Algorithmes de tri

On le sait, la recherche d'un élément dans une longue liste est fortement accélérée si une relation d'ordre permet de comparer les éléments de celle-ci et que surtout ils sont rangés dans cette liste dans l'ordre (croissant ou décroissant).

En python, des relations d'ordre existent pour à peu près tous les objets :
- Pour des types numériques `int` et `float` mais, heureusement (!), pas pour `complex`
- Pour des chaînes de caractères
- Pour des listes et des tuples (ordre lexicographique) mais pas en mélangeant les deux types.
- Des ensembles (hors programme, mais pas si compliqué : un ensemble s'écrit `s = {o1, o2, o3}` et la relation d'ordre implémentée correspond, sans surprise, à l'inclusion).
  Bien sûr, la relation d'ordre sur les ensembles n'est pas totale...

Sans surprise toutefois, python ne sait comparer des objets de types peu compatibles (une chaîne de caractères et un nombre) et ne définit pas de relation d'ordre entre des nombres complexes, ni entre des dictionnaires.

Ce chapitre s'intéresse alors aux algorithmes permettant de trier une liste d'objets deux à deux comparables, ce qui ne se limite donc pas à des listes de nombres...

1 Algorithmes naïfs

Les algorithmes qui suivent sont, nous le verrons, peu efficaces, mais ils présentent l'intérêt d'être facilement mis en oeuvre, et ne sont pas à dédaigner pour le traitement de listes de longueurs raisonnables (jusqu'à quelques centaines de termes)

1.1 Tri par sélection

Le tri par sélection fonctionne de la manière suivante : on parcourt le tableau à trier, qu'on supposera formé de n termes, à la recherche de son plus grand élément, et on le place à la fin, puis on recommence sur les $n-1$ premiers termes du tableau, et ainsi de suite.

En pseudo-code, le tri par sélection peut être réalisé ainsi :
Entrées : Un tableau (ou une liste) T de n éléments à trier
Sorties : Le tableau modifié et trié
pour i *variant de* $n-1$ *à* 1 **faire**
  $i_m \leftarrow 0$
  pour j *variant de* 1 *à* i **faire**
    si $T[j] > T[i_m]$ **alors**
      $i_m \leftarrow j$
    fin
  fin
  $T[i], T[i_m] \leftarrow T[i_m], T[i]$
fin

1.2 Tri par insertion

Le tri par insertion est sans doute celui que l'on choisirait pour trier à la main, par exemple, un paquet de copies. On ajoute un à un les objets à une liste déjà triée (une liste d'un seul objet étant bien sûr triée...) en l'insérant à la bonne place dans la liste déjà obtenue.

Bien entendu, si le $n+1$-ème objet doit prendre place à la position i au sein des objets précédemment triés numérotés de 1 à n, cela suppose que l'objet précédemment indicé par n se place désormais à l'indice $n+1$, celui anciennement placé en $n-1$ se retrouvera en n, jusqu'à celui qui était à la place i se retrouvera à la place $i+1$.

(L'utilisation d'une autre structure de données qu'une liste python, comme par exemple une liste chaînée pourrait rendre ce déplacement de données, ou ce renommage plus rapide, mais cela ne changerait rien à la complexité de l'algorithme décrit.)

En pseudo-code, le tri par insertion peut être réalisé de la manière suivante (on suppose les éléments de T numérotés, comme en python,de 0 à $n-1$) :

Entrées : Un tableau (ou une liste) T de n éléments à trier
Sorties : Le tableau modifié et trié
pour *i variant de* 1 *à* $n-1$ **faire**
  $x \leftarrow T[i]$
  $j \leftarrow i$
  tant que $j > 0$ *et* $x < T[j-1]$ **faire**
    $T[j] \leftarrow T[j-1]$
    $j \leftarrow j-1$
  fin
  $T[j] \leftarrow x$
fin

1.3 Tri à bulle

Le tri à bulle doit son nom à une bulle d'air piégée entre deux feuilles imperméables et qu'un appui sur l'une entraine le déplacement.

Le principe est le suivant : on parcourt la liste à trier de gauche à droite en échangeant au fur et à mesure les couples d'éléments qui ne sont pas correctement classés. Le résultat de ce premier passage est que le dernier élément de la liste est bien le plus grand. Un second parcours de la gauche à la droite en s'arrêtant une case avant la fin conduit à ce que l'avant-dernier terme de la liste soit à sa place.

Et ainsi de suite...

