Cours 3 : Listes, itérateurs de listes et tris

1 Un type récursif déjà vu : les entiers

Les entiers naturels (donc positifs) correspondent à plusieurs schémas récursifs et peuvent être vus comme un type récursif (de différentes façons).

Un **entier** est de la forme :
- soit `0` *(* cas de base/terminal *)*
- soit `n+1` où `n` est un **entier**. *(* cas général/récursif *)*

Ou bien, un **entier** est de la forme :
- soit `0` *(* cas de base/terminal *)*
- soit `2*(n+1)` où `n` est un **entier**. *(* cas général/récursif *)*
- soit `2*n+1` où `n` est un **entier**. *(* cas général/récursif *)*

Ce sont des définitions d'un type récursif. D'une définition d'un type récursif découle naturellement des fonctions récursives (même récursion que le type), par exemple la factorielle pour le premier schéma, la puissance indienne pour le second.

2 Structure de données : liste

Définition Une α-**liste** est :
- soit la liste vide *(* cas de base/terminal *)*
- soit `a::l` où `a` est un α et `l` une α-**liste**. *(* cas général/récursif *)*

Remarques
- la structure de données "liste" est **homogène**, i.e. tous les éléments d'une liste ont le même type α.
- une liste non vide se présente toujours sous la forme `tete::queue`, les différents éléments ne sont accessibles que de cette façon. Pas d'accès direct indexé comme pour les tableaux.
- structure de données **dynamique** : on peut "ajouter" ou "retirer" des éléments (ce n'est qu'un abus de langage, puisqu'il n'y a pas d'effets de bord).

```
# [];;                (* la liste vide *)
- : 'a list = []
# 1::2::3::[];;
- : int list = [1;2;3]
# [[1;2];[];[3]];;
- : int list list = [[1;2];[];[3]]
```

On remarque l'équivalence des écritures a::b::c::[] et [a; b; c].

Accès à la tête et à la queue d'une liste On accède à la tête (resp. queue) d'une liste à l'aide de la fonctions List.hd (head) (resp. List.tl (tail)) ou en utilisant le filtrage :

```
let rec somme_liste liste =
  match liste with
  | []          -> 0                              (* cas de base/terminal *)
  | tete::queue -> tete + (somme_liste queue) (* cas general/recursif *)
```

Cette fonction suit la récursivité naturelle (structurelle) des listes.

3 Itérateurs de listes

3.1 L'itérateur List.map

```
List.map f [t$_1$;t$_2$; $\ldots$ ;t$_n$] = [f t$_1$;f t$_2$; $\ldots$ ;f t$_n$]
```

▷ **Exercice 1**

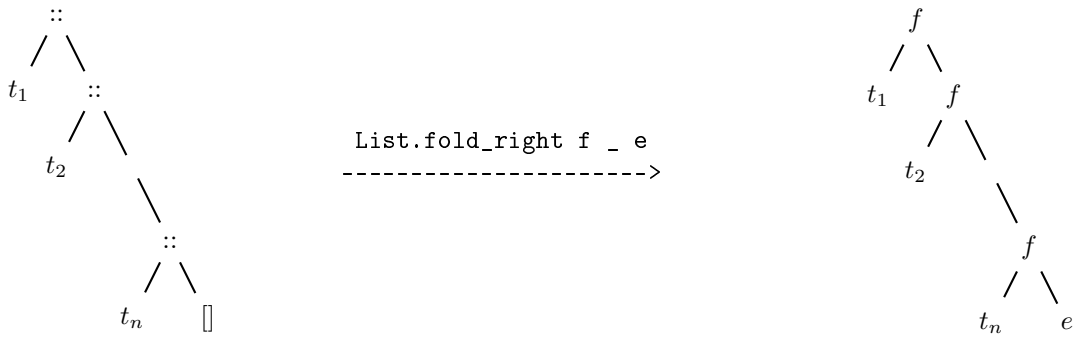
1. *Donner le type de l'itérateur List.map.*
2. *Écrire cet itérateur.*
3. *Écrire string_of_int_list, qui transforme une liste d'entiers en une liste de chaînes de caractères, en utilisant List.map.*

