
Induction sur les listes

Planning

- Équations récursives sur les listes
- Complexité de fonctions sur les listes
 - Tri par insertion
 - Tri rapide
- Preuve de propriétés par récurrence sur les listes

Équations récursives sur les listes

On cherche à écrire des programmes selon les mêmes schémas récursifs vus pour les entiers, en commençant par l'équation :

$$f(l) = g(l, f(h(l)))$$

Exemple I : la longueur d'une liste

Problème : définir une fonction qui calcule la longueur d'une liste.

Type de la fonction recherchée :

`longueur` : 'a liste → entier

Équation récursive :

$$\text{longueur}(l) = g(l, \text{longueur}(h(l)))$$

On pose

$$h(l) = \text{reste}(l) \quad g(l, k) = 1 + k$$

Cas particulier :

$$l = \text{nil} \quad (\text{reste}(\text{nil}) \text{ n'est pas défini}).$$

Ceci donne :

```
longueur(l) = 0
             si liste_vide(l)
longueur(l) = 1 + longueur(reste(l))
             si liste_non_vide(l)
```

En OCAML :

```
# let rec longueur l = match l with
  [] -> 0
  | p::r -> 1 + longueur r;;
val longueur : 'a list -> int = <fun>
```

Exemple II : inversion d'une liste

Problème : définir une fonction qui construit la liste inverse d'une liste donnée.

Type de la fonction :

`inv_seq` : 'a liste → 'a liste

Équation récursive :

$$\text{inv_seq}(l) = g(l, \text{inv_seq}(h(l)))$$

On pose :

$$h(y) = \text{reste}(y) \quad g(l, z) = \text{ajout_fin}(\text{premier}(l), z)$$

avec `ajout_fin` une fonction que nous allons construire plus avant.

Cas particulier :

$$l = \text{nil} \quad (\text{premier}(\text{nil}) \text{ et } \text{reste}(\text{nil}) \text{ ne sont pas définis}).$$

Ceci donne :

```
inv_seq(l) = l
            si liste_vide(l)
inv_seq(l) = ajout_fin(premier(l), inv_seq(reste(l)))
            si liste_non_vide(l)
```

En OCAML :

```
# let rec inv_seq l = match l with
  [] -> []
  | p::r -> ajout_fin(p, inv_seq r);;
val inv_seq : 'a list -> 'a list = <fun>
```

Équations sur les listes avec plusieurs arguments

Même équation que sur les entiers :

$$f(y, x) = g(y, x, f(h(y), i(x)))$$

Exemple I : la concaténation

Problème : définir une fonction qui concatène deux listes données.

Type de la fonction :

`concat` : 'a liste × 'a liste → 'a liste

Équation réursive :

$$\text{concat}(y, x) = g(y, x, \text{concat}(h(y), i(x)))$$

On pose :

$$h(y) = \text{reste}(y) \quad i(x) = x$$

$$g(y, x, z) = \begin{cases} x & \text{si liste_vide}(y) \\ \text{cons}(\text{premier}(y), z) & \text{si liste_non_vide}(y) \end{cases}$$

Ceci donne :

$$\begin{aligned} \text{concat}(l, x) &= x \\ &\quad \text{si liste_vide}(l) \\ \text{concat}(l, x) &= \text{cons}(\text{premier}(l), \text{concat}(\text{reste}(l), x)) \\ &\quad \text{si liste_non_vide}(l) \end{aligned}$$

En OCAML :

```
# let rec concat(l1,l2) = match l1 with
  [] -> l2
  | p::r -> p::concat(r,l2);;
val concat : 'a list * 'a list -> 'a list = <fun>
```

Exemple II : ajout fin

Problème : définir une fonction qui ajoute un élément à la fin d'une liste.

