

Programmation C et Structures de données (Prog 2)

Cours 3

Un peu d'algorithmique dans un tableau

Pierre Fouilhoux & Christophe Tollu
pierre.fouilhoux@lipn.fr et ct@lipn.univ-paris13.fr

30 janvier 2024

1 Recherche d'un élément dans un tableau

- Rechercheursive
- Recherche dichotomique

2 Deux premiers algorithmes de tri d'un tableau

- Tri par sélection
- Tri par insertion

Contexte du code de ce cours

```
1 #define<stdio.h>
2 #define<stdlib.h>
3 #define<time.h>
4
5 typedef int item;
6
7 #define N 100 /* Taille max du tableau */
8
9 int main () {
10     item tab[N];
11
12     int taille = saisir_entier(0, N);
13
14     srand (time(NULL));
15
16     initialiser_alea_tab(tab, taille);
17
18     /* affichage tableau */
19     afficher_tab_vingt(tab, taille);
20
21     ....
22
23     return EXIT_SUCCESS;
24 }
```

- I5 permet de définir un type "générique" : c'est-à-dire que *item* va représenter le type défini ici : ainsi en changeant simplement *int* par un autre type (et en recompilant) tous les tableaux de type *item* et fonctions manipulant ce type *item* changent de type (attention, le type d'affichage, lui, reste non générique).
- I4 Germe pour le générateur pseudo-aléatoire
- I6 Initialisation des valeurs du tableau à des entiers dans l'intervalle [0,99]
- I9 Affichage des 20 premières case du tableau (ou toutes les cases si moins de 20 cases)

Contexte du code de ce cours

/ Fonction qui initialise pseudo-aleatoirement un tableau t de $taille$ cases avec des nombres entre 0 et 99 */*

```
1
2 void initialiser_alea_tab (item *t, int taille) {
3     int i;
4     for (i = 0; i < taille; ++i)
5         t[i] = rand()%100;
6 }
```

/ Fonction qui affiche les 20 premières valeurs d'un tableau t de $taille$ cases (ou toutes ses cases si moins de 20 cases)*/*

```
1 void afficher_tab_vingt (item *t, int taille) {
2     int i;
3     taille = taille > 20 ? 20 : taille;
4     for (i = 0; i < taille; ++i)
5         printf("%5d ", t[i]);
6     printf("\n");
7 }
```

Contexte du main pour ce cours

/* Fonction qui échange les valeurs de deux items dont les adresses sont reçues comme premier et deuxième paramètres d'entrée */

```
1 void echanger_item (item *x, item *y) {  
2     item tmp = *x;  
3     *x = *y;  
4     *y = tmp;  
5 }
```

1 Recherche d'un élément dans un tableau

- Recherche cursive
- Recherche dichotomique

2 Deux premiers algorithmes de tri d'un tableau

1 Recherche d'un élément dans un tableau

- Recherche cursive
- Recherche dichotomique

2 Deux premiers algorithmes de tri d'un tableau

Recherche d'un élément dans un tableau

Problème : Recherche d'un élément dans un tableau

Entrée : Un tableau t de *taille* cases ($taille \geq 0$)
Un élément x

Sortie : Si x est dans t , renvoyer l'indice d'une case de t contenant x
Sinon renvoyer -1

Algorithme

- On appelle **algorithme** un programme qui se termine en un nombre fini d'opérations et qui répond à un problème donné quelles que soient les entrées du problème.
- Une entrée possible pour un problème s'appelle une **instance** du problème.

Recherche d'un élément dans un tableau

Regardons le programme suivant (sous forme d'une fonction) :

```
1 int recherche_cursive_ite(item x, item *t, int taille){
2     int i=0;
3     while ( (i<taille) && (t[i]!=x) )
4         i++;
5     if (i==taille)
6         return -1;
7     else
8         return i;
9 }
```

Pour l'instance ($x = 7$, $t = \{3, 5, 7, 9, 1, 2\}$, $taille = 6$) :

0	1	2	3	4	5
3	5	7	9	1	2



$i = 0$

Recherche d'un élément dans un tableau

Regardons le programme suivant (sous forme d'une fonction) :

```
1 int recherche_cursive_ite(item x, item *t, int taille){  
2     int i=0;  
3     while ( (i<taille) && (t[i]!=x) )  
4         i++;  
5     if (i==taille)  
6         return -1;  
7     else  
8         return i;  
9 }
```

Pour l'instance ($x = 7$, $t = \{3, 5, 7, 9, 1, 2\}$, $taille = 6$) :

0	1	2	3	4	5
3	5	7	9	1	2

↑
 $i = 1$

Recherche d'un élément dans un tableau

Regardons le programme suivant (sous forme d'une fonction) :

```
1 int recherche_cursive_ite(item x, item *t, int taille){  
2     int i=0;  
3     while ( (i<taille) && (t[i]!=x) )  
4         i++;  
5     if (i==taille)  
6         return -1;  
7     else  
8         return i;  
9 }
```