3.2 Les deux itérateurs ––fold––

3.2.1 List.fold_right

```
List.fold_right f [t$_1$;t$_2$; $\ldots$ ;t$_n$] e = (f t$_1$ (f t$_2$ ($\ldots$ (f t$_n$
```

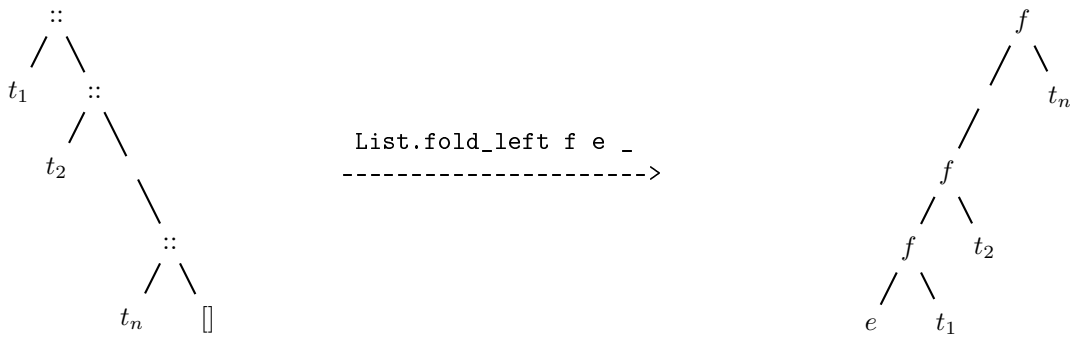
Son effet peut être spécifié graphiquement par cette transformation structurelle d'arbres (de structure de données en structure de contrôle/appels de fonctions) :



3.2.2 List.fold_left

```
List.fold_left f e [t$_1$;t$_2$; $\ldots$ ;t$_n$] = (f ($\ldots$(f (f e t$_1$) t$_2$)$\l
```

Son effet peut être spécifié graphiquement par cette transformation non-structurelle d'arbres :



La fonction List.fold_right est l'itérateur fondamental avec lequel on peut écrire n'importe quelle fonction structurelle sur les listes, i.e. définie par filtrage direct. Même List.fold_left peut s'écrire avec List.fold_right. Ces itérateurs parcourent une seule fois chaque élément de la liste : complexité linéaire $\Theta(length(liste))$ (sans compter l'effet de la fonction appliquée).

▷ **Exercice 2**
- *Donner le type des itérateurs List.fold_right et List.fold_left.*
- *Écrire ces itérateurs.*
- *Écrire la fonction rev (qui renverse une liste) à l'aide des deux itérateurs. Quelle version a la complexité la plus faible ?*

3.3 Autres itérateurs

Il existe d'autres itérateurs sur les listes. Ils sont décrits dans la documentation en ligne `https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`.

Les avantages des itérateurs structurels :
- garantit la terminaison
- simplifie les calculs de complexité
- permet la simplification, l'optimisation de code

4 Tris

4.1 Complexité des tris par comparaison

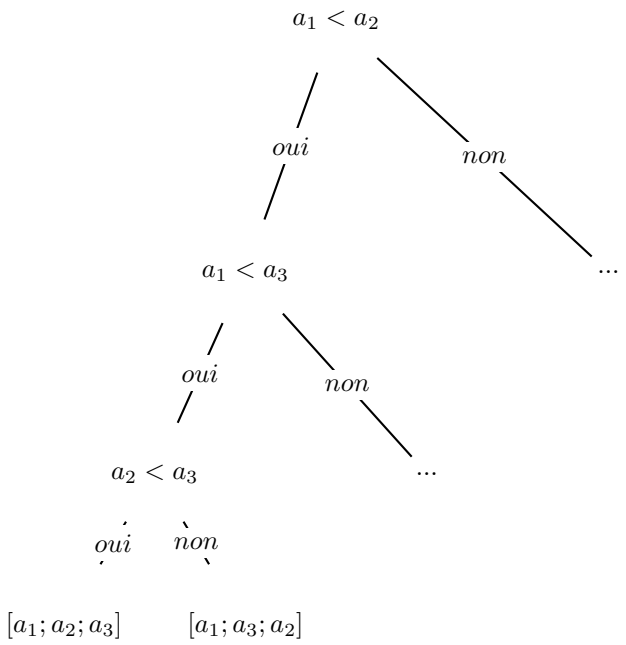
Question : Quelle est la meilleure complexité (en terme de comparaison d'éléments deux à deux) possible pour un algorithme de tri par comparaison ?

