TD XIX

TRI RADIX

Le *tri radix* est généralement la manière la plus efficace de trier des entiers ou des flottants (mais il ne permet pas de trier des données plus compliquées, comme par exemple des chaînes de caractères).

1 Tri stable

1.1 Tri suivant une clé

Pour l'instant, nous avons essentiellement trié des tableaux de nombres, suivant l'ordre « naturel », c'est-à-dire la relation d'ordre usuelle $\leqslant$ sur $\mathbb{N}$, sur $\mathbb{Z}$, sur $\mathbb{R}$...

Il est cependant très courant de vouloir trier des données suivant un critère plus ou moins arbitraire :

- une liste de triplets (Nom, Prénom, Date de naissance) par date de naissance croissante ;
- une liste de listes par longueurs décroissante (les listes les plus longues en premier, les listes les plus courtes en dernier) ;
- une liste de complexes par module croissant ;
- une liste de triplets (Nom, Prénom, Date de naissance) par date de naissance croissante, puis en cas d'égalité par nom croissant, puis en cas d'égalité par prénom croissant.

Remarque

Dans les trois premiers exemples, l'ordre des éléments dans la liste triée n'est pas entièrement défini : les listes $(1, i, 2)$ et $(i, 1, 2)$, par exemple, sont toutes les deux triées par module croissant. Nous y reviendrons dans la partie suivante.

Pour spécifier l'ordre suivant lequel on souhaite trier, le plus courant est de fournir une *fonction de comparaison* : c'est la manière standard de procéder en OCaml, en C, en C++, en JavaScript[1]...Une telle fonction prend deux éléments à trier et renvoie un entier qui vérifie :

- $\texttt{compare}(x, y) < 0$ si x doit être placé avant y (si x est « plus petit » que y pour l'ordre considéré) ;
- $\texttt{compare}(x, y) > 0$ si x doit être placé après y (si x est « plus grand » que y pour l'ordre considéré) ;
- $\texttt{compare}(x, y) = 0$ si x et y sont « équivalents » pour l'ordre considéré, ce qui signifie que l'on peut placer x avant ou après y, indifféremment.

Remarque

Pour qu'une telle fonction de comparaison ait un sens, il faut qu'elle vérifie certaines propriétés. Nous y reviendrons quand nous étudierons plus en détail les ensembles ordonnés, mais pour l'instant nous pouvons l'ignorer : si la fonction traduit effectivement le fait pour x de devoir être placé avant ou après y, ces propriétés seront toujours vérifiées.

Exercice XIX.1 — `List.sort` et `Array.sort` p. 7

En OCaml, les fonctions de tri pré-définies ont le type suivant :

```
List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
Array.sort : ('a -> 'a -> int) -> 'a array -> unit
```

Il y a une différence dans le type d'arrivée (ce qui est normal puisque `List.sort` renvoie une nouvelle liste alors que `Array.sort` modifie le tableau fourni en argument), mais les deux fonctions ont un premier argument de type `'a -> 'a -> int`. Cet argument est la fonction de comparaison qui définit l'ordre suivant lequel on souhaite trier.

1. Mais pas en Python.

1. Que va renvoyer `List.sort (fun x y -> x - y) [3; 2; 4; 1; 7]` ?
2. Même question pour `List.sort (fun x y -> y - x) [3; 2; 4; 1; 7]`.
3. Comment trier une liste d'entiers par valeur absolue décroissante ? *La fonction `abs : int -> int` permet de calculer la valeur absolue d'un entier.*
4. Comment trier une liste de type `(int * int) list` suivant le critère suivant :
   - par valeur absolue croissante de la première composante ;
   - en cas d'égalité, par seconde composante décroissante.

   Comme la fonction de comparaison est ici un peu plus compliquée, il est fortement conseillé de la définir séparément.

Remarque
Si l'on veut trier une liste (ou un tableau) suivant l'ordre « usuel » (celui correspondant à l'opérateur <=), on peut utiliser la fonction prédéfinie `compare : 'a -> 'a -> int`.

I.2 Tri stable

Comme nous l'avons vu plus haut, une fonction de comparaison ne définit pas en général un ordre unique sur les éléments. Considérons par exemple la liste $u = [(3,4),(5,4),(2,7),(6,1),(3,3),(7,2),(2,1)]$. Si l'on souhaite trier les éléments (x,y) de u par somme croissante, toutes les réponses suivantes sont acceptables :

- $[(2,1),(3,3),(3,4),(6,1),(5,4),(2,7),(7,2)]$;
- $[(2,1),(3,3),(6,1),(3,4),(5,4),(2,7),(7,2)]$;
- $[(2,1),(3,3),(6,1),(3,4),(7,2),(2,7),(5,4)]$;
- $[(2,1),(3,3),(6,1),(3,4),(7,2),(5,4),(2,7)]$;
- ...

