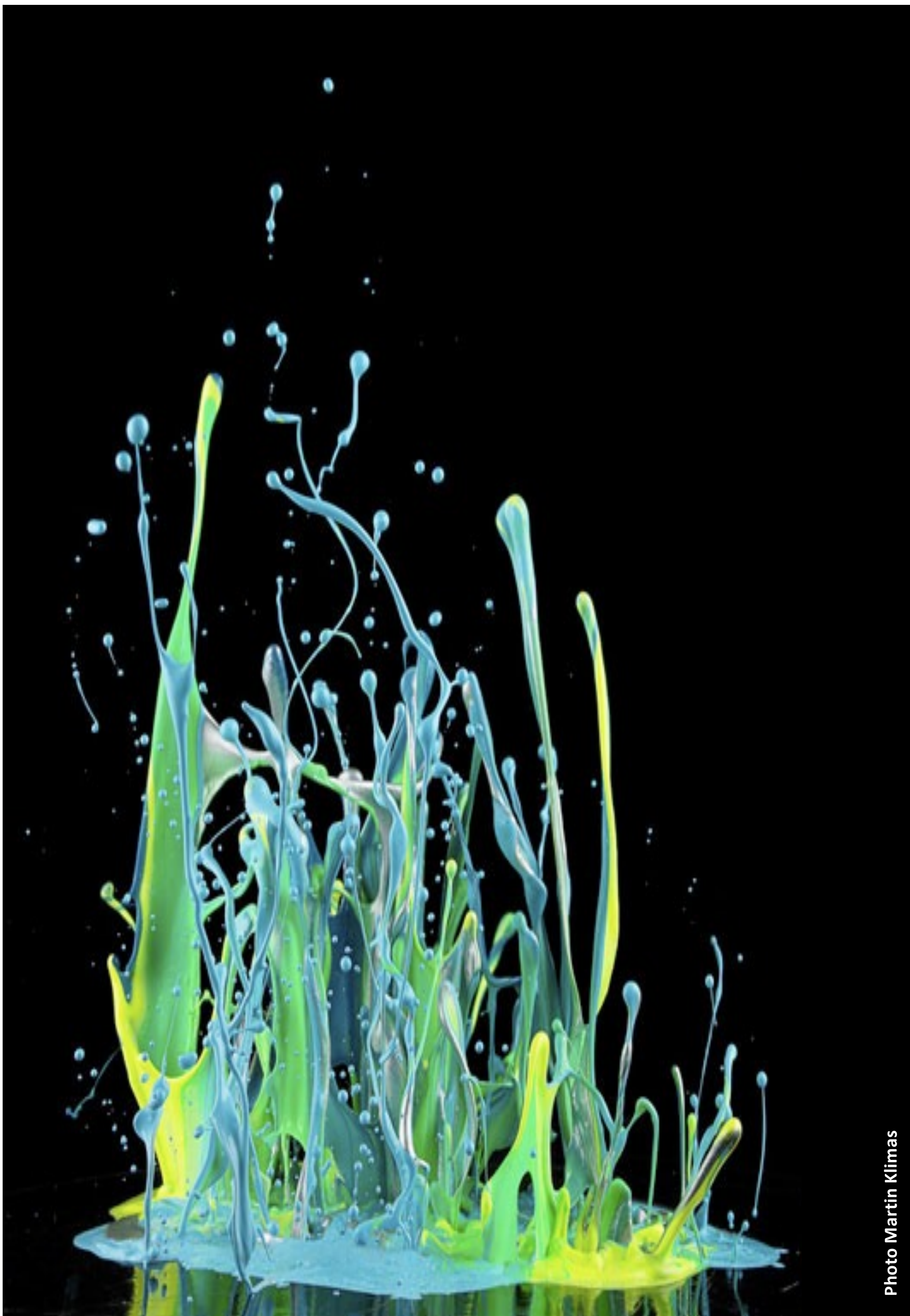



EPFL

# Programmation pour Ingénieur

*Tri et tables*

*ME 3e semestre*

*rev. 2025.1*

Christophe Salzmann

Laboratoire d'Automatique

# Today

## Tri par comparaison

- Tri *brute force*
- Tri rapide

## Tri sans comparaison

- Counting sort
- Bucket sort
- Radix sort

## Tables

- Lookup table
- Table de hachage

# Tri

Définition:

- Ordonner des données de type comparables dans un ordre croissant ou décroissant

But:

- Trier des données
- Accélérer l'accès dans le cas d'une recherche dichotomique

Complexité

- $O(n^2)$: solution *brut force*
- $O(n\log_2 n)$: tri rapide
- $O(n)$: cas particuliers

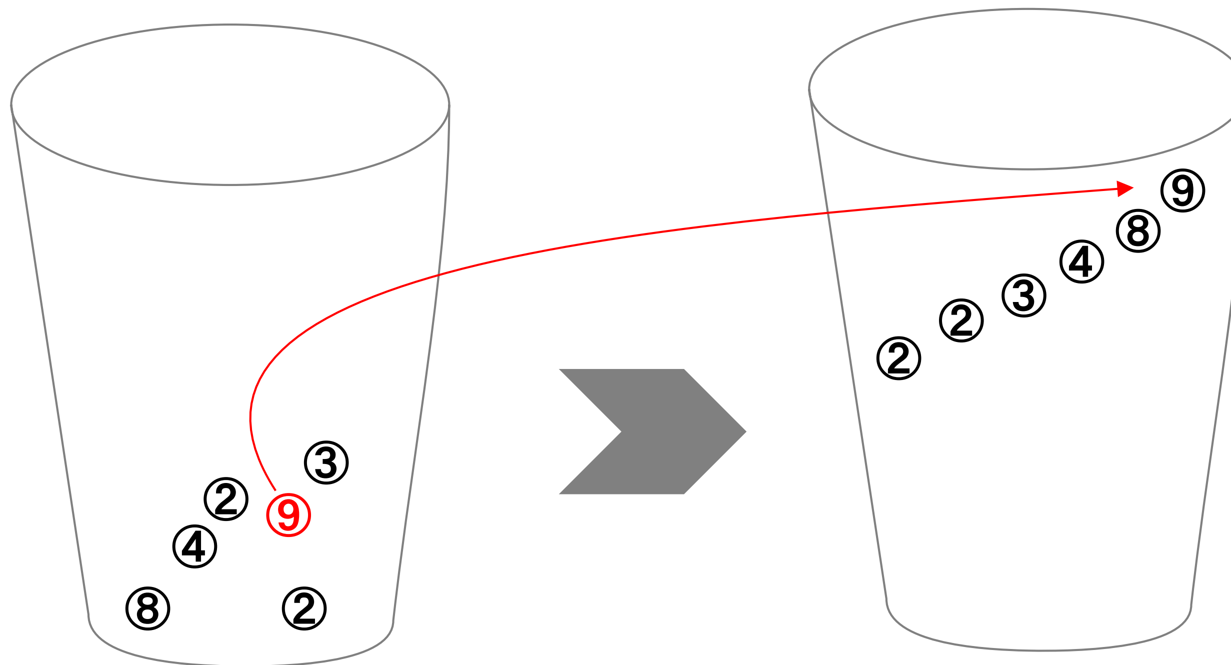
Rappel implémentation de listes avec

- Tableaux:
  - ajout/suppression coût élevé, accès direct, taille mémoire fonction des éléments
- Pointeurs:
  - ajout/suppression coût faible, pas d'accès direct, surcoût taille mémoire, complexité

# Tri par bulle

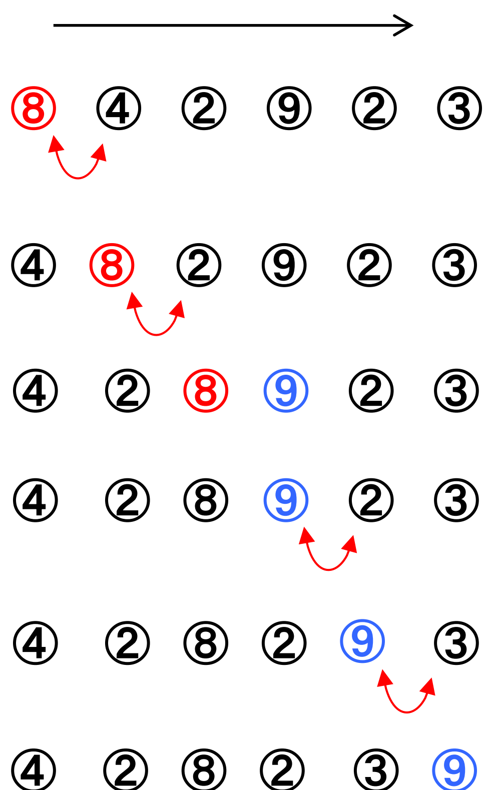
## Idée

Faire "remonter" la plus grande valeur, de la même manière qu'une bulle dans un verre. Faire de même avec la seconde plus grande valeur et ainsi de suite.



# Tri par bulle

Faire "remonter" la plus grande valeur en partant de la gauche.




