Structurer les données est indispensable pour les manipuler dans les programmes.

Il faut cependant savoir retrouver les données dans ces structures, c'est le but des algorithmes de recherche, et de tri.

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform.

Ada Lovelace (1815-1852) - Notes on the Sketch of The Analytical Engine.

Le problème est de trouver un élément dans une structure linéaire (tableau).

- l'élément peut ne pas être présent
- l'élément peut être présent à plusieurs endroits
- si la structure a plusieurs dimensions, il faut fouiller chaque dimension

Technique intuitive : la recherche séquentielle

- on parcourt la structure dans l'ordre « naturel »
- on s'arrête au bout, ou quand on trouve l'élément
- parfois on peut s'arrêter plus tôt

Recherche séquentielle : on s'arrête quand on trouve l'élément, ou quand on arrive au bout du tableau

```
// 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
```

Variante : on cherche le nombre d'occurences d'un élément

```
// 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
```

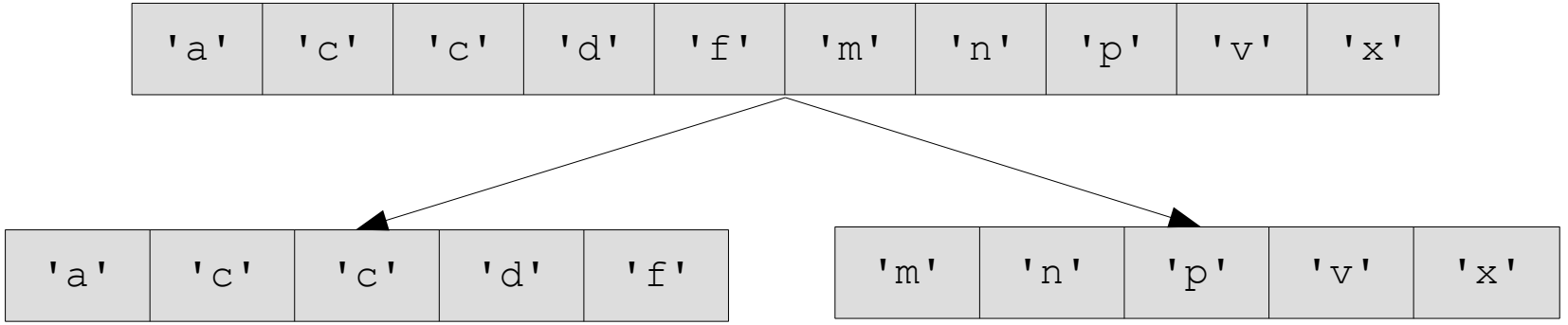
Optimisation : si le tableau est trié, on peut s'arrêter plus « tôt »

```
// 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
```

Remarque : il n'y a optimisation que si on trouve « assez vite » un point d'arrêt

Principe 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.



Ce principe est utile :

- si on peut lancer des programmes en parallèle, on mettra environ deux fois moins de temps pour rechercher l'élément.

- si le tableau est trié, on peut ne chercher l'élément que dans une seule des parties.

Recherche par dichotomie : le tableau est supposé trié par ordre croissant et on cherche un élément *e* dans un tableau *t*

Principe de l'algorithme :

0- on regarde l'élément situé au milieu de *t* :

1- s'il s'agit de *e* c'est gagné.

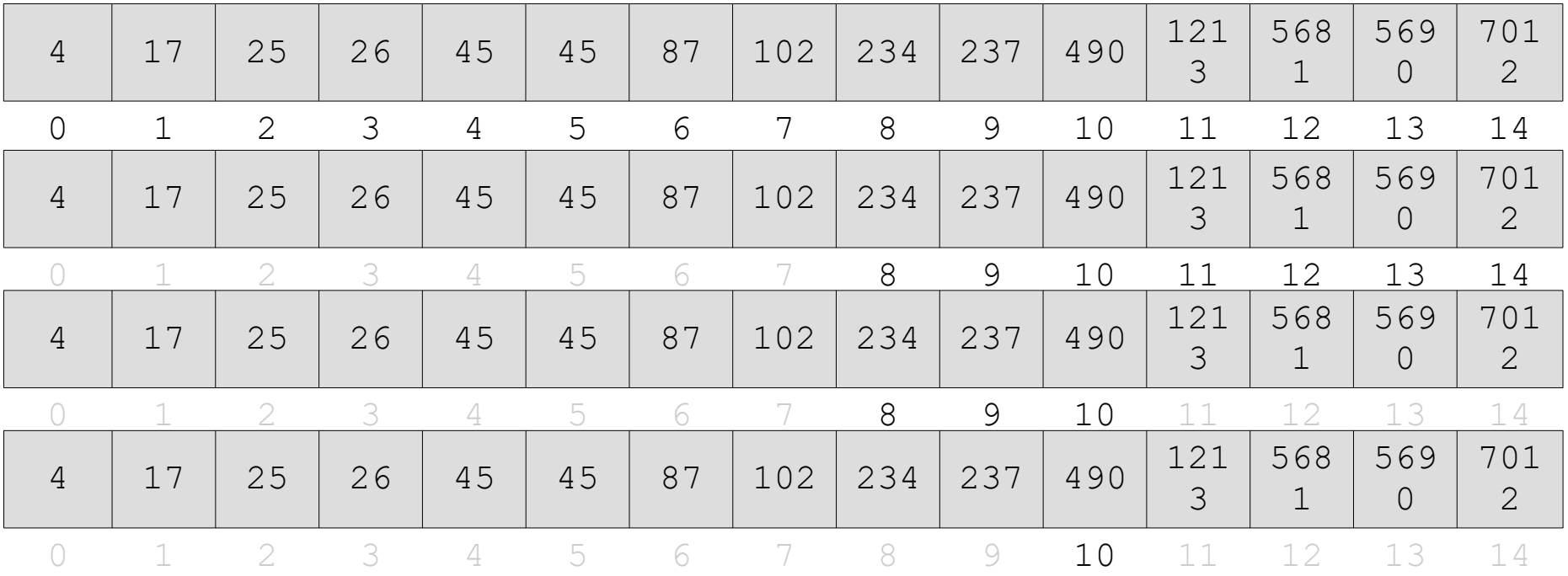
2- s'il est plus grand que *e*, on cherche dans la moitié gauche