Définition XIX.1 – Tri stable

Un tri (suivant une clé) est dit *stable* si, pour tous x, y de la liste initiale tels que $\texttt{compare}(x,y) = 0$, x est avant y dans la liste triée si et seulement si il était avant y dans la liste initiale.

Remarques

- Dans l'exemple ci-dessus, un tri *stable* par somme croissante donne nécessairement le premier résultat proposé. Il y a trois couples dont la somme vaut 9, ils doivent nécessairement être dans le même ordre à l'arrivée qu'au départ, c'est-à-dire $(5,4),(2,7),(7,2)$.
- Quelle que soit la clé choisie pour le tri, le résultat d'un tri stable est uniquement défini.
- Un tri est stable s'il n'échange la position relative de deux éléments que lorsque c'est nécessaire pour respecter l'ordre demandé.

Le caractère stable ou non est une propriété de l'algorithme de tri utilisé. En OCaml, il existe deux fonctions `List.stable_sort` et `Array.stable_sort`, dont la stabilité est garantie ; ce n'est pas le cas pour `List.sort` et `Array.sort`[2] Un des principaux intérêts d'un tri stable est que l'on peut effectuer plusieurs tris successifs suivant des critères simples et obtenir le même résultat qu'après un unique tri suivant un critère plus compliqué.

Exercice XIX.2 – Tris stables successifs p. 7

On considère une liste de couples d'entiers, et l'on souhaite trier cette liste :

- par somme croissante ;
- en cas d'égalité, par première composante croissante.

Par exemple, sur la liste $u = [(3,4),(5,4),(2,7),(6,1),(3,3),(7,2),(2,1)]$, on doit obtenir :

$$[(2,1),(3,3),(3,4),(6,1),(2,7),(5,4),(7,2)]$$

2. À l'heure actuelle, `List.sort` utilise le même algorithme que `List.stable_sort` (et est donc stable), mais `Array.sort` utilise un algorithme différent (qui n'est pas stable).

On considère les deux fonctions suivantes :

```
let cmp_somme (x, y) (x', y') -> x + y - (x' + y')

let cmp_premiere (x, _) (x', _) -> x - x'
```

Si u est donnée sous la forme d'un `(int * int) array`, comment la trier en utilisant deux appels à `Array.stable_sort` ?

2 Tri radix

Principe

Supposons que l'on dispose d'un tableau de n entiers, tous compris entre 0 et 999. L'idée du tri radix (en base 10 dans cet exemple) est d'effectuer trois passes successives de tri :

- une première en ne tenant compte que du chiffre des unités ;
- une seconde en ne tenant compte que du chiffre des dizaines ;
- une troisième en ne tenant compte que du chiffre des centaines.

Si chaque étape de tri est effectuée de manière stable, le tableau sera trié par ordre croissant à la fin.

Exercice XIX.3 p. 7

Dérouler les grandes étapes de l'algorithme sur la liste :

$$(123, 211, 312, 321, 133, 121, 213, 30, 103, 200)$$

Pour effectuer un tri radix *efficace*, il faut :

- que chaque passe de tri s'effectue rapidement ;
- que le nombre de passes soit aussi petit que possible.

Pour le premier point, une première idée est de travailler en base 2 plutôt qu'en base 10 (ou autre) : en effet, extraire le k-ème chiffre en base 2 peut se faire de manière très efficace à l'aide d'opérations *bitwise*, comme nous l'avons vu en cours. Il reste ensuite à trouver une manière efficace d'effectuer l'étape de tri, mais nous en parlerons plus tard.

Pour le deuxième point en revanche, la base 2 n'est pas idéale : en effet, pour un nombre n donné, plus la base est petite, plus il y a de chiffres. Pour diminuer le nombre d'étapes tout en gardant des opérations arithmétiques efficaces, l'idée est alors de *regrouper les bits* par paquet de taille fixée. Cela revient en fait à remplacer la base 2 par une base 2^p pour un certain p.

Remarque

Pour fixer l'intuition, on peut remarquer que 754215 possède 6 chiffres en base 10, 3 chiffres en base $10^2 = 100$ ($75 \cdot 100^2 + 42 \cdot 100 + 15$) et deux chiffres en base $10^3 = 1000$ ($754 \cdot 1000 + 215$).

