Variables indicées et tableaux

Pour désigner une séquence d'éléments **de même type**

Notation habituelle :

$$v_1, v_2, v_3, \ldots, v_n$$

On parle de **vecteur**

Exemples

point dans l'espace : (6, 5, –2)

prix moyen du carburant à chaque trimestre : (1.29, 1.34, 1.31, 1.28)

Autres exemples :

—

Structure de tableau

Pour représenter des vecteurs/n-tuples d'éléments de type T (occupant s cellules)

Représentation immédiate : séquence contigüe de Ts

adresses	*contenus*
a	K_1
a + s	K_2
a + 2s	K_3
a + (n-1)s	K_n

Utilisation des tableaux

Sélection d'un élément :

$$x \leftarrow t[k]$$

Affectation d'une valeur à un élément (l'ancienne est effacée)

$$t[k] \leftarrow e$$

Opérations globales sur les tableaux

Affectation globale

$$V \leftarrow (e1, e2, \ldots, en)$$

équivalent à $V[1] \leftarrow e1$, $V[2] \leftarrow e2$, …

$$V \leftarrow W$$

équivalent à $V[1] \leftarrow W[1]$, $V[2] \leftarrow W[2]$, …

Temps d'exécution proportionnel à la taille de V.

Affectation de tranches

$V[i\ldots j] \leftarrow (e_0, \ldots, e_r)$

équivalent à $V[i] \leftarrow e_0$, $V[i+1] \leftarrow e_1$, …, $V[j] \leftarrow e_r$

r = i - j

Un exemple: vérification de date

Problème: écrire un algorithme qui vérifie si une date (jour, mois, année) est correcte

.

fonction vérifDate(j, m, a)

si (j < 1 ou m < 1 ou m > 12) retourne faux

bisextile ← a modulo 4= 0 // correct entre 2000 et 2099)

si m = 1 retourne j ≤ 31

si m = 2 et bisextile retourne j ≤ 29

si m = 2 et non bisextile retourne j ≤ 28

si m = 3 retourne j ≤ 31

...

si m = 12 retourne j ≤ 31

retourne faux

Le même avec un tableau

On crée un tableau des nombres de jours.

nbJours[1..12] ← (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

Qu'on utilise dans la fonction

fonction vérifDate(j, m, a)

si (j < 1 ou m < 1 ou m > 12) retourne faux

si (a modulo 4= 0) nbJours[2] = 29 sinon nbJours[2] = 28

retourne j ≤ nbJours[m]

En Java

```
class GestionDates {
    static int[ ] nbJours = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    static boolean verifDate(int j, int m, int a) {
        if (j < 1 || m < 1 || m > 12) return false;
        if (a % 4 == 0) nbJours[ 2 ] = 29
        else nbJours[ 2 ] = 28;
        return j <= nbJours[ m ]
    }
}
```


Tableau ≠ Ensemble !

Ordre

ensembles : {a, c, g} = {g, c, a}

tableaux : (a, r, g) ≠ (r, g, a)

Accès

tableaux : 1er élément, 2e, 3e, etc.

ensembles : ---

Nb d'occurences

ensembles : {a, c, a, b, a} = {a, c, b}

tableaux : (a, a, a, a) ≠ (a)

Opérations

ensembles : union, intersection, différence

tableau : affectation d'une valeur à un élément

Exemple : le crible d’Eratosthène

Pour trouver tous les nombres premiers de 2 à n (encore !)

- Faire une liste des nombres
- Barrer tous les multiples de 2 supérieurs à 2
- Barrer tous les multiples de 3 supérieurs à 3
- 4 est déjà barré
- Barrer tous les multiples de 5 supérieurs à 5
- 6 est déjà barré
- Barrer tous les multiples de 7 supérieurs à 7
- 8 est déjà barré
- 9 est déjà barré
- 10 est déjà barré
- Barrer tous les multiples de 11 supérieurs à 11

etc. Les nombres qui restent sont premiers (ils ne sont multiples de personne)

Algorithme semi-formel

```
faire une liste des nombres de 2 à n
p <-- 2
tant que ( p * p ≤ n) {
    barrer tous les multiples de p
    p <-- le plus petit nombre non barré supérieur à p
}
```

Remarque

Tout nombre < n qui n’est pas premier est multiple d’un nombre inférieur à $\sqrt{n}$.

