TP 9 : Tri radix

Petit TP de retour sur la programmation en C et OCaml. On va utiliser les méthodes de tri incluses en OCaml et en C. On laisse de côté les exercices et questions marquées d'un ★ dans un premier temps.

I. Sur les algorithmes de tri intégrés à C / OCaml

Pour l'instant, on a implémenté des algorithmes de tri en OCaml et C en ordonnant les éléments de la liste en utilisant la relation d'ordre <= intégrée aux langages. Si cet ordre est relativement naturel pour certains types (`int`, `float` en OCaml, l'ordre lexicographique sur les types `string list array` en OCaml, `int`, `float` en C), il n'est pas toujours adapté. On peut vouloir, par exemple, trier une liste d'étudiants représentés par un enregistrement selon leur date de naissance : il n'y a alors aucune garantie que la relation d'ordre implémentée par défaut sur les enregistrements en OCaml n'ordonne les étudiants selon leur date de naissance (et il n'y a pas de tel ordre en C). Les fonctions de tri en OCaml et C demandent donc de spécifier une fonction de comparaison.

Exercice 1 – Tri en OCaml

OCaml propose plusieurs fonctions pour trier des listes et tableaux. Les deux principales sont :

- `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list`
- `Array.sort : ('a -> 'a -> int) -> 'a array -> unit`
- Dans les deux cas, le premier argument est une fonction de comparaison. Elle définit un ordre $<$ sur le type `'a` en renvoyant un entier négatif si le premier argument est plus petit que le second, un entier positif si le premier argument est plus grand que le second, et 0 si les deux arguments sont égaux.
- La fonction `List.sort` renvoie la liste en entrée triée selon l'ordre défini par la fonction de comparaison.
- La fonction `Array.sort` trie le tableau en entrée selon l'ordre défini par la fonction de comparaison *en place*, c'est-à-dire en modifiant le tableau en entrée.

▶ **Question 1** Que renvoie `List.sort (fun x y -> x - y) [7; 1; 3; 2; 0]`? Que renvoie `List.sort (fun x y -> y - x) [7; 1; 3; 2; 0]`?

▶ **Question 2** En utilisant `List.sort`, écrire une fonction `tri_1 : int list -> int list` triant la liste en entrée par valeur absolue croissante. *On pourra utiliser la fonction* `abs : int -> int`.

Dès qu'elle devient un peu subtile, on écrit cette fonction de comparaison séparément.

★ **Question 3** De même, écrire une fonction `tri_2 : (int * int) list -> (int * int) list` triant la liste en entrée selon le premier élément des couples par ordre de valeur absolue croissante, et selon le second élément des couples par ordre décroissant en cas d'égalité du premier élément.

Remarque 1 – Fonction `compare`

La fonction `compare : 'a -> 'a -> int` est une fonction de comparaison sur le type `'a` correspondant[a] à l'ordre définit par les opérateurs `(<)` et `(>)`.

a. presque, il y a une subtilité sur les flottants.

Pour rappel, la fonction `List.length` est de complexité linéaire en la taille de la liste.

★ **Question 4** Proposer une fonction `tri_4 : 'a list list -> 'a list list` triant une liste de listes selon la longueur des listes dans l'ordre croissant. On pourra utiliser la fonction `List.sort` avec une fonction de comparaison adaptée.

▶ **Question 5** Quelle est sa complexité ? Proposer une alternative de complexité temporelle en $\mathcal{O}\left(n \log n + \sum_{k=0}^{n-1} |l_i|\right)$ avec $l = [l_0; \dots; l_{n-1}]$ la liste de listes en entrée.