Exercice XIX.4 — Nombre de passes p. 7

On considère que l'on souhaite trier des entiers non signés codés sur w bits, et que l'on va utiliser la base 2^p.

1. Combien d'étapes faudra-t-il faire dans le tri radix (au maximum) ?
2. Comme nous le verrons plus tard, le coût d'une étape croît avec p. Si $w = 32$, quelles sont les valeurs de p qui ont une chance d'être optimales ?

Algorithme pour une passe

Chaque passe du tri radix va se faire suivant le principe suivant :

- on a un tableau `in` d'entiers non signés, que l'on souhaite trier, de manière stable, par valeur croissante du chiffre de poids $\texttt{radix}^k$;
- le tableau `in` ne sera pas modifié : on renverra un nouveau tableau `out` ;
- on commence par calculer un tableau `hist` de longueur `radix` indiquant, pour chaque valeur $\texttt{i} \in [0 \dots \texttt{radix} - 1]$, combien d'éléments du tableau `in` ont leur chiffre de poids $\texttt{radix}^k$ égal à `i` ;
- on calcule ensuite un tableau `sums`, également de longueur `radix`, indiquant pour chaque `i` combien d'éléments de `in` ont leur chiffre de poids $\texttt{radix}^k$ strictement inférieur à `i` ;
- on remarque que les `x` de `in` dont le chiffre considéré vaut `i` occuperont des cases consécutives du tableau `out` à partir de la case `sums[i]` ;
- on parcourt le tableau `in` en répartissant les éléments dans le tableau `out` suivant la valeur de leur chiffre (il faut mettre à jour `sums` au fur et à mesure).

Les figures XIX.1 et XIX.2 illustrent le principe de l'algorithme, avec $\texttt{radix} = 4$ et $\texttt{k} = 0$ (on s'intéresse donc au chiffre des unités en base 4).

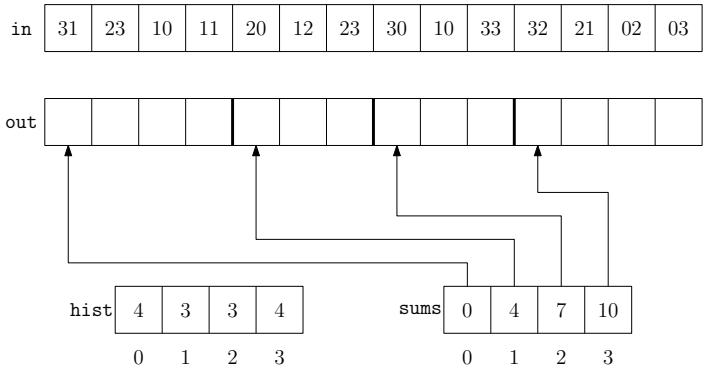

Figure XIX.1 – État initial après le calcul de `hist` et `sums`.

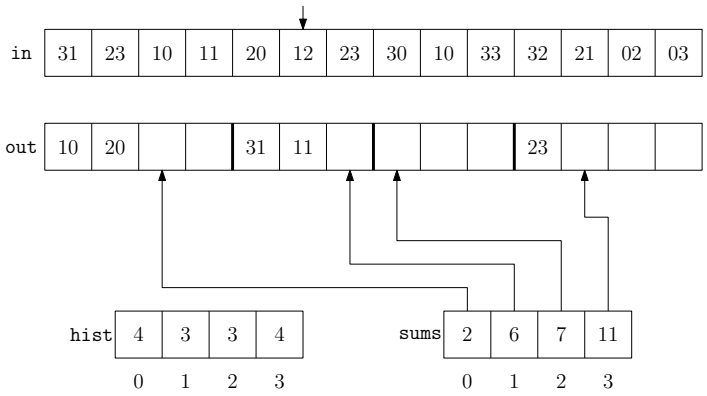

Figure XIX.2 – État après avoir traité 5 éléments (l'élément 12 pointé sera le prochain à être traité).

