TP07 Algorithmes de tri

Pensez à écrire votre code *en premier lieu* sur feuille pour vous entraîner aux futures épreuves écrites d'informatique.

Dans ce TP, on va essayer de décrire plusieurs algorithmes de tris en partant des plus basiques (tri bulle, tri par insertion) pour monter en raffinement et descendre en temps de calcul, comme avec le tri-fusion ou le tri par baquets.

Partie I

Bubble sort (ou tri bulle)

On commence par concevoir une fonction `emporte_max_local(liste)` qui, pour une liste `liste` donnée en argument, modifie ladite liste de sorte à visiter toutes les positions de gauche à droite tout en emportant dans le déplacement le plus grand élément trouvé jusqu'à présent, un peu comme si vous remontiez une allée de bibliothèque et que vous n'avez le droit d'emporter qu'un seul livre sans encore savoir celui que vous allez préférer dans la série : quand vous en trouvez un digne d'intérêt, vous le prenez en main et continuez votre chemin, mais si vous en trouvez un mieux, vous remettez l'ancien à l'endroit où vous prenez le nouveau.

Sur un exemple, la fonction doit avoir les effets successifs suivants (elle ne renvoie rien mais modifie à chaque fois la liste donnée en argument qui change donc à chaque appel) :

```
>>> L = [0, 10, 7, 5, 11, 13, 6]
>>> L
[0, 10, 7, 5, 11, 13, 6]
>>> emporte_max_local(L)
>>> L # Le 10 s'est déplacé jusqu'à croiser le 11, puis c'est le 13
[0, 7, 5, 10, 11, 6, 13]
>>> emporte_max_local(L)
>>> L # Le 7 et le 11 ont l'occasion d'être emportés
[0, 5, 7, 10, 6, 11, 13]
>>> emporte_max_local(L)
>>> L # Puis le 10 peut sauter au-dessus du 6
[0, 5, 7, 6, 10, 11, 13]
>>> emporte_max_local(L)
>>> L # qui se fait alors rattraper par le 7
[0, 5, 6, 7, 10, 11, 13]
```

1. Implémenter la fonction `emporte_max_local(liste)` décrite précédemment. Elle doit modifier la liste en une seule passe (donc pas de boucle de boucles ici) avec une unique instruction conditionnelle autorisée en ayant bien sûr l'effet voulu sur les exemples.

Stop Gitlab

Allez sur GithubDesktop pour faire un commit. Choisissez (avec pertinence) le résumé. Pensez, si possible, à appuyer sur le bouton «Push origin» en haut à droite pour mettre à jour sur le web.

On voit que les appels successifs à cette fonctions finissent par trier la liste dans son ensemble. Il est d'ailleurs intéressant de réfléchir au « pourquoi » de la chose et on cherchera plus tard dans le cours comment « démontrer » ce fait de manière inattaquable.

2. Écrire une fonction `bubble_sort(liste)` qui modifie la liste en argument jusqu'à la trier en se contentant d'appeler itérativement la fonction `emporte_max_local` autant de fois que nécessaire (pensez à ce qui se passe si jamais la liste est triée en ordre décroissant pour connaître le nombre minimal d'appels) [1]

Stop Gitlab

Allez sur GithubDesktop pour faire un commit. Choisissez (avec pertinence) le résumé. Pensez, si possible, à appuyer sur le bouton «Push origin» en haut à droite pour mettre à jour sur le web.

Partie II

Insertion sort (ou tri par insertion)

Le tri par insertion est le tri du joueur de carte : vous ramassez vos cartes une par une de la main droite et vous la mettez au bon endroit dans votre main gauche (qui porte donc les cartes déjà triées). On va le découper en deux étapes : une première étape pour l'insertion proprement dite, et la seconde qui va appeler cette fonction d'insertion autant de fois qu'il y a de cartes à insérer.

3. Écrire une fonction `insertion(liste, i_depart)` qui prend la liste complète `liste` en argument ainsi que l'emplacement `i_depart` actuel de la carte qu'il faut insérer. Le but sera de la pousser vers la gauche jusqu'à trouver un élément plus petit qu'elle ou atteindre l'extrémité gauche de la liste. La fonction modifie la liste mais ne renvoie rien. Contrairement au tri bulle et sa fonction `emporte_max_local`, on se contente de bouger un unique élément jusqu'à trouver plus petit que lui.

1. Point bonus : mettez dans la variable `agent_mystere` (sans accent...) le nom de l'agent des services sociaux dans l'image précédente.

