Feuille de TP n° 1 : tris, méthode diviser pour régner, recherche exhaustive

Note : on effectuera les tris par **ordre croissant**.

Exercice 1. ***Génération aléatoire d'une liste d'entiers***
En utilisant le module Python random, écrire une fonction `liste_aleatoire(n,v_min, v_max)` qui renvoie une liste de longueur n d'entiers aléatoires compris entre v_min et v_max.

Exercice 2. ***Recherche dichotomique***
Implémenter la recherche dichotomique vue en cours, qui prend en entrée une liste `L` triée et un entier `x`, et renvoie la position de `x` dans la liste s'il y est, et `False` sinon.

Si le temps le permet, on pourra présenter une version récursive et une version itérative de cette fonction. Pour la version itérative comme pour la version récursive, on pourra utiliser des variables `imin` et `imax` qui nous donnent les bornes de l'intervalle dans lequel doit se trouver `x`, s'il appartient à la liste.

On rappelle que la recherche dichotomique divise par deux la taille de cet intervalle à chaque étape, en comparant `x` avec l'élément dont l'indice est au milieu de l'intervalle.

Exercice 3. ***Tri par insertion***
Dans cet exercice, on implémente en Python un tri par insertion sur une liste de longueur n où n est un entier positif non nul.

Méthode du tri par insertion par ordre croissant

Pour chaque position i de 1 à $n-1$, on sait que les éléments précédents la position i sont dans l'ordre croissant et on insère l'élément $T[i]$ parmi eux selon l'ordre croissant.

a) Écrire un algorithme de tri par insertion.

b) Écrire une fonction `tri_insertion(L)` qui implémente votre algorithme de tri par insertion sur une liste L. La liste L sera triée en place.

c) Tester votre fonction `tri_insertion` sur une liste aléatoire d'entiers.

Exercice 4. ***Tri shell***
Dans cet exercice, on implémente en Python un tri shell sur une liste de longueur n où n est un entier positif non nul.

Imaginé par Donald Shellen 1959, le tri de Shell est une optimisation du tri par insertion.

Un gros avantage du tri par insertion est que sa complexité est d'autant plus proche de la linéarité que le tableau en entrée est proche d'un tableau trié. L'idée est de faire un pré-traitement pour avoir un tableau quasiment trié avant d'effectuer le tri par insertion. Pour ce pré-traitement, on part de la constatation suivante : dans le tri par insertion, un élément se rapproche de sa place finale en progressant lentement, case par case. On peut accélérer son rapprochement en le déplaçant par des sauts h supérieur à 1.

Méthode du tri shell

On construit une suite de sauts décroissante jusqu'à la valeur 1. Puis pour chaque saut h, on trie selon l'algorithme du tri par insertion, séparément, les éléments d'indices congrus à $0[h]$, ceux congrus à $1[h]$, jusqu'à $h-1[h]$.

On constate que dans la pratique, il est plus simple de traiter selon le principe précédent tous les éléments de l'indice h jusqu'à la fin du tableau en même temps, plutôt que de le faire pour ceux congrus à $0[h]$, **puis** pour ceux congrus à $1[h]$, ... **puis** pour ceux congrus à $h-1[h]$. C'est ce qui est fait dans l'algorithme ci-dessous.

Algorithme du tri shell

On construit d'abord une liste `sauts` de sauts strictement décroissante et terminant par 1.

```
fonction tri_shell(liste L)
    pour h dans sauts
        pour i = h à n-1 faire
            temp = L[i]
            j = i
            tant que j >= h et L[j-h] > temp
                L[j] = L[j-h]
                j = j-h
            L[j] = temp
    retourner L
```

Connaître la suite de sauts conduisant à la meilleure complexité de ce tri est à l'heure actuelle un problème ouvert. Empiriquement, les premiers espacements optimaux sont : 1, 4, 10, 23, 57, 132, 301, 701. On prolonge cette suite par une croissance géométrique de raison $2,3$.

a) Comparer l'algorithme de tri shell avec celui du tri par insertion.

b) Ecrire une fonction `tri_shell(L)` qui implémente l'algorithme de tri shell.

c) Tester votre fonction `tri_shell` sur une liste aléatoire d'entiers.

Exercice 5. ***Méthode diviser pour régner : tri rapide***

Dans cet exercice, on implémente en Python un tri rapide sur une liste de longueur n où n est un entier positif non nul.

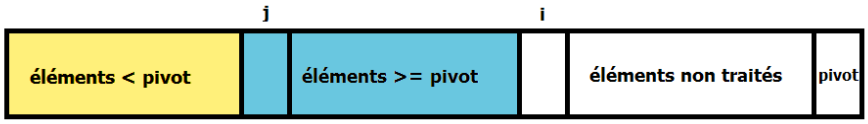
Méthode du tri rapide

- La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. C'est le **partitionnement**.
- Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à des sous-tableaux de longueur 1.