3- s'il est plus petit que *e*, on cherche dans la moitié droite

Remarques :

- ça ne peut marcher que si le tableau est trié
- on coupe le tableau en deux parties pas forcément égales en taille

<u>Exemple</u> : rechercher 490 dans le tableau d'entiers suivant.



<u>Remarque</u> : on trouve 490 en 4 itérations, une recherche séquentielle aurait demandé 11 itérations

```
// 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] > e) alors
            j <- (j+i)/2 - 1;
         sinon
            i <- (j+i)/2 + 1;
         finsi
      finsi
   fintantque
   retourne trouve;
fin
```

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.

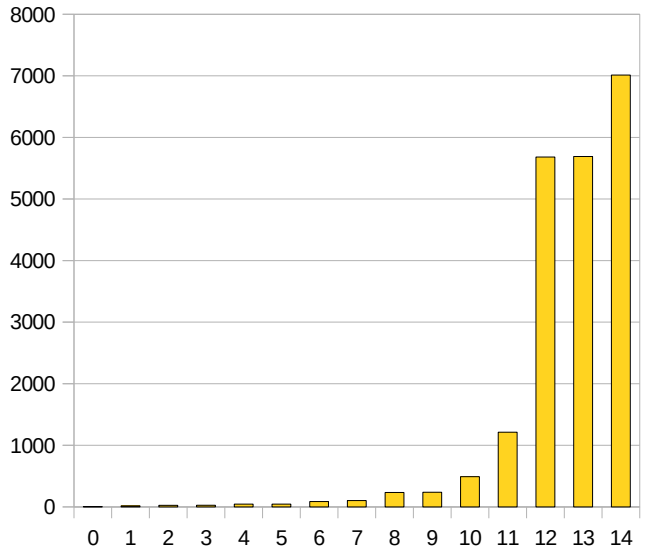
Exemple : chercher un mot dans le dictionnaire se fait par interpolation

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).

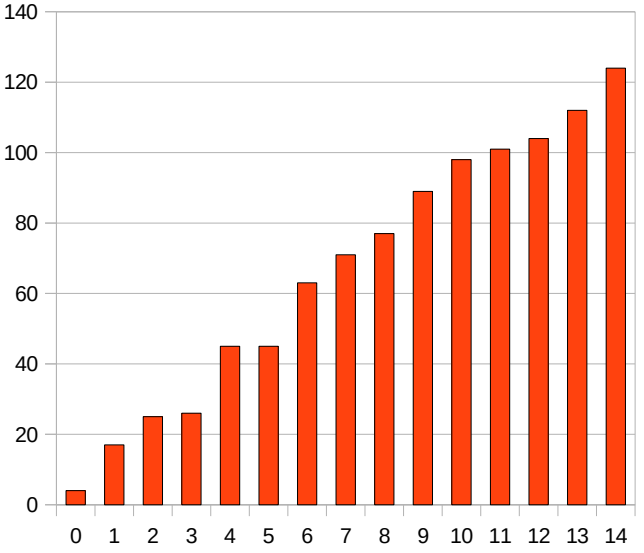
Un cas simple : les éléments sont répartis linéairement dans le tableau.