Programmation du tri radix

Dans toute la suite, on suppose que :

- on souhaite trier un tableau d'entiers de type `uint32_t` ou `uint64_t`, et que l'on a fait l'un des `typedef` suivants :

```
typedef uint32_t ui;
// ou bien
typedef uint64_t ui;
```

- le nombre d'éléments dans le tableau à trier est inférieur à $2^{31} - 1$ (soit environ deux milliards), de sorte que l'on peut utiliser des `int` pour tous nos compteurs ;
- on a défini une constante globale `BLOCK_SIZE`, et que l'on effectuera le tri en base $2^{\texttt{BLOCK\_SIZE}}$ (c'est-à-dire en regroupant les bits par paquets de `BLOCK_SIZE`) ;
- on notera `radix` notre base (autrement dit, $\texttt{radix} = 2^{\texttt{BLOCK\_SIZE}}$).

Exercice XIX.5 – Définition des constantes p. 8

Définir les constantes globales $\texttt{RADIX} = 2^{\texttt{BLOCK\_SIZE}}$ et $\texttt{MASK} = 2^{\texttt{BLOCK\_SIZE}} - 1$.

Exercice XIX.6 – Fonctions utilitaires p. 8

Écrire deux fonctions :

```
void copy(ui *out, ui *in, int len);
void zero_out(int *arr, int len);
```

L'appel `copy(out, in, len)` doit recopier le contenu du tableau `in` sur le tableau `out` (on suppose que les deux tableaux ont la même taille).
L'appel `zero_out(arr, len)` doit remplacer toutes les valeurs présentes dans le tableau `arr` par des zéros.
Remarque
Il existe bien sûr des fonctions pour faire cela dans la bibliothèque standard, mais ici les ré-écrire pour notre usage particulier ne nous coûte pas grand-chose.

Exercice XIX.7 – Extraction d'un chiffre en base `radix` p. 8

Écrire une fonction `extract_digit` qui renvoie la valeur du chiffre de poids $\texttt{radix}^k$ dans l'écriture de n en base `radix`.

```
ui extract_digit(ui n, int k);
```

On procédera à l'aide d'opérations bitwise *: la technique a déjà été vue exactement en exercice. On pourra supposer que la valeur de* `k` *correspond bien à un chiffre qui existe*

Exercice XIX.8 – Calcul de l'histogramme p. 8

Écrire une fonction `histogram` prenant en entrée un tableau d'entiers `arr` et un entier `k` et renvoyant un tableau `hist` de longueur `radix` tel que `hist[i]` soit égal au nombre d'éléments de `arr` dont le `k`-ème chiffre en base `radix` vaut `i` (pour $0 \leqslant \texttt{i} < \texttt{radix}$).
Ce calcul doit absolument se faire en un seul parcours du tableau `arr`.

```
int* histogram(ui *arr, int len, int k);
```

Exercice XIX.9 – Sommes préfixes p. 8

Écrire une fonction `prefix_sum` prenant en entrée un tableau `hist` et renvoyant un tableau `sums` de même longueur tel que `sums[i]` soit égal à $\sum_{j=0}^{i-1} \texttt{hist[j]}$. On aura donc en particulier $\texttt{sums[0]} = 0$.
Ce calcule doit absolument se faire en un seul parcours du tableau `hist`.

```
int* prefix_sums(int* hist, int len);
```

Exercice XIX.10 p. 9

Écrire une fonction `radix_pass` ayant le prototype suivant :

```
void radix_pass(ui *out, ui *in, int len, int k);
```

Cette fonction implémentera l'algorithme décrit ci-dessus en considérant le chiffre de poids $\texttt{radix}^k$.

Exercice XIX.11 p. 9

Écrire la fonction `radix_sort`. Cette fonction ne renverra rien mais modifiera le tableau passé en argument. Pour l'instant, on essaiera surtout d'écrire une fonction correcte (quitte à faire des copies de tableaux superflues).

```
void radix_sort(ui *arr, int len)
```

Exercice XIX.12 — Complexité du tri radix p. 9

Déterminer la complexité totale du tri radix en fonction de la largeur w des entiers, de la valeur ℓ de `BLOCK_SIZE` et de la longueur n du tableau.

Optimisations

Dans cette partie, on s'intéresse aux performances pratiques de notre implémentation. Mesurer ces performances n'a pas trop de sens avec nos options de compilation usuelles ; on utilisera plutôt :

```
$ gcc -O3 -march=native -DNDEBUG -Wall -Wextra -o radix.out radix.c
```

Pour mesurer le temps écoulé lors de l'exécution d'une partie du programme, on pourra ajouter l'entête `#include <time.h>` et utiliser la fonction `clock`. La différence entre deux appels à `clock()` peut être divisée par la constante `CLOCKS_PER_SEC` pour obtenir le temps (en secondes) entre les deux appels.