Voici l'algorithme du tri à bulle en pseudo-code :
Entrées : Un tableau (ou une liste) T de n éléments à trier
Sorties : Le tableau modifié et trié
pour *i variant de* 1 *à* $n-1$ **faire**
  pour *j variant de* 1 *à* $n-i$ **faire**
    si $T[j] < T[j-1]$ **alors**
      $T[j-1], T[j] \leftarrow T[j], T[j-1]$
    fin
  fin
fin

2 Diviser pour régner

2.1 Tri par fusion

Le principe du tri par fusion est le suivant : ayant à trier $2n$ termes, on trie les n premiers, puis les n derniers, et ensuite il n'y a plus qu'à fusionner les deux sous-tableaux obtenus. L'algorithme qui conduit à fusionner deux listes triées est assez simple.

Exercice 1 *Compléter la fonction suivante :*

```
def fusion(L1, L2):
    """ En entrée : deux listes triées L1 et L2, en sortie une liste formée de éléments
    de L1 et L2 fusionnés en une seule liste triée """
    L = []
    n, m = len(L1), len(L2)
    i, j = 0, 0
    for i in range(n+m):
        ....
    return L
```

```
def fusion(L1, L2):
    """ En entrée : deux listes triées L1 et L2, en sortie une liste formée de éléments
    de L1 et L2 fusionnés en une seule liste triée """
    L = []
    n, m = len(L1), len(L2)
    i, j = 0, 0
    for k in range(n+m):
        if i < n:
            if j < m and L2[j] < L1[i]:
                L.append(L2[j])
                j += 1
            else:
                L.append(L1[i])
                i += 1
        else:
            L.append(L2[j])
            j += 1
    return L
```

Ayant écrit la fonction qui fusionne deux listes triées, trier une liste quelconque devient très facile avec la récursivité :

```
def triFusion(L):
    if len(L) < 2: # ne pas oublier le test d'arrêt !
        return L
    n = len(L) // 2
    return fusion(triFusion(L[:n]), triFusion(L[n:]))
```

2.2 Tri rapide (Quicksort)

Tout le sel de l'algorithme de tri rapide provient de la fonction de partitionnement : on choisit une valeur dans le tableau qui sert de pivot, et on modifie le tableau pour que ses premiers termes soient tous inférieurs au pivot, suivis alors du pivot, puis suivent tous les termes strictement supérieurs au pivot.

Le pivot est alors à sa place définitive, et si on trie la première partie de la liste (jusqu'au pivot, non compris) et la seconde partie de la liste (depuis le pivot non compris, jusqu'à la fin)

Pour des raisons d'efficacité (on trie « en place »), on préfère souvent travailler sur une partie de la liste, entre les indices par exemple `deb` (compris) et `fin` (non compris)

Voici en pseudo-code l'algorithme de partitionnement :

Entrées : Un tableau (ou une liste) T de n éléments, et trois indices deb, fin et $pivot$ tels que
$0 \leqslant deb \leqslant pivot < fin \leqslant n$

Sorties : L'indice de la position finale du pivot

$v \leftarrow T[pivot]$
$pos \leftarrow deb$
$T[pivot] \leftarrow T[fin-1]$
pour i *de* deb *à* $fin-2$ **faire**
  si $T[i] \leqslant v$ **alors**
    $T[i], T[pos] \leftarrow T[pos], T[i]$
    $pos \leftarrow pos + 1$
  fin
fin
$T[fin-1], T[pos] \leftarrow T[pos], v$
retourner pos

3 Analyse de complexité

Avant d'étudier la complexité des algorithmes présentés, une petite recherche permet d'établir une borne inférieure qu'on ne pourra améliorer :

