Tris

Etant donnée une liste $[a_0, a_1, \cdots, a_{n-1}]$ de nombres (réels ou flottants) ayant n éléments, on cherche à la trier par ordre croissant (on peut aussi demander un ordre décroissant quitte à changer tous les signes $\leqslant$ et $\geqslant$). On peut formuler ce problème de deux façons :

1. ou bien renvoyer un nouvelle liste, différente de la liste initiale, comprenant les memes éléments que la liste initiale mais triée ;
2. ou bien déplacer les éléments déjà présents et modifier la liste initiale (*tri en place*).

Dans la première partie du cours, on se concentre sur des tris qui **créent de nouvelles listes**. On les éxécutera de façon détaillée en prenant comme exemple la liste `l=[4,3,2,7,5,1,6,8]`.

On présentera donc l'évolution de la liste l_t, qui avant le tri est vide, à la fin du tri est $[1,2,3,4,5,6,7,8]$.

1 Méthodes présentes dans les listes - coût temporel du tri

1.1 Trier une liste Python en une commande

Tri en place (on trie la liste en la modifiant) : `l.sort`

Tri avec création d'une nouvelle liste (on conserve l'ancienne liste) : `l2=sorted(l)`

Concours correspondant : Centrale, éventuellement les autres. On risque de vous demander de reprogrammer ces fonctions aux Mines ou à l'X, et de les utiliser à Centrale.

1.2 Evaluation du coût temporel du tri

A savoir :

1. On note $n = |l|$ la longueur de la liste ou du tableau à trier.
2. L'opération de référence de base dans l'évaluation de la complexité des tris est le nombre de **comparaisons**. On suppose que l'opération consistant à comparer deux éléments du tableau au moyen de la relation $\leqslant$ est de coût constant. On évaluera la complexité des différents algorithmes en fonction de ce nombre de comparaisons.
3. **Théorème** : Aucun algorithme de tri ne peut faire mieux que $O(n \ln n)$ opérations de comparaison, en moyenne et dans le pire des cas.
4. L'algorithme de tri utilisé par Python a été créé en 2002. Il a une complexité en $O(n \ln n)$ dans le pire des cas. Nous n'allons pas l'étudier dans ce cours : pour plus de détails, regardez par exemple l'article Wikipedia **Timsort**.

2 Tris élémentaires (PCSI) : $O(n^2)$ dans le pire des cas

2.1 Tri par insertion

Le tri par insertion consiste à recopier progressivement les éléments de l dans une nouvelle liste, en plaçant le nouvel élément à sa place à chaque fois.

```
1
2  def inserer(liste_triee,x):
3      n=len(liste_triee)
4      i=0
5      while i<n and x>liste_triee[i]:
6          i+=1
7      liste_triee.insert(i,x)
8      return liste_triee
9
10 def tri_insertion(l):
11     liste_triee=[]
12     for x in l:
13         inserer(liste_triee,x)
14     return liste_triee
```

Exemple

La liste l_t est successivement : `[4]`, `[3,4]`, `[2,3,4]`, `[2,3,4,7]`, `[2,3,4,5,7]`, `[1,2,3,4,5,7]`, `[1,2,3,4,5,6,7]` puis enfin `[1,2,3,4,5,6,7,8]`.

Analyse des couts en nombre de comparaisons :

la fonction inserer a un cout en $O(lgr(liste\_triee))$ dans le pire des cas.

Dans la fonction tri insertion, la fonction inserer est appelée $O(|l|)$ fois, et liste_triee augmente d'un élément à chaque fois : cela donne $O(|l|) \times O(|l|) = O(|l|^2)$ comparaisons dans le pire des cas.

2.2 Tri par sélection

Dans le tri par sélection, on chercher successivement le plus petit élément de la liste, plus le 2eme plus petit, le 3eme etc.... On insère les éléments de l trouvés dans une nouvelle liste, en queue de celle-ci.

Exemple

On a l_t qui est successivement `[1]`, `[1,2]`, `[1,2,3]`, ..., `[1,2,3,4,5,6,7,8]`.

