Cours de programmation séquentielle

Tris par fusion, base, et sélection

1 Buts

- Utilisation de tableaux unidimensionnels.
- Implémentations d'algorithmes de tris.
- Manipulation de pointeurs de tableaux.

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. Nous avons déjà vu l'algorithme du tri par sélection. Les algorithmes des tris par fusion et par base seront décrit ci-dessous. 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()`, et `selection_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 différentes fonctions.
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 par sélection.

3 Description des tris par fustion et par base

3.1 Tri par fusion (merge sort)

Le tri par fusion (en anglais, merge sort) est un tri par comparaison. Il s'appuie sur l'opération de *fusion* qui produit une liste triée à partir de deux listes triées. Ce tri procède itérativement en triant d'abord les paires de nombres, puis les

groupes de 4 nombres, ensuite de 8, et ainsi de suite jusqu'à obtenir un tableau trié. A noter qu'il faut attention au cas général où le nombre d'éléments n'est pas une puissance de 2.

Pour son implémentation, le tri par fusion nécessite d'utiliser une zone temporaire de stockage des données de taille égale à celle de la liste de nombres à trier. On considère le cas du tri d'une liste de nombres entiers stockés dans un tableau.

3.1.1 Principe de l'algorithme

Soit *size* la taille du tableau à trier.

- Pour l'étape $i = 0$ à $\lceil \log_2(size) \rceil - 1$, on répète:
  - Appliquer l'opération de fusion aux paires de sous-tableaux successifs de taille 2^i (ou moins pour l'extrémité)

A noter qu'à l'étape i, les sous-tableaux de taille 2^i sont triés et que la dernière paire de sous-tableaux peut être incomplète (vide ou avec moins que 2^i éléments).

La fonction de fusion (merge) produit à partir de deux tableaux triés, un 3ème tableau trié qui contient les éléments de ces deux tableaux. Si un des tableaux est vide, le résultat de la fusion est identique à l'autre tableau.

Elle procède de la manière suivante:

- Parcourir simultanément les deux tableaux jusqu'à atteindre la fin de l'un des deux
  - Comparer l'élément courant des deux tableaux
  - Ecrire le plus petit élément dans le tableau résultat
  - Avancer de 1 dans les tableaux du plus petit élément et résultat
- Copier les éléments du tableau restant dans le tableau résultat

3.1.2 Illustration sur un exemple

Soit la liste de nombres entiers stockés dans un tableau de taille 9:

0	1	2	3	4	5	6	7	8
5	**-5**	1	**6**	4	**-6**	2	**-9**	2

0. On fusionne les éléments successifs (ce qui revient à les mettre dans l'ordre) et on obtient:

0	1	2	3	4	5	6	7	8
-5	5	**1**	**6**	-6	4	**-9**	**2**	2

1. On fusionne les paires successives (sous-tableaux de taille 2) et on a:

0	1	2	3	4	5	6	7	8
-5	1	5	6	**-9**	**-6**	**2**	**4**	2

2. On fusionne les sous-tableaux successifs de taille 4 et on a:

0	1	2	3	4	5	6	7	8
-9	-6	-5	1	2	4	5	6	**2**

3. On fusionne les sous-tableaux successifs de taille 8 et on obtient le résultat final:

0	1	2	3	4	5	6	7	8
-9	-6	-5	1	2	2	4	5	6

3.1.3 Complexité

L'algorithme présenté précédemment nécessite un certain nombre d'opérations lié à la taille N du tableau.

Il y a essentiellement $\log_2(N)$ étapes.

A chaque étape, le tableau est parcouru une fois avec un nombre constant d'opérations effectué pour chacune des cases du tableau. En effet, l'opération de fusion implique de ne parcourir qu'une seule fois chacun des deux tableaux qu'on fusionne dans un 3ème tableau.

Ainsi, la complexité du tri par fusion est $\mathcal{O}(N \cdot \log_2(N)$.

3.2 Tri par base (radix sort)

Le tri par base (en anglais, radix sort) est un tri qui n'utilise pas la notion de comparaison, mais celle de classement successif dans des alvéoles (buckets).

