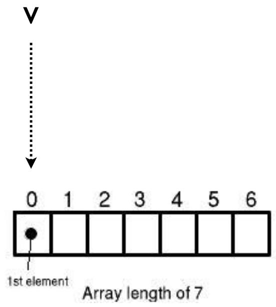
OPTION PF2 - Automne 2016
Jean-Paul Roy
L2-Info&Maths, Faculté des Sciences

http://deptinfo.unice.fr/~roy

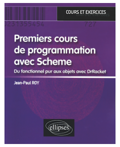
Cours n°10

Université Nice SOPHIA ANTIPOLIS

Les Vecteurs



$$\vec{v}\begin{pmatrix} x_0 \\ x_1 \\ \cdots \\ x_{n-1} \end{pmatrix}$$



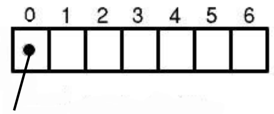
pp. 265-269

Avantages et inconvénients des Listes

- *des **listes <u>chaînées</u>** de Lisp/Scheme, pas celles de Python !...*

POUR	CONTRE
• Programmables par récurrence, non mutables. • Le texte d'une fonction est une liste ! Donc peut être traité comme un objet de calcul [*analysé, transformé, compilé*...]. • Extensions syntaxiques.	• Non mutables. • Accès **séquentiel**. L'élément numéro k est accessible en temps O(k). *- pas gênant pour les arbres...*

Les Vecteurs



Vecteur de longueur 7

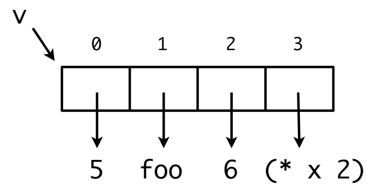
le premier élément (ou première composante) a pour numéro 0

- **Un vecteur est une suite finie de variables numérotées** qui sont les composantes du vecteur [analogie avec un vecteur en maths].
- Ce sont les ***tableaux*** de C/Java, de **taille fixée** et non modifiable.
- Contrairement à C/Java, un vecteur Scheme peut contenir des **éléments de type quelconque** [nombres, listes, symboles, vecteurs...]
- L'accès à l'élément numéro k se fait en temps **O(1)**. On gagne en **efficacité** ce qu'on perd en souplesse : **impossible d'ajouter ou de supprimer un élément** !
- Le choix entre liste et vecteur se décide en fonction du problème !

Construction et affichage d'un vecteur

- Construction en **extension** avec la fonction `(vector x y ...)` :

```
> (define v (vector 5 'foo (* 2 3) '(* x 2)))
> v
#(5 foo 6 (* x 2))
> (vector? v)
#t
> (vector-length v)
4
```



- Il est *déconseillé* de définir un vecteur de manière littérale [à réserver aux vecteurs ***constants*** !] :

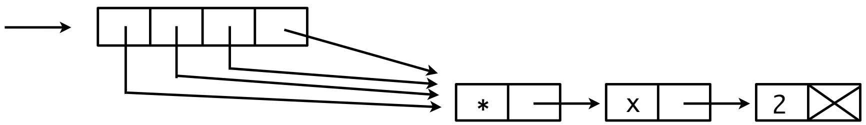
```
> (define v1 #(5 foo 6 (* x 2)))
> v1
#(5 foo 6 (* x 2))
```

```
> (equal? v v1)
#t
> (eq? v v1)
#f
> (immutable? v)
#f
> (immutable? v1)
#t
```

- Construction par la **longueur** `(make-vector n x)`. Mais attention :

```
> (define v2 (make-vector 4 1))
> v2
#(1 1 1 1)
> (define v3 (make-vector 4 '(* x 2)))
> v3
#((* x 2) (* x 2) (* x 2) (* x 2))
```

v3



- Construction par **compréhension** `(build-vector n f)` :

```
> (build-vector 5 (lambda (i) (* i 3)))
#(0 3 6 9 12)
```

Comment programmeriez-vous `build-vector` *s'il n'existait pas ?...*

Accès/Modification des composantes d'un vecteur

- **L'accès à l'élément numéro k** [0 ≤ k ≤ n-1] se fait <u>en temps constant</u> !

```
> (define v (build-vector 10 sqr))
> v
#(0 1 4 9 16 25 36 49 64 81)
> (vector-ref v 5)                    ; v[5] en Java/C/Python
25
> (vector-ref v 20)
ERROR : index 20 out of range [0, 9]
```

