



# LSINF1121: Intro 2

## Algorithmes de tri et recherche dichotomique

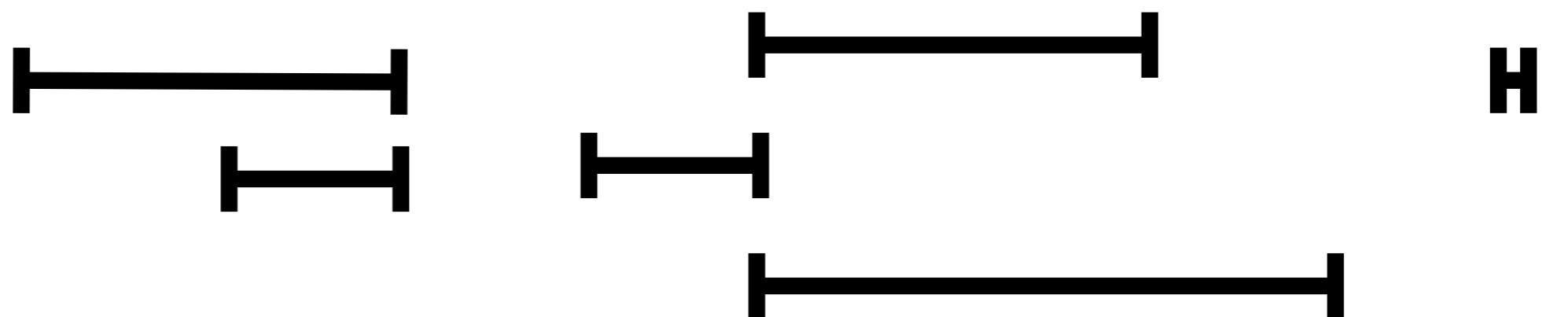
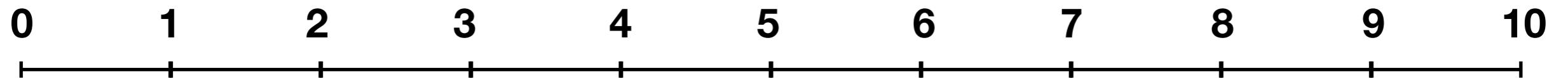
*Pierre Schaus*

# Union d'intervalles

Écrivez une méthode qui prend en entrée un tableau d'intervalles et qui retourne l'union de ces intervalles comme un tableau d'intervalles disjoints. On considère que les intervalles d'input sont donnés sous la forme de deux tableaux  $int[] min$ ,  $int[] max$ ; où le  $i$ ème intervalle est donné par  $(min[i], max[i])$ . Exemple d'entrée  $min=[5,0,1,6,2]$   $max=[7,2,2,8,3]$  donnerait en sortie  $min=[0,5]$ ,  $max=[3,8]$ .

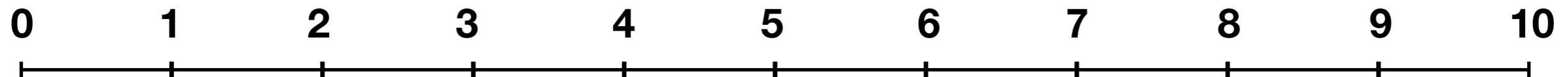
Ecrivez le pseudo-code. Quelle est la complexité de votre méthode ?