Pour l'instance ($x = 7$, $t = \{3, 5, 7, 9, 1, 2\}$, $taille = 6$) :

0	1	2	3	4	5
3	5	7	9	1	2



$i = 2$

Recherche d'un élément dans un tableau

Regardons le programme suivant (sous forme d'une fonction) :

```
1 int recherche_cursive_ite(item x, item *t, int taille){
2     int i=0;
3     while ( (i<taille) && (t[i]!=x) )
4         i++;
5     if (i==taille)
6         return -1;
7     else
8         return i;
9 }
```

Terminaison et preuve d'algorithme

Le programme, pour toute instance (x, t, taille)

- se termine en au plus taille itérations de la boucle
- fournit bien la sortie demandée. En effet, en fin d'exécution, on est sorti de la boucle
 - soit car $i \geq \text{taille}$ dans ce cas :
 - soit $i = \text{taille} = 0$ et le tableau est vide : la fonction renvoie -1
 - soit la boucle a parcouru toutes les cases du tableau sans trouver x : la fonction renvoie -1
 - soit car $i < \text{taille}$ et $t[i] \neq x$ est faux, alors $\text{taille} \geq 0$ et i correspond à une case telle que $t[i] = x$: la fonction renvoie alors i .

Ce programme est donc un algorithme répondant au problème de recherche d'un élément dans un tableau !

Recherche d'un élément dans un tableau

Regardons le programme suivant (sous forme d'une fonction) :

```
1 int recherche_cursive_ite(item x, item *t, int taille){
2     int i=0;
3     while ( (i<taille) && (t[i]!=x) )
4         i++;
5     if (i==taille)
6         return -1;
7     else
8         return i;
9 }
```

Terminaison et preuve d'algorithme

Le programme, pour toute instance (x, t, taille)

- se termine en au plus taille itérations de la boucle
- fournit bien la sortie demandée. En effet, en fin d'exécution, on est sorti de la boucle
 - soit car $i \geq \text{taille}$ dans ce cas :
 - soit $i = \text{taille} = 0$ et le tableau est vide : la fonction renvoie -1
 - soit la boucle a parcouru toutes les cases du tableau sans trouver x : la fonction renvoie -1
 - soit car $i < \text{taille}$ et $t[i] \neq x$ est faux, alors $\text{taille} \geq 0$ et i correspond à une case telle que $t[i] = x$: la fonction renvoie alors i .

Ce programme est donc un algorithme répondant au problème de recherche d'un élément dans un tableau !

Recherche d'un élément dans un tableau

Regardons le programme suivant (sous forme d'une fonction) :

```
1 int recherche_cursive_ite(item x, item *t, int taille){
2     int i=0;
3     while ( (i<taille) && (t[i]!=x) )
4         i++;
5     if (i==taille)
6         return -1;
7     else
8         return i;
9 }
```

Terminaison et preuve d'algorithme

Le programme, pour toute instance (x, t, taille)

- se termine en au plus taille itérations de la boucle
- fournit bien la sortie demandée. En effet, en fin d'exécution, on est sorti de la boucle
 - soit car $i \geq \text{taille}$ dans ce cas :
 - soit $i = \text{taille} = 0$ et le tableau est vide : la fonction renvoie -1
 - soit la boucle a parcouru toutes les cases du tableau sans trouver x : la fonction renvoie -1
 - soit car $i < \text{taille}$ et $t[i] \neq x$ est faux, alors $\text{taille} \geq 0$ et i correspond à une case telle que $t[i] = x$: la fonction renvoie alors i .

Ce programme est donc un algorithme répondant au problème de recherche d'un élément dans un tableau !

Recherche d'un élément dans un tableau

Comment mesurer la vitesse de cet algorithme ?

```
1 int recherche_cursive_ite(item x, item *t, int taille){  
2     int i=0;  
3     while ( (i<taille) && (t[i]!=x) )  
4         i++;  
5     if (i==taille)  
6         return -1;  
7     else  
8         return i;  
9 }
```

Nombre d'opérations élémentaires de l'algorithme

- L'exécution d'un algorithme produit un nombre fini d'opérations élémentaires (additions, tests, modulus etc).
- On peut mesurer la vitesse d'un programme en évaluant le nombre d'opérations en fonction de la taille de l'instance.

Recherche d'un élément dans un tableau

Comment mesurer la vitesse de cet algorithme ?

```
1 int recherche_cursive_ite(item x, item *t, int taille){  
2     int i=0;  
3     while ( (i<taille) && (t[i]!=x) )  
4         i++;  
5     if (i==taille)  
6         return -1;  
7     else  
8         return i;  
9 }
```

Nombre d'opérations élémentaires de l'algorithme

- L'exécution d'un algorithme produit un nombre fini d'opérations élémentaires (additions, tests, modulus etc).
- On peut mesurer la vitesse d'un programme en évaluant le nombre d'opérations en fonction de la taille de l'instance.