8 est la valeur la plus à gauche

8 < 4 ? ✘ -> échange

8 < 2 ? ✘ -> échange

8 < 9 ? ✔ -> pas d'échange,
9 devient la plus grande valeur

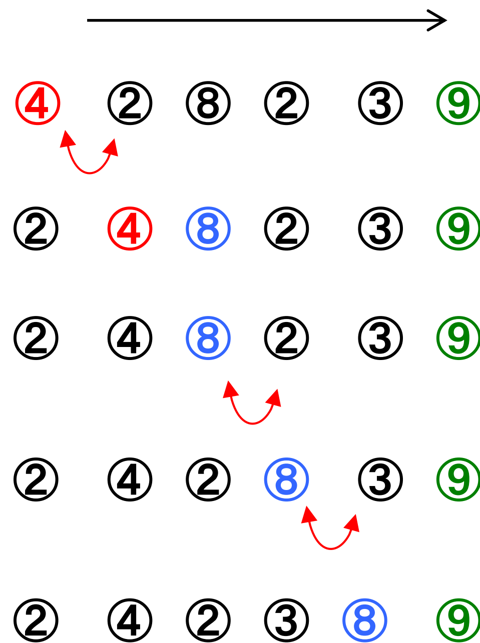
9 < 2 ? ✘ -> échange

9 < 3 ? ✘ -> échange

Après le premier passage,
la plus grande bulle est
placée correctement.

# Tri par bulle

Faire "remonter" la prochaine plus grande valeur.



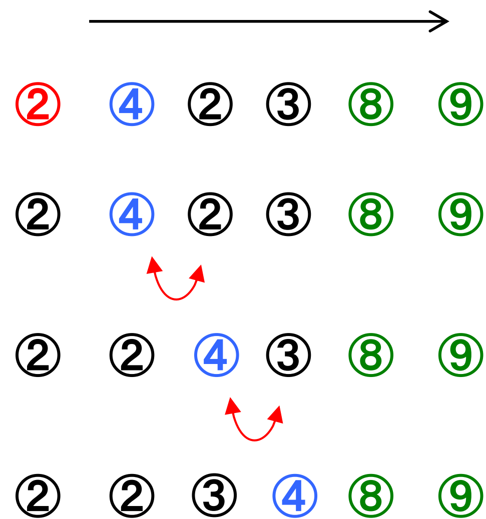
4 est la valeur la plus à gauche

...

Après le deuxième passage,
les deux plus grandes bulles
sont placées correctement.

# Tri par bulle

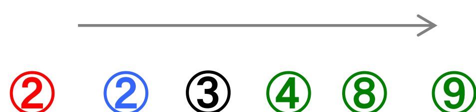
Faire "remonter" la prochaine plus grande valeur.




Après le troisième passage,
les trois plus grandes bulles
sont placées correctement.

# Tri par bulle

Faire "remonter" la prochaine plus grande valeur.



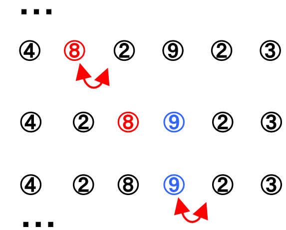
Après le 4$^{e}$ passage, les 4 plus grandes bulles sont placées correctement.

Après le 5$^{e}$ passage, les 5 plus grandes bulles sont placées correctement.

Après le ***n***$^{e}$ passage, les ***n*** bulles sont placées correctement.

-> les données sont triées

# Tri par bulle




Faire "remonter" la prochaine plus grande valeur.

```
void TriBulle(int maListe[], int sz) {
   int i,j,tmp;
   for (i=0; i<sz; i++)                // n-1 passages
      for (j=0; j<sz-i-1; j++)         //    sur les n - i - 1 elements
         if (maListe[j] > maListe[j+1]) {
             tmp = maListe[j];         // echange les 2 valeurs
             maListe[j] = maListe[j+1];
             maListe[j+1] = tmp;
          }
}
```

Complexité: ***n***-1 passages sur ***n***-***i***-1 éléments $\sum_{i=0}^{n-1}(n-i-1) = \frac{n(n-1)}{2} = O(n^2)$

*In place* -> tableau

# Tri par bulle

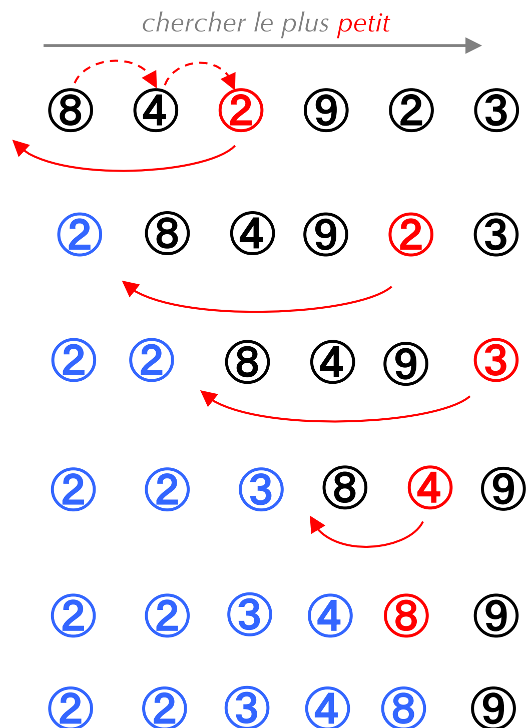
Ajout d'un test pour stopper l'exécution (**break**) de la boucle **for** si le tableau est entièrement trié.

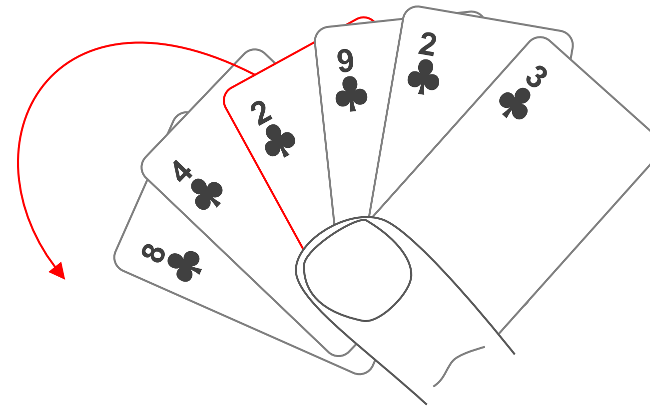
```
Void TriBulle(int maListe[], int sz) {
   int i,j,tmp;
   int inOrder;
   for (i=0; i<sz; i++) {
      inOrder=true;
      for (j=0; j<sz-i; j++)
         if (maListe[j] > maListe[j+1]) {
             inOrder=false;     // echange donc pas entierement trie
             tmp = maListe[j];
             maListe[j] = maListe[j+1];
             maListe[j+1] = tmp;
          }
       if (inOrder) break;     // si aucun echange, arrete la boucle!
      }
}
```

# Tri par insertion

## Idée

Comme avec un jeu de cartes, chercher le plus petit élément du tableau et le placer tout à gauche. Procéder ainsi de suite après s'être décalé d'un élément sur la droite. Complexité en $O(n^2)$.



