Tableau d'enregistrements

Regrouper plusieurs enregistrements dans un tableau est parfaitement possible!

Exemple : représenter les données concernant 50 personnes

```
algorithme monAlgorithme
     // déclaration d'un enregistrement
     enregistrement Personne
          chaine nom;
          chaine prenom;
          entier age;
          réel taille;
     finenregistrement
     ...
     Personne[50] t;
début
     // Initialisation
     t[0].nom <- "Duchmol";
     t[0].prenom <- "Robert";
     t[0].age <- 24;
     t[0].taille <- 1.80;
     ...
fin
```

Enregistrement de tableaux (1/2)

Les champs des enregistrements peuvent être de n'importe quel type, y compris de type tableau !

Exemple : on veut représenter pour chaque personne ses diplomes et ses enfants.

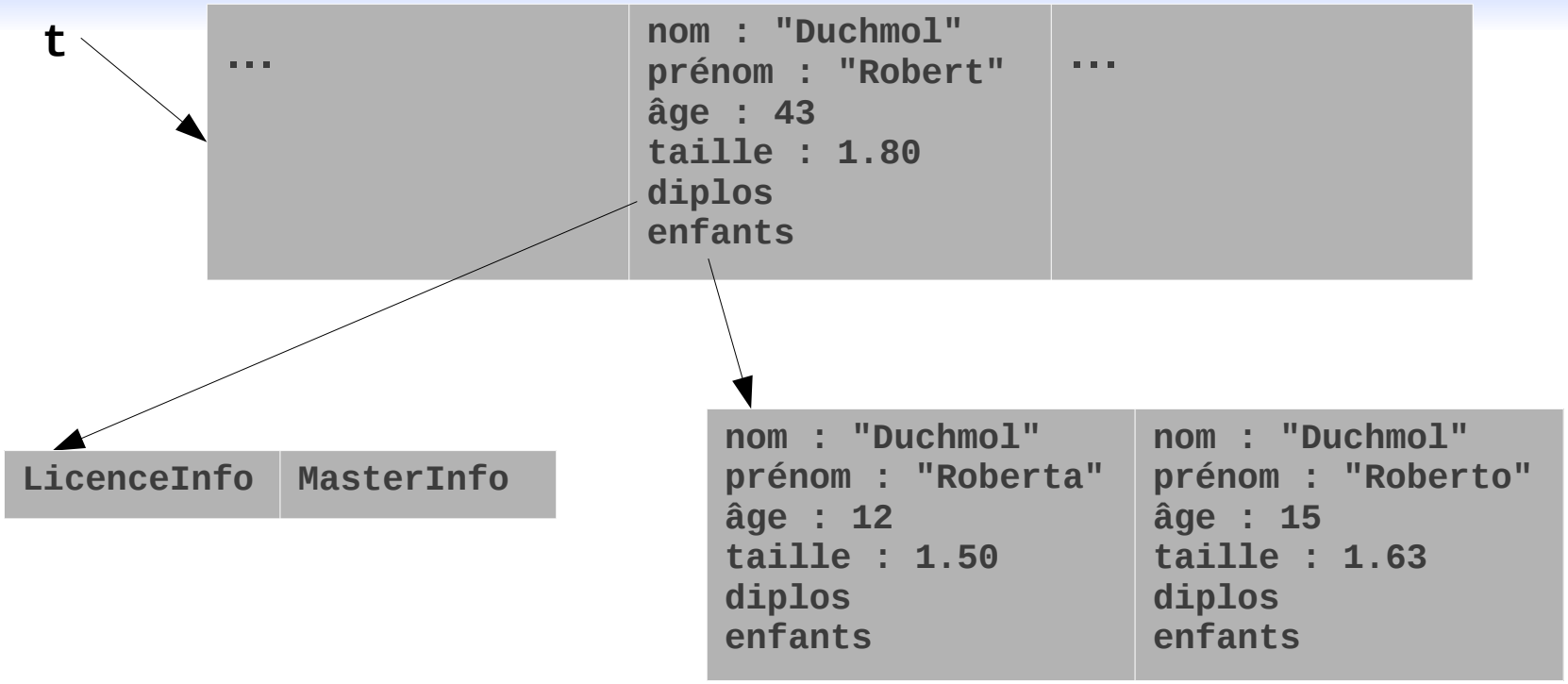
```
enum Diplome{LicenceInfo, LicenceMaths, ..., MasterInformatique, ...};

enregistrement Personne
     chaine nom;
     chaine prenom;
     entier age;
     réel taille;
     Diplome[] diplos;
     Personne[] enfants;
finenregistrement;
```

Enregistrement de tableaux (2/2)

Exemple : Robert Duchmol a une licence et un master d'informatique et deux enfants, Roberta, 12 ans et 1.50m et Roberto, 15 ans et 1.63m.

```
Personne p,e1,e2;
p.nom <- "Duchmol";
p.prenom <- "Robert";
p.age <- 43;
p.taille <- 1.80;
redim p.diplos[2];
p.diplos[0] <- Diplome.LicenceInfo;
p.diplos[1] <- Diplome.MasterInfo;
e1.nom <- "Duchmol";
e1.prenom <- "Roberta";
e1.age <- 12;
e1.taille <- 1.50;
e2.nom <- "Duchmol";
e2.prenom <- "Roberto";
e2.age <- 15;
e2.taille <- 1.63;
redim p.enfants[2];
p.enfants[0] <- e1;
p.enfants[1] <- e2;
```

Tableau d' enregistrements de tableaux




```
Personne[3] t;
... // initialisation des données
t[1].diplos[0]; // vaut LicenceInfo
t[1].enfants[0]; // désigne l'enregistrement concernant Roberta Duchmol
t[1].enfants[0].âge; // vaut 12
t[1].enfants[1].enfants; // tableau vide
```

