Tri par insertion - Tri shell - Tri par paquets

1) Généralités

Trier un ensemble de données deux à deux comparables avec une relation d'ordre donnée, c'est les ranger dans l'ordre. Les données peuvent être simples, comme une liste de nombres entiers ou flottants, une liste de mots, mais aussi complexes, comme des fiches de produits que l'on peut classer par prix, nom, état des stocks, etc.

Les algorithmes de tris sont nombreux et il est important de choisir un algorithme bien adapté à la nature des données à trier. les questions qu'il faut se poser sont les suivantes:

- Les données à trier sont-elles aléatoires et avec quelle répartition, ou bien partiellement, voire presque totalement triées ?
- De quelle mémoire RAM dispose-t-on pour fairele tri, en plus de celle déjà utilisée pour charger les données à trier ?

2) Choix d'un l'algorithme efficace en fonction des réponses

La complexité d'un algorithme de tri est calculée en fonction du nombre n de données à trier. Bien sûr, si n est "petit", n'importe quel algorithme est efficace. Dans le pire des cas, les meilleurs algorithmes de tri ont une complexité $O(n \ln(n))$, mais certains algorithmes peuvent avoir une complexité en $O(n)$ dans des situations particulières.

3) Tri par insertion

Le tri par sélection est un algorithme de tri "en place". La liste triée est globalement invariante, il n'y a que des échanges de termes. On veut trier par ordre croissant une liste $s = [s_0, \ldots s_{n-1}]$ de nombres:

- On trie $[s_0, s_1]$ en insérant $x = s_1$ à sa place en le comparant à s_0. (Si $s_1 \geqslant s_0$, s_1 est déjà à sa place)
- Puis on trie $[s_0, s_1, s_2]$ en insérant $x = s_2$ à sa place en le comparant à s_1 puis éventuellement à s_0. (Si $x \geqslant s_1$, x est déjà à sa place)
- Puis on trie $[s_0, s_1, s_2, s_3]$ en insérant $x = s_3$ à sa place en le comparant à s_2 puis éventuellement à s_1 puis éventuellement à s_0. (Si $x \geqslant s_2$, x est déjà à sa place)

A chaque comparaison défavorable $s_j > s_i$ (avec $j < i$) on décale s_j d'une position vers la droite pour préparer l'insertion de x

- Et ainsi de suite jusqu'à trier $[s_0, \ldots s_{n-1}]$ en insérant $x = s_{n-1}$ à sa place.

Pour trier la liste s = [4,2,3,1], les valeurs successives différentes de s seront:

```
[4, 2, 3, 1]
[4, 4, 3, 1],[2, 4, 3, 1]
[2, 4, 4, 1],[2, 3, 4, 1]
[2, 3, 4, 4],[2, 3, 3, 4],[2, 2, 3, 4],[1, 2, 3, 4]
```

Q0) Ecrire les valeurs successives différentes de s = [3, 6, 5, 4, 1, 2] triée par l'algorithme du tri par insertion.

Q1) Ecrire une fonction tri_insertion(s) retournant la liste de nombres s triée par l'algorithme du tri par insertion. Tester.

Q2) Prouver que la complexité $C(n)$ (où n = len(s) est le nombre de termes de la liste s à trier) est $C(n) = O(n^2)$. Le coût est ici le nombre de comparaisons et d'affectations. Lorsque la liste est presque triée, la complexité est-elle meilleure ?

4) Tri shell

Le tri Shell est un tri en place qui est une remarquable amélioration du tri par insertion.

On exploite le fait que le tri par insertion est très efficace sur les listes presques triées. Pour trier une liste s, le tri shell va pré-trier partiellement s en effectuant des tris par insertion "modulo d" sur des sous listes de s, où d prendra une suite de valeurs bien choisies.

Plus précisément, pour trier par ordre croissant une liste $s = [s_0, \ldots s_{n-1}]$ de nombres avec le tri shell:

- On choisit la liste des écarts d : cette liste_ecarts doit être une liste strictement décroissante d'entiers qui se termine par 1.
- Pour $d \in$ liste_ecarts, on effectue un tri par insertion "modulo d" sur la liste s, c'est à dire:

pour i variant de d à $n-1$, insérer à sa place $x = s_i$ dans la liste $\left[\ldots, s_{i-2*d}, s_{i-d}, s_i \right]$

A chaque comparaison "défavorable", $s_{j-d} > s_j$ on décale s_{j-d} en s_j pour préparer l'insertion de x .

- Après le pré-tri "modulo d", la liste s vérifie $\forall\, i \in [\![d, n-1]\!]$, $s_{i-d} \leqslant s_i$. Comme la dernière valeur de la liste des écarts est $d = 1$, le dernier pré-tri est un tri par insertion classique qui nous assure que la liste sera effectivement triée.