Recherche d'un élément dans un tableau

Comment mesurer la vitesse de cet algorithme ?

```
1 int recherche_cursive_ite(item x, item *t, int taille){
2     int i=0;
3     while ( (i<taille) && (t[i]!=x) )
4         i++;
5     if (i==taille)
6         return -1;
7     else
8         return i;
9 }
```

Pour la fonction recherche_cursive_ite

- Ici la taille d'une instance est le nombre de case du tableau : *taille*
- Le nombre d'opérations d'une itération de la boucle est au plus $c_1 = 5$.
- en dehors de la boucle, il y a $c_2 = 3$ opérations.
- Si la boucle fait n itérations : on effectue alors au plus $c_1 n + c_2$ opérations.
- **Dans le meilleur des cas (ne dépendant pas de *taille*) :**
x est dans la première case $n = 1$, l'algorithme fait $c_1 + c_2$ opérations.
- **Dans le pire des cas (ne dépendant pas de *taille*) :**
soit x est dans la dernière case, soit x n'est pas dans le tableau : l'algorithme fait $c_1 \cdot \text{taille} + c_2$ opérations.

Recherche d'un élément dans un tableau

Comment mesurer la vitesse de cet algorithme ?

```
1 int recherche_cursive_ite(item x, item *t, int taille){
2     int i=0;
3     while ( (i<taille) && (t[i]!=x) )
4         i++;
5     if (i==taille)
6         return -1;
7     else
8         return i;
9 }
```

Pour la fonction `recherche_cursive_ite`

- Ici la taille d'une instance est le nombre de case du tableau : *taille*
- Le nombre d'opérations d'une itération de la boucle est au plus $c_1 = 5$.
- en dehors de la boucle, il y a $c_2 = 3$ opérations.
- Si la boucle fait n itérations : on effectue alors au plus $c_1 n + c_2$ opérations.
- **Dans le meilleur des cas (ne dépendant pas de *taille*) :**
x est dans la première case $n = 1$, l'algorithme fait $c_1 + c_2$ opérations.
- **Dans le pire des cas (ne dépendant pas de *taille*) :**
soit x est dans la dernière case, soit x n'est pas dans le tableau : l'algorithme fait $c_1 \cdot \text{taille} + c_2$ opérations.

Recherche d'un élément dans un tableau

Même principe cursive en version récursive !

L'idée de la récursivité ici est :

- soit l'élément x est dans la première case
- soit on repose la question de recherche sur le reste du tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille-1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille -1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

Commentaires sur le code

- l3 C'est le cas de base où le tableau est vide
- l4 C'est un autre cas de base où x est dans la première case
- l5 Si on arrive en ligne 5, c'est que le tableau a au moins 1 case et que x n'est pas dans la première case : on interroge alors le tableau d'adresse $t + 1$, c'est-à-dire le sous-tableau de t commençant une case plus loin et ayant $taille - 1$ case.
- l6 On a deux cas :
 - soit x n'a pas été trouvé dans la suite du tableau : on renvoie -1
 - sinon x est dans la case $1 + i$ du tableau (c'est-à-dire qu'on compense le décalage de 1)

Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille-1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

rech_curs_rec(7,996,6)

0	1	2	3	4	5
3	5	7	9	1	2

996 1000 1004 1008 1012 1016

Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille-1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

rech_curs_rec(7,996,6)

0	1	2	3	4	5
3	5	7	9	1	2

996 1000 1004 1008 1012 1016

	0	1	2	3	4	
rech_curs_rec(7,1000,5)	3	5	7	9	1	2
	996	1000	1004	1008	1012	1016

Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille-1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

rech_curs_rec(7,996,6)

0	1	2	3	4	5
3	5	7	9	1	2

996 1000 1004 1008 1012 1016

	0	1	2	3	4	
rech_curs_rec(7,1000,5)	3	5	7	9	1	2

996 1000 1004 1008 1012 1016

			0	1	2	3
rech_curs_rec(7,1004,4)	3	5	7	9	1	2
	996	1000	1004	1008	1012	1016

Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille-1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

rech_curs_rec(7,996,6)

0	1	2	3	4	5
3	5	7	9	1	2

996 1000 1004 1008 1012 1016

	0	1	2	3	4	
rech_curs_rec(7,1000,5)	3	5	7	9	1	2

996 1000 1004 1008 1012 1016

rech_curs_rec(7,1004,4)

		0	1	2	3
3	5	7	9	1	2

996 1000 1004 1008 1012 1016

→ retourne 0

Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille-1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

rech_curs_rec(7,996,6)

0	1	2	3	4	5
3	5	7	9	1	2

996 1000 1004 1008 1012 1016

rech_curs_rec(7,1004,4)