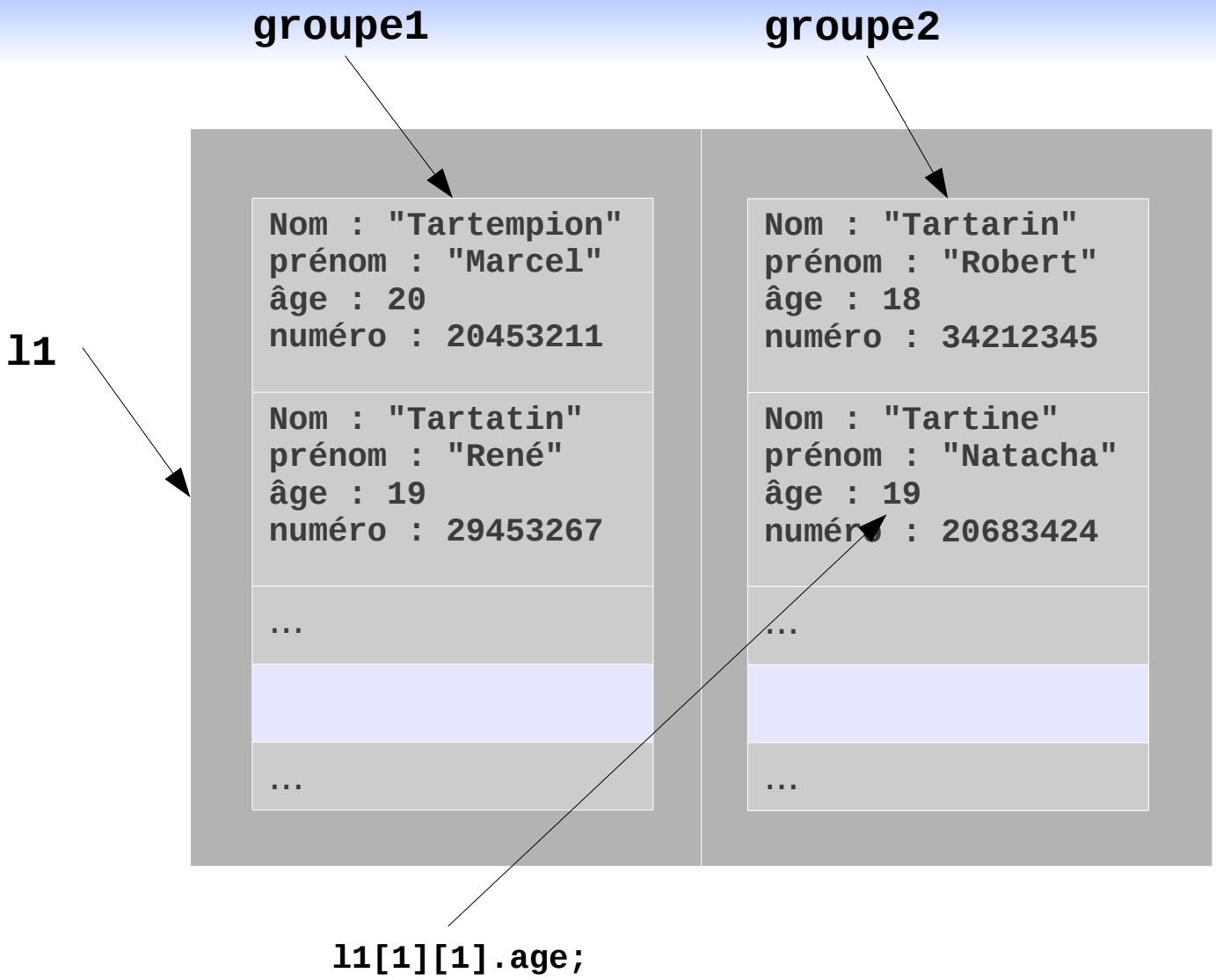
Tableau multidimensionnel (1/2)

Les éléments contenus dans un tableau peuvent être de n'importe quel type, y compris de type tableau.

Exemple : on veut représenter les étudiants de L1 info en deux groupes.

```
enregistrement Etudiant
     chaine nom;
     chaine prenom;
     entier numéro,age;
finenregistrement
Etudiant[30] groupe1;
... // initialisation de groupe1
Etudiant[28] groupe2;
... // initialisation de groupe2
Etudiant[][] l1 <- {groupe1,groupe2};
début
     écrire l1[0].longueur; // affiche 30
     écrire l1[1][27].age; // affiche l'age du 28e étudiant du groupe 2
     ...
fin
```

Tableau multidimensionnel (2/2)



Déclaration d'un tableau multidimensionnel

Déclaration de tableaux multidimensionnels en algorithmique :

```
// tableaux statiques
entier[2][3] t; // déclaration d'un tableau d'entiers de dimension 2
réel[5][34][12] toto; // déclaration d'un tableau de réels de dimension 3

// tableaux dynamiques
entier[][] t;
redim t[2][3];
réel[5][][] toto;
redim toto[][34][12];
```

Déclaration de tableaux multidimensionnels en Java :

```
int[][] t;
t = new int[2][3];
float[][][] toto = new float[5][34][12];
char[][4] truc;
truc = new char[7][];
```

Initialisation d'un tableau multidimensionnel

Initialisation d'un tableau multidimensionnel par énumération :

```
algorithme JeuDeMorpion1
     chaine[][] morpion <- {{"","",""},{"","",""},{"","",""}};
début
     ...
     morpion[1][1] <- "croix";
     morpion[2][2] <- "rond";
     ...
fin
```

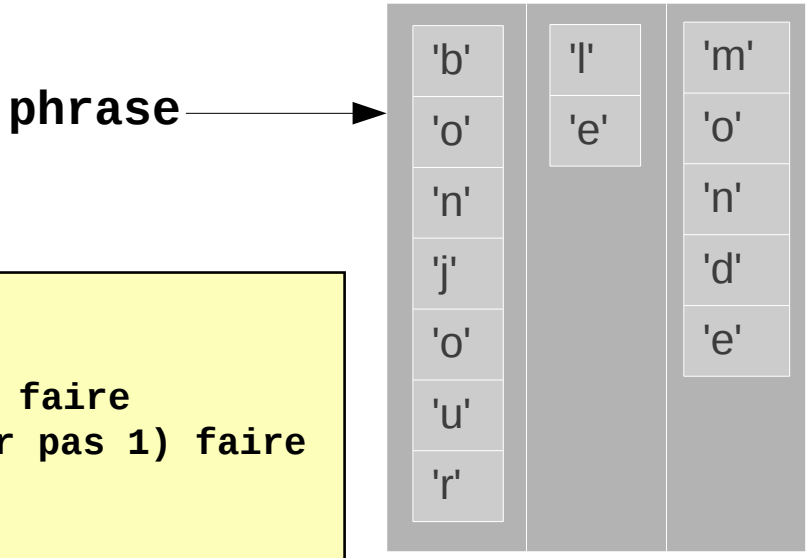
Initialisation d'un tableau multidimensionnel par boucles imbriquées :

```
algorithme JeuDeMorpion2
     chaine[3][3] morpion;
     entier i,j;
début
     pour (i allant de 0 à morpion.longueur pas 1) faire
          pour (j allant de 0 à morpion[i].longueur pas 1) faire
               morpion[i][j] <- "";
          finpour
     finpour
     ...
fin
```

Initialisation d'un tableau multidimensionnel

Attention : les tableaux initialisés par énumération ne contiennent pas forcément des tableaux de la même taille.

