Algorithmes de tris

S6 - Algorithmique (1)

Dans cette partie du cours, nous allons étudier deux algorithmes de tris : le tri par insertion et le tri par sélection.

Étant donné un tableau de nombres, l'objectif est d'écrire une fonction qui renvoie un tableau contenant les mêmes nombres mais dans l'ordre croissant.

1. Tri par insertion

Le principe

> **!** Principe de l'algorithme
>
> En commençant par le deuxième élément du tableau :
>
> - On compare l'élément courant avec l'élément précédent.
> - Si l'élément courant est plus petit, on échange les deux éléments.
> - On continue à comparer et échanger l'élément courant avec les éléments précédents jusqu'à ce que l'élément courant soit plus grand que l'élément précédent.

Animation du tri par insertion :

Exemple : Soit à trier le tableau $[5, 2, 7, 3]$.

- On commence par le deuxième élément du tableau, c'est-à-dire l'élément 2. On compare l'élément 2 avec l'élément 5. L'élément 2 est plus petit que l'élément 5, on échange les deux éléments. Le tableau est donc $[2, 5, 7, 3]$.
- On continue avec l'élément 7. L'élément 7 est plus grand que l'élément 5, on ne fait rien.
- On continue avec l'élément 3. L'élément 3 est plus petit que l'élément 7, on échange les deux éléments. Le tableau est donc $[2, 5, 3, 7]$. On continue à comparer et échanger l'élément 3 avec les éléments précédents jusqu'à ce que l'élément 3 soit plus grand que l'élément 5. Le tableau est donc $[2, 3, 5, 7]$. L'algorithme est terminé. Le tableau est trié.

Programmation

```
def tri_insertion(tableau: list) -> list:
    """Tri en place par insertion le tableau passé en paramètre."""
    for i in range(1, len(tableau)):                                    ①
        j = i                                                           ②
        while j > 0 and tableau[j] < tableau[j-1]:                      ③
            tableau[j], tableau[j-1] = tableau[j-1], tableau[j]         ④
            j -= 1                                                      ⑤
    return tableau
```

① On commence à l'indice 1 qui correspond au deuxième élément du tableau.
② On stocke l'indice courant dans une variable `j` pour pouvoir le modifier.

③ Tant que l'indice courant est supérieur à 0 et que l'élément courant est plus petit que l'élément précédent, on échange les deux éléments.
④ On échange les deux éléments.
⑤ L'élément courant est maintenant l'élément précédent, on décrémente donc l'indice courant.

Test de l'algorithme :

```
tri_insertion([5, 2, 4, 6, 1, 3])
```

```
[1, 2, 3, 4, 5, 6]
```

Preuve de terminaison

Montrons que l'algorithme se termine.

D'une part, il est certain que la boucle `for`, boucle bornée par nature, se termine. D'autre part, la boucle `while` se termine aussi. La variable `j` est un **variant de boucle**. À chaque itération, sa valeur diminue de 1 : elle finit donc toujours par atteindre 0.

La terminaison de l'algorithme est donc prouvée.

Preuve de correction

Montrons que l'algorithme trie bien le tableau.

Pour cela, considérons la propriété suivante : à chaque itération, le sous-tableau composé des `i` premiers éléments est trié. Montrons que cette propriété est un **invariant de boucle**.

- **Initialisation** : au début de l'algorithme, le sous-tableau composé uniquement du premier élément est trié.
- **Conservation** : supposons que le le sous-tableau composé des `i` premiers éléments est trié : $[e_0, e_1, ..., e_{i-1}]$ avec $e_0 \leqslant e_1 \leqslant ... \leqslant e_{i-1}$. L'algorithme considère alors l'élément e_i et le compare avec les éléments précédents. Si e_i est plus petit que e_{i-1}, on échange les deux éléments. On continue alors à comparer e_i avec les éléments précédents jusqu'à ce que e_i soit plus grand que l'élément précédent. Le sous-tableau composé des `i+1` premiers éléments est alors trié.
- **Conclusion** : à la fin de l'algorithme `i` a la valeur `n-1` ce qui correspond à l'indice du dernier élément du tableau. Le sous-tableau composé des `n` premiers éléments est donc trié. Or, `n` est le nombre d'éléments du tableau, donc le tableau entier est trié.

La correction de l'algorithme est donc prouvée.

Complexité

On recherche la complexité dans **le pire des cas**. Le pire des cas est le cas où le tableau de départ est rangé dans l'ordre décroissant.

Notons n la taille du tableau de départ.

La boucle `for` comporte $n-1$ itérations.

Dans le cas où le tableau de départ est rangé dans l'ordre décroissant, la boucle `while` comporte 1 opération, puis 2, puis 3, etc. jusqu'à $n-1$ opérations pour la dernière itération. On obtient donc la somme suivante pour le nombre total d'opérations :

$$1+2+3+...+(n-1)=\frac{n(n-1)}{2}$$

Sachant que $\frac{n(n-1)}{2}=\frac{n^2-n}{2}$, il s'agit donc d'une **complexité quadratique**, en $\mathcal{O}(n^2)$.

