Algorithmes de tri

Algorithmique

Généralités sur les tris

Nous nous intéresserons au tri d'une liste d'objets qui sont munis d'un ordre total. C'est à dire que deux éléments sont toujours comparables grâce à une relation d'ordre. Parmi les types en Python munis d'un ordre total on peut citer :

— les entiers ;

— les flottants ;

— les chaînes de caractères.

Un **algorithme de tri** range les éléments d'une collection du plus petit au plus grand, en s'appuyant uniquement sur des comparaisons entre deux éléments. Pour des nombres, on compare avec « est inférieur ou égal à » ; pour des mots, on compare selon l'ordre du dictionnaire. Dans ce chapitre, la collection à trier est toujours une liste Python.

Exemples

— Voici une liste d'entier `[42, 3, 5, 9, 21, 9, 14]` qui une fois triée sera `[3, 5, 9, 9, 14, 21, 42]`. C'est la relation d'ordre $\leq$ qui est utilisée.

— Dans le cas d'une chaîne de caractères, c'est la relation d'ordre lexicographique. Dans le cas de la liste : `M = ["salut", "tout", "le", "monde"]` la liste triée sera `['le', 'monde', 'salut', 'tout']`.

— Attention car la comparaison des chaînes de caractères peut être biaisée par la casse (Majuscule / minuscules).

```
>>> 'a' < 'A'
False
>>> 'A' < 'a'
True
>>> 'B' < 'a'
True
>>> 'Z' < 'a'
True
>>> 'Bonjour' < 'bonjour'
True
```

Ainsi, selon vous, quel est le résultat du tri de la liste suivante ?

`N = ["Ah !", "Non !", "C'est", "un", "peu", "court", "jeune", "homme !"]`

Tri en place

Un tri est dit « en place » s'il est effectué directement dans la structure de donnée initiale, et ne nécessite pas l'allocation d'une nouvelle structure.

Tri stable

Un tri est dit stable s'il préserve l'ordonnancement initial des éléments que l'ordre considère comme égaux.

Prenons des élèves rangés par ordre alphabétique, que l'on veut trier par note croissante :

```
Alice 14   Bruno 12   Chloé 14   David 9   Emma 12
```

Un tri **stable** donne `David 9, Bruno 12, Emma 12, Alice 14, Chloé 14` : à note égale, les élèves restent dans l'ordre alphabétique. Un tri **instable** peut donner `David 9, Emma 12, Bruno 12, Chloé 14, Alice 14` : les notes sont rangées, mais l'ordre alphabétique des ex æquo est perdu. La stabilité permet de trier selon un critère, puis selon un autre, sans défaire le premier classement.

Échange de deux valeurs dans une liste

Écrire une fonction `echanger()` qui prend en paramètres une liste `L` et deux entiers `i` et `j` et qui renvoie la liste `L` où les éléments d'indice `i` et `j` ont été échangés.

```python
def echanger(L, i, j):
    """Échange les éléments aux indices i et j dans la liste L"""
    L[i], L[j] = L[j], L[i]
    return L
```

Vérifier qu'une liste est bien triée

Une liste `L` est bien triée si et seulement si :

$$\forall i, j \in [0, \dots, n-1], i < j \implies \mathrm{L}[i] \leq \mathrm{L}[j]$$

Écrire une fonction `est_triee()` qui prend en paramètre une liste `L` et renvoie le booléen `True` si la liste est triée, `False` sinon.

```python
def est_triee(L):
    """Vérifie si la liste L est triée en ordre croissant"""
    for i in range(len(L) - 1):
        if L[i] > L[i + 1]:
            return False
    return True
```

Tri par sélection

Le principe du **tri par sélection** sur une liste d'entiers est le suivant :

— on parcourt l'intégralité de la liste à la recherche du plus petit élément ;

— une fois *sélectionné* ce plus petit élément, on le permute avec le tout premier élément de la liste (celui d'indice 0) ; le plus petit élément de la liste est alors en première position ;

— on parcourt alors le reste de la liste pour sélectionner son plus petit élément, que l'on permute alors avec le deuxième élément de la liste (d'indice 1) ;

— on recommence jusqu'à placer l'avant-dernier élément à sa place et la tâche est terminée.

Le tri par sélection se fait en place et est non stable.

Exemple

Voici une illustration du fonctionnement du tri par sélection sur un tableau de 6 éléments :

Étape	Pos 0	Pos 1	Pos 2	Pos 3	Pos 4	Pos 5
Initiale	12	9	3	7	14	11
i = 0, min = 3	**3**	9	12	7	14	11
i = 1, min = 7	3	**7**	12	9	14	11
i = 2, min = 9	3	7	**9**	12	14	11
i = 3, min = 11	3	7	9	**11**	14	12
i = 4, min = 12	3	7	9	11	**12**	14

Trace détaillée :

— **Itération 0 :** Recherche du minimum dans [12, 9, 3, 7, 14, 11]. Minimum = 3 à l'indice 2. On échange 12 et 3.

— **Itération 1 :** Recherche du minimum dans [9, 12, 7, 14, 11]. Minimum = 7 à l'indice 3. On échange 9 et 7.

