



R. Malgouyres, R. Zrouer et F. Feschet
Initiation à l'algorithmique et à la programmation en C,
Cours avec 129 exercices corrigés,
DUNOD, Collection Sciences Sup, 2011, 2^e édition

Algorithmique et programmation en C

TP n° 9

Comparaison expérimentale d'algorithmes de tris

Objectifs :

Le but du TP est d'implémenter plusieurs algorithmes de tris et de comparer leurs temps d'exécution sur des jeux de données aléatoires. On verra au passage les fonctions `srand`, `rand` et `clock`.

1 Un générateur aléatoire de données

Exercice 1 Définir dans un fichier `tris.h` une structure `Date` comprenant

- L'année (entier entre 1995 et 2005);
- Le mois (entier entre 1 et 12);
- Le jour dans le mois (pour simplifier, entier entre 1 et 30);
- L'heure (entier entre 0 et 24).

Les fonctions des exercices 2 et 3 seront mises dans un fichier `runtimes.c`

Exercice 2 Écrire une fonction qui prend en paramètre un nombre n , et un tableau, et remplit les n premiers éléments du tableau avec des dates aléatoires. **On supposera que le tableau de dates a été alloué à l'extérieur de la fonction.**

on utilisera la fonction `rand`, qui ne prend aucun paramètre, et qui retourne un nombre entier aléatoire compris entre 0 et `RAND_MAX`. (inclure la bibliothèque `stdlib.h` pour utiliser `RAND_MAX`) On fera un modulo pour obtenir un entier dans les domaines de valeurs convenant pour les champs de la structure `date`.

Pour ne pas obtenir toujours la même de nombres aléatoires, on initialisera le générateur aléatoire en fonction de l'heure de lancement du programme par l'instruction suivante mis au début du `main` :

```
srand(time(NULL));
```

Pour cela, il faut inclure la bibliothèque `time.h`.

Exercice 3 Faire une fonction d'affichage d'un tableau de dates. Faire une fonction de test qui saisit un entier n au clavier, qui alloue un tableau de n dates, qui appelle le générateur aléatoire de dates, et qui affiche le tableau. La fonction libère ensuite la mémoire. Tester avec un `main` dans un fichier `main.c`.

2 Implémentation des tris par ordre chronologique

On mettra les fonctions de tri dans un fichier `tris.c`.

Exercice 4 Écrire une fonction `DateCompare` qui prend en paramètre deux dates d_1 et d_2 , qui renvoie -1 si la date d_1 est antérieure à d_2 , qui renvoie $+1$ si la date d_1 est postérieure à la date d_2 , et qui renvoie 0 si les dates sont égales.

Exercice 5 Écrire une fonction de tri par sélection d'un tableau de dates. La fonction doit avoir pour prototype

```
void TriSelection(Date *tab, int n);
```

Exercice 6 Le but de cet exercice est de modifier la fonction de test de l'exercice 3 pour qu'elle puisse tester tous les algorithmes de tris. Modifier la fonction de test pour qu'elle prenne en paramètre un **pointeur de fonctions** (voir explications en annexe) de type

```
void (*fonction)(Date*, int);
```

La fonction de test doit donc maintenant avoir pour prototype :

```
void TesteTri(void (*fonction)(Date*, int))
```

On modifiera `TesteTri` la fonction pour qu'elle réalise les opérations suivantes :

- Saisie d'un entier n .
- Allocation d'un tableau de dates;
- Appel du générateur aléatoire;
- Affichage des données du tableau;
- Appel de la fonction de tri `fonction` passée en paramètre de `TesteTri`;
- Affichage des données triées.
- Libération de la mémoire.

Exercice 7 Écrire un programme principal qui appelle la fonction `TesteTri` avec pour paramètre la fonction `TriSelection`.

Exercice 8 Écrire une fonction de tri par insertion d'un tableau de dates. La fonction doit avoir pour prototype

```
void TriInsertion(Date *tab, int n);
```

Tester en modifiant le `main`.

Exercice 9 Écrire une fonction de tri par bulles d'un tableau de dates. La fonction doit avoir pour prototype

```
void TriBulle(Date *tab, int n);
```

Tester en modifiant le `main`.

Exercice 10 Écrire une fonction de tri rapide d'un tableau de dates. La fonction doit avoir pour prototype

```
void QuickSort(Date *tab, int n);
```

Tester en modifiant le `main`.