On donne les quelques exemples suivants pour le comportement attendu.

```
>>> L = [0, 10, 7, 5, 11, -13, 6]
>>> L
[0, 10, 7, 5, 11, -13, 6]
>>> insertion(L, 3)
>>> L # Le 5 coule jusqu'entre le 0 et le 10
[0, 5, 10, 7, 11, -13, 6]
>>> insertion(L, 5)
>>> L # Le -13 va jusqu'au fond de la liste
[-13, 0, 5, 10, 7, 11, 6]
>>> insertion(L, 5)
>>> L # Si on l'appelle n'importe comment, cela peut ne rien faire...
[-13, 0, 5, 10, 7, 11, 6]
```

Stop Gitlab

Allez sur GithubDesktop pour faire un commit. Choisissez (avec pertinence) le résumé. Pensez, si possible, à appuyer sur le bouton «Push origin» en haut à droite pour mettre à jour sur le web.

Maintenant qu'on dispose de la procédure d'insertion, il suffit de l'appliquer séquentiellement en commençant par la gauche et en allant jusqu'à la droite pour obtenir une liste triée.

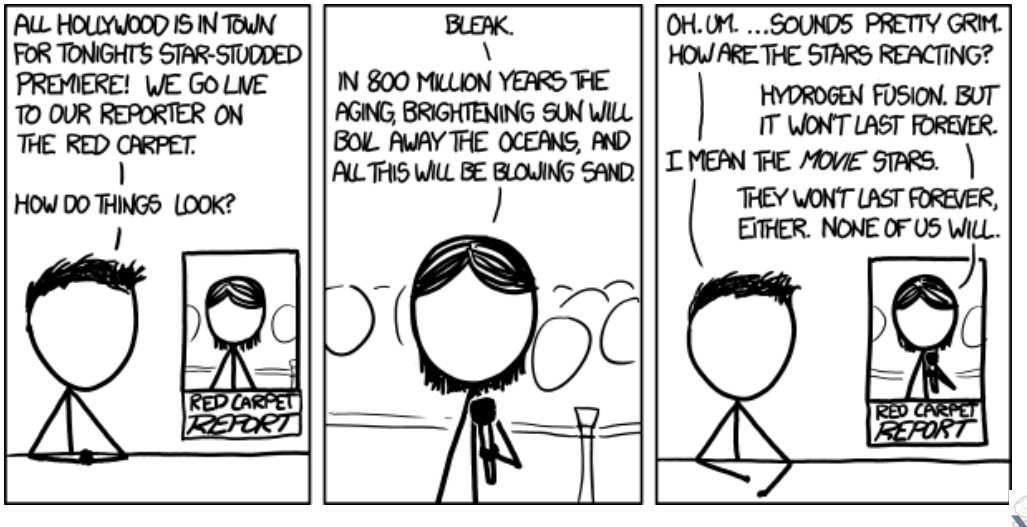
4. Écrire une fonction `insertion_sort(liste)` qui modifie la liste donnée en argument et la rend triée en appliquant autant de fois que nécessaire la fonction `insertion` définie précédemment.

Stop Gitlab

Allez sur GithubDesktop pour faire un commit. Choisissez (avec pertinence) le résumé. Pensez, si possible, à appuyer sur le bouton «Push origin» en haut à droite pour mettre à jour sur le web.

À noter que vous devriez pouvoir prouver [2] votre algorithme à l'aide de l'invariant de boucle suivant :

« Après l'étape i, la liste située entre les positions 0 et i est triée. »



«But what's the buzz about the film ?» «We're hoping it's distracting.»

2. Cela pourra être un exercice intéressant quand on aura fait le cours sur la question. En attendant, vous pouvez regarder la vidéo suivante pour avoir une idée de la méthode : `https://youtu.be/AtR-RlsBeQw`

Partie III

Merge sort (ou tri fusion)

La complexité des deux algorithmes précédents n'étant pas optimale dans le cas général (seulement quadratique en moyenne), on a cherché à développer d'autres méthodes qui assurent une complexité minimale. Parmi ceux-ci figure le « fusion sort » dont on va chercher à comprendre le principe.

Partons de la fin : on suppose que l'on a déjà réussi à trier deux listes qui correspondent aux deux moitiés de la liste principale que l'on veut trier. Il suffit alors d'insérer les éléments dans le bon ordre pour construire une liste globalement triée qui correspond à la fusion des deux moitiés.

