TD2 : Algorithmes combinatoires

Thèmes et objectifs

— algorithmes combinatoires
— raffinage, complexité
— applications des itérateurs

Dans ce genre d'exercice combinatoire, il est bon de s'aider d'une abstraction du problème, en général l'abstraction qui consiste simplement à compter le nombre de solutions du problème, sans les engendrer. La formule combinatoire arithmétique servira de guide à la réalisation de l'algorithme recherché.

On est rapidement perdu dans les listes de listes de listes, etc. Donc, utiliser un code couleur différent pour les vraies listes d'un côté (les éléments calculés : permutations, parties, etc) et les ensembles (de listes) ou les ensembles d'ensembles (de listes) de l'autre peut aider.

1 Parties d'un ensemble

1.1 Contrat

Le contrat donne quelque chose comme : ▷ **Support étudiant**

```
(* ***************************************************************
 parties : 'a list -> 'a list list
 argument: l, une liste quelconque d'éléments
 résultat: une liste de listes, dont les 2^(List.length l) éléments
          sont les sous-listes de l. Si les élements de l sont
          différents, les listes éléments du résultat seront
          différentes aussi, et formeront les parties de l.
   *************************************************************** *)
```

1.2 Raffinage fonctionnel

▷ **Exercice 1** ▷ ***Support étudiant***

— *Donner une formulation récursive comptant le nombre de parties d'un ensemble de cardinal n.*

▷ ***Solution***

$$\underbrace{2^0}_{\textit{parties de } 0 \textit{ éléments}} \triangleq 1$$
$$\underbrace{2^{n+1}}_{\textit{parties de } n+1 \textit{ éléments}} \triangleq 2 \times \underbrace{2^n}_{\textit{parties de } n \textit{ éléments}}$$

Le terme $2 \times \ldots$ a ici une interprétation simple : après avoir mis de côté un élément e quelconque parmi les $n+1$, il faut, pour **chaque** *partie σ des n éléments restants (avec `List.map` donc), engendrer 2 parties des n éléments. Ces 2 parties sont σ à laquelle on ajoute ou non e. Il faut ensuite les regrouper ensemble dans une seule et même liste avec `List.flatten` ou `@`. Au final on aura bien créé $2 \times 2^n = 2^{n+1}$ parties. Comme on part d'une liste pour représenter l'ensemble de départ, il y a tout intérêt à faire que e soit le premier élément* *de cette liste, puisqu'ainsi on obtient une décomposition récursive structurelle du problème (et on conserve l'ordre des éléments). Reste le cas terminal :* $2^0 = 1$*, ce qui signifie* `parties [] = [ [] ]`.

▷ **Exercice 2**

- *Écrire la fonction* `ajout`*, qui à partir d'un élément* e *et d'ensembles* $\{E_1, \ldots, E_n\}$ *renvoie l'ensemble* $\{E_1, \{e\} \cup E_1, \ldots, E_n, \{e\} \cup E_n\}$.
- *Écrire la fonction* `parties`*, qui renvoie l'ensemble des parties d'un ensemble.*

▷ ***Support étudiant***

▷ ***Solution***

Attention *: dans les exos combinatoires, il ne faut surtout pas tester l'ordre dans lequel les résultats (ici les parties) sont produits. Le type* `list` *externe du résultat est utilisé comme un ensemble, pas comme une liste. L'ordre produit est dans l'implantation, pas dans la spécification. Par contre, on a spécifié qu'on renvoyait des sous-listes qui conservent l'ordre initial entre les éléments.*

```
(* ajout : 'a -> 'a list list -> 'a list list *)
let ajout t parties_q =
 List.flatten (List.map (fun partie_q -> [ partie_q; t::partie_q ]) parties_q)

(* OU BIEN, pas dans le même ordre *)
let ajout t parties_q =
                                          parties_q
 @ List.map (fun partie_q -> t::partie_q) parties_q

(* parties : 'a list -> 'a list list *)
let rec parties l =
 match l with
 | []   -> [ [] ]
 | t::q -> ajout t (parties q)

(* OU BIEN on reconnaît un List.fold_right *)
let parties l = List.fold_right ajout l [ [] ]

(* TEST
   parties [] = [ [] ]
   let parties_1 = parties [1]
   List.length parties_1 = puissance 2 1
   List.mem []  parties_1
   List.mem [1] parties_1
   let parties_123 = parties [1; 2; 3]
   List.length parties_123 = puissance 2 3
   List.mem []        parties_123
   List.mem [1]       parties_123
   List.mem [2]       parties_123
   List.mem [3]       parties_123
   List.mem [1; 2]    parties_123
   List.mem [1; 3]    parties_123
   List.mem [2; 3]    parties_123
   List.mem [1; 2; 3] parties_123
*)
```

2 Permutations d'une liste

2.1 Contrat

Le contrat donne quelque chose comme ▷: **Support étudiant**

```
(* ***************************************************************
 permutations : 'a list -> 'a list list
 argument: l, une liste quelconque d'éléments
 résultat: une liste de listes, dont les (List.length l)! éléments
           ont même longueur que l. Si les élements de l sont
           différents, les listes éléments du résultat seront
           différentes aussi, et formeront les permutations de l.
   *************************************************************** *)
```

2.2 Raffinage fonctionnel

▷ **Exercice 3**

— *Donner une formulation récursive comptant le nombre de permutations d'un ensemble de taille n.*

