Algorithmes sur les listes

G. Dewaele

Lycée Louis-le-Grand

Idée générale

On utilise la récursion :

```
# let rec foo = function
    | []   -> ...
    | h::t -> ... (foo t) ...

val foo : ... list -> ... = <fun>
```


Avec des variantes : cas particulier pour `h::[]`, condition sur `h`, argument supplémentaire (accumulateur), etc.

Somme des éléments d'une liste

On cherche un lien entre $\mathrm{sum}(\mathtt{h} :: \mathtt{t})$ et $\mathrm{sum}(\mathtt{t})$!

Somme des éléments d'une liste

On cherche un lien entre $\mathrm{sum}(\mathtt{h} :: \mathtt{t})$ et $\mathrm{sum}(\mathtt{t})$!

$$\mathrm{sum}(\mathtt{h} :: \mathtt{t}) = \mathtt{h} + \mathrm{somme}\big(\mathtt{q}\big)$$

Somme des éléments d'une liste

On cherche un lien entre $\mathrm{sum}(\mathtt{h} :: \mathtt{t})$ et $\mathrm{sum}(\mathtt{t})$!

$$\mathrm{sum}(\mathtt{h} :: \mathtt{t}) = \mathtt{h} + \mathrm{somme}(\mathtt{q})$$


```
# let rec somme = function
    | []   -> 0
    | t::q -> t + somme q

val somme : int list -> int = <fun>
```

Somme des éléments d'une liste

On cherche un lien entre $\mathrm{maximum}(\mathtt{t} :: \mathtt{q})$ et $\mathrm{maximum}(\mathtt{q})$

Somme des éléments d'une liste

On cherche un lien entre $\mathrm{maximum}(\mathtt{t} :: \mathtt{q})$ et $\mathrm{maximum}(\mathtt{q})$

$$\mathrm{maximum}(\mathtt{t} :: \mathtt{q}) = \max(\mathtt{t}, \mathrm{maximum}(\mathtt{q}))$$

Somme des éléments d'une liste

On cherche un lien entre $\mathrm{maximum}(\mathtt{t} :: \mathtt{q})$ et $\mathrm{maximum}(\mathtt{q})$

$$\mathrm{maximum}(\mathtt{t} :: \mathtt{q}) = \max(\mathtt{t}, \mathrm{maximum}(\mathtt{q}))$$


```
# let rec maximum = function
    | []    -> failwith "Liste vide"
    | t::[] -> t
    | t::q  -> max t (maximum q);;

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

Index du maximum

La récursion immédiate n'est pas possible

Index du maximum

La récursion immédiate n'est pas possible


```
# let rec index_et_maximum = function
    | []    -> failwith "Liste vide"
    | t::[] -> 0, t
    | t::q  -> let i, m = index_et_maximum q in
                 if m>t then (i+1, m) else (0, t)

val index_et_maximum : 'a list -> int * 'a = <fun>
```

Index du maximum

On encapsule ensuite la fonction


```
# let index_maximum lst =
    let rec index_et_maximum = function
      | []    -> failwith "Liste vide"
      | t::[] -> 0, t
      | t::q  -> let i, m = index_et_maximum q in
                   if m>t then (i+1, m) else (0, t)
    in fst (index_et_maximum lst);;

val index_maximum : 'a list -> int = <fun>
```

Plus longue séquence croissante

Pour une liste `t::q`, la plus longue séquence croissante :

- commence avec `t`
- *ou* est la plus longue séquence croissante de `q`

Plus longue séquence croissante

Pour une liste `t::q`, la plus longue séquence croissante :

- commence avec `t`
- *ou* est la plus longue séquence croissante de `q`

Dans le premier cas, sa longueur est 1 plus la séquence croissante *au début* de `q`

Plus longue séquence croissante

Cela donne donc :


```
# let plsc lst =
    let rec aux = function
      | [] -> 0, 0
      | t1::t2::q when t1<=t2
        -> let actu, maxi = aux (t2::q)
             in 1+actu, max (1+actu) maxi
      | t::q -> 1, max 1 (snd (aux q))
    in snd (aux lst);;

val plsc : 'a list -> int = <fun>
```

Sommes cumulées

Problème : construction récursive de la droite vers la gauche

Sommes cumulées

Problème : construction récursive de la droite vers la gauche

Solution : un argument renseignant sur ce qu'il y a à gauche (somme des termes)

```
# let cumsum lst =
    let rec cumsum_offset offset = function
      | [] -> []
      | t::q -> let nt = t+offset
                in nt::cumsum_offset nt q
  in cumsum_offset 0 lst;;

val cumsum : int list -> int list = <fun>
```

Sommes cumulées

On peut utiliser une application partielle ici !


```
# let cumsum =
    let rec cumsum_offset offset = function
      | [] -> []
      | t::q -> let nt = t+offset
                in nt::cumsum_offset nt q
  in cumsum_offset 0;;

val cumsum : int list -> int list = <fun>
```

Tri bulle

C'est une variante de tri par sélection :

- on détermine le plus petit élément
- on le place en tête
- on trie le reste

Tri bulle

Pour extraire le plus petit élément :

```
# let rec min_et_reste = function
    | []    -> failwith "Liste vide"
    | t::[] -> t, []
    | t::q  -> let m, r = min_et_reste q in
                 if m<t then m, t::r
                        else t, m::r;;

val min_et_reste : 'a list -> 'a * 'a list = <fun>
```

Tri bulle

Puis, pour trier :

```
# let rec tri_bulle = function
    | []  -> []
    | lst -> let m, r = min_et_reste lst
                in m::tri_bulle r;;

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

Tri par insertion

```
# let rec insertion elem = function
    | [] -> [elem]
    | t::q when elem>t -> t::insertion elem q
    | lst -> elem::lst

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



```
# let rec tri_insertion = function
    | [] -> []
    | t::q -> insertion t (tri_insertion q);;

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


Dans le sens opposé à notre fonction C !

Tri rapide

On partitionne une liste pour un pivot donné :


```
# let rec partition pivot = function
    | [] -> [], []
    | t::q -> let lst1, lst2 = partition pivot q
                in if t<pivot then t::lst1, lst2
                              else lst1, t::lst2;;

val partition : 'a -> 'a list
                        -> 'a list * 'a list = <fun>
```

Tri rapide

Puis on utilise la partition


```
# let rec tri_rapide = function
    | [] -> []
    | pivot::q -> let lst1, lst2 = partition pivot q
                  in tri_rapide lst1
                       @ pivot :: tri_rapide lst2;;

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