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



indice interpolé de 490 = 1

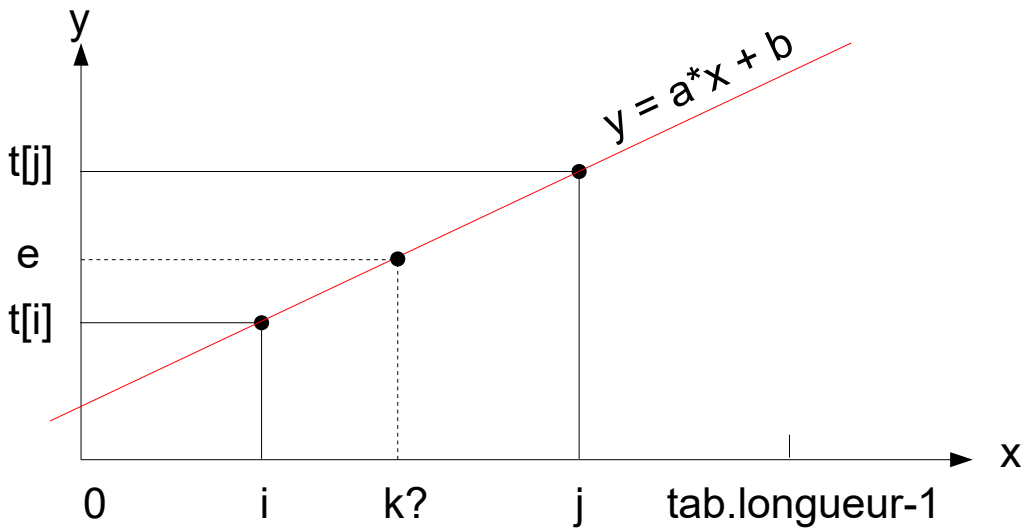


indice interpolé de 71 = 8

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.

La recherche d'élément parait 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(entier 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
```

Optimisation : on peut s'arrêter dès qu'on trouve deux éléments qui ne sont pas dans le bon ordre.