— **Itération 2 :** Recherche du minimum dans [12, 9, 14, 11]. Minimum = 9 à l'indice 3. On échange 12 et 9.

— **Itération 3 :** Recherche du minimum dans [12, 14, 11]. Minimum = 11 à l'indice 5. On échange 12 et 11.

— **Itération 4 :** Recherche du minimum dans [14, 12]. Minimum = 12 à l'indice 5. On échange 14 et 12.

— **Résultat final :** [3, 7, 9, 11, 12, 14]

Questions :

1. Observer qu'une fois que l'avant-dernier élément est en place, le dernier l'est aussi.
2. À vous de tracer les étapes du tri par sélection sur le tableau suivant : [9, 4, 5, 4, 1, 3]

Programme complet

```python
def tri_selection(L):
    '''Prend en argument une liste et la retourne triée par sélection'''
    assert type(L) == list, "Fournir un type list en argument !"
    for i in range(len(L) - 1):
        indice_du_min = i  # l'indice temporaire du plus petit élément
        for j in range(i + 1, len(L)):
            if L[j] < L[indice_du_min]:
                indice_du_min = j
        # échange l'élément à la position i avec le minimum trouvé
        echanger(L, i, indice_du_min)
```

```
    return L
```

Invariant de boucle : À la fin de l'itération `i`, les `i + 1` premiers éléments de la liste sont triés et à leur position définitive.

Tri par insertion

Dans le **tri par insertion**, le début de la liste est déjà rangé ; à chaque étape, on prend l'élément d'indice `i` et on le fait remonter jusqu'à sa place parmi ceux qui le précèdent.

C'est la méthode que l'on applique sans y penser quand on range des cartes à jouer au fur et à mesure qu'on les reçoit : on tient en main des cartes déjà classées, on prend la carte suivante (l'élément d'indice `i`), on la fait glisser vers la gauche tant qu'elle est plus petite que sa voisine, puis on la pose. On recommence avec la carte suivante jusqu'à la dernière.

Sur une longue liste en désordre, le tri par insertion devient vite trop lent : son coût est quadratique, et des tris plus élaborés comme le tri fusion (vu en terminale) font beaucoup mieux. Il garde pourtant deux qualités. Sur une liste courte, il est très rapide car chaque étape est simple. Sur une liste déjà presque rangée, chaque élément ne se déplace que de quelques cases, et le tri se fait en un temps presque linéaire. C'est pourquoi la fonction `sort` de Python l'utilise elle-même pour ranger les petits morceaux de liste avant de les fusionner.

Enfin, le tri par insertion est **stable** (un élément ne dépasse jamais un élément égal) et **en place** (il travaille dans la liste elle-même, sans en créer une autre).

Exemple

Voici une illustration du fonctionnement du tri par insertion sur un tableau de 6 éléments :

Étape	Pos 0	Pos 1	Pos 2	Pos 3	Pos 4	Pos 5
Initiale	12	9	3	7	14	11
i = 1, insérer 9	9	**12**	3	7	14	11
i = 2, insérer 3	**3**	9	12	7	14	11
i = 3, insérer 7	3	**7**	9	12	14	11
i = 4, insérer 14	3	7	9	12	**14**	11
i = 5, insérer 11	3	7	9	**11**	12	14

Trace détaillée :

— **Itération i = 1 :** Partie triée = [12], élément = 9. $9 < 12$, donc décalage. Résultat = [9, 12, 3, 7, 14, 11]

— **Itération i = 2 :** Partie triée = [9, 12], élément = 3. $3 < 12$ puis $3 < 9$, décalages. Résultat = [3, 9, 12, 7, 14, 11]

— **Itération i = 3 :** Partie triée = [3, 9, 12], élément = 7. $7 < 12$, décalage, $7 < 9$, décalage, $7 \geq 3$, stop. Résultat = [3, 7, 9, 12, 14, 11]

— **Itération i = 4 :** Partie triée = [3, 7, 9, 12], élément = 14. $14 \geq 12$, pas de décalage. Résultat = [3, 7, 9, 12, 14, 11]

— **Itération i = 5 :** Partie triée = [3, 7, 9, 12, 14], élément = 11. $11 < 14$, décalage, $11 < 12$, décalage, $11 \geq 9$, stop. Résultat = [3, 7, 9, 11, 12, 14]

— **Résultat final :** $[3, 7, 9, 11, 12, 14]$

Questions :

À vous de tracer les étapes du tri par insertion sur le tableau suivant : $[9, 4, 5, 4, 1, 3]$

Programme complet

```python
def tri_insertion(L):
    """
    Entrée : une liste d'éléments int ou float, ou str
    Sortie : la liste constituée des même éléments, triée par ordre croissant
    """
    for i in range(1, len(L)):
        en_cours = L[i]
        j = i
        # décalage des éléments de la liste vers la droite
        while j > 0 and L[j - 1] > en_cours:
            L[j] = L[j - 1]  # décalage
            j -= 1
        # on insère l'élément à sa place
        L[j] = en_cours
    return L
```

Invariant de boucle : À la fin de l'itération `i`, les `i + 1` premiers éléments de la liste sont triés (mais pas nécessairement à leur position définitive).