Analyse des couts en nombre de comparaisons :

Trouver le minimum coute $O(|l|)$ comparaisons, la liste perd un élément à chaque fois ce qui donne : $O(|l|) + O(|l|-1) + \cdots + O(3) + O(2) + O(1) = \mathrm{O}(|l|^2)$ comparaisons.

3 Tri rapide – quicksort

3.1 Principe

Le principe est :

1. de choisir un élément de la liste a (par exemple le premier)
2. de séparer la liste en deux : l^- et l^+, contenant respectivement les éléments $\leqslant a$ et $> a$
3. de trier récursivement l^- et l^+
4. de concaténer : $l^- + [a] + l^+$

Une telle méthode est appelée **diviser pour régner**.

3.2 Code

```
1
2
3  def separe_pivot(l):
4      lmoins=[]
5      lplus=[]
6      if len(l)==0:
7          return [],[]
8      a=l[0]
9      for i in range(1,len(l)):
10         x=l[i]
11         if x<=a :
12             lmoins.append(x)
13         else:
14             lplus.append(x)
15     return lmoins, lplus
16
17
18
19 def tri_rapide(l):
20     if len(l)==0:
21         return l
22     lmoins, lplus=separe_pivot(l)
23     return tri_rapide(lmoins)+[l[0]]+tri_rapide(lplus)
```

3.3 Exemple

On part de $l =$ `[4,3,2,7,5,1,6,8]`. On prend 4 comme pivot, on sépare en deux listes :`[3,2,1]` et `[7,5,6,8]`.

- Tri de la liste `[3,2,1]` : on prend 3 comme pivot, on sépare en deux listes `[2,1]` et `[]`
  - Tri de la liste `[2,1]` : `[1,2]`
  - Tri de la liste `[]` : `[]`

  On trie donc la liste `[3,2,1]` en la liste `[1,2,3]`
- Tri de la liste `[7,5,6,8]` : on prend 7 comme pivot, on sépare en deux listes `[5,6]` et `[8]`
  - Tri de la liste `[5,6]` : `[5,6]`
  - Tri de la liste `[8]` : `[8]`

  On trie donc la liste `[7,5,6,8]` en la liste `[5,6,7,8]`

On trie donc la liste `[4,3,2,7,5,1,6,8]` en la liste `[1,2,3,4,5,6,7,8]`.

3.4 Complexité

Dans le pire des cas, la liste est déjà triée : on effectue a alors à chaque fois $l^- = []$, $l^+ =$ toute la liste . Le cout est alors en $O(n^2)$. De façon plus générale si la liste est déjà « presque triée » l'algorithme pose problème.

Un très bon cas est celui où a est à chaque fois un éléments médian dans la liste.

On a alors $T_n = \mathrm{O}(n) + T_{\frac{n}{2}} + T_{\frac{n}{2}} = \mathrm{O}(n) + 2T_{\frac{n}{2}}$. Ainsi $T_n = \mathrm{O}(n \log n)$.

En moyenne (ie si on munit l'ensemble des n-listes ayant les éléments $a_1, \cdots a_n$ d'une loi uniforme) on sait montrer que l'on a un coût en $O(n \ln n)$. Cf exo de fin de chapitre.

En pratique il arrive fréquemment, pour se prémunir du pire des cas, que l'on **mélange aléatoirement** la liste avant le la trier, rajoutant un cout en $O(n)$.

4 Tri fusion – mergesort

4.1 Principe

Le principe est :

1. de séparer la liste en deux : l_1 et l_2, contenant respectivement chacune la moitié des éléments de l (environ)
2. de trier récursivement l_1 et l_2 (lorsque l_1 et l_2 ont 0, 1 ou 2 éléments, on trie « à la main »);
3. de réaliser une *fusion croissante* des deux listes l_1 et l_2. Par exemple, la fusion croissante de $[1,2,5,7]$ et $[3,4,5,6]$ est $[1,2,3,4,5,5,6,7]$

Une telle méthode est appelée **diviser pour régner**.

