

module 3 :

Les listes indexées (vecteurs)

Un *type de données abstrait* est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble. Ces opérations comprennent les moyens d'accéder aux éléments de l'ensemble, et aussi, lorsque l'objet est dynamique, les possibilités de le modifier.

Une *structure de données* est l'implémentation explicite d'un type de données abstrait, avec la réalisation des opérations d'accès, de construction, et de modification afférentes.

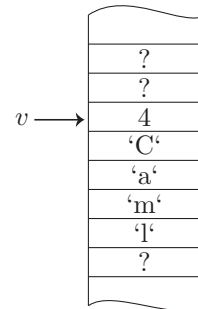
Dans ce module, nous nous intéressons au type abstrait *liste indexée*, implémenté en Caml par le type `vect`.

1. Caractéristiques d'une liste indexée

Une *liste* est une suite finie $(a_1, a_2, \dots, a_{n-1})$ d'objets de même type. Comme son nom l'indique, une liste *indexée* présente pour principale caractéristique de posséder une *indexation*, c'est-à-dire la capacité d'accéder *en temps constant* à chacun de ses éléments. En revanche, la réalisation de cet index impose à une liste indexée une *taille fixée* lors de sa création¹.

Pour comprendre cette limitation, il faut revenir à la notion de case mémoire. Lorsque l'on crée un vecteur, il faut spécifier sa taille et son contenu initial. Par exemple, l'instruction suivante crée un vecteur de 4 cases :

```
# let v = [| 'C'; 'a'; 'm'; 'l' |] ;;
v : char vect = [| 'C'; 'a'; 'm'; 'l' |]
```



Dans la mémoire, un bloc de 5 cases consécutives a été associé à ce vecteur : la première case contient la taille du vecteur, les suivantes son contenu.

Les cases mémoires étant *consécutives* et de même taille, un simple calcul algébrique permet de connaître l'adresse de la case d'indice k ; c'est ce qui explique pourquoi l'accès à un élément se fait avec un coût constant.

En revanche, les cases mémoires qui précèdent et qui suivent celles occupées par le vecteur peuvent être utilisées à d'autres usages ; c'est pourquoi il n'est pas possible de modifier la taille d'un vecteur.

2. Représentation d'un polynôme par un vecteur

On peut représenter le polynôme $p(x) = \sum_{k=0}^n a_k x^k$ de $\mathbb{Z}[X]$ par le vecteur $[|a_0; a_1; \dots; a_n|]$. Pour illustrer la manipulation des vecteurs, nous allons nous intéresser au problème de l'évaluation de ce polynôme en un point b .

• Évaluation par puissances croissantes

Pour calculer $p(b)$, on peut itérer la suite (u_0, \dots, u_n) définie par : $\forall j \in \llbracket 0, n \rrbracket, u_j = \sum_{k=0}^j a_k b^k$. Celle-ci se calcule à l'aide des relations :

$$u_0 = a_0 \quad \text{et} \quad \forall j \in \llbracket 1, n \rrbracket, u_j = u_{j-1} + a_j b^j.$$

¹Nous étudierons plus tard les listes *dynamiques*, implémentées en Caml par le type `list`, qui possèdent les caractéristiques opposées : une taille qui peut varier, mais en contrepartie un temps d'accès aux éléments non constant.

Cette itération nécessite le calcul des valeurs de b^j pour $j \in \llbracket 1, n \rrbracket$, calcul qui peut se faire en itérant la suite (x_0, \dots, x_n) définie par : $\forall j \in \llbracket 0, n \rrbracket, x_j = b^j$. Celle-ci se calcule à l'aide des relations :

$$x_0 = 1 \quad \text{et} \quad \forall j \in \llbracket 1, n \rrbracket, x_j = bx_{j-1}.$$

En d'autres termes il s'agit d'itérer les suites $(u_j)_{0 \leq j \leq n}$ et $(x_j)_{0 \leq j \leq n}$ à l'aide des relations :

$$\begin{cases} x_0 = 1 \\ u_0 = a_0 \end{cases} \quad \text{et} \quad \forall j \in \llbracket 1, n \rrbracket, \begin{cases} x_j = bx_{j-1} \\ u_j = u_{j-1} + a_j x_j \end{cases}$$

En voici la version impérative :

```
let évalue p b =
  let n = vect_length p - 1 in
  let x = ref 1 and u = ref p.(0) in
  for j = 1 to n do x := b * !x ;
                    u := !u + p.(j) * !x
                    done ;
  !u ;;
```

Essayons avec $p(x) = 3x^2 + 2x + 1$ et $b = -2$:

```
# let p = [|1;2;3|] and b = -2 in évalue p b ;;
- : int = 9
```

Évaluer le coût de cet algorithme, c'est dénombrer le nombre d'opérations algébriques : ici $2n$ multiplications et n additions. Peut-on faire mieux ?

• Algorithme de Hörner

Cet algorithme utilise la relation de récurrence : $p(b) = a_0 + b \left(\sum_{k=1}^n a_k b^{k-1} \right) = a_0 + b \left(\sum_{k=0}^{n-1} a_{k+1} b^k \right)$ et possède donc une version récursive naturelle :

```
let hörner p b =
  let n = vect_length p - 1 in
  let rec aux = fonction
    k when k = n -> a.(n)
    | k           -> a.(k) + b * (aux (k+1))
  in aux 0 ;;
```

Cet algorithme effectue n multiplications et n additions ; il est donc meilleur que le précédent.