- **La modification de l'élément numéro k** :

```
> (vector-set! v 5 30)                        ; v[5] = 30
> v
#(0 1 4 9 16 30 36 49 64 81)
> (vector-set! v 5 (+ (vector-ref v 5) 1))    ; v[5] = v[5]+1
> v
#(0 1 4 9 16 31 36 49 64 81)
```

Exemple 1 : maximum d'un vecteur de nombres réels

```
(define (maximum v)                 ; avec in-range
  (define n (vector-length v))
  (define res (vector-ref v 0))
  (for ([i (in-range 1 n)])
    (set! res (max res (vector-ref v i))))
  res))
```

```
> (maximum #(6 2 8 -3 5 1))
8
```

```
(define (maximum v)                 ; avec in-vector
  (define res (vector-ref v 0))
  (for ([x (in-vector v)])
    (set! res (max x res)))
  res)
```

Exemple 2 : échange dans un vecteur

- On veut échanger les éléments numéro i et j d'un vecteur v :

```
(define (vector-swap! v i j)          ; v[i] ↔ v[j]
  ...)
```

```
> v
#(0 1 4 9 16 31 36 49 64 81)
> (vector-swap! v 2 4)
> v
#(0 1 16 9 4 31 36 49 64 81)
```

Cf TP !

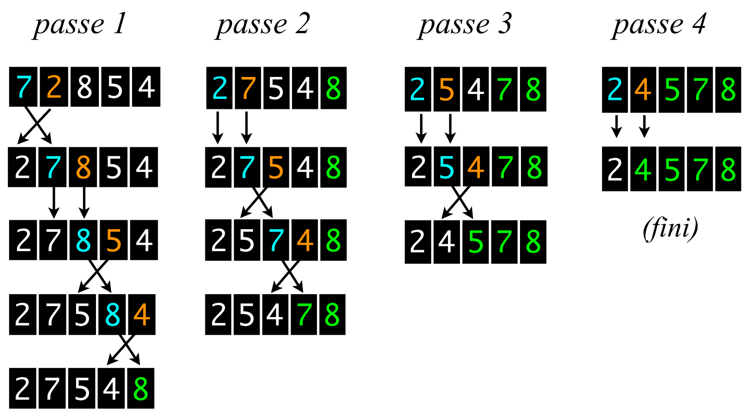
Exemple 3 : somme des éléments d'un vecteur

```
(define (somme v)
  (apply + (vector->list v)))       ; juste pour le fun :-)
```

Vous trouverez sûrement une solution plus... orthodoxe !

Exemple 3 : le tri de la bulle [bubble sort]

- *Fun* aussi, mais mauvaise complexité : $O(n^2)$.
- Chaque passe contribue à faire remonter le maximum vers la droite, comme une bulle vers la surface...




```
(define (bubble-sort! v)
  (define n (vector-length v))
  (for ([passe (in-range 1 n)])
    (for ([i (in-range 0 (- n passe))])
      (when (> (vector-ref v i) (vector-ref v (+ i 1)))
        (vector-swap! v i (+ i 1))))
    (printf "--- passe ~a --> v = ~s\n" i v)))
```

```
> (define w #(7 5 8 4 3))
> (bubble-sort! w)
--- passe 1 ---> v = #(5 7 4 3 8)
--- passe 2 ---> v = #(5 4 3 7 8)
--- passe 3 ---> v = #(4 3 5 7 8)
--- passe 4 ---> v = #(3 4 5 7 8)
> w
#(3 4 5 7 8)
```
← *w reste modifié en sortie !*

- La librairie `"sorting-6"` fournit une fonction `(vector-sort! rel? v)` de tri sur vecteur. Demander `(require (lib "sorting-6.ss" "rnrs"))`

Exemple 4 : le Triangle de Pascal

```
1
1  1
1  2   1
1  3   3   1
1  4   6   4   1
1  5   10  10  5   1
1  6   15  20  15  6   1
```

C_5^3

$$C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$$

- On se propose de calculer le triangle jusqu'à la ligne n [n ≥ 0], sous la forme d'un vecteur de vecteurs ! Une sorte de matrice...
- Commençons par définir le squelette du résultat :

```
> (build-vector 6 (lambda (i) (make-vector (+ i 1) 1)))
#(#(1) #(1 1) #(1 1 1) #(1 1 1 1) #(1 1 1 1 1) #(1 1 1 1 1 1))
```