4.2 Code

```
def fusion_listes_triees(l1, l2):
    l=[]
    n=len(l1)
    m=len(l2)
    i=0
    j=0
    liste_1_est_epuisee=(n==0)
    liste_2_est_epuisee=(m==0)
    while not(liste_1_est_epuisee) and not(liste_2_est_epuisee):
        if l1[i]<l2[j]:
            l.append(l1[i])
            i+=1
            liste_1_est_epuisee=(i>=n)
        else:
            l.append(l2[j])
            j+=1
            liste_2_est_epuisee=(j>=m)
    return l+l1[i:]+l2[j:]


def tri_fusion(l):
    n=len(l)
    if n <2:
        return l
    if n==2:
        if l[0]>l[1]:
            return [l[1], l[0]]
        else:
            return l
    else:
        m=n//2
        return fusion_listes_triees(tri_fusion(l[0:m]),tri_fusion(l[m:]))
```

4.3 Exemple

Tri de $l = [4,3,2,7,5,1,6,8]$.

1. On coupe l en deux listes : `[4,3,2,7]` et `[5,1,6,8]`
2. On trie les deux listes :
   (a) Tri de `[4,3,2,7]`
      i. On coupe en deux listes : `[4,3]` et `[2,7]`
      ii. On trie les deux listes : on trouve `[3,4]` et `[2,7]`
      iii. On fusionne les deux listes de façon croissante : `[2,3,4,7]`
   (b) Tri de `[5,1,6,8]`
      i. On coupe en deux listes : `[5,1]` et `[6,8]`
      ii. On trie les deux listes : on trouve `[1,5]` et `[6,8]`
      iii. On fusionne les deux listes de façon croissante : `[1,5,6,8]`
3. On fusionne les deux listes de façon croissante : on trouve `[1,2,3,4,5,6,7,8]`.

4.4 Complexité

Si on appelle T_n le cout de l'algorithme pour une liste à n éléments, on a alors

$$T_n = T_{\lfloor \frac{n}{2} \rfloor} + T_{\lceil \frac{n}{2} \rceil} + O(n) \approx 2T_{\frac{n}{2}} + O(n)$$

le $O(n)$ provenant du coût de la fusion croissante des deux listes triées.

On a alors $T_n = O(n\log(n))$.

5 Applications des tris : médiane d'une liste

5.1 L'astuce consistant à trier d'abord

Il arrive qu'un certain programme P portant sur une liste ou un tableau ait un cout en $O(n^2)$, mais un cout seulement en $O(n)$ si le tableau était trié. Il est alors intéressant de trier d'abord le tableau t, puis de lui appliquer P. En effet comparons :

1. Appliquer $P(t)$: $O(n^2$ opérations
2. Trier t en t', puis appliquer $P(t')$: $\mathrm{O}(n\ln(n)) + \mathrm{O}(n) = \mathrm{O}(n\ln n)$ opérations

5.2 Exemple du calcul de la médiane d'une liste

▲ DÉFINITION 1

Soit l une liste, un tableau ou un ensemble de nombres de taille n, et soit $m \in \mathbb{R}$. On dit que m est une **médiane** de l si : $\text{card}\left(\{k \mid l[k] \leqslant m\}\right) = \lceil \frac{n}{2} \rceil$ et $\text{card}\left(\{k \mid l[k] > m\}\right) = \lfloor \frac{n}{2} \rfloor$

Une médiane m est donc une valeur telle qu'il y a (en gros) autant d'éléments inférieurs (ou égaux) à m que d'éléments supérieurs (strictement) à m.

Le calcul le plus naif de la médiane d'une consiste à rechercher le minimum de la liste, à le supprimer, et à répéter l'opération $\lceil \frac{n}{2} \rceil$ fois.

