Cours de programmation séquentielle

Tris divers

1 Buts

- Utilisation de tableaux unidimensionnels.
- Implémentations d'algorithmes de tris.
- Utilisation de `make`.

2 Énoncé

L'objectif de ce travail pratique est d'implémenter les tris vus en cours. En particulier, vous devez écrire les codes en C des tris par base et par fusion, puis si le temps le permet le tri par sélection et le tri rapide. Les pseudo-codes de ces tris se trouve dans les slides du cours. Pour vous aider, on vous fournit un squelette que vous devez compléter.

Ce squelette (à télécharger ici) contient:

1. Une fonction `main()` qui lit la ligne de commande, génère un tableau et permet de sélectionner son algorithme tri.
2. Les signatures des fonctions des divers tris `radix_sort()`, `merge_sort()`, `selection_sort()`, et `quick_sort()`. Ces fonctions prennent en argument un tableau et sa taille et modifie le tableau durant leurs exécutions.
3. L'affichage des tableaux `print()` et la vérification si le tri a réussi.

Vous aurez plusieurs tâches.

1. Diviser le code en différents fichiers et écrire un `Makefile`.
2. Ajouter des variables `size`, `seed`, `sorting_algo_number` dans le code et lire la la ligne de commande pour faire en sorte que le code compile et s'exécute.
3. Écrire le code C correspondant au tri par base et au tri par fusion.
4. Écrire une fonction permettant de vérifier si le tri a réussi.
5. En option ajouter le tri rapide et le tri par sélection.

2.1 Remarque

2.1.1 Le `k`-ème bit d'un entier `n`

Pour le tri par base, vous devez implémenter une fonction qui retourne le `k`-ème bit d'un nombre `n`. Pour ce faire, vous pouvez utiliser la syntaxe suivante:

```
(n >> k) & 1
```

où on commence par décaler tous les bits de `n` de `k` bits vers la droite, puis on fait un `&`-logique bit à bit avec `1` (on fait un masque). Ainsi pour un nombre 4-bits

```
n = 13 // 1101
k = 2
n >> k == 0011 // 3 en décimal
0011 & 0001 == 0001 // soit 1 en décimale
```

2.1.2 L'échange de tableaux statiques

Certains tris à implémenter, vous devez échanger les pointeurs vers des tableaux statiques.

En effet, on pourrait se dire que le code suivant

```
int a[] = {1, 2, 3, 4};
int b[] = {5, 6, 7, 8};
swap(&a, &b);
// ici en fait a est toujours le tableau 1, 2, 3, 4
// et b est toujours 5, 6, 7, 8
```

ferait pointer le tableau `a` sur les données de `b` et vice-versa. Hors ce n'est absolument pas le cas car les tableaux statiques **ne sont pas que des pointeurs**. Quand on passe un tableau statique en argument à une fonction, il est effectivement transformé pointeur. Mais cela ne suffit pas.

L'impossibilité de faire un échange de tableaux statique est dûe à l'impossibilité en `C` d'assigner un tableau statique à un autre tableau statique. Ainsi,

```
int a[3] = {1, 2, 3};
int b[3] = a;
```

n'est pas une syntaxe valide et c'est ce qu'on essaie de faire dans le code précédent.

Pour pouvoir effectuer le travail qui vous est demandé, il faut soit faire de l'allocation dynamique (ce qu'on a pas encore fait en cours) ou alors tricher un peu. En fait, pour réussir à faire un échange de pointeurs de mémoire, il faut définir deux nouvelles variables, qui sont des vrais pointeurs (elles). En transformant le code ci-dessus en

```
int a[] = {1, 2, 3, 4};
int b[] = {5, 6, 7, 8};
int *c = a;
int *d = b;
swap_ptr(&c, &d);
// ici en fait a est toujours le tableau 1, 2, 3, 4
// et b est toujours 5, 6, 7, 8
// mais c pointe sur 5, 6, 7, 8
// et d pointe sur 1, 2, 3, 4
```

on a l'effet voulu. On échange effectivement les pointeurs vers les données voulues.

Attention Si vous n'avez rien compris à ce qui précède, je ne suis pas choqué.

Vous pouvez à la place de faire l'implémentation ci-dessus, faire un échange des valeurs se trouvant à l'intérieurs des tableaux plutôt que de tenter d'échanger des pointeurs. Cela ralentira un peu l'exécution, mais c'est quand même plus compréhensible.

2.1.3 La copie du tableau final

Quand on échange les pointeurs vers des tableaux statiques dans la fonction de tri, un problème peut survenir. Prenons le code suivant très simple

```
void times_two(int size, int tab[], int tmp[]) {
    for (int i = 0; i < size; ++i) {
        tmp[i] = 2 * tab[i];
    }
}
void do_things(int size, int tab[]) {
    int *tab_ptr = tab;
    int tmp[size];
    int *tmp_ptr = tmp;
    for (int i = 0; i < 3; ++i) {
        times_two(size, tab, tmp)
        swap(&tab_ptr, &tmp_ptr);
    }
}
int size = 4;
int a[size] = {1, 2, 3, 4};
do_things(size, a);
```

on pourrait se dire que `a` pointerait vers `{8, 16, 24, 32}`. Il n'en est rien. En effet, dans la fonction `do_things()`, le pointeur `tab` n'est qu'une **copie** du pointeur de `a`. Ainsi, le pointeur vers `a` n'est jamais vraiment modifié, seul `tab` l'est et il est détruit à la fin de `do_things()`. Ainsi, comme on échange trois fois le pointeur, on ne modifie qu'une seule fois les données qui sont pointées par `a`, la valeur contenue dans `a` sera ainsi `{4, 8, 12, 16}`. Pour finir l'implémentation de la fonction, il faut encore copier les données dans `tab` si le besoin s'en fait sentir (si on allait jusqu'à `4` dans la boucle `for` de `do_things()` on aurait pas besoin de faire de copie car la dernière modification est dans les données pointées par `a`).

```
void do_things(int size, int tab[]) {
    int *tab_ptr = tab;
    int tmp[size];
    int *tmp_ptr = tmp;
    for (int i = 0; i < 3; ++i) {
        times_two(size, tab, tmp)
        swap(&tab_ptr, &tmp_ptr);
    }
    if // trouver la bonne condition
    {
        copy(size, tmp, tab); // copier les valeur de tmp dans tab
    }
```

```
}
```