NSI - Première

Tri par insertion et méthodes natives

qkzk

2021/06/02

Le tri par insertion est un autre algorithme de tri.

Algorithme

Tri par insertion

```
Je débute avec un tableau non trié plein et un tableau trié vide
Tant qu'il y a des objets non triés :
    Je choisis un objet
    Je l'insère parmi les objets triés de telle sorte que celui-ci reste trié
```

Insérer un élément dans un tableau trié

```
Entrée : des objets triés, un élément e à insérer
Sortie : aucune

Je prends l'objet le plus à droite (le plus grand)
Tant qu'il est plus grand que l'élément e :
    prendre l'objet à gauche de celui  que je tiens

Insérer e à droite de l'objet courant.
```

Exemple

Partons d'un tableau non trié comme `[5, 1, 3, 7]`

Le tableau trié est vide : `[]`

- On choisit 5. On l'insère dans le tableau trié : `[5], [1, 3, 7]`
- On choisit 1. On le compare avec 5, il est plus petit, il va à gauche. Il n'y a rien d'autre à comparer, on a fini : `[1, 5], [3, 7]`
- On choisit 3. On le compare à 5 (plus petit), on le compare à 1 (plus grand). 3 va à droite de 1 : `[1, 3, 5], [7]`
- On choisit 7. On le compare à 5 (plus grand), 7 va à droite de 5. `[1, 3, 5, 7], []`

On a terminé, il ne reste aucun objet à trier.

Tri par insertion : version avec indice (à retenir)

```
Fonction tri par insertion (tableau t):

    i = 1
    Pour i allant de 1 à longueur de t - 1:
        j = i
        Tant que j > 0 et t[j - 1] > t[j] :
            echanger(t, j - 1, j)
            j = j - 1
        i = i + 1
```

Fonction échanger

```
Fonction échanger(tableau t, indice k, indice l):

    temp = t[k]
    t[k] = t[l]
    t[l] = temp
```

Remarquez bien que les fonctions précédentes ne renvoient rien et modifient le tableau qu'elles reçoivent en paramètre.

Le tri natif en Python

Comme tous les langages modernes, Python propose des outils pour trier des tableaux. Il utilise pour cela un autre algorithme appelé Timsort, qu'on retrouve dans Java, Javascript, Swift et Rust.

Trier en place avec la méthode `sort`

```
>>> tableau = [5, 3, 1, 7]
>>> tableau.sort()      # modifie tableau, ne renvoie rien.
>>> tableau
[1, 3, 5, 7]
```

Créer une *copie* trié avec la fonction `sorted`

```
>>> tableau = [5, 3, 1, 7]
>>> sorted(tableau)     # renvoie une copie triée
[1, 3, 5, 7]
```

Notez bien les distinctions entre les deux outils.

Les paramètres optionnels `key` et `reverse`

Lorsqu'on trie des objets complexes, on peut spécifier une clé pour les trier.

Par exemple :

```
>>> mots = ["bbb", "aaaa", "d", "cc"]
>>> sorted(mots)        # par défaut, l'odre lexicographique (lettres puis chiffres)
["aaaa", "bbb", "cc", "d"]
>>> sorted(mots, key=len) # on trie selon la longueur
["d", "cc", "bbb", "aaaa"]
```

`key` est une fonction qui doit pouvoir s'appliquer à chaque objet du tableau.

On peut définir ses propres clés :

```
>>> points = [(1, 2), (0, 3), (3, 1)]
>>> sorted(points, key=lambda point: point[1])  # on trie selon la seconde coordonnée
[(3, 1), (1, 2), (0, 3)]
```

On a ici utilisé une fonction *lambda*, à comprendre dans le sens "quelconque", donc anonyme, qui associe à chaque point sa seconde coordonnée.

Quel serait l'ordre obtenu si on remplaçait `lambda point: point[1]` par `lambda point: point[0]` ?

Les deux outils présentés acceptent aussi un paramètre booléen optionnel `reverse`.

Lorsque `reverse=True` est passé en paramètre, le tri est effectué dans l'ordre décoissant.

```
mots = [3, 4, 2, 1]
sorted(mots, reverse=True)
[4, 3, 2, 1]
```