# Union of intervals



Input:  $[10,10],[2,4],[3,4],[5,6],[6,9],[6,8]$  Output:  $[2,4],[5,9],[10,10]$

# Algo1



**min=2 max=4**



**min=2 max=4**



**new Interval(2,4) min=5 max=6**



**min=5 max=8**



**min=5 max=9**



**new Interval(5,9) min=10,max=10**

H

**new Interval(10,10)**



H

**Step2: Considerer les intervalles triés un par un et mettre à jour les valeur min/max**

**Step1: trier les intervalles en fonction de leur minimum**

# Algo1

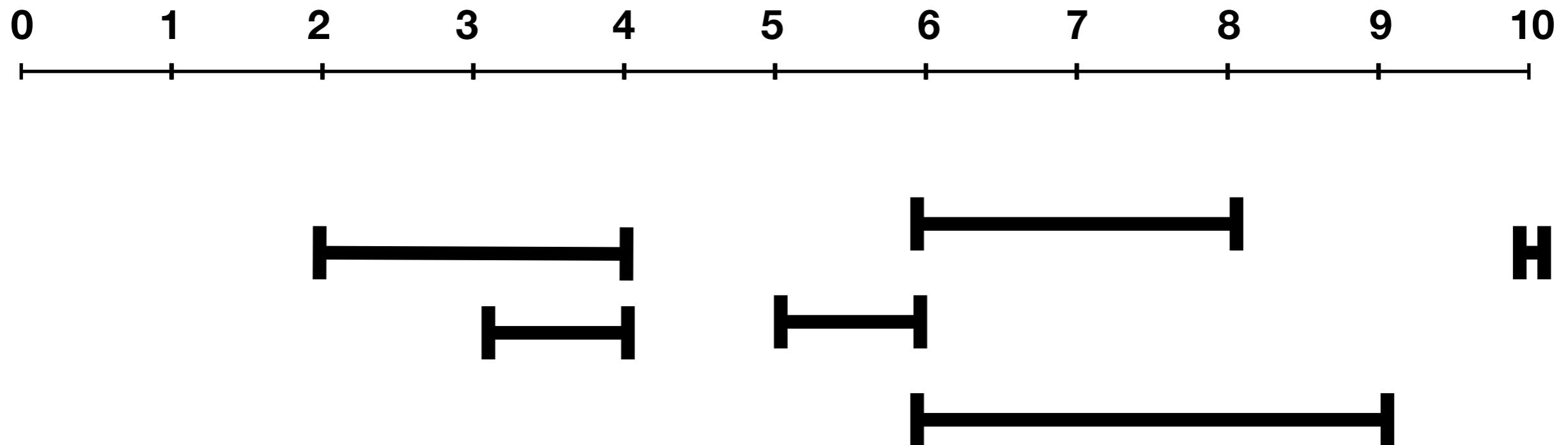
```
public static Interval [] union(Interval [] intervals) {  
    if (intervals.length == 0) return new Interval[0];  
    ArrayList<Interval> res = new ArrayList<>();  
    Arrays.sort(intervals);  
    int min = intervals[0].min;  
    int max = intervals[0].max;  
    int i = 1;  
    while (i < intervals.length) {  
        if (intervals[i].min > max) {  
            res.add(new Interval(min,max));  
            min = intervals[i].min;  
            max = intervals[i].max;  
        } else {  
            max = Math.max(max,intervals[i].max);  
        }  
        i++;  
    }  
    res.add(new Interval(min,max));  
    return res.toArray(new Interval[0]);  
}
```

# Complexité temporelle ?

- $O(n)$  où  $n$  est le nombre d'intervalles en input
- $\Theta(n)$  où  $n$  est le nombre d'intervalles en input
- $O(n \log(n))$  où  $n$  est le nombre d'intervalles en input
- $\Theta(n \log(n))$  où  $n$  est le nombre d'intervalles en input



# Autre approche: algorithme de Sweep



- On crée une séquence triée d'événement en début et fin d'intervalle (+1 pour ouverture d'intervalle et -1 pour la fermeture).
- $(2,+1), (3,+1), (4,-1), (4,-1), (5,+1), (6,+1), (6,+1), (6,-1), (8,-1), (9,-1), (10,+1), (10,-1)$
- Pour chaque nouveau pas de temps, on traite tous les événements un par un et on met à jour le compteur.
- Si lorsqu'on a traité tous les événements le compteur est à zero, on a détecté un nouvel intervalle.