0	1	2	3
3	5	7	9
1	2		

996 1000 1004 1008 1012 1016

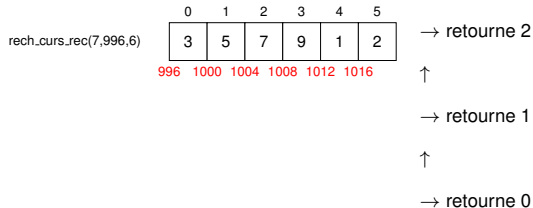
→ retourne 1

↑

→ retourne 0

Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille-1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```



Recherche d'un élément dans un tableau

```
1 int recherche_cursive_rec(item x, item *t, int taille){
2     int i;
3     if (taille==0) return -1;
4     if (t[0]==x) return 0;
5     i=recherche_cursive_rec(x, t+1, taille -1);
6     if (i==-1) return -1;
7         else return 1+i;
8 }
```

Vitesse de l'algorithme

- On peut constater que le code converge vers un des deux cas de base :
 - soit tableau vide : c'est que x n'est pas dans le tableau
 - soit vers le cas où x est en première case d'un sous-tableau**Donc l'algorithme se termine.**
- On peut voir que le nombre d'opérations d'un appel est bornée par une constante $K = 8$. Ainsi le nombre d'opérations est proportionnel au nombre d'appels récurrents.
- Notons C_n le nombre d'appels réalisés pour un tableau de n cases : c'est-à-dire l'appel *recherche_cursive_rec*(x, t, n). Alors on constate que :
 - $C_0 = 1$ c'est l'appel lui-même.
 - $C_n = 1 + C_{n-1}$ l'appel courant plus autant d'appels que ceux fait par la fonction pour un tableau de $n - 1$ cases.
- Ainsi C_n est une suite très simple dont on peut calculer le terme général : $C_n = n + 1$.
- On a donc $K(n + 1)$ opérations pour un tableau de n cases.

Recherche d'un élément dans un tableau

Itératif ou récursif ?

- On peut donc noter que les deux implémentations de la recherche cursive en programmation récursive ou itérative ont le même ordre de grandeur d'opérations utilisées : on dit que la recherche cursive est de l'ordre de n (où n est le nombre d'éléments du tableau).
- Mais l'implémentation récursive précédente peut empiler jusqu'à n appels récursifs et donc consommer davantage de mémoire.

Mais on peut écrire une version terminale !

```
1 int recherche_cursive_terminale(item x, item *t, int taille, int decalage){
2     if (taille==0) return -1;
3     if (t[0]==x) return decalage;
4     return recherche_cursive_terminale(x, t+1, taille -1, decalage+1);
5 }
```

Itératif ou récursif terminale ?

Une fonction itérative et une fonction récursive terminale implémentant la même idée pour un même problème sont équivalentes en temps et en consommation mémoire.

Recherche d'un élément dans un tableau

Itératif ou récursif ?

- On peut donc noter que les deux implémentations de la recherche cursive en programmation récursive ou itérative ont le même ordre de grandeur d'opérations utilisées : on dit que la recherche cursive est de l'ordre de n (où n est le nombre d'éléments du tableau).
- Mais l'implémentation récursive précédente peut empiler jusqu'à n appels récursifs et donc consommer davantage de mémoire.

Mais on peut écrire une version terminale !

```
1 int recherche_cursive_terminale(item x, item *t, int taille, int decalage){
2     if (taille==0) return -1;
3     if (t[0]==x) return decalage;
4     return recherche_cursive_terminale(x, t+1, taille-1, decalage+1);
5 }
```

Itératif ou récursif terminale ?

Une fonction itérative et une fonction récursive terminale implémentant la même idée pour un même problème sont équivalentes en temps et en consommation mémoire.

Recherche d'un élément dans un tableau

Itératif ou récursif ?

- On peut donc noter que les deux implémentations de la recherche cursive en programmation récursive ou itérative ont le même ordre de grandeur d'opérations utilisées : on dit que la recherche cursive est de l'ordre de n (où n est le nombre d'éléments du tableau).
- Mais l'implémentation récursive précédente peut empiler jusqu'à n appels récursifs et donc consommer davantage de mémoire.

Mais on peut écrire une version terminale !

```
1 int recherche_cursive_terminale(item x, item *t, int taille, int decalage){  
2     if (taille==0) return -1;  
3     if (t[0]==x) return decalage;  
4     return recherche_cursive_terminale(x, t+1, taille -1, decalage+1);  
5 }
```

Itératif ou récursif terminale ?

Une fonction itérative et une fonction récursive terminale implémentant la même idée pour un même problème sont équivalentes en temps et en consommation mémoire.

