Tri

1 Description du problème

On s'intéresse au problème du tri d'une liste.

Plus précisément, on dispose d'une liste L de n éléments comparables, et on veut rendre la liste des éléments de L dans l'ordre croissant (il est facile d'adapter les idées pour obtenir l'ordre décroissant).

```
    def tri_naif(L):
        """
        L'idée est d'aller chercher le plus petit élément de L
        puis de le placer en première position.
        On recommencer à partir de la position 2.
        """
        n = len(L)
        for i in range(0, n-1):
            # on cherche le minimum de L[i:]
            m = i # indice courant du minimum
            for j in range(i, n):
                if L[j] < L[m]:
                    m = j
            # le minimum de L[i:] est à la position m,
            # plaçons le au début, c'est à dire en i
            L[i], L[m] = L[m], L[i]
        return L
```

```
    L = [-1,5,2,4,6]; tri_naif(L)
```

```
[-1, 2, 4, 5, 6]
```

```
    L = list("azertyuiop")
    tri_naif(L)
```

```
['a', 'e', 'i', 'o', 'p', 'r', 't', 'u', 'y', 'z']
```

2 Tri par insertion.

2.1 Description de l'algorithme

On appelle également parfois ce tri le tri à bulles.

Le principe est simple.

Une liste à 1 élément est déjà triée

Si une liste possède n éléments et que les n-1 premiers sont dans l'ordre, alors il sufffit d'insérer le dernier au bon endroit pour obtenir une liste triée

L'algorithme est une répétition de ces idées : on tri les deux premiers éléments de L, puis on insère le troisème élément, puis le quatrième et ainsi de suite

L'insertion se fait en inversant les éléments : on fait remontrer l'éléments à insérer jusqu'à la bonne place.

Exemple à la main Soit à trier le tableau L = [2,1,5,1]

1. Etape 1 : [1, 2, 5, 1] et les deux premiers éléments sont triés.
2. Etape 2 : [1, 2, 5, 1] et les 3 premiers éléments sont triés.
3. Etape 3 : [1, 2, 1, 5] puis [1, 1, 2, 5]

```
def tri_insertion(L):
    for i in range(1, len(L)):
        # on va placer l'élément d'indice i au bon endroit dans la liste
        j = i - 1 # indice juste avant
        x = L[i] # utile pour la preuve.
        while j >= 0 and L[j] > x:
            L[j + 1] = L[j]
            # on fait reculer l'élément en position j de 1 place.
            j = j - 1
        L[j + 1] = x
    return L
```

```
L = [2,1,4,5, 0,1]
```

```
tri_insertion(L)
```

```
[0, 1, 1, 2, 4, 5]
```

```
import random
```

```
%timeit [random.random() for _ in range(5000)]
%timeit tri_naif([random.random() for _ in range(5000)])
```

```
1000 loops, best of 3: 476 µs per loop
1 loop, best of 3: 889 ms per loop
```

```
%timeit tri_insertion([random.random() for _ in range(5000)])
```

```
1 loop, best of 3: 924 ms per loop
```

2.2 Correction de l'algorithme

Prouvons que notre algorithme est correct, ie que la liste retournée est bien triée dans l'ordre croissant.

On note n la longueur de la liste à trier. On va montrer que pour tout i entre 1 et $n-1$, à la fin de la boucle for d'indice i, les éléments d'indices entre 0 et i de la liste L sont triés dans l'ordre croissant.

Avant de rentrer dans la boucle for, l'élément d'indice 0 est seul dans la sous-liste considérée et est donc une liste triée.

Supposons que pour un i < n-1, les i premiers éléments de la liste sont triés et montrons qu'après le passage dans la boucle pour cette valeur de i, les i+1 premiers éléments sont triés. On a donc $L[0:i]$ qui est triée, j = i-1 (c'est l'indice du dernier des éléments déjà trié) et x = L[i] qui est le seul des i+1 premiers éléments à n'éventuellement pas être à sa place.

Montrons maintenant que la propriété "les éléments d'indices entre i et j+1 consistuent une liste triée et ils sont tous supérieurs à x" est invariante par passage dans la boucle while. Au départ, j+1=i et donc la liste évoquée par cette propriété est [x] et est donc triée et consitituée d'éléments $\geq x$.