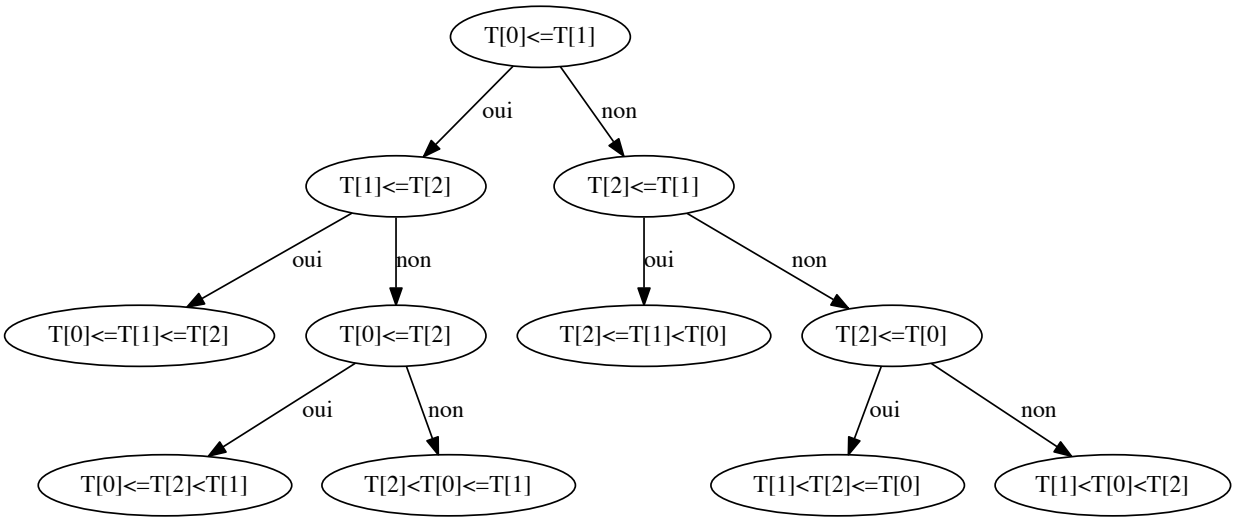
Proposition 3.1 *Soit E un ensemble de n valeurs deux à deux distinctes, et, étant donné un tableau T formé de ces n éléments, on cherche à concevoir un algorithme permettant de remettre dans l'ordre (croissant a priori, mais peu importe) les n éléments de ce tableau. Pour ce faire, on sait comparer deux éléments de T, et échanger deux éléments dans le tableau.*

Alors le nombre de comparaisons qu'il est nécessaire d'effectuer, en moyenne, est au moins de $\log_2(n!)$.

⟦L'argument qui suit, en toute rigueur, établit seulement que le nombre de comparaisons effectuées dans le pire des cas est au moins en $\log_2(n!)$. La preuve qu'il en va de même en moyenne ne sera pas donnée ici.

On visualise l'algorithme envisagé comme un arbre de décision, où à chaque noeud de l'arbre une comparaison est faite entre deux valeurs du tableau, et que selon le résultat on accède à l'un des deux fils du noeud. Un point de terminaison de l'arbre, également appelée feuille, correspond à la détermination de l'ordre des n éléments de T (et donc termine l'algorithme de tri).

Par exemple, pour un tableau de trois éléments, un arbre binaire de décision pourrait être :



La question qui se pose alors pour nous est de connaître quelle hauteur suffirait-il d'avoir pour un arbre binaire de décision permettant de connaître quel est l'ordre entre les n éléments de notre tableau ? Si l'arbre binaire est *complet* (on dit aussi quelquefois *parfait*) et que s'il est de hauteur h, alors il aura 2^h feuilles.

Ainsi, la hauteur minimale d'un arbre binaire de décision comptant f feuilles est donnée par $\lceil \log_2(f) \rceil$.

Dans notre exemple, il y a $n!$ feuilles, si bien que la hauteur recherchée est $\lceil \log_2(n!) \rceil$. ⟧

Remarque 3.2 Pour connaître l'ordre de grandeur de $\log_2(n!)$ pour de grandes valeurs de n, on utilise l'équivalent de Stirling : $n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$ ce qui conduit à une hauteur équivalente à $n \log_2(n)$.

Ainsi un algorithme de tri qui fonctionne par comparaisons ne saurait avoir une complexité meilleure, en moyenne, que $O(n \log n)$.

3.1 Algorithmes naïfs

3.1.1 Tri par sélection

Pour le tri par sélection, l'analyse est rapide, puisqu'il y a toujours $n-1+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2}$ comparaisons et $n-1$ échanges de valeurs. La seule inconnue porte sur le nombre de fois que l'affectation $i_n \leftarrow j$

est réalisée, mais cela ne change rien à la complexité qui est toujours quadratique et qui est déterminée par le nombre de comparaisons.

3.1.2 Tri par insertion

Pour le tri par insertion, et dans le pire des cas (tableau trié à l'envers), alors le nombre de comparaisons entre termes du tableau est de $1+2+\cdots+n-1=\frac{n(n-1)}{2}$ et le nombre d'affectations de valeurs du tableau est de $1+2+\cdots+n-1+n-1=\frac{(n+1)(n-1)}{2}$ ce qui conduit à une complexité, dans le pire des cas, quadratique.