$S = (\min[0], +1), (\max[0] + 1, -1), \dots, (\min[n-1], +1), (\max[n-1] + 1, -1), (+\infty, 0)$ .

Sort the sequence lexicographically:

- increasingly in 'a',
- decreasingly in 'b'

```
int start = S[0].min
overlap = S[0].b // always +1
union = {}
int i = 1
while (i < 2n) {
    overlap += S[i].b
    if (overlap == 0) {
        // closed interval
        union.append([start..S[i].a])
        start = S[i+1].a // fine because we have a dummy element at 2n
    }
    i += 1
}
```

## Question2

Vous devez trier un grand tableau qui a pour propriété qu'il ne contient que des valeurs dans l'ensemble  $\{0, 1, 2\}$ .

- Quel algorithme de tri suggérez-vous?
- Ecrivez le code.
- Quelle sera la complexité pour trier le tableau?
- Discutez cette complexité par rapport à la borne inférieure d'un algorithme de tri (Proposition 1 pages 280-281).

# Warmup

- Si je vous dis que je dois trier un ensemble qui contient
- $1 \times 0, 3 \times 1, 2 \times 5$ , est-ce que vous êtes capables de reconstruire l'input trié ?
  - Oui bien sûr:  $[0, 1, 1, 1, 5, 5]$
- L'algorithme “counting” sort consiste à reconstruire l'input trié sur base des compteurs

# Reconstruction de l'input

- Quelle est la complexité pour reconstruire l'input trié pour des valeurs de compteur  $k_1 \times 0, k_2 \times 1, k_3 \times 2$  ?
  - ▶  $O(k_1+k_2+k_3)$
  - ▶  $\Theta(k_1+k_2+k_3)$
  - ▶  $O(n \log(n))$  où  $n=k_1+k_2+k_3$
  - ▶  $\Theta(n \log(n))$  où  $n = k_1+k_2+k_3$



# Counting sort

```
int [] input =  
    new int[]{0,1,2,0,0,2,0,1,0,2,1,0,1,1,0,0,2,2,2,2,0};  
int [] counters = new int[3];  
for (int i : input) counters[i]++;  
int [] ouput = new int[input.length];  
int j = 0;  
for (int i = 0; i <= 2; i++) {  
    while (counters[i] > 0) {  
        counters[i]--;  
        ouput[j] = i;  
        j++;  
    }  
}
```

# Complexité Counting sort sur des valeurs {0,1,2}

- $n$  = nombre de valeurs à trier

- $O(n)$
- $\Theta(n)$
- $O(n \log(n))$
- $\Theta(n \log(n))$
- $O(n^2)$
- $\Theta(n^2)$

**wooclap**

# Est-ce que le counting sort fonctionne toujours ?

- Si je dois trier des valeurs entre 0 et k ? Quelle est la complexité de Counting-sort pour trier n valeurs ?
  - ▶  $O(n)$
  - ▶  $\Theta(n)$
  - ▶  $O(n+k)$
  - ▶  $\Theta(n+k)$
  - ▶  $O(n \log(n))$
  - ▶  $\Theta(n \log(n))$

woo<sup>clap</sup>

# More robust counting sort

- Mon input contient seulement trois valeurs possibles {20007, 1368910, 900045}, est-ce que je peux adapter counting-sort pour obtenir une complexité de  $\Theta(n)$  ?

- Discutez cette complexité par rapport à la borne inférieure d'un algorithme de tri (Proposition 1 pages 280-281).

Prop1: Aucun algorithme basé sur les comparaisons ne peut garantir pouvoir trier  $N$  objets en moins que  $\sim N \lg N$  comparaisons.

- Le counting sort n'est pas basé sur les comparaisons.
- Il fonctionne bien que s'il y a peu de valeurs différentes à trier (= la différence entre la plus grande et la plus petite n'est pas trop grande  $\leq N$ ).

# Le mode d'un tableau

Le mode d'un tableau de nombres est le nombre qui apparaît le plus fréquemment dans le tableau.