1 Recherche d'un élément dans un tableau

- Recherche cursive
- Recherche dichotomique

2 Deux premiers algorithmes de tri d'un tableau

Recherche d'un élément dans un tableau trié

Problème : Recherche d'un élément dans un tableau trié

Entrée : Un tableau trié de taille cases ($taille \geq 0$)
Un élément x

Sortie : Si x est dans t , renvoyer l'indice d'une case de t contenant x
Sinon renvoyer -1

Quel algorithme ?

- La fonction précédente `recherche_cursive_ite` répond bien au problème : c'est bien un algorithme pour ce problème.
- Mais est-elle la plus rapide alors qu'elle n'utilise pas le fait que le tableau est trié ?

Recherche d'un élément dans un tableau **trié**

Idée de la dichotomie

Comme le tableau est trié : prenons m la case du milieu du tableau

- soit x est dans la case m
- soit x dans un des deux sous-tableaux $[0 \dots m - 1]$ ou $[m + 1 \dots \text{taille} - 1]$

Il est très naturel de coder cela récursivement :

```
1 int recherche_dicho_rec(item x, item *t, int taille) {
2     int m = taille / 2;
3     if (taille <= 0) return -1;
4     if (t[m] == x) return m;
5     if (t[m] > x)
6         return recherche_dicho_rec(x, t, m);
7     else {
8         int res = recherche_dicho_rec(x, t + m + 1, taille - (m + 1));
9         if (res == -1)
10             return -1;
11         else
12             return m + 1 + res;
13     }
14 }
```

Recherche d'un élément dans un tableau trié

```
1 int recherche_dicho_rec(item x, item *t, int taille) {
2     int m = taille/2;
3     if (taille <= 0) return -1;
4     if (t[m] == x) return m;
5     if (x < t[m])
6         return recherche_dicho_rec(x, t, m);
7     else {
8         int res = recherche_dicho_rec(x, t + m + 1, taille - (m + 1));
9         if (res == -1)
10             return -1;
11         else
12             return m + 1 + res;
13     }
14 }
```

Commentaires sur le code

l3&4 Deux cas de bases : tableau vide ou élément dans la case du milieu

l5 A partir de cette ligne, on sait que le tableau a au moins une case et que x n'est pas dans la case m .
Donc deux cas en écartant le sous-tableau où ne peut pas être x

l6 Ici $x < t[m]$: on relance sur le sous-tableau $[0 \dots m - 1]$: donc le tableau d'adresse t de m cases

l8 Ici $t[m] < x$: on relance sur le sous-tableau $[m + 1 \dots \text{taille} - 1]$ donc le tableau d'adresse $t + m + 1$ de $\text{taille} - (m + 1)$ cases.

l6&9 Si x n'a pas été trouvé récursivement : on retourne -1

l12 Si x a été trouvé dans le tableau d'adresse $t + m + 1$ de $\text{taille} - (m + 1)$ cases, on renvoie le bon indice décalé dans le tableau t .

Recherche d'un élément dans un tableau trié

Et on peut écrire ce code en version récursive terminale pour éviter les empilements d'appels en mémoire.

```
1 int recherche_dicho_terminale (item x, item *t, int taille , int decalage) {  
2     int m = taille / 2;  
3     if (taille <= 0) return -1;  
4     if (t[m] == x) return decalage + m;  
5     if (x < t[m])  
6         return dico_terminale(x, t, m, decalage);  
7     else  
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);  
9 }
```

Recherche d'un élément dans un tableau trié

```
1 int recherche_dicho_terminale (item x, item *t, int taille , int decalage) {
2     int m = taille / 2;
3     if (taille <= 0) return -1;
4     if (t[m] == x) return decalage + m;
5     if (x < t[m])
6         return dico_terminale(x, t, m, decalage);
7     else
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);
9 }
```

Commentaires sur le code

- l1 Ajout d'un paramètre indiquant le décalage des indices du tableau passé en paramètre par rapport à l'indice 0 du tableau d'origine.
- l6 Ici $x < t[m]$: le sous-tableau $[0 \dots m - 1]$ est le tableau d'adresse t de m cases de décalage 0 (en plus du décalage précédent)
- l8 Ici $t[m] < x$: le sous-tableau $[m + 1 \dots \text{taille} - 1]$ est le tableau d'adresse $t + m + 1$ de $\text{taille} - (m + 1)$ cases de décalage $m + 1$ (en plus du décalage précédent)

Recherche d'un élément dans un tableau trié

```
1 int recherche_dicho_terminale (item x, item *t, int taille , int decalage) {  
2     int m = taille / 2;  
3     if (taille <= 0) return -1;  
4     if (t[m] == x) return decalage + m;  
5     if (x < t[m])  
6         return dico_terminale(x, t, m, decalage);  
7     else  
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);  
9 }
```

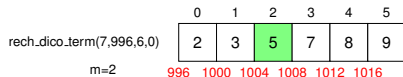
rech_dico_term(7,996,6,0)