- Ensuite nous itérons sur ce vecteur, en remplissant chaque ligne en fonction de la précédente :

```
(define (pascal n)                                        ; n ≥ 2
  (define res (build-vector (+ n 1)
                 (lambda (i) (make-vector (+ i 1) 1))))
  (for ([i (in-range 2 (+ n 1))])
    (define v (vector-ref res (- i 1)))     ; ligne i-1
    (define w (vector-ref res i))           ; ligne i
    (for ([j (in-range 1 i)])
      (vector-set! w j
        (+ (vector-ref v j) (vector-ref v (- j 1))))))
  res)
```

```
> (pascal 5)
#(#(1) #(1 1) #(1 2 1) #(1 3 3 1) #(1 4 6 4 1) #(1 5 10 10 5 1))
```

- **Affichage** propre du Triangle de Pascal jusqu'à la ligne n incluse :

```
(define (pascal-print n)
  (define T (pascal n))
  (for ([i (in-range 0 (+ n 1))])
    (for ([j (in-range 0 (+ i 1))])
      (printf "~a\t" (vector-ref (vector-ref T i) j)))
    (printf "\n")))
```

i = numéro de ligne
j = numéro de colonne

```
> (pascal-print 5)
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
```

Exemple 5 : les matrices

```
#lang racket
(require math) ; non utilisé ici
(define A (matrix [[0 1]
                   [1 2]
                   [2 3]]))
```

- Une matrice sera vue ici comme un vecteur dont les éléments sont les vecteurs-lignes de la matrice !

$$\begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \end{pmatrix} \qquad A = \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix}$$

```
(define (build-matrix nblig nbcol f)      ; M[i,j] = (f i j)
  (build-vector nblig                     ; i = ligne, j = colonne
                (lambda (i)
                  (build-vector nbcol
                                (lambda (j) (f i j))))))
```

```
> (define A (build-matrix 3 2 +))
> A
#(#(0 1) #(1 2) #(2 3))
```

```
> (define A #(#(0 1)
              #(1 2)
              #(2 3))
```

```
(define (nb-lignes M)
  (vector-length M))

(define (nb-colonnes M)
  (vector-length (vector-ref M 0)))

(define (matrix-ref M i j)               ; Mij
  (vector-ref (vector-ref M i) j))       ; i=0..nblig-1 et j=0..nbcol-1

(define (matrix-set! M i j x)            ; Mij = x
  (vector-set! (vector-ref M i) j x))
```

La boîte à outils du type abstrait

```
(define (print-matrix M)
  (for ([i (in-range 0 (nb-lignes M))])
    (printf "[")
    (for ([j (in-range 0 (- (nb-colonnes M) 1))])
      (printf "~a\t" (matrix-ref M i j)))
    (printf "~a]\n" (matrix-ref M i (- (nb-colonnes M) 1)))))
```

```
> (print-matrix A)
[0   1]
[1   2]
[2   3]
```

à la Maple...

- Les algorithmes sur matrices se basent sur le type abstrait :

```
> (define (mat-id n)        ; matrice-unité d'ordre n
    (build-matrix n n delta))
> (define (delta i j) (if (= i j) 1 0))))
> (mat-id 3)
#(#(1 0 0) #(0 1 0) #(0 0 1))
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
(define (mat+ M1 M2)         ; addition matricielle
  (let ((l1 (nb-lignes M1)) (c1 (nb-cols M1))
        (l2 (nb-lignes M2)) (c2 (nb-cols M2)))
    (when (not (and (= l1 l2) (= c1 c2)))
        (error 'mat+ "Mauvais formats de matrices !" M1 M2))
    (build-matrix l1 c1
      (λ (i j) (+ (matrix-ref M1 i j) (matrix-ref M2 i j))))))
```

```
> A
#(#(0 1) #(1 2) #(2 3))
> (mat+ A A)
#(#(0 2) #(2 4) #(4 6))
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 2 & 4 \\ 4 & 6 \end{pmatrix}$$

Exemple 6 : le drapeau hollandais

• Problème célèbre dû à Edsger Dijkstra. Il concerne un vecteur mutable ne contenant que des symboles `'bleu`, `'blanc`, `'rouge` :

