Algorithmes de tri

Table des matières

Soit **tab** un tableau (à une dimension, donc une liste sous Python) ne contenant que de valeurs numériques (ou éventuellement non numériques mais comparables deux à deux par une relation d'ordre) ;
les n éléments numériques du tableau **tab** sont numérotés de **0** à **n-1** et l'on se propose de les **trier** dans un ordre donné, par exemple **dans l'ordre croissant**.

Le tableau est donc, a priori, initialement non trié.
Nous allons étudier plusieurs algorithmes donnant une solution au problème et les comparer. On verra qu'ils peuvent être plus ou moins efficaces selon la taille du tableau **tab** à trier.

1 Tri par insertion

1.1 Principe

L'idée est de trier progressivement le tableau : supposant que $t[0:k]$ est déjà trié, on insère $t[k]$ àsa place parmi les valeurs de trié $t[0:k]$ (en décalant les plus grandes valeurs d'un cran vers la droite si besoin) de sorte que $t[0:k+1]$ se retrouve trié.
Plus précisément, pour k allant de 1 à $n-1$:

- on stocke la valeur de $t[k]$ dans une variable *temp* et on initialise j à la valeur k.
- tant que $j > 0$ et $temp < t[j-1]$ (on a unue valeur supérieure à *temp*), on décale $t[j-1]$ d'un cran vers la droite et on décrémente j
- à la sortie de la boucle on installe *temp* à sa place

1.2 Algorithme

```
1 def triParInsertion(Tableau tab):
2   n = taille(tab) \# n : taille
      du tableau
3   pour k de 1 a n-1:
4     temp = tab[k]
5     j = k
6     tant que j > 0 et tab[j - 1]
        > temp:
7       tab[j] = tab[j - 1]
8       j = j - 1
9     tab[j] = temp
```

Remarque 1 : Cette version trie le tableau "en place". Aussi le tableau entré en argument est changé. On peut vouloir conservé le tableau initial. Dans ce cas on peut travailler sur une vraie copie du tableau créée à l'extérieur de la fonction, soit créer une copie du tableau dans la fonction qui devra alors retourner le tableau trié.
Remarque 2 : le test **while** n'est valable ici que parce que l'on a une évaluation paresseuse des expressions booléennes, pour éviter d'avoir à tester $temp < t[-1]$.

1.3 Correction, complexité

1.3.1 Justification

Terminaison Pour la boucle **while**, la valeur de j diminue strictement àchaque passage. Il y aura donc au maximum k itérations et donc la boucle **while** termine bien.
La boucle **for** termine évidemment.

Correction L'algorithme retourne bien une liste triée. Pour ce faire il suffit de considérer l'invariant de boucle " aprés l'exécution de la $k-$ième itération de la boucle **for**, les valeurs $t[0], \ldots, t[k]$ sont les valeurs initiales rangées par ordre croissant"

1.3.2 Complexité

On considérera toujours la complexité en termes de comparaisons entre deux valeurs du tableau. Dans notre algorithme, à chaque comparaison entre deux valeurs du tableau, on effectue un test $j > 0$, au plus une affectation et au plus une décrémentation. Donc le nombre total d'opérations effectuées est asymptotiquement proportionnel au nombre de comparaison entre deux valeurs du tableau.

Pire des cas Il est obtenu lorsque tous les tests $temp < t[j-1]$ ont une valeur **True**, c'est àdire lorsque la liste de départ est rangée dans l'ordre décroissant. Ainsi on effectue $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$ comparaisons : **la complexité dans le pire des cas est $O\left(n^2\right)$**.

Meilleur des cas Il est obtenu lorsque tous les premiers tests de la boucle **while** ont une valeur **False**, c'est àdire lorsque la liste de départ est rangée dans l'ordre croissant. Ainsi on effectue $\sum_{k=1}^{n-1} 1 = n-1$ comparaisons : **la complexité dans le meilleur des cas est $O\left(n\right)$**.

Complexité en moyenne HP On peut estimer qu'en moyenne l'insertion se fait au milieu du sous-tableau concerné. On obtiendrait $\sum_{k=1}^{n-1} \frac{k}{2} = \frac{n(n-1)}{4}$ comparaisons : La complexité moyenne est $O\left(n^2\right)$.