```
fonction avec retour booléen testTrie2(entier 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
```

Le problème est de placer les éléments d'une structure linéaire dans l'ordre

- les éléments doivent pouvoir être comparés
- on peut trier par ordre croissant ou décroissant
- si la structure a plusieurs dimensions, on peut trier chaque dimension

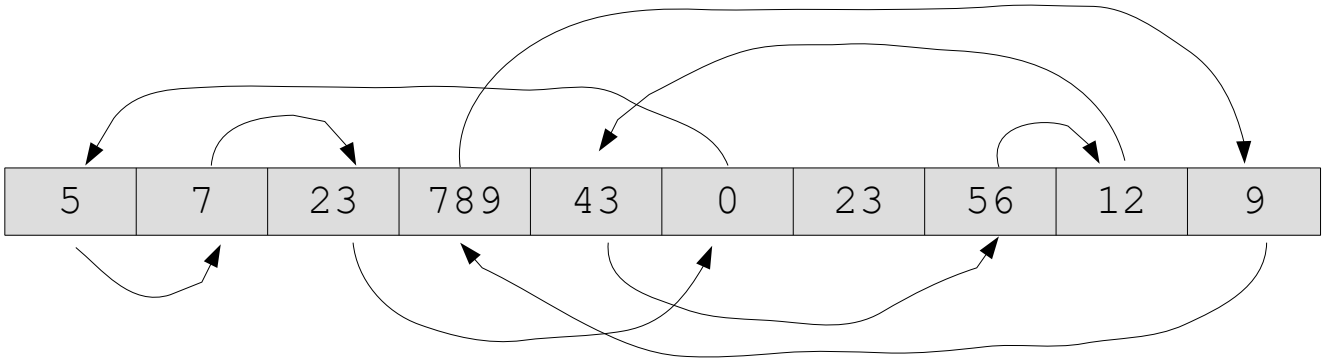
De nombreux algorithmes de tri existent, qui ont chacun leurs avantages et inconvénients en fonction des données à trier.

Certains algorithmes utilisent des structures de données plus complexes (arbres en particulier).

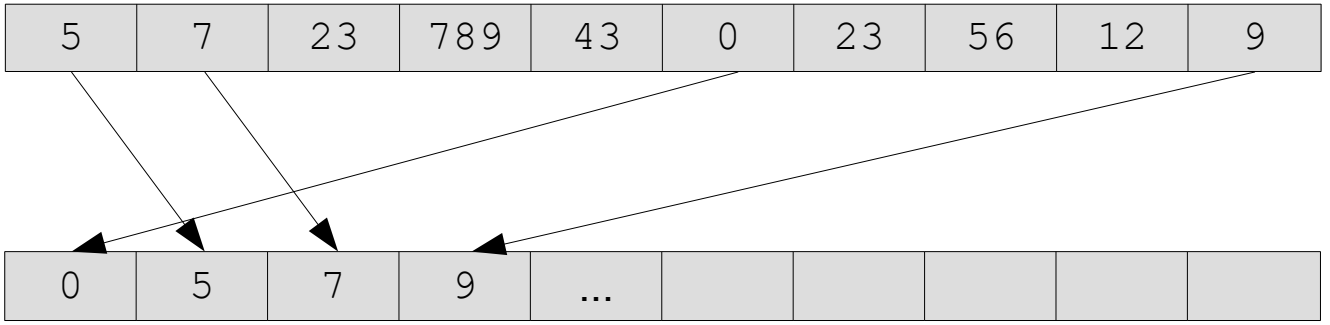
On ne voit ici que quelques tris de base.

Occupation mémoire d'un algorithme de tri :

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

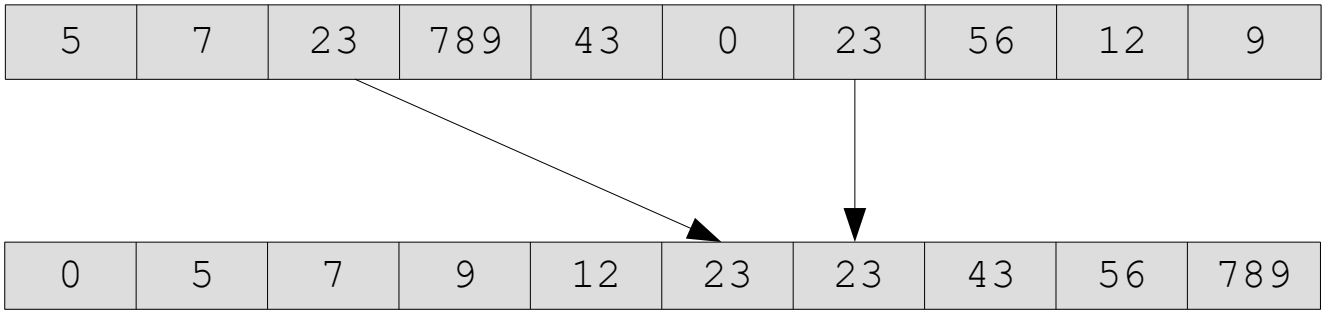


- tri pas en place : on crée une nouvelle structure de données

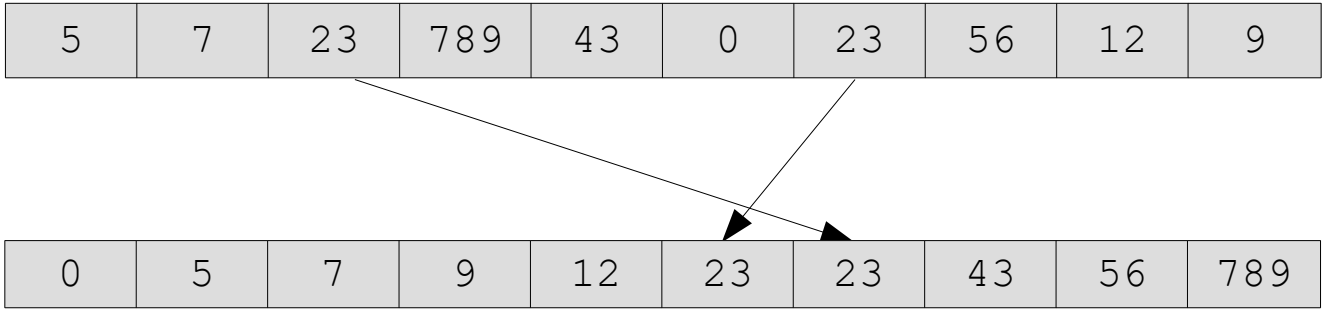


Stabilité d'un algorithme de tri :

- 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 (bubble sort) : on remonte le plus grand élément par permutations et on recommence jusqu'à ce que le tableau soit trié.

Exemple : trier par ordre croissant le tableau suivant

17	2	268	415	45	45	102	701
2	17	268	45	45	102	415	701
2	17	45	45	102	268	415	701
2	17	45	45	102	268	415	701

...

Remarques :

- le tri à bulle est en place. Il est stable si on permute uniquement les éléments différents.
- on pourrait faire remonter le plus petit élément.
- on pourrait s'arrêter dès que le tableau est trié

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
```

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

Optimisation : on s'arrête dès que le tableau est trié.

