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

Même équation que sur les entiers :

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

On pourra utiliser un type abstrait.

Le type abstrait `'a liste`

Domaine :

`'a liste`

Opérations de construction :

`nil` : `'a liste`

`cons` : `'a` $\times$ `'a liste` $\rightarrow$ `'a liste`

Opérations de test (une ou deux) :

`liste_vide` : `'a liste` $\rightarrow$ `booléen`

`liste_non_vide` : `'a liste` $\rightarrow$ `booléen`

Opérations d'accès :

`premier` : $\{l \in$ `'a liste` $|$ `non_vide`(l) $\} \rightarrow$ `'a`

`reste` : $\{l \in$ `'a liste` $|$ `non_vide`(l) $\} \rightarrow$ `'a liste`

Exemple I : la longueur d'une liste

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

Déclaration du type :

$$\texttt{longueur} : \texttt{'a liste} \rightarrow \texttt{entier}$$

Équation récursive :

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

On pose

$$h(l) = \texttt{reste}(l) \qquad g(l,k) = 1 + k$$

Cas particulier :

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

Ceci donne :

$$\begin{aligned} \texttt{longueur}(l) &= 0 \\ &\quad \text{si } \texttt{liste\_vide}(l) \\ \texttt{longueur}(l) &= 1 + \texttt{longueur}(\texttt{reste}(l)) \\ &\quad \text{si } \texttt{liste\_non\_vide}(l) \end{aligned}$$

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.

Déclaration du type :

$$\texttt{inv\_seq} : \texttt{'a liste} \rightarrow \texttt{'a liste}$$

Équation récursive :

$$\texttt{inv\_seq}(l) = g(l, \texttt{inv\_seq}(h(l)))$$

On pose :

$$h(y) = \texttt{reste}(y) \quad g(l,z) = \texttt{ajout\_fin}(\texttt{premier}(l), z)$$

Cas particulier :

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

Ceci donne :

$$\begin{aligned}\texttt{inv\_seq}(l) &= l \\ &\quad \text{si } \texttt{liste\_vide}(l) \\ \texttt{inv\_seq}(l) &= \texttt{ajout\_fin}(\texttt{premier}(l), \texttt{inv\_seq}(\texttt{reste}(l))) \\ &\quad \text{si } \texttt{liste\_non\_vide}(l)\end{aligned}$$

En OCAML :