Exercice 1. On veut écrire une version récursive de ce tri-insertion. Pour cela :

- On écrit une fonction **insere(x,t)** qui insère de façon récursive x à sa place dans t : si le tableau est vide ou si x est plus grand que le dernier terme du tableau, on retourne $t.append(x)$, sinon on insére x dans le sous-tableau t privé du dernier et on retourne le résultat obtenu auquel on ajoute en bout le dernier élément de t.
- On écrit une fonction **tri_inser(t)** qui retourne t si t posséde au plus 1 élément et retourne le résultat de l'insertion de $t[-1]$ dans une version triée (récursivement) du sous-tableau $t[0:-1]$

2 Tri rapide (quicksort)

2.1 Principe

La méthode consiste à placer un élément (appelé **pivot**) du tableau à sa place définitive en permutant tous les éléments de telle sorte que ceux qui lui sont inférieurs se retrouvent à sa gauche et ceux qui lui sont supérieurs se retrouvent à sa droite : cette opération s'appelle le **partitionnement**.
Ensuite, il ne reste plus qu'à trier récursivement les deux tableaux séparés par le **pivot**
Concrètement, pour partitionner un sous-tableau :
• on choisit le pivot : soit de façon automatique (le premier, le dernier...) soit aléatoirement ;
• on place tous les éléments inférieurs au pivot dans un tableau t1 et ceux supérieurs au pivot dans un tableau t2
• on retourne le tableau trié (récursivmeent) obtenu à partir de t1, concaténé avec [pivot] et avec le tableau trié obtenu à partir de t2

2.2 Algorithme

```
def quicksort(t):
    if len(t) <=1:
        return(t)
    else:
        pivot = t[0]
        t1, t2 = [],[]
        for x in t[1:]:
            if x<pivot:
                t1.append(x)
            else:
                t2.append(x)
        return(quicksort(t1) + [pivot] + quicksort(t2))
```

2.3 Correction, complexité

2.3.1 Justification

Terminaison L'algorithme se termine bien car les longueurs des tableaux $t1$ et $t2$ transmis lors des appels récursifs sont strictement inférieures à la longueur du tableau t

Correction On peut effectuer la récurrence suivante : P_n " l'algorithme quicksort appliqué à un tableau de longueur inférieure ou égale à n retourne le tableau trié", ce qui se démontre aisément

2.3.2 Complexité

On note $C(n)$ le nombre de comparaisons entre deux éléments du tableau.

Pire des cas Il est obtenu lorsque parmi les tableaux $t1$ et $t2$, l'un des deux est toujours vide. Cela lorsque le tableau initial est trié ou rangé dans l'ordre inverse.

On obtient alors : $\forall n \geqslant 2, C(n) = n-1+C(n-1)$ avec $C(0) = C(1) = 0$. On obtient alors $C(n) = \dfrac{n(n-1)}{2}$:

la complexité dans le pire des cas est $O\left(n^2\right)$.

On peut améliorer cette complexité dans le pire des cas, il suffirait de choisir le pivot de façon aléatoire

Complexité en moyenne On estime que le choix du pivot découpe en deux tableaux $t1$ et $t2$ dont les tailles sont équiprobables.

On obtient alors : $\forall n \geqslant 2, C(n) = n - 1 + \dfrac{1}{n}\displaystyle\sum_{p=0}^{n-1}(C(p) + C(n-1-p))$ avec $C(0) = C(1) = 0$.

Donc **la complexité en moyenne est $O\left(n\log_2(n)\right)$**

Dem.

3 Tri fusion

3.1 Principe

On coupe en deux parties à peu près égales les données à trier, on applique le tri fusion à chacune des deux parties (la récursivité ou l'itération va bien s'arrêter car le tri fusion d'un tableau d'un seul élément est trivial) et on fusionne les deux parties.

3.2 Algorithme de fusion

On veut écrire un algorithme de fusion de deux tableaux triés $t1$ et $t2$.
Une première possibilité est de raisonner "récursivement" :

- si l'un des tableaux est vide, on renvoie l'autre
- sinon on renvoie le tableau commençant par la plus petite valeur α entre $t1[0]$ et $t2[0]$ suivie de la fusion de $t1$ et $t2$ en enlevant α du tableau dont il est issu.

Cela donne

```
def fusion(t1,t2):
    if t1 == []:
        return(t2)
    elif t2 ==[]:
        return(t1)
    elif t1[0] < t2[0]:
        return( [t1[0]] + fusion(t1[1:],t2))
    else :
        return( [t2[0]] + fusion(t1,t2[1:]))
```

Une autre possibilité est de partir d'un tableau résultat *tres* initialisé au tableau vide et de décrire les deux tableaux $t1$ et $t2$, ajouter à *tres* le plus petit des deux éléments et de se déplacer d'un cran à dans le tableau dont on a pris ce plus petit élément. Puis une fois qu'on est arrivé en fin d'un tableau, on ajoute les éléments restants du second tableau. Cela donne le principe suivant :

— On initialise deux entiers $i1$ et $i2$ à0 et une liste t à []

— tant que $i1 < len(t1)$ et $i2 < len(t2)$, on compare $t1[i1]$ et $t2[i2]$, et on ajoute à *tres* le plus petit d'entre eux et on incrémente l'indice correspondant $i1$ ou $i2$

— à la sortie de la boucle précédente, on a soit $i1 = len(t1)$ soit $i2 = len(t2)$ donc on a décrit complétement un des deux tableaux. On ajoute alors *tres* les éléments restants du tableau qui n'a pas été décrit complétement.

