

Tri par base et par fusion

Programmation séquentielle en C, 2026-2027

Orestis Malaspinas (A401)

2026-10-12

Informatique et Systèmes de Communication, HEPIA

Le tri par base (radix sort)

Tri par base (radix sort)

- N'utilise pas la notion de comparaison, mais celle de classement successif dans des catégories (alvéoles).
- Pour simplifier
 - Tri de nombres entiers dans un tableau.
 - On considère que des nombres ≥ 0 (sans perte de généralité).
 - On considère ensuite la représentation binaire de ces nombres.

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 parcourt le tableau une 2ème fois et on place à la suite les éléments dont le bit est 1.
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. À 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.

Illustration sur un exemple (1/6)

Soit la liste de nombres entiers :

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

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

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

Illustration sur un exemple (2/6)

- Écrivons les éléments en représentation binaire.
- La valeur maximale est 15, on a besoin de 4 bits.

Indices	0	1	2	3	4	5	6	7	8
Valeurs	14	4	10	15	13	3	11	0	11
Binaire	1110	0100	1010	1111	1101	0011	1011	0000	1011

Illustration sur un exemple (3/6)

- On considère le bit de poids faible

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

Illustration sur un exemple (3/6)

- On considère le bit de poids faible

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

- On obtient le tableau :

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

Illustration sur un exemple (4/6)

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

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

Illustration sur un exemple (4/6)

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

Indices	0	1	2	3	4	5	6	7	8
Binaire	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
Binaire	0 1 0 0	0 0 0 0	1 1 0 1	1 1 1 0	1 0 1 0	1 1 1 1	0 0 1 1	1 0 1 1	1 0 1 1

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

Indices	0	1	2	3	4	5	6	7	8
Binaire	0 1 0 0	0 0 0 0	1 1 01	1 1 10	1 0 10	1 1 11	0 0 1 1	1 0 11	1 0 11
Binaire	0 0 0 0	1 0 1 0	0 0 1 1	1 0 1 1	1 0 1 1	0 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1

Illustration sur un exemple (5/6)

- On passe au 4ème (dernier) bit et on obtient le tableau final :

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

Illustration sur un exemple (5/6)

- On passe au 4ème (dernier) bit et on obtient le tableau final :

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

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

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

Illustration sur un exemple (6/6)

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

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

- Et alors?

Illustration sur un exemple (6/6)

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

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

- Et alors?
- Et alors rien. C'est fini.

Pseudo-code

```
rien radix_sort(entier taille, entier tab[taille]):  
# initialisation  
    entier val_min = valeur_min(taille, tab)  
    entier val_max = valeur_max(taille, tab)  
    decaler(taille, tab, val_min)  
    entier nb_bits = nombre_de_bits(val_max - val_min)  
# algo  
    entier tab_tmp[taille]  
    pour pos de 0 à nb_bits-1:  
        alveole_0(taille, tab, tab_tmp, pos) # 0 -> taille  
        alveole_1(taille, tab, tab_tmp, pos) # taille -> 0  
        echanger(tab, tab_tmp)  
# post-traitement  
    decaler(taille, tab, -val_min)
```

Un peu plus de détails (1/2)

La fonction `decaler()`

```
rien decaler(entier taille, entier tab[taille], entier val):  
    pour i de 0 à taille-1:  
        tab[i] -= val
```

Un peu plus de détails (1/2)

La fonction `decaler()`

```
rien decaler(entier taille, entier tab[taille], entier val):  
    pour i de 0 à taille-1:  
        tab[i] -= val
```

La fonction `echanger()`

```
rien echanger(entier tab[], entier tab_tmp[]):  
# échanger les tableaux (sans copier les valeurs)
```

- Attention : si le nombre d'échanges est impair, le résultat se trouve dans la mémoire de `tab_tmp`, il faut le recopier dans `tab`.