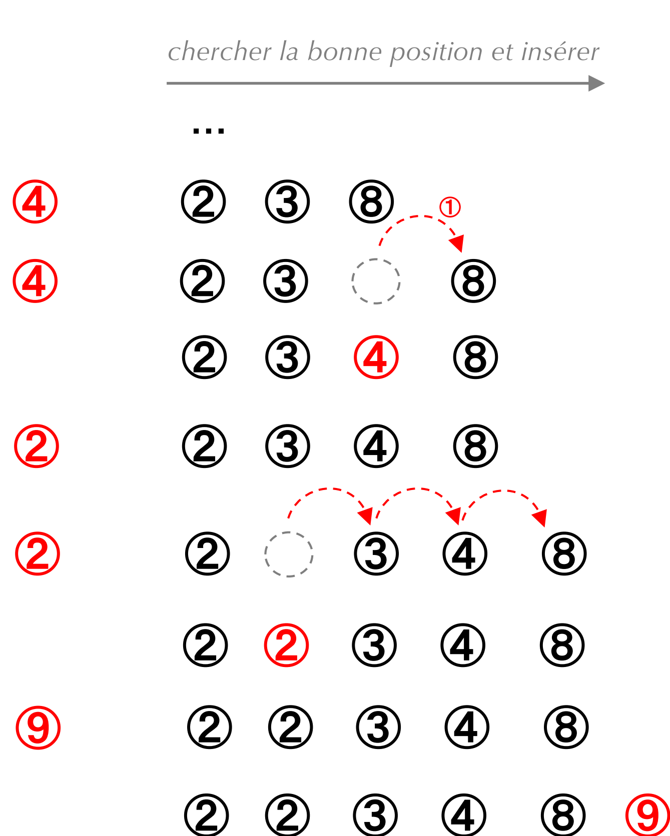



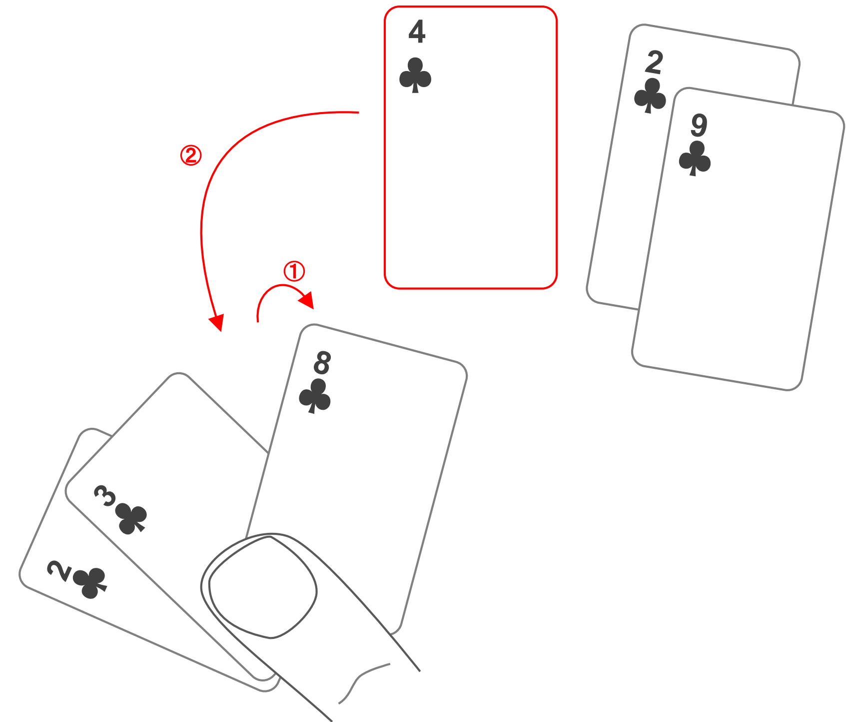
Structure de données? Liste chaînée? Tableau ?

# Tri par insertion 2

## Idée

Insérer le nouvel élément dans le tableau au bon endroit. Le tableau est initialement trié (ou initialisé à 0). Complexité en $O(n^2)$.



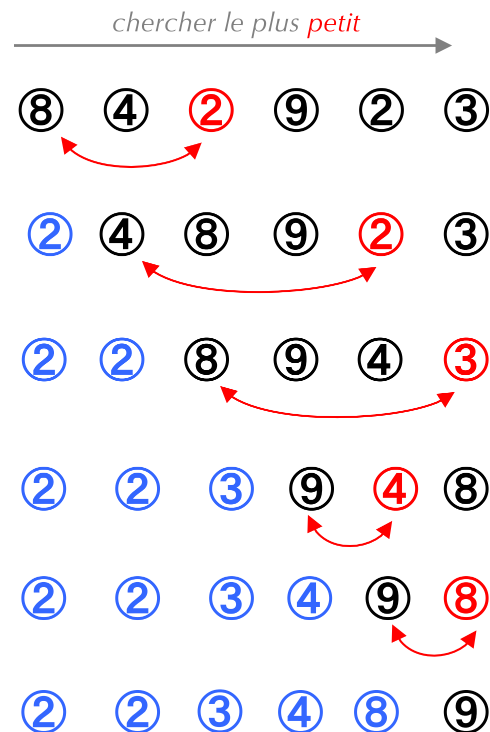



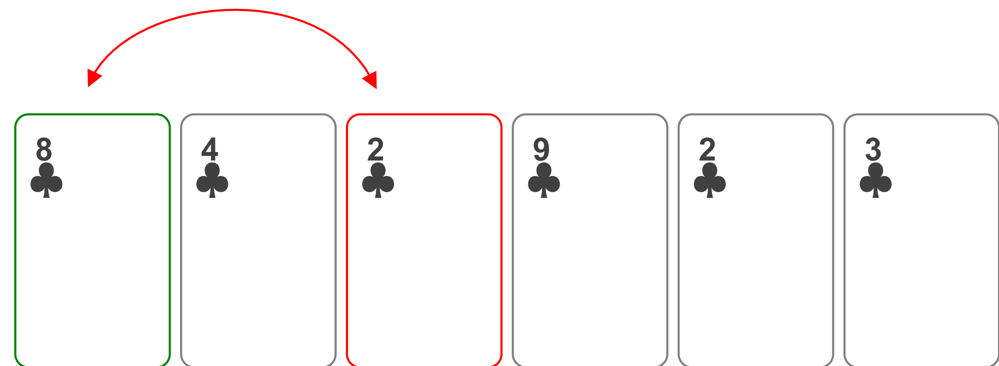

Structure de données? Liste chaînée? Tableau ?

# Tri par sélection

## Idée

Similaire au tri par insertion. Chercher le plus petit élément du tableau et l'échanger (*en place*) avec l'élément le plus à gauche. Procéder ainsi de suite après s'être décalé d'un élément sur la droite. Complexité en $O(n^2)$.



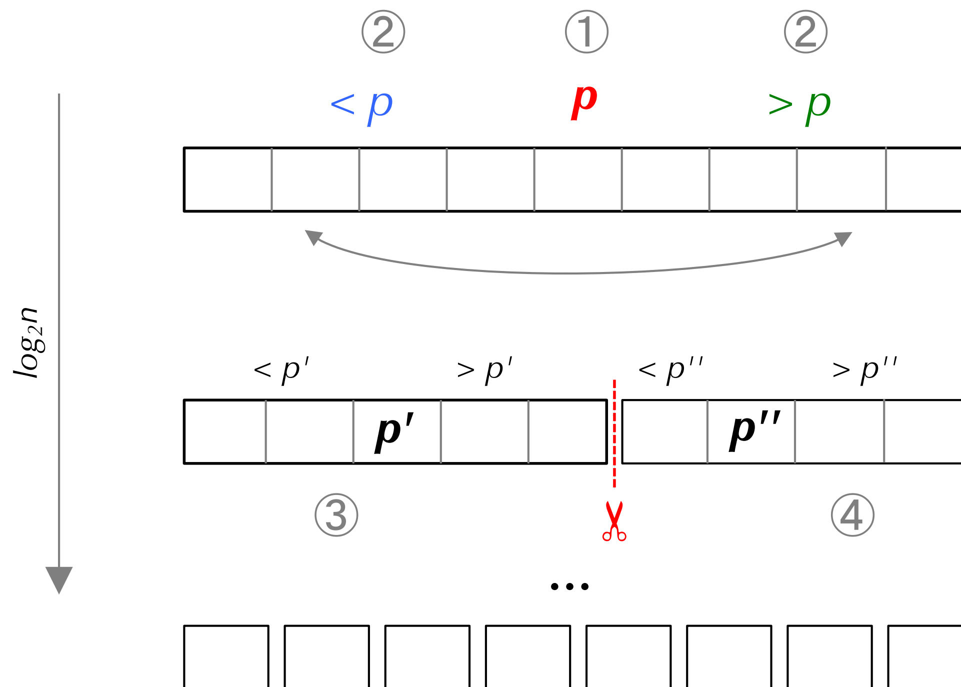




*In-place* -> tableau

# Tri rapide - *quicksort*

## Idée

Employer la stratégie **diviser-pour-régner** afin de réduire le nombre d'opérations. A chaque étape, diviser les données à trier en deux parts ~égales.




1. Choisir un pivot ***p***, généralement la valeur au centre du tableau.
2. Echanger les éléments du tableau de telle manière que les valeurs plus petites que ***p*** se trouvent à gauche de ***p*** et les valeurs plus grandes que ***p*** à droite. Les valeurs égales à ***p*** peuvent aller d'un côté ou de l'autre.
3. Appliquer de manière récursive les points ① et ② sur les parties gauche ③ et droite ④, jusqu'à ce que le tableau soit entièrement trié.

Complexité: en moyenne $O(n\log_2 n)$

# Tri rapide - *quicksort*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 12 | 5 | 26 | 7 | 14 | 3 | 7 | 2 |

1 < 7 ?   2 > 7 ?

pivot = 7

1<7 ✔ -> déplacement sur la gauche

2>7 ✘ -> pas de déplacement, à échanger

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 12 | 5 | 26 | 7 | 14 | 3 | 7 | 2 |

12 < 7 ?

12<7 ✘ -> pas de déplacement, à échanger

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 26 | 7 | 14 | 3 | 7 | 12 |

échange 12<->2

déplacements gauche et droite

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 26 | 7 | 14 | 3 | 7 | 12 |

5 < 7 ?   7 > 7 ?

5<7 ✔ -> déplacement sur la gauche

7>7 ✘ -> pas de déplacement, à échanger

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 26 | 7 | 14 | 3 | 7 | 12 |

26 < 7 ?   7 > 7 ?