▷ ***Solution***

Une équation récursive qui détermine les (ou plutôt le nombre de) solutions est (...suspense...) la factorielle !! En effet, il y a $n!$ permutations d'un ensemble de taille n.

$$\begin{aligned}\underbrace{0!}_{\textit{permutations de } 0 \textit{ éléments}} &\triangleq 1\\ \underbrace{(n+1)!}_{\textit{permutations de } n+1 \textit{ éléments}} &\triangleq (n+1) \times \underbrace{n!}_{\textit{permutations de } n \textit{ éléments}}\end{aligned}$$

Le terme $(n+1) \times \ldots$ a ici une interprétation simple : après avoir mis de côté un élément e quelconque parmi les $n+1$, il faut, pour **chaque** *permutation σ des n éléments restants (avec `List.map` donc), engendrer $n+1$ permutations des $n+1$ éléments. Ces $n+1$ permutations doivent bien sûr contenir e et être toutes différentes. Il faut ensuite les regrouper ensemble dans une seule et même liste avec `List.flatten` ou `@`. Au final on aura bien créé $(n+1) \times n! = (n+1)!$ permutations. Une solution est d'insérer e à toutes les positions possibles dans la liste σ. Il y a bien $n+1$ telles positions. Comme on part d'une liste pour représenter l'ensemble de départ, il y a tout intérêt à faire que e soit le premier élément de cette liste, puisqu'ainsi on obtient une décomposition récursive structurelle du problème. Reste le cas terminal : $0! = 1$, ce qui signifie `permutations [] = [ [] ]`.*

▷ **Exercice 4**

— *Écrire la fonction `insertions`, qui insère un élément à toutes les positions d'une liste.*

— *Écrire la fonction `permutations`, qui renvoie l'ensemble des permutations d'un ensemble.*

▷ ***Solution***

```
(* insertions : 'a -> 'a list -> 'a list list *)
let rec insertions e l =
 match l with
 | []   -> [[e]]                    (* insertion de e en fin             *)
 | t::q -> (e::l)                   (* insertion de e avant t            *)
```

```
          ::                          (* OU                                *)
          List.map (fun l -> t::l)    (* on ajoute t en tête des ...       *)
                   (insertions e q)   (* insertions de e après t, i.e. dans q *)
(* TEST
   insertions 1 [] = [ [1] ]
   let insertions_1_23 = insertions 1 [2; 3]
   List.length insertions_1_23  = 3
   List.mem [1; 2; 3] insertions_1_23
   List.mem [2; 1; 3] insertions_1_23
   List.mem [2; 3; 1] insertions_1_23
*)

(* permutations : 'a list -> 'a list list *)
let rec permutations l =
 match l with
 | []   -> [[]]
 | t::q -> List.flatten (List.map (insertions t) (permutations q))

(* OU *)

let permutations l = List.fold_right (fun t permutations_q -> List.flatten (List.map (insertions t) p

(* TEST
   permutations [] = [ [] ]
   permutations [1] = [ [1] ]
   let permutations_123 = permutations [1; 2; 3]
   List.length permutations_123 = fact 3
   List.mem [1; 2; 3] permutations_123
   List.mem [1; 3; 2] permutations_123
   List.mem [2; 1; 3] permutations_123
   List.mem [2; 3; 1] permutations_123
   List.mem [3; 1; 2] permutations_123
   List.mem [3; 2; 1] permutations_123
```

3 Combinaisons

▷ **Support étudiant**

3.1 Contrat

Le contrat donne quelque chose comme :

```
(* ****************************************************************
 combinaisons : 'a list -> int -> 'a list list
 argument: l, une liste quelconque d'éléments supposés différents
           k, le nombre d'élements distincts à tirer
 résultat: une liste de combinaisons. Chaque combinaison est elle-même
           une liste d'éléments, dont les éléments
           sont ceux de l.
   **************************************************************** *)
```

3.2 Raffinage fonctionnel

▷ **Exercice 5** *Donner une formulation récursive comptant le nombre de combinaisons de k éléments d'un ensemble à n éléments.*

▷ ***Solution***

$$\begin{aligned}
&C(n,0) = 1\\
&C(0,k+1) = 0\\
&C(n+1,k+1) = C(n,k+1) + C(n,k)
\end{aligned}$$

$C(n,0) = 1$ *puisqu'on tire l'ensemble vide, puis* $C(0,k+1) = 0$ *(aucun élément à tirer pour construire une combinaison non vide). Avec une combinaison de* $C(n,k)$ *et un élément supplémentaire* e*, on peut obtenir d'une part une combinaison de* $C(n+1,k)$ *en ne tirant pas* e*, et on peut obtenir d'autre part une combinaison de* $C(n+1,k+1)$ *en tirant* e*.*

▷ **Exercice 6**

— *Écrire la fonction* `combinaisons` *(contrat+code+tests)*

▷ ***Solution***

```
(* combinaisons : 'a list -> int -> 'a list list *)
let rec combinaisons l k =
 match l, k with
 | _   , 0 -> [ [] ]
 | []  , _ -> []
 | t::q, _ ->   List.map (fun combi -> t::combi) (combinaisons q (k-1))
              @ (combinaisons q k)
(* TEST
   combinaisons [1;2] 0 = []
   combinaisons [] 1 = [ ]
   List.mem [1;3] (combinaisons [1;2;3] 2)
   List.mem [1;2] (combinaisons [1;2;3] 2)
   List.mem [2;3] (combinaisons [1;2;3] 2)
*)
```