Déclaration du type :

`ajout_fin` : 'a liste × 'a → 'a liste

On cherche :

$$\text{ajout_fin}(y, e) = g(y, e, \text{ajout_fin}(h(y), i(e)))$$

On pose :

$$h(y) = \text{reste}(y) \quad i(e) = e$$

$$g(y, e, z) = \begin{cases} \text{cons}(e, \text{nil}) & \text{si liste_vide}(y) \\ \text{cons}(\text{premier}(y), z) & \text{si liste_non_vide}(y) \end{cases}$$

Cas particulier :

$$l = \text{nil} \quad (\text{reste}(\text{nil}) \text{ n'est pas défini}).$$

Ceci donne :

$$\begin{aligned} \text{ajout_fin}(l, e) &= \text{cons}(e, \text{nil}) \\ &\quad \text{si liste_vide}(l) \\ \text{ajout_fin}(l, e) &= \text{cons}(\text{premier}(l), \text{ajout_fin}(\text{reste}(l), e)) \\ &\quad \text{si liste_non_vide}(l) \end{aligned}$$

En OCAML :

```
# let rec ajout_fin(l,e) = match l with
  [] -> [e]
  | p::r -> p::ajout_fin(r,e);;
val ajout_fin : 'a list * 'a -> 'a list = <fun>
```

Notions de complexité

Permet de mesurer la consommation des ressources lorsqu'on exécute un algorithme :

- le **coût en temps** (i.e. le nombre d'opérations nécessaires)
- le **coût en espace** (i.e. la quantité de mémoire nécessaire)

Ces deux critères ne sont pas totalement indépendants.

Principes de base

- Mesurer la complexité en **fonction de la taille des données** de l'algorithme.
- Evaluer le coût **exact** est difficile.
- Certaines opérations sont **négligeables** par rapport au coût total.
- On s'intéresse aux opérations dont le nombre évolue de façon significative lorsque la taille des données varie. Si f est une fonction caractérisant le coût d'un algorithme en fonction de la taille n des données, on s'intéresse à la façon dont croît $f(n)$ lorsque n croît. Plus précisément, on cherche généralement à montrer que $f(n)$ ne croît pas plus vite qu'une autre fonction $g(n)$.

Ordres de grandeur

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

Complexité sur les listes : problème du tri

Problème : définir une fonction qui associe à une liste d'éléments la liste formée des mêmes éléments mais triés par ordre croissant.

Déclaration du type :

```
tri : 'a liste → 'a liste
```

Hypothèse de travail : On dispose d'une fonction permettant de comparer deux éléments quelconques de type 'a.

Méthode par insertion

Idée : On ramène le problème $\text{tri}(l)$ au problème $\text{tri}(\text{reste}(l))$.

Équation réursive :

$$\text{tri_ins}(l) = g(l, \text{tri_ins}(h(l)))$$

On pose

$$h(l) = \text{reste}(l) \quad g(l, z) = \text{insertion}(\text{premier}(l), z)$$

Cas particulier :

$$l = \text{nil} \quad (\text{reste}(\text{nil}) \text{ n'est pas défini}).$$

Ceci donne :

$$\begin{aligned} \text{tri_ins}(l) &= l \\ &\quad \text{Si liste_vide}(l) \\ \text{tri_ins}(l) &= \text{insertion}(\text{premier}(l), \text{tri_ins}(\text{reste}(l))) \\ &\quad \text{Si liste_non_vide}(l) \end{aligned}$$

En OCAML :

```
# let rec tri_ins l = match l with
  [] -> []
  | p::r -> insertion(tri_ins r,p);;
val tri_ins : 'a list -> 'a list = <fun>
```

Insertion d'un élément dans une liste

Problème : définir une fonction qui permet d'insérer un élément "au bon endroit" d'une liste d'éléments triés par ordre croissant.

