EA4 – Éléments d'algorithmique
TD n° 4 : dichotomie et tris

Exercice 1 : occurrences et dichotomie

1. Écrire une fonction `occurrencesNaif(T, x)` qui renvoie le nombre d'*occurrences* (apparitions) d'un élément `x` dans un tableau `T`. Quelle est sa complexité dans le pire cas pour un tableau de taille n ?

On suppose maintenant que le paramètre `T` est un ***tableau trié***.

2. Que peut-on dire des occurrences d'un élément `x` dans `T` ?

3. Écrire une fonction `trouvePremier(T, x, begin=0, end=None)`, *la plus efficace possible*, qui renvoie `None` si `x` n'apparait pas dans `T` et sa première position dans le tableau sinon. Quelle est la complexité dans le pire cas pour un tableau de taille n ?

4. En déduire une fonction `occurrencesDicho(T, x)` qui renvoie le nombre d'occurrences de `x` dans `T` *le plus efficacement possible*. Quelle est la complexité de `occurrencesDicho(T,x)` dans le pire cas, pour un tableau de longueur n ?

Exercice 2 : anneau coupé

On dit qu'un tableau `T` de n éléments distincts est un anneau coupé s'il existe un certain indice `k` (inconnu a priori) pour lequel `T[k:] + T[:k]` est trié en ordre décroissant (pas forcément strict).

1. Dans un anneau coupé, quel est le nombre maximal d'indice `i` tel que `T[i-1] < T[i]` ?

2. En déduire un algorithme `est_un_anneau(T)` qui teste si `T` est un anneau coupé, et ayant comme complexité dans le pire case $\Theta(n)$.

On suppose maintenant que T est un anneau coupé.

3. Étant donné une position i de T, quel test permet de déterminer en temps constant si $i \leqslant m$, où m est la position (inconnue a priori) du minimum de T ?

4. En déduire un algorithme `minimum_anneau(T)` aussi efficace que possible pour déterminer la position du minimum d'un anneau coupé T.

5. Quelle est la complexité de cet algorithme pour un tableau de longueur n ?

6. En déduire un algorithme `maximum_anneau(T)` de complexité $\Theta(log(n))$ dans le pire cas qui retourne la position du maximum d'un anneau coupé `T`.

Exercice 3 : minimum local

Soit `T` un tableau de n éléments comparables, par exemple des entiers positifs. On dit que `T` possède un *minimum local en position i* si $\texttt{T}[i] \leqslant \texttt{T}[i-1]$ et $\texttt{T}[i] \leqslant \texttt{T}[i+1]$ (cela nécessite donc en particulier que $0 < i < n-1$).

1. Déterminer les 5 minima locaux du tableau

7	9	7	7	2	1	3	7	5	4	7	3	3	6

.

2. Écrire un algorithme `min_local(T)` qui renvoie un minimum local de `T` s'il en possède, et `None` sinon. Quelle est sa complexité dans le pire cas ?

On suppose dorénavant que `T` satisfait la propriété suivante :

$$\texttt{T[0]} \geqslant \texttt{T[1]} \text{ et } \texttt{T}[n-2] \leqslant \texttt{T}[n-1].$$

avec n la taille de T.

3. Montrer que sous cette hypothèse, `T` possède au moins un minimum local.

4. Soit $i > 0$ un indice tel que $\texttt{T}[i] > \texttt{T}[i-1]$. Montrer que $i + 1 \geqslant 3$.

5. Soit $i < n-1$ un indice tel que $\texttt{T}[i] > \texttt{T}[i+1]$. Montrer que $n - i \geqslant 3$.

6. En déduire un algorithme *le plus efficace possible* pour déterminer un minimum local d'un tel tableau. Quelle est sa complexité ?

Exercice 4 : quelques variantes du tri par insertion

On considère les deux descriptions suivantes du tri par insertion dans un tableau, où l'insertion est réalisée par échanges successifs :

```
def triInsertionParLaGauche(T) :
    for i in range(1, len(T)) :
        for j in range(i+1) :
            if T[i] < T[j] : break
        for k in range(j, i) :
            T[i], T[k] = T[k], T[i]

def triInsertionParLaDroite(T) :
    for i in range(1, len(T)) :
        for j in range(i, 0, -1) :     # pour j de i à 1 par pas de -1
            if T[j-1] <= T[j] : break
            T[j-1], T[j] = T[j], T[j-1]
```

1. Combien chacun de ces algorithmes fait-il de comparaisons dans le pire cas ? dans le meilleur cas ? Même question pour les échanges, et pour le cumul de ces deux types d'opérations. Que peut-on en déduire sur la complexité comparée des deux algorithmes ?

2. Montrer l'invariant suivant pour la boucle principale de l'une ou l'autre de ces deux versions :

> *« À la fin du tour de boucle d'indice `i`, le sous-tableau `T[:i+1]` contient les mêmes éléments qu'initialement, triés en ordre croissant. »*

3. Comment tirer parti de cet invariant pour diminuer le nombre de comparaisons effectuées dans le pire cas ?

4. Combien d'affectations un échange nécessite-t-il ? Si l'insertion de `T[`i`]` se fait en position k, combien d'affectations sont donc effectuées par les versions ci-dessus ?

5. Comment diminuer ce nombre d'affectations ?

6. Quel est l'effet de ces améliorations sur la complexité de l'algorithme ?