5. Écrire une fonction `fusion(L1, L2)` qui prend deux listes `L1` et `L2` qui sont supposées déjà triée et renvoie une nouvelle liste (triée) qui contient l'ensemble des élément de `L1` et `L2`.

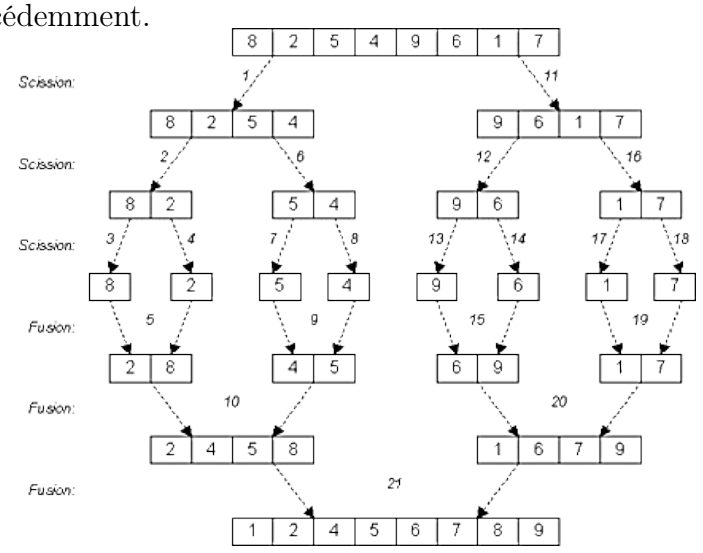
Maintenant que l'on sait fusionner deux moitiés de listes déjà triées, il n'y a plus qu'à savoir les trier. Pour ce faire, on va utiliser un algorithme récursif :

— Si la liste donnée à trier est de taille 1 ou 0, alors elle est déjà triée et on peut la renvoyer telle quelle.

— sinon, on applique le tri fusion sur la première moitié (en stockant le résultat dans une variable bien nommée), de même sur la seconde moitié (pareil en stockant le résultat) et finalement on renvoie la fusion des deux listes triées précédentes en utilisant la fonction `fusion`.

6. Écrire une fonction `merge_sort(liste)` qui prend une liste non triée en entrée et renvoie une liste triée (par tri fusion) en sortie. Votre fonction devra forcément :
   — ne comporter qu'une unique instruction conditionnelle (pour les listes de taille 1 ou 0) ;
   — être récursive et s'appeler par deux fois (il y a deux moitiés à trier) ;
   — appeler la procédure de fusion définie précédemment.

En raisonnant sur le dessin ci-contre, et notamment en comptant le nombre k d'étapes effectuées lors de la remontée par fusions successives, en fonction de la taille n de la liste, vous pouvez comprendre pourquoi on dit que la complexité de cet algorithme de tri est en $\mathcal{O}(n\ln(n))$



Partie IV

Bucket sort (ou tri par baquet)

Un petit dernier pour la route : le « bucket sort » revient à mettre les éléments (par exemple des cartes) sur des tas et de recommencer en variant le point de comparaison. Supposons que vous ayez sélectionné toutes les figures dans un jeu de tarot (à savoir les rois, les dames, les cavaliers et les valets) et que vous vouliez reclasser les familles ensemble. Vous prenez donc votre tas de carte et séparez selon la figure : les rois entre eux, les reines entre elles, etc. Une fois ce premier tour achevé, vous récupérez (dans l'ordre) les rois, les dames, les cavaliers et enfin les valets puis vous recommencez à les mettre sur des tas, mais en

regardant cette fois-ci la couleur (pique, cœur, carreau ou trèfle). Quand vous remettrez vos tas ensembles, les figures de chaque couleur seront séparées et classées dans l'ordre.

Ici, on va se restreindre à trier des entiers positifs et la caractéristique de tri (couleur et valeur dans l'exemple précédent) sera le chiffre des unités, puis celui des dizaines, des centaines, etc.

