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. Décrire l'exécution de `trouvePremier` pour `T = [1, 2, 3, 4, 4, 5, 5, 7, 8]` et `x = 4`.

5. Écrire une fonction `occurrencesDicho(T, x)` qui renvoie le nombre d'occurrences de `x` dans `T` *le plus efficacement possible*. Il faudra se servir de `trouvePremier`. Quelle est la complexité de `occurrencesDicho(T,x)` dans le pire cas, pour un tableau de longueur n ?

Exercice 2 : 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) :
      if T[j-1] <= T[j] : break
      T[j-1], T[j] = T[j], T[j-1]
```

1. Décrire la dernière étape du tri par insertion de `T = [0, 2, 3, 5, 7, 8, 4]` (c'est-à-dire au moment de l'insertion de 4) pour chacun de ces deux algorithmes.

2. 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 ?

3. 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.* »

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

5. 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 ?

6. Comment diminuer ce nombre d'affectations ?

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

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 : tri à bulles

On considère une nouvelle méthode de tri, appelée *tri à bulles* :

```
def triBulles(T) :
    for i in range(len(T)-1, 0, -1) :
        for j in range(i) :
            if T[j] > T[j+1] : T[j], T[j+1] = T[j+1], T[j]
    return T
```

1. Appliquer cette méthode de tri sur le tableau `T = [5, 1, 4, 2, -5, 7, 6]`.

2. Expliquer le fonctionnement du tri à bulles. Exhiber l'invariant qui montre que l'algorithme renvoie bien une version triée de T.

3. Combien d'opérations élémentaires sont effectuées pour trier un tableau de taille `n` avec cet algorithme ?

4. Montrer que si aucun échange n'est fait à l'étape i, l'algorithme peut être arrêté.

5. Montrer que si, à l'étape i, aucun échange n'est fait après le rang j alors on peut réduire la borne de la boucle principale (sur i).

6. Écrire une fonction `triBullesOptimise` qui implémente ces deux optimisations. Quelle est sa complexité ?

7. Les grands éléments qui se trouvent au début du tableau remontent rapidement (on dit que ce sont des *lièvres*) alors que les petits éléments à la fin ne descendent que d'une case à chaque étape (on dit que ce sont des *tortues*). Proposer une version modifiée de `triBulles` appelée `triShaker` où les tortues avancent aussi vite que les lièvres.

8. Prouver sa correction et calculer sa complexité.