Réponse :

Un algorithme de tri prend :
- en entrée : $a_1, \ldots, a_n$ (n éléments)
- en sortie : $a_{\sigma(1)} < \ldots < a_{\sigma(n)}$, σ : permutation des n éléments

La complexité est calculée en fonction du nombre de comparaisons, c'est-à-dire indépendamment du langage (récursif, impératif, etc).

Un algorithme de tri peut être résumé par l'arbre suivant (cas particulier pour $n = 3$) :



L'ordre dans lequel les valeurs à trier sont entrées dans la liste de départ n'est pas important. *A priori*, un algorithme de tri peut réaliser en premier n'importe quelle comparaison $a_i < a_j$, puis suivant le résultat, tenter une (autre) comparaison et ainsi de suite jusqu'à ce qu'un ordre total sur les a_i soit reconstitué. Tout algorithme de tri par comparaison correspond donc à un tel arbre binaire (de comparaisons) avec des choix différents de comparaisons à chaque nœud de l'arbre pour chaque algorithme. Toute exécution d'un algorithme est un parcours de la racine vers une feuille de son propre arbre de comparaisons.

Nous nous intéressons à la complexité du pire cas $c(n)$ (en nombre de comparaisons). Nous allons chercher à déterminer $s(n)$ tel que $s(n) \leq c(n)$, c'est à dire que quel que soit l'algorithme de tri par comparaison, la branche la plus longue est au moins de taille $s(n)$.

Cas particulier : Nous montrons que $s(3) = 3$.
Le nombre de permutations de 3 éléments est $3! = 6$. L'arbre de comparaison a donc 6 feuilles. Or, un arbre binaire à 6 feuilles, est au moins de hauteur 3, i.e., il y a toujours un chemin de longueur 3 ou plus.

Généralisation : Nous montrons que $s(n) = \Theta(n \log n)$, i.e., un tri par comparaison est au moins de complexité $n \log n$.
Le nombre de permutations possibles est $n!$. Si l'arbre (binaire) est de hauteur maximum h, il a au maximum 2^h feuilles, donc pour être sûr de pouvoir distinguer les $n!$ cas, il faut :

$$2^h \geq n!$$

i.e. :

$$h \geq \log_2(n!)$$

mais $n! > (n/e)^n$ (Stirling), donc

$$h \geq n \log_2 n - n \log_2 e.$$

donc $c(n) \geq n \log n$.

4.2 Des algorithmes de tri par comparaison

4.2.1 Une implémentation immédiate

Puisqu'il s'agit de trouver la bonne permutation, il suffit de les calculer toutes et de ne garder que celle qui est ordonnée (ou la première qui est ordonnée).

Complexité : $\Theta((n-1) * n!)$!
- $n!$ permutations
- $n-1$ comparaison d'éléments par permutation

4.2.2 Tri par insertion

Analyse récursive :
- Si je sais trier une liste de taille $n-1$ comment puis-je trier une liste de taille n ?
- Réponse : j'insère l'élément souhaité à "sa place".
- $\Rightarrow$ Besoin d'une fonction auxiliaire qui insère un élément dans une liste triée.

```
(* insertion : 'a -> 'a list -> 'a list *)
(* insere un element dans une liste triee par ordre croissant *)
let rec insertion x liste =
 match liste with
 | []          -> [x]
 | tete::queue -> if x <= tete then x::liste
                                else tete::(insertion x queue)
```

L'algorithme est alors une récursivité qui dans le cas général insère la tête de la liste dans la queue triée (appel récursif).

```
(* tri_insertion : 'a list -> 'a list *)
(* trie une liste par ordre croissant *)
let rec tri_insertion liste =
 match liste with
 | []          -> []
 | tete::queue -> insertion tete (tri_insertion queue)

let tri_insertion l = List.fold_right insertion l []
```

Complexité (en nombre de comparaisons) : On étudie la complexité du pire cas, correspondant à l'insertion en fin de liste.

$$\begin{aligned} C_{max}(0) &= 0 \\ C_{max}(n+1) &= n + C_{max}(n) \end{aligned}$$

La fonction génératrice des $C(n)$ permet de conclure, mais on peut reconnaître directement dans $C(n)$ une expression de la somme des $n-1$ premiers entiers, soit :

$$C_{max}(n) = \frac{n(n-1)}{2}$$

D'où : $C_{max}(n) = \Theta(n^2)$. De manière similaire, $C_{moy}(n) = \Theta(n^2)$, $C_{min}(n) = \Theta(n)$.