Donc on peut arrêter l’algorithme quand p * p > n.

Structure de données : tableau de booléens

au début

v	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

barrer les multiples de 2

v	f	f	v	f	v	f	v	f	v	f	v	f	v	f	v	f
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

puis ceux de 3

v	f	f	v	f	v	f	v	v	v	f	v	f	v	v	v	f
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

etc.

Algorithme avec un tableau de booléens

pour i **de** 2 **à** n { Barré[i] <-- faux }

p <-- 2

max <-- racine carrée de n

tant que (p ≤ max) {

    si (non Barré [p]) {

        // Barrer les multiples de p

        i <-- p + p

        tant que i ≤ n { Barré[i] <-- vrai ; i <-- i + p }

    }

    p <-- p + 1

}

Complexité en temps

√n fois

π (√n) fois

n / p – 1 fois

```
tant que ( p ≤ max) {
    si ( non Barré [p] ) {
        i <-- p + p
        tant que i ≤ n { Barré[ i ] <-- vrai ; i <-- i + p }
    }
    p <-- p + 1
}
```

$\pi(x)$ = nombre de nb. premiers inférieurs à x

nb opérations **[ ]** = $n/2 + n/3 + n/5 + n/7 + n/11 + \ldots + n/p_k - \pi(\sqrt{n})$

(p_k = plus grand nombre premier inférieur à $\sqrt{n}$)

Le calcul de la complexité … est complexe

nb opérations **[ ]** = $n/2 + n/3 + n/5 + n/7 + n/11 + \ldots + n/p_k - \pi(\sqrt{n})$

(p_k = plus grand nombre premier inférieur à $\sqrt{n}$)

On sait

$$1/p_1 + 1/p_2 + 1/p_3 + \ldots + 1/p_k \sim= \log(\log(k))$$

Mais que vaut k ?

Combien y a-t-il de nombres premiers inférieurs à √n ?

Legendre a trouvé :

$$\pi(x) \sim= x / (\log(x) - 1.0836) \text{ (Legendre)}$$

Donc

nb. opérations $\sim= n \log(\log(\sqrt{n} / (\log(\sqrt{n} - 1.08))$

$= n \log(\log(\sqrt{n})) - n \log(\log(\log(\sqrt{n} - 1.08)))$

$\in O(n \log(\log(\sqrt{n})))$

Accès associatif / recherche

But : trouver une valeur dans un tableau T

Le meilleur algorithme : regarder successivement dans T[0], T[1], … etc.

Complexité : O(taille de T)

Donc

Tout algorithme basé sur l’accès associatif est inefficace

(à moins d’avoir beaucoup de processeurs en parallèle)

Algorithmes 1

T un tableau indicé de 0 à n-1

résultat = 1ère position de x dans T

résultat = -1 si x n’est pas dans T

```
fonction recherche (x, T) {
    i ← 0
    tant que ( i < n et T[ i ] ≠ x ) i ← i + 1
    si i = n retourne –1 sinon retourne i
}
```

Cet algorithme est **faux**.

Algorithmes 2

T un tableau indicé de 0 à n-1

résultat = 1ère position de x dans T

résultat = -1 si x n'est pas dans T

fonction recherche (x, T) {
    i ← 0
    tant que (i < n) {
        si T[i] = x retourne i
        sinon i ← i + 1
    }
    retourne –1
}

