* Motivations : • le problème du tri est facile à énoncer et amène tout de suites les questions algorithmiques fondamentales (complexité en temps, en espace, correction, ...)
• Utile comme pré-traitement. (marche de Jarvis pour l'enveloppe convexe).
• Trier des objets selon leur clef.

## I] le problème du tri

Entrée : $n$ éléments $a_1 \dots a_n$ d'un ensemble $E$ totalement ordonné.

Sortie : une réorganisation $a_{\sigma(1)} \dots a_{\sigma(n)}$ des éléments d'entrée, où $\sigma \in \mathfrak{S}_n$, telle que :

$$a_{\sigma(1)} \leq \dots \leq a_{\sigma(n)}$$

• Rq : Dans la pratique, les objets sont ordonnés selon leur clef.

Déf 1 : un tri est stable s'il vérifie :

$$\forall i \leq j \in [\![1;n]\!], \quad \left( a_i = a_j \Rightarrow \sigma(i) \leq \sigma(j) \right)$$

ex 2 : Pour l'entrée $a_1 a_2 a_3 = 1, 2, 1$, les deux permutations suivantes correspondent à un tri

$$\sigma_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \qquad \sigma_2 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

Seul $\sigma_1$ correspond à un tri stable.

Déf 3 : un algorithme de tri est dit en place (ou sur place) s'il n'utilise, en plus de la mémoire occupée par l'entrée, qu'une quantité de mémoire indépendante de l'entrée.

• Critères de comparaison des algorithmes de tri :
- Complexité temporelle (pire cas / en moyenne).
- Complexité spatiale.
- Stabilité.
- Caractère en place.

## II] Les tris par comparaison.

Déf 4 : un algorithme de tri est dit par comparaison si le tri qu'il effectue repose uniquement sur des comparaisons deux à deux des éléments d'entrée.

Thm 5 : La complexité, en nombre de comparaisons, d'un algorithme de tri par comparaisons est d'ordre de grandeur supérieur ou égal à $n \log n$ :

$$\boxed{n_{comp} = \Omega(n \log)}$$

• tri par sélection : on recherche le minimum des éléments pas encore triés. On le place à la suite des éléments déjà triés.
Complexité dans le pire des cas : $\mathcal{O}(n^2)$

• tri par insertion : Le principe est d'insérer un à un les éléments parmi ceux qu'on a déjà triés.

```
insert_sort(v) :
    Pour i = 1 à (taille(v) - 1) :
        j = i
        Tant que v(j) < v(j-1) et j ≥ 1 :
            inverser v(j) et v(j+1)
            j := j - 1
```

- tri en place.
- tri stable.
- Complexité dans le pire des cas : $\mathcal{O}(n^2)$.
- très efficace sur des petits tableaux.

• tri fusion : on découpe l'ensemble des objets à trier en deux ensembles de même taille que l'on trie récursivement, puis on fusionne les deux ensembles triés.

Complexité dans le pire des cas : $\mathcal{O}(n \log n)$

• tri rapide : on choisit un pivot p. On sépare les objets à trier en deux sous-ensembles $E_1 = \{$ éléments $< p \}$ et $E_2 = \{$ éléments $\geq p \}$. On trie récursivement $E_1$ et $E_2$. On renvoie la concaténation $E_1 \cup \{p\} \cup E_2$.

```
quick_sort(v) :
    Si taille(v) ≥ 2 :
        p = v(0)
        i = 0
        j = taille(v) - 1
        Tant que i < j :
            Si v(i+1) ≥ p :
                inverser v(i+1) et v(j)
                j := j - 1
            Sinon :
                inverser v(i) et v(i+1)
                i := i + 1
        quick_sort(v[0, ..., i-1])
        quick_sort(v[i+1, ..., taille(v) - 1])
```

- tri en place
- Complexité dans le pire des cas : $\mathcal{O}(n^2)$
- Complexité en moyenne : $\mathcal{O}(n \log n)$

• tri par tas :

Déf 6 : un tas est un arbre binaire quasi-complet dont l'étiquette de chaque noeud est plus grande que celles de ses fils.

Principe : On forme un tas avec les éléments de l'entrée. On extrait le maximum itérativement.

DEV : Algorithme du tri par tas, correction et complexité en $\mathcal{O}(n \log n)$

# III) Autres tris.

## 1) Avec des hypothèses sur l'entrée.

• tri par dénombrement : on suppose que les clefs sont des entiers compris entre 0 et N fixé. On construit un tableau C linéairement tel que C[i] contient le nombre de clefs inférieures ou égales à i $\in [0; N]$.
Ainsi, on connaît directement la position d'un élément x dans le tableau de sortie.

```
counting_sort(v):
    C = [0, 0, ..., 0] de taille N
    Pour i = 0 à (taille(v) - 1) :
        C[v(i)] := c[v(i)] + 1
    Pour i = 1 à (N-1) :
        C[i] := C[i] + C[i-1]
    B = [0, 0, ..., 0] de taille taille(v)
    Pour i = (taille(v) - 1) à 0 :
        B[c[v(i)]] = v(i)
        C[v(i)] = C[v(i)] - 1
```

- tri stable
- Complexité : $\mathcal{O}(N+n)$

• tri par paquets : on suppose que les clefs sont des flottants répartis uniformément sur l'intervalle $[0, 1[$. On les sépare suivant l'intervalle de la forme $[\frac{k}{n}, \frac{k+1}{n}[$ auquel ils appartiennent. On trie chaque case par insertion puis on concatène les n cases.

- Complexité en espérance : $\mathcal{O}(n)$

• tri par base : les clefs sont constituées de K sous-clefs entières bornées, de poids plus ou moins fort. On trie par dénombrement selon chaque sous-clefs, en commençant par celles de poids faible.

- tri stable
- Complexité : $\mathcal{O}(n)$

Rq 7 : La stabilité du tri par dénombrement est essentielle.

ex 8 : entiers en binaire / dates au format JJ.MM.AAAA

## 2) Tri topologique.

Entrée : $G = (S, A)$ un graphe orienté acyclique.

Sortie : Une liste L constituée des éléments de S telle que : Si $(u,v) \in A$, Alors u apparaît avant v dans L.

DEV : algorithme du tri topologique et correction

Rq 9 : Un graphe orienté acyclique est la représentation d'un ordre partiel sur ses sommets.

## 3) Tri externe.

Problématique : Les données sont trop volumineuses pour tenir en mémoire centrale. Elles sont donc stockées sur un support externe pour lequel l'accès et l'écriture sont des opérations coûteuses.

Principe : On charge successivement en mémoire des sous-listes qu'on sait trier et recopier sur le support. Puis on fusionne les sous-listes ordonnées sur le support.