Tris 2

> **Objectifs**
>
> — Comprendre le principe, la mise en œuvre et la complexité du tri par insertion.
> — Comprendre le principe, la mise en œuvre et la complexité du tri par comptage.

Introduction

Nous avons déjà vu deux tris par comparaison : le tri sélection, où à chaque parcours du tableau, on récupère l'élément le plus grand pour le ranger correctement ; et le tri à bulles, où durant le parcours du tableau, l'élément le plus grand remonte par échange successif.

Nous allons voir dans ce chapitre deux nouveaux algorithmes de tris : le tri par insertion, du même type que les deux précédents, et le tri par comptage qui nécessitera certaines conditions et un tableau supplémentaire pour fonctionner.

Tri par insertion

Principe

— On prend un élément dans le tableau
— On le déplace en décalant les autres vers la gauche jusqu'à ce qu'il soit bien placé
— On recommence avec l'élément de droite

Ainsi, le début du tableau est trié, et on recommence jusqu'à avoir trié tout le tableau. **C'est le tri naturel des cartes.**

> **Exemple visuel**
>
> Soit le tableau initial : `[6, 4, 8, 1, 7, 5, 2, 3]`
> **Principe :** On insère chaque élément à sa place dans la partie déjà triée (à gauche).
>
> 1. `[6] | 4, 8, 1, 7, 5, 2, 3` → Le premier élément est trivialemment trié

2. `[4, 6] | 8, 1, 7, 5, 2, 3` → Insérer 4 : on décale 6 et on place 4
3. `[4, 6, 8] | 1, 7, 5, 2, 3` → Insérer 8 : déjà à sa place
4. `[1, 4, 6, 8] | 7, 5, 2, 3` → Insérer 1 : décaler 8, 6, 4 puis placer 1
5. `[1, 4, 6, 7, 8] | 5, 2, 3` → Insérer 7 : décaler 8 puis placer 7
6. `[1, 4, 5, 6, 7, 8] | 2, 3` → Insérer 5 : décaler 8, 7, 6 puis placer 5
7. `[1, 2, 4, 5, 6, 7, 8] | 3` → Insérer 2 : décaler puis placer
8. `[1, 2, 3, 4, 5, 6, 7, 8]` → Insérer 3 : décaler puis placer

Le symbole `|` sépare la partie triée (gauche) de la partie non triée (droite).

⚠ Pseudo-code

```
ALGORITHME TriInsertion(T):
DONNEES
  T : tableau d'entiers de taille n
VARIABLES:
  i, j, temp : entiers
DEBUT
  i ← 2
  TQ i   n FAIRE
    j ← i
    temp ← T[j]
    TQ j > 1 ET temp < T[j-1] FAIRE
      T[j] ← T[j-1]
      j ← j-1
    FTQ
    T[j] ← temp
    i ← i + 1
  FTQ
FIN
```

Arrêt

Dans la boucle intérieure, comme dans la boucle principale, les tests d'arrêt dépendent de variables qui sont incrémentées ou décrémentées, donc l'arrêt est assuré.

Ainsi, l'algorithme s'arrête.

Validité

Posons la propriété $\mathcal{P}(k)$ suivante pour l'entier $k \in [1, n-1]$:

$\mathcal{P}(k)$: "À la fin de l'itération k, le tableau de 1 à k+1 est trié dans l'ordre croissant et la variable i vaut k+1"

- **Initialisation :** Lors du premier passage dans la boucle principale, $i = 2$, on compare $T[2]$ et $T[1]$. Si $T[2] \geq T[1]$ alors on ne fait rien (car on remet $temp = T[2]$ dans $T[2]$) et le tableau de 1 à 2 est trié. Sinon, $T[2]$ reçoit $T[1]$ et $T[1]$ reçoit $temp$, ce qui veut dire qu'on a interverti les deux éléments et donc le tableau de 1 à 2 est dans l'ordre croissant.
- **Hérédité :** On suppose $\mathcal{P}(k)$ vraie pour un $k \in [1, n-1]$. Au prochain passage dans la boucle, la variable `i` vaut $k+2$. $temp$ prend la valeur $T[k+2]$ et tant qu'il y a un élément qui précède et qui est plus grand que $temp$, on décale l'élément. Ainsi, à la fin de la boucle intérieure, $temp$ est compris entre un élément plus grand que lui et un élément plus petit que lui (ou alors le bord du tableau).

