TP 10 : Algorithmes de tri. Comparaison

Dans ce TP, on revoit les tris vus en cours, et on les compare ! Récupérez l'annexe du TP sur le site web, elle contient notamment des tris déja écrits, et un squelette des fonctions à implémenter.

1 Des listes aléatoires

Question 1. *Listes aléatoires*. Cette question est essentielle pour la suite. La fonction `randrange`, du module `random` (déja importée dans l'annexe), est telle que `randrange(a,b)` fournit un entier aléatoire de $[\![a, b[\![$. Écrire une fonction `genere_liste(N,M)` générant une liste aléatoire de taille N dont les entiers sont dans $[\![0, M[\![$.

```
>>> genere_liste(10,100)
[22, 99, 9, 98, 42, 23, 65, 21, 51, 49]
```

2 Le tri à bulles

Ce tri n'est pas très efficace. Toutefois, il est facile à implémenter !

Question 2. Écrire une fonction `passage(L,p)` prenant en entrée une liste, et un entier p vérifiant $0 < p \leq$ `len(L)`, et parcourant `L[0:p]` de gauche à droite. Dès que deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. *Remarque :* votre boucle `for` ne prendra que $p - 1$ valeurs. Par exemple (la fonction ne renvoie rien) :

```
>>> L = [3, 1, 2, 0] ; passage(L,4) ; L
[1, 2, 0, 3]
```

Question 3. Modifier votre fonction pour qu'elle renvoie de plus un booléen indiquant à l'issue du passage si au moins un échange a été effectué (Votre fonction renverra donc `False` si et seulement si la portion `L[0:p]` était triée au départ).

```
>>> L = [3, 1, 2, 0] ; passage(L,4) ; L
True
[1, 2, 0, 3]
>>> L = [0, 1, 2, 3] ; passage(L,4) ; L
False
[0, 1, 2, 3]
```

Soit `L` une liste de taille n. On peut voir qu'à l'issue d'un passage sur toute la longueur de `L`, le plus grand élément est remonté tout à droite. Refaire un passage amène le deuxième plus grand élément à l'avant dernière position, toutefois il était inutile de le faire sur toute la liste, seulement sur les $n - 1$ premiers éléments. De même, refaire un troisième passage sur les $n - 2$ premiers éléments amène le troisième plus grand élément à sa position définitive, etc... Le tri à bulles consiste donc à faire $n - 1$ passages successifs sur `L`, sur des portions de taille $n, n - 1, \ldots, 2$. Toutefois, si l'un des passages renvoie `False`, on peut s'arrêter car on sait que la liste est triée.

Question 4. Écrire la fonction `tri_bulles(L)` prenant en entrée une liste, et effectuant au plus $n - 1$ passages sur des portions de taille décroissante, comme expliqué ci-dessus. Votre fonction ne renvoie rien. Dans le meilleur cas (si la liste est triée), votre fonction n'effectuera qu'un passage. Vérifier que votre fonction est bien correcte en lançant `test_tri(tri_bulles)`. Si vous obtenez une erreur, il faut reprendre vos fonctions !

3 Le tri fusion

Question 5. Le tri fusion est incomplet dans l'annexe, mais la fonction `fusion` est donnée. Le compléter, si possible sans le cours ! Rappel : l'algorithme renvoie une copie triée de la liste passée en paramètre.

— une liste de taille au plus 1 est triée ;

— sinon, il suffit de séparer la liste en deux parties de même taille à un élément près, trier récursivement les deux parties, et fusionner le résultat.

4 Comparaisons des tris

Dans cette section, nous allons comparer les tris que nous avons vus : tri par sélection, tri à bulles, tri par insertion, tri fusion et tri rapide. Pour cela, nous allons lancer des tris sur des listes aléatoires et regarder le temps d'exécution. Le principe général d'une telle mesure est le suivant, en utilisant la fonction `process_time` du module `time` (déja importée) :

```
t = process_time() #sauvegarde du temps
[... script dont on ve mesurer le temps ...]
t2 = process_time() - t #temps d'exécution du script
```

En effet, c'est équivalent à regarder l'heure, faire une tâche, et regarder l'heure à nouveau : la soustraction des deux temps donne le temps d'exécution de la tâche.