0	1	2	3	4	5
2	3	5	7	8	9

m=2 996 1000 1004 1008 1012 1016

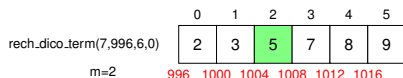
Recherche d'un élément dans un tableau trié

```
1 int recherche_dicho_terminale (item x, item *t, int taille , int decalage) {
2     int m = taille / 2;
3     if (taille <= 0) return -1;
4     if (t[m] == x) return decalage + m;
5     if (x < t[m])
6         return dico_terminale(x, t, m, decalage);
7     else
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);
9 }
```



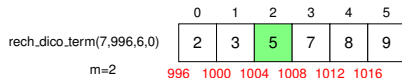
Recherche d'un élément dans un tableau trié

```
1 int recherche_dicho_terminale (item x, item *t, int taille , int decalage) {
2     int m = taille/2;
3     if (taille <= 0) return -1;
4     if (t[m] == x) return decalage + m;
5     if (x < t[m])
6         return dico_terminale(x, t, m, decalage);
7     else
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);
9 }
```



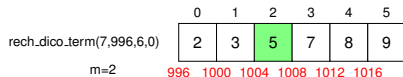
Recherche d'un élément dans un tableau trié

```
1 int recherche_dicho_terminale (item x, item *t, int taille, int decalage) {  
2     int m = taille/2;  
3     if (taille <= 0) return -1;  
4     if (t[m] == x) return decalage + m;  
5     if (x < t[m])  
6         return dico_terminale(x, t, m, decalage);  
7     else  
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);  
9 }
```



Recherche d'un élément dans un tableau trié

```
1 int recherche_dicho_terminale (item x, item *t, int taille , int decalage) {  
2     int m = taille / 2;  
3     if (taille <= 0) return -1;  
4     if (t[m] == x) return decalage + m;  
5     if (x < t[m])  
6         return dico_terminale(x, t, m, decalage);  
7     else  
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);  
9 }
```



Recherche d'un élément dans un tableau trié

Quelle est la vitesse de cet algorithme de recherche dichotomique récursif ?

```
1 int recherche_dicho_terminale (item x, item *t, int taille, int decalage) {  
2     int m = taille / 2;  
3     if (taille <= 0) return -1;  
4     if (t[m] == x) return decalage + m;  
5     if (x < t[m])  
6         return dico_terminale(x, t, m, decalage);  
7     else  
8         return dico_terminale(x, t + m + 1, taille - (m + 1), decalage + m + 1);  
9 }
```

Vitesse de l'algorithme

- On peut constater que le code converge vers un des deux cas de base :
 - soit tableau vide : c'est que x n'est pas dans le tableau
 - soit vers le cas où x est dans la case du milieu**Donc l'algorithme se termine.**
- On peut voir que le nombre d'opérations d'un appel est bornée par une constante $K = 12$. Ainsi le nombre d'opérations est proportionnel au nombre d'appels récursifs.
- Notons C_n le nombre d'appels réalisés pour un tableau de n cases : c'est-à-dire l'appel *recherche_cursive_rec*($x, t, n, 0$). Alors on constate que :
 $C_0 = 1$: c'est l'appel lui-même.
 $C_n \leq 1 + C_{\lfloor \frac{n}{2} \rfloor}$

c'est l'appel lui-même plus le nombre d'appels réalisé sur un demi-tableau, qui est au plus de taille $\lfloor \frac{n}{2} \rfloor$.
(Remarquons qu'il n'est pas évident d'écrire C_n car on coupe le tableau "à peu près" en deux...)

Quelle est la vitesse de cet algorithme de recherche dichotomique récursif ?

Pour simplifier les calculs (et sans perte de généralité), on peut supposer que le nombre de cases n est une puissance de 2.

On peut alors noter $n = 2^k$ avec k un entier.

Remarquons alors qu'en appliquant le logarithme, on a

$$\log(n) = \log(2^k) = k \cdot \log(2)$$

donc
$$k = \frac{\log(n)}{\log(2)} = \log_2(n)$$

Recherche d'un élément dans un tableau trié

Quelle est la vitesse de cet algorithme de recherche dichotomique récursif ?

On a alors

$$C_0 = 1$$

$$C_n \leq C_{2^k} \leq 1 + C_{\frac{n}{2}} = 1 + C_{2^{k-1}}$$

En réitérant plusieurs fois la récursivité de la formule, on obtient

$$C_n \leq 1 + C_{2^{k-1}} \leq 2 + C_{2^{k-2}} \leq 3 + C_{2^{k-3}} \leq \dots \leq k + C_{2^0} \leq k + 2$$

$$\text{Donc } C_n \leq \log_2(n) + 2$$

Vitesse de la recherche dichotomique

- La recherche dichotomique a une vitesse de l'ordre de $\log_2(n)$ opérations réalisées pour un tableau de n cases.
- C'est bien plus rapide que la recherche cursive qui est de l'ordre de n : mais il faut pour l'utiliser avoir un tableau trié !