On a donc un cout égal à $\mathrm{O}(n) + \mathrm{O}(n-1) + \mathrm{O}(n-2) + \cdots + \mathrm{O}(n - \lceil \frac{n}{2} \rceil)$, qui est majoré par $n\mathrm{O}(n) = \mathrm{O}(n^2)$ et minoré par $\mathrm{O}(n - \lceil \frac{n}{2} \rceil) \times \lceil \frac{n}{2} \rceil = \mathrm{O}(\frac{(n-1)^2}{4}) = \mathrm{O}(n^2)$

On peut remarquer cependant, que dans le cas où la liste l est triée, la médiane est l'élément au milieu de la liste. Ainsi si on trie l ($\mathrm{O}(n\ln(n))$ opérations de comparaison) puis que l'on récupère le milieu (cout constant) on obtient la médiane de la liste pour un cout $\mathrm{O}(n\ln n)$.

```
def mediane(liste):
    liste2=liste.copy()
    liste2.sort()
    indice_maximal=len(liste2)-1
    indice_milieu=indice_maximal//2
    return liste2[indice_milieu]
```

Exercice 1 Ecrire une fonction qui donne le k eme plus petit nombre d'une liste l ; l et k seront des arguments de la fonction. Attention : on a $k \geqslant 1$ mais l est indicée à partir de 0

Remarque

Le calcul du k ieme élément généralise le calcul de la médiane. Cependant on peut faire bien mieux que $\mathrm{O}(n\ln n)$ opérations, cf algorithme Quickselect exos de fin de chapitre.

6 Généralisation aux ensembles ordonnés et pré-ordonnés

6.1 Rappel de définitions

▲ DÉFINITION 2

Une relation binaire R sur E est une relation d'ordre sur E si elle est :
réflexive : $\forall x \in E\ xRx$,
antisymétrique : $\forall (x,y) \in E^2\ (xRy \text{ et } yRx) \Longrightarrow x = y$,
transitive : $\forall (x,y,z) \in E^3\ (xRy \text{ et } yRz) \Longrightarrow xRz$.
Le couple (E,R) est alors appelé **ensemble ordonné**.
La relation R est **totale** si on a $\forall (x,y) \in E^2\ (xRy \text{ ou } yRx)$.
Lorsque R est une relation d'ordre totale, on dit que (E,R) est un ensemble **totalement ordonné.**

1. $(\mathbb{R}, \leqslant)$ est bien un ensemble totalement ordonné
2. (mots en français, ordre du dictionnaire) est bien un ensemble totalement ordonné
3. L'ordre des mots du dictionnaire est appelé l'ordre **lexicographique**. On peut le préciser :
   - $(\mathbb{Z}^2, \preceq)$ où $(a,b) \preceq (c,d)$ si, et seulement si, $a < c$ ou $(a = c$ et $b \leqslant d)$;
   - $(\mathbb{R}^3, \preceq)$ où $(a,b,c) \preceq (u,v,w)$ si, et seulement si, $(a < u)$ ou $(a = u$ et $b < v)$ ou $(a = u, b = v$ et $c < w)$ ou $((a,b,c) = (u,v,w))$;
   - $(\mathbb{R}^n, \preceq)$ où $(a_1, \cdots, a_n) \preceq (b_1, \cdots, b_n)$ si, et seulement si, :
     $\left(\exists j \in [\![1, n-1]\!]\ a_1 = b_1, a_2 = b_2, \cdots, a_j = b_j \text{ et } a_{j+1} < b_{j+1}\right)$ ou $(a_1, \cdots, a_n) = (b_1, \cdots, b_n)$.
4. $(\mathcal{P}(E), \subset)$ est une relation d'ordre non totale, l'ordre est dit partiel.
5. La relation $\preceq$ définie sur $\mathbb{R}^2$ par $(a,b) \preceq (c,d)$ si, et seulement si, $a \leqslant c$ n'est pas une relation d'ordre. Cependant elle est réflexive, ~~antisymétrique~~, transitive et totale. On dit que c'est une relation de **préordre** total.

6.2 Tris

Tout ce que nous avons décrit dans les parties qui précèdent se généralise sans difficulté si les éléments $a_0, \cdots a_n$ sont des éléments d'un ensemble E muni d'un ordre total.

On peut même se passer de relation d'ordre : une relation de préordre total suffit. Cette remarque est pratique : dans nombre de cas on a des objets composés, du type (critere numérique, individu) et l'on souhaite uniquement trier les objets selon le premier critère.

Les ordres lexicographiques sont implémentés en Python entre types de bases et tuples. Testez par exemple `'ab'<'c'`, `(1,2)<=(1,1)`, `(1,2)<=(1,3,-1)` `True<False` ou `((1,1),'ab',3)<((1,1),'a',5)`.