Cela donne

```
def fusionite(t1,t2):
    i1, i2, n1, n2 = 0, 0, len(t1), len(t2)
    tres=[]
    while i1<n1 and i2 <n2:
        if t1[i1] < t2[i2]:
            tres.append(t1[i1])
            i1 += 1
        else:
            tres.append(t2[i2])
            i2 += 1
    if i1 == n1:
        for x in range(i2, n2):
            tres.append(t2[x])
    else:
        for x in range(i1, n1):
            tres.append(t1[x])
    return(tres)
```

La complexité de ces deux algorithmes est $O(len(t1) + len(t2))$

Cependant, pour éviter de remplir la pile d'appels récursifs, on aura intéret à utiliser la version itérative de l'algorithme de fusion

3.3 Algorithme de tri

On suppose écrit un algorithme de fusion. On reprend le principe de l'algorithme de tri-fusion.

— Si le tableau n'a qu'une seule valeur, le tableau est trié

— Sinon, on coupe le tableau en deux, on trie (récursivmeent) les deux sous-tableaux obtenus, puis on fusionne les résultats

```
def tri_fu(t):
    if len(t) <2:
        return(t)
    else :
        m = len(t)//2
        return(fusion(tri_fu(t[:m]), tri_fu(t[m:])))
```

3.4 Justification

La terminaison est assurée par le fait que dans les appels récursifs, les tableaux concernés sont de longueur de plus en plus petite.
La correction s'obtient par récurrence forte sur la longueur du tableau.

3.5 Complexité

On note $C(n)$ le nombre de comparaisons entre deux éléments du tableau. On note $\lfloor x \rfloor$ la partie entière du réel x et $\lceil x \rceil$ le plus petit entier supérieur ou égal àx.
L'algorithme du tri par fusion est un exemple typique illustrant le paradigme "diviser pour régner" : on ramène

le cas à traiter à plusieurs objets de taille strictement inférieure.
On va calculer la complexité dans tous les cas (car le coût sera identique dans tous les cas).
Pour trier un tableau de taille n, il faut partitionner le tableau, ce qui ne nécessite aucune comparaison, en deux tableaux : $t1$ de taille $\left\lfloor \frac{n}{2} \right\rfloor$ et $t2$ de taille $\left\lceil \frac{n}{2} \right\rceil$. On trie ensuite $t1$ (cout : $C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$) puis $t2$ (cout : $C\left(\left\lceil \frac{n}{2} \right\rceil\right)$).
Et enfin on fusionne les deux tableaux (cout : n). Ainsi on a la relation de récurrence :
$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$
On montre alors $\boxed{C(n) = \Theta\left(n \log_2(n)\right)}$

Dem.

4 Calcul de la mediane

La médiane d'une série statistique donnée par un tableau t est la valeut $m = t[i]$ où i est un indice tel que :

— au moins la moitié des valeurs du tableau sont inférieures ou égales à $t[i]$

— au moins la moitié des valeurs du tableau sont supérieures ou égales à $t[i]$

Un algorithme évident pour déterminer une telle médiane est :

— de trier le tableau t

— de retourner la valeur $t[n//2]$ avec n la longueur du tableau

Ainsi, si on trie le tableau, la complexité de recherche de la médiane est au mieux de l'ordre $n\ln(n)$...

5 Complexité optimale d'un algorithme de de tri

Petit complément culturel.
En fait, le tri-fusion (ou le tri rapide) est optimal en termes de complexité en comparaisons : on ne peut pas faire mieux, sans information complémentaire [1], que la complexité en $n\ln(n)$. En effet :
Prenons un tableau de longueur n. Il y a $n!$ ordres possibles pour les éléments de ce tableau. Pour trier ce tableau, il faut effectuer un certain nombre N de tests. Ainsi, avec toutes les réponses possibles de ces N tests, on distingue 2^N permutations distinctes du même tableau.
Ainsi, tout algorithme de tri doit au moins comporter un nombre de tests N tel que $2^N \geqslant n!$. Donc $N \geqslant \log_2(n!)$.
Comme $\ln(n!) \sim n\ln(n)$, on a donc un nombre de comparaisons au moins de l'ordre de $n\ln(n)$

1. Si par exemple on sait que le tableau n'est constitué que par des entiers compris entre 1 et p, il suffit de créer un tableau de longueur p contenant les occurrences de chacun de ces entiers dans le tableau puis de créer le tableau trié correspondant, le tout pour une complexité linéaire