Tris

Présentation

Le tri est une fonction importante de l'informatique, souvent comme étape préliminaire à d'autre algorithme. Il importe de connaître les algorithmes principaux et de les utiliser à bon escient. Les deux tris présentés ici, le tri partition et le tri fusion, sont les plus fréquemment utilisés. Deux applications sont proposées en fin de TD.

Préliminaires

Dans ce TD, les données sont stockées dans des listes. Afin de vérifier facilement la correction des fonctions écrites, les listes fournies en argument seront des permutations des n premiers entiers naturels, par exemple $[4,1,2,0,3]$ pour $n=5$.

Le but de cette première partie est de créer les outils pour construire rapidement de telle listes de taille arbitraire.

- Q1 : Ecrire une fonction **liste(n)** qui renvoie la liste ordonnée des n premiers entiers naturels : **liste(4)** renvoie $[0,1,2,3]$.

Pour les mélanger , on utilise un *générateur pseudo-aléatoire congruentiel linéaire* : on utilise la suite $x_{k+1} = (a\,x_k + c) \mod m$, avec $a = 69069$, $c = 1$, $x_0 = 42$ et $m = 2^{32}$. On admet que cette suite est de période m, et que donc x_n prend successivement toutes les valeurs entières de $[0,m[$, dans un ordre pseudo aléatoire.

On propose le code suivant :

```
x,a,m,c = [42],69069,2**32,1
def rand():
    x[0]=(a*x[0]+c) % m
    return x[0]/m
```

- Q2 : Vérifier que chaque appel à **rand()** fournit un nouveau nombre aléatoire. de quel type ? dans quel intervalle ?
- Q3 : En déduire une fonction **randint(n)** utilisant **rand()**, qui fournit un entier naturel aléatoire $< n$, de type **int**.

Mélangeur de Knuth

L'algorithme de Knuth permet de mélanger les éléments d'une liste L de taille n :

> Pour i allant de n-1 à 1(inclus) faire :
> j ← entier aléatoire dans [0,i] # i inclus
> échanger L[j] et L[i]

- Q4 : Ecrire une fonction **permutation(n)** qui génère une permutation aléatoire de $[0,n[$. **permutation(3)** renverra $[2,0,1]$ par exemple.

Tri partition

Le tri partition consiste à séparer la liste en deux listes, les petits et les grands, par rapport à une valeur pivot (le dernier élément de la liste initiale ici), puis de trier récursivement les petits et les grands.

Une version fonctionnelle, i.e qui renvoie une version triée de la liste fournie en argument sans la modifier, est présentée ici :

```
    def tripartition(L)
        if len(L) < 2 : return L
        pivot = L[-1]
        petits = [x for x in L if x < pivot]
        pivots = [x for x in L if x == pivot]
        grands = [x for x in L if x > pivot]
        return tripartition(petits) + pivots + tripartition(grands)
```

- Q5 : Expliquer le rôle de chaque ligne.
- Q6 : Tester cette fonction pour des permutations de taille 10 puis 100.
- Q7 : Evaluer le temps d'exécution dans la console avec l'instruction **%timeit tripartition(L)** . En déduire la complexité expérimentale parmi $\{O(n), O(n \ln n), O(n^2)\}$, dans le pire et le meilleur cas.

On souhaite maintenant en écrire un version "en place", c'est à dire qui réordonne les éléments de la liste fournie en argument, et qui ne renvoie rien.

On adopte la structure suivante :

```
def tripartition2(L):
    tri(L,0,len(L))

def tri(L,i,j): # trie la liste L en place de l'indice i inclus à l'indice j exclus.
    if j-i >= 2  : # au moins 2 éléments
        ip = partition(L,i,j) # ip est l'indice du pivot
        tri(L,i,ip)
        tri(L,ip+1,j)

def partition(L,i,j):
    """
    partitionne en place L[i:j] suivant le schema suivant :

    invariant : petits(<=) dans L[i:J], grands dans L[J:K+1].

     ------------------------------------------------------------
     |  petits   |        grands     |     non testés    |pivot|
     ------------------------------------------------------------
     |i          |J                K|                   | j-1 |j

    - état final :
     ------------------------------------------------------------
     |     petits    |pivot|             grands                |
     ------------------------------------------------------------
     |i              | J-1 |J                                 K|j

    - entrée : une liste et 2 indices

    - sortie : J-1 (indice du pivot)

```

```
    """
    J=  # initialisation
    for K ...

```

- Q8 :
  - a) Donner l'état du tableau suivant après chacune des 10 étapes.

4	**7**	**1**	**9**	**0**	**6**	**2**	**3**	**8**	**5**

  - b) Ecrire la fonction **partition**, de complexité linéaire en respectant l'invariant proposé et la tester. On utilisera un boucle mettant à jour le tableau et la variable J.

- Q9 : Reprendre les questions Q6 et Q7 pour la fonction **tripartition2**. Comparer et conclure.

Tri fusion

Le code de la fonction est le suivant :

```
def trifusion(L):
    m=len(L)//2
    if m==0 : return L
    return fusion (trifusion(L[:m]),trifusion(L[m:]))
```

On coupe donc la liste en 2 , on trie récursivement chacune des deux parties, et on fusionne les deux parties en remettant les éléments dans l'ordre :

fusion([0,2,5],[1,3,4]) renvoie $[0,1,2,3,4,5]$.

- Q10 : Ecrire la fonction **fusion**, de complexité linéaire.
- Q11 : Reprendre les questions Q6 et Q7 pour la fonction **trifusion**. Comparer et conclure.

Application 1 : médiane

On définit la médiane d'un liste S comme l'élément d'indice $\lfloor n/2 \rfloor$ dans la liste triée des élément de S, de taille n.

- Q12 : En utilisant un tri, Ecrire une fonction **mediane(L)** qui renvoie l'élément médian en $O(n \ln n)$ dans le pire cas.
- Q13 : Déterminer l'élément médian de **permutation(101)**
- Q14 : Ecrire une fonction **médiane2 (L)** en utilisant **partition** et de complexité linéaire dans le meilleur des cas, en s'inspirant de la recherche dichotomique.

Application 2 : Loto

- Q15 : Ecrire une fonction **loto()** qui génère une liste représentant un tirage de 6 entiers différents aléatoires dans $\{1,2,\ldots,48,49\}$. on pourra s'inspirer de la fonction **permutation**.

- Q16 : En se servant des fonctions précédemment écrites, écrire une fonction **bon_numeros(T1,T2)**, qui compte le nombre de boules communes aux deux tirages T_1 et T_2 (triés) . cette fonction devra être de complexité linéaire.
- Q17 : Calculer expérimentalement la probabilité de trouver au moins 3 bons numéros, a partir d'une simulation de $100\,000$ tirages.
- Q18 : Simulez des tirages jusqu'à trouver les 6 bons numéros d'un tirage de référence. Conclure.