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



```
# 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 concatenation

Problème : définir une fonction qui concatène deux listes.

Déclaration du type :

$$\texttt{concat} : \texttt{'a liste} \times \texttt{'a liste} \rightarrow \texttt{'a liste}$$

Équation récursive :

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

On pose :

$$h(y) = \texttt{reste}(y) \qquad i(x) = x$$

$$g(y, x, z) = \begin{cases} x & \text{si } \texttt{liste\_vide}(y) \\ \texttt{cons}(\texttt{premier}(y), z) & \text{si } \texttt{liste\_non\_vide}(y) \end{cases}$$

Ceci donne :

$$\begin{aligned} \texttt{concat}(l, x) &= x \\ &\quad \text{si } \texttt{liste\_vide}(l) \\ \texttt{concat}(l, x) &= \texttt{cons}(\texttt{premier}(l), \texttt{concat}(\texttt{reste}(l), x)) \\ &\quad \text{si } \texttt{liste\_non\_vide}(l) \end{aligned}$$

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 :

$$\texttt{ajout\_fin} : \texttt{'a liste} \times \texttt{'a} \rightarrow \texttt{'a liste}$$

On cherche :

$$\texttt{ajout\_fin}(y, e) = g(y, e, \texttt{ajout\_fin}(h(y), i(e)))$$

On pose :

$$\color{red}{h}(y) = \texttt{reste}(y) \qquad \color{red}{i}(e) = e$$

$$\color{magenta}{g}(y,e,z) = \begin{cases} \texttt{cons}(e,\texttt{nil}) & \text{si } \texttt{liste\_vide}(y) \\ \texttt{cons}(\texttt{premier}(y),z) & \text{si } \texttt{liste\_non\_vide}(y) \end{cases}$$

Cas particulier :

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

Ceci donne :

$$\texttt{ajout\_fin}(l,e) = \texttt{cons}(e,\texttt{nil})$$
$$\text{si } \texttt{liste\_vide}(l)$$

$$\texttt{ajout\_fin}(l,e) = \texttt{cons}(\texttt{premier}(l),\texttt{ajout\_fin}(\texttt{reste}(l),e))$$
$$\text{si } \texttt{liste\_non\_vide}(l)$$

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)

Ce 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 côut 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 croit 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 :

$$\texttt{tri} : \texttt{'a liste} \rightarrow \texttt{'a liste}$$

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 $\texttt{tri}(l)$ au problème $\texttt{tri}(\texttt{reste}(l))$.

Équation récursive :

$$\texttt{tri\_ins}(l) = g(l, \texttt{tri\_ins}(h(l)))$$

On pose

$$h(l) = \texttt{reste}(l) \qquad g(l, z) = \texttt{insertion}(\texttt{premier}(l), z)$$

Cas particulier :

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

Ceci donne :

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

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 :

$$\texttt{insertion} : \texttt{'a liste} \times \texttt{'a} \rightarrow \texttt{'a liste}$$

Équation récursive :

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

On pose :

$$h(l) = \texttt{reste}(l) \qquad i(e) = e$$

$$g(l,e,z) = \begin{cases} \texttt{cons}\ (e, \texttt{nil}) & \text{si } \texttt{liste\_vide}(l) \\ \texttt{si}\ e \leq \texttt{premier}(l) & \\ \quad \texttt{alors}\ \ \texttt{cons}(e, l) & \\ \quad \texttt{sinon}\ \ \texttt{cons}(\texttt{premier}(l), z) & \text{si } \texttt{liste\_non\_vide}(l) \end{cases}$$

Ceci donne :

$$\begin{array}{lcl} \texttt{insertion}(l, e) & = & \texttt{cons}\ (e, \texttt{nil}) \\ & & \text{si } \texttt{liste\_vide}(l) \\ \texttt{insertion}(l, e) & = & \texttt{si}\ e \leq \texttt{premier}(l) \\ & & \quad \texttt{alors}\ \ \texttt{cons}(e, l) \\ & & \quad \texttt{sinon}\ \ \texttt{cons}(\texttt{premier}(l), \texttt{insertion}(\texttt{reste}(l) \\ & & \text{si } \texttt{non\_liste\_vide}(l) \end{array}$$

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 élélements on doit :

- Trier une liste de $n-1$ élélements
- Insérer un élément dans une séquence triée de $n-1$ éléments

C'est à dire

- Trier une liste de $n-2$ élélements
- 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
- $\ldots$
- 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 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 : $1+1+1+1+\ldots+1$ ($n-1$ fois)
  donc un coût en $O(n)$
- Cas le pire : $(n-1)+(n-2)+\ldots+1$, soit : $\frac{n(n-1)}{2}$
  donc un coût en $O(n^2)$

Méthode rapide

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

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

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

La méthode satisfait l'équation

$$\texttt{tri\_rap}(l) = \texttt{concat}(\texttt{tri\_rap}(l_1), \texttt{cons}(e, \texttt{tri\_rap}(l_2)))$$

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. $\forall x \in l_1 \;\; x \leq e$ et $\forall y \in l_2 \;\; e < y$.

Déclaration du type :

$$\texttt{partage} : \texttt{'a liste} \times \texttt{'a} \rightarrow \texttt{'a liste} \times \texttt{'a liste}$$

Équation récursive :

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

On pose :

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

et

$$\begin{aligned} g(l, e, (z_1, z_2)) = \;& \texttt{si premier}(l) \leq e \\ & \quad \texttt{alors } (\texttt{cons}(\texttt{premier}(l), z_1), z_2) \\ & \quad \texttt{sinon } (z_1, \texttt{cons}(\texttt{premier}(l), z_2)) \end{aligned}$$

Cas particulier :

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

Ceci donne :

$$\begin{aligned} \texttt{partage}(l, e) &= (\texttt{nil}, \texttt{nil}) \\ & \quad\; \text{si } \texttt{liste\_vide}(l) \\ \texttt{partage}(l, e) &= \texttt{soit } (s_1, s_2) = \texttt{partage}(\texttt{reste}(l), e) \texttt{ dans} \\ & \quad \texttt{si premier}(l) \leq e \\ & \quad\quad \texttt{alors } (\texttt{cons}(\texttt{premier}(l), s_1), s_2) \\ & \quad\quad \texttt{sinon } (s_1, \texttt{cons}(\texttt{premier}(l), s_2)) \\ & \quad\; \text{si } \texttt{liste\_non\_vide}(l) \end{aligned}$$

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 par insertion

Pour trier une liste de n élélements on doit :

1. Partager une liste de $n-1$ élélements 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 :

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

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[]`

        2.2.2.2.2) `tri_rap[]`

Le nombre de comparaisons est :

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

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

Preuves de propriétés par récurrence

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

Exemple I

Propriété à démontrer : pour toute liste l, $\texttt{concat}(l, \texttt{nil}) = l$

Preuve : par induction sur l.

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

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

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

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

Exemple II

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

$$\texttt{ajout\_fin}(e, \texttt{concat}(l_1, l_2)) = \texttt{concat}(l_1, \texttt{ajout\_fin}(e, l_2))$$

Preuve : par induction sur l_1.

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

$$\begin{aligned} &\texttt{ajout\_fin}(e, \texttt{concat}(l_1, l_2)) &&= \\ &\texttt{ajout\_fin}(e, \texttt{concat}(\texttt{nil}, l_2)) &&= \\ &\texttt{ajout\_fin}(e, l_2) &&= \\ &\texttt{concat}(\texttt{nil}, \texttt{ajout\_fin}(e, l_2)) &&= \\ &\texttt{concat}(l_1, \texttt{ajout\_fin}(e, l_2)) && \end{aligned}$$

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

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

Exemple III

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

$$\texttt{inv\_seq}(\texttt{concat}(l_1, l_2)) = \texttt{concat}(\texttt{inv\_seq}(l_2), \texttt{inv\_seq}(l_1))$$

Preuve : par induction sur l_1.

– Cas de base : $l_1 = \texttt{nil}$.

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

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

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