Tris naïfs

Résumé

Classer des copies par ordre alphabétique, ranger des fichiers par date, organiser un classement sportif : **trier** des données est l'une des opérations les plus courantes en informatique. Un algorithme de tri réorganise les éléments d'une liste dans un ordre donné (croissant, alphabétique, etc.). Il existe de nombreuses façons de trier, plus ou moins rapides. Cette fiche présente les deux algorithmes de tri au programme de NSI : le **tri par sélection** et le **tri par insertion**. Ils sont simples à comprendre et permettent d'acquérir les réflexes fondamentaux (comparer, échanger, maintenir un invariant) que l'on retrouve dans tous les algorithmes plus avancés.

Prérequis : Les listes, Recherche d'extremum.

① Mémo

Tri par sélection

Principe À chaque étape, on sélectionne le minimum de la partie non triée et on l'échange avec le premier élément de cette partie.

Invariant Après l'étape k, les k plus petits éléments sont à leur place définitive dans `L[0:k]`.

Complexité $O(n^2)$ dans **tous** les cas : toujours $\frac{n(n-1)}{2}$ comparaisons.

Tri par insertion

Principe On parcourt la liste de gauche à droite. Pour chaque élément, on l'insère à sa place dans la partie déjà triée (à gauche) en décalant les éléments plus grands.

Invariant Après l'étape i, les éléments `L[0:i+1]` sont triés entre eux.

Complexité $O(n)$ si la liste est presque triée (quelques décalages), $O(n^2)$ dans le pire cas (liste inversée).

Propriété 1 (Comparaison des deux tris)

- Le tri par sélection effectue toujours le même nombre de comparaisons, quel que soit l'état initial de la liste.
- Le tri par insertion s'adapte : il est quasi-linéaire sur une liste presque triée.
- Les deux sont en $O(n^2)$ dans le pire cas, ce qui les rend inadaptés aux grandes listes.

⚠ Erreurs classiques

Code erroné	Code correct	Explication
`L2 = L.sort()` puis utiliser `L2`	`L2 = sorted(L)`	`.sort()` trie en place et renvoie `None` ; `sorted()` renvoie une nouvelle liste triée. `L2` vaut donc `None`.
Tri par sélection : `L[i] = L[i_min]` `L[i_min] = L[i]`	`L[i], L[i_min] = L[i_min], L[i]`	Sans échange simultané, la première ligne écrase `L[i]` : on perd la valeur qui devait aller dans `L[i_min]`.
Tri par insertion : `while L[j] > cle:` sans `j >= 0`	`while j >= 0 and L[j] > cle:`	Sans la condition `j >= 0`, on accède à `L[-1]` (dernier élément en Python) au lieu de s'arrêter : résultat faux.
Tour de boucle inutile : `for i in range(n):` dans le tri par sélection	`for i in range(n - 1):`	`range(n)` fait un dernier tour sans effet : `range(n - 1)` suffit, car le dernier élément est automatiquement en place quand tous les autres sont triés. Le tri reste correct, mais non optimal.

② Exemples

Programme 1 · Tri par sélection

```
def tri_selection(L):
    n = len(L)
    for i in range(n - 1):
        i_min = i
        for j in range(i + 1, n):
            if L[j] < L[i_min]:
                i_min = j
        L[i], L[i_min] = L[i_min], L[i]
```

Trace sur `[5, 3, 8, 1, 4]` :

Étape	Zone non triée	Min trouvé	Liste après échange
1	[5, 3, 8, 1, 4]	1 (indice 3)	[**1**, 3, 8, 5, 4]
2	[3, 8, 5, 4]	3 (en place)	[1, **3**, 8, 5, 4]
3	[8, 5, 4]	4 (indice 4)	[1, 3, **4**, 5, 8]
4	[5, 8]	5 (en place)	[1, 3, 4, **5**, 8]

Programme 2 · Tri par insertion

```
def tri_insertion(L):
    for i in range(1, len(L)):
        cle = L[i]
        j = i - 1
        while j >= 0 and L[j] > cle:
            L[j + 1] = L[j]
            j -= 1
        L[j + 1] = cle
```

Trace sur `[5, 3, 8, 1, 4]` :

Étape	Clé	Décalages	Liste après insertion
1	3	5 décalé	[**3**, 5, 8, 1, 4]
2	8	aucun	[3, 5, **8**, 1, 4]
3	1	8, 5, 3 décalés	[**1**, 3, 5, 8, 4]
4	4	8, 5 décalés	[1, 3, **4**, 5, 8]

③ Exercices

Exercice 1 *Trace du tri par sélection* Donner l'état de la liste `[7, 2, 5, 1, 3]` après chaque passage du tri par sélection. Indiquer à chaque étape quel échange est effectué.