3 Comparaison des performances

Pour mesurer des temps de calcul, nous utiliserons la fonction `clock` de la bibliothèque `time.h`, qui donne des dates en temps *CPU*. La fonction `clock` a pour prototype :

```
clock_t clock(void);
```

Le type `clock_t` est un type entier spécialement défini dans `time.h`.

Principe de mesure d'un temps de calcul.

Pour mesurer le temps de calcul pris par une fonction, nous allons répéter l'appel à cette fonction un certain nombre de fois. Soit *nbIterations* le nombre de fois que nous faisons exécuter la fonction. (En pratique dans ce *TP*, nous prendrons *nbIterations* constant égal à 50.) Pour mesurer un temps d'exécution, nous aurons les principales étapes suivantes :

- Récupérer la date t_1 ;
- Répéter *nbIterations* fois l'algorithme (avec une éventuelle génération des données à chaque fois) ;
- Récupérer la date t_2 ;
- Retourner $(t_2 - t_1)/nbIterations$.

Exercice 11 Écrire une fonction de prototype

```
clock_t TesteRuntime(Date *tab, int n, void (*fonction)(Date*, int));
```

qui mesure le temps d'exécution d'une fonction de tri passée en paramètre. On supposera que le tableau a été alloué à l'extérieur de la fonction `TesteRuntime`.

Exercice 12 Écrire une fonction de prototype

```
void EnregistreRuntimes(char *nomfichier, int nmax,
                        void (*fonction)(Date*, int));
```

qui calcule des temps d'exécution pour des valeurs de n allant de 0 à `nmax` avec un pas de `nmax/20` (soit 20 valeurs de n différentes). On enregistrera les temps d'exécution dans un fichier (dont le nom est passé en paramètre) au format suivant :

```
n_0 t_0
n_1 t_1
n_2 t_2
n_3 t_3
...
n_19 t_19
```

où `n_i` est une valeur de n et `t_i` est le temps de calcul correspondant, séparés par **un seul espace**.

On allouera le tableau de dates une seule fois avec une taille `nmax` et on pensera à libérer la mémoire.

Exercice 13 Dans le `main`, appeler 4 fois la fonction `EnregistreRuntimes` avec pour paramètres les 4 fonctions de tri implémentées. On prendra `nbIterations = 50` et `n = 1000` pour la mise au point.

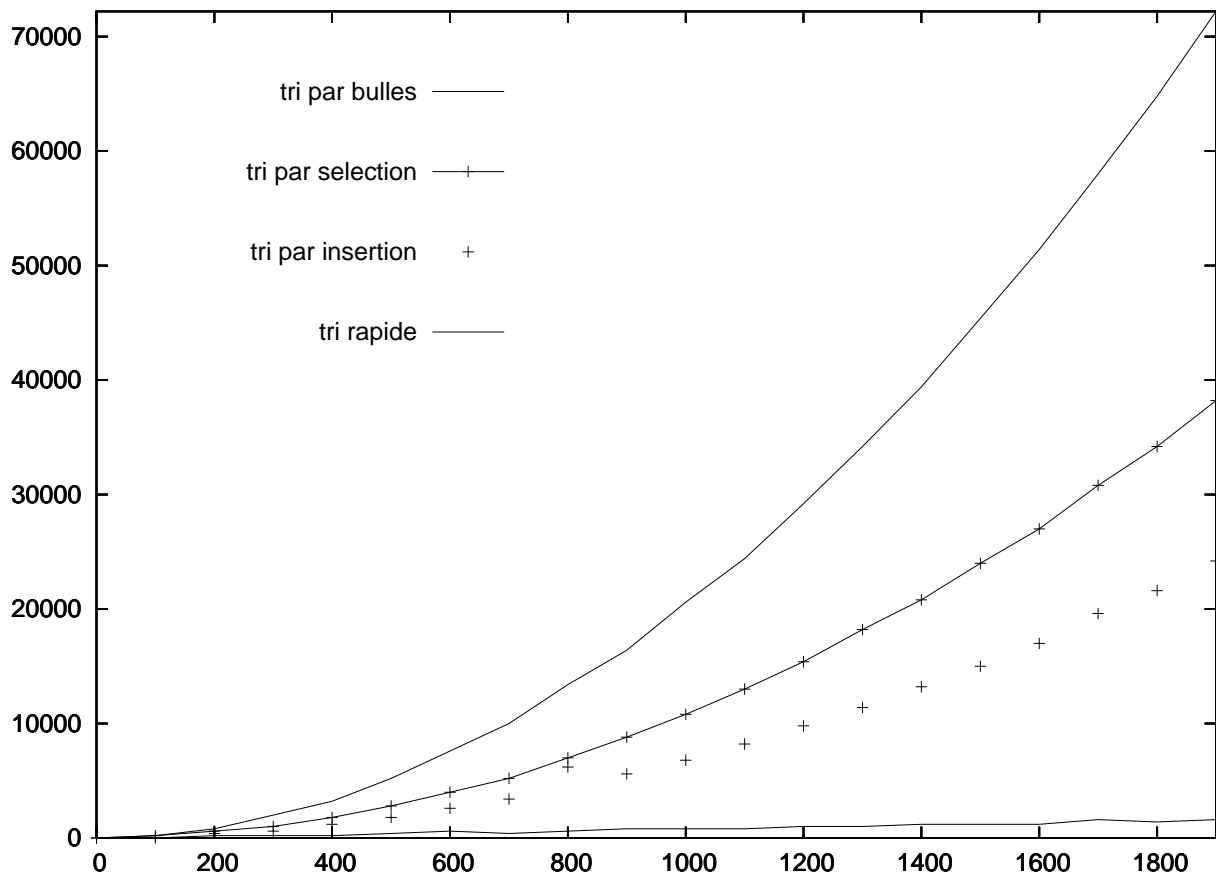
On générera les fichiers suivants (**respecter obligatoirement les noms de fichier!**) :