  Or d'après l'hypothèse de récurrence, le tableau de 1 à $k+1$ est trié. Donc le sous tableau de 1 jusqu'à la valeur de $temp$ est trié, et comme on a décalé tous les éléments, le sous tableau de la valeur de $temp$ à $k+2$ est aussi trié. Ainsi le tableau de 1 à $k+2$ est trié.
- **Conclusion :** La propriété étant vraie pour $k = 1$, héréditaire pour $k \in [1, n-1]$, elle est donc vraie pour tout $k \in [1, n-1]$. Ainsi, $\mathcal{P}(n-1)$ est vraie et le tableau est trié de 1 à n.

Complexité

Boucle intérieure : j prend les valeurs de i à 2 dans le pire des cas. Donc la complexité du pire cas est en $\Theta(i-1) \times 2 = \Theta(i)$.

Dans le meilleur des cas, le tableau est rangé et la complexité est en $\Theta(2) = \Theta(1)$.

Ainsi la complexité moyenne de la boucle intérieure est en $O(i)$.

Boucle principale : i prend les valeurs de 2 à n, il y aura donc $n-1$ passages dans la boucle. Ainsi la complexité de l'algorithme sera :

$$C(n) = \sum_{i=2}^{n}(4 + O(i)) = 4(n-1) + O\left(\sum_{i=2}^{n} i\right) = 4n - 4 + O\left(\frac{(n-1)(n-2)}{2}\right) = O(n^2)$$

> **!** Remarque
>
> Les tris comparatifs dont la complexité est en $\Theta(n^2)$ ou en $O(n^2)$ sont considérés comme **lents**.
> Le meilleur tri comparatif est en $O(n\log(n))$ et il est prouvé que l'on ne peut pas faire

mieux (pour les tris comparatifs).

Tri par comptage

Principe

Si les éléments à trier sont bornés, on peut se servir d'un tableau annexe pour compter les nombres présents, il suffit ensuite de les écrire dans le bon ordre.

L'inconvénient est d'avoir recours à un tableau supplémentaire, ce qui augmente la complexité en espace (mais va diminuer la complexité en opération).

Exemple visuel

Soit le tableau initial : `[6, 4, 8, 1, 7, 2, 5, 3]`
Étape 1 : Comptage - On crée un tableau de comptage `C` de taille 8 (le maximum) initialisé à 0.
On parcourt le tableau et pour chaque valeur, on incrémente `C[valeur]` :

- Lire 6 → `C[6] = 1`
- Lire 4 → `C[4] = 1`
- Lire 8 → `C[8] = 1`
- Lire 1 → `C[1] = 1`
- Lire 7 → `C[7] = 1`
- Lire 2 → `C[2] = 1`
- Lire 5 → `C[5] = 1`
- Lire 3 → `C[3] = 0` puis quand on lit le dernier 3 → **Attention :** le dernier est 3 → `C[7] = 2` (car il y a deux 7)

Après comptage, le tableau `C` contient :

Indice	1	2	3	4	5	6	7	8
Compte	1	1	0	1	1	1	2	1

Étape 2 : Reconstruction - On parcourt `C` et pour chaque indice j, on écrit j autant de fois que `C[j]` dans le tableau de sortie :

- `C[1] = 1` → Écrire 1
- `C[2] = 1` → Écrire 2
- `C[3] = 0` → Ne rien écrire
- `C[4] = 1` → Écrire 4
- `C[5] = 1` → Écrire 5
- `C[6] = 1` → Écrire 6
- `C[7] = 2` → Écrire 7, 7
- `C[8] = 1` → Écrire 8