Exercice XIX.13 p. 9

Tester différentes valeurs de `BLOCK_SIZE` et essayer de déterminer la valeur idéale.

Exercice XIX.14 p. 9

En pratique, le facteur limitant pour les performances (dans le cas du tri radix) est le sous-système mémoire. Il est donc très important de minimiser le nombre de lectures et d'écriture : il est possible de ne faire que 5 passes de lecture et 4 passes d'écriture pour trier un tableau de `uint32_t` avec un `BLOCK_SIZE` de 8. Déterminer si c'est bien le cas pour la version que vous avez écrite, et la modifier le cas échéant. *Attention, il est très facile d'introduire des bugs ici. On prendra bien soin de tester les modifications apportées, et ce pour plusieurs valeurs de `BLOCK_SIZE`.*

Cas des entiers signés

Exercice XIX.15 p. 11

Écrire une version de `radix_sort` permettant de trier un tableau d'entiers signés.

Solutions

Correction de l'exercice XIX.1 page 1

1. On trie par ordre croissant : `[1; 2; 3; 4; 7]`.
2. On trie par ordre décroissant : `[7; 4; 3; 2; 1]`.
3. `List.sort (fun x y -> abs y - abs x) liste`.
4.

```
let cmp (x, y) (x', y') =
  if abs x < abs x' then -1
  else if abs x > abs x' then 1
  else y' - y

let trier liste = List.sort cmp liste
```

Correction de l'exercice XIX.2 page 2

Il faut faire :

```
Array.stable_sort cmp_premiere u;
Array.stable_sort cmp_somme u
```

Correction de l'exercice XIX.3 page 3

On obtient successivement :

- $(30, 200, 211, 321, 121, 312, 123, 133, 213, 103)$
- $(200, 103, 211, 312, 213, 321, 121, 123, 30, 133)$
- $(30, 103, 121, 123, 133, 200, 211, 213, 312, 321)$

Correction de l'exercice XIX.4 page 3

1. En k étapes on traite kp bits, il faut donc que $kp \geqslant w$, c'est-à-dire $k \geqslant \frac{w}{p}$. Le nombre d'étapes nécessaire vaut donc $\lceil w/p \rceil$.
2. On peut éliminer les valeurs de p telles que $\lceil w/p \rceil = \lceil w/(p-1) \rceil$, puisque $p-1$ sera toujours préférable dans ce cas (même nombre d'étape, et chaque étape est moins coûteuse). Les valeurs potentiellement intéressantes sont donc (pour $w = 32$) :
   - $p = 1$, 32 étapes ;
   - $p = 2$, 16 étapes ;
   - $p = 3$, 11 étapes ;
   - $p = 4$, 8 étapes ;
   - $p = 5$, 7 étapes ;
   - $p = 6$, 6 étapes ;
   - $p = 7$, 5 étapes ;
   - $p = 8$, 4 étapes ;
   - $p = 11$, 3 étapes ;
   - $p = 16$, 2 étapes ;
   - $p = 32$, 1 étape (cette valeur est complètement irréaliste comme on le verra une fois l'algorithme expliqué).

Correction de l'exercice XIX.5 page 5

```
const int BLOCK_SIZE = 5; // par exemple
const int RADIX = 1 << BLOCK_SIZE;
const int MASK = RADIX - 1;
```

Correction de l'exercice XIX.6 page 5

```
void copy(ui *out, ui *in, int len){
    for (int i = 0; i < len; i++){
        out[i] = in[i];
    }
}

void zero_out(int *arr, int len){
    for (int i = 0; i < len; i++){
        arr[i] = 0;
    }
}
```

Correction de l'exercice XIX.7 page 5

```
ui extract_digit(ui n, int k){
    return (n >> (k * BLOCK_SIZE)) & MASK;
}
```

Correction de l'exercice XIX.8 page 5

```
int* histogram(ui *arr, int len, int k){
    int* hist = malloc(RADIX * sizeof(int));
    zero_out(hist, RADIX);
    for (int i = 0; i < len; i++){
        int digit = extract_digit(arr[i], k);
        hist[digit]++;
    }
    return hist;
}
```

Correction de l'exercice XIX.9 page 5

```
int* prefix_sum(int *hist, int len){
    int* sums = malloc(len * sizeof(int));
    int s = 0;
    for (int i = 0; i < len; i++){
        sums[i] = s;
        s += hist[i];
    }
    return sums;
}
```