Remarque — Avancé

Le dernier exemple montre que sous Python l'ordre lexicographique est en fait défini de façon récursive.

Entre listes et tableaux les comportements sont différents.

1. Sur les listes l'ordre lexicographique est implanté comme sur les tuples et de façon récursive : Tester `[1,'ab']<[1,'a']` par exemple.
2. Sur les tableaux, l'ordre lexicographique compare élément par élément du tableau, et retourne le tableau des résultats. Par exemple `np.array([1,2])<np.array([1,3])` renvoie un numpy.array de taille 2, consitués d'éléments booléens.

7 Tri avec clé

On peut trier la liste $[x0, x1, \ldots, x_n]$ en fonction d'une fonction f appelée **clé** : la liste triée le sera en fonction des valeurs $f(x_i)$.

Exemples

1. Liste de points (un point= tableau numpy 2D), on trie selon la distance à $(0,0)$

```
def tri_eucli0(points):
    def f(X):
        return sum(X**2)
    return sorted(points, key=f)
```

2. Liste de points (un point= tableau numpy 2D), on trie selon la distance à un certain point Y

```
def tri_eucli(points, Y):
    def f(X):
        return sum((X-Y)**2)
    return sorted(points, key=f)
```

3. Liste de mots (un mot= chaine de caractère), on trie par longueur décroissante

```
def tri_mots_taille(L):
   return sorted(L, key=len, reverse=True)
```

4. Si L contient des couples (p,a) où p est le prix d'un objet et a sa durabilité en années, affichons les dix couples (p,a) les plus rentables
   - en plusieurs lignes de code :

```
def f(x):
    return x[1]/x[0]
L2=sorted(L, key=f)
print(L2[:10])
```

   - en **une ligne de code.**

```
print(sorted(L, key=lambda x: x[1]/x[0])[:10])
```

8 Tri par comptage

Attention
Valable uniquement pour les listes d'entiers naturels.

```
def tri_comptage(l):
    L=[]
    d={}
    for x in l:
        if x in d:
            d[x]+=1
        else:
            d[x]=1

    for x in d:
        for _ in range(d[x]):
            L.append(x)
    return L
```

Complexité linéaire **mais** c'est dû aux informations que l'on a en amont, savoir que la liste est à valeurs entières. Aucun tri **général** de peut battre les $O(n \ln n)$ comparaisons dans le pire des cas.

9 Version « en place » de différents tris

Dans ce qui précède, on s'est concentré sur la crétaion d'une nouvelle liste triée. On peut néanmoins souhaiter effectuer des tris en place : on modifie la liste initiale qui devient triée. On n'a plus la facilité de créer un nouvel espace de stockage, mais on gagne en cout spatial, et on gagne en complexité temporelle car on ne recopie pas de listes.

9.1 Tri par insertion en place

On procède à des échanges au sein de la liste. Le principe est de maintenir le début de la liste triée, et d'insérer l'élément juste à droite au bon endroit.

```

def insere_en_place(l,j):
    for k in range(j):
        if l[j]<l[k]:
            l[j], l[k]=l[k], l[j]
    return l


def tri_insertion_en_place(l):
    for j in range(len(l)):
        insere_en_place(l,j)
    return l
```

9.2 Tri rapide en place

La fonction `partition(t,g,d)` modifie le tableau `t` ainsi : elle place le pivot `t[g]` dans le segment `t[g,d-1]`, en permutant les éléments de `t` de sorte qu'on ait `t[k]<pivot<=t[k']` pour tous $k < p \leqslant k'$. De plus `p`, la position finale du pivot, est renvoyée

La fonction `tri_rapide_en_place` trie alors récursivement `t` en appelant `partition` sur les bons indices.

```


def partition(t,g,d):
    p=g
    pivot=t[g]
    for k in range(g+1,d):
        if t[k]<pivot:
            p+=1
            t[p],t[k]=t[k],t[p]
    t[p],t[g]=t[g],t[p]
    print(t)
    return p


def tri_rapide_en_place(t):
    def tri(g,d):
        if g<d:
            p=partition(t,g,d)
            tri(g,p)
            tri(p+1,d)
    tri(0, len(t))
```