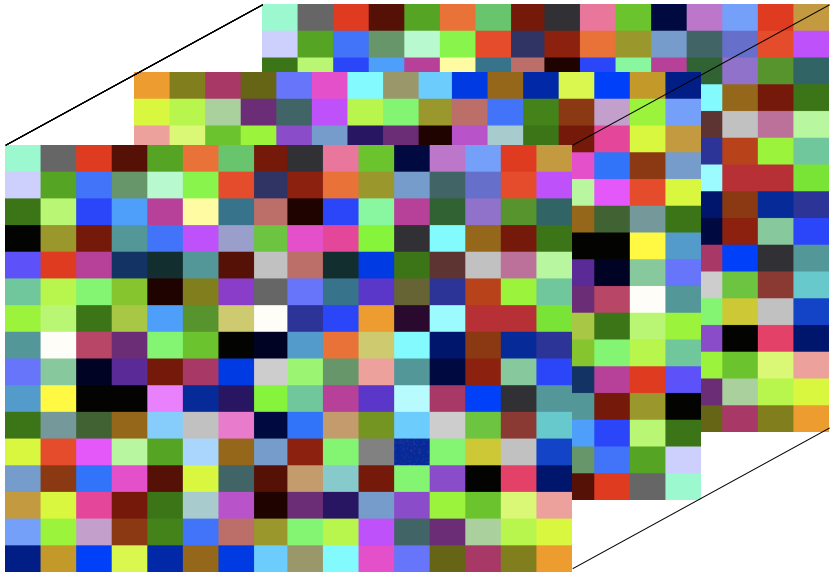
```
caractères[][] phrase <- {{'b','o','n','j','o','u','r'},
                          {'l','e'},
                          {'m','o','n','d','e'}};
```




```
écrire phrase[1][3];
entier i,j;
pour (i allant de 0 à phrase.longueur pas 1) faire
     pour (j allant de 0 à phrase[i].longueur pas 1) faire
          écrire phrase[i][j];
     finpour
finpour
```

Tableau de dimension supérieure à 2

Exemple : dans un logiciel d'infographie, on veut représenter des images organisées en calques.



```
enum Couleur {rouge, gris, bleu, kaki, violet, bleu_clair, ...};

Couleur[3][16][16] image;
image[0][0][0] <- Couleur.bleu_clair;
image[0][1][0] <- Couleur.gris;
...
```

Structuration des données

Il ne faut pas hésiter à structurer les données au maximum en définissant des enregistrements et des tableaux et en les combinant car cela :

- facilite l'écriture des algorithmes et des programmes
- permet de passer des paramètres complexes aux fonctions
- permet de faire retourner par les fonctions des valeurs multiples

Exemple : écrire une fonction qui prend en paramètres toutes les données concernant des personnes et renvoie les noms des personnes majeures.

```
fonction avec retour chaine[] majeures(Personne[] t)
...
```

Problème : dans les structures de données, retrouver une valeur nécessite de parcourir la structure. Il faut trouver des méthodes pour réaliser ces recherches efficacement.

Recherche dans un tableau

Première idée : recherche par parcours séquentiel. On parcourt le tableau case par case jusqu'à ce qu'on trouve l'élément.

Exemple : recherche séquentielle dans un tableau de chaines.

```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
fonction avec retour booléen rechercheElement1(chaine[] tab, chaine e)
    entier i;
    booléen trouve;
début
    i <- 0;
    trouve <- faux;
    tantque (i < tab.longueur et non trouve) faire
        si (tab[i] = e) alors
            trouve <- vrai;
        sinon
            i <- i + 1;
        finsi
    fintantque
    retourne trouve;
fin
```

Recherche dans un tableau multidimensionnel

```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
fonction avec retour booléen rechercheElement2(chaine[][] tab, chaine e)
    entier i,j;
    booléen trouve;
Début
    trouve <- faux;
    i <- 0;
    tantque (i < tab.longueur et non trouve) faire
        j <- 0;
        tantque (j < tab[i].longueur et non trouve) faire
            si (tab[i][j] = e) alors
                trouve <- vrai;
            sinon
                j <- j + 1;
            finsi
        fintantque
        i <- i + 1;
    fintantque
    retourne trouve;
fin
```

Recherche des occurences

Si on veut trouver le nombre d'occurences de la valeur recherchée, il faut parcourir toute la structure et ne pas s'arrêter dès qu'on trouve la valeur.

```
// cette fonction renvoie le nombre de fois où x est présente dans tab
fonction avec retour entier rechercheElement3(chaine[] tab, chaine e)
  entier i, nbOc;
début
  i <- 0;
  nbOc <- 0;
  tantque (i < tab.longueur) faire
      si (tab[i] == e) alors
          nbOc <- nbOc + 1;
      finsi
      i <- i + 1;
  fintantque
  retourne nbOc;
fin
```

Recherche dans un tableau trié (1/2)

Si le tableau est trié par ordre croissant (respectivement décroissant), on peut s'arrêter dès que la valeur rencontrée est plus grande (respectivement plus petit) que la valeur cherchée.

Exemple : chercher le caractère 'l' dans le tableau suivant

'a'	'c'	'c'	'd'	'f'	'm'	'n'	'p'	'v'	'x'

Remarque : cette optimisation n'est efficace que si on rencontre « assez vite » un élément plus grand que celui recherché.

