Informatique – T.D. 2

1. *Création d'un tableau aléatoire* : pour tester les programmes de tri, on peut bien sûr taper à la main les valeurs du tableau initial, mais cela s'avère fastidieux si l'on veut essayer avec de grands tableaux !

Écrire une fonction recevant pour paramètres deux entiers naturels n et v_max et renvoyant un tableau (liste Python) de n valeurs générées aléatoirement, comprises entre 0 et $v\_max - 1$.

On pourra utiliser `random.randrange` : `randrange(v_max)` renvoie justement une valeur entière "pseudo-aléatoire" de l'intervalle $[\![0, v\_max[\![$.

2. © *Tri par dénombrement* : on sait que les algorithmes de tri d'un tableau de taille n **opérant par comparaisons et échanges des données du tableau** ont une complexité dans le cas le plus défavorable au moins de l'ordre de grandeur de $n \ln n$. On propose ici un algorithme de tri d'entiers **qui n'effectue aucune comparaison** entre les données à trier.

On suppose ainsi que l'on veut trier par ordre croissant n entiers dont les valeurs sont toutes dans l'intervalle $[\![0, v\_max[\![$. Certaines valeurs peuvent figurer plusieurs fois dans la liste des données à trier ; ces valeurs figureront alors avec le même nombre d'occurrences après le tri !

Exemple : pour $v\_max = 10$ et $n = 9$, avec les données à trier 6,4,2,8,4,2,3,6,4, les données triées sont 2,2,3,4,4,4,6,6,8.

a) Expliciter un algorithme de tri, dont la complexité doit être un $O(v\_max + n)$, fondé sur le principe suivant : on compte, pour chaque entier compris entre 0 et v_max, son nombre d'occurrences parmi les entiers à trier, puis on en déduit le résultat du tri (*on pourra utiliser un tableau auxiliaire à préciser*).

On justifiera sommairement la complexité de l'algorithme proposé.

b) Il s'agit de programmer l'algorithme de la question précédente. Écrire en Python une fonction *tri_den* recevant pour paramètres un tableau t et un entier v_max (t ne contenant que des valeurs de l'intervalle $[\![0, v\_max[\![$) et renvoyant un tableau contenant les valeurs de t triées selon le principe précédent.

3. *Tri par sélection* : un peu différent du tri par insertion, voici un autre algorithme de tri quadratique. Le principe est le suivant : trier progressivement le tableau, par exemple de gauche à droite, en déterminant la plus petite des valeurs non encore triées et en la mettant à sa place.

Plus précisément, étant donné un tableau t de taille n, pour k allant de 0 à $n - 2$,

- déterminer un indice j tel que $t[j]$ soit la plus petite valeur de la tranche $t[k : n]$;
- échanger $t[k]$ et $t[j]$.

a) Programmer en Python une version itérative de cet algorithme. Par souci de modularité, pour déterminer j on écrira une fonction *indice_min* recevant comme paramètre un tableau t de longueur n et deux entiers g et d tels que $0 \leq g \leq d \leq n$ et renvoyant j tel que $t[j]$ soit la plus petite valeur de $t[g : d]$.

Justifier le programme et évaluer sa complexité.

b) Élaborer et programmer en Python une version récursive du tri par sélection.

Justifier et évaluer la complexité.

4. *Version procédurale du tri fusion*

Écrire un programme `tri_fusion(t)` **modifiant** le tableau t fourni en paramètre, de sorte qu'il soit trié, en utilisant le principe du tri fusion. On évitera toute recopie de tranche de tableau, hormis au moment de mettre en place dans t le résultat d'une fusion, stocké dans un tableau auxiliaire.

On définira :

- une fonction auxiliaire itérative `fusion(g,m,d)` plaçant dans la tranche $t[g : d]$ le résultat de la fusion des deux tranches $t[g : m]$ et $t[m : d]$ supposées triées ;
- une fonction auxiliaire récursive `tri(g,d)` triant la tranche $t[g : d]$.

5. *Comparaisons expérimentales* : une astuce classique pour dénombrer les opérations d'un certain type (par exemple les comparaisons entre éléments de tableau...) durant l'exécution d'un programme consiste à initialiser à 0 une variable globale, disons nb, puis à incrémenter nb à chaque opération devant être décomptée ! Pour ce faire, il suffit de remplacer dans le programme ladite opération par un appel à une fonction qui renverra le même résultat, mais en incrémentant au passage la variable globale nb...

Par exemple, on pourra remplacer `if t[i]<t[j]` par `if test(t[i]<t[j])` où la fonction `test` est définie par :

```
def test(valeur):
    global nb
    nb+=1
    return valeur
```

Mettre en œuvre cette idée pour tester les programmes de tri du cours et d'éventuelles variantes...

On peut aussi chronométrer le temps d'exécution d'un tri, en utilisant par exemple `time.perf_counter`. Un appel à `perf_counter()` renvoie un flottant correspondant à l'instant dudit appel. C'est la différence entre deux de ces flottants qui mesure le temps écoulé entre les deux appels : on prendra donc soin de mémoriser le résultat du premier appel !

6. *Tri de Shell et question des seuils*

Tout le monde avait remarqué que le tri par insertion est très efficace sur un tableau presque trié, mais qu'il est inefficace en moyenne car il ne déplace les valeurs que vers une case voisine (contrairement au tri rapide).

En 1959, Donald Shell a publié son idée "diabolique" pour améliorer le tri par insertion "classique" d'un tableau t :

- on fixe un pas p suffisamment grand (cf. ci-dessous) et l'on trie (par insertion !) les sous-tableaux formés d'éléments séparés de p places. Précisément, pour r allant de 0 à $p-1$, on applique le tri par insertion aux éléments $t\,[pq+r]$, pour les q de $\mathbb{N}$ tels que $pq+r < len(t)$.

- on réitère avec des pas p de plus en plus petits, en terminant avec $p=1$, cette dernière étape consistant à trier le tableau (presque trié à ce stade...) par un tri par insertion classique.

On profite ainsi de l'avantage du tri par insertion, tout en minimisant son inefficacité, puisque l'on commence par déplacer des éléments éloignés les uns des autres.

L'étude de la complexité de cet algorithme est très... complexe et recèle des problèmes encore ouverts à ce jour. Son efficacité est très sensible au choix des pas successifs.

Une liste des premières valeurs optimales desdits pas a été déterminée de façon empirique. Il s'agit de $[1, 4, 10, 23, 57, 132, 301, 701]$. Si l'on a besoin de la prolonger, pour de très grands tableaux, il est conseillé de partir de $p = 701$ et de réitérer la formule `p = int(2.3*p)`.

Moyennant quoi le tri de Shell s'avère d'une efficacité comparable à celle du tri rapide, voire parfois meilleure avec des langages (comme Python...) qui ne gèrent pas très bien la récursivité.

Cela étant, un bon choix est d'utiliser un programme de tri "hybride" en décidant d'un *seuil*, taille de tableau en-deçà de laquelle on applique le tri de Shell (un tri itératif efficace) au lieu de poursuivre les appels récursifs jusqu'à obtenir un tableau de taille 1 !

Les tableaux de grande taille sont ainsi soumis à un algorithme de tri récursif (tri fusion ou tri rapide), mais les appels récursifs s'arrêtent dès que la taille du tableau passe au-dessous du seuil.

a) Programmer le tri de Shell et comparer expérimentalement ses performances à celles des algorithmes du cours.

b) Programmer un algorithme hybride tel que celui décrit ci-dessus et évaluer ses performances avec divers choix du seuil.