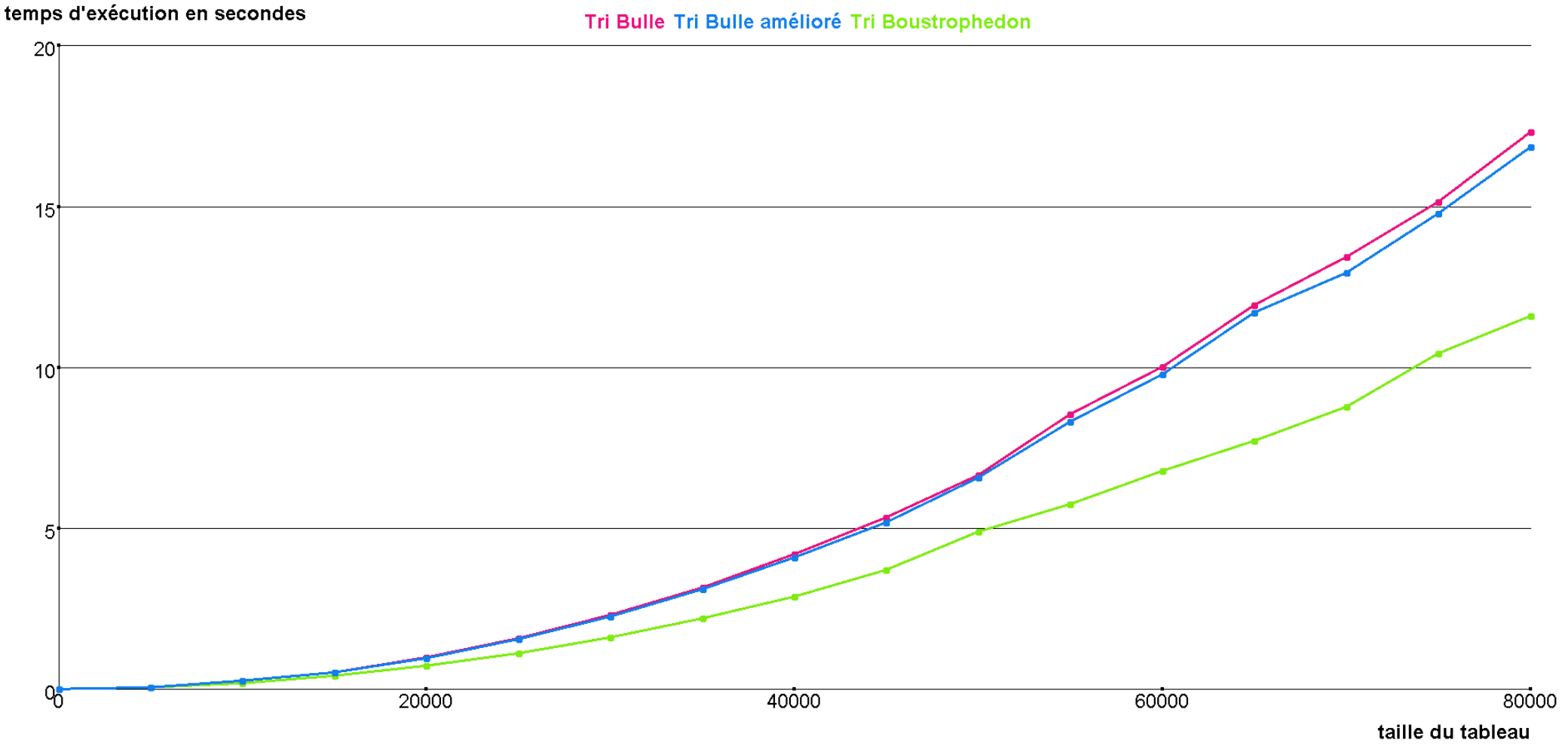
```
fonction sans retour triBulle(entier tab[])
   entier i,j,temp;
   booléen trié;
début
   trié <- faux;
   i <- tab.longueur-2;
   tantque ((non trié) et (i > 0)) faire
     trié <- vrai;
     pour (j allant de 0 à i pas 1) faire
       si (tab[j] > tab[j+1]) alors
         temp <- tab[j];
     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

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

Amélioration : à chaque itération, on remonte le plus grand élément à un bout et le plus petit élément à l'autre bout.

```
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 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;
     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;
   fintantque
fin
```

Test expérimental sur des tableaux d'entiers générés aléatoirement



Tri par sélection (selection sort) : 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
```

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.