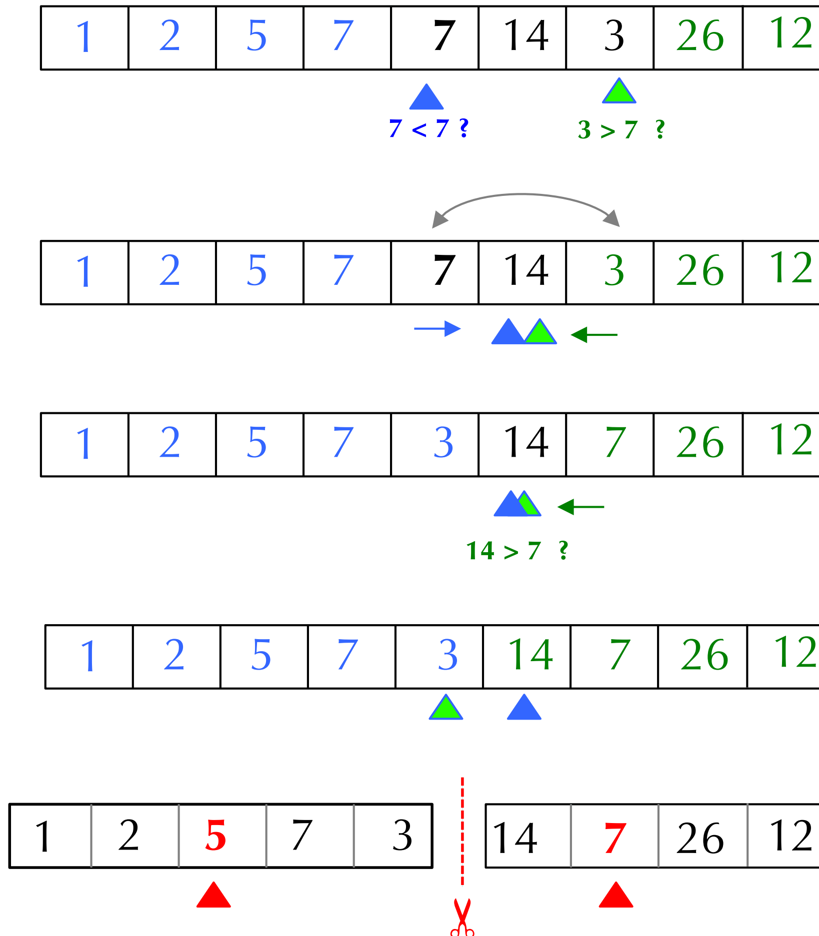
26<7 ✘ -> pas de déplacement, à échanger

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 7 | 7 | 14 | 3 | 26 | 12 |

échange 7<->26

déplacements gauche et droite

# Tri rapide - *quicksort*




7<7 ✘ -> pas de déplacement, à échanger

3>7 ✘ -> pas de déplacement, à échanger

échange 7<->3

déplacements gauche et droite

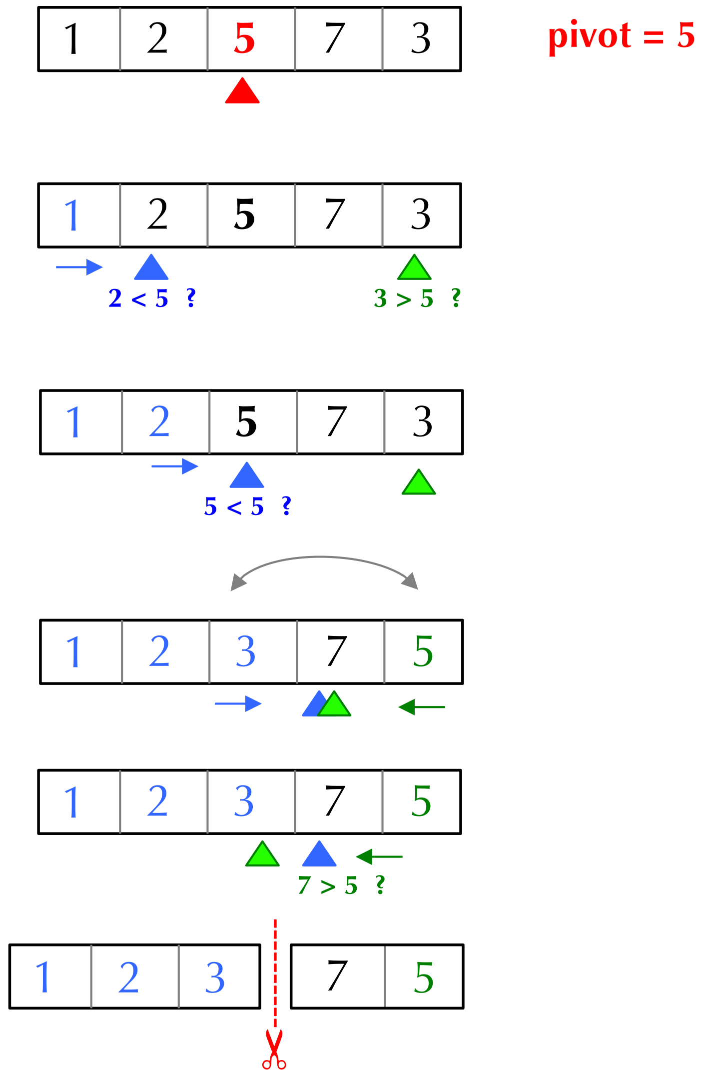
14>7 ✔ -> déplacement sur la gauche

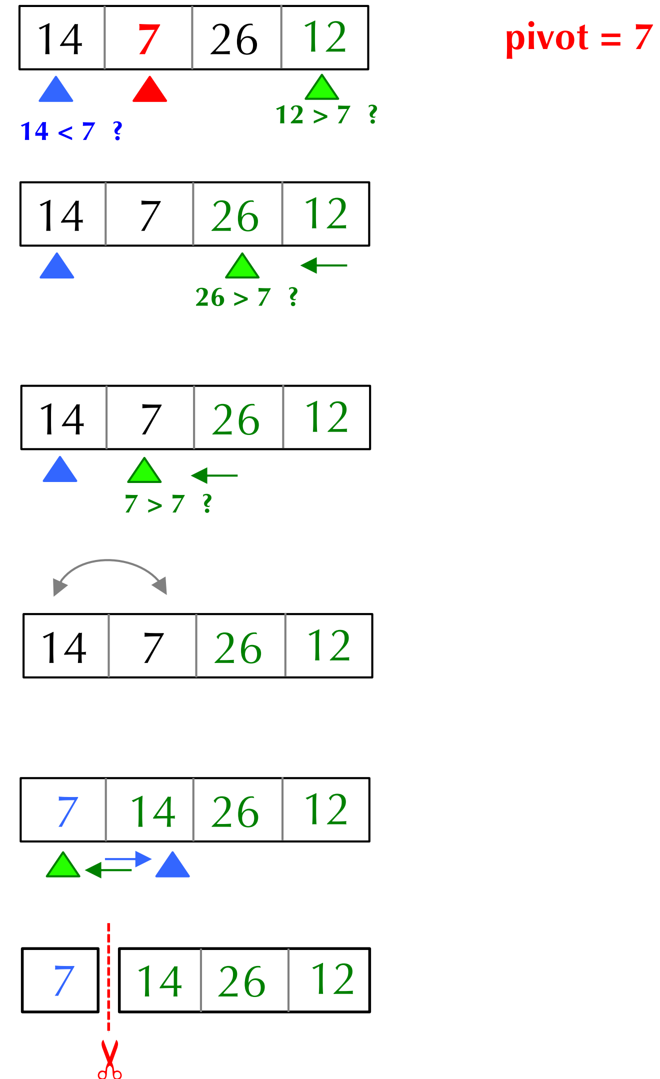
gauche **>** droite -> partition

**quicksort** ( partie gauche )

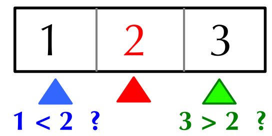
**quicksort** ( partie droite )

