Algorithmes de tri (1/2)

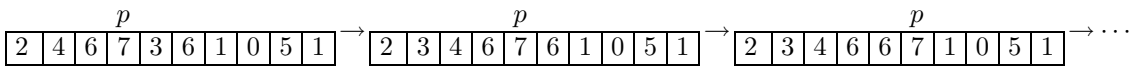
On a vu au cours d'un TP précédent le tri à bulle qui fonctionne sur le principe de balayages de la liste à trier, et d'échanges de couples de valeurs consécutives. On va aborder ici deux nouveaux algorithmes de tri, un élémentaire : le tri par insertion, et un plus élaboré : le tri fusion, lequel s'appuie sur le paradigme « diviser pour régner » et la récursivité ce qui conduit on va le voir à un algorithme beaucoup plus efficace.

Commencez par lancer spyder, et téléchargez à l'adresse `https://cahier-de-prepa.fr/pcsi-bdb/docs?rep=101` le fichier `TPtris2.py`, sauvegardez celui-ci dans votre répertoire personnel puis ouvrez-le dans l'éditeur de Spyder.

1 Tri par insertion

Le tri par insertion est l'algorithme qu'on va naturellement utiliser pour trier par exemple un paquet de copies (que ce soit selon la note ou le nom…) : on prend la première copie, puis la seconde qu'on place sur ou sous la première, puis la troisième qu'on insère parmi les deux premières et ainsi de suite…

Supposons une liste de n termes à trier, et que ses p premiers termes soient déjà bien ordonnés, alors on prend le $p+1$-ème terme, on le compare avec le p-ième. Si nécessaire on décale ce dernier d'une case à droite, et on regarde le terme précédent, et ainsi de suite jusqu'à trouver la place de ce qui était le $p+1$-ème terme :



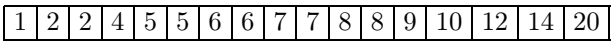
Voici en pseudo-code l'algorithme de tri par insertion (on suppose dans ce qui suit les éléments de T numérotés, comme en python, de 0 à $n-1$) :

2 Tri fusion

Le tri fusion s'appuie sur la remarque suivante : si deux listes sont triées (mettons que ce soit dans le sens croissant), alors il n'est pas trop difficile d'en déduire une nouvelle liste, également triée, et formée exactement des éléments combinés des deux listes précédentes.

Exemple, les deux listes triées :

conduisent à la liste :



Question 1. Compléter la fonction `fusion` suivante, admettant en arguments deux listes `L1` et `L2` (qu'on ne modifiera pas) supposées triées dans le sens croissant et renvoyant la liste `L`, également triée dans le sens croissant, formée des éléments de `L1` et `L2`.

```
def fusion(L1, L2):
    L = [] # la liste L sera complétée des éléments de L1 et L2, un par un.
    n, m = len(L1), len(L2)
    i, j = 0, 0 # i permettra de parcourir L1, et j L2
    for _ in range(n+m): # la liste fusionnée comportera n+m termes
        ...
    return L
```

Question 2. Evaluer, en fonction des longueurs respectives des listes `L1` et `L2`, le nombre d'affectations, comparaisons effectuées dans la fonction précédente (ce qui donne une idée de la complexité, en temps, de l'algorithme présenté)

Question 3. Grâce à la récursivité, on déduit alors un algorithme pour trier une liste : on coupe celle-ci en deux parties, aussi égales que possibles en taille, on trie la première partie, on trie la seconde, et on n'a plus qu'à fusionner celles-ci.

(Voir l'annexe pour d'éventuels rappels de syntaxe sur les listes).

Ecrire alors une fonction récursive d'entête `def triFusion(L):` qui admet en argument une liste de nombres `L` (ou d'objets python comparables comme pourraient l'être des chaînes de caractères, des tuples ou listes de nombres…) et qui renvoie une nouvelle liste formée des mêmes éléments que `L` mais triée dans le sens croissant (On ne modifiera pas la liste `L`)

Indication : on n'oubliera pas le test d'arrêt !