Exercice 2 *Trace du tri par insertion* Donner l'état de la liste `[6, 2, 8, 4, 1]` après chaque insertion. Indiquer à chaque étape le nombre de décalages.

Exercice 3 *Vérifier qu'une liste est triée* Écrire une fonction `est_triee(L)` qui renvoie `True` si `L` est triée par ordre croissant (au sens large), `False` sinon.

Exercice 4 *Comparaison expérimentale des tris*

1. Écrire une fonction `tri_selection_compteur(L)` qui trie et renvoie le nombre de comparaisons.
2. Écrire une fonction `tri_insertion_compteur(L)` qui trie et renvoie le nombre de comparaisons.
3. Comparer les deux sur trois types de listes de taille 20 : aléatoire, déjà triée, triée à l'envers. Quelle conclusion en tirer ?

Solutions des exercices

Corrigé de l'exercice 1.

1. Minimum de `[7, 2, 5, 1, 3]` = `1` (indice 3). Échange `L[0]` et `L[3]` : `[1, 2, 5, 7, 3]`.
2. Minimum de `[2, 5, 7, 3]` = `2` (indice 1), déjà en place : `[1, 2, 5, 7, 3]`.
3. Minimum de `[5, 7, 3]` = `3` (indice 4). Échange `L[2]` et `L[4]` : `[1, 2, 3, 7, 5]`.
4. Minimum de `[7, 5]` = `5` (indice 4). Échange `L[3]` et `L[4]` : `[1, 2, 3, 5, 7]`.

Nombre total de comparaisons : $4 + 3 + 2 + 1 = 10 = \frac{5\times4}{2}$.

Corrigé de l'exercice 2.

1. Clé = 2. On décale 6. Résultat : `[2, 6, 8, 4, 1]`. (1 décalage.)
2. Clé = 8. Aucun décalage ($8 > 6$). Résultat : `[2, 6, 8, 4, 1]`. (0 décalage.)
3. Clé = 4. On décale 8 puis 6. Résultat : `[2, 4, 6, 8, 1]`. (2 décalages.)
4. Clé = 1. On décale 8, 6, 4, 2. Résultat : `[1, 2, 4, 6, 8]`. (4 décalages.)

Total des décalages : $1 + 0 + 2 + 4 = 7$.

Corrigé de l'exercice 3.

```python
def est_triee(L):
    for i in range(len(L) - 1):
        if L[i] > L[i + 1]:
            return False
    return True
```

Principe : on compare chaque élément au suivant. Dès qu'on trouve `L[i] > L[i+1]`, la liste n'est pas triée.

Vérification : `est_triee([1, 1, 2, 3])` renvoie `True`; `est_triee([1, 3, 2])` renvoie `False`; `est_triee([])` et `est_triee([5])` renvoient `True`.

Corrigé de l'exercice 4.

```python
def tri_selection_compteur(L):
    L = L[:]
    n, nb = len(L), 0
    for i in range(n - 1):
        i_min = i
        for j in range(i + 1, n):
            nb += 1
            if L[j] < L[i_min]:
                i_min = j
        L[i], L[i_min] = L[i_min], L[i]
    return nb

def tri_insertion_compteur(L):
    L = L[:]
```

```
    nb = 0
    for i in range(1, len(L)):
        cle = L[i]
        j = i - 1
        while j >= 0:
            nb += 1
            if L[j] > cle:
                L[j + 1] = L[j]
                j -= 1
            else:
                break
        L[j + 1] = cle
    return nb
```

Résultats pour $n = 20$:

- Sélection : 190 comparaisons dans **tous** les cas ($\frac{20\times19}{2} = 190$).
- Insertion sur liste triée : 19 comparaisons (aucun décalage).
- Insertion sur liste inversée : 190 comparaisons (pire cas).
- Insertion sur liste aléatoire : environ 100 comparaisons (cas moyen).

Conclusion : le tri par insertion s'adapte à l'état initial de la liste, pas le tri par sélection.