Recherche d'un élément dans un tableau trié

Quelle est la vitesse de cet algorithme de recherche dichotomique récursif ?

On a alors

$$C_0 = 1$$

$$C_n \leq C_{2^k} \leq 1 + C_{\frac{n}{2}} = 1 + C_{2^{k-1}}$$

En réitérant plusieurs fois la récursivité de la formule, on obtient

$$C_n \leq 1 + C_{2^{k-1}} \leq 2 + C_{2^{k-2}} \leq 3 + C_{2^{k-3}} \leq \dots \leq k + C_{2^0} \leq k + 2$$

Donc $C_n \leq \log_2(n) + 2$

Vitesse de la recherche dichotomique

- La recherche dichotomique a une vitesse de l'ordre de $\log_2(n)$ opérations réalisées pour un tableau de n cases.
- C'est bien plus rapide que la recherche cursive qui est de l'ordre de n : mais il faut pour l'utiliser avoir un tableau trié !

Itératif ou récursif terminal ?

- On peut écrire une version itérative du principe de dichotomie :
il faut pour cela utiliser deux itérateurs d et f repérant les débuts et fin du sous-tableau restant à explorer.
- En fait, on peut toujours **dérécursifier** un code récursif !
Pour cela, il suffit de simuler la pile d'exécution de l'appel des fonctions.
- Ainsi on peut avoir une version itérative de la recherche dichotomique de l'ordre de $\log_2(n)$.
Mais on préfère largement cette écriture récursive terminale très lisible !

1 Recherche d'un élément dans un tableau

2 Deux premiers algorithmes de tri d'un tableau

- Tri par sélection
- Tri par insertion

Problème du tri d'un tableau

Problème : tri d'un tableau

Entrée : Un tableau t de *taille* cases ($taille \geq 0$)

Sortie : Le tableau t trié par ordre croissant.

Algorithme de tri

- Un algorithme de tri est un algorithme permettant de trier tous les tableaux sans faire d'hypothèse sur les éléments du tableau
- sauf que ces éléments sont comparables ($<$, \leq , $==$) deux à deux.

- 1 Recherche d'un élément dans un tableau
- 2 Deux premiers algorithmes de tri d'un tableau
 - Tri par sélection
 - Tri par insertion

Tri d'un tableau par sélection

Idée du tri par sélection

On recherche le plus petit élément du tableau
et on l'échange avec le premier élément.

Puis on réitère sur le tableau à partir du deuxième élément.

```
1 \begin{lstlisting}[language=c]
2 void trier_par_selection(item *t, int taille) {
3     int i, j, pos_min;
4     for (i = 0; i < taille - 1; i++) {
5         /* Recherche de l'indice du plus petit element entre i et taille-1 */
6         pos_min=i;
7         for (j=i+1;j<taille; j++)
8             if (t[j] < t[pos_min]) pos_min = j;
9
10        echanger_item (t+i, t+pos_min);
11    }
12 }
```

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
3	9	5	7	1

↑ ↑

$i = 0$ $j = 1$

$pos_min = 0$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
3	9	5	7	1



$i = 0$



$j = 2$

$pos_min = 0$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
3	9	5	7	1



$i = 0$



$j = 3$

$pos_min = 0$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
3	9	5	7	1



i = 0



j = 4 pos_min = 4

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	9	5	7	3



$i = 0$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	9	5	7	3



$i = 1$ $j = 2$

$pos_min = 2$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	9	5	7	3



$i = 1$



$j = 3$

$pos_min = 2$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	9	5	7	3



i = 1



j = 4 pos_min = 4

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	3	5	7	9



$i = 1$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {
2     int i, j, pos_min;
3     for (i = 0; i < taille - 1; i++) {
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */
5         pos_min=i;
6         for (j=i+1;j<taille; j++)
7             if (t[j] < t[pos_min]) pos_min = j;
8
9         echanger_item (t+i, t+pos_min);
10    }
11 }
```

0	1	2	3	4
1	3	5	7	9



$i = 2$ $j = 3$

$pos_min = 2$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	3	5	7	9

 $i = 2$  $j = 4$ $pos_min = 2$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	3	5	7	9



i = 2

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	3	5	7	9



$i = 3$ $j = 4$ $pos_min = 3$

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {  
2     int i, j, pos_min;  
3     for (i = 0; i < taille - 1; i++) {  
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */  
5         pos_min=i;  
6         for (j=i+1;j<taille; j++)  
7             if (t[j] < t[pos_min]) pos_min = j;  
8  
9         echanger_item (t+i, t+pos_min);  
0     }  
1 }
```

