10; 7; 4; 4; 3; 2; 1]
```

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

Évaluation de `trier (<=) [ 6; 4; 1; 5 ]` ?

Complexité du tri par insertion

- 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

Complexité du tri par insertion

- L'insertion d'un élément `x` dans une liste `l'` de m éléments nécessite des comparaisons :
  - ▷ meilleur cas : une seule comparaison (`x` est plus petit que le premier élément de `l'`)
  - ▷ pire cas : m comparaisons (`x` est plus grand que tous les éléments de `l'`)
  - ▷ en moyenne : $m/2$ comparaisons
- Coût total d'un tri par insertion :
  - ▷ meilleur cas : $1 + 1 + 1 + 1 + \ldots + 1$ ($n - 1$ fois)
    donc un coût en $O(n)$
  - ▷ cas le pire : $(n-1) + (n-2) + \ldots + 1$, soit : $n(n-1)/2$
    donc un coût en $O(n^2)$
  - ▷ en moyenne : $(n-1)/2 + (n-2)/2 + \ldots + 1/2$, soit : $n(n-1)/4$
    donc un coût en $O(n^2)$

Tri rapide

Principe

- 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)

Partage d'une liste

- 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::l' ->
      let g, d = partage p l' in
      if x <= p then (x::g, d)
      else (g, x::d)
  ;;
val partage : 'a -> 'a list -> 'a list * 'a list = <fun>
```

Évaluation de la fonction `partage`

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

- Évaluation de `partage 5 [ 1; 9; 3; 7 ]` ?

Tri rapide (quick sort)

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

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

Complexité de tri rapide

- On s'intéresse toujours au nombre de comparaisons
- Celles-ci ont lieu lors du partage de la liste par rapport au pivot
  - ▷ le coût du partage d'une liste de n éléments est n comparaisons
- Le coût du tri de la liste `l` de n éléments est donc
  le coût du partage de `l'` 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`

Complexité de tri rapide

- Dans le pire des cas l'une des deux listes est vide à chaque partage

```
tri_rapide [1; 2; 3; 4]
```
$\longrightarrow^*$ `(tri_rapide [])@[1]@(tri_rapide [2;3;4])`
$\longrightarrow^*$ `[1]@(tri_rapide [2;3;4])`
$\longrightarrow^*$ `[1]@((tri_rapide [])@[2]@(tri_rapide [3;4]))`
$\longrightarrow^*$ `[1]@([2]@(tri_rapide [3;4]))`
$\longrightarrow^*$ `[1]@([2]@((tri_rapide [])@[3]@(tri_rapide [4])))`
$\longrightarrow^*$ `[1]@([2]@([3]@(tri_rapide [4])))`
$\longrightarrow^*$ `[1]@([2]@([3]@((tri_rapide [])@[4]@(tri_rapide []))))`
$\longrightarrow^*$ `[1]@([2]@([3]@([4]@[])))`
$\longrightarrow^*$ `[1; 2; 3; 4]`

- Le nombre de comparaisons est de
  $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$
- la complexité au pire est donc en $O(n^2)$

Complexité de tri rapide

- Dans le meilleur des 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]@([6]@[5]@[7])
⟶* [1; 2; 3; 4; 5; 6; 7]
```

- Le nombre de comparaison est de l'ordre de :

$$(n-1) + 2 * ((n-1)/2) + 4 * ((n-1)/4) + \ldots$$

- La profondeur de l'arbre est de l'ordre de $\log_2(n)$
- La complexité dans le meilleur des cas est donc $n \log_2(n)$
- Dans le cas en moyenne c'est aussi $n \log_2(n)$

Algorithmes de Tri

Tri Fusion

Principe

- Soit une liste `l` à trier :
  - si la liste est vide ou si elle a un seul élément, alors elle est triée
  - sinon, partager la liste en deux sous-listes `l1` et `l2` plus petites
  - trier récursivement `l1` et `l2`, on obtient deux listes `tl1` et `tl2`
  - fusionner les deux listes `tl1` et `tl2`

Couper une liste en deux

- 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 l (l1,l2) =
      match l with
      | [] -> l1, l2
      | x :: l' -> couper_rec l' (x::l2,l1)
    in
    couper_rec l ([], [])
  ;;
val couper : 'a list -> 'a list * 'a list = <fun>
# let _ = 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 ]` ?

Fusion de deux listes

- Le fonction `fusion` fusionne deux listes triées

```
# let rec fusion l1 l2 =
    match l1, l2 with
    | [], _ -> l2
    | _, [] -> l1
    | x1::l1', x2::l2' ->
      if x1 <= x2 then x1 :: fusion l1' l2
      else x2 :: fusion l1 l2'
  ;;
val fusion : 'a list -> 'a list -> 'a list = <fun>
```

Évaluation de la fonction `fusion`

- Évaluation de `fusion [ 1; 3; 5 ] [ 2; 5; 6 ]` ?

Tri fusion

Finalement, l'algorithme de 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>
```

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