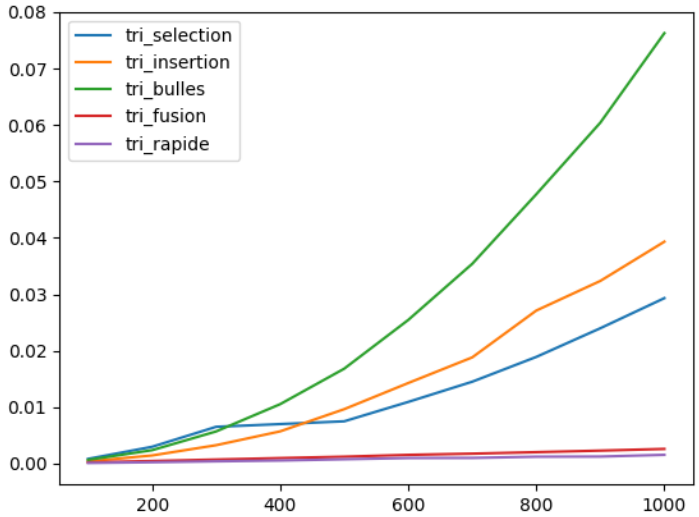
Question 6. En suivant le principe ci-dessus, écrire une fonction `temps(tri, N, M)` prenant en entrée une fonction de tri, deux entiers N et M, générant une liste aléatoire de taille N dont les entrées sont dans $[\![0, M[\![$ et renvoyant le temps d'exécution du tri sur cette liste. *Remarque :* même si c'est assez négligeable, prendre uniquement en compte le tri, pas la génération de la liste dans le temps d'exécution.

```
>>> temps(tri_selection,1000,10**6)
0.042807720000000105
>>> temps(tri_rapide,1000,10**6)
0.0036738430000000655
```

Question 7. Déduire de la fonction précédente une fonction `liste_temps(tri, tailles, M)`, prenant en entrée `tri` et `M` comme dans la fonction précédente, et une liste `tailles` de tailles de liste. Votre fonction renvoie sous forme de liste les temps d'exécution sur des listes de tailles prises dans `tailles`.

Rappel sur `matplotlib.pyplot`. Ce module est déja importé dans l'annexe. On peut tracer un graphe en y ajoutant une étiquette comme ceci : `plt.plot(X,Y,label="etiquette")` (où `X` et `Y` sont des listes d'abscisses et d'ordonnées). Réaliser `plt.legend()` rajoute la légende, et enfin `plt.show()` affiche le graphe. Dans le cas de plusieurs tracés sur un même graphique, on réalise plusieurs `plt.plot` avant `plt.legend()` et `plt.show()`.

Question 8. À l'aide de la question précédente, réaliser un graphique semblable à celui de la figure suivante : en abscisse se trouve la taille de la liste à trier (on a pris des tailles de 100 à 1000 par pas de 100, les éléments des listes à trier sont pris aléatoirement dans $[\![0, 10^6[\![$), en ordonnée le temps d'exécution de chacun des tris.



On observe donc une efficacité bien meilleure des tris fusion et rapide sur les tris quadratiques (c'est-à-dire en $O(n^2)$: sélection, insertion, bulles. Pour avoir une figure plus régulière, on pourrait moyenner plusieurs temps pour une même taille.

Question 9. Reprendre la question pour des listes de taille variant entre 1000 et 10000 par pas de 1000, seulement pour les tris efficaces (fusion, rapide). Vérifier qu'il y a un léger avantage au tri rapide dans la pratique (il s'effectue en place contrairement au tri fusion).

Question 10. Vérifier que le tri rapide est par contre peu efficace sur des listes où il y a beaucoup d'éléments égaux. (on pourra prendre les éléments aléatoirement dans $[\![0, 5[\![$, par exemple). *Rappel : dans le pire cas, la complexité du tri rapide est quadratique.. On a vu que ce pire cas se produisait notamment sur des listes triées ou presque triées. Les listes où il y a peu d'éléments différents se rapprochent des listes presque triées.*

5 Un autre algorithme de tri : le tri tableau

On veut implémenter un algorithme de tri de complexité $O(\ell\sqrt{\ell})$ avec ℓ la taille de la liste, ce qui en fait un tri plus efficace que les tris quadratiques, mais moins que le tri fusion [1] Pour trier une liste de taille ℓ, l'idée est d'utiliser une structure annexe à la liste, appelée un tableau (liste de listes) *croissant*, qu'on va définir dans la suite.