0	1	2	3	4
1	3	5	7	9



i = 3

Tri d'un tableau par sélection

```
1 void trier_par_selection(item *t, int taille) {
2     int i, j, pos_min;
3     for (i = 0; i < taille - 1; i++) {
4         /* Recherche de l'indice du plus petit element entre i et taille-1 */
5         pos_min=i;
6         for (j=i+1;j<taille; j++)
7             if (t[j] < t[pos_min]) pos_min = j;
8
9         echanger_item (t+i, t+pos_min);
10    }
11 }
```

Preuve de l'algorithme

- On peut voir **qu'après l'itération** i de la boucle sur i , les cases d'indice 0 à i contiennent les $i + 1$ plus petites valeurs du tableau rangées dans l'ordre croissant
- La proposition précédente est vraie pour toute valeur de i , donc aussi pour $i = \text{taille} - 1$ quand la fonction se termine. Donc à la fin le tableau est trié.

Tri d'un tableau par sélection

Vitesse du tri par sélection

- Le nombre total de comparaisons de valeurs est

$$\begin{aligned}\sum_{i=0}^{taille-2} ((taille - 1) - (i + 1)) &= \sum_{i=0}^{taille-2} (taille - 2 - i) \\ &= (taille - 2) + (taille - 3) + \dots + 2 + 1 + 0 \\ &= \frac{(taille - 3)(taille - 2)}{2}\end{aligned}$$

- Quelle que soit l'aspect du tableau (même déjà trié)

Le tri par sélection est de l'ordre de n^2

où n est le nombre d'éléments du tableau.

1 Recherche d'un élément dans un tableau

2 Deux premiers algorithmes de tri d'un tableau

- Tri par sélection
- Tri par insertion

Tri d'un tableau par insertion

Idée du tri par insertion

- Si besoin, on inverse l'ordre croissant des 2 premières cases
Les 2 premières cases sont alors triées.
- Puis on insère la 3ème case à la bonne place parmi les 2 premières.
Les 3 premières cases sont alors triées.
- ...
- Puis on réitère cela pour chaque case i
en la faisant descendre parmi les $i - 1$ premières cases
Les i premières cases sont alors triées.

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
9	1	5	7	3



$i = 1$



$j = 1$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	9	5	7	3



$i = 1$



$j = 1$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	9	5	7	3



$i = 2$



$j = 2$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	9	7	3



$i = 2$



$j = 1$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	9	7	3



$i = 2$



$j = 1$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	9	7	3



$i = 3$



$j = 3$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	7	9	3



$i = 3$



$j = 2$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	7	9	3



$i = 3$



$j = 2$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	7	9	3



$i = 4$



$j = 4$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	7	3	9



$i = 4$



$j = 3$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	5	3	7	9



$i = 4$



$j = 2$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	3	5	7	9



$i = 4$



$j = 1$

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i = 1; i < taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

0	1	2	3	4
1	3	5	7	9

Tri d'un tableau par insertion

```
1 void trier_par_insertion (item *t, int taille) {  
2     int i, j;  
3     for (i=1; i<taille; i++) {  
4         j = i;  
5         while (j > 0 && t[j] < t[j-1]) {  
6             echanger_item (t+j-1, t+j);  
7             j--;  
8         }  
9     }  
0 }
```

Preuve de l'algorithme

- On peut constater **qu'après l'itération i** de la boucle suivante, les valeurs occupant les cases d'indice 0 à i sont rangées dans l'ordre croissant.
- Donc à la fin de la fonction, le tableau est trié.

Vitesse du tri par insertion

- **Dans le cas le plus défavorable** : par exemple si le tableau est en ordre décroissant.

le nombre total de comparaisons de valeurs est :

$$\sum_{i=1}^{taille-1} = 1 + 2 + \dots + (taille - 2) + (taille - 1) = \frac{taille(taille - 1)}{2}$$

- **Dans le cas le plus favorable** : par exemple si le tableau est déjà trié
le nombre total de comparaisons de valeurs est :

$$1 + 1 + \dots + 1 + 1 = taille - 1$$

Tri par sélection ou par insertion

- On peut voir que dans **le pire des cas**, les tris par sélection ou par insertion ont le même ordre de grandeur d'opérations (en n^2 où n est le nombre d'éléments du tableau).
- Mais le tri par sélection a une vitesse fixe quel que soit le tableau alors que le tri par insertion va plus vite dans les cas favorables !

Mais nous verrons d'autres tris plus rapides (de l'ordre de $n\log(n)$).

Tri par sélection ou par insertion

- On peut voir que dans **le pire des cas**, les tris par sélection ou par insertion ont le même ordre de grandeur d'opérations (en n^2 où n est le nombre d'éléments du tableau).
- Mais le tri par sélection a une vitesse fixe quel que soit le tableau alors que le tri par insertion va plus vite dans les cas favorables !

Mais nous verrons d'autres tris plus rapides (de l'ordre de $n\log(n)$).