2. Tri par sélection

Le principe

> **!** Principe de l'algorithme
>
> En commençant par le premier élément du tableau :
>
> - On recherche le plus petit élément parmi les éléments suivants du tableau.
> - On échange l'élément courant avec le plus petit élément.

Exemple : Soit à trier le tableau $[5, 2, 3, 7]$.

- On commence par le premier élément, 5. On recherche le plus petit élément parmi les éléments suivants du tableau, c'est-à-dire 2. On échange 5 et 2 : le tableau est maintenant $[2, 5, 3, 7]$.
- L'élément courant est maintenant le deuxième du tableau, c'est donc encore 5. On recherche le plus petit élément parmi les éléments suivants du tableau, c'est-à-dire 3. On échange 5 et 3 : le tableau est maintenant $[2, 3, 5, 7]$.
- L'élément courant est maintenant le troisième du tableau, c'est donc encore 5. On recherche le plus petit élément parmi les éléments suivants du tableau, mais aucun n'est plus petit que 5. On ne fait donc rien.
- L'élément courant est maintenant le dernier du tableau, c'est donc 7. On ne fait donc rien et le tableau est trié.

Programmation

```
def tri_selection(tableau: list) -> list:
    """Trie en place par sélection le tableau passé en paramètre."""
    for i in range(len(tableau)):                                    ①
        min = i                                                      ②
        for j in range(i+1, len(tableau)):                           ③
            if tableau[j] < tableau[min]:                            ④
                min = j
        tableau[i], tableau[min] = tableau[min], tableau[i]          ⑤
    return tableau
```

① On commence à l'indice 0 qui correspond au premier élément du tableau.
② On stocke l'indice courant dans une variable `min` pour pouvoir le modifier.
③ On parcourt le tableau à partir de l'indice `i+1` jusqu'à la fin.
④ Si l'élément courant est plus petit que l'élément stocké dans `min`, on met à jour `min`.
⑤ On échange les deux éléments.

Test de l'algorithme :

```
tri_selection([5, 2, 4, 6, 1, 3])
```

```
[1, 2, 3, 4, 5, 6]
```

Preuve de terminaison

L'algorithme se termine puisqu'il comporte deux boucles `for` qui sont toutes deux bornées par nature.

Preuve de correction

Montrons que l'algorithme trie bien le tableau.

Pour cela, considérons la propriété suivante : à chaque itération, le sous-tableau composé des `i` premiers éléments est trié. Montrons que cette propriété est un **invariant de boucle**.

- **Initialisation** : au début de l'algorithme, le sous-tableau composé uniquement du premier élément est trié.
- **Conservation** : supposons que le le sous-tableau composé des `i` premiers éléments est trié : $[e_0, e_1, ..., e_{i-1}]$ avec $e_0 \leqslant e_1 \leqslant ... \leqslant e_{i-1}$. Par construction, tous les éléments suivants sont supérieurs à e_{i-1}. L'algorithme considère alors l'élément e_i et le compare avec les éléments suivants. Si un élément est plus petit que e_i, on échange les deux éléments. Le sous-tableau composé des `i+1` premiers éléments est alors trié.
- **Conslusion** : à la fin de l'algorithme `i` a la valeur `n-1` ce qui correspond à l'indice du dernier élément du tableau. Le sous-tableau composé des `n` premiers éléments est donc trié. Or, `n` est le nombre d'éléments du tableau, donc le tableau entier est trié.

La correction de l'algorithme est donc prouvée.

Complexité

On recherche la complexité dans **le pire des cas**. Le pire des cas est le cas où le tableau de départ est rangé dans l'ordre décroissant.

Notons n la taille du tableau de départ.

La boucle `for i` comporte $n-1$ itérations. La boucle `for j` comporte $n-1$ itérations pour la première itération, puis $n-2$ itérations pour la seconde itération, etc. jusqu'à 1 opération pour la dernière itération. On obtient donc la somme suivante pour le nombre total d'opérations :

$$n-1+(n-2)+(n-3)+...+1=\frac{n(n-1)}{2}$$

Sachant que $\frac{n(n-1)}{2}=\frac{n^2-n}{2}$, il s'agit donc d'une **complexité quadratique**, en $\mathcal{O}(n^2)$.