Tri par insertion (insertion sort) : 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
```

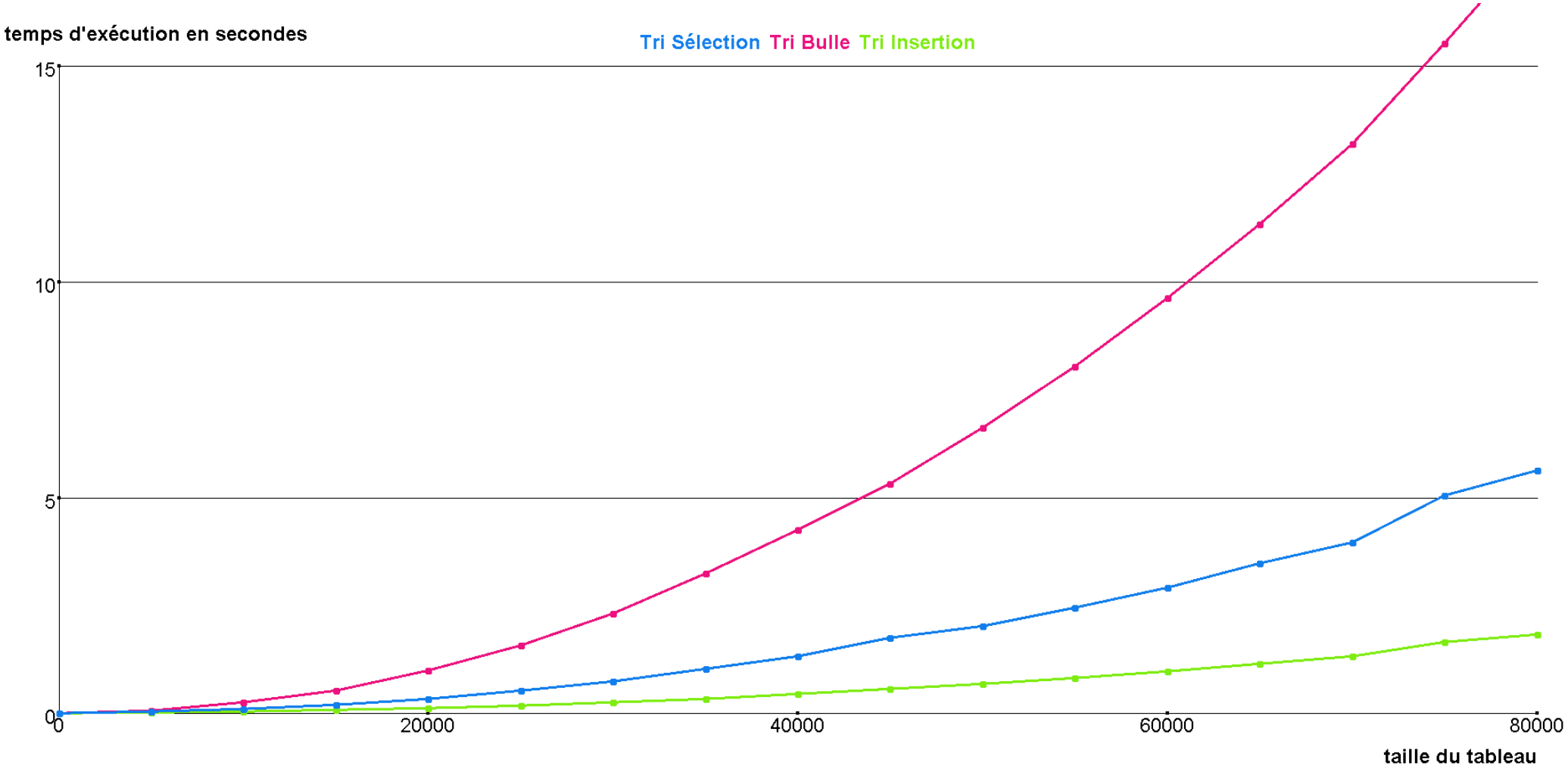
L'algorithme de tri par insertion donné ici est en place et stable (il serait instable si on utilisait ≥ au lieu de >).

Le tri par 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.

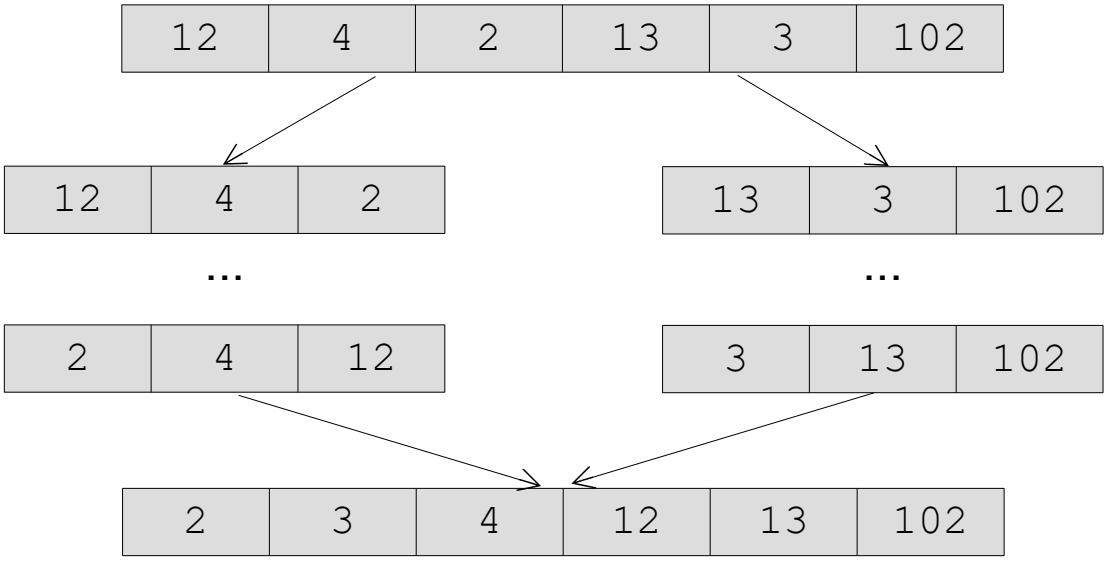
Le tri par insertion nécessite en moyenne de l'ordre de $n^2/4$ comparaisons et déplacements : placer le $i$ème élément demande en moyenne i/2 comparaisons.

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

Test expérimental sur des tableaux d'entiers générés aléatoirement.



Tri fusion (merge sort, John von Neumann, 1945) : on coupe le tableau en deux et on trie chacune des sous-parties (diviser pour régner), puis on reconstitue le tableau entier en fusionnant les deux parties et en respectant l'ordre.



Pour écrire l'algorithme on va appliquer le principe à l'envers : on fusionne des blocs de une case, puis des blocs de deux cases, etc.

Fusionner les deux sous-parties d'un tableau « en place », oblige à réaliser de nombreux décalages. Pour gagner du temps, on va consommer de l'espace mémoire en utilisant un tableau temporaire.