Remarque 2 – ★ Tri en C

La bibliothèque standard C `stdlib.h` propose une fonction de tri `qsort` similaire :

C
```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
           // compar est ici un pointeur vers une fonction
           // de comparaison. On peut accéder au pointeur d'une
            ↪ fonction
           // avec le nom de la fonction, par exemple :
           // int (*compar)(const void *, const void *) = compare;
```

Ici, le premier argument est un pointeur vers le premier élément du tableau à trier, le deuxième argument est le nombre d'éléments du tableau, le troisième argument est la taille (en octet) d'un élément du tableau et le quatrième argument est une fonction de comparaison. Cette fonction de comparaison a le même rôle que celle d'OCaml. La fonction `qsort` trie le tableau en entrée selon l'ordre défini par la fonction de comparaison *en place*, c'est-à-dire en modifiant le tableau en entrée [a]. Attention : **la fonction de comparaison compare des pointeurs vers les éléments du tableau, et non les éléments eux-mêmes**. Ces pointeurs sont des pointeurs génériques : des pointeurs vers `void`, c'est-à-dire des pointeurs vers n'importe quel type. On peut donc les convertir en pointeurs vers n'importe quel type en utilisant la syntaxe de conversion de type habituelle.

a. Et non, le fait de pouvoir écrire des fonctions prenant des fonctions en entrée ne fait pas de C un langage fonctionnel.

★ **Question 6** Écrire une fonction `void tri_3(int* tab, int n)` triant un tableau d'entiers par valeur absolue croissante.

★ **Question 7** Écrire une fonction `void tri_4(double** tab, int n)` triant un tableau de pointeur vers des flottants par valeur absolue croissante de la valeur pointée. *La fonction* `double fabs(double x)` *renvoie la valeur absolue de* `x`.

★ **Question 8** Écrire une fonction `void tri_5(complexe* tab, int n)` triant un tableau de complexes par module décroissant.

Exercice 2 – Algorithme de tri stable

Définition 3 – Algorithme de tri stable

Un algorithme de tri est dit *stable* si l'ordre relatif des éléments égaux pour la relation d'ordre utilisée est conservé.

Par exemple, si on trie la liste `[(1, 2); (2, 1); (1, 1)]` selon le premier élément des couples seulement, on obtient `[(1, 2); (1, 1); (2, 1)]` si l'algorithme est stable. En appliquant `List.sort compare` sur cette liste, on obtient `[(1, 1); (1, 2); (2, 1)]` : l'ordre relatif des éléments égaux pour la relation d'ordre n'est pas conservé, l'algorithme n'est pas stable.

Remarque 4

— Les algorithmes de tri par insertion, par sélection et par partition-fusion sont stables (si l'on prend garde aux conditions des cas de base).
— Dans un algorithme de tri stable, il n'y a qu'une seule sortie possible pour une entrée donnée.
— Certains algorithmes ne sont pas stables : par exemple, l'algorithme de tri rapide n'est pas stable.

Remarque 5

OCaml propose plusieurs fonctions de tri selon vos besoins : `List.stable_sort` est garanti stable, alors que `List.fast_sort` est la fonction la plus rapide entre `List.sort` et `List.stable_sort`[a]. Les mêmes trois fonctions existent dans le module `Array` (sauf que cette fois-ci, la fonction `Array.sort` n'est pas stable puisqu'elle est implémentée par le tri par tas, alors que `Array.stable_sort` est implémentée par un tri fusion).
En C, la fonction `qsort` n'est pas stable.

a. Ces fonctions existent surtout pour des raisons de compatibilité. En vérité, les trois sont implémentées par le tri fusion, avec un *twist* lui permettant d'être de complexité spatiale constante pour le tas et logarithmique sur la pile.

On souhaite trier un tableau de couples d'entiers selon :
— D'abord, leur somme, par ordre croissant.
— En cas d'égalité de la somme, selon leur premier élément, par ordre croissant.

★ **Question 1** Écrire une fonction `compare_somme : (int * int) -> (int * int) -> int` comparant deux couples selon la somme de leurs éléments. De même, écrire une fonction `compare_premier : (int * int) -> (int * int) -> int` comparant deux couples selon leur premier élément.

★ **Question 2** Écrire une fonction `tri_stable : (int * int) array -> (int * int) array` triant le tableau en entrée selon l'ordre défini ci-dessus, en utilisant deux appels à la fonction `Array.stable_sort`.

★ **Question 3** Sans utiliser la fonction `List/Array.stable_sort`, écrire une fonction `stabiliser_liste : ('a -> 'a -> int) -> 'a list -> 'a list` triant la liste en entrée de manière stable selon l'ordre défini par la fonction de comparaison.