Déclaration du type :

```
insertion : 'a liste × 'a → 'a liste
```

Équation réursive :

$$\text{insertion}(l, e) = g(l, e, \text{insertion}(h(l), i(e)))$$

On pose :

$$h(l) = \text{reste}(l) \quad i(e) = e$$

$$g(l, e, z) = \begin{cases} \text{cons}(e, \text{nil}) & \text{Si liste_vide}(l) \\ \text{si } e \leq \text{premier}(l) \\ \quad \text{alors } \text{cons}(e, l) \\ \quad \text{sinon } \text{cons}(\text{premier}(l), z) & \text{Si liste_non_vide}(l) \end{cases}$$

Ceci donne :

```
insertion(l,e) = cons(e,nil)
                 si liste_vide(l)
insertion(l,e) = si e <= premier(l)
                 alors cons(e,l)
                 sinon cons(premier(l),insertion(reste(l),e))
                 si non_liste_vide(l)
```

En OCAML :

```
# let rec insertion(l,e) = match l with
  [] -> [e]
  | p::r -> if e <= p
            then e::l
            else p::insertion(r,e);;
val insertion : 'a list * 'a -> 'a list = <fun>
```

Complexité du tri par insertion

Pour trier une liste de n éléments on doit :

- Trier une liste de $n - 1$ éléments
 - Insérer un élément dans une séquence triée de $n - 1$ éléments
- C'est à dire
- Trier une liste de $n - 2$ éléments
 - Insérer un élément dans une séquence triée de $n - 2$ éléments
 - Insérer un élément dans une séquence triée de $n - 1$ éléments
- C'est à dire
- Insérer un élément dans une séquence triée de 1 élément
 - Insérer un élément dans une séquence triée de 2 éléments
 - ...
 - Insérer un élément dans une séquence triée de $n - 2$ éléments
 - Insérer un élément dans une séquence triée de $n - 1$ éléments

Complexité de l'insertion d'un élément dans une liste triée de n éléments

- Meilleur cas : 1 comparaison
- Cas le pire : n comparaisons

Coût du tri par insertion d'une liste de n éléments

- 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 : $\frac{n(n-1)}{2}$ donc un coût en $O(n^2)$

Mais le meilleur cas est très rare : il y en a juste un parmi les $n!$ listes de longueur n .

En moyenne, le tri par insertion se conduit mal : l'élément à insérer tombe à peu près au milieu de la liste déjà triée, ce qui donne une complexité moyenne pour le tri par insertion de $O(n^2)$.

Méthode rapide

Idee : On ramène le problème $\text{tri}(l)$ au problème $\text{tri}(l')$ avec l' une sous-liste de l qui n'est pas forcément $\text{reste}(l)$.

Comment ? On partage la liste l en deux sous-listes l_1 et l_2 par rapport à un pivot e de l de telle sorte que

$$\forall x \in l_1 \quad x \leq e \quad \text{et} \quad \forall y \in l_2 \quad e > y$$

La méthode satisfait l'équation

$$\text{tri_rap}(l) = \text{concat}(\text{tri_rap}(l_1), \text{cons}(e, \text{tri_rap}(l_2)))$$

Le partage d'une liste selon un pivot

Problème : définir une fonction qui associe à une liste l et à un élément e deux listes l_1 et l_2 t.q. $\forall x \in l_1 \quad x \leq e$ et $\forall y \in l_2 \quad e > y$.

Déclaration du type :

```
partage : 'a liste * 'a -> 'a liste * 'a liste
```

Équation réursive :

$$\text{partage}(l, e) = g(l, e, \text{partage}(h(l), e))$$

On pose :

$$h(l) = \text{reste}(l)$$

et

$$g(l, e, (z_1, z_2)) = \begin{cases} \text{si } \text{premier}(l) \leq e \\ \text{alors } (\text{cons}(\text{premier}(l), z_1), z_2) \\ \text{sinon } (z_1, \text{cons}(\text{premier}(l), z_2)) \end{cases}$$