# Tri rapide - *quicksort*

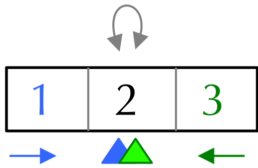


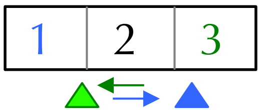



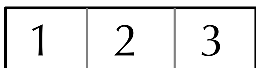
# Tri rapide - *quicksort*

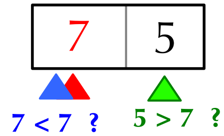



pivot = 2

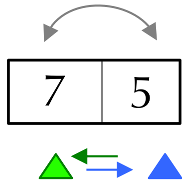


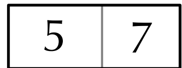



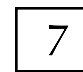


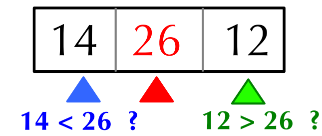




pivot = 7

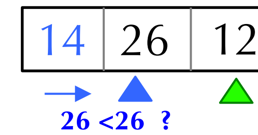


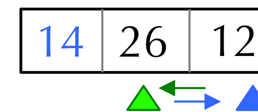



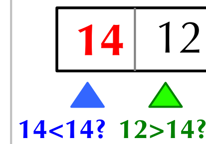


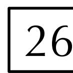




pivot = 26

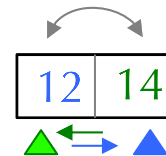


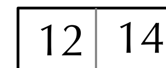



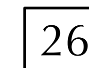




pivot = 14



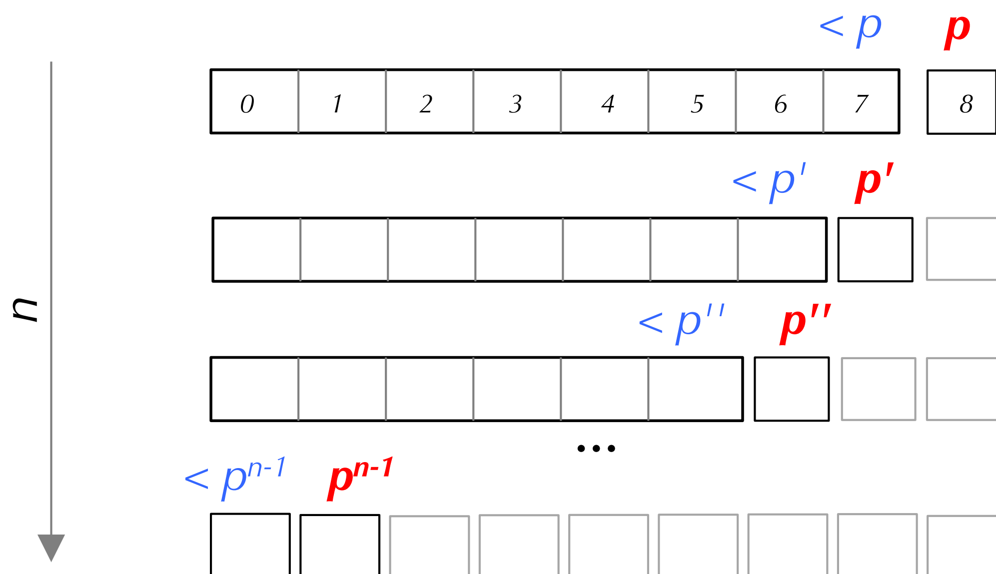
# Tri rapide - *quicksort*

## Cas défavorable pour *quicksort*

Il est possible de choisir le pivot ***p***, de telle manière que la partition des données à trier ne soit plus équilibrée.

Ex.

Considérons que le tableau d'entrée est déjà trié de manière croissante. Choisir ***p*** comme étant la valeur à l'extrême droite du tableau de telle manière qu'elle soit toujours la plus grande valeur du tableau à trier. La partition du tableau sera plus équilibrée.




Complexité: $O(n^2)$

# Tri rapide - *quicksort*

```
void QuickSort(int arr[], int left, int right) {
   int g = left, d = right;
   int tmp;
   int pivot = arr[(left + right)/2];  // choisit le pivot au centre
   while (g <= d) {                    // tant que les curseurs g et d ne se croisent pas
      while (arr[g] < pivot)           // deplace le curseur gauche sur la droite
              g++;                     //  tant que la valeur sous le curseur est < pivot
      while (arr[d] > pivot) d--;      // idem avec curseur droite
            if (g <= d) {
                  tmp = arr[g]; arr[g] = arr[d]; arr[d] = tmp; // echange

                  g++;                 // deplace le curseur gauche sur la droite
                  d--;                 // deplace le curseur droite sur la gauche
            }
      }                                 // partition --
   if (left < d)  QuickSort(arr, left, d);  // applique QuickSort sur la partie gauche
   if (g < right) QuickSort(arr, g, right); // idem partie de droite
}
```

# *c's - qsort*

La librairie standard `<stdlib.h>` implémente l'algorithme quicksort au travers de la fonction `qsort()`. Elle permet de trier les éléments contenus dans `base[]`.

```
void qsort (void* base, size_t num, size_t size,
            int (*comparator) (const void*, const void*));
```

La fonction `qsort()` est à même de travailler sur différents types de données. Pour cela, l'utilisateur devra fournir un pointeur sur une fonction **`comparator()`** qui effectue les comparaisons à chaque fois que c'est nécessaire.

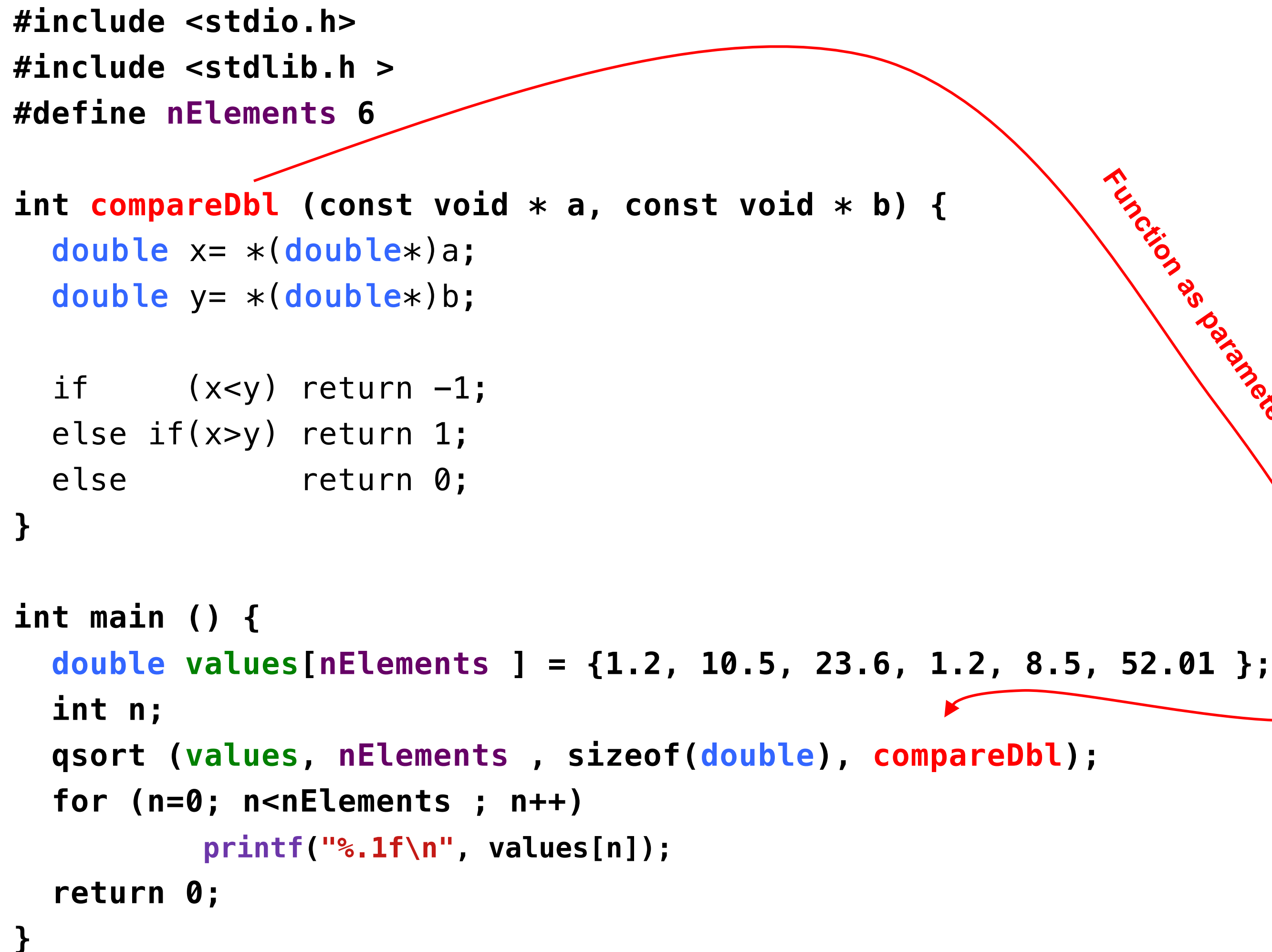
Elle retourne **-1** si a<b, **1** si a>b et **0** si a=b. Ex. pour la comparaison d'entiers

```
int comparator(const void* a, const void* b){
            return ( *(int*)a - *(int*)b );
    }
```