Rappels. Dans le problème, on va considérer des listes de listes. Si `T` est une liste comportant n listes toutes de tailles m, on peut visualiser `T` comme un tableau à deux dimensions, de taille $n \times m$. Les lignes sont indexées de 0 à $n-1$, et les colonnes sont indexées de 0 à $m-1$. Chaque ligne du tableau correspond à une liste de `T`, et la colonne d'indice j est constitué des éléments `T[i][j]` pour tout $0 \leq i < n$. La figure suivante montre une liste de taille 4×3 et sa représentation visuelle.

```
[[1,9,14], [5,12,17], [8,20,24], [16, 25, 30]]
```

1	9	14
5	12	17
8	20	24
16	25	30

FIGURE 1: Représentation d'un tableau par une liste de listes

Les éléments de ce tableau sont donc indexés par des couples d'entiers : pour $0 \leq i < n$ et $0 \leq j < m$, l'élément associé à (i, j) est accessible par `T[i][j]`. Par exemple l'élément `T[1][2]` est ici 17 : lignes et colonnes sont indexés de haut en bas et de gauche à droite. Le nombre de lignes est accessibles par `len(T)`, le nombre de colonnes par exemple par `len(T[0])`.

L'élément ∞. Dans ce problème, on va faire usage d'un infini. En Python, on peut créer un infini à l'aide de `float('inf')`. l'élément obtenu est supérieur à tout entier ou flottant, par exemple :

```
>>> Inf=float('inf')
>>> Inf>4444**4444
True
>>> Inf>2.0**1000
True
>>> Inf>Inf
False
>>> Inf>=Inf
True
```

Vous pourrez, dans tout le TP, stocker un infini dans une variable `Inf`, que vous considérerez comme globale.

Un tableau `T` de taille $n \times m$ est dit croissant si tout élément du tableau est inférieur ou égal aux éléments situés immédiatement à sa droite ou immédiatement en dessous, s'ils existent. Par exemple le tableau de la figure 1 est croissant. Le suivant l'est également :

1	2	5	8
1	5	12	17
8	17	∞	∞

Il est facile de voir que le plus petit élément d'un tableau croissant de taille $n \times m$ est `T[0][0]` et le plus grand est `T[n-1][m-1]`.

1. Ceci dit, on peut avec les mêmes idées obtenir un tri en $O(n \log n)$, mais il faut organiser les éléments autrement !

5.1 Préambule

Télécharger l'annexe sur le site web. Elle contient quelques exemples de tableaux.

Question 11. Écrire une fonction `init(n,m)` prenant en entrée deux entiers strictement positifs, et renvoyant un tableau croissant dont tous les éléments sont ∞.

```
>>> init(3,4) #à l'affichage, l'infini est inf.
[[inf, inf, inf, inf], [inf, inf, inf, inf], [inf, inf, inf, inf]]
```

Rappel : `[[Inf]*m]*n` ne marche pas : les éléments de la liste ainsi créée sont tous une référence vers une *même* liste en mémoire. Il faut créer n fois une liste de taille m.

5.2 Diminution ou augmentation d'un élément

Dans cette section, on va augmenter ou diminuer l'un des éléments d'un tableau croissant, et rétablir la croissance du tableau.

Diminution d'un élément. Si l'on diminue l'élément `T[i][j]`, on va considérer le plus grand élément entre `T[i][j]`, `T[i-1][j]` et `T[i][j-1]` (s'ils existent !). Si cet élément (disons x) n'est pas `T[i][j]`, il suffit d'échanger dans le tableau les éléments `x` et `T[i][j]` pour rétablir la condition de croissance en case (i, j). Le problème est alors reporté sur la case où se trouvait `x`. La figure suivante montre la succession de tableaux obtenus pour rétablir la croissance du tableau, l'élément en bas à droite ayant été diminué (de 30 à 7).

1	9	14
5	12	17
8	20	24
16	25	**7**

1	9	14
5	12	17
8	20	24
16	**7**	25

1	9	14
5	12	17
8	**7**	24
16	20	25

1	9	14
5	**7**	17
8	12	24
16	20	25

1	**7**	14
5	9	17
8	12	24
16	20	25

FIGURE 2: Rétablissement de la structure de tableau croissant

Question 12. Écrire une fonction `cmax(T,i,j)` renvoyant le couple d'indices $(a, b) \in \{(i, j), (i-1, j), (i, j-1)\}$, tel que $(a, b) \in [\![0, n-1]\!] \times [\![0, m-1]\!]$ (où n et m sont les dimensions du tableau) et `T[a][b]` maximal. Là encore, attention à ne pas essayer d'accéder à des cases non indicées par des couples de $[\![0, n-1]\!] \times [\![0, m-1]\!]$.

```
>>> cmax([[1, 9, 14], [5, 12, 17], [8, 20, 24], [16, 25, 7]], 3, 2)
(3, 1)
>>> cmax([[1, 9, 14], [5, 12, 17], [8, 20, 24], [16, 25, 7]], 0, 0)
(0, 0)
```