Complexité

La complexité d'un algorithme est une mesure du temps d'exécution (ou de l'espace mémoire) en fonction de la taille de l'entrée.

Tri par sélection

Pour chaque position `i` de 0 à $n - 2$:

— On parcourt les éléments de `i + 1` à `n - 1` pour trouver le minimum

— Nombre de comparaisons : $(n - 1) + (n - 2) + \ldots + 1 = \frac{n(n-1)}{2}$

Cas	Formule	Résultat
Meilleur cas	$\frac{n(n-1)}{2}$ comparaisons, $n - 1$ échanges (parfois sans effet)	$O(n^2)$
Pire cas	$\frac{n(n-1)}{2}$ comparaisons, $n - 1$ échanges	$O(n^2)$
Cas moyen	$\frac{n(n-1)}{2}$ comparaisons	$O(n^2)$

Le tri par sélection a toujours une complexité **quadratique** car il parcourt systématiquement la liste pour chaque position.

Tri par insertion

Pour chaque position `i` de 1 à $n - 1$:

— Dans le pire cas, on décale les i éléments précédents

— Total : $1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$ décalages

Cas	Comparaisons	Décalages	Complexité
Meilleur cas (liste triée)	$n-1$	0	$O(n)$
Pire cas (liste inversée)	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$O(n^2)$
Cas moyen	$\frac{n(n-1)}{4}$	$\frac{n(n-1)}{4}$	$O(n^2)$

Le tri par insertion est **linéaire** sur une liste presque triée et **quadratique** en général, ce qui le rend plus efficace que la sélection en pratique.

Comparaison résumée

Algorithme	Meilleur cas	Cas moyen	Pire cas	Stable	En place
Sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	Non	Oui
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Oui	Oui

Comparaison expérimentale

Le code suivant mesure les temps d'exécution des deux algorithmes sur des listes de tailles croissantes :

```python
import time
import random

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

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

# Tests de performance
tailles = [200, 400, 800, 1600]

for taille in tailles:
    # Génération d'une liste aléatoire
    L_aleatoire = [random.randint(0, 10000) for _ in range(taille)]

    # Test tri par sélection
    L1 = L_aleatoire.copy()
```

```
    debut = time.perf_counter()
    tri_selection(L1)
    temps_selection = time.perf_counter() - debut

    # Test tri par insertion
    L2 = L_aleatoire.copy()
    debut = time.perf_counter()
    tri_insertion(L2)
    temps_insertion = time.perf_counter() - debut

    ratio = temps_selection / temps_insertion
    print(f"Taille {taille:4d} | Sélection: {temps_selection:.4f}s | "
          f"Insertion: {temps_insertion:.4f}s | Ratio: {ratio:.2f}")
```

Résultats attendus : Pour une liste aléatoire, les deux algorithmes ont des performances similaires ($O(n^2)$). Cependant, le tri par insertion est plus rapide sur une liste presque triée.

Exercices

Exercice 1 Traçage manuel du tri par sélection

Tracez pas à pas l'exécution du tri par sélection sur la liste suivante :

Liste initiale : [15, 3, 9, 1, 12]

Pour chaque itération, indiquez :

— La partie triée (en gras)

— Le minimum trouvé dans la partie non triée

— L'état de la liste après l'échange

Exercice 2 Implémentation de la fonction `echanger`

Implémenter une fonction `echanger(L, i, j)` qui :

— Prend une liste `L` et deux indices `i` et `j`

— Échange les éléments aux indices `i` et `j`

— Retourne la liste modifiée

Testez votre fonction avec :

```
L = [5, 2, 8, 1]
echanger(L, 0, 3)  # Doit retourner [1, 2, 8, 5]
```

Exercice 3 Comparaison de stabilité

Soit une liste de tuples (nom, score) :

```
donnees = [("Alice", 90), ("Bob", 90), ("Charlie", 85)]
```

1. Triez cette liste par score en utilisant le tri par insertion
2. Triez cette liste par score en utilisant le tri par sélection

3. Comparez les résultats. Laquelle des deux ordonnances préserve l'ordre initial des personnes ayant le même score ?

Exercice 4 Analyse de complexité

Pour un tableau de taille $n = 1000$:

1. Calculez le nombre exact de comparaisons effectuées par le tri par sélection.
2. Calculez le nombre exact de comparaisons pour le tri par insertion dans le **pire cas**.
3. Si le tri par sélection prend 10 millisecondes, combien de temps (approximativement) le tri par sélection prendrait-il pour $n = 10000$?

Exercice 5 Preuve par invariant

Prouvez que le tri par sélection est correct en utilisant l'invariant :

Invariant : À la fin de l'itération `i`, les `i + 1` premiers éléments de la liste contiennent les `i + 1` plus petits éléments de la liste originale, **et ils sont triés**.

1. Montrez que l'invariant est vrai à l'initialisation (avant la première itération)
2. Montrez que si l'invariant est vrai après l'itération `i`, il reste vrai après l'itération `i + 1`
3. Montrez que l'invariant implique la correction de l'algorithme