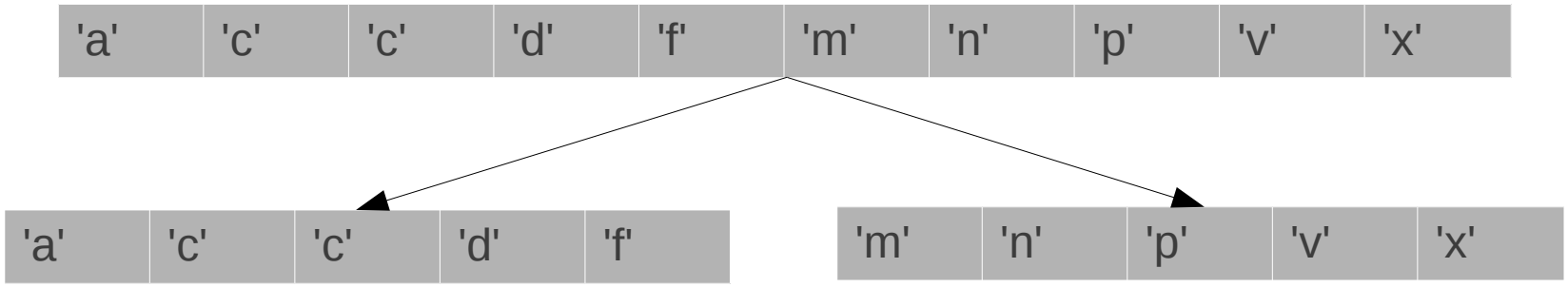
Recherche dans un tableau trié (2/2)

```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
// le tableau tab est supposé trié par ordre croissant
fonction avec retour booléen rechercheElement4(chaine[] tab, chaine e)
    entier i;
    booléen trouve,possible;
début
    i <- 0;
    trouve <- faux;
    possible <- vrai;
    tantque (i < tab.longueur et possible et non trouve) faire
        si (tab[i] = e) alors
            trouve <- vrai;
        sinon
            si (tab[i] > x) alors
                 possible <- faux;
            sinon
                i <- i + 1;
            finsi
        finsi
    fintantque
    retourne trouve;
fin
```

Recherche dichotomique (1/4)

Principe *divide ut imperes* (diviser pour régner) : on découpe les données en différentes parties qu'on traite séparément.

Exemple : pour rechercher un élément dans un tableau, on le coupe en deux et on chercher dans chaque partie.



Premier cas où ce principe est utile : si on peut lancer des programmes en parallèle, on mettra environ deux fois moins de temps pour rechercher l'élément.

Deuxième cas où ce principe est utile : si le tableau est trié, on peut ne chercher l'élément que dans une seule des parties.

Recherche dichotomique (2/4)

Principe de la recherche par dichotomie : supposons que le tableau est trié par ordre croissant.

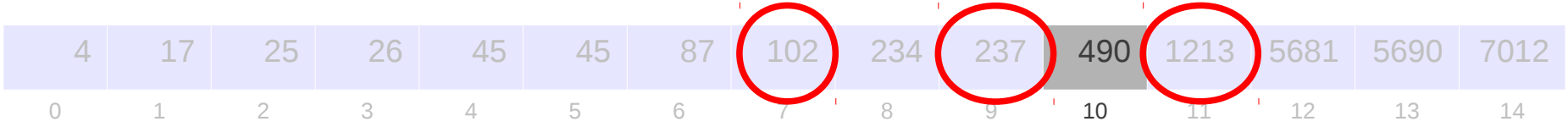
On regarde l'élément situé au milieu du tableau :

1- s'il s'agit de l'élément recherché, c'est gagné.

2- s'il est plus grand que l'élément recherché, on cherche dans la moitié gauche

3- s'il est plus petit que l'élément recherché, on cherche dans la moitié droite

Exemple : rechercher 490 dans le tableau d'entiers suivant.



Recherche dichotomique (3/4)

On doit savoir à chaque itération sur quel intervalle d'indices on travaille. Ici, on cherche l'élément entre les indice i et j. On s'arrête quand l'intervalle de recherche est vide.

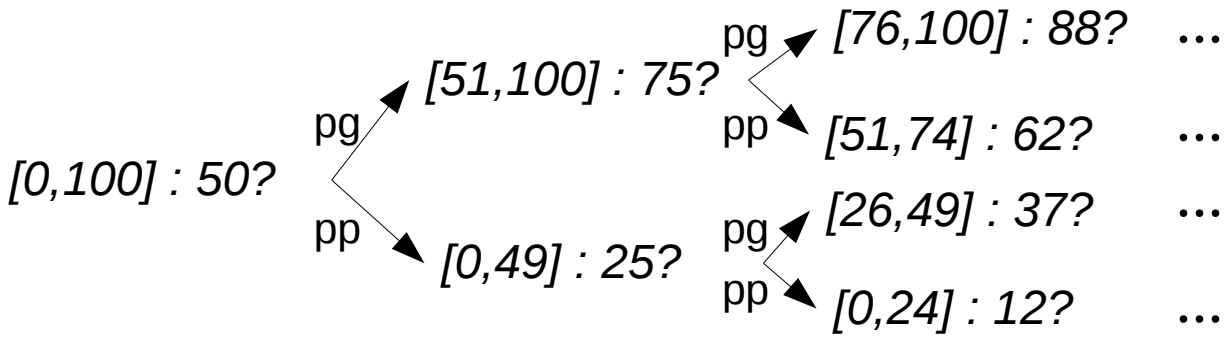
```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
// le tableau tab est supposé trié par ordre croissant
fonction avec retour booléen rechercheElement3(chaine[] tab, chaine e)
    entier i, j;
    booléen trouve;
début
    trouve <- false;
    i <- 0;
    j <- tab.longueur-1;
    tantque (i <= j et non trouve) faire
        si (tab[(j+i)/2] = e) alors
            trouve <- vrai;
        sinon
            si (tab[(j+i)/2] > x) alors
                j <- (j+i)/2 - 1;
            sinon
                i <- (j+i)/2 + 1;
            finsi
        finsi
    fintantque
    retourne trouve;
fin
```

Recherche dichotomique (4/4)

Exemple : rechercher 490 dans le tableau d'entiers suivant.

Le principe de dichotomie peut être utilisé dès qu'on a un intérêt à découper en deux un espace de recherche.

Exemple : le jeu «plus grand, plus petit». Il s'agit de trouver un nombre choisi entre 0 et 100.



Recherche par interpolation (1/3)

Principe de la recherche par interpolation : on améliore la recherche par dichotomie en coupant le tableau non pas au milieu, mais à un endroit «proche» de la valeur cherchée.

La recherche par interpolation suppose qu'on puisse interpoler l'indice probable de la valeur recherchée.

Exemple : chercher un mot dans le dictionnaire se fait par interpolation, car on est capable d'estimer, plus ou moins précisément, la page où le mot peut se trouver.

Interpoler l'indice où on peut trouver un élément dans un tableau implique qu'on connaisse à l'avance la répartition des éléments (au moins approximativement).