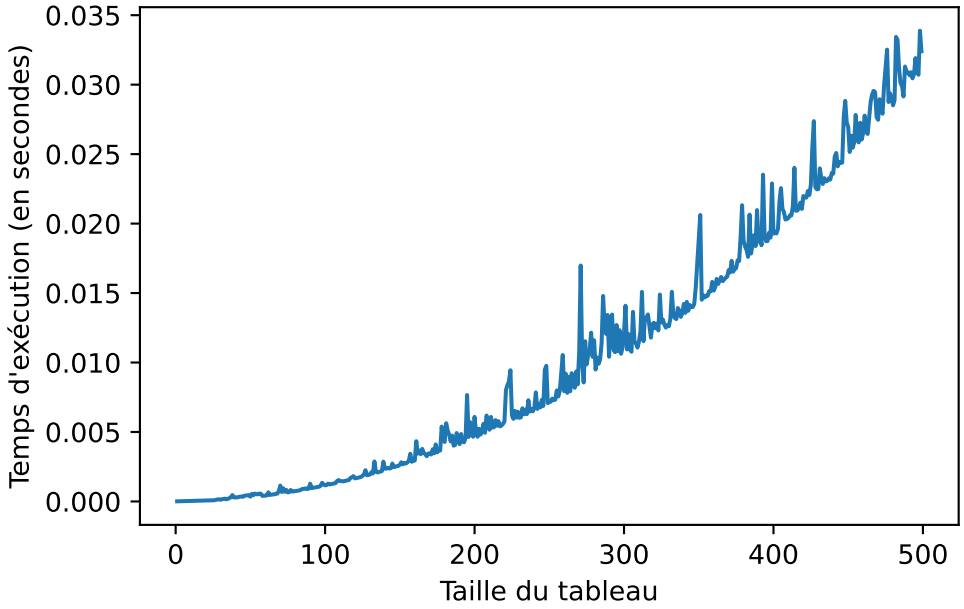
3. Observation expérimentale de la complexité

Tri par insertion

```
import timeit
import matplotlib.pyplot as plt

tailles = [i for i in range(1, 500)]
temps = []
# on applique le tri dans le pire des cas : tableau trié dans l'ordre décroissant
for n in tailles:
    temps.append(timeit.timeit(
```

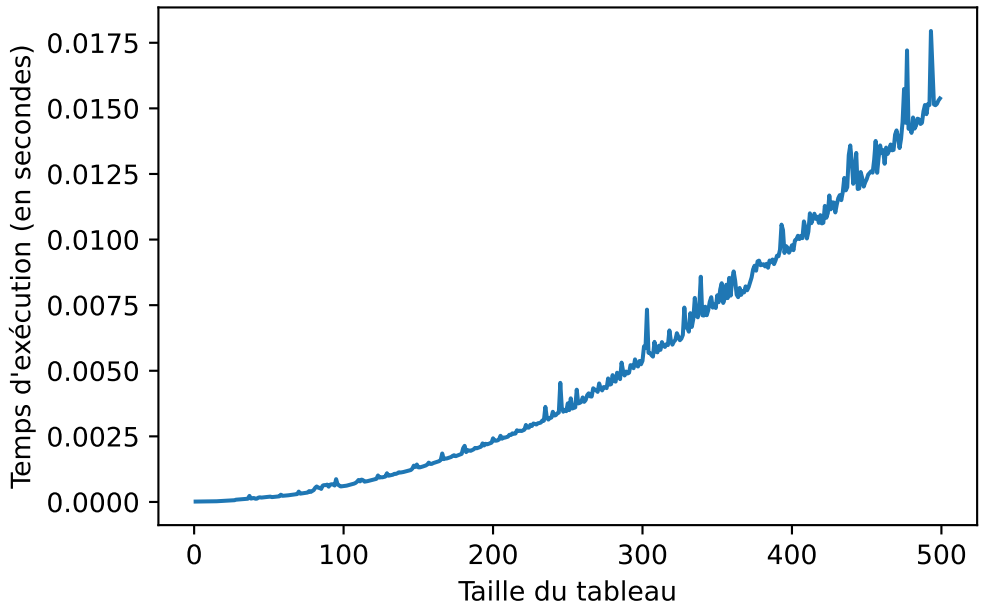
```
        "tri_insertion([n-k for k in range(n)])",
        globals=globals(),
        number=1
    ))
plt.plot(tailles, temps)
plt.xlabel("Taille du tableau")
plt.ylabel("Temps d'exécution (en secondes)")
plt.show()
```



Tri par sélection

```
import timeit
import matplotlib.pyplot as plt

tailles = [i for i in range(1, 500)]
temps = []
# on applique le tri dans le pire des cas : tableau trié dans l'ordre décroissant
for n in tailles:
    temps.append(timeit.timeit(
        "tri_selection([n-k for k in range(n)])",
        globals=globals(),
        number=1
    ))
plt.plot(tailles, temps)
plt.xlabel("Taille du tableau")
plt.ylabel("Temps d'exécution (en secondes)")
plt.show()
```



Dans les deux cas, la forme grossièrement parabolique de la courbe est caractéristique de la complexité quadratique.

Compléments

Sur le site interstices.info un article très complet sur les algorithmes de tri, avec des animations pour mieux comprendre.