Algorithmes 3

T un tableau indicé de 0 à n-1

résultat = 1ère position de x dans T

résultat = -1 si x n'est pas dans T

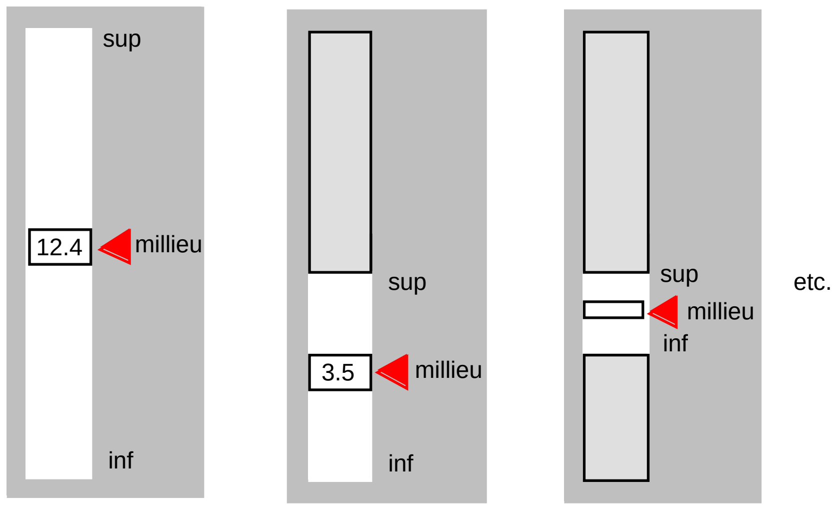
fonction entière recherche (x, T) {
    i ← 0
    tant que (i < n et ensuite T[i] ≠ x) i ← i + 1
    si i = n retourne –1 **sinon** retourne i
}

Utilise l'**évaluation partielle** des expressions booléennes.

Opérateur && en Java.

Recherche dans un tableau trié

On cherche 6.25



Algorithme

Précondition : x est trié par ordre croissant des valeurs.

Invariant : si x se trouve dans T, il est entre les positions inf et sup.

fonction dichotomique(T, x) {
inf <-- 0 ; sup <-- taille T – 1;
tant que inf <= sup {
    millieu <-- (sup + inf) / 2 ;
    si (T[millieu] = x) retourner millieu
    sinon **si** (T[millieu] > x) sup <-- millieu - 1
    sinon inf <-- millieu + 1
}
retourner -1 /* pas trouvé */
}

Complexité

Nombre d'itérations =

si $n = 2^k$

au maximum k itérations ($2^k, 2^{k-1}, 2^{k-2}, \ldots, 8, 4, 2, 1$)

$k = \log_2(n)$

Complexité : $O(\log_2(n))$

Tableaux et tris

On considère un tableau A d'éléments de type T

On suppose qu'il y a une opération ≤ qui permet de comparer deux T (relation d'ordre total)

On veut produire un nouveau tableau A' tel que

- les éléments de A' sont les mêmes que ceux de A
- si $0 \le i < j < n$ alors $A'[i] \le A'[j]$

Tri par recherche du plus petit

Principe :

- chercher le plus petit élément de A
- l'échanger avec A[0]
- chercher le plus petit dans A[1 .. n–1]
- l'échanger avec A[1]
- chercher le plus petit dans A[2 .. n–1]
- l'échanger avec A[2]
- etc.
-

Algorithme - par recherche du plus petit

```
trier (tableau de n éléments A)
pour i de 0 à n-2 {
    min ← i
    pour j de i+1 à n–1 {
        si A[ j ] < A[ min ] { min ← j }
    }
    t ← A[ i ] ; A[ i ] ← A[ min ] ; A[ min ] ← t
}
```

Complexité

Nombre de comparaisons :

$$n-1 + n-2 + \ldots + 2 + 1$$

$$= n(n-1)/2$$

$$\in O(n^2)$$

Recherche du plus petit - preuve

```
pour i de 0 à n-2 {
    min ← i
    pour j de i+1 à n–1 {
        si A[ j ] < A[ min ] { min ← j }
        --> A[ min ] ≤ A[ k ] (i ≤ k ≤ j)
    }
    --> A[ min ] < A[ k ] (i ≤ k ≤ n–1)
    t ← A[ i ] ; A[ i ] ← A[ min ] ; A[ min ] ← t
    --> A[ i ] ≤ A[ k ] (i ≤ k ≤ n–1)
}
--> A[ i ] ≤ A[ k ] (i ≤ k ≤ n–1) (0 ≤ i ≤n–2)
```


Tri par bulles

```
pour i de 0 à n–2 {
    si A[ i+1 ] < A [ i ] {
        // mettre A [ i+1 ] à sa place
        j <-- i
        tant que j ≥ 0 et ensuite A[ j ] > A [j+ 1] {
            échanger A[ j ] et A [ j+1 ]
            j <-- j–1
        }
    }
}
```


Exemple

1 -- 4 -- 7 -- 21 -- 22 -- 6 -- 44 -- 12

....

