TP 1 : Quelques tests sur les tris et mélanges parfaits

1 Quelques aspects des tris

Le but de cette section est d'observer quelques aspects temporels des tris. Téléchargez tout d'abord les tris du cours sur mon site web.

Exercice 1. *Génération aléatoire de listes.* La fonction `randint` du module `random` permet de générer aléatoirement des entiers : `randint(a,b)` renvoie un entier arbitraire de l'intervalle $[\![a,b]\!]$. À l'aide de cette fonction, écrire une fonction `genere_liste(n,N)` permettant de générer aléatoirement une liste de taille n, dont les entiers sont dans $[\![0,N]\!]$.

Exercice 2. *Inversions dans une liste.* Pour `L` une liste de taille n, on dit que deux indices $0 \leq i < j < n$ forment une *inversion* de la liste si `L[i]` $>$ `L[j]`. Écrire une fonction `compte_inversions(L)` permettant de dénombrer les inversions dans une liste.

Exercice 3. *Inversions et échanges dans le tri à bulles.* On va vérifier expérimentalement dans cet exercice que le nombre d'échanges effectués dans le tri à bulles correspond exactement au nombre d'inversions de la liste. Recopiez le tri à bulles sous le nom `echanges_bulles(L)`. Modifiez le code pour que :

— la fonction modifie une copie de la liste et non la liste initiale ;

— elle renvoie le nombre d'échanges effectués durant le tri.

Vérifiez-expérimentalement que `echanges_bulles(L)` et `compte_inversions(L)` coïncident, en générant des listes aléatoires.

> *Remarque : la complexité du tri par insertion est également liée au nombre d'inversions : en notant $I(L)$ le nombre d'inversions de `L`, on peut montrer que la complexité du tri sur une liste de taille n est $O(n+I(L))$. Ce n'est malheureusement pas le cas du tri à bulles qui peut toujours avoir une complexité en n^2 avec seulement $O(n)$ inversions.*

Exercice 4. *Temps d'exécution d'un tri.* On rappelle que la fonction `clock` du module `time` permet de mesurer les temps d'exécution de scripts. L'utilisation est la suivante :

```
from time import clock #importation de clock
[...]
t=clock() #on mesure le temps
[script] #on exécute un script
t2=clock()-t #on mesure le temps à nouveau et on soustrait t
#t2 contient alors le temps d'exécution du script !
```

À l'aide de `clock`, écrire une fonction `mesure_temps(f,L)` permettant de calculer le temps d'éxécution du tri `f` (qui peut être n'importe quelle fonction de tri, oui c'est possible !) sur la liste `L`. Ici on fera une copie de la liste avant de lancer le tri (ne pas prendre le temps de la copie en compte) pour ne pas modifier `L`.

Exercice 5. *Comparaison des tris.* On rappelle ici la syntaxe permettant de relier des points (x_i, y_i) par lignes brisées (et donc de tracer des courbes), en supposant que les x_i sont placés dans une liste `X` et les y_i dans une liste `Y`.

```
import matplotlib.pyplot as plt
plt.plot(X,Y,label="un nom de courbe")
#on peut tracer d'autres courbes ici avant d'afficher

plt.legend(loc="upper right") #localisation de la légende (lower/upper/left/right/center)
plt.show() #pour afficher
```

Pour des tailles de listes allant de 100 à 1000 par pas de 100 (par exemple) exécutez chacun des tris sur une liste aléatoire (prenez un nombre N assez grand, comme 10^6, pour ne pas biaiser le tri), mesurez le temps d'exécution, et tracez trois courbes (une pour chacun des tris). On peut éventuellement moyenner sur plusieurs listes. Comparez.

Exercice 6. *Comparaison des tris sur des listes presque triées.* À l'aide de `randint` et de la fonction `echange`, écrire une fonction `melange(L,k)` mélangeant la liste `L` en tirant aléatoirement deux indices de la liste et en permutant les éléments à ses indices, cette opération étant effectuée k fois. On dit (dans cet exercice) qu'une liste est k-presque triée si elle est obtenue à partir d'une liste triée et application de la fonction précédente. Reprendre l'étude précédente en remplaçant la fonction `genere_liste(n,N)` par celle-ci :

```
def genere_liste_2(n,N):
    L=genere_liste(n,N)
    L.sort() #on laisse Python trier
    melange(L,10) #des listes "10-presque triées"
    return L
```

Commentez ! *Explications : une liste k-presque triée a un petit nombre d'inversions.*