Preuve de correction de la fonction `partition` : Remarquons pour commencer que les invariants de boucle suivants sont maintenus :

1. $p \leqslant k$
2. le segment t[k+1 :d] est inchangé lors des $k - g$ premières itérations, car au pire on permute t[p] et t[k] ; mais on a $p \leqslant k$.
3. Pour tout $j \in [g+1, p]$, on a `t[j]`$\leqslant$`pivot`. En effet plaçons nous dans une itération de la boucle ; deux cas sont possibles. Ou bien p est constant, et l'invariant de boucle est hérité. Ou bien p augmente de 1 et devient $p' := p + 1$; dans l'algorithme on remplace immédiatement t[p'] par t[k], avec t[k]<pivot. Ainsi t[p']<pivot, et l'invariant est maintenu.

Ainsi d'après 2, chaque élément t[k] du tableau *initial* **est lu exactement une fois**, à savoir à la k eme itération de la boucle. En effet t[k] n'est pas lu avant la k ieme itération, est inchangé jusque là. Puis est lu lors de la k eme.

Or, p augmente de 1 si, et seulement si, on a `t[k]<pivot`. Ainsi à l'issue de l'algorithme p vaut $g + N$ avec N le nombre d'éléments du tableau initial `t[g+1:d]` qui sont strictement inférieurs au pivot.

Corollaire : **p contient l'indice où devra être placé le pivot dans le tableau final** `t[g:d]` à l'issue de l'algorithme.

Or d'après le troisième invariant, à l'issue de l'algorithme, on dispose de N éléments, à savoir t[g+1], ..., t[g+N]. Ils sont tous inférieurs au pivot. Comme N est aussi le nombre d'éléments inférieurs au pivot dans tout le tableau, **on les a tous trouvés**. Ainsi **tous** les éléments inférieurs au pivot ont un indice inférieur ou égal à $p = g + N$, et tous ceux supérieurs un indice à droite de p. On a donc bien partitionné le tableau comme voulu.

Enfin la dernière ligne du programme place le pivot en position.

10 Autres ressources

Visualisation en couleur des algorithmes de tri `https://imgur.com/gallery/voutF`

11 Exercices

Exercice 2 1. Ecrire une fonction `est_stmt_monotone` qui teste si une liste passée en argument est strictement monotone.
2. Ecrire une fonction `est_monotone` qui teste si une liste passée en argument est monotone.
3. Plus difficile : mêmes questions, mais vous n'avez pas le droit de lire deux fois la liste.

Exercice 3 On dispose d'une liste l contenant uniquement des 0 et des 1. Ecrire un code qui produit une liste l' égale aux éléments de l triés par ordre croissant. Vous viserez un coût en $O(n)$ opérations.

Exercice 4 Ecrire une fonction qui teste qu'une liste donnée en argument est croissante.
Ecrire une fonction qui trie une liste en place selon le procédé suivant : on en permute les termes au hasard tant que la liste n'est pas croissante.
Donner la complexité dans le pire des cas.

Exercice 5 Ecrire une fonction qui renvoie la longueur du plus long segment constant d'une liste.
Ecrire une fonction qui renvoie la longueur du plus long segment strictement croissant d'une liste.
Ecrire une fonction qui renvoie la longueur du plus long segment croissant d'une liste.
$\star$ Ecrire une fonction qui renvoie la longueur du plus long segment monotone d'une liste.

Exercice 6 On se donne deux listes l_1 et l_2, de longueurs respectives notées a et b, chacune formées de flottants deux à deux distincts. Ecrire une fonction qui compte le nombre d'éléments en commun dans les deux listes :

1. Avec une complexité $O(ab)$
2. Avec une complexité $O(a \ln a + b \ln b)$.
3. Avec une complexité $O((a+b) \ln \min(a,b))$

Exercice 7 On appelle l une liste de longueur n, qu'on va supposer formée des éléments $1, 2, 3, \ldots, n$ dans un certain ordre. On lui applique l'algotithme de tri rapide. On va compter le nombre de comparaisons effectuées : au pire ; et en moyenne.