....

....

1 -- 4 -- 7 -- 21 -- 22 -- 6 -- 44 -- 12

1 -- 4 -- 7 -- 21 -- 6 -- 22 -- 44 -- 12

1 -- 4 -- 7 -- 6 -- 21 -- 22 -- 44 -- 12

1 -- 4 -- 6 -- 7 -- 21 -- 22 -- 44 -- 12

1 -- 4 -- 6 -- 7 -- 21 -- 22 -- 44 -- 12

1 -- 4 -- 6 -- 7 -- 21 -- 22 -- 44 -- 12

Tri "Shellsort"

Une succession de (pseudo) tris pas bulle

Avec des écarts E de 2^n, 2^{n-1}, 2^{n-2}, …, 4, 2, 1

```
    E <-- de plus gde puissance de 2 inférieure à N
    tant que E > 0
        pour i de 0 à n – E
            si A[ i+E ] < A [ i ]
                // "descendre" A [ i + E ]
                j <-- i
                tant que j ≥ 0 et ensuite A[ j ] > A[ j+ E ]
                    échanger A[ j ] et A [ j+E ]
                    j <-- j – E
        E <-- E / 2
```

Quicksort

procedure Trier (T, a, b)

si b – a < 3 : trier par échange

sinon {

<< *Choisir une valeur pivot p dans T[a … b]*

(p.ex. la 1ère ou au hasard) >>

<< *(Partition) déplacer les éléments de T[a … b]*

de telle manière que

T[m] = p

T [a … m–1] ne contient que des valeurs ≤ p

T [m+1 … b] ne contient que des valeurs > p >>

Trier(T, a, m–1) ; Trier(T, m+1, b)

}

Technique de déplacement

Déplacer les éléments de T[a … b] de telle manière que

T [a … m–1] ≤ p ; T[m] = p ; T [m+1 … b] > p

Algorithme

```
i <-- a ; j <-- b
p <-- T[(a + b) / 2] // pivot
tant que i ≤ j {
    tant que j ≥ a et ensuite T[ j ] ≥ p { j <-- j – 1}
    tant que i ≤ b et ensuite T[ i ] < p { i <-- i + 1}
    si i < j {
        x <-- T[ i ] ; T[ i ] <-- T[ j ] ; T[ j ] <-- x // échange
        i <-- i+1 ; j <-- j–1
}
```

Complexité en temps : O(b – a)

Complexité de Quicksort

Dans le meilleur des cas : on divise en deux parties égales à chaque fois

T(a, b) = O(b – a) + T(a, m–1) + T(m+1, b)

T(a, m–1) = O(m–1 – a) + T(a, k–1) + T(k+1, m–1)

T(m+1, b) = O(b – m–1) + T(m+1, r–1) + T(r+1, b)

T(a, b) = 1 répartition + 2 tris

= 1 répartition + 2 répartitions + 4 tris

= 1 répartition + 2 répartitions + 4 répartitions + 8 tris

etc.

On a $\log_2$(b–a) "étages"

==> Complexité T(a, b) : O($\log_2$(b–a) (b–a))

Théorème. Un tri basé sur la comparaison ne peut faire mieux (en moyenne).

Mauvais cas

Si on n'a pas de chance, les partitions sont déséquilibrées

Au pire : T[a … b] --> T[a], T[a+1 … b]

On en revient au tri par recherche du plus petit : O(n2)

Comment bien choisir le pivot ?

T[a] : mauvais si T est déjà trié

T[(a+b)/2] : bon si T est déjà trié

médiane d'un échantillon de valeurs (évite le pire cas)