- `triselection.dat` pour le tri par sélection;
- `triinsertion.dat` pour le tri par insertion;
- `tribulle.dat` pour le tri par bulles;
- `trirapide.dat` pour le quicksort.

Pour visualiser, récupérez le script `visualiseur` :

```
$ cp /home/prof/remalgou/tris/visualiseur .
```

et exécutez ce script. Le graphique des temps de calcul apparaît. Une fois le programme au point, essayer avec $n = 2000$, $n = 3000$. Que constate-t-on ?

FIGURE 1: Résultats pour $n = 2000$

A Pointeurs de fonctions

Un pointeur de fonctions en C est une variable qui permet de désigner une fonction C . Comme n'importe quelle variable, on peut mettre un pointeur de fonctions soit en variable dans une fonction, soit en paramètre dans une fonction.

On déclare un pointeur de fonction comme un prototype de fonction, mais on ajoute une étoile (*) devant le nom de la fonction. Dans l'exemple suivant, on déclare dans le main un pointeur sur des fonctions qui prennent en paramètre un `int`, et un pointeur sur des fonctions qui retournent un `int`.

```
#include <stdio.h>

int SaisisEntier(void)
{
    int n;
    printf("Veuillez entrer un entier : ");
    scanf("%d", &n);
    return n;
}
```

```
void AfficheEntier(int n)
{
    printf("L'entier n vaut %d\n", n);
}

int main(void)
{
    void (*foncAff)(int); /* déclaration d'un pointeur foncAff */
    int (*foncSais)(void); /*déclaration d'un pointeur foncSais */
    int entier;

    foncSais = SaisisEntier; /* affectation d'une fonction */
    foncAff = AfficheEntier; /* affectation d'une fonction */

    entier = foncSais(); /* on exécute la fonction */
    foncAff(entier); /* on exécute la fonction */
    return 0;
}
```

Dans l'exemple suivant, la fonction est passée en paramètre à une autre fonction, puis exécutée.

```
#include <stdio.h>

int SaisisEntier(void)
{
    int n;
    printf("Veuillez entrer un entier : ");
    scanf("%d", &n);
    getchar();
    return n;
}

void AfficheDecimal(int n)
{
    printf("L'entier n vaut %d\n", n);
}

void AfficheHexa(int n)
{
    printf("L'entier n vaut %x\n", n);
}

void ExecAffiche(void (*foncAff)(int), int n)
{
    foncAff(n); /* exécution du paramètre */
}
```

```
int main(void)
{
    int (*foncSais)(void); /*déclaration d'un pointeur foncSais */
    int entier;
    char rep;

    foncSais = SaisisEntier; /* affectation d'une fonction */
    entier = foncSais(); /* on exécute la fonction */

    puts("Voulez-vous afficher l'entier n en décimal (d) ou en hexa (x) ?");
    rep = getchar();
    /* passage de la fonction en paramètre : */
    if (rep == 'd')
        ExecAffiche(AfficheDecimal, entier);
    if (rep == 'x')
        ExecAffiche(AfficheHexa, entier);

    return 0;
}
```