Dans le meilleur des cas (tableau trié dans le bon sens) alors le nombre de comparaisons se réduit à $n-1$ et le nombre d'affectations (inutiles dans ce cas) est également de $n-1$ et on obtient une complexité linéaire en n.

On a tout à penser que, en moyenne, la complexité sera quadratique, comme dans le pire des cas, et on peut penser que le temps d'exécution sera en moyenne deux fois plus court que dans le pire des cas (ce qui ne change rien à la complexité quadratique devinée)

3.1.3 Tri à bulle

Le premier parcours de gauche à droite du tableau réalise $n-1$ comparaisons, et un nombre d'échanges qui lui est entre 0 et $n-1$.

Le nombre total de comparaisons est donc de $(n-1)+(n-2)+\cdots+1=n(n-1)/2$ et le nombre total d'échanges est, lui, inconnu, de 0 à $n(n-1)/2$.

Il n'en demeure pas moins que dans tous les cas, le pire, le meilleur ou le cas moyen la complexité est quadratique selon n le nombre de termes du tableau à trier.

3.2 Tri fusion et tri rapide

3.2.1 Tri fusion

Notons $T(n)$ le temps nécessaire pour trier n éléments par l'algorithme de tri fusion. Alors il faut pour fusionner deux listes de longueur n un temps proportionnel à n (complexité linéaire).

On peut donc considérer qu'il existe $c>0$ tel que $T(2n)=2T(n)+cn$.

En notant $u_p=T(2^p)$, on a donc pour tout p, $u_{p+1}=2u_p+c2^p$, et à partir de $T(1)=u_0=0$ il vient pour tout p, $u_p=cp2^p$.

Il vient donc, pour $n=2^p$, $T(n)=cp2^p=cn\log_2(n)$. En admettant que T est croissante, on peut alors affirmer que $T(n)=n\log n$ (ce qui, on l'a vu, est optimal)

3.2.2 Tri rapide

On devine que l'efficacité du tri rapide provient elle aussi du découpage du tableau à trier en deux sous-tableaux plus petits, mais contrairement au tri fusion qui découpe le tableau en deux tableaux de même taille (ou d'une taille qui ne diffère que d'un élément) on n'a pas de certitude qu'après le partitionnement le pivot se trouvera au centre du tableau et que les deux sous-tableaux obtenus seront de tailles similaires, sinon égales.

Dans le pire des cas, le pivot peut être le plus grand ou le plus petit terme du tableau, et dans ce cas les deux sous-tableaux obtenus sont, pour l'un, le tableau vide, et pour l'autre, un tableau de $n-1$ termes.

Si cette situation se répète à toutes les étapes de l'algorithme, alors la complexité obtenue est quadratique, car le partitionnement d'un tableau de n termes nécessite $n-1$ comparaisons, si bien que dans ce cas le nombre de comparaisons est de $n-1+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$.

Ce pire des cas peut être obtenu assez aisément : si on fait le choix comme pivot du dernier terme d'un tableau et que celui-ci s'avère être déjà classé (dans le bon sens ou le contraire, peu importe) alors le pivot se trouve à chaque fois être le plus petit ou le plus grand élément, ce qui conduit à la complexité quadratique évoquée

plus haut (de l'intérêt de choisir un pivot au hasard, et non systématiquement le premier ou le dernier élément du tableau.

La complexité en moyenne est, en revanche, bien meilleure. Notons encore $T(n)$ le nombre moyen de comparaisons qu'il faut réaliser pour trier un tableau de n termes avec l'algorithme du tri rapide.

On note qu'alors, pour ce qui est du premier pivot, les n positions qu'il peut occuper après le partitionnement sont équiprobables, et comme l'opération de partitionnement est réalisée en effectuant $n-1$ comparaisons, il vient :

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

Mais alors il vient

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1)$$

ou encore

$$nT(n) = (n+1)T(n-1) + 2(n-1)$$

puis

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

et ainsi, en notant pour tout n, $u_n = \frac{T(n)}{n+1}$, alors pour tout n, $u_n = u_{n-1} + \frac{2(n-1)}{n(n+1)} = u_{n-1} + \frac{2}{n+1} - \frac{2}{n} + \frac{2}{n+1}$ et il vient donc $u_n = u_0 + 2H_{n+1} - 4 + \frac{2}{n+1}$ où $H_n = \sum_{k=1}^{n} \frac{1}{k}$ Un équivalent bien connu de H_n est $\ln n$ et ainsi $u_n \sim \ln n$ et ainsi $T(n) \sim n \ln n$ et on retrouve une complexité optimale, en moyenne, pour le tri de n valeurs.