Cas particulier :

$$l = \text{nil} \text{ (premier(nil) n'est pas défini).}$$

Ceci donne :

```
partage(l,e) = (nil,nil)
              si liste_vide(l)
partage(l,e) = soit (s1,s2) = partage(reste(l),e) dans
              si premier(l) ≤ e
                alors (cons(premier(l),s1),s2)
                sinon (s1,cons(premier(l),s2))
              si liste_non_vide(l)
```

En OCAML :

```
# let rec partage(l,e) = match l with
  [] -> ([],[])
  | p::r -> let (s1,s2)=partage(r,e) in
            if p<=e
              then (p::s1,s2)
              else (s1,p::s2);;
val partage : 'a list * 'a -> 'a list * 'a list = <fun>
```

En OCAML :

```
# let rec tri_rap(l)= match l with
  [] -> []
  | p::r -> let (s1,s2) = partage(r,p) in
            concat(tri_rap(s1), p::tri_rap(s2));;
val tri_rap : 'a list -> 'a list = <fun>
```

Complexité du tri rapide

Pour trier une liste de n éléments on doit :

1. Partager une liste de $n - 1$ éléments en deux sous-listes (s_1, s_2) ($n - 1$ comparaisons).
2. Trier la liste s_1 .
3. Trier la liste s_2 .

Mais quelles sont les longueurs de s_1 et de s_2 ?

Cas le meilleur

À chaque étape s_1 et s_2 ont la même longueur.

Exemple :

1. `tri_rap[4 ; 2 ; 6 ; 3 ; 5 ; 1 ; 7]` provoque deux appels
 - (a) `tri_rap[2 ; 3 ; 1]` provoque deux appels
 - i. `tri_rap[1]` provoque deux appels `tri_rap[]`
 - ii. `tri_rap[3]` provoque deux appels `tri_rap[]`
 - (b) `tri_rap[6 ; 5 ; 7]` provoque deux appels
 - i. `tri_rap[5]` provoque deux appels `tri_rap[]`
 - ii. `tri_rap[7]` provoque deux appels `tri_rap[]`

Le nombre de comparaisons est :

$$(n - 1) + 2 \times \left[\frac{n-1}{2} - 1 \right] + 4 \times \left[\frac{n-1}{4} - 1 \right] + \dots$$

La profondeur de l'arbre est de l'ordre de $\log_2(n)$.

La complexité dans le meilleur des cas est donc $O(n \times \log_2(n))$.

Cas le pire

À chaque étape l'une des listes s_1 ou s_2 est vide.

Exemple :

`tri_rap[1 ; 2 ; 3 ; 4 ; 5]` provoque deux appels
 1) `tri_rap[]`
 2) `tri_rap[2 ; 3 ; 4 ; 5]` provoque deux appels
 2.1) `tri_rap[]`
 2.2) `tri_rap[3 ; 4 ; 5]` provoque deux appels
 2.2.1) `tri_rap[]`
 2.2.2) `tri_rap[4 ; 5]` provoque deux appels
 2.2.2.1) `tri_rap[]`
 2.2.2.2) `tri_rap[5]` provoque deux appels
 2.2.2.2.1) `tri_rap[]`

Le nombre de comparaisons est :

$$(n-1) + (n-2) + \dots + 1 = \frac{n \times (n-1)}{2}$$

La complexité au pire est donc en $O(n^2)$.

Mais le cas le pire est très rare : en moyenne, le tri rapide se conduit bien, et a une complexité $O(n \times \log_2(n))$

Preuves de propriétés par récurrence

- On utilise la définition inductive de l'ensemble de listes
- On utilise la définition récursive d'une ou plusieurs fonctions et/ou propriétés.

Exemple I

Propriété à démontrer : pour toute liste l , `concat(l, nil) = l`

Preuve : par induction sur l .