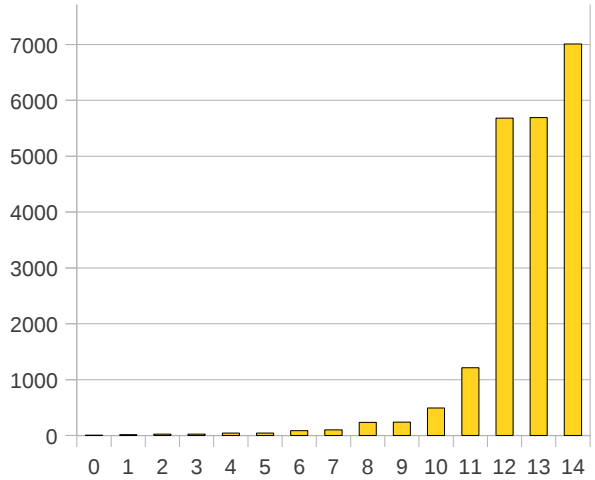
Un cas simple : les éléments sont répartis linéairement dans le tableau.

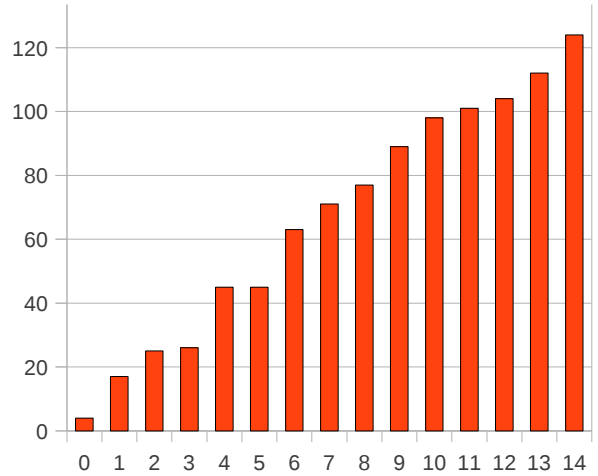
Recherche par interpolation (2/3)

Dans le tableau jaune, l'interpolation linéaire donne 1 comme indice probable de 490 (l'indice réel est 11).

Dans le tableau rouge, l'interpolation linéaire donne 8 comme indice probable de 71 (l'indice réel est 8).

4	17	25	26	45	45	87	102	234	237	490	1213	5681	5690	7012
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

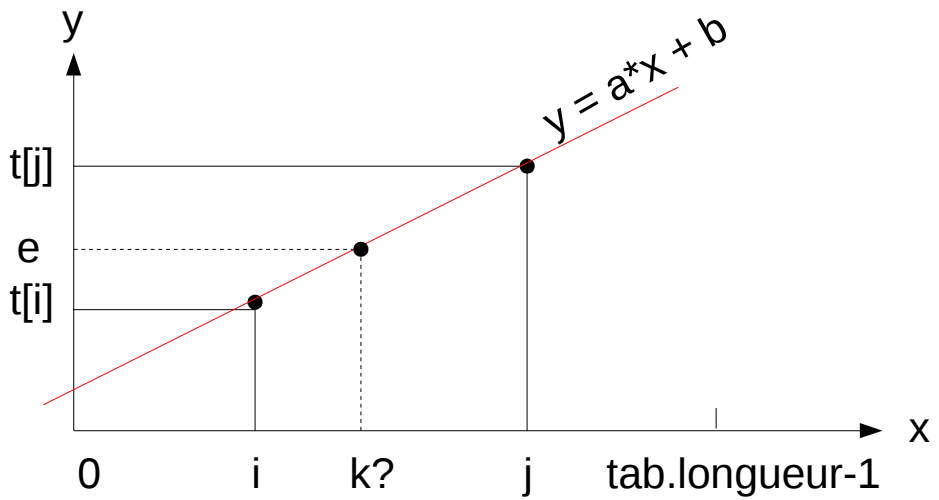




4	17	25	26	45	45	63	71	77	89	98	101	104	112	124
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Recherche par interpolation (3/3)

Si on suppose une répartition ordonnée et linéaire des valeurs dans un tableau tab, on peut interpoler linéairement la place d'un élément e recherché entre des indices i et j.



a = (tab[j] – tab[i]) / (j – i)
b = tab[i] - a*i

k = (e – b) / a

L'algorithme de recherche par interpolation est le même que celui de recherche dichotomique, mais à chaque itération, au lieu de tester l'élément situé à l'indice (i+j)/2, on teste celui situé en k.

Tableau trié (1/2)

La recherche d'élément est plus efficace dans les tableaux triés. Il est donc intéressant de pouvoir tester si un tableau est trié et, si ce n'est pas le cas, de pouvoir le trier.

Tester si un tableau est trié par ordre croissant : on s'assure que chaque case (sauf la dernière) a un contenu plus petit que celui de la case suivante.

```
fonction avec retour booléen testTrie1(int[] tab)
     entier i;
     booléen b;
début
     b <- VRAI;
     pour (i allant de 0 à tab.longueur-2 pas 1) faire
          si (tab[i] > tab[i+1]) alors
               b <- FAUX;
          finsi
     finpour
     retourne b;
fin
```

Tableau trié (2/2)

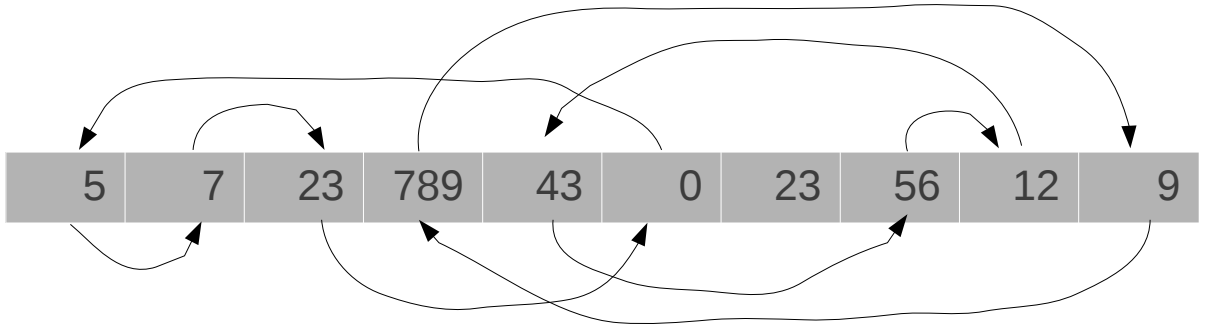
On peut améliorer l'algorithme précédent en s'arrêtant dès qu'on trouve deux éléments qui ne sont pas dans le bon ordre.

```
fonction avec retour booléen testTrie1(int[] tab)
     entier i;
     booléen b;
début
     b <- VRAI;
     i <- 0;
     tantque (i < tab.longueur-1 et b) faire
          si (tab[i] > tab[i+1]) alors
               b <- FAUX;
          finsi
          i <- i+1;
     finpour
     retourne b;
fin
```