Question 4. Dans le script téléchargé figure déjà une implémentation du tri par insertion dans la fonction `triInsertion`. Celle-ci modifie *en place* `L` pour qu'elle soit triée, mais renvoie `None`. Pour comparer nos deux fonctions sur une même liste, on pourra donc invoquer en premier lieu la fonction `triFusion` sur une liste non triée, puis la fonction `triInsertion`.

A l'aide de la fonction `perf_counter` (voir plus loin les rappels de syntaxe python si nécessaire) écrire une fonction d'entête `def compareTris(n):` qui prend en argument un entier strictement positif `n`, génére une liste non triée de `n` termes (bien sûr, on fera appel à la fonction `listeAleatoire`) en réalise le tri avec les deux algorithmes de tri par fusion et par insertion et mesure le temps passé dans chaque, que l'on affichera.

On pourra tester des valeurs de `n` telles que $500, 1000, 2000, 4000, 8000$.

3 Tri rapide (quicksort)

Là où le tri fusion s'appuie sur un algorithme qui effectue la fusion de deux listes triées, le tri rapide s'appuie sur un algorithme qui partitionne une liste autour d'une valeur pivot, choisie parmi les valeurs de la liste à trier.

Partons par exemple de la liste `L` :

7	2	1	8	6	5	11	9	10	2	6

On va partir de la dernière valeur de `L` qui va servir de pivot : on va réorganiser les 11-1 premiers termes de `L` pour que figurent en tête tous les termes inférieurs (ou égaux) à celui-ci, puis que suivent tous ceux strictement supérieurs au pivot. Un seul parcours, de gauche à droite, de ces 11-1 termes suffit :

Le premier terme de la liste 7 est strictement supérieur au pivot 6 (il fera partie de la seconde partie de notre liste à la fin de notre travail).

Le second 2 est lui inférieur à notre pivot, donc on le déplace en tête (on échange pour ce faire les deux premiers termes) et notre liste devient :

2	7	1	8	6	5	11	9	10	2	6

Notez que les termes grisés ne bougeront plus !

le terme suivant 1 est lui aussi inférieur à notre pivot, donc il vient compléter notre tête de liste en échangeant deux valeurs :

2	1	7	8	6	5	11	9	10	2	6

le suivant 8 est supérieur à notre pivot, donc on le laisse en place, tandis que le terme suivant 6 est inférieur ou égal à notre pivot donc il rejoint la tête de liste :

2	1	6	8	7	5	11	9	10	2	6

on fait de même pour le terme suivant 5, on laisse en place les trois suivants et on déplace encore le terme 2 en tête de liste pour parvenir à :

2	1	6	5	2	8	11	9	10	7	6

Garder notre valeur pivot en queue de liste n'a plus guère de sens, et on la déplace à sa position définitive en échangeant encore deux valeurs dans notre liste :

2	1	6	5	2	6	11	9	10	7	8

On le voit alors, notre valeur pivot est désormais bien placée, et ce qu'il reste à faire, c'est de trier la tête de liste (jusqu'au pivot non compris) et la queue de liste (à partir du pivot non compris)

Question 5. Compléter la fonction suivante, qui prend en argument une liste `L`, deux entiers `deb<fin` et qui modifie en place `L` pour partitionner `L[deb:fin]` selon la démarche proposée ci-dessus. Comme ci-dessus, on prendra comme pivot la dernière valeur de `L[deb:fin]` (c'est-à-dire `L[fin-1]`). Notre fonction renverra l'indice de la position finale du pivot, laquelle nous permet de connaître quelles sont les parties obtenues par ce partitionnement.

```python
def partition(L, deb, fin):
    v = L[fin-1] # plus commode de nommer la valeur du pivot
    pos = deb # à tout moment la tête de liste sera formée de L[deb:pos]
    for i in range(deb, fin-1):
        if L[i] <= v:
            ...

    # on pense ici à placer le pivot à sa place définitive
    L[pos], L[fin-1] = ..., ...
    return pos
```

Question 6. Ecrire alors une fonction récursive, d'entête `def triRapideRec(L, deb, fin):` qui modifie en place `L` pour trier les termes d'indices `i` tels que `deb<=i<fin` à l'aide de l'algorithme décrit précédemment,

puis en déduire une fonction d'entête `def triRapide(L):` qui en modifiant en place `L` en réalise le tri.