★ **Question 4** De même, écrire une fonction `stabiliser_tableau : ('a -> 'a -> int) -> 'a array -> unit` triant de manière stable le tableau en entrée selon l'ordre défini par la fonction de comparaison.

★ **Question 5** Quel changement observez-vous sur la complexité de votre algorithme de tri ?

Exercice 3 – ★ Tri radix (ou tri par base)

Ce TP est libre, il est à travailler si vous en avez le temps et envie. Une correction sera disponible sur le cahier de prépas un jour. Il nécessite notamment de connaître la représentation des entiers non signés bornés en mémoire, un petit rappel étant prévu si besoin.

. Principe Supposons que l'on souhaite trier un tableau de n éléments dans $[\![0, 999]\!]$. L'idée du tri radix (en base 10 ici) est d'effectuer le tri en trois « passes » :
— On trie les éléments selon leur chiffre des unités, en utilisant un tri stable.

— On trie les éléments selon leur chiffre des dizaines, en utilisant un tri stable.
— On trie les éléments selon leur chiffre des centaines, en utilisant un tri stable.

▶ **Question 1** Appliquer cette stratégie sur le tableau suivant :

```
{123, 456, 789, 321, 654, 987, 634, 312, 124, 458, 799, 199}
```

▶ **Question 2** Pourquoi est-il nécessaire d'utiliser un tri stable ?

Pour que cette approche soit efficace, il faut donc simultanément :
— minimiser le nombre de passes ;
— optimiser chaque passe en la rendant la plus efficace possible.

Ces deux points sont liés : par exemple, si l'on voulait diminuer le nombre de passes, on ne pourrait n'en faire qu'une seule en triant les éléments selon leur chiffre des centaines, dizaines et unités. Au contraire, en se plaçant en base 2, on peut trier des éléments entre 0 et $2^{10} - 1 = 1023$ en 10 passes. Pour s'aider de la représentation des entiers en mémoire, on utilisera des bases de la forme 2^p.

Remarque 6

Pour rappel, on peut représenter des entiers non signés $n \in [\![0, 2^w - 1]\!]$ par w bits $b_{w-1}, \dots, b_0 \in \{0, 1\}$ de la manière suivante :

$$n = \sum_{k=0}^{w-1} b_k 2^k$$

On peut donc représenter n par un tableau de w bits, où le bit d'indice k est égal à b_k. Les langages C et OCaml permettent de manipuler directement la représentation binaire des entiers. En C, on peut utiliser les opérateurs << et >> pour décaler les bits d'un entier vers la gauche ou vers la droite (les bits décalés au-delà de la représentation sont ignorés). Par exemple, `1 << 3` vaut 8 et `8 >> 3` vaut 1. De même, puisque $7 = 4 + 2 + 1$, `7 >> 1` vaut $3 = 2 + 1$ et `7 >> 2` vaut 1.

Fixons une base $base = 2^p$ et considérons que l'on souhaite trier un tableau `in` de n entiers non signés sur w bits, dans l'intervalle $[\![0, 2^w - 1]\!]$.

▶ **Question 3** Déterminer le nombre de passes nécessaires dans ce cas, en fonction de p et w. Dans la suite, on note ce nombre $radix$.

On considère maintenant que p divise w. On va maintenant décrire comment effectuer une passe du tri radix.

Pour chaque passe, on va utiliser une variante du tri comptage, dont le principe est comme suis :
— En entrée, on a un tableau `in` de n entiers non signés sur w bits.
— On crée un tableau `out` de n entiers non signés sur w bits. L'objectif de la passe est de remplir ce tableau des valeurs de `in` triées par valeur croissante du chiffre de poids $base^k$ (où k est le numéro de la passe et $base = 2^p$).
— On crée un tableau `hist` de taille $radix$ tel que `hist[i]` contienne le nombre d'éléments de `in` ayant le chiffre de poids $base^k$ égal à i.
— On en déduit un tableau `sums` de taille $radix$ tel que `sums[i]` contienne le nombre d'éléments de `in` ayant le chiffre de poids $base^k$ strictement inférieur à i.
— On remplit `out` en parcourant `in` : pour chaque élément `in[i]`, on regarde son chiffre de poids $base^k$ et on le place dans `out` à l'indice `sums[in[i]]`, puis on incrémente `sums[in[i]]`.