Correction de l'exercice XIX.10 page 6

On suit précisément les étapes décrites dans l'algorithme et illustrées sur les schémas. Attention à ne pas oublier de libérer les deux tableaux auxiliaires.

```
void radix_pass(ui *out, ui *in, int len, int k){
    int* hist = histogram(in, len, k);
    int* sums = prefix_sum(hist, RADIX);
    for (int i = 0; i < len; i++){
        int digit = extract_digit(in[i], k);
        out[sums[digit]] = in[i];
        sums[digit]++;
    }
    free(hist);
    free(sums);
}
```

Correction de l'exercice XIX.11 page 6

On peut faire nettement plus efficace, mais la version ci-dessous est la plus simple, et elle a la bonne complexité. Pour le calcul de `nb_digits`, on a remarqué que $\lceil w/b \rceil = 1 + \lfloor (w-1)/b \rfloor$, sinon il faut utiliser un `if (w % b == 0) {...} else {...}`.

```
void radix_sort(ui *in, int len){
    int nb_digits = 1 + (sizeof(ui) * 8 - 1) / BLOCK_SIZE;
    for (int k = 0; k < nb_digits; k++){
        ui *tmp = malloc(len * sizeof(ui));
        radix_pass(tmp, in, len, k);
        copy(in, tmp, len);
        free(tmp);
    }
}
```

Correction de l'exercice XIX.12 page 6

Pour chaque passe :

- le calcul de l'histogramme est en $O(n + 2^\ell)$ (puisqu'on crée un tableau de taille 2^ℓ) ;
- le calcul des sommes préfixes est en $O(2^\ell)$;
- la répartition des éléments dans `out` est en $O(n)$.

Une passe est donc en $O(n+2^\ell)$ et comme il y a $O(w/\ell)$ passes, on a au total du $O\left(\frac{w(n+2^\ell)}{\ell}\right)$.

Correction de l'exercice XIX.13 page 6

Tester les valeurs de `BLOCK_SIZE` avant d'avoir optimisé le programme n'est pas forcément idéal, mais cela permet quand même d'avoir une idée. Le plus souvent, le meilleur choix est 8, mais les valeurs 6, 7 et 11 peuvent aussi être intéressantes (ça dépend de détails architecturaux, et il faut donc tester pour savoir ce qui est le mieux *sur une machine donnée*).

Correction de l'exercice XIX.14 page 6

On propose ci-dessous un code raisonnablement optimisé :

```c
void inplace_prefix_sum(int* t, int n){
    int s = 0;
    for (int i = 0; i < n; i++){
        int tmp = t[i];
        t[i] = s;
        s += tmp;
    }
}

void radix_sort_2(ui* t, int len){
    int nb_digits = 1 + (sizeof(ui) * 8 - 1) / BLOCK_SIZE;
    ui* t_aux = malloc(len * sizeof(ui));
    int* hists = malloc(RADIX * nb_digits * sizeof(int));

    // initialize hists
    for (int i = 0; i < RADIX * nb_digits; i++){
        hists[i] = 0;
    }

    // Compute all histograms
    for (int i = 0; i < len; i++){
        for (int k = 0; k < nb_digits; k++){
            int digit = extract_digit(t[i], k);
            hists[k * RADIX + digit]++;
        }
    }

    for (int i_digit = 0; i_digit < nb_digits; i_digit++){
        // select the current histogram
        int *hist = &hists[i_digit * RADIX];

        // replace it with its prefix sum
        inplace_prefix_sum(hist, RADIX);

        // scatter
        for (int i = 0; i < len; i++){
            int digit = extract_digit(t[i], i_digit);
            t_aux[hist[digit]] = t[i];
            hist[digit]++;
        }

        // switch t and t_aux
        ui* tmp = t;
        t = t_aux;
        t_aux = tmp;
    }

    if (nb_digits % 2 != 0){
        copy(t_aux, t, len);
        free(t);
    } else {
        free(t_aux);
    }

    free(hists);
}
```

Sur ma machine, on gagne un facteur 2 environ par rapport à la première version, pour arriver à approximativement 8 nano-secondes par élément du tableau[a]

a. Pour un tableau d'un million d'éléments. La complexité est linéaire en n pour l et w fixés, mais en réalité il y a de petites variations.

Correction de l'exercice XIX.15 page 6

Le plus simple est de trier comme nous l'avons fait jusqu'à présent et de rajouter une ultime passe dans laquelle on place tous les éléments négatifs avant les positifs (en utilisant le même principe que pour une passe du tri radix, mais avec deux catégories au lieu de 2^{radix}).