1. Montrer que si la liste l est **déjà triée**, alors on effectue n^2 comparaisons.

2. Montrer que l'on n'effectue **jamais** plus de n^2 comparaisons.
3. Que proposez vous pour éviter ce **pire des cas** ?

1. On suppose donc que la liste l a été prétraitée de sorte qu'elle soit une permutation tirée au hasard avec équiprobabilité parmi les $n!$ permutations possibles.
2. Donner un univers Ω, P formalisant cela.
3. On note
$$l = [x_1, \dots, x_n]$$
On appelle $A_{i,j}$ l'événement « i et j sont comparés au cours de l'algotihme »On va montrer que
$$P(A_{i,j}) = \frac{2}{j-i+1}$$
   (a) Soit Z le premier élément (dans l'ordre de l'algorithme quicksort) **appartenant au segment** $[\![i,j]\!]$ qui serve de pivot. Montrer que $A_{i,j} = (Z=i) \cup (Z=j)$.
   (b) Donner la loi de Z (ne pas chercher à justifier trop précisément).
   (c) Conclure
4. Soit N le nombre de comparaisons. Exprimer N comme somme de variables de bernoulli faisant intervenir les événements précédents
5. Montrer que $\sum_{1\leqslant i<j\leqslant n} \frac{2}{j-i+1} \sim 2(n+1)\,H_n - 4n$ où $H_n = \sum_{k=1}^{n} \frac{1}{k}$ désigne le n-ème nombre harmonique.
6. En déduire que $E(N) \sim 2n \ln n$
7. Conclusion ?

Exercice 8 Ecrire un algorithme qui prend en entrée une liste d'entiers ou de flottants, et teste si elle ets injective, c'est-à-dire si ses éléments sont deux à deux distincts. On cherchera à minimiser la complexité. Trois solutions au moins...

Exercice 9 Algorithme Quickselect (programmation rapide pour déterminer le $k$ieme élément d'une liste)
En algorithmique, quickselect est un algorithme de sélection qui retourne le $k$ème plus petit élément dans une liste non ordonnée (et ayant au moins k éléments).
Fonctionnement de quick select :

1. Choisir un pivot a,
2. Classer les éléments de l selon leur place par rapport au pivot (comme dans Quick sort), au moyen du comparateur $<$.
3. **Si** l^- est de taille $k-1$, retourner a
4. **Sinon si** la liste l^- est de taille $\geqslant k$, appeler quickselect sur (l^-, k);
   sinon appeler quickselect sur $(l^+, k-(\text{len}(l^-)+1))$.

Questions

1. Expliquer pourquoi cet algorithme termine et fonctionne.
2. Implémenter cet algorithme en Python.

12 Solutions

Solution

Solution 5.2

```
def kieme_nombre(liste,k):
    liste2=sorted(liste)
    return liste2[k-1]
```

Solution

Solution 11

```
def est_stmt_monotone(l):
    i=0
    longueur_l=len(l)
    liste_est_stmt_monotone=(len(l)<=1) or not(l[0]==l[1])
    while i<=longueur_l-3 and liste_est_stmt_monotone:
        liste_est_stmt_monotone= ((l[i]-l[i+1])*(l[i+1]-l[i+2]) > 0)
        i+=1
    return liste_est_stmt_monotone


def est_monotone(l):
    longueur_l=len(l)
    if longueur_l<=2:
        return True
    i=0
    while i<=longueur_l-2 and (l[i]==l[i+1]):
        i+=1
    if i==longueur_l-1:
        return True

    est_croissante, est_decroissante= (l[i]<l[i+1], l[i]>l[i+1])
        #information sur le sens de variation lorsqu'on rencontre le premier couple de
            termes
        #consecutifs non constant dans la liste
    while i<=longueur_l-2 and (est_croissante or est_decroissante):
        est_croissante= est_croissante and (l[i]<= l[i+1])
        est_decroissante= est_decroissante and (l[i] >= l[i+1])
        i+=1## on doit memoriser le sens de variation, au cas ou l'on rencontrerait
            encore des plateaux
    return (est_croissante or est_decroissante)
```