7. Écrire une fonction `buckets(liste, i)` qui distribue les nombres de la liste donnée en argument en les classant selon le chiffre devant 10^i dans leur décomposition en écriture décimale. La fonction doit renvoyer une liste de dix listes, une pour chaque chiffre (de 0 à 9) que l'on peut trouver devant 10^i. Par exemple, on devrait obtenir les résultats suivants :

```
>>> L = [0, 10, 7, 205, 11, 413, 6]
>>> buckets(L, 0)
[[0, 10], [11], [], [413], [], [205], [6], [7], [], []]
>>> buckets(L, 1)
[[0, 7, 205, 6], [10, 11, 413], [], [], [], [], [], [], [], []]
>>> buckets(L, 2)
[[0, 10, 7, 11, 6], [], [205], [], [413], [], [], [], [], []]
>>> buckets(L, 3)
[[0, 10, 7, 205, 11, 413, 6], [], [], [], [], [], [], [], [], []]
```

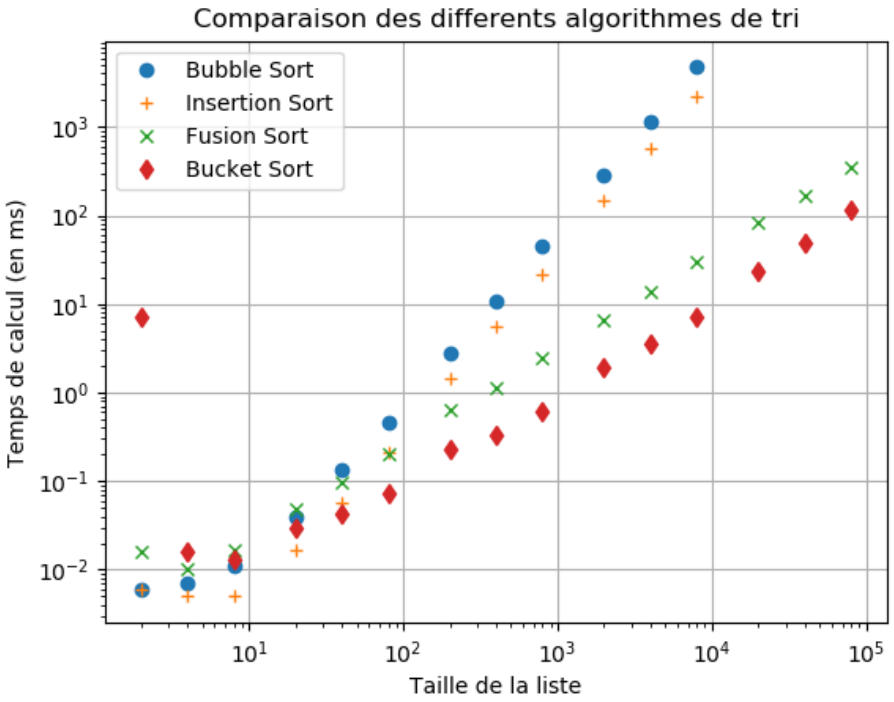
Il faut à présent récupérer les éléments dans les différents baquets en conservant l'ordre obtenu.

8. Écrire une fonction `recupere_buckets(liste_de_listes)` qui renvoie une simple liste contenant tous les éléments de la première liste, puis tous ceux de la deuxième, puis tous ceux de la troisième, etc. Attention, on va écrire un algorithme général, il n'est pas dit qu'il y ait effectivement 10 liste dans `liste_de_listes`. En repartant des exemples précédents, on devrait obtenir

```
>>> L = [0, 10, 7, 205, 11, 413, 6]
>>> recupere_buckets(buckets(L, 0))
[0, 10, 11, 413, 205, 6, 7]
>>> recupere_buckets(buckets(L, 1))
[0, 7, 205, 6, 10, 11, 413]
>>> recupere_buckets(buckets(L, 2))
[0, 10, 7, 11, 6, 205, 413]
>>> recupere_buckets(buckets(L, 3))
[0, 10, 7, 205, 11, 413, 6]
```

9. Enfin, armés de ces deux fonctions, on peut à présent écrire la fonction `bucket_sort(liste)` qui prend une liste non triée d'entiers positifs en argument et la trie en utilisant la méthode décrite plus haut. On pensera à bien appeler les deux fonctions définies précédemment et on réfléchira à une condition d'arrêt adéquate en fonction de la valeur du maximum de la liste.

Pour information, on présente sur la page suivante l'évolution des temps de tri pour des listes aléatoires pour les divers algorithmes présentés dans ce TP en fonction de la taille n de la liste (avec des échelles logarithmiques). Quel semble être le meilleur algorithme ? Ont-ils tous la même complexité asymptotique moyenne sur des listes aléatoires ?



StackSort connects to StackOverflow, searches for 'sort a list', and downloads and runs code snippets until the list is sorted.