Pour trier la liste s = [3, 6, 5, 4, 1, 2] avec liste_ecarts = [3, 1], les valeurs successives de s sont:
d = 3 : [3, 6, 5, 4, 6, 2], [3, 1, 5, 4, 6, 2], [3, 1, 5, 4, 6, 5], [3, 1, 2, 4, 6, 5]
d = 1 : [3, 3, 2, 4, 6, 5], [1, 3, 2, 4, 6, 5], [1, 3, 3, 4, 6, 5], [1, 2, 3, 4, 6, 5], [1, 2, 3, 4, 6, 6], [1, 2, 3, 4, 5, 6]

Le choix de la liste des écarts est un point crucial. Empiriquement, dans l'ordre inverse, le meilleur choix semble être [1, 4, 10, 23, 57, 132, 301, 701]. (voir Wikipédia) Si la liste à trier comporte plus de 701 termes, on peut calculer les écarts suivants avec la formule $d_{n+1} = \lfloor 2.25\, d_n \rfloor$.

Q3) Ecrire une fonction tri_shell(s) retournant la liste de nombres s triée par l'algorithme du tri shell. Tester sur diverses listes.

5) Comparaison expérimentale de l'efficacité des tris

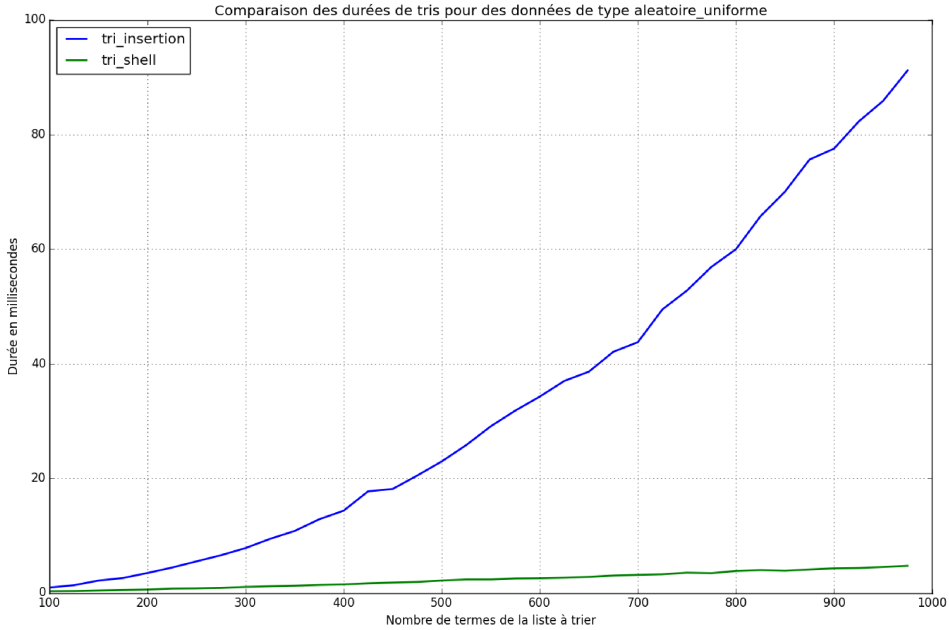
Q4) Le calcul théorique de la complexité du tri shell est trop difficile. Comparer exprimentalement, en traçant sur un même graphe les durées des tris sur des listes aléatoires de flottants, l'efficacité du tri par insertion et du tri shell.
On donne les deux fonctions:

```
import random, time
import numpy as np
import matplotlib.pyplot as plt
def duree_tri(tri, liste_a_trier):
    """ Durée du tri en millisecondes """
    t = time.perf_counter()   # time.clock() avec Python 3.2
    tri(liste_a_trier)
    return 1000 * (time.perf_counter() - t)  # time.clock() avec Python 3.2
```

```
def graphe_tris(plage_n):
    """ Graphe des durées de tris insertion et tri shell sur des listes de n
     données, avec n dans plage_n, ayant une répartition aléatoire uniforme"""

#    plt.rcParams['figure.figsize'] = 16, 10 # pour redimensionner le graphe
# DI et DS sont les listes des durées des tris par insertion ou shell
    DI, DS = [], []
    for n in plage_n:
# s est la liste des n données à trier, et on en fait une copie indépendante
        s = [random.random() for k in range(n)]
        s_copie = list(s)
# On calcule les durées des tris
        DI.append(duree_tri(tri_insertion, s))
        DS.append(duree_tri(tri_shell, s_copie))
# On génère les graphes des durées des divers tris
    plt.plot(plage_n, DI, linewidth = 2)
    plt.plot(plage_n, DS, linewidth = 2)
# Titres et légendes
    plt.legend(("Tri insertion", "Tri shell"), loc = 'upper left')
    plt.xlabel("Nombre de termes de la liste à trier")
    plt.ylabel("Durée en millisecondes")
    titre = "Comparaison des durées de tris pour des données aléatoires \
uniformes"
    plt.title(titre)
    plt.grid(True)
    plt.show()
```

Avec plage_n = list(range(10, 1000, 10)), voilà le genre de graphe très parlant que vous devriez obtenir:



