Algorithmes de tris

David Auger, Jérémie Cabessa

Laboratoire DAVID, UVSQ

Introduction

- Importance des algorithmes de tris en informatique: très utilisés en pratiques et utile pour la compréhension de l'algorithmique.
- On ne trie pas que des nombres, mais tout types de données sur lequel on peut mettre un ordre total.

Introduction

- Importance des algorithmes de tris en informatique: très utilisés en pratiques et utile pour la compréhension de l'algorithmique.
- On ne trie pas que des nombres, mais tout types de données sur lequel on peut mettre un ordre total.

Tri à bulles

- **Idée:** Permutations répétées d'éléments contigus qui ne sont pas dans le bon ordre. À chaque i-ème parcours du tableau, on pousse le i-ème plus grand éléments vers la droite, à sa bonne place.

Gif from Wikipedia

Tri à bulles

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Complexité: $\mathcal{O}(n^2)$

Tri à bulles

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Complexité: $\mathcal{O}(n^2)$

Tri à bulles

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Complexité: $\mathcal{O}(n^2)$

Tri à bulles

```python
def tri_bulles(tab):

    for k in range(len(tab) - 1):           # parcours tableau

        permut = False

        for i in range(len(tab) - k - 1):   # parcours éléments non-triés

            if tab[i] > tab[i+1]:           # si éléments mal triés

                tab[i], tab[i+1] = tab[i+1], tab[i] # alors permutation
                permut = True

        if permut == False:  # si plus de permutations

            break            # alors tableau trié

    return tab
```

Tri par insertion

- **Idée 1:** Cartes déjà triées dans la main gauche et cartes non-triées posées sur la table à droite. On prend les cartes de droite une par une pour les insérer à leur bonne place dans la main gauche.

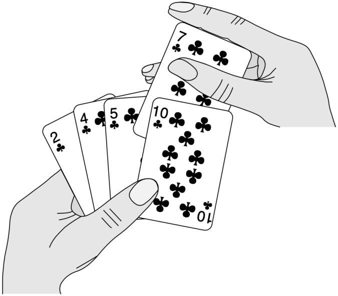

Figure 2.1 Tri de cartes à jouer, via tri par insertion.

Tri par insertion

- **Idée 2:** On pioche le i-ème élément `x`, ce qui laisse un trou dans la partie gauche du tableau `t[0:i]`. Puis on décale le trou à gauche et on re-insère `x` dans le trou lorsque la bonne position est trouvée.

Gif from Wikipedia

Tri par insertion

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Tri online: contrairement au tri à bulles, on peut donner les éléments un par un.
- Tri efficace si données presque triées: boucles `while` courtes.
- Complexité: $\mathcal{O}(n^2)$

Tri par insertion

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Tri online: contrairement au tri à bulles, on peut donner les éléments un par un.
- Tri efficace si données presque triées: boucles `while` courtes.
- Complexité: $\mathcal{O}(n^2)$

Tri par insertion

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Tri online: contrairement au tri à bulles, on peut donner les éléments un par un.
- Tri efficace si données presque triées: boucles `while` courtes.
- Complexité: $\mathcal{O}(n^2)$

Tri par insertion

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Tri online: contrairement au tri à bulles, on peut donner les éléments un par un.
- Tri efficace si données presque triées: boucles `while` courtes.
- Complexité: $\mathcal{O}(n^2)$

Tri par insertion

- Tri en place (in place): pas de mémoire extérieure.
- Tri stable: éléments de même valeur non inversés.
- Tri online: contrairement au tri à bulles, on peut donner les éléments un par un.
- Tri efficace si données presque triées: boucles `while` courtes.
- Complexité: $\mathcal{O}(n^2)$

Tri par insertion

```
def tri_insertion(tab):

    for i in range(1, len(tab)):

        x = tab[i]          # élément pioché avec la main droite
        j = i               # à insérer dans la main gauche (tab[0:i] trié)

        while (j > 0) and (tab[j-1] > x):   # tant que x mal placé
                                             # dans la main gauche
            tab[j] = tab[j-1]                # on recule...
            j -= 1

        tab[j] = x                           # insère x à sa bonne position

    return tab
```

Tri par sélection

- **Idée:** On cherche le min, puis on le place en début de tableau par permutation. On répète l'opération pour placer le second min dans la deuxième case, etc.