```
// fusion des parties de tab d'indices dans [a,b] et [b+1,c] avec a<b<=c
// les elements de ces parties sont tries en ordre croissant
fonction sans retour fusion(entier tab[],entier a, entier b, entier c)
   entier i, j, k, t[c-a+1];
début
   pour (i allant de 0 à t.longueur-1 pas de 1) faire
     t[i] <- tab[a+i];
   finpour
   i <- 0, j <- b-a+1, k <- a;
   tantque(k <= c){
     si (i <= b-a et (j = c-a+1 ou t[i] <= t[j])) alors
       tab[k] <- t[i];
       i++;
     sinon
       tab[k] <- t[j];
       j++;
     finsi
     k++;
   fintantque
fin
```

Attention, il faut gérer le cas où le dernier bloc dépasse du tableau.

```
fonction sans retour triFusion(entier tab[])
   entier i, n;
debut
   n <- 1;
   tantque (n <= tab.longueur) faire
       pour (i allant de 0 à tab.longueur-1 pas 2n) faire
         si (i+2n-1 < tab.longueur) alors
           fusion(tab,i,i+n-1,i+2n-1);
         sinon
           si (i+n <=tab.longueur-1) alors
             fusion(tab,i,i+n-1,tab.longueur-1);
           finsi
         finsi
       finpour
       n <- 2n;
   fintantque
fin
```

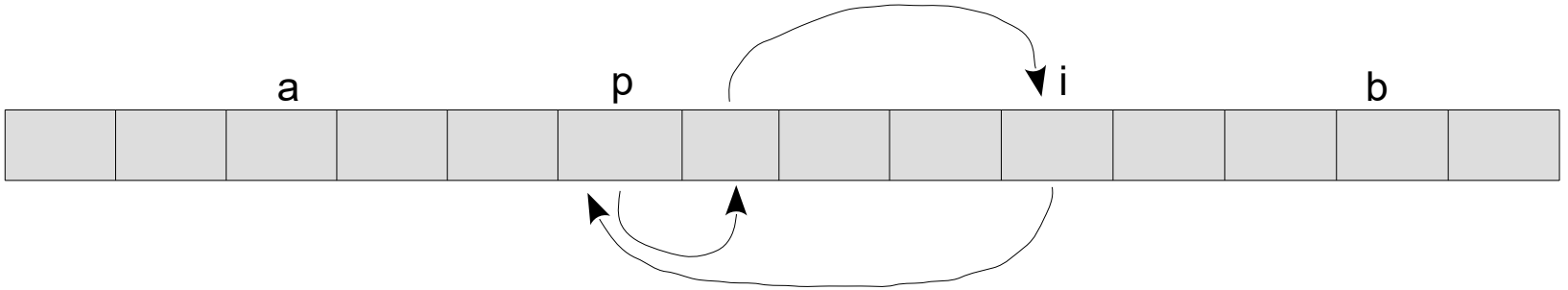
Remarque : l'algorithme de tri fusion donné ici n'est pas en place mais stable (il serait instable si on utilisait < au lieu de ≤ dans l'algorithme de fusion).

Tri rapide (quick sort, Charles Hoare, 1960) : on trie un élément (dit pivot) en déplaçant à sa gauche les éléments plus petits et à sa droite les éléments plus grands. Puis on recommence l'opération sur les deux sous parties gauche et droite.

4	7	12	15	55	67	69	102

Pour partitionner une section [a,b] d'un tableau t, on peut le parcourir séquentiellement :

- au départ le pivot est dans la case p=a
- à tout moment, le pivot est dans la case p et tous les éléments entre a et p-1 sont plus petits que le pivot



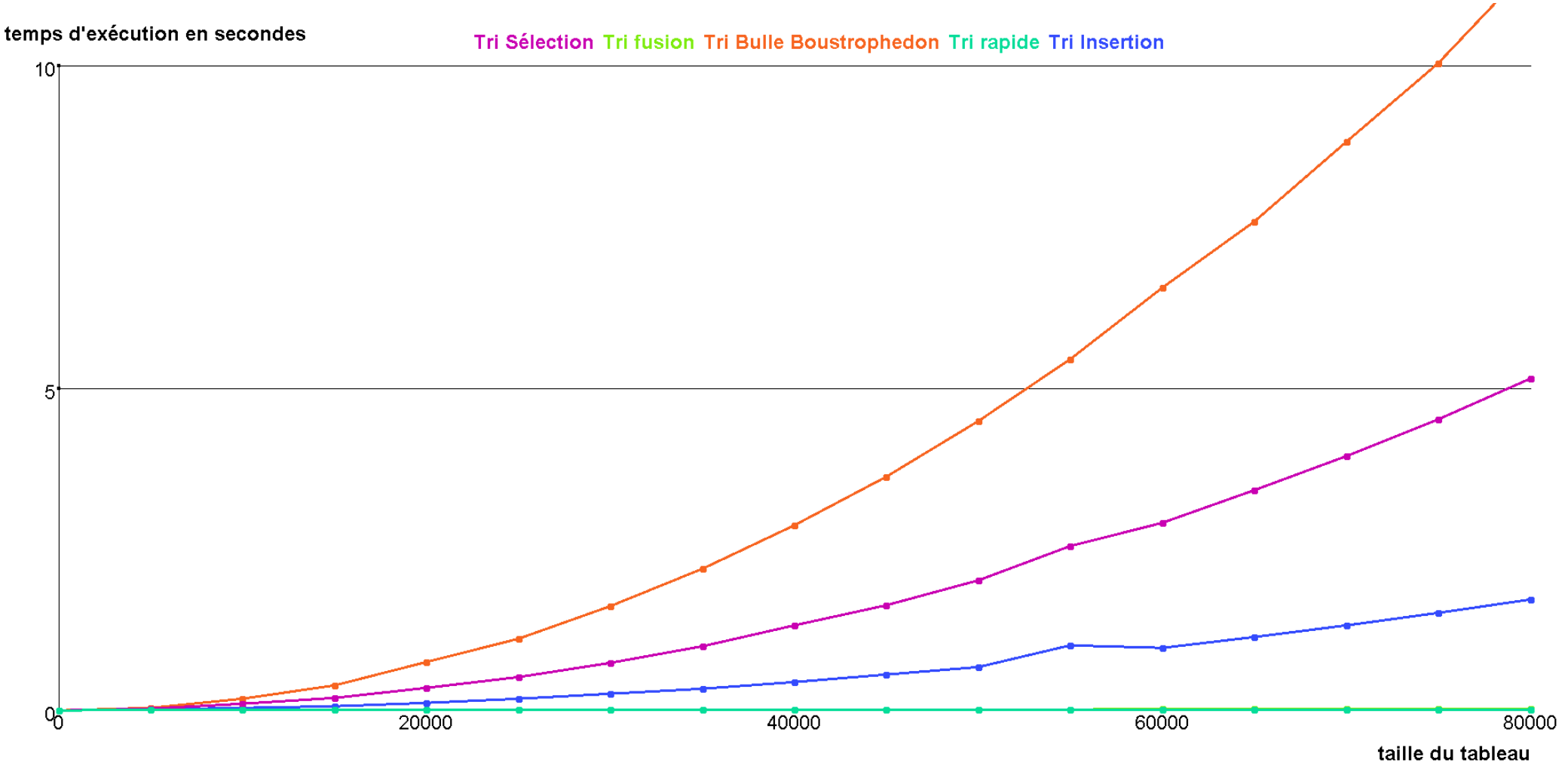
On partitionne en prenant pour pivot le premier élément de l'intervalle d'indices et en perdant complètement l'ordre initial. On pourrait imaginer un algorithme de partition stable, mais il serait moins efficace.