**`void*`** est un pointeur sur une valeur dont le type n'est pas connu. C'est un pointeur générique.

**`Num`** définit le nombre d'éléments à trier et **`size`** la taille en bytes de ces éléments.

# *qsort* – double[]



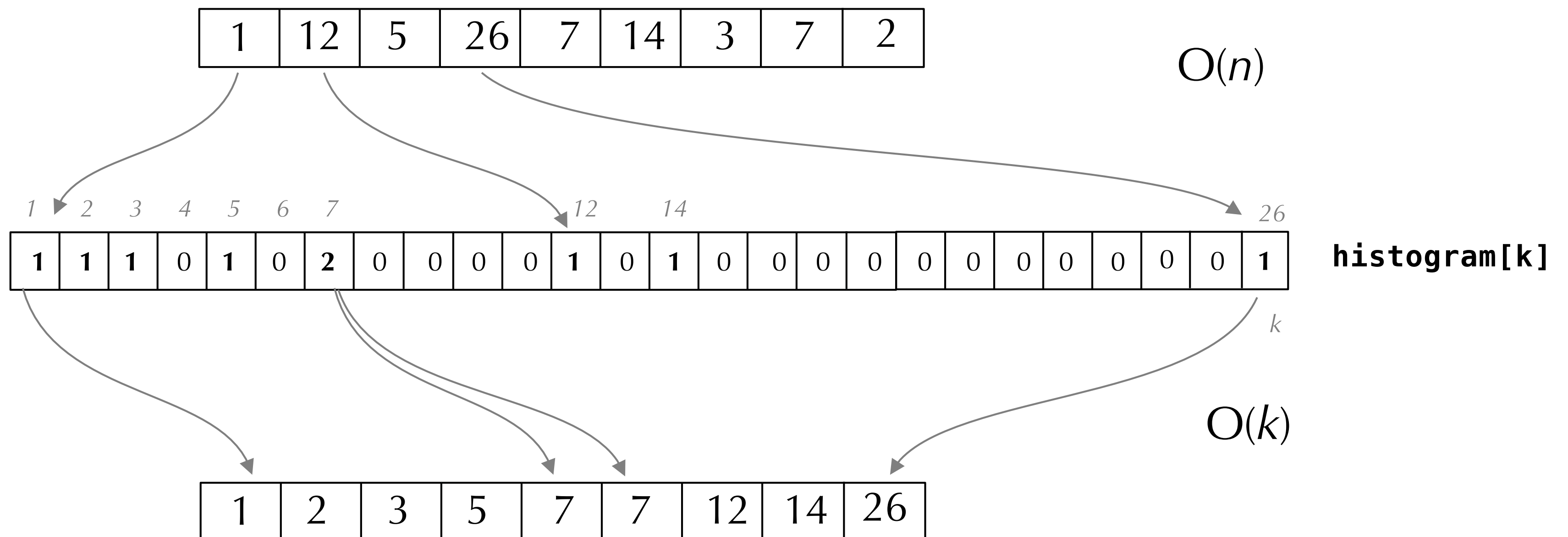

```
#include <stdio.h>
#include <stdlib.h >
#define nElements 6

int compareDbl (const void * a, const void * b) {
  double x= *(double*)a;
  double y= *(double*)b;

  if      (x<y) return -1;
  else if(x>y) return 1;
  else          return 0;
}

int main () {
  double values[nElements ] = {1.2, 10.5, 23.6, 1.2, 8.5, 52.01 };
  int n;
  qsort (values, nElements , sizeof(double), compareDbl);
  for (n=0; n<nElements ; n++)
          printf("%.1f\n", values[n]);
  return 0;
}
```

```
Cout:
  1.2,
  1.2,
  8.5,
  10.5,
  23.6,
  52.01,
```

# Counting sort

**Idée**

Ne plus trier les données par comparaison, mais les mettre directement à la *bonne position*, puis lire le nombre d'éléments par position. Cela équivaut à faire l'**histogramme** des valeurs et à l'afficher.

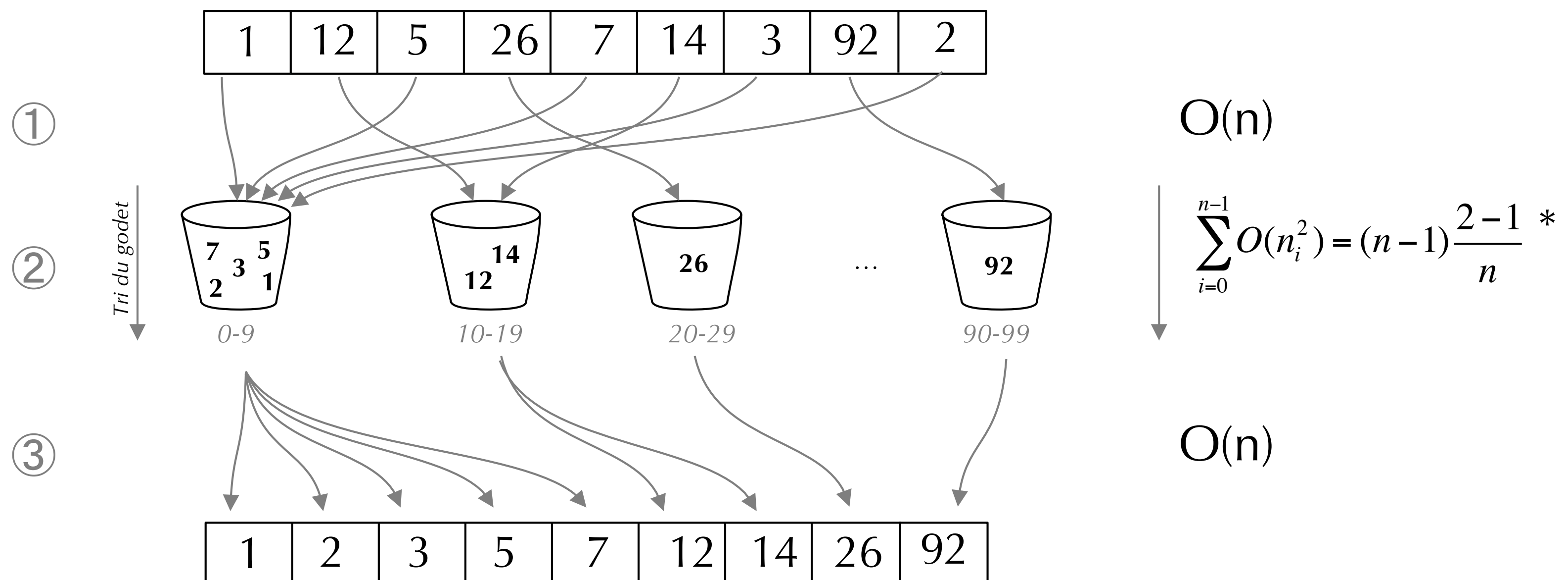



*Tri de chaines de caractères ?*

# Bucket sort

## Idée

De la même manière que pour le *counting sort,* placer les valeurs à la bonne position. Réduire le nombre de positions en les regroupant dans des godets (*buckets*). Trier ensuite chaque godet indépendamment, puis fusionner le contenu des godets. Complexité en O(n).




* voir conditions et démonstration dans Cormen et al., § 8.4

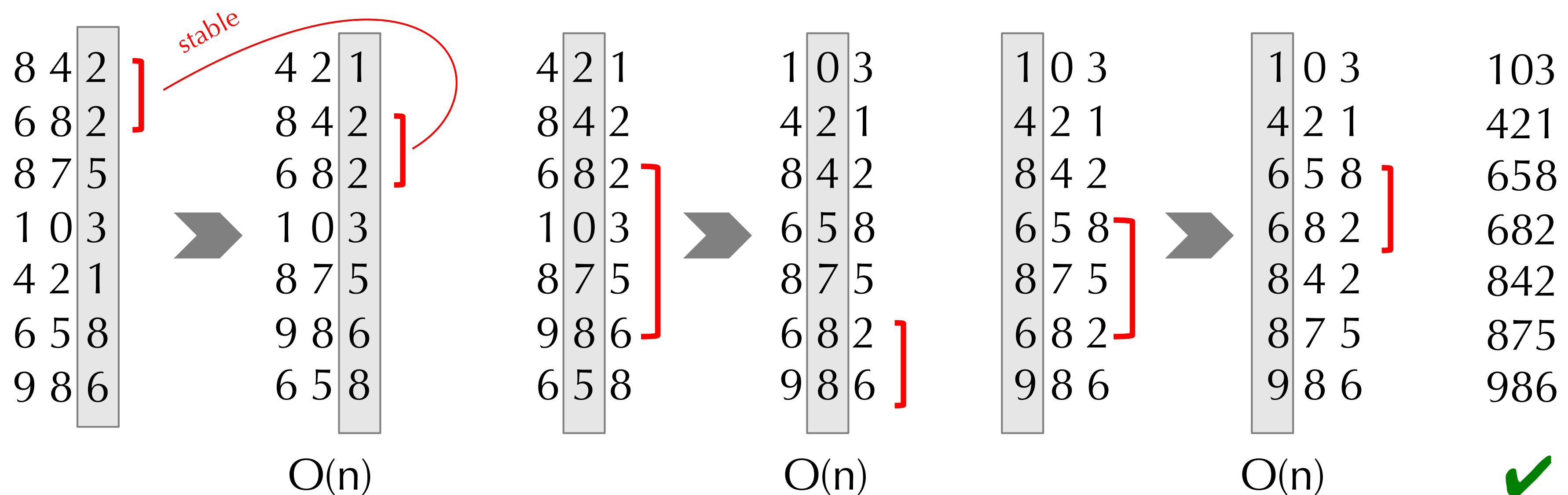
# Radix sort

## Idée

Trier *n* entiers représentés à l'aide de *d* digits. A l'aide du *counting sort,* trier les nombres en regardant uniquement les unités ($10^0$), puis trier les dizaines ($10^1$) et ainsi de suite jusqu'au $10^d$ digit.

Condition: *counting sort* doit être **stable,** i.e ordre d'entrée = ordre de sortie.