Si on exécute un passage dans la boucle while c'est que $L[j] > x$. On remplace l'élément d'indice j+1 par l'élément d'indice j qui était plus petit que les éléments d'indices de j à i. On ne modifie pas le caractère trié de la liste. Ensuite j est décrémenté. L'élément d'indice j+1 qui est notre ancien élément d'indice j, est donc > x et tous les éléments entre j+1 et i consituent une liste triée d'éléments $\geq x$.

On sort de la boucle while en au maximum i-1 étape, cette boucle se termine donc. Avant de prouver que la liste des éléments entre 0 et i sont triés, remarquons que les éléments de la listes ne sont qu'échangés (en effet, le dernier L[j] à être "écrasé" est remplacé par x à la sortie de la boucle, et x était le premier élément à être "écrasé"), et donc on ne supprime ne ni cré d'éléments.

De plus, il y a 2 conditions de sortie. Examinons les.

On est sorti car j = -1. D'après la propriété invariante (appelé invariant de boucle), les éléments d'indices entre 0 et i consituent une liste triée (d'élément triés $\geq x$).

On est sorti car $L[j] \leq x$. Dans ce cas on sait que $L[j+1:i+1]$ est triée et consitité d'éléments triés $\geq x$ et donc on a bien $L[j:i+1]$ qui se retrouve triée et constituée d'éléments $\geq x$. Comme de plus, $L[0:i]$ était triée par hypothèse de récurrence, $L[0:j]$ est triée et consituée d'éléments $\leq L[j] \leq x$.

Finalement la liste $L[0:i+1]$ est bien triée

Dans les deux cas, on a placé l'élément d'indice i à la place qui lui revient dans la liste des i+1 premiers éléments.

Pour conclure : par récurrence, à la fin du dernier passage dans la boucle for (pour $i = n-1$), les n premiers éléments de la liste sont triés et donc L est triée !

2.3 Complexité

On se place dans un cas extrême : le tableau est trié et dans ce cas on fait exactement 0 passage dans la boucle while à chaque fois. Il y a donc $3 \times (n-1)$ opérations effectuées.

Le deuxième cas traitable est l'autre extrême : le tableau est trié, mais dans l'ordre décroissant. Dans ce cas, c'est à chaque fois la condition $j < 0$ qui fait qu'on sort du while. Pour chaque i entre 1 et $n-1$, on effectue donc 2 affectations + i fois 2 comparaisons et 3 affectations. Remaquez qu'il s'agit du cas le plus défavorable, car on effectue le nombre maximum de passage dans la boucle while.

Le nombre total d'opérations est donc $\sum_{i=1}^{n-1}(2+5i) = (n-1) + \frac{5}{2}n(n-1)$ qui est de l'ordre de grandeur de n^2.

L'étude dans le cas général n'est pas aussi facile. On peut montrer que l'ordre de grandeur reste n^2

3 Tri rapide

3.1 Algorithme

On l'appelle souvent "quicksort". C'est un algorithme récursif qui utilise le principe "diviser pour régner".

Encore une fois, le cas d'arrêt de la récursion (le cas facile, que l'on traite directement sans plus d'appel à la fonction) sont les listes de longueur 0 et 1.

Si la liste est plus longue, on choisit un "pivot" (suivant une méthode à déterminer) qui est un élément de la liste. On considère ensuite 2 listes : la liste des éléments inférieurs au pivot, et la liste des éléments supérieurs. On trie ensuite les deux sous-listes, le résultat du tri total est la concaténation des listes triées, avec le pivot inséré entre elles. Les éléments égaux aux pivots sont mis dans la première liste.

L = [2,1,4,0]

On choisit 2 comme pivot, les sous-listes sont L1 = [1, 0] et L2 = [4].