Question 13. En déduire une fonction `monter(T,i,j)` qui rétablit la croissance sur un tableau où l'élément indicé par (i, j) a éventuellement été modifié, à la baisse. On fera usage d'une boucle `while`. On impose une complexité $O(n + m)$.

```
>>> T=[[1, 9, 14], [5, 12, 17], [8, 20, 24], [16, 25, 7]]
>>> monter(T,3,2)
>>> T
[[1, 7, 14], [5, 9, 17], [8, 12, 24], [16, 20, 25]]
```

Indications : votre fonction n'a rien à renvoyer, puisqu'elle modifie le tableau. Ici, pour échanger les éléments indicés par (i, j) et (a, b), on écrira par exemple l'instruction `T[i][j], T[a][b] = T[a][b], T[i][j]`. Pour la condition d'arrêt, il suffit que `cmax(T,i,j)` soit égal à `(i,j)` pour que le coefficient soit bien placé.

Question 14. Justifier que votre fonction est bien correcte, et qu'elle a la complexité désirée.

Augmentation d'un élément.

Question 15. À l'inverse, on peut choisir d'augmenter l'élément `T[i][j]`. Écrire une fonction `descendre(T,i,j)` qui rétablit la croissance sur le tableau, toujours avec une complexité $O(n+m)$. On écrira au préalable une fonction `cmin`.

5.3 Insertion dans un tableau non plein, suppression d'un élément

On dit qu'un tableau croissant n'est pas plein s'il comporte des éléments ∞. Si c'est le cas, il y en a nécessairement un dans le coin en bas à droite (indexé par $(n-1, m-1)$ avec $n \times m$ les dimensions du tableau). Insérer un élément x dans le tableau revient à remplacer cet infini par x et à rétablir la croissance du tableau.

Question 16. Écrire une fonction `inserer(T,x)` qui insère `x` dans le tableau croissant `T`. Si celui-ci est plein, on affichera une erreur à l'écran. On impose toujours une complexité $O(n+m)$: il suffit de remplacer l'infini en bas à droite par `x` et d'appeler `monter`.

```
>>> T
[[1, 5, 29, inf], [3, inf, inf, inf], [inf, inf, inf, inf], [inf, inf, inf, inf]]
>>> inserer(T,7)
>>> T
[[1, 5, 7, 29], [3, inf, inf, inf], [inf, inf, inf, inf], [inf, inf, inf, inf]]
```

De même, un tableau croissant est dit vide s'il est rempli uniquement d'élément ∞. Pour un tableau non vide, supprimer le minimum revient à remplacer l'élément en haut à gauche par ∞, et rétablir la structure de croissance.

Question 17. Écrire de même une fonction `suppression_min(T)` qui, s'il est non vide, supprime du tableau `T` son plus petit élément **et le renvoie**. Si le tableau est vide on affichera une erreur à l'écran. On impose toujours une complexité $O(n+m)$.

```
>>> T=[[1, 2, 5, 8], [3, 5, 12, 17], [8, 17, Inf, Inf]]
>>> suppression_min(T) #le min est renvoyé.
1
>>> T #le tableau a été modifié.
[[2, 5, 5, 8], [3, 12, 17, inf], [8, 17, inf, inf]]
```

5.4 Le tri tableau

Le principe du tri tableau pour une liste est simple : on insère les éléments un à un dans un tableau croissant initialement vide (rempli d'infinis), puis on les retire un à un : ils sortent dans l'ordre croissant. Pour trier une liste de taille n et minimiser la complexité, on a intérêt à construire un tableau croissant dont les dimensions sont environ $\sqrt{n} \times \sqrt{n}$. Attention à avoir assez de place dans le tableau (prendre $\lfloor \sqrt{n} \rfloor + 1$ comme largeur et longueur). Pour rappel, on peut obtenir $\lfloor \sqrt{n} \rfloor$ facilement en Python :

```
>>> int(5**0.5)
2
>>> int(120**0.5)
10
```

Question 18. À l'aide des questions précédentes, écrire un algorithme `tri_tableau(L)` triant une liste à l'aide d'un tableau croissant, qu'on pourra prendre carré. Votre fonction ne renverra rien, mais modifiera la liste `L`.

```
>>> L
[37, 6, 1, 8, 42, 27, 20, 7, 31, 32]
>>> tri_tableau(L) #la fonction ne renvoie rien
>>> L
[1, 6, 7, 8, 20, 27, 31, 32, 37, 42]
```

Question 19. Justifier que la complexité du tri est $O(n\sqrt{n})$.