Invariant de boucle pour le partitionnement

On veut avoir l'invariant de boucle suivant, portant sur des indices $j \leqslant i$,

- $\forall k \in [indice\_debut, j[, \quad T[k] < pivot$
- $\forall k \in [j, i[, \quad T[k] \geqslant pivot$



Invariant de boucle du partitionnement

Méthode pour le partitionnement

- Echanger le pivot avec le dernier élément.
- Initialiser i et j à 0 ce qui assure que l'invariant est vrai au départ.
- Faire une boucle sur i variant de 0 à indice_fin-1.

- A chaque itération, on veut que l'invariant de boucle vrai pour i devienne vrai pour $i+1$.
  Pour obtenir ce résultat, il suffit de faire :
  `si T[i]<pivot alors échanger T[i] et T[j] puis incrémenter j de 1.`
- Une fois la boucle terminée, échanger le pivot en dernière position avec `T[j]`.

a) Ecrire un algorithme de partitionnement puis un algorithme de tri rapide.

b) Ecrire une fonction `tri_rapide(L)` qui implémente votre algorithme de tri rapide sur une liste L. La liste L sera triée en place.

c) Tester votre fonction `tri_rapide` sur une liste aléatoire d'entiers.

Exercice 6. ***Exercice optionnel : tri fusion***

Dans cet exercice, on implémente en Python un tri fusion sur une liste de longueur n.

Méthode du tri fusion pour l'ordre croissant

- Si le tableau n'a qu'un élément, il est déjà trié.
- Sinon, séparer le tableau en deux parties de longueurs égales à 1 près.
- Trier récursivement les deux parties avec l'algorithme du tri fusion.
- Fusionner les deux tableaux triés en un seul tableau trié.

Méthode de la fusion de deux tableaux triés `Tg` et `Td` dans un tableau `T`

- Pour `k` de 0 à n-1, on copie en `T[k]` le plus petit de `Tg[g]` et `Td[d]`.
- Quand on a épuisé `Tg` ou `Td`, on recopie systématiquement les éléments de l'autre tableau.

a) Ecrire un algorithme de tri fusion.

b) Ecrire une fonction `tri_fusion(L)` qui implémente votre algorithme de tri par fusion sur une liste L. **La liste L ne sera pas triée en place**.

c) Tester votre fonction `tri_fusion` sur une liste aléatoire d'entiers.

Exercice 7. ***Comparaison expérimentale de la complexité en temps***

a) Adapter le code suivant pour comparer la complexité des algorithmes de tri précédents. Ce code est disponible sur Moodle (fichier `courbes_performances.py`).

```
def courbes_performances(l_min,l_max,raison):
    import random as r
    import time
    import matplotlib.pyplot as plt

    # on cree une liste géométrique des tailles d'entrees que l'on veut tester
    ln=list()
    n=l_min
    while n<=l_max:
        ln.append(n)
        n=n*raison

    # on cree pour chaque fonction a tester, une liste des durees obtenues
    durees_selection=list()
    durees_insertion=list()
```

```
# on execute les tests pour chaque taille d'entrees
for n in ln:
    # on definit les valeurs min et max contenues dans les entrees
    v_min=0
    # pour limiter le nombre de repetition d'une meme valeur,
    # on prend prend v_max nettement plus grand que la taille de l'entree
    v_max=n*10
    L=liste_aleatoire(n,v_min, v_max)

    # la methode copy retourne une copie superficielle de la liste L
    L1=L.copy()
    t1 = time.perf_counter()
    tri_selection(L1)
    t2 = time.perf_counter()
    durees_selection.append((t2-t1))

    L2=L.copy()
    t1 = time.perf_counter()
    tri_insertion(L2)
    t2 = time.perf_counter()
    durees_insertion.append(t2-t1)

plt.plot(ln ,durees_selection,label="Tri Selection")
plt.plot(ln ,durees_insertion, label="Tri Insertion")
plt.legend()
plt.show()
plt.close()
```

b) Quelles réflexions vous inspirent ces courbes ?

Exercice 8. ***Méthode de recherche exhaustive : recherche de carrés magiques***
Un algorithme de recherche exhaustive appelée aussi recherche par force brute essaie toutes les solutions possibles pour le problème.

a) Ecrire un algorithme de type recherche exhaustive pour construire un carré magique normal d'ordre n, c'est-à-dire un carré magique nxn utilisant tous les nombres de 1 à n^2.

b) Calculer la complexité de votre algorithme.

c) Créer une liste C de longueur n dont les éléments sont des listes de longueur n. Attention à ce que les sous-listes possède des identifiants (`id`) distincts. Toutes les valeurs sont initialisées à 0.

d) Ecrire une fonction `somme_ligne(C,i)` qui retourne la somme des éléments de la ligne i du carré C et une fonction `somme_colonne(C,j)` qui fait la même chose avec la colonne j.

e) Ecrire une fonction `carre_magique(n)` qui affiche un carré magique normal d'ordre n par une méthode de recherche exhaustive.

f) Par des tests, expérimenter comment évolue le temps d'exécution en fonction de n.

g) Evaluer par le calcul la complexité de cet algorithme. Le résultat expérimental était-il prévisible ?