Pour trier L1, on choisit 1 comme pivot, les listes obtenues sont [0] et []. L1 triée est donc a liste [0, 1]

Donc L triée est [0, 1, 2, 4]

```
    def partitionne(L, pivot):
        """
        Retourne les deux sous-listes précédemment mentionnées
        plus une liste des éléments égaux au pivot
        """
        L1 = []
        Lp = []
        L2 = []
        for e in L:
            if e < pivot:
                L1.append(e)
            elif e == pivot:
                Lp.append(e)
            else:
                L2.append(e)
        return L1, Lp, L2

    def rassemble(L1, Lp, L2):
        """
        Reconstruit la liste L
        """
        return L1 + Lp + L2
```

```
    def choisit_pivot(L):
        """
        Retourne l'indice du pivot choisi.
        """
        return 0

    def quicksort(L):
        if len(L) < 2:
            return L
        pivot = L[choisit_pivot(L)]
        L1, Lp, L2 = partitionne(L, pivot)
        L1 = quicksort(L1)
        L2 = quicksort(L2)
        return rassemble(L1, Lp, L2)
```

```
    L = [2,1,4,0, -1, 5 , 6 , 0]
```

```
    quicksort(L)
```

```
[-1, 0, 0, 1, 2, 4, 5, 6]
```

```
    %timeit quicksort([random.random() for _ in range(5000)])
    %timeit tri_insertion([random.random() for _ in range(5000)])
```

```
100 loops, best of 3: 12 ms per loop
1 loop, best of 3: 910 ms per loop
```

3.2 Analyse de complexité

On note $T(n)$ le nombre d'opérations nécessaires au tri d'une liste de longueur n.

3.2.1 Cas le plus favorable.

Le cas le plus favorable est le cas où le pivot est bien choisit et les listes à trier sont de longueur n/2, on peut approximer par $T(n) = 2n + 2T(n/2)$. Cette fois la suite n'est pas d'une forme classique.

On a noté $2n$ le coût de la partition et de la concaténation de listes.

Pour traiter ce cas, on suppose que $n = 2^k$ est une puissance de 2 et on note $T_2(k) = T(n) = T(2^k)$.

La relation devient $T_2(k) = 2 \times 2^k + 2T_2(k-1) = 2 \times 2^k + 2(2 \times 2^{k-1} + 2T_2(k-2)) = 4 \times 2^k + 4T_2(k-2) = 6 \times 2^k + 8T_2(k-3) = \cdots = 2k2^k + 2^k T_2(0) \approx 2k2^k$.

On admet que cette formule se généralise pour $T(n)$ et on obtient $T(n) = O(n\ln(n))$ qui est asymptotiquement infiniment meilleur que la complexité du tri à bulle.

3.2.2 Cas défavorable

Avec notre choix de pivot, le cas le plus défavorable est obtenu quand une liste comporte tous les éléments sauf le pivot, c'est à dire quand la liste de départ est déjà triée !

Dans ce cas $T(n) = O(n) + T(n-1) = nO(n) = O(n^2)$.

3.2.3 Cas général

Le cas général est hors de notre portée, mais avec un meilleur choix de pivot, on peut assez facilement éviter le cas défavorable et se rapprocher en pratique de $O(n \ln(n))$.

4 Tri fusion

4.1 Principe

Principe de fonctionnement :

— On coupe la liste en deux sous-listes dont les longueurs sont égales ou distante de 1
— On tri récursivement les deux sous-listes
— la liste finale est obtenue par "fusion" des résultats, ie on reconstruit une liste triée constituée des éléments de deux listes déjà triées

```
    def fusion(L1, L2):
        """
        L1 et L2 sont deux listes triées.
        Retourne la liste triée contenant les éléments de L1 et L2.
        Si des éléments sont égaux dans les listes L1 et L2,
        on place d'abord les éléments de L1
        """
        L = []
        i1 = 0 # indice courant dans L1
        i2 = 0 # indice courant dans L2
        n1 = len(L1)
        n2 = len(L2)
        while i1 < n1 and i2 < n2:
            if L1[i1] <= L2[i2]:
                L.append(L1[i1])
                i1 += 1
            else:
                L.append(L2[i2])
                i2 += 1
        # on a épuisé une des deux listes, il suffit de mettre tous les éléments
        # restant dans l'autre à la fin de L
        L.extend(L1[i1:])
        L.extend(L2[i2:])
        return L

    def tri_fusion(L):
```

```
        n = len(L)
        if n <= 1: # cas d'arrêt de la récursion
            return L
        moitie = n // 2
        L1 = tri_fusion(L[:moitie])
        L2 = tri_fusion(L[moitie:])
        return fusion(L1, L2)
```

```
L = [2,1,4,0, -1, 5 , 6 , 0]; tri_fusion(L)
```

```
[-1, 0, 0, 1, 2, 4, 5, 6]
```

```
%timeit tri_fusion([random.random() for _ in range(5000)])
```

```
100 loops, best of 3: 17.2 ms per loop
```

4.1.1 Analyse de complexité

Cette fois encore, on note $T(n)$ le nombre d'opérations (comparaison, affectation) nécessaires au tri d'une liste de longueur n.