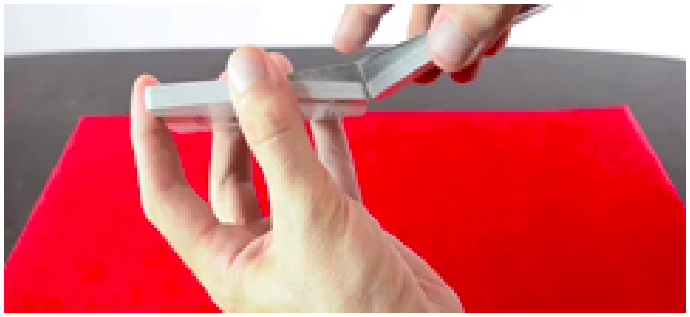
2 Mélanges de cartes parfaits

Dans cette section, on implémente un paquet de n cartes comme une liste de taille n, chaque carte étant associée à un numéro, par exemple de 0 à $n-1$. Un paquet trié peut être considéré comme la liste des entiers de 0 à $n-1$, qu'on peut obtenir par `list(range(n))`. On ne considérera ici que des paquets de tailles paires.

Exercice 7. *Mélange faro.* Le mélange faro est un mélange assez technique : c'est un mélange américain parfait (voir la photo). On doit donc couper le paquet en deux parties de même taille, et imbriquer les deux paquets en prenant une carte de chaque paquet. Il y a deux mélanges possibles :

— le *faro-out* : la première carte et la dernière carte du paquet sont inchangées. Appelons A la première moitié du paquet et B la deuxième, alors après le mélange faro, la première carte du paquet est la première carte de la partie A, suivi de la première carte de B, suivi de la deuxième de A, et ainsi de suite jusqu'à la dernière carte de B.

— le *faro-in* : avec les mêmes notations, une fois le mélange effectué, la première carte du paquet est la première de B, suivi de la première de A, et ainsi de suite jusqu'à la dernière de A.



1. Écrire une fonction `imbrique(A,B)` permettant d'imbriquer deux listes de même tailles, en commençant par le premier élément de la liste `A`.
2. En déduire deux fonctions `faro_in(L)` (resp. `faro_out(L)`) renvoyant la liste obtenue après mélange faro-in (resp. faro-out) de la liste `L`. La liste n'est pas modifiée, on renvoie une nouvelle liste.
3. Puisque le mélange faro (version in ou out) consiste à réaliser toujours la même permutation du paquet, il existe un entier n tel que réaliser n mélanges faro revient à laisser le paquet inchangé. Déterminer le plus petit $n > 0$ qui convient pour un mélange faro-in et un mélange faro-out, dans un paquet de 52 cartes (vous devez trouver un nombre plus petit pour un faro-out !)

Exercice 8. *Un petit travail sur les permutations.* On va expliquer ce qu'on a trouvé dans la section précédente via une étude des permutations de $[\![0, n-1]\!]$. Une permutation de $[\![0, n-1]\!]$ est simplement représentée par la liste des images : σ est représentée par la liste `L` si `L[i]` vaut $\sigma(i)$ pour tout i.

1. L'orbite d'un élément j par σ est l'ensemble $\{\sigma^k(i) \mid k \in \mathbb{N}\}$. Écrire une fonction `orbite(L,i)` calculant l'orbite de i sous l'action de la permutation associée à `L`. Chaque élément de l'orbite ne doit figurer qu'une fois. Plus précisément, on renverra le *cycle* contenant les images successives de i.

```
>>> orbite([1, 2, 0, 3, 4, 5], 0)
[0, 1, 2]
```

2. Une permutation se décompose en cycles à supports disjoints. À partir de la fonction précédente, écrire une fonction `decomposition(L)` renvoyant la décomposition en cycles à supports disjoints de la permutation associée à `L`.

```
>>> decomposition([1, 2, 0, 3, 4, 5])
[[0, 1, 2], [3], [4], [5]]
>>> decomposition([3, 2, 1, 5, 0, 4])
[[0, 3, 5, 4], [1, 2]]
```

3. L'ordre d'une permutation est le PPCM des longueurs des cycles. Écrire une fonction `PGCD(a,b)` calculant le PGCD de deux entiers a et b via l'algorithme d'Euclide. En déduire une fonction `PPCM(a,b)`, puis une fonction `ordre(L)` donnant l'ordre de la permutation associée à `L`.

```
>>> ordre([3, 2, 1, 6, 0, 4, 5])
10
```

4. Expliquer le résultat de la question 3 de l'exercice précédent, via une décomposition en cycle de la permutation associée au faro-in / faro-out.