Tris naïfs

Exercice 1 *Variantes du tri à bulles*

On s'intéresse au tri d'un tableau T. On suppose que les indices de T commencent à 0 et on appelle n la longueur du tableau T, c'est-à-dire son nombre d'éléments.

On rappelle le principe du tri à bulles : il consiste à appliquer à un tableau T un certain nombre de fois la procédure suivante (on appellera cela un passage vers la droite) :

```
def passageDroite(T):
    for j in range(len(T)-1):
        if T[j] > T[j+1]:
            T[j],T[j+1] = T[j+1],T[j]
```

Le tri à bulles consiste à appliquer $n-1$ passages vers la droite sur un tableau T de longueur n. Nous avons vu en cours que cet algorithme trie correctement, que sa complexité est $O(n^2)$, et que cette borne de complexité est optimale.

1. Si aucun échange n'est réalisé au cours d'un passage, que dire du tableau T ? En déduire un nouvel algorithme pour trier un tableau qui ne fait pas de passage superflu.
2. Déterminer la complexité de cette variante du tri à bulles. La borne obtenue est-elle optimale ?

On souhaite maintenant faire une nouvelle variante du tri à bulles, que nous appellerons tri à bulles optimisé. L'idée est de se souvenir de la position du dernier échange réalisé entre deux valeurs lors d'un passage pour économiser des étapes.

3. Proposer un algorithme réalisant le tri à bulles optimisé.
4. Démontrer que votre algorithme est correct.
5. Exprimer sa complexité avec $O(\cdot)$. La borne de complexité obtenue est-elle optimale ?

On définit également la procédure suivante :

```
def passageGauche(T):
  for j in range(len(T)-2,-1,-1):
    if T[j] > T[j+1]:
      T[j],T[j+1] = T[j+1],T[j]
```

On considère maintenant la fonction suivante :

```
def triBullesGD(T):
  for j in range(int(len(T)/2)):
    passageGauche(T)
    passageDroite(T)
```

6. Montrer que cette procédure trie le tableau donné en entrée par ordre croissant. En particulier, on donnera l'invariant de la boucle principale.
7. Déterminer la complexité dans le pire des cas de cet algorithme.

Exercice 2 *Tri par insertion*

On considère un tableau T indexé à partir de 0. Appelons n la longueur du tableau T. Le tri par insertion consiste, pour chaque $0 \leqslant i \leqslant n-1$, à insérer la valeur représentée en position i à sa place parmi les valeurs représentées aux positions 0 à $i-1$ qui sont supposées être déjà triées en ordre croissant (mais ne sont pas nécessairement les $i-1$ plus petites valeurs représentées dans le tableau).

Détaillons un peu le principe exposé ci-dessus. Un tableau à un seul élément est trié par définition. Pour $i = 1$, on cherche à positionner $T[1]$ correctement vis-à-vis de $T[0]$ de façon à obtenir un tableau trié constitué de ces deux éléments. Plus généralement, si $i \leqslant n-1$, on suppose que les i premiers éléments de T ont été réordonnés de façon à ce que l'on ait $T[0] \leqslant T[1] \leqslant \ldots \leqslant T[i-1]$. L'objectif est alors d'insérer $T[i]$ à sa place dans cette liste.

1. Écrire un algorithme prenant en entrée un tableau T et réalisant le tri de T par la méthode décrite ci-dessus (c'est le tri par insertion).
2. Donner la complexité de cet algorithme.
3. Écrire une variante du tri par insertion de façon à faire la recherche de la position d'insertion par un algorithme de recherche dichotomique (nous appellerons cet algorithme le tri par insertion optimisé).
4. Donner la complexité du tri par insertion optimisé.

Exercice 3 *Nombre de comparaisons des différents algorithmes*

On s'intéresse au nombre de comparaisons entre deux éléments du tableau faites par les différents algorithmes :

— tri à bulles (vu en cours) ;

— variante du tri à bulles ;

— tri à bulles optimisé ;

— tri sélection ou du maximum (vu en cours) ;

— tri par insertion ;

— tri par insertion optimisé.

1. Pour chacun de ces algorithmes de tri, étudier le nombre de comparaisons faites par l'algorithme dans le pire des cas. Exprimer cette quantité avec la notation $O(\cdot)$.
2. Montrer l'optimalité de chacune des bornes obtenues.
3. On souhaite trier des tableaux de grands nombres. On suppose de plus qui si le tableau à trier contient n éléments, chaque nombre est représenté par $k(n)$ bits avec $k(n) \gg \frac{n}{\log n}$. On suppose de plus que les éléments peuvent être échangés en temps $O(1)$ (comme c'est la cas en Python si ces grands nombres sont donnés par une liste contenant la suite de leurs bits). Donner un algorithme pour comparer deux nombres de $k(n)$ bits et déterminer sa complexité. Quel algorithme de tri choisir ? (Pour répondre à cette question, on prendra en compte le coût de toutes les opérations réalisées au cours du tri.)