On a $T(n) = 2T(n/2) + 5n$ (au pire 5 opération pour ajouter un élément à L dans fusion), et avec $n = 2^k$ on trouve $T_2(k) = 2T(k-1) + 5 \times 2^k$.

En réappliquant la relation plusieurs fois, $T_2(k) = 4T(k-2) + 5 \times 2 \times 2^{k-1} + 5 \times 2^k = 2^k T_2(0) + 5k2^k$

En repassant à n, $T(n) = n + 5n \log_2(n)$, qui est de l'ordre de grandeur $n \ln(n)$.

5 Amélioration des implémentations

5.1 Le tri rapide

Notre première implémentation est mauvaise car on cré énormément de listes qui sont en fait inutiles. Il suffirait de réordonner partiellement la liste L pour séparer les éléments plus petits des éléments plus grand que le pivot.

```
    def part_en_place(L, m, M):
        """
        Cré une partition de L entre les indices m et M en plaçant le pivot
        choisit à sa position finale.
        Tous les éléments avant le pivot lui seront inférieurs,
         ceux après seront supérieurs.
        Retourne l'indice ou se situe le pivot.
        """
        # le principe est similaire à celui du tri à bulles.
        indice_pivot = m
        pivot = L[indice_pivot]
        indice_stock = m # endroit courant de la liste où placer
        # les éléments inférieurs au pivot
```

```
        L[indice_pivot], L[M] = L[M], L[indice_pivot] # place le pivot en dernière
        # position, de manière temporaire
        for i in range(m, M): # c'est le pivot en dernier
            if L[i] < pivot: # on le place au début de la liste
                L[i], L[indice_stock] = L[indice_stock], L[i]
                indice_stock += 1 # on continuera à placer les éléments inférieurs
                # juste après
        L[indice_stock], L[M] = L[M], L[indice_stock] # remettre le pivot
        # juste après les éléments
        # qui lui sont inférieurs
        return indice_stock

    def tri_rapide_rec(L, m, M):
        if m >= M - 1:
            return L
        indice_pivot = part_en_place(L, m, M)
        tri_rapide_rec(L, m, indice_pivot - 1)
        tri_rapide_rec(L, indice_pivot + 1, M)
        return L

    def tri_rapide(L):
        return tri_rapide_rec(L, 0, len(L) - 1)
```

```
    L = [2,1,4,0, -1, 5 , 6 , 0]; tri_rapide(L)
```

```
[-1, 0, 0, 1, 2, 4, 5, 6]
```

```
    %timeit tri_rapide([random.random() for _ in range(5000)])
```

```
100 loops, best of 3: 10.1 ms per loop
```

Le tri obtenu est beacoup plus rapide pour les listes "longues", mais ce n'est pas le cas pour les "petites" listes

```
    %timeit tri_insertion([random.random() for _ in range(10)])
    %timeit tri_rapide([random.random() for _ in range(10)])
```

```
100000 loops, best of 3: 5.92 µs per loop
100000 loops, best of 3: 7.52 µs per loop
```

5.1.1 Une autre optimisation

On remarque que pour les "petites" listes, notre algorithme sophistiqué est moins bon que le tri par insertion ! Profitons-en...

```
%pylab inline
```