En fait, on peut montrer que l'évaluation de polynômes ne peut se faire avec moins d'opérations ; en ce sens, il s'agit d'un algorithme optimal.

3. Algorithmes de recherche

De nombreux problèmes d'informatique se ramènent à la recherche d'un élément dans une liste ; nous allons nous y intéresser dans le cadre des listes indexées.

La recherche séquentielle ordinaire consiste à parcourir le vecteur jusqu'à éventuellement trouver un élément satisfaisant à nos exigences. Pour fixer les idées, considérons une fonction booléenne `convient` de type : `'a -> bool` et un vecteur `v` de type `'a vect`. Il s'agit de déterminer un entier `i`, s'il en existe, tel que `convient v.(i) = true`.

Lorsque l'existence de `i` est assurée, on procède comme ci-dessous :

– version impérative :

```
let recherché1 v =
  let i = ref 0 in
  while not convient v.(!i) do i := !i + 1 done ;
  !i ;;
```

– version fonctionnelle :

```
let recherché1 v =
  let rec aux = fonction
    i when convient v.(i) -> i
    | i                     -> aux (i+1)
  in aux 0 ;;
```

Lorsque l'existence de i n'est pas assurée, l'algorithme précédent peut conduire à une erreur de type :

Invalid_argument "vect_item"

signifiant qu'on a cherché à accéder dans un vecteur au contenu d'une case n'existant pas. On modifie l'algorithme ainsi :

– version impérative :

```
exception Non_trouvé ;;

let recherche2 v =
  let n = vect_length v in
  let i = ref 0 in
  while i < n & not convient v.(!i)
  do i := !i + 1 done ;
  if i < n then !i else raise Non_trouvé ;;
```

– version fonctionnelle :

```
exception Non_trouvé ;;

let recherche2 v =
  let n = vect_length v in
  let rec aux = fonction
    i when i = n      -> raise Non_trouvé
  | i when convient v.(i) -> i
  | i                  -> aux (i+1)
  in aux 0 ;;
```

Notons que l'algorithme impératif utilise l'évaluation paresseuse du «et» logique : lorsque expr1 s'évalue en *false*, le résultat est renvoyé par $\text{expr1} \ \& \ \text{expr2}$ sans que expr2 soit évalué.

Ces algorithmes de recherche ont à l'évidence un coût linéaire.

• Recherche dichotomique

Considérons maintenant un ensemble E muni d'une relation d'ordre total \preccurlyeq , et un vecteur $v = [v_0, \dots, v_{n-1}]$ d'éléments de E triés par ordre croissant. On désire calculer le rang d'un élément x de E , s'il se trouve dans cette liste.

La méthode de recherche *dichotomique* est la suivante :

on compare x et v_k , avec $k = \lfloor \frac{n-1}{2} \rfloor$; si $x = a_k$, c'est terminé ; si $x \prec a_k$, on recherche x dans le vecteur $[v_0, \dots, v_{k-1}]$; si $v_k \prec x$, on cherche x dans le vecteur $[v_{k+1}, \dots, v_{n-1}]$.

Cet algorithme s'écrit très naturellement de manière récursive :

```
let recherche_dicho v x =
  let rec cherche i j =
    let k = (i+j)/2 in match () with
    | _ when j < i      -> raise Not_found
    | _ when x = v.(k) -> k
    | _ when x < v.(k) -> cherche i (k-1)
    | _                 -> cherche (k+1) j
  in cherche 0 (vect_length v - 1) ;;
```

Notons t_n le nombre de comparaisons effectuées pour trouver un élément dans une liste triée de longueur n . Cette fonction étant croissante, on dispose des relations :

$$t_0 = 0 \quad \text{et} \quad \forall n \geq 1, t_n \leq 1 + t_{\lfloor \frac{n}{2} \rfloor}$$

qui permettent de montrer par récurrence que : $t_n \leq 1 + \log_2 n$.

Il s'agit donc d'un coût logarithmique, bien meilleur qu'une recherche linéaire.

Remarque. La notion d'arbre de décision permet de montrer que tout algorithme de recherche dans une liste de longueur n effectuée dans le pire des cas au moins $1 + \lceil \log_2 n \rceil$ comparaisons.

4. Exercices

1 – On reprend la modélisation des polynômes de $\mathbb{Z}[X]$: $P = \sum_{k=0}^n a_k X^k$ est représenté par le vecteur (a_0, \dots, a_n) .

Rédiger une fonction calculant le produit de deux polynômes. Évaluer le coût de cette fonction lorsque les deux polynômes ont même degré n .

2 – En vous inspirant du schéma de Hörner, définir la fonction qui à un polynôme P et un entier a associe le polynôme $P(X + a)$. Évaluer le coût de votre algorithme.

- 3 – Utiliser le principe de la recherche dichotomique pour écrire une fonction Caml de type `int -> int` qui pour un entier $n \in \mathbb{N}$ calcule $\lceil \sqrt{n} \rceil$.
- 4 – Imaginez que l'on modifie l'algorithme de recherche dichotomique de manière à obtenir un algorithme de recherche trichotomique qui sépare la liste en trois sous-listes de tailles approximativement égales. Évaluer le coût de cet algorithme (on pourra se restreindre au cas où n est une puissance de 3), et comparer avec le coût de l'algorithme dichotomique.