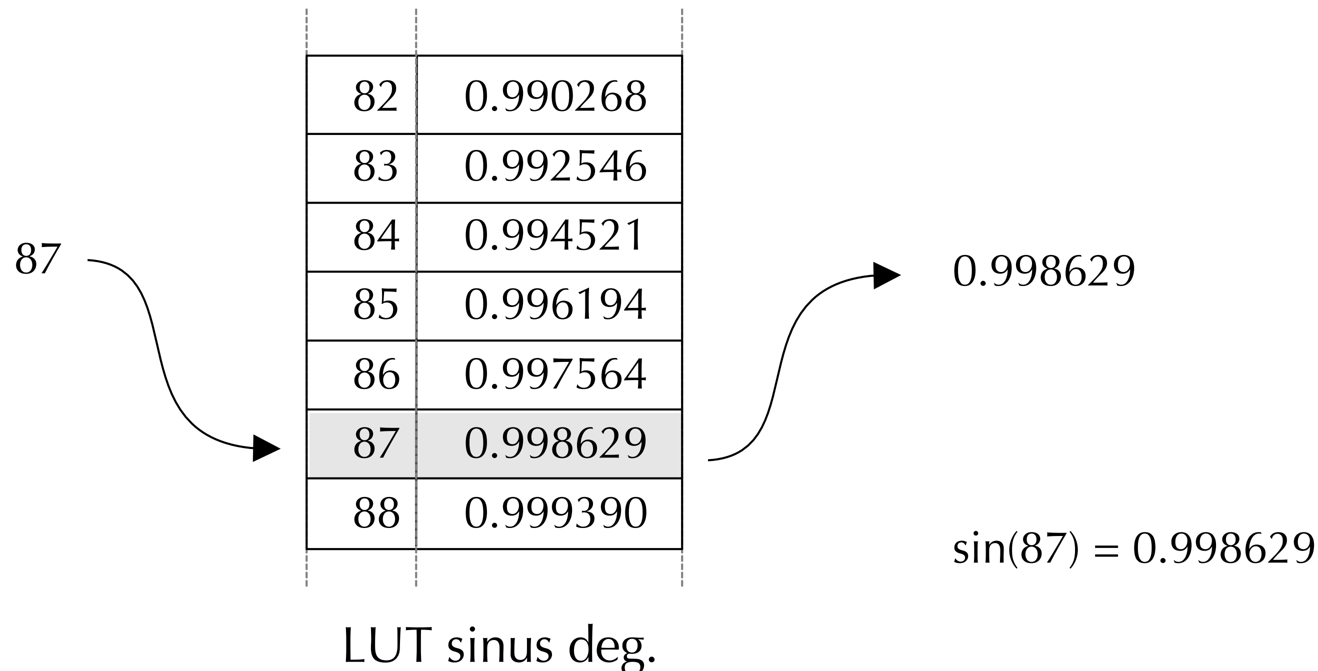
Complexité: O(*d*+*n*)



# Lookup table

Une **l**ook**u**p **t**able (LUT) ou table de correspondance permet d'associer une valeur de sortie pré-calculée à une valeur d'entrée. C'est l'équivalent des formulaires ou tables imprimés sur papier au siècle dernier. L'accès à la table est beaucoup plus rapide que le calcul de la valeur mémorisée.

Ex. table de sinus.




LUT sinus deg.

# Color Lookup table

Une **c**olor **l**ook**u**p **t**able (CLUT) permet de choisir une sélection de couleurs parmi l'ensemble des couleurs possibles. Cela permet de réduire l'espace mémoire d'une image.

Ex. *web standardized clut*, choix de 256 ($2^8$) couleurs parmi $2^{24}$ couleurs possibles.

La taille mémoire est diminuée par 3.

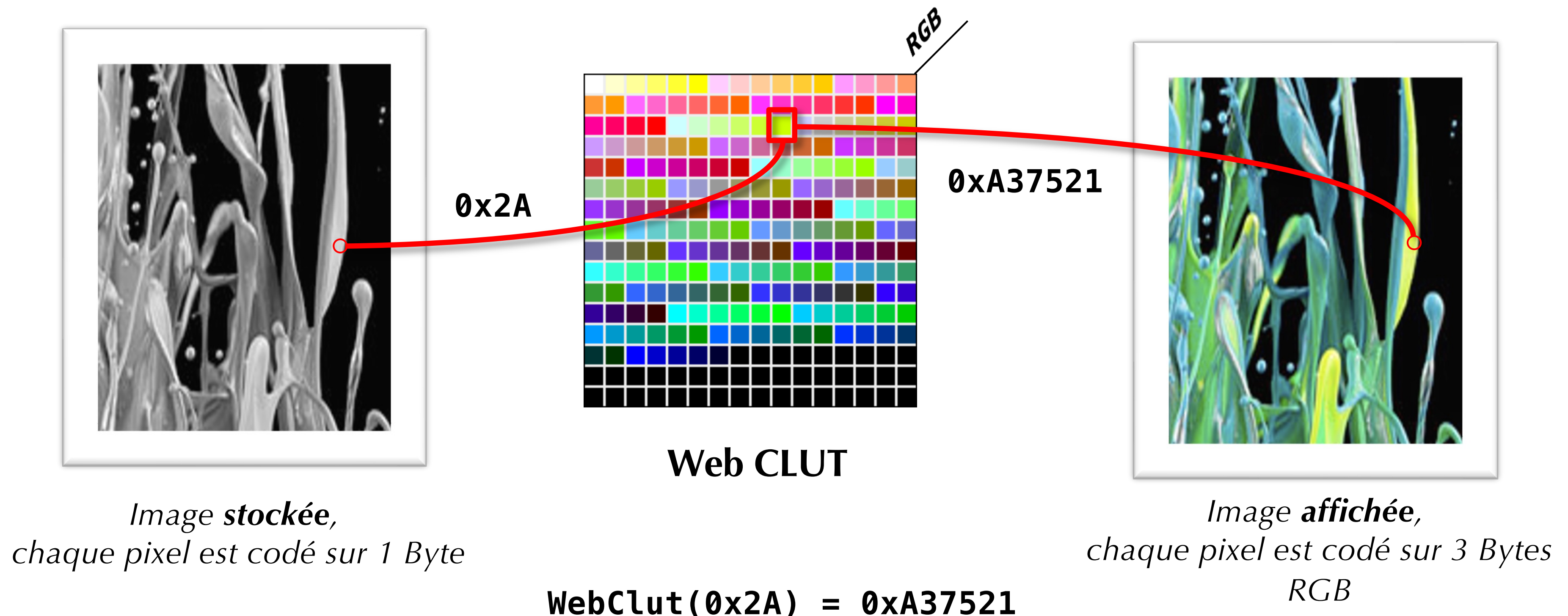



*Image **stockée**, chaque pixel est codé sur 1 Byte*

*Image **affichée**, chaque pixel est codé sur 3 Bytes RGB*

# Ce que j'ai appris

## Structure de données pour tri

- *In-place* (échange 2 à 2) -> tableau
- *Insertion* (dans une liste) -> liste chainée (pointeur)

## Tri simple

- Tri *brute force,* tri bulle, tri par insertion, tri par sélection
- Tri rapide, quicksort
- `<stdlib.h>` `qsort()`

## Tri sans comparaison

- Counting sort
- Bucket sort
- Radix sort

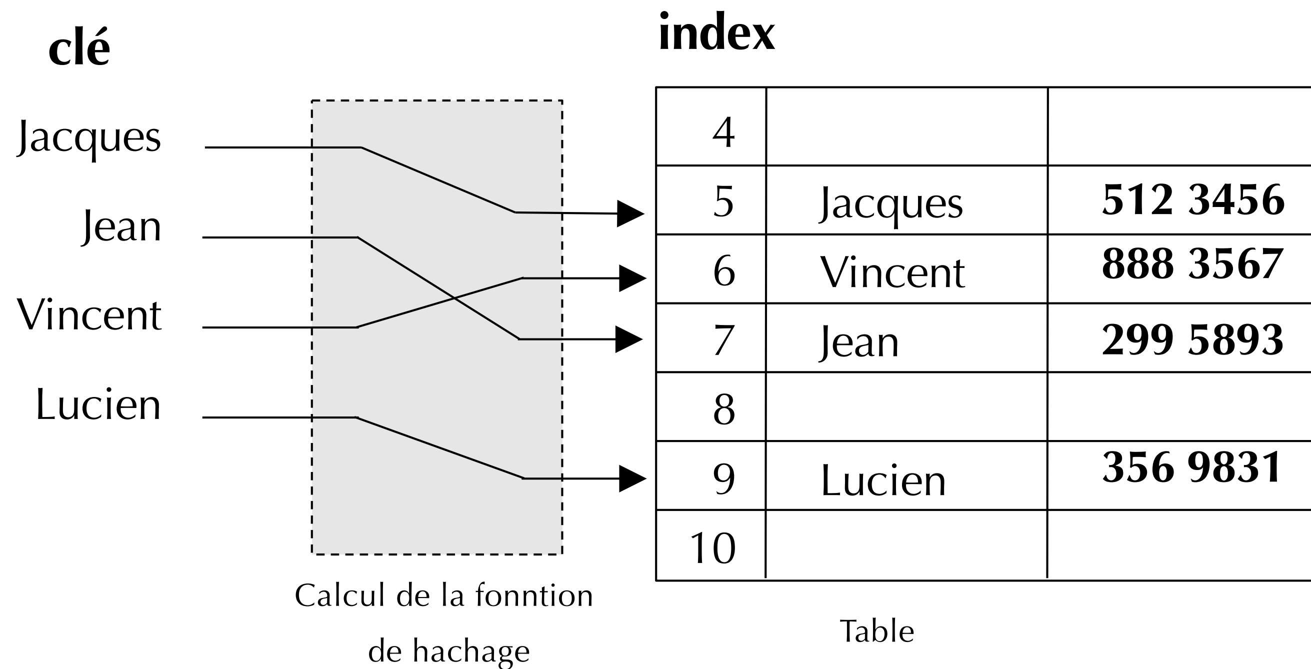
## Tables (*slides supplémentaires, pas dans le cours*)

- Lookup table
- Table de hachage

# Slides supplémetaires

# Table de hachage

Une table de hachage permet d'associer une **clé** à un élément par exemple un nom à un numéro de téléphone. Cette clé permet un accès instantané en O(1) à l'élément stocké dans la table indexée.