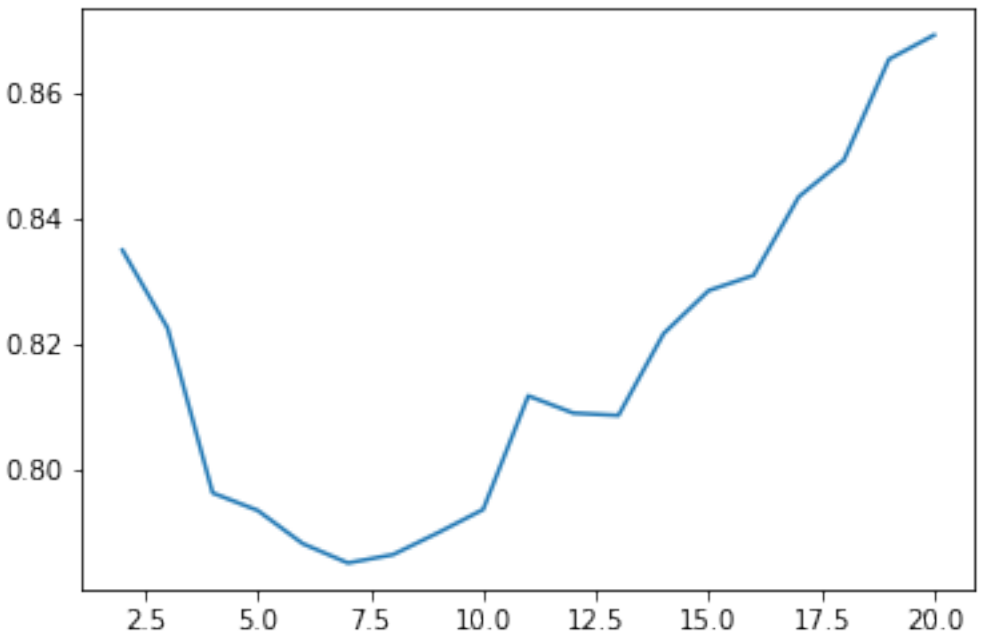
```
Populating the interactive namespace from numpy and matplotlib
```

```
import timeit

X = list(range(2, 21))
Y1 = []
glob = {"tri_insertion": tri_insertion, "random": random, "quicksort": quicksort, "tri_fu
for n in X:
    glob["n"] = n
    temps_inser = timeit.timeit('tri_insertion([random.randint(0, 10**3) for _ in range(n
                                globals=glob)
    temps_quick = timeit.timeit('quicksort([random.randint(0, 10**3) for _ in range(n)])'
                                globals=glob)
    Y1.append(temps_inser/temps_quick)
```

```
plt.plot(X, Y1)
```

```
[<matplotlib.lines.Line2D at 0x7f89e4100f28>]
```



```
    def tri_insertion_partiel(L, m, M):
        """
        tri les éléments de L[m:M+1] en utilisant le tri par insertion
        """
        for i in range(m + 1, M + 1):
            # on va placer l'élément d'indice i au bon endroit dans la liste
            j = i - 1 # indice juste avant
            x = L[i]
            while j >= 0 and L[j] > x:
                L[j + 1] = L[j]
                # on fait reculer l'élément en position j de 1 place.
                j = j - 1
            L[j + 1] = x
        return L

    def tri_rapide_rec2(L, m, M):
        if M - m + 1 < 10:
            return tri_insertion_partiel(L, m, M)
        indice_pivot = part_en_place(L, m, M)
        tri_rapide_rec2(L, m, indice_pivot - 1)
        tri_rapide_rec2(L, indice_pivot + 1, M)
        return L

    def quicksort2(L):
        return tri_rapide_rec2(L, 0, len(L) - 1)
```

```
    L =  np.random.randint(0, 100, 15); L
```

```
array([23, 79, 77, 91, 15, 70, 62, 39, 10, 53, 73, 81, 94, 16, 55])
```

```
    quicksort2(L)
```

```
array([10, 15, 16, 23, 39, 53, 55, 62, 70, 73, 77, 79, 81, 91, 94])
```

```
    def liste_chaine(n):
        chaine = "a"*15 + "b"*15
        L1 = list(chaine)
        L = [""]*n
        for i in range(n):
            random.shuffle(L1)
            L[i] = "".join(L1)
        return L
    L = liste_chaine(5000)
```

```
%timeit random.shuffle(L)
%timeit random.shuffle(L); quicksort2(L)
%timeit random.shuffle(L); tri_rapide(L)
%timeit random.shuffle(L); quicksort(L)
```

```
10000 loops, best of 3: 111 µs per loop
100 loops, best of 3: 10.2 ms per loop
100 loops, best of 3: 10.5 ms per loop
100 loops, best of 3: 11.7 ms per loop
```