```
// partitionne la section [a,b] de tab utilisant tab[a] comme pivot
// renvoie l'indice du pivot après partition
fonction avec retour entier partition(entier tab[], entier a, entier b)
   entier pivot, i, temp;
debut
   pivot <- a;
   pour (i allant de a+1 à b pas de 1) faire
     si (tab[i]<tab[pivot]) alors
       temp <- tab[i];
       tab[i] <- tab[pivot+1];
       tab[pivot+1] <- tab[pivot];
       tab[pivot]  temp;
       pivot <- pivot+1;
     finsi
   finpour
   retourne pivot;
fin
```

On doit mémoriser au fur et à mesure les sections de tableau à partitionner.

```
fonction sans retour triRapide(entier tab[])
   entier pile[tab.longueur];
   entier top = 1, a, b, p;
debut
   pile[0] <- 0;
   pile[1] <- tab.longueur-1;
   tantque (top>=0) faire
     a <- pile[top-1];
     b <- pile[top];
     top <- top-2;
     p <- partition(tab,a,b);
     si (p-1>a) alors
       pile[top+1] <- a;
       pile[top+2] <- p-1;
       top <- top+2;
     finsi
     si (p+1<b) alors
       pile[top+1] <- p+1;
       pile[top+2] <- b;
       top <- top+2;
     finsi
   fintantque
fin
```

Test expérimental sur des tableaux d'entiers générés aléatoirement.



Test expérimental sur des tableaux d'entiers générés aléatoirement.