Exemple  $\text{mode}([4, 6, 2, 4, 3, 1]) = 4$ .

- Donnez un algorithme efficace pour calculer le mode d'un tableau de  $n$  nombres.
- Quid si on sait que le tableau ne contient que des valeurs de  $0$  à  $k$  ?

# Mode d'un tableau: Solution 1

1. Trier les éléments
  2. Trouver l'élément qui se répète avec le plus de fois consécutivement en parcourant le tableau trié
- Complexité:
    - $O(n \log(n))$
    - $\Theta(n \log(n))$
    - $\Theta(n)$
    - $O(n)$
    - $\Theta(n^2)$

**wooclap**

# Recall QuickSort

```
public static void sort(Comparable[] a) {
    // StdRandom.shuffle(a);
    sort(a, 0, a.length - 1);
}

// quicksort the subarray from a[lo] to a[hi]
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}

// partition the subarray a[lo..hi] so that a[lo..j-1] <= a[j] <= a[j+1..hi]
// and return the index j.
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo;
    int j = hi + 1;
    Comparable v = a[lo];
    while (true) {
        // find item on lo to swap
        while (less(a[++i], v)) {
            if (i == hi) break;
        }
        // find item on hi to swap
        while (less(v, a[--j])) {
            if (j == lo) break;          // redundant since a[lo] acts as sentinel
        }
        // check if pointers cross
        if (i >= j) break;
        exch(a, i, j);
    }
    // put partitioning item v at a[j]
    exch(a, lo, j);
    // now, a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
    return j;
}
```

# Mode d'un tableau: Solution QuickSort

- Lors de l'étape du pivoting (méthode partition), nous pouvons compter le nombre d'éléments qui sont égaux au pivot et maintenir le meilleur *candidat mode* ainsi que sa fréquence.
- Cette information peut être utilisée pour éviter des appels récursifs dans quick sort:
  - Un appel récursif est lancé seulement si le sous-tableau à partitionner est plus grand que la fréquence du mode courant.

# Et si les range des valeurs est entre 0 et k ?

- Dans ce cas utiliser compter les valeurs dans un tableau de taille k. Si  $k < n$ , complexité  $\Theta(n)$

# Trouver une paire

Étant donné deux ensembles  $S_1$  et  $S_2$  (chacun de taille  $n$ ), et un nombre  $x$ .

Décrivez un algorithme efficace pour trouver s'il existe une paire  $(a,b)$  avec  $a \in S_1, b \in S_2$  telle que  $a+b=x$ .

Quelle est la complexité de votre algorithme?

Quid si les ensembles sont dans des tableaux déjà triés ?

# S1 et S2 sont non triés et taille n

- Trier le premier ensemble S1
- Pour chaque valeur  $v$  de S2, on cherche une valeur  $x-v$  dans S1 trié avec une recherche dichotomique.
- Complexité ?
  - $O(n \log(n))$
  - $\Theta(n \log(n))$
  - $\Theta(n)$
  - $O(n)$
  - $\Theta(n^2)$

**wooclap**

# Quid si les deux ensembles sont déjà triés

- Si pour une valeur  $v$  de  $S_1$ ,  $v + \min(S_2) > x$ , il ne faut pas lancer la recherche dichotomique pour les valeurs suivantes plus grandes que  $v$ .
- Cela consiste à réduire à l'avance les bornes des deux tableaux
- Exemple:  $x = 100$
- $S_1 = [10, 13, 46, 70, 80, \textcolor{red}{101}, 108]$
- $S_2 = [10, 22, 30, 70, 82, \textcolor{red}{104}, 111]$

# Et pour un seul tableau ?

- Trouver deux entrées dont la somme fait  $x$  ?
- Exemple [5,10,1,150,151,155,**18,50**,30]  $x=68$
- Quid si le tableau est déjà trié ?
  - ▶ [1,5,10,**18**,30,**50**,150,151,155]  $x=68$