Gif from Wikipedia

Tri par sélection

- Tri en place (in place): pas de mémoire extérieure.
- Tri non stable: éléments de même valeur inversés.
- Peut être intéressant si comparaisons faciles mais échanges lents (peu utilisé en pratique).
- Complexité: $\mathcal{O}(n^2)$

Tri par sélection

- Tri en place (in place): pas de mémoire extérieure.
- Tri non stable: éléments de même valeur inversés.
- Peut être intéressant si comparaisons faciles mais échanges lents (peu utilisé en pratique).
- Complexité: $\mathcal{O}(n^2)$

Tri par sélection

- Tri en place (in place): pas de mémoire extérieure.
- Tri non stable: éléments de même valeur inversés.
- Peut être intéressant si comparaisons faciles mais échanges lents (peu utilisé en pratique).
- Complexité: $\mathcal{O}(n^2)$

Tri par sélection

- Tri en place (in place): pas de mémoire extérieure.
- Tri non stable: éléments de même valeur inversés.
- Peut être intéressant si comparaisons faciles mais échanges lents (peu utilisé en pratique).
- Complexité: $\mathcal{O}(n^2)$

Tri par sélection

```
def tri_selection(tab):
    for i in range(len(tab) - 1):       # parcours tableau
        idx_min = i                     # indice minimum actuel
                                        # partie gauche du tableau triée
        for j in range(i+1, len(tab)):  # parcours partie droite tableau
            if tab[j] < tab[idx_min]:   # si plus petit que min actuel
                idx_min = j             # alors mise à jour indice min actuel
        tab[i], tab[idx_min] = tab[idx_min], tab[i] # permutation
    return tab
```

Tri par fusion (ou tri dichotomique)

- **Idée:** *Diviser pour régner (divide and conquer).* On procède de manière récursive. On sépare le tableau en 2 sous-tableaux (gauche et droite), on trie chacun des sous-tableau (appel récursif), et on fusionne les 2 sous-tableaux triés. Lorsque les sous-tableaux sont de taille 1, il n'y a rien à faire.

Gif from Wikipedia

Tri par fusion (ou tri dichotomique)

- Tri pas en place (not in place): nécessite des tableaux externes pour les appels récursifs.
- Tri stable: éléments de même valeur non-inversés.
- Complexité: $\mathcal{O}(n \log(n))$

Tri par fusion (ou tri dichotomique)

- Tri pas en place (not in place): nécessite des tableaux externes pour les appels récursifs.
- Tri stable: éléments de même valeur non-inversés.
- Complexité: $\mathcal{O}(n \log(n))$

Tri par fusion (ou tri dichotomique)

- Tri pas en place (not in place): nécessite des tableaux externes pour les appels récursifs.
- Tri stable: éléments de même valeur non-inversés.
- Complexité: $\mathcal{O}(n\log(n))$

Tri par fusion

```python
def fusion(tab1, tab2):

    n1, n2 = len(tab1) , len(tab2)
    tab = [None] * (n1 + n2)        # tableau final
    i1, i2 = 0, 0                   # indices de tab1 et tab2

    for i in range(n1 + n2):        # remplir tableau final

        # si tab2 fini ou tab1[i1] < tab2[i2]
        if i2 == n2 or (i1 < n1 and  tab1[i1] < tab2[i2]):

            # compléter tab avec éléments restants de tab1
            tab[i] = tab1[i1]
            i1 += 1

        # sinon, compléter tab avec éléments restants de tab2
        else:

            tab[i] = tab2[i2]
            i2 += 1

    return tab
```

Tri par fusion

```
def tri_fusion(tab):

    if len(tab) <= 1:

        return tab

    else:

        tab_g = tri_fusion(tab[0: len(tab)//2])          # appel récursif
        tab_d = tri_fusion(tab[len(tab)//2: len(tab)])   # appel récursif
        return fusion(tab_g, tab_d)                      # fusion
```

Tri par fusion: complexité

- Preuve de la complexité du tri par fusion en $\mathcal{O}(n\log(n))$: on a $\mathcal{O}(\log(n))$ appels récursifs, chacune faisant appel à la procédure `fusion(...)` qui est (au plus) en $\mathcal{O}(n)$. On a donc une complexité en $\mathcal{O}(n\log(n))$.

Fusion itérative (variante du tri par fusion)

- **Idée:** On fusionne les paires de sous-tableaux de tailles successives $1, 2, 4, 8, \ldots, 2^k, \ldots$
- Procédure non récursive, dite "de bas en haut".

Fusion itérative (variante du tri par fusion)

- **Idée:** On fusionne les paires de sous-tableaux de tailles successives $1, 2, 4, 8, \ldots, 2^k, \ldots$
- Procédure non récursive, dite “de bas en haut”.

Timsort (variante du tri par fusion)

- **Idée:** On cherche d'abord les sous-tableaux monotones (donc déjà triés). Puis on fusionne les paires de sous-tableaux monotones comme dans le cas de la fusion itérative.
- Procédure non récursive, dite “de bas en haut”.

Timsort (variante du tri par fusion)

- **Idée:** On cherche d'abord les sous-tableaux monotones (donc déjà triés). Puis on fusionne les paires de sous-tableaux monotones comme dans le cas de la fusion itérative.
- Procédure non récursive, dite "de bas en haut".