6) Pour aller plus loin: le tri par paquets

Pour trier une liste aléatoire de flottants avec une répartition à peu près uniforme, il y a mieux que le tri shell. Le tri par paquets est un algorithme de tri adapté à cette situation. Ce n'est pas un tri en place, il consomme de la mémoire supplémentaire. Son principe est le suivant:

Pour trier par ordre croissant une liste $s = [s_0, \ ... s_{n-1}]$ de n nombres:

- Un premier parcours de la liste permet de déterminer le maximum m et le minimum M de la liste s
- On crée une liste $T = [[], \ ..., []]$ de $n+1$ listes toutes vides au début. Pour $k \in [\![0, n]\!]$, la sous liste $T[k]$ devra contenir le bon "paquet" de termes de la liste s, c'est à dire les termes $x = s[i]$ vérifiant $m + k\left(\frac{M-m}{n}\right) \leqslant x < m + (k+1)\left(\frac{M-m}{n}\right)$.
- On parcourt la liste s et on ajoute chacun de ses termes $s[i]$ à la bonne sous liste $T[k]$ de la liste T. (Se rendre compte que le bon k est $k = \lfloor$qqch$\rfloor$).

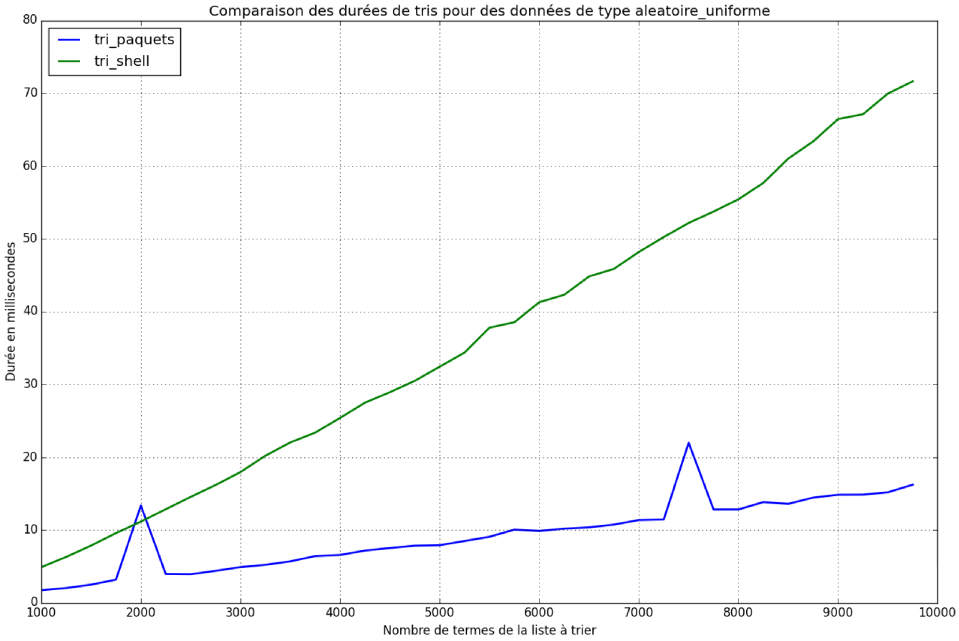
Si la répartition de s est uniforme, chaque paquet $T[k]$ contiendra en moyenne un terme de s. De plus, les paquets $T[k]$ sont rangés par ordre croissant, c'est à dire que $\forall \ k, k' \in [\![0, \ ..., n]\!], \ k < k' \ \Rightarrow \ \forall \ x \in T[k], \forall \ x' \in T[k'], \ x < x'$.

- On trie chacun des paquets $T[k]$ avec un tri par insertion, et le résultat est la concaténation $T[0] + T[1] + ... + T[n]$.

Par exemple, avec une liste de 20 termes:

s = [9.48, 7.45, 7.88, 7.07, 8.87, 6.55, 9.59, 8.71, 8.81, 9.8, 9.38, 3.72, 8.59, 5.53, 7.19, 3.65, 7.6, 2.75, 8.09, 9.1]
mini = 2.75, maxi = 9.8
paquets = [[2.75], [], [3.72, 3.65], [], [], [], [], [5.53], [], [], [6.55], [], [7.07, 7.19], [7.45, 7.6], [7.88], [8.09], [8.71, 8.59], [8.87, 8.81], [9.38, 9.1], [9.48, 9.59, 9.8], []]
s triée = [2.75, 3.65, 3.72, 5.53, 6.55, 7.07, 7.19, 7.45, 7.6, 7.88, 8.09, 8.59, 8.71, 8.81, 8.87, 9.1, 9.38, 9.48, 9.59, 9.8]

Q5) Ecrire une fonction tri_paquets(s) triant par ordre croissant une liste de nombres s et tester son efficacité. Vous devriez arriver à avoir quelque chose comme cela en le comparant au tri shell sur des listes aléatoires uniformes:



Expliquer les petits “pics” du tri par paquets.