Solution

Solution 11

```
def tri_liste_0_1(l):
    nombre_de_un= sum(l)
    return [0]*(len(l)-nombre_de_un)+[1]*nombre_de_un
```

Solution

Solution 11

```


def est_croissante(l):
    i=0
    while i+1<len(l) and l[i]<=l[i+1]:
        i+=1
    return i==len(l)-1


def tri_tres_lent(l):
    while not(est_croissante(l)):
        random.shuffle(l)
    return l
```

Pire du pire des cas : la liste l ets formée de n éléments deux à deux distincts, et **jamais** on ne tire au hasard la liste

ordonnée. Cela arrive avec une probabilité nulle, mais cet événement est bien non vide, cf cours de proba PC*.
Cout en moyenne : chaque appel à est_croissante est un $O(n)$, le temps d'attente pour tirer la liste ordonnée est $T \sim \mathcal{G}(\frac{1}{n!})$, on a donc en moyenne $n!$ itérations dans la boucle, chacune avec un cout en $O(n)$. Cela fait une complexité **surexponentielle** en $O(nn!) = O((n+1)!)$.

Solution

Solution 11 On a $A_{i,j} = (Z = i) \cup (Z = j)$ car si $i < Z < j$ les nombres i et j sont placés dans des sous listes différentes et ne seront plus jamais comparés (et n'ont pas été comparés avant car Z est le premier).
On a Z qui suit une loi **unfiorme** car la permutation tiére est uniforme. Aucun nombr de la litse $[\![i,j]\!]$ n'est donc favorisé.
On a :

$$\begin{aligned}
\mathbb{E}[N] &= \sum_{1\leqslant i<j\leqslant n} \mathbb{P}\left(A(i,j)\right) \\
&= \sum_{1\leqslant i<j\leqslant n} \frac{2}{j-i+1} \\
&= 2\sum_{1\leqslant i\leqslant n-1} \left(\sum_{i+1\leqslant j\leqslant n} \frac{1}{j-i+1} \right) \\
&= 2\sum_{1\leqslant i\leqslant n-1} \left(\sum_{2\leqslant \ell\leqslant n-i+1} \frac{1}{\ell} \right) \\
&= 2\sum_{2\leqslant m\leqslant n} \left(\sum_{2\leqslant \ell\leqslant m} \frac{1}{\ell} \right) \\
&= 2\sum_{2\leqslant \ell\leqslant n} \frac{1}{\ell} \left(\sum_{\ell\leqslant m\leqslant n} 1 \right) \\
&= 2\sum_{2\leqslant \ell\leqslant n} \frac{n+1-\ell}{\ell} \\
&= 2(n+1)\sum_{2\leqslant \ell\leqslant n} \frac{1}{\ell} - 2\sum_{2\leqslant \ell\leqslant n} \frac{\ell}{\ell} \\
&= 2\left((n+1)(H_n-1)-(n-1)\right) \\
&= 2(n+1)\,H_n - 4n \\
&\sim 2n\,\ln(n).
\end{aligned}$$

Solution

Solution 11 Un algorithme naif parcourt la liste avec deux indices i<j, et teste si l[i]==l[j].

```
def est_injective(l):
    n=len(l)
    for i in range(n):
        for j in range(i+1,n):
            if l[i]==l[j]:
                return False
    return True
```

Le cout est en $O(|l|^2)$.
Un autre algorithme consiste à trier la liste puis à tester si les éléments cote à cote sont bien différents :

```
def est_injective_tri(l):
    l2=sorted(l)
    for i in range(len(l2)-1):
        if l2[i]==l2[i+1]:
            return False
    return True
```

Le cout est en $O(n\ln n) + O(n) = O(n\ln n)$. Cet algorithme est donc préférable.
Enfin si on a le droit à un dictionnaire une solution efficace :

```
def est_injective_dico(l):
    d={}
```

```
    for x in l:
        if x in d:
            return False
        else:
            d[x]=True
    return True
```

coût linéaire en la longueur de la liste, sous réserve que le nombre de colisions est petit. Remarque : on ne tient pas compte du cout spatial.