```
> (define DRAP (random-drapeau 10))
> DRAP
#(rouge rouge blanc rouge bleu blanc bleu rouge blanc rouge)
```

• Il s'agit de **trier** ce vecteur sous la forme :

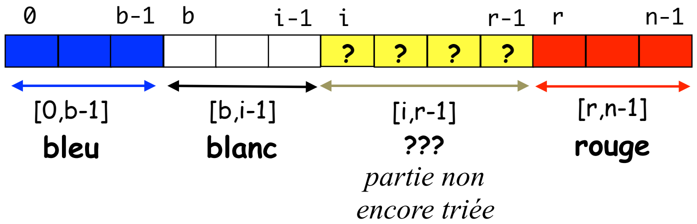
```
> (trier-drapeau DRAP)
#(bleu bleu blanc blanc blanc rouge rouge rouge rouge rouge)
```

• Mais avec la contrainte : **n échanges au plus sur place** !

• *C'est donc mieux que la borne inférieure d'un tri, qui est O(n log n)...*

• Le problème illustre l'utilisation d'un **invariant** pour construire la boucle. Quelle est la **situation générale** en plein milieu du calcul ?

Le vecteur est partiellement trié !

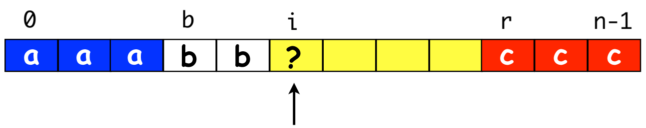


• Au départ : `b = 0`, `i = 0`, `r = n`, et la situation générale est vérifiée ! Il reste à la maintenir à chaque tour de boucle...

• On stoppe lorsque `[i,r-1]` = ∅, soit `i ≥ r`.

• Algorithme en *pseudo-langage* :

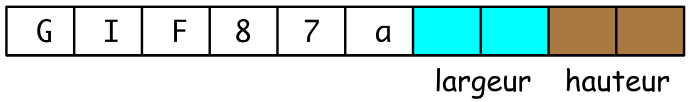
```
Algorithme DrapeauHollandais (vecteur v) : void
 b = 0 ; i = 0 ; r = length(v)
 tant que i < r
   x = v[i]
   si x = bleu alors { v[b] ↔ v[i] ; b = b + 1 ; i = i + 1 }
   sinon si x = blanc alors i = i + 1
   sinon { r = r - 1; v[r] ↔ v[i] }
```



Exemple 2 Un **fichier d'octets** est en réalité un fichier de **caractères** lisibles par read-char. Prenons l'exemple d'une image GIF.

`http://www.colosseumbuilders.com/imageformats/gif87a.txt`

• Je me renseigne avec Google et je trouve des informations...



• Celles-ci m'expliquent que les 6 premiers octets [char] contiennent le type de l'image : GIF87a ou GIF89a.

• Il me suffit donc d'ouvrir un port d'entrée sur un fichier `xxx.gif` et d'en lire 6 caractères [le type de l'image] puis encore 2 pour la largeur et 2 autres pour la hauteur.

– pour le type il suffit d'afficher les caractères !

– pour chaque dimension, il faut *mélanger* les deux octets en les traduisant d'abord en entiers n_1 et n_2 et en calculant $n_1 + 256*n_2$

- Aussitôt dit, aussitôt programmé !

```
(define (size-of-gif-image fich)
  (define v (make-vector 4))
  (call-with-input-file fich
    (lambda (p-in)
      ; je lis les 6 premiers octets [type de l'image]
      (printf "L'image est de type ~a\n"
               (for/list ([i (in-range   6)]) (read-char p-in)))
      ; je stocke les 4 octets suivants dans un vecteur
      (define v (build-vector 4 (lambda (i) (read-byte p-in))))
      (printf "v = ~a\n" v)
      (printf "L'image a donc pour taille ~a x ~a\n"
              (+ (vector-ref v 0) (* 256 (vector-ref v 1)))
              (+ (vector-ref v 2) (* 256 (vector-ref v 3)))))))
```

```
> (size-of-gif-image "matrix.gif")
L'image est de type (G I F 8 9 a)
v = #(106 1 244 1)
L'image a donc pour taille 362 x 500
```

v

byte	*byte*		
106	1	244	1
poids faible	*poids fort*		

little endian