On considère le cas du tri d'une liste de nombres entiers stockés dans un tableau. Dans un premier temps, on applique un décalage aux éléments de la liste pour se ramener au cas où le plus petit élément est 0. On considère ensuite la représentation binaire de ces nombres.

3.2.1 Principe de l'algorithme

1. On considère le bit le moins significatif.
2. On parcourt une 1ère fois le tableau et on place à la suite dans un 2ème tableau les éléments dont le bit est 0; puis on reparcourt le tableau une deuxième fois et on met à la suite les éléments dont le 1er bit est 1 (le tableaux a donc tous les éléments avec le premier bit à 0 au début et tous ceux à 1 ensuite).
3. On répète l'étape 2 en regardant le bit suivant et en permutant le rôle des deux tableaux.

On utilise donc deux tableaux pour réaliser ce tri. A noter qu'à chaque étape, l'ordre des éléments dont le bit est à 0 (respectivement à 1) reste identique dans le 2ème tableau par rapport au 1er tableau.

3.2.2 Illustration sur un exemple

Soit la liste de nombres entiers:

0	1	2	3	4	5	6	7	8
5	-5	1	6	4	-6	2	-9	2

Le plus petit élément est -9. On commence donc par décaler les valeurs de 9.

0	1	2	3	4	5	6	7	8
14	4	10	15	13	3	11	0	11

Ecrivons à présent les éléments en représentation binaire. Comme la valeur maximale est 15, on a besoin de 4 bits.

0	1	2	3	4	5	6	7	8
111**0**	010**0**	101**0**	111**1**	110**1**	001**1**	101**1**	000**0**	101**1**

On applique à présent l'algorithme.

1. On considère le bit de poids faible et on obtient le tableau:

0	1	2	3	4	5	6	7	8
11**1**0	01**0**0	10**1**0	00**0**0	11**1**1	11**0**1	00**1**1	10**1**1	10**1**1

2. On passe au 2ème bit et on obtient le tableau:

0	1	2	3	4	5	6	7	8
0**1**00	0**0**00	1**1**01	1**1**10	1**0**10	1**1**11	0**0**11	1**0**11	1**0**11

3. On passe au 3ème bit et on obtient le tableau:

0	1	2	3	4	5	6	7	8
0000	**1**010	**0**011	**1**011	**1**011	**0**100	**1**101	**1**110	**1**111

4. On passe au dernier bit et on obtient le tableau final:

0	1	2	3	4	5	6	7	8
0000	0011	0100	1010	1011	1011	1101	1110	1111

En revenant à la représentation décimale, on a le tableau trié:

0	1	2	3	4	5	6	7	8
0	3	4	10	11	11	13	14	15

Pour revenir aux valeurs intiale, il faut décaler de 9 dans l'autre sens.

0	1	2	3	4	5	6	7	8
-9	-6	-5	1	2	2	4	5	6

3.2.3 Complexité

L'algorithme implémenté précédemment nécessite un certain nombre d'opérations lié à la taille du tableau.

Voici une liste de parcours utilitaires de tableau:

1. Recherche de la valeur minimum `val_min`
2. Recherche de la valeur maximum `val_max`
3. Décalage des valeurs dans l'intervalle `0..val_max-val_min`
4. Décalage inverse pour revenir dans l'intervalle `val_min..val_max`
5. Copie éventuelle du tableau temporaire dans le tableau originel

On a donc un nombre de parcours fixe (4 ou 5) qui se font en $\mathcal{O}(N)$ où N est la taille du tableau.

La partie du tri à proprement parler est une boucle sur le nombre de bits b de `val_min..val_max`.

A chaque passage à travers la boucle, on parcourt 2 fois le tableau: la 1ère fois pour s'occuper des éléments dont le bit courant à 0; la 2ème pour ceux dont le bit courant est à 1.

A noter que le nombre d'opérations est de l'ordre de b pour la lecture d'un bit et constant pour la fonction d'échange de tableaux.

Ainsi, la complexité du tri par base est $\mathcal{O}(b \cdot N)$.

4 Remarques

4.1 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
```

4.2 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.

4.3 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
    }
}
```