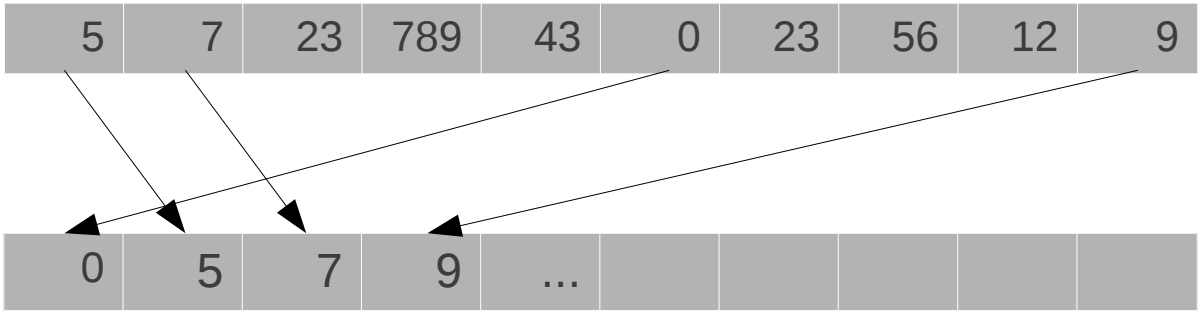
Tri de tableaux (1/2)

Occupation mémoire d'un algorithme de tri :

- tri en place : les éléments sont triés dans la structure de données



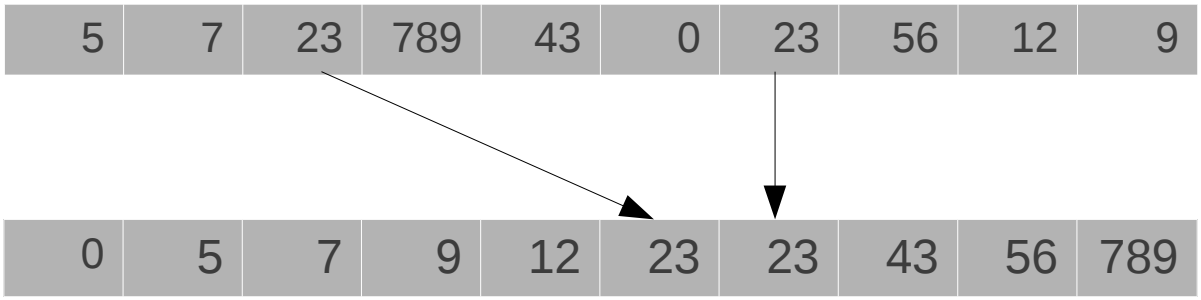
- tri pas en place : on crée une nouvelle structure de données



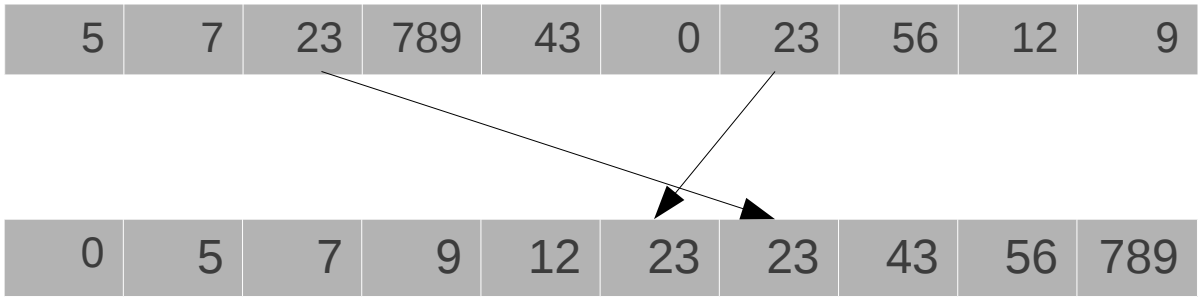
Tri de tableaux (2/2)

Stabilité d'un algorithme de tri :

- tri stable : l'algorithme ne modifie pas l'ordre initial des éléments égaux



- tri instable : l'algorithme modifie l'ordre initial des éléments égaux



Tri à bulle (1/4)

Principe du tri à bulle (bubble sort) : on parcourt le tableau et on fait remonter le plus grand (ou le plus petit) élément à un bout par permutations et on recommence jusqu'à ce que le tableau soit trié.

Exemple : trier par ordre croissant le tableau suivant

701	17	2	268	415	45	45	102

1- on fait passer le plus grand élément au bout par permutations successives

17	2	268	415	45	45	102	701

2- on recommence sur le tableau moins le dernier élément

2	17	268	45	45	102	415	701

Tri à bulle (2/4)

3- on recommence sur le tableau moins les deux derniers éléments

2	17	45	45	102	268	415	701

...

7- au septième parcours, on compare juste les deux premiers éléments

2	17	45	45	102	268	415	701

Le tri à bulle est en place. Il est stable si on permute uniquement les éléments différents.

Tri à bulle (3/4)

Algorithme de tri à bulle d'un tableau d'entier, par ordre croissant :

```
fonction sans retour triBulle(entier[] tab)
     entier i,j,temp;
début
     pour (i allant de tab.longueur-2 à 1 pas -1) faire
          pour (j allant de 0 à i pas 1) faire
               si (tab[j] > tab[j+1]) alors
                    temp <- tab[j];
                    tab[j] <- tab[j+1];
                    tab[j+1] <- temp;
               finsi
          finpour
     finpour
fin
```

NB: ce tri est stable, il serait non stable si on avait mis **tab[j] >= tab[j+1]**

Tri à bulle (4/4)

Optimisation possible : on peut s'arrêter dès que le tableau est trié.

```
fonction sans retour triBulle(entier[] tab)
     entier i,j,temp;
     booléen trié;
début
     trié <- faux;
     i <- tab.longueur-2;
     tantque ((non trié) et (i > 0)) faire
          trié <- vrai;
          pour (j allant de 0 à i pas 1) faire
               si (tab[j] > tab[j+1]) alors
                    temp <- tab[j];
                    tab[j] <- tab[j+1];
                    tab[j+1] <- temp;
                    trié <- faux;
               finsi
          finpour
          i <- i-1;
     fintantque
fin
```

Cette amélioration n'est efficace que si le tableau est déjà « un peu » trié et qu'il devient trié « assez vite ».