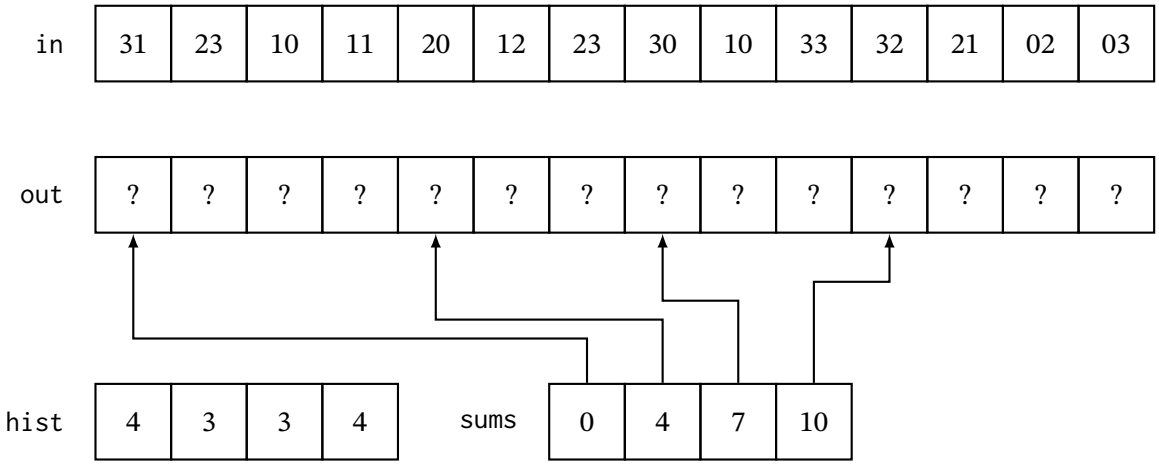

FIGURE 1. – Illustration de la première passe après calcul des tableaux `hist` et `sums`

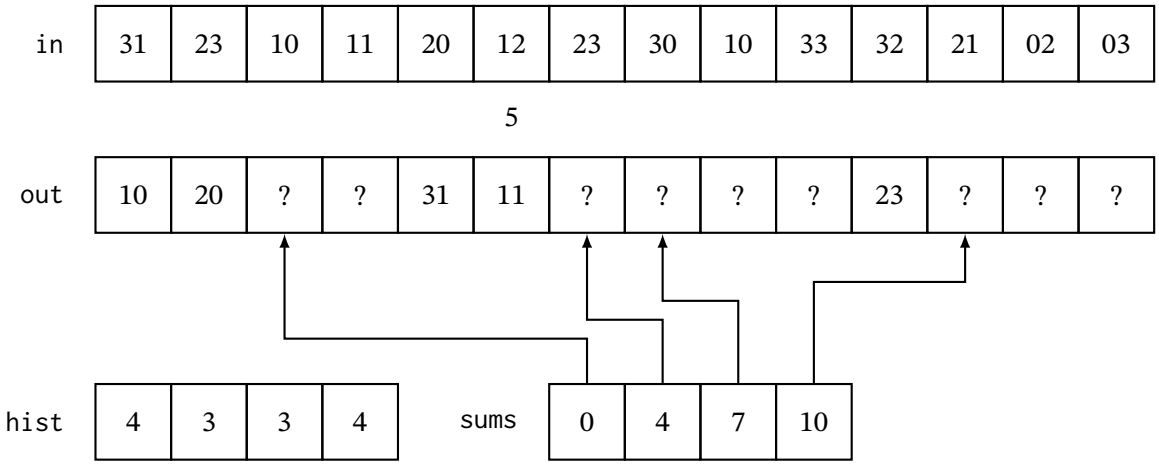

FIGURE 2. – Illustration de la première passe après 5 itérations du remplissage de `out`

Implémentons le tri radix en C. On va trier un tableau de `uint64_t`, c'est-à-dire d'entiers non signés sur 64 bits.

```c
typedef uint64_t ui;
```

Considérons le nombre d'éléments à trier $n < 2^{31}-1$ (pour éviter des problèmes de représentation par les `int` habituels[a]). On va utiliser un tableau de `ui` et de taille n pour stocker les éléments à trier, et deux tableaux de `int` et de taille *radix* pour stocker les tableaux `hist` et `sums`.
On définit les constantes globales suivantes :

```c
// w
#define UI_SIZE 64
// p
#define BLOCK_SIZE 8
// radix
#define RADIX ???
#define BASE 1 << BLOCK_SIZE
```

▶ **Question 4** Écrire une fonction `void copy(ui* in, ui* out, int n)` copiant le tableau `in` dans le tableau `out`, tous deux de taille n.