# Table de hachage - clé

Le calcul de la fonction de hachage *h* est primordial. Il doit permettre de générer une position unique (index) dans la table à partir de la clé.

clé -> *h*(clé) = index

Dans le cas idéal, le calcul de la fonction de hachage *h* sur l'ensemble *C* des clés possibles devrait générer un ensemble *I* d'index de taille [1..taille de C].

*h*(C) = I    de [1..taille de C]

Dans la pratique, une telle fonction de hachage *h* est très difficile, voire impossible, à trouver. Une fonction de hachage *h* réelle ne pourra pas garantir que deux clés différentes génèrent deux index différents. Il y aura une **collision**.

**collision:** pour clé1 ≠ clé2  ⇔  *h*(clé1) =  *h*(clé2)

# Table de hachage - clé

Une bonne fonction de hachage doit calculer l'index de manière rapide, elle doit également limiter le nombre de **collisions** autant que possible.

Il existe plusieurs solutions pour calculer une telle fonction, cela dépendra de notre connaissance ou non de l'ensemble des clés possibles.

Dans notre exemple, les clés sont des chaînes de caractères (noms). La fonction de hachage pourrait être:

*h*(clé) = (∑ caractères de la clé) **modulo** *taille*) +1

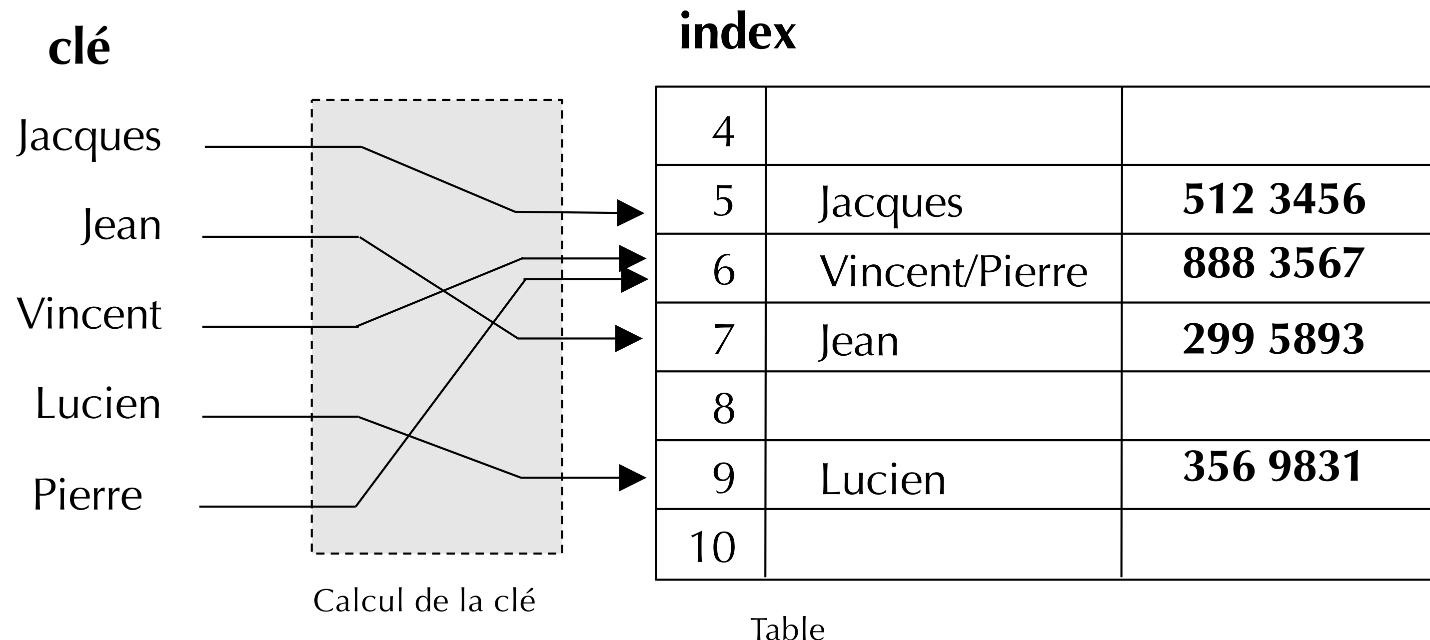
avec caractère *i* de la clé c: ascii_code(c[i])
taille: taille de la table, ~taille de C

Ex. *h*('Vincent') = ∑ (86, 105, 110, 99, 101, 110, 116) **modulo** 10) + 1 = 6
*V i n c e n t*

et *h*('Jean') = 7, *h*('Jacques') = 5, *h*('Lucien') = 9

# Table de hachage

Si nous ajoutons une nouvelle clé "Pierre", *h*('Pierre') = 6, c'est également l'index de "Vincent" => il y a collision.

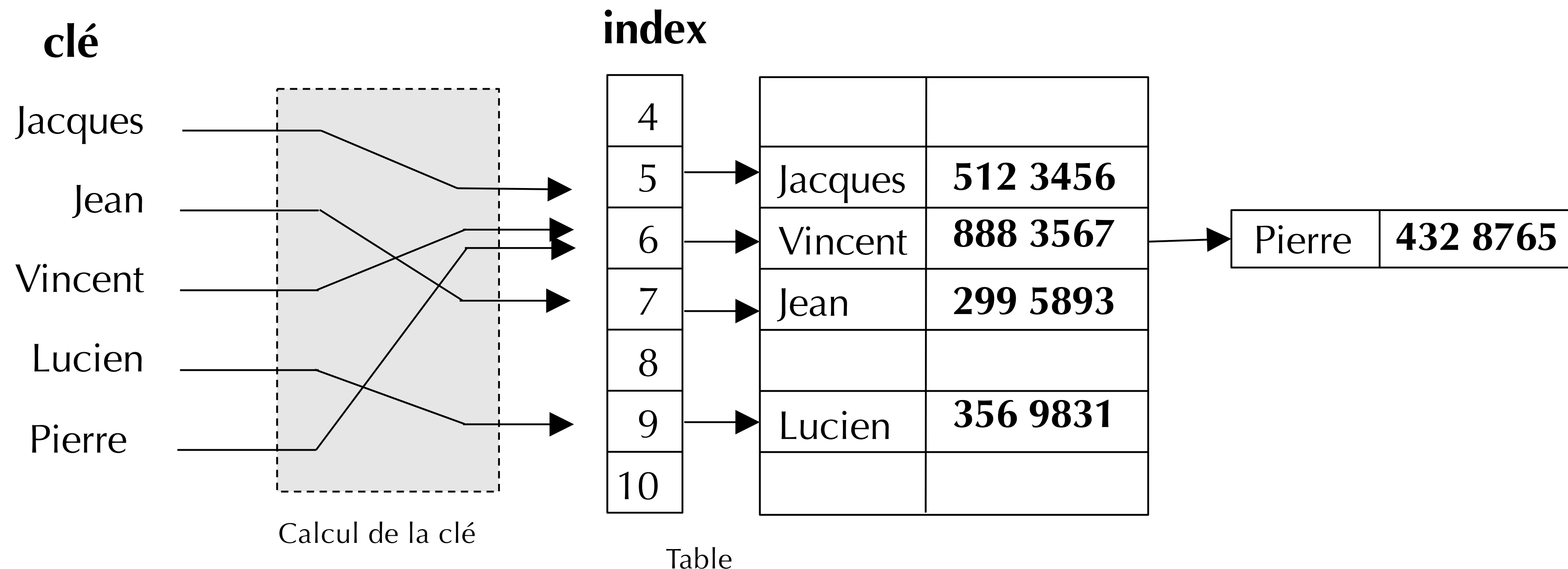



Il existe plusieurs possibilités pour résoudre cette collision:

- Changer la fonction de hachage *h*, rien ne garanti qu'il n'y aura pas de nouvelles collisions avec d'autres clés.
- Etendre la table de hachage: hachage ouvert.
- Employer une case contiguë pour stocker l'entrée en collision, alternativement effectuer un deuxième hachage à l'aide d'une autre fonction de hachage: hachage fermé.

# Hachage ouvert

Etendre la table de hachage ou hachage ouvert. La table est "ouverte" vers la droite, l'entrée unique est remplacé par une liste d'entrée ayant cet index.




Dans le cas de Pierre, le temps d'accès n'est plus forcément O(1) car il faut chercher dans la liste chainée l'élément qui nous intéresse. Dans le cas le plus défavorable: toutes les clés génèrent le même index, il y aura une recherche linéaire, c-à-d. O(n).