Tri à bulle boustrophédon (1/2)

Une autre amélioration possible du tri bulle est le tri Boustrophedon (du nom des écritures qui changent de sens à chaque ligne) :

1- on parcourt le tableau de gauche à droite, en faisant remonter l'élément le plus grand

2- on redescend le tableau de droite à gauche, en faisant descendre le plus petit

3- on recommence en ignorant les 2 éléments déjà triés

Le tri Boustrophedon est plus efficace car il permet de faire descendre plus rapidement les petits éléments en bas du tableau.

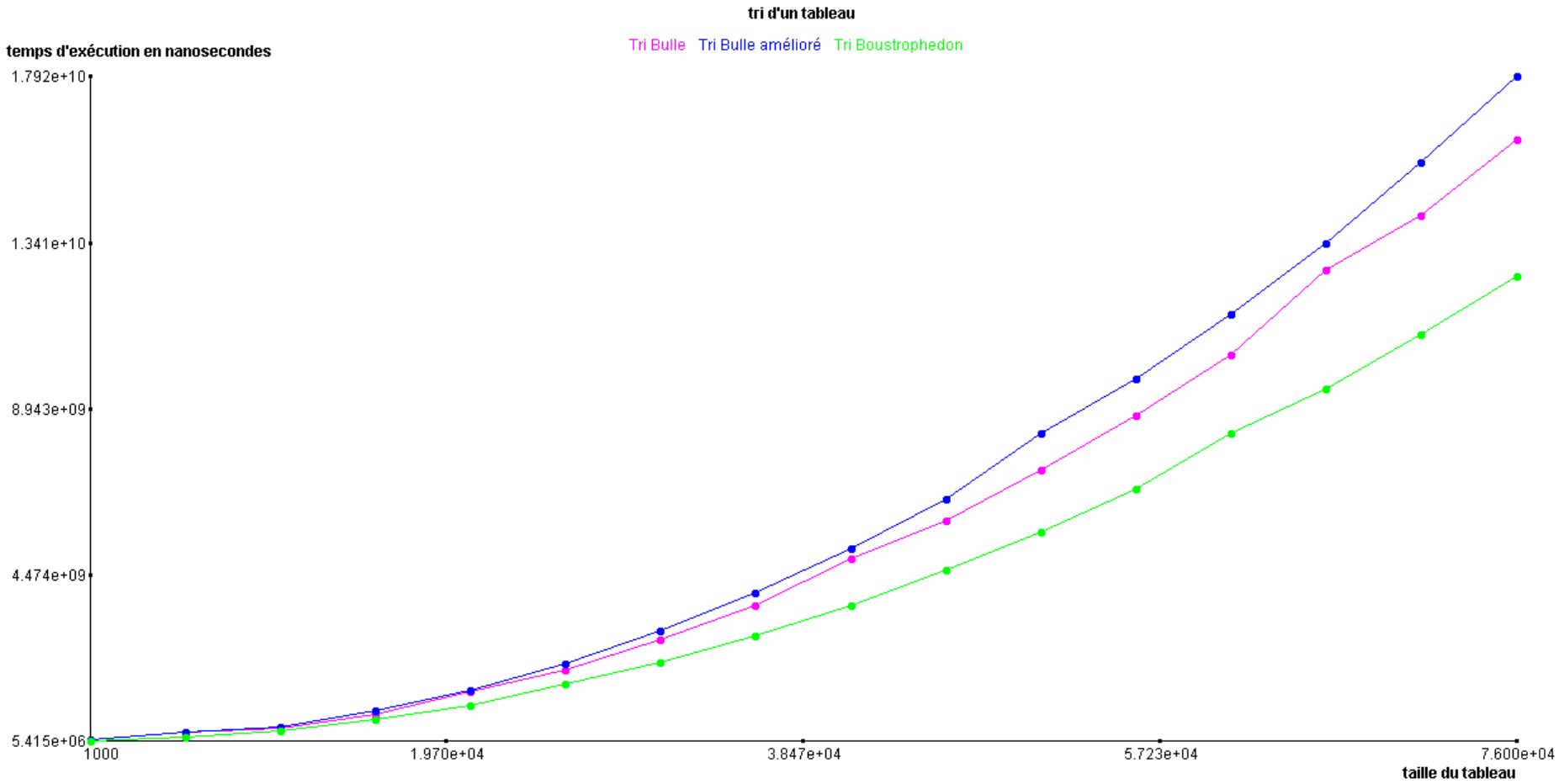
Tri à bulle boustrophédon (2/2)

On travaille entre les indices i et j : tous les éléments d'indice inférieur ou égal à i ou supérieur ou égal à j sont déjà triés.

```
fonction sans retour triBulleBoustrophédon(entier[] tab)
     entier i,j,k,temp;
     booléen trié;
début
     trié <- faux; j <- tab.longueur; i <- -1;
     tantque ((non trié) et (j > i)) faire
          trié <- vrai;
          pour (k allant de j+1 à i-2 pas 1) faire
               si (tab[k] > tab[k+1]) alors
                    temp <- tab[k]; tab[k] <- tab[k+1]; tab[k+1] <- temp;
                    trié <- faux;
               finsi
          finpour
          i <- i-1;
          pour (k allant de i-1 à j+2 pas -1) faire
               si (tab[k] < tab[k-1]) alors
                    temp <- tab[k]; tab[k] <- tab[k-1]; tab[k-1] <- temp;
                    trié <- faux;
               finsi
          finpour
          j <- j+1;
     fintantque
fin
```

Comparaison expérimentale des tris à bulle

Test sur des tableaux d'entiers générés aléatoirement.