4.2.3 Tri par fusion

Analyse récursive :

— Si je sais trier une liste de taille $n/2$ comment je trie une liste de taille n ?
— Réponse : Je fusionne deux listes triées.
⇒ Besoin de deux fonctions auxiliaires
  — une qui découpe une fonction en deux en deux sous-listes de même taille ± 1.
  — une qui fusionne deux listes triées

On coupe une liste en deux listes de même taille ± 1.

```
(* decompose : 'a list -> 'a list * 'a list *)
(* decompose une liste en deux listes de tailles egales a plus ou moins un element *)
let rec decompose liste =
 match liste with
 | []              -> [], []
 | [_]             -> liste, []
 | e1::e2::queue -> let (l1,l2)= (decompose queue) in (e1::l1, e2::l2)
```

On recompose deux listes triées.

```
(* recompose : 'a list -> 'a list  -> 'a list *)
(* fusionne deux listes triees par ordre croissant pour en faire une seule triee par ord
let rec recompose liste1 liste2 =
 match liste1, liste2 with
 | []           , _              -> liste2
 | _            , []             -> liste1
 | tete1::queue1, tete2::queue2 -> if tete1 < tete2
                                   then tete1 :: recompose queue1 liste2
                                   else tete2 :: recompose liste1 queue2
```

L'algorithme de tri consiste alors à :

— couper la liste en deux
— trier les deux sous-listes (appels récursifs)
— fusionner les deux sous-listes triées

```
(* tri_fusion : 'a list -> 'a list *)
(* trie une liste par ordre croissant *)
let rec tri_fusion liste =
 match liste with
 | []
 | [_]  -> liste
 | _    -> let (l1,l2) = decompose liste in recompose (tri_fusion l1) (tri_fusion l2)
```

Remarque : Stratégie de diviser pour régner (divide and conquer). Mais il faut être certain de diviser sinon la fonction boucle et on ne règne pas ! Pour être certain que les appels récursifs se fassent sur des listes strictement plus petites, il faut avoir liste vide et singleton comme cas terminaux.

Complexité du tri fusion Le tri fusion est un algorithme de complexité uniforme, quelles que soient les données. Ainsi, $C_{min}(n) = C_{moy}(n) = C_{max}(n) = C(n)$, avec :

$$\begin{aligned} C(0) &= 0 \\ C(1) &= 0 \\ C(2n+2) &= 2C(n+1) + 2n + 1 \\ C(2n+3) &= C(n+2) + C(n+1) + 2n + 2 \end{aligned}$$

Ici, on ne va même pas essayer d'intégrer $C(n)$ car ça paraît trop compliqué *a priori*. Par contre, on peut l'encadrer par une fonction intégrable. L'idée est d'exploiter la dichotomie réalisée par le tri fusion. On va donc considérer des listes qui se comportent simplement et uniformément lorsqu'on les coupe en 2 : les listes de taille 2^n. Sachant qu'on a l'encadrement suivant, toujours sous l'hypothèse que $C(n)$ est monotone croissante :

$$C(2^{\lceil \log_2 n \rceil - 1}) < C(n) \leq C(2^{\lceil \log_2 n \rceil})$$

L'étude de la fonction génératrice des $C(2^n)$ permet de conclure, mais trop compliqué. Avec quelques astuces, on s'en sort directement. Tout d'abord :

$$\begin{array}{ll} C(2^0) & = C(1) = 0 \\ C(2^{n+1}) & = 2^{n+1} - 1 + 2C(2^n) \end{array}$$

Puis, en posant $D(n) = \frac{C(2^n)}{2^n}$:

$$\begin{array}{ll} D(0) & = \frac{C(1)}{1} = 0 \\ D(n+1) & = 1 - \frac{1}{2^{n+1}} + \frac{2C(2^n)}{2^{n+1}} = 1 - \frac{1}{2^{n+1}} + D(n) \end{array}$$

