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 ?
2. On suppose maintenant que le paramètre `T` est un ***tableau trié***. Que peut-on dire des occurrences d'un élément `x` dans `T` ? En déduire une fonction `occurrencesDicho(T, x)` qui renvoie le nombre d'occurrences de `x` dans `T` *le plus efficacement possible.*

Exercice 2 : occurrences dans des tableaux d'entiers

On suppose ici que `T` est un tableau ***d'entiers positifs*** (a priori non trié).

1. En utilisant la fonction `occurrencesNaif` de l'exercice 1, écrire une fonction `compteNaif(T)`, qui renvoie un tableau d'entiers `S` tel que `S[i]` est le nombre d'occurrences de `i` pour chaque élément `i` présent dans `T`. Quelle doit être (au minimum) la taille de `S` ? Quelle est la complexité de `compteNaif` ?
2. Écrire une fonction améliorée `compteOptimal(T)` plus efficace que `compteNaif(T)`. Quelle est sa complexité ?
3. Comment vérifier le plus efficacement possible si un tableau `T` est une *permutation*, c'est-à-dire s'il contient exactement une fois chaque entier entre 1 et $n =$`len(T)` ?

Exercice 3 : 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 ?

On va maintenant étudier plus précisément `triInsertionParLaDroite`.

2. Montrer l'invariant suivant pour la boucle principale :

   « 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. »

Une inversion de `T` est un couple d'éléments `T[i] < T[j]` mal ordonnés, c'est-à-dire dont les positions vérifient $i > j$.

3. Quel est l'effet de l'échange `T[j-1], T[j] = T[j], T[j-1]` sur l'ensemble des inversions de `T`, et donc sur leur nombre ?

4. Combien cet algorithme fait-il d'échanges par inversion du tableau initial ? À combien d'affectations cela correspond-il ?

5. Comment diminuer ce nombre d'affectations ?

6. Combien de comparaisons l'algorithme effectue-t-il ?

7. Comment tirer parti de l'invariant montré à la question 2 pour diminuer le nombre de comparaisons effectuées dans le pire cas ?

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

Exercice 4 : tri à bulles (bonus)

Le tri à bulles est un algorithme de tri qui consiste à faire remonter progressivement les plus grands éléments d'un tableau. L'algorithme parcourt le tableau et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération `len(T)`-1 fois.

```
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]
```

1. Prouver sa correction et calculer sa complexité (évaluée en nombre de comparaisons d'éléments et d'affectations).

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

3. 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).

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

5. 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`, qu'on appellera `triShaker`, où les tortues avancent aussi vite que les lièvres.

6. Prouver sa correction et calculer sa complexité.