▶ **Question 5** Écrire une fonction `void zero(int* tab, int n)` initialisant à 0 tous les éléments du tableau `tab` de taille n.

▶ **Question 6** En utilisant les opérateurs `<< >>`, écrire une fonction `ui get_digit(ui x, int k)` renvoyant le chiffre de poids $base^k$ de l'entier `x`.
En cas de difficulté, deux conseils : on peut utiliser le format `"%lx"` pour afficher un entier `ui` en hexadécimal (2 caractères représentent un octet) dans `printf`, et n'hésitez pas à me demander une solution si vous êtes mal à l'aise / ne connaissez pas la représentation des entiers en mémoire.

▶ **Question 7** En déduire la fonction `int* histogram(ui* in, int n, int k)` renvoyant le tableau `hist` de taille $radix$ tel que `hist[i]` est le nombre d'éléments de `in` ayant le chiffre de poids $base^k$ égal à i.

▶ **Question 8** En déduire la fonction `int* prefix_sums(int* hist, int k)` renvoyant le tableau `sums` de taille $radix$ tel que `sums[i]` est le nombre d'éléments de `in` ayant le chiffre de poids $base^k$ strictement inférieur à i.

▶ **Question 9** En déduire une fonction `void radix_sort(ui* tab, int n)` triant le tableau `in` de taille n et stockant le résultat dans le tableau `out`. *Éventuellement : on pourra commencer par écrire cette fonction sans se soucier de libérer la mémoire allouée au fur et à mesure.*

▶ **Question 10** Déterminer la complexité de cet algorithme en fonction de n, p et w. *On considère que la fonction* `get_digit` *est en* $\mathcal{O}(1)$.

On va maintenant pouvoir tenter d'optimiser les valeurs choisies au cours de l'implémentation de notre algorithme. Pour cela, on va changer la commande de compilation pour activer toutes les optimisations du compilateur :

```
# gcc -O3 -DNDEBUG -Wall -Wextra -Werror  -o radix radix.c
```
Shell

- `-O3` active toutes les optimisations du compilateur.
- `-DNDEBUG` désactive les assertions.
- `-Wall -Wextra -Werror` permet de conserver les erreurs habituelles.
- On ne met pas `-fsanitize=...` puisque cette option instrumente votre exécutable en ajoutant des tests de débogage.

On peut mesurer approximativement le temps d'exécution d'une fonction de votre programme en utilisant la bibliothèque `time.h`. Cette bibliothèque fournit une fonction `clock_t clock(void)` renvoyant le nombre de tours d'horloge écoulés depuis le début de l'exécution du programme. On peut donc mesurer le temps d'exécution d'une fonction `f` en mesurant `clock()` avant et après l'appel à `f`, puis en divisant avec une division flottante par la constante `CLOCKS_PER_SEC`.

▶ **Question 11** Tester différentes valeurs de p pour trouver une valeur minimisant la complexité. On pourra générer des tableaux aléatoires avec la fonction suivante :

```c
#define imax_bits(m) ((m)/((m)%255+1) / 255%255*8 + 7-86/((m)%255+12))
#define RAND_MAX_WIDTH imax_bits(RAND_MAX)

ui rand64(void) {
    uint64_t r = 0;
    for (int i = 0; i < 64; i += RAND_MAX_WIDTH) {
        r <<= RAND_MAX_WIDTH;
        r ^= (unsigned) rand();
    }
    return r;
}
```

▶ **Question 12** Proposer une optimisation de cette fonction utilisant quatre allocations mémoire.

a. Un petit calcul d'ordre de grandeur : le plus grand tableau selon cette condition prendrait $(2^{31} - 2) \times 8$ octets, soit environ 34 gigaoctets.