- Cas de base : $l = \text{nil}$.

$$\text{concat}(\text{nil}, \text{nil}) = \text{nil}$$

- Cas inductif : $l = \text{cons}(p, r)$.

$$\begin{aligned} \text{concat}(l, \text{nil}) &= \\ \text{concat}(\text{cons}(p, r), \text{nil}) &= \\ \text{cons}(p, \text{concat}(r, \text{nil})) &=_{h,r} \\ \text{cons}(p, r) &= l \end{aligned}$$

Exemple II

Propriété à démontrer : pour tout élément e et toute liste l_1, l_2

$$\text{ajout_fin}(e, \text{concat}(l_1, l_2)) = \text{concat}(l_1, \text{ajout_fin}(e, l_2))$$

Preuve : par induction sur l_1 .

- Cas de base : $l_1 = \text{nil}$.

$$\begin{aligned} \text{ajout_fin}(e, \text{concat}(l_1, l_2)) &= \\ \text{ajout_fin}(e, \text{concat}(\text{nil}, l_2)) &= \\ \text{ajout_fin}(e, l_2) &= \\ \text{concat}(\text{nil}, \text{ajout_fin}(e, l_2)) &= \\ \text{concat}(l_1, \text{ajout_fin}(e, l_2)) & \end{aligned}$$

- Cas inductif : $l_1 = \text{cons}(p, r)$.

$$\begin{aligned} \text{ajout_fin}(e, \text{concat}(l_1, l_2)) &= \\ \text{ajout_fin}(e, \text{concat}(\text{cons}(p, r), l_2)) &= \\ \text{ajout_fin}(e, \text{cons}(p, \text{concat}(r, l_2))) &= \\ \text{cons}(p, \text{ajout_fin}(e, \text{concat}(r, l_2))) &=_{h,r} \\ \text{cons}(p, \text{concat}(r, \text{ajout_fin}(e, l_2))) &= \\ \text{concat}(\text{cons}(p, r), \text{ajout_fin}(e, l_2)) &= \\ \text{concat}(l_1, \text{ajout_fin}(e, l_2)) & \end{aligned}$$

Exemple III

Propriété à démontrer : pour tout élément e et toute liste l_1, l_2

$$\text{inv_seq}(\text{concat}(l_1, l_2)) = \text{concat}(\text{inv_seq}(l_2), \text{inv_seq}(l_1))$$

Preuve : par induction sur l_1 .

- Cas de base : $l_1 = \text{nil}$.

$$\begin{aligned} \text{inv_seq}(\text{concat}(l_1, l_2)) &= \\ \text{inv_seq}(\text{concat}(\text{nil}, l_2)) &= \\ \text{inv_seq}(l_2) &=_{\text{Exemple I}} \\ \text{concat}(\text{inv_seq}(l_2), \text{nil}) &= \\ \text{concat}(\text{inv_seq}(l_2), \text{inv_seq}(\text{nil})) &= \\ \text{concat}(\text{inv_seq}(l_2), \text{inv_seq}(l_1)) & \end{aligned}$$

- Cas inductif : $l_1 = \text{cons}(p, r)$.

$$\begin{aligned} \text{inv_seq}(\text{concat}(l_1, l_2)) &= \\ \text{inv_seq}(\text{concat}(\text{cons}(p, r), l_2)) &= \\ \text{inv_seq}(\text{cons}(p, \text{concat}(r, l_2))) &= \\ \text{ajout_fin}(p, \text{inv_seq}(\text{concat}(r, l_2))) &=_{h,r} \\ \text{ajout_fin}(p, \text{concat}(\text{inv_seq}(l_2), \text{inv_seq}(r))) &=_{\text{Exemple II}} \\ \text{concat}(\text{inv_seq}(l_2), \text{ajout_fin}(p, \text{inv_seq}(r))) &= \\ \text{concat}(\text{inv_seq}(l_2), \text{inv_seq}(\text{cons}(p, r))) &= \\ \text{concat}(\text{inv_seq}(l_2), \text{inv_seq}(l_1)) & \end{aligned}$$