Tri par sélection (1/2)

Principe du tri par sélection : on parcourt le tableau pour trouver le plus grand élément, et une fois arrivé au bout du tableau on y place cet élément par permutation. Puis on recommence sur le reste du tableau.

```
fonction sans retour triSelection(entier[] tab)
     entier i,j,temp,pg;
debut
     i <- tab.longueur-1;
     tantque (i > 0) faire
          pg <- 0;
          pour (j allant de 0 à i pas 1) faire
               si (tab[j] > tab[pg]) alors
                    pg <- j;
               finsi
          finpour
          temp <- tab[pg];
          tab[pg] <- tab[i];
          tab[i] <- temp;
          i <- i-1;
     fintantque
fin
```

Tri par sélection (2/2)

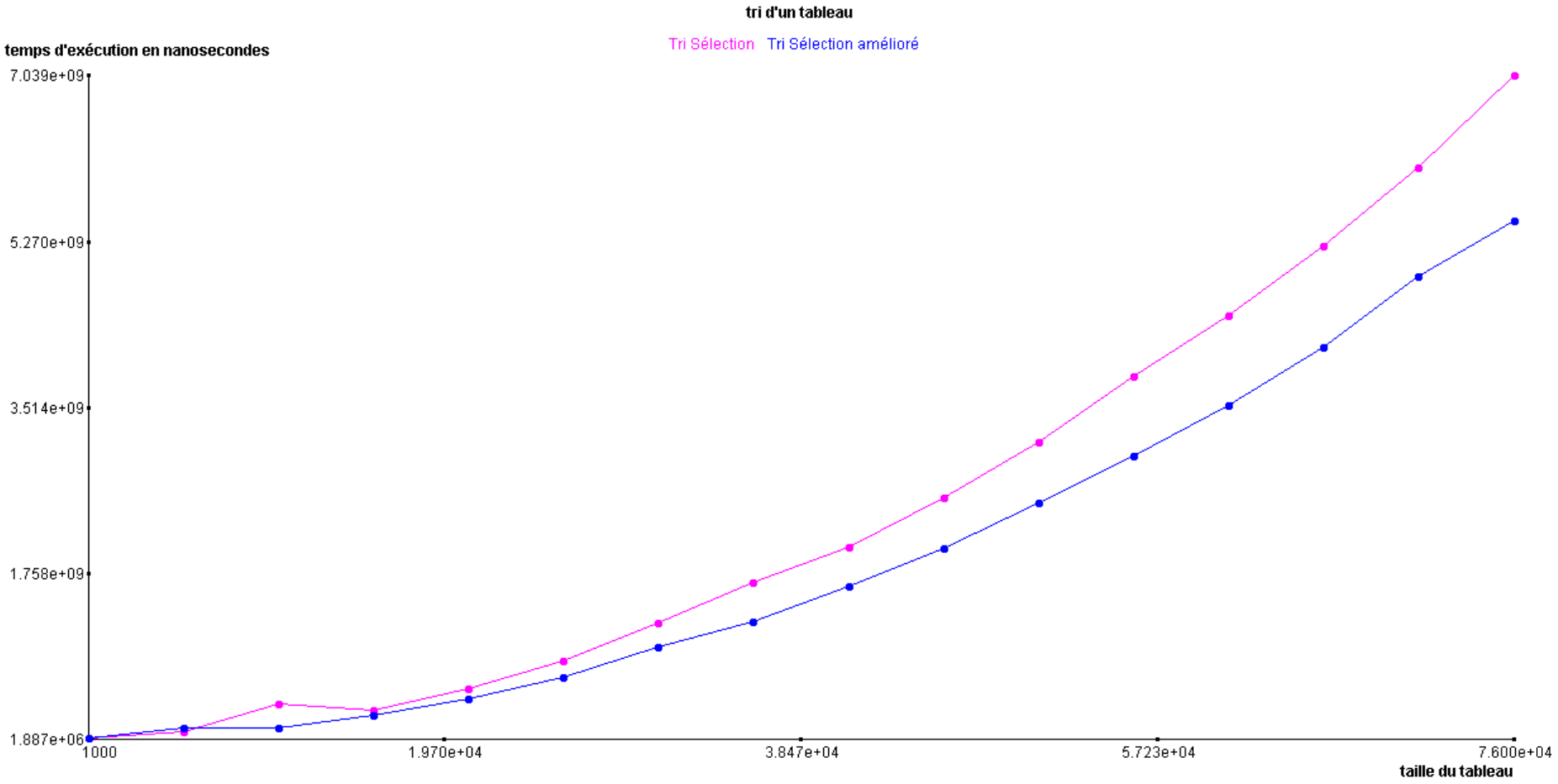
Le tri sélection n'effectue qu'une permutation pour remonter le plus grand élément au bout du tableau, il est donc plus efficace que le tri à bulle.

Le tri sélection est en place et l'algorithme donné ici est stable. Il serait instable si on remplaçait la comparaison **`tab[j] > tab[pg]`** par **`tab[j] >= tab[pg].`**

Le tri sélection peut être amélioré en positionnant à chaque parcours du tableau le plus grand et le plus petit élément, selon le même principe que celui utilisé pour le tri à bulle boustrophédon. Dans ce cas, on fera 2 fois moins de tours de boucle tantque (on trie 2 éléments à chaque tour) mais à chaque tour, on effectuera plus d'opérations.

Comparaison expérimentale des tris par sélection

Test sur des tableaux d'entiers générés aléatoirement.



Tri par insertion (1/2)

Principe du tri par insertion : on prend deux éléments qu'on trie dans le bon ordre, puis un 3e qu'on insère à sa place parmi les 2 autres, puis un 4e qu'on insère à sa place parmi les 3 autres, etc.

2	17	45	45	102	268	415	701

```
fonction sans retour triInsertion(entier[] tab){
  entier i, j, val;
debut
  pour (i allant de 1 à tab.longueur-1 pas 1) faire
      val <- tab[i];
      j <- i;
      tantque ((j > 0) et (tab[j-1] > val)) faire
          tab[j] <- tab[j-1];
          j <- j-1;
      fintantque
      tab[j] <- val;
  finpour
fin
```

Tri par insertion (2/2)

L'algorithme de tri par insertion donné ici est en place et stable (il serait instable si on utilisait ≥ au lieu de >).

Le tri par insertion est l'algorithme utilisé « naturellement » pour trier des cartes ou des dossiers.

Le tri par sélection nécessite en moyenne de l'ordre de $n^2/2$ comparaisons, n étant la taille du tableau : on fera d'abord n-1 tours de la boucle pour, puis n-2, et ainsi de suite jusqu'à 1.

Le tri par insertion nécessite en moyenne de l'ordre de $n^2/4$ comparaisons et déplacements : placer le ième élément demande en moyenne i/2 comparaisons.

Le tri par insertion est donc environ deux fois plus rapide que le tri par sélection.

Comparaison expérimentale des tris

Test sur des tableaux d'entiers générés aléatoirement.