Résultat : `[1, 2, 4, 5, 6, 7, 7, 8]`

⚠ Pseudo-code

```
ALGORITHME TriComptage(T):
DONNEES
  T : tableau d'entiers de taille n
  max : maximum du tableau T
VARIABLES
  i, j, k : entiers
  C : tableau d'entiers de taille max initialisé à 0
DEBUT
  i ← 1
  TQ i   n FAIRE
    C[T[i]] ← C[T[i]] + 1
    i ← i + 1
  FTQ
  j ← 1
  k ← 1
  TQ j   max FAIRE
    TQ C[j] > 0 FAIRE
      T[k] ← j
      k ← k + 1
      C[j] ← C[j] - 1
    FTQ
    j ← j + 1
  FTQ
FIN
```

Arrêt

De la même façon que pour les autres tris, la première boucle est basée sur une variable i qui est incrémentée de 1 à chaque tour, donc qui dépassera nécessairement n.

Et pour la seconde boucle, elle est basée sur la variable j qui est incrémentée de 1 donc passera au-dessus de la valeur max à un moment. Pour la boucle intérieure à la seconde boucle, `C[j]` est décrémentée de 1, donc elle sera égale à 0 à un certain moment. Toutes les boucles s'arrêtent donc.

Validité

À la fin de la première boucle, nous avons créé un tableau C dont les indices sont les valeurs du tableau T et dont les éléments sont le nombre d'occurrences dans le tableau T.

Comme les indices sont rangés dans l'ordre, en recopiant dans le même ordre les indices dont les valeurs sont non nulles, on a automatiquement un tableau rangé par ordre croissant.

Complexité

Première boucle : i prend toutes les valeurs entre 1 et n. Ainsi il y aura n passages dans la boucle.

Le coût de la première boucle est en $\Theta(n)$.

Seconde boucle : j prend les valeurs de 1 à max, il y aura donc max passages dans la boucle.

Pour sa boucle intérieure, comme nous avons enregistré n éléments en tout, on peut considérer que pour un indice donné, il y a $\Theta(1)$ passages dans cette boucle.

De ce fait, la complexité de cette deuxième boucle est en $\Theta(max)$.

Ainsi la complexité de l'algorithme sera :

$$C(n) = \Theta(n) + \Theta(max) = \Theta(n + max)$$

> **!** Remarque importante
>
> Le tri par comptage n'est **pas un tri par comparaison**. Il exploite la structure des données (entiers bornés) pour obtenir une complexité linéaire en fonction de n et de la valeur maximale.
> Ce tri est très efficace quand max est du même ordre de grandeur que n (par exemple $max = O(n)$), donnant une complexité $\Theta(n)$.

Cependant, si $max \gg n$ (par exemple $max = 10^9$ et $n = 100$), le tri devient inefficace.

Récapitulatif des algorithmes de tri

Algorithme	$\check{C}(n)$ (meilleur)	$\hat{C}(n)$ (pire)	$\bar{C}(n)$ (moyen)	Cas d'utilisation intéressant
Sélection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Petits tableaux, simplicité
Bulles	$\Theta(n)$ (optimisé)	$\Theta(n^2)$	$O(n^2)$	Tableaux presque triés
Insertion	$\Theta(n)$	$\Theta(n^2)$	$O(n^2)$	Tableaux petits ou presque triés
Comptage	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	Valeurs entières bornées par k

Choix d'un algorithme de tri

- **Tableaux presque triés** : Tri par insertion ou tri à bulles optimisé
- **Petits tableaux (n < 50)** : Tri par insertion (simple et efficace)
- **Valeurs entières bornées** : Tri par comptage (très rapide si $k = O(n)$)
- **Cas général** : Utiliser des tris en $O(n \log n)$ comme le tri fusion ou le tri rapide (vus dans le prochain chapitre)