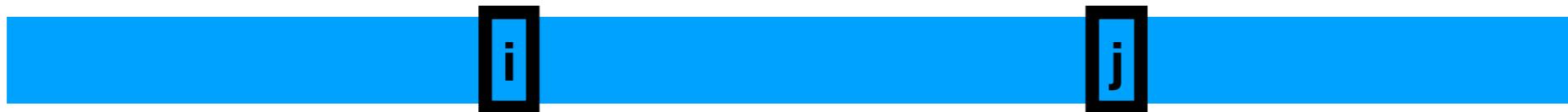
# Pour un tableau trié

- Initialiser  $i = 0, j = t.length - 1$
- Répéter tant que  $i < j$ :
  - ▶ Si  $T[i] + T[j] > x$ 
    - \*  $j = j - 1$
  - ▶ Si  $T[i] + T[j] < x$ 
    - \*  $i = i + 1$
- Sinon return  $(i, j)$

Complexité ?

# Pourquoi cet algorithme fonctionne-t-il ?

- Preuve par induction



- Hypothèse
  - ▶ Pour tout  $k < i$  et  $l \in \{k..n-1\}$ :  $T[k]+T[l] \neq x$
- Algo:  $T[i]+T[j] > x \Rightarrow j = j-1$ 
  - ▶ Il faut montrer
    - \* Pour tout  $k < j$ :  $T[k]+T[j] \neq x$
    - \* Par hyp: ok pour  $k < i$ , également ok pour  $k > i$  car les valeurs sont triées. Cela ne ferait qu'éloigner d'avantage de  $x$ .

# Union de tableaux

- Donnez un algorithme pour calculer l'union de deux ensembles A et B.
- Supposons un second temps, que l'ensemble A déjà trié a une taille  $n$  et l'ensemble B également trié a une taille  $n^2$ . Quelle seraient la complexité, est-ce que votre algorithme change ?

# Union de deux tableaux

- Soit  $m$  et  $n$  la taille des ensembles
- Solution1: tout mettre dans un grand tableau et trier  $O((m+n) \log(m+n))$  et ensuite retirer les doublons.
- Solution2: Trier chaque ensemble séparément et ensuite collecter les éléments en retirant les doublons.  $O(m \log(m) + n \log(n))$ .
  - Si  $m = n$ , Solution1 a la même complexité temporelle que Solution2
- Solution3 (intéressante si grosse différence de tailles sur les tableaux)
  - Supposons des ensembles de taille  $n$  et  $n^2$ , pour chaque élément du petit, on fait une recherche dichotomique sur le grand:  $n \log(n^2) = 2 n \log(n)$ .
  - Pour chaque élément du petit, on fait une recherche dichotomique sur le grand pour savoir s'il faut ajouter cet élément dans l'union.  $O(n \log(n^2))$  pour cette étape donc  $O(n^2 + n \log(n^2))$  au total. Si on avait fait l'inverse au aurait:  $O(n^2 \log(n) + n)$  ce qui est un peu moins bien.

# Recherche dans une matrice

- Étant donné une matrice de nombres entiers qui sont triés le long des lignes et des colonnes, comment trouver un nombre donné dans la matrice de manière efficace ?
- Indice: Il existe un algorithme en temps  $O(n+m)$  pour une matrice  $n \times m$ .
  - Pour cela commencez dans le coin supérieur droit et comparez avec le nombre recherché. Quelles parties de la matrice pouvez-vous élaguer dans votre recherche en fonction du résultat?

- Recherche de 4

1	5	10	11
3	7	12	14
4	8	13	17
8	10	20	25
13	14	22	26

- Recherche de 20

1	5	10	11
3	7	12	14
4	8	13	17
8	10	20	25
13	14	22	26



# LSINF1121: Intro 2

## Algorithmes de tri et recherche dichotomique

*Pierre Schaus*