Question 7. Reprendre la fonction `compareTris` de la question 4 pour comparer les temps d'exécution du tri fusion et du tri rapide.

Question 8. Qu'advient-il si on essaye de trier avec le tri rapide tel qu'implémenté une liste telle que `list(range(5000,0,-1))` ?

Comment expliquer ce phénomène, et comment pourrait-on le corriger ?

4 Pour les plus rapides : une étude de complexité du tri fusion

On l'a vu, fusionner deux listes d'une longueur totale de n termes prend un temps proportionnel à n (on parle d'une complexité *linéaire* selon n) qu'on estimera à Cn. Si on note $T(n)$ le temps (en toute circonstance) nécessaire à trier une liste de n termes, alors on obtient assez naturellement la relation suivante :

$$T(2n) = 2T(n) + Cn$$

On note alors pour tout n, $u_n = T(2^n)$.

Question 9. Montrer que la suite (u_n) satisfait à une relation de récurrence très simple que l'on précisera. En posant $u_0 (= T(1)) = 0$, calculer pour tout n, u_n. (On trouvera $u_n = \lambda n 2^n$ pour un λ que l'on précisera)

Question 10. Si $p = 2^n$, exprimer en fonction de p la valeur de $T(p)$.

Remarque 1. Le tri rapide est, en moyenne, d'une même complexité que le tri fusion, et en pratique un peu plus efficace… le plus souvent ! Il présente surtout l'avantage d'agir en place et de ne pas nécessiter plus de place en mémoire que la place qu'occupe notre liste à trier. Au contraire du tri fusion qui nécessite en pratique une place en mémoire qui est le double de celle qu'occupe notre liste à trier.

Néanmoins, dans certains cas, comme on a pu le voir, la complexité du tri rapide peut se révéler aussi mauvaise que celles des algorithmes élémentaires étudiés dans le TP précédent sur les algorithmes de tri (à savoir une complexité quadratique selon la taille n de la liste à trier.)

5 Rappels de syntaxe python

1. **Listes** :

   - Longueur d'une liste : étant donnée une liste `L`, `len(L)` renvoie le nombre de termes qui constituent `L`. On rappelle que le premier est d'indice 0, le dernier d'indice `len(L)-1`.

   - Etant donnée une liste `L`, `L.append(ob)` modifie la liste `L` pour lui adjoindre, à la fin, l'objet `ob`. Attention : cette instruction ne renvoit rien (ou plus précisément, renvoie `None`)

   - Tranches (slices en anglais) : d'une liste donnée `L`, alors avec la syntaxe `L[a:b]` on obtient une nouvelle liste formée des éléments de `L` d'indices `k` tels que `a<=k<b`. Quelles que soient les valeurs de `a` et de `b`, ceci ne déclenche pas d'erreur, mais peut conduire à une liste vide.

     Avec la syntaxe `L[a:]` on construit la liste formée des éléments de `L` d'indices `k` tels que `k>=a`

Avec la syntaxe `L[:b]` on construit la liste formée des éléments de `L` d'indices `k` tels que `k<b`.

On ne l'utilisera pas dans ce tp, mais on peut spécifier également un pas : avec `L[a:b:p]` on obtient la liste des éléments `L[a], L[a+p], L[a+2*p]`... pour tous les entiers `n=a+k*p` tels que `a<=n<b`.

Un cas particulier pratique : `L[::-1]` renvoie la liste formée des mêmes éléments que `L`, mais retournée.

Exemple : si `L=[1,4,2]` alors `L[::-1]` renvoie la liste `[2,4,1]`.

2. `perf_counter` : on rappelle un exemple d'utilisation :

```
from time import perf_counter
start = perf_counter()
unTraitementLong()
stop = perf_counter()
tempsEcoulé = stop - start # contient le temps écoulé dans unTraitementLong() en secondes
```