Par intégration de $D(n)$:

$$\begin{array}{ll} D(n) & = \underbrace{(1 - \frac{1}{2^n}) + (1 - \frac{1}{2^{n-1}}) + \ldots + (1 - \frac{1}{2^1})}_{n} \\ & = n - \sum_{k=1}^{n} \frac{1}{2^k} \\ & = n - 1 + \frac{1}{2^n} \\ \\ C(2^n) & = (n-1)2^n + 1 \end{array}$$

L'encadrement initial devient donc :

$$1 + (\lceil \log_2 n \rceil \ - 2) * 2^{\lceil \log_2 n \rceil \ - 1} < C(n) \leq 1 + (\lceil \log_2 n \rceil \ - 1) * 2^{\lceil \log_2 n \rceil}$$

Donc, puisque $\lceil \log_2 n \rceil = \Theta(\log_2 n)$ et $2^{\lceil \log_2 n \rceil} = \Theta(2^{\log_2 n}) = \Theta(n)$, on obtient finalement $C(n) = \Theta(n * \log n)$

4.3 Un algorithme de tri de complexité linéaire (facultatif)

À vrai dire, un tri sans comparaisons n'est pas réellement un tri, mais plutôt un algorithme de rangement. On dispose d'un nombre fixe de boîtes dans lesquelles on va déposer chaque élément correspondant à cette boîte. L'ordre entre les boîtes correspond à l'ordre attendu entre les éléments.

Un cas typique est celui où on veut trier un ensemble de N nombres appartenant tous à l'intervalle entier $[0, P]$. Ici, on rangera un nombre de valeur $i \in [0, P]$ dans la boîte i.

Comment implanter ? Réponse : L'implantation se fait très bien avec des tableaux, en incrémentant la $i^{\text{ème}}$ case lorsqu'on rencontre la valeur i. Ainsi, le rangement s'effectue clairement en temps linéaire, et la construction de la liste triée résultat aussi.

En OCaml, pas de tableau mais un tableau est une fonction des indices dans les valeurs, alors utilisons des fonctions ! ! Par contre, on aura à nouveau des comparaisons, mais les éléments à trier sont de nouveau arbitraires...

```
let tri_lineaire l =
 let rec tri l =
  match l with
```

```
  | []    -> (fun i -> 0)
  | t::q -> let occ_q = tri q
            in (fun i -> if i=t then occ_q i + 1 else occ_q i)
in tri l
```

Le temps de construction de la fonction résultat seule est bien linéaire en n, mais il reste à parcourir les indices i pour récupérer les occurrences stockées dans la fonction résultat (ce serait pareil pour le tableau). Les n images de f sont calculées chacune en temps $\Theta(n)$ par rapport au simple accès en temps constant d'une case d'un tableau. Ce qui fait au total une complexité en $\Theta(n^2)$!

Remarque : On pourrait améliorer cette solution pour retrouver $\Theta(n \log n)$...

Finalement, le tri sans comparaison (en impératif) est linéaire car :
- les éléments à trier sont des indices de tableau (pas de conversion vers des entiers consécutifs, pas de comparaison coûteuse).
- l'accès aux cases d'un tableau est en temps constant (parce que les indices, i.e. le nombre de boîtes, sont bornés).

5 Exercices

5.1 Listes

▷ **Exercice 3**

1. *Écrire les fonctions hd et tl.*
2. *Écrire la fonction deuxieme qui renvoie le deuxième élément d'une liste.*
3. *Écrire la fonction taille qui renvoie la longueur d'une liste.*
4. *Écrire append (concaténation), i.e.*
   append [1;2] [3;4;5] = [1;2;3;4;5].
5. *Écrire map, qui applique une fonction donnée à tous les éléments d'une liste, i.e.*
   map f [a; b; c] = [f a; f b; f c].
6. *Donner la version terminale de taille .*
7. *Écrire une fonction rev de renversement d'une liste.*
8. *Écrire les fonctions n_a_zero et zero_a_n, telles que :*

   n_a_zero n = [n; n-1;$|ldots$; 1 ; 0]
   zero_a_n n = [0; 1 ;$|ldots$; n-1; n]

   Attention : utilisation de append interdite.