Tris

Résumé

Trier :

Algorithme de **tri**

> Algorithme qui, partant d'une liste, renvoie une version ordonnée de la liste.

$$[5, 1, 4, 3, 2] \rightarrow [1, 2, 3, 4, 5]$$

Tri par sélection

```
Je débute avec un tableau non trié plein et un tableau trié vide.
Tant qu'il y a des objets non triées :
   Je cherche le plus petit des objets non triés,
   Je le place à la suite des objets déjà triée.
fin Tant que
```

Le plus petit des objets non triés

```
Entrée : Des objets
Sortie : L'objet le plus petit

Je prends un objet
Pour chacun des autres:
    S'il est plus petit que l'objet choisi
      Alors j'échange.
    Fin Si
    Je mets l'autre de côté.
Fin Pour
```

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.
```

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)]
```

Codes des algorithmes du tri par sélection et par insertion

Les tris

```python
def tri_select(tab: list) -> None:
    """
    Effectue un tri par sélection de tab.
    Modifie `tab` en place.
    Ne renvoie rien
    """
    for i in range(len(tab) - 1):
        mini = i
        for j in range(i + 1, len(tab)):
            if tab[j] < tab[mini]:
                mini = j
        tab[i], tab[mini] = tab[mini], tab[i]


def tri_insertion(tab: list) -> None:
    """
    Effectue un tri par insertion de tab.
    Modifie `tab` en place.
    Ne renvoie rien.

    Version très peu efficace qui réalise beaucoup d'échanges inutiles.
    Sera améliorée en TP.
    """
    for i in range(len(tab)):
        j = i
        while j > 0 and tab[j] < tab[j - 1]:
            tab[j], tab[j - 1] = tab[j - 1], tab[j]
            j -= 1


def tri_insertion_ameliore(tab: list) -> None:
    """
    Effectue un tri par insertion de tab.
    Modifie `tab` en place.
    Ne renvoie rien.

    Version très peu efficace qui réalise beaucoup d'échanges inutiles.
    Sera améliorée en TP.
    """
    for i in range(len(tab)):
        j = i
        val = tab[j]
        while j > 0 and val < tab[j - 1]:
            tab[j] = tab[j - 1]
            j -= 1
        tab[j] = val
```

Fonctions pour tester les codes précédents

```python
def tableau_aleatoire(taille: int, elem_max: int) -> list:
    """
    Renvoie un tableau aléatoire d'entiers entre 1 et `elem_max` inclu contenant `taille` éléments.
    """
    from random import randint

    return [randint(1, elem_max) for _ in range(taille)]


def est_trie(tab: list) -> bool:
    """Vrai ssi `tab` est trié par ordre croissant."""
    for i in range(len(tab) - 1):
        if tab[i] > tab[i + 1]:
            return False
    return True


def test():
    """Teste les fonctions de tri."""

    assert est_trie([1])
    assert est_trie([])
    assert est_trie([1, 2, 3])
    assert not est_trie([3, 2, 1])
    assert not est_trie([3, 1, 2])

    tab = [3, 1, 5]
    tri_select(tab)
    assert est_trie(tab)

    for _ in range(100):
        tab = tableau_aleatoire(10, 10)
        tri_select(tab)
        assert est_trie(tab)

    tab = [3, 1, 5]
    tri_insertion(tab)
    assert est_trie(tab)

    for _ in range(100):
        tab = tableau_aleatoire(10, 10)
        tri_insertion(tab)
        assert est_trie(tab)

    tab = [3, 1, 5]
    tri_insertion_ameliore(tab)
    assert est_trie(tab)

    for _ in range(100):
        tab = tableau_aleatoire(10, 10)
        tri_insertion_ameliore(tab)
        assert est_trie(tab)


if __name__ == "__main__":
    test()
```

Coût des algorithmes de tri

Le coût des algorithmes de *parcours séquentiels* est *linéaire.*

Il progresse linéairement avec la taille du tableau.

S'il faut 10 secondes pour obtenir le max d'un tableau d'un million d'éléments, il faudra 20 secondes pour obtenir le max d'un tableau de deux millions d'éléments...

Le coût du tri par sélection et du tri par insertion est *quadratique*

Cela signifie qu'il progresse avec le *carré* de la taille du tableau.

S'il faut 5 secondes pour trier 1 000 éléments, combien faut-il de temps pour en trier 10 000 ?

$\frac{10\ 000}{1\ 000} = 10$ donc on multiplie par 10^2 : il faudra $5 \times 10^2 = 500$ secondes.

Mesure empirique du coût

On détermine le pire cas possible : celui d'un tableau trié à l'envers (cf TP sur capytale)

Deux approches possibles :

1. Mesurer le temps
2. Compter le nombre d'opérations.

Pour mesurer le temps :

1. On crée un tableau comme il faut,
2. On lance le chrono,
3. On trie,
4. On arrête le chrone.
5. La durée est la différence :

```
from time import time

# tableau trié à l'envers
tab = list(range(10000))
tab.reverse()

start = time()
tri_select(tab)
end = time()
duree = end - start
print(duree)
```

Pour compter le nombre d'échanges

1. On insère un compteur initialisé à 0 au début de la fonction,
2. On incrémente le compteur à chaque opération mesurée
3. On renvoie le compteur à la fin.

Par exemple pour compter les *comparaisons* du tri par sélection.

```python
def tri_select_avec_compteur(tab: list) -> None:
    compteur = 0
    for i in range(len(tab) - 1):
        mini = i
        for j in range(i + 1, len(tab)):
            if tab[j] < tab[mini]:
                compteur += 1
                mini = j
        tab[i], tab[mini] = tab[mini], tab[i]
    return compteur
```

Et pour le tester, on faut comme au dessus, sauf qu'on n'a plus besoin de relever le temps.