Un peu plus de détails (2/2)

La fonction `alveole_0()`

```
rien alveole_0(entier taille, entier tab[taille],
               entier tab_tmp[taille], entier pos):
    entier k = 0
    pour i de 0 à taille-1:
        si bit(tab[i], pos) == 0:
            tab_tmp[k] = tab[i]
            k = k + 1
```

Un peu plus de détails (2/2)

La fonction `alveole_0()`

```
rien alveole_0(entier taille, entier tab[taille],
               entier tab_tmp[taille], entier pos):
    entier k = 0
    pour i de 0 à taille-1:
        si bit(tab[i], pos) == 0:
            tab_tmp[k] = tab[i]
            k = k + 1
```

La fonction `alveole_1()`

```
rien alveole_1(entier taille, entier tab[taille],
               entier tab_tmp[taille], entier pos):
    # pareil que alveole_0 mais pour les bits à 1
    # et dans l'autre sens:
    # i et k partent de taille-1 et décroissent
```

Le tri par fusion (merge sort)

Tri par fusion (merge sort)

- Tri par comparaison.
- Idée : deux listes triées sont fusionnées pour donner une liste triée plus longue.
- Itérativement, on trie 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.

Principe de l'algorithme

- Soit `taille` la taille du tableau à trier.
- Pour `i = 0` à `entier(log2(taille))` :
 - Fusion des paires de sous-tableaux successifs de taille `2**i` (ou moins pour l'extrémité)

Principe de l'algorithme

- Soit `taille` la taille du tableau à trier.
- Pour `i = 0` à `entier(log2(taille))` :
 - Fusion des paires de sous-tableaux successifs de taille `2**i` (ou moins pour l'extrémité)
- Remarques :
 - Au début de l'étape `i`, les sous-tableaux de taille `2**i` sont triés.
 - La dernière paire de sous-tableaux peut être incomplète (vide ou avec moins de `2**i` éléments).

Exemple de tri par fusion

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

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

Exemple de tri par fusion

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

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

- Fusions successives (en rouge le sous-tableau de gauche, en vert celui de droite) :

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

Pseudo-code

```
rien tri_fusion(entier taille, entier tab[taille]):  
    entier tab_tmp[taille]  
    entier nb_etapes = entier(log2(taille)) + 1  
    pour etape de 0 à nb_etapes - 1:  
        entier gauche = 0  
        entier t_tranche = 2**etape  
        tant que (gauche < taille):  
            entier milieu = min(gauche+t_tranche, taille)  
            entier droite = min(gauche+2*t_tranche, taille)  
            fusion(  
                tab[gauche..milieu-1],  
                tab[milieu..droite-1],  
                tab_tmp[gauche..droite-1])  
            # bornes incluses  
            gauche += 2*t_tranche  
    echanger(tab, tab_tmp)
```

Une idée ?

Une idée?

- Parcourir les deux tableaux jusqu'à atteindre la fin d'un des deux
 - Comparer l'élément courant des 2 tableaux
 - Écrire le plus petit élément dans le tableau résultat
 - Avancer de 1 dans le tableau du plus petit élément et dans le tableau résultat
- Copier les éléments du tableau restant dans le tableau résultat

La fonction de fusion (pseudo-code)

```
# hyp: tab_g et tab_d sont triés
rien fusion(entier tab_g[], entier tab_d[], entier res[]):
    entier g = taille(tab_g)
    entier d = taille(tab_d)
    entier i_g = 0, i_d = 0
    pour i de 0 à g + d - 1:
        si i_g < g et i_d < d:
            si tab_g[i_g] <= tab_d[i_d]:
                res[i] = tab_g[i_g]
                i_g = i_g + 1
            sinon:
                res[i] = tab_d[i_d]
                i_d = i_d + 1
        sinon si i_g < g:
            res[i] = tab_g[i_g]
            i_g = i_g + 1
        sinon si i_d < d:
            res[i] = tab_d[i_d]
            i_d = i_d + 1
```