Algorithmes de tri

TABLE DES MATIÈRES

I INTRODUCTION

Il est naturel de se poser la question du tri des éléments d'un tableau (ou d'une liste), par exemple pour y faire une recherche dichotomique. On peut trier des valeurs numériques mais aussi d'autres types de données pourvu qu'on sache les ordonner. On triera ici des données numériques par ordre croissant.

II UN TRI LENT : LE TRI INSERTION

1 Présentation

On veut trier un tableau, par exemple $[5,3,7,12,9]$.

On va successivement insérer chaque élément parmi les premiers déjà triés, comme on le ferait pour trier un jeu de cartes. Sur l'exemple,

- on commence avec : $[5,3,7,2,12,9]$ par placer 3 à sa place parmi les deux premiers éléments :

$$[3,5,7,2,12]$$

- puis l'élément suivant 7 : $[5,3,7,2,12]$ parmi les deux premiers déjà triés :

$$[3,5,7,2,12]$$

- puis l'élément suivant 2 : $[5,3,7,2,12]$ parmi les deux premiers déjà triés :

$$[2,3,5,7,12]$$

- puis l'élément suivant 12 : $[2,3,5,7,12]$ parmi les deux premiers déjà triés :

$$[2,3,5,7,12]$$

2 Insertion

On commence par une fonction qui insère l'élément se trouvant en position `i` dans le tableau `T` parmi les i premiers supposés triés.

```
def insere(T, i):
    """insere x = T[i] parmi les i
premiers éléments supposés triés"""
    x = T[i]
    j = i
    while j > 0 and T[j - 1] > x:
        T[j] = T[j - 1]
        j -= 1
    T[j] = x
```

On note, au départ,

$$T = [x_0, \dots, x_{i-1}, x_i = x, \dots, x_n]$$

avec $x_0 < \dots < x_{i-1}$.

Invariant de sortie :

$$T = [x_0, \dots, x_j, x_j, \dots, x_{i-1}, x_{i+1}, \dots, x_n]$$

et

$$x_j > x$$

Terminaison : j est une suite d'entiers naturels strictement décroissante.
Correction : A la fin de l'algorithme,

$$T = [x_0, \dots, x_{j-1}, x, x_{j+1}, \dots, x_{i-1}, x_{i+1}, \dots, x_n]$$

avec $x_{j-1} \leqslant x < x_j$ et `T[:i+1]` est bien trié.
Complexité : Au mieux $\mathrm{O}(1)$, au pire $\mathrm{O}(i)$.

3 Le tri

On peut alors écrire la fonction de tri :

```
def tri_insertion(T):
    """trie le tableau T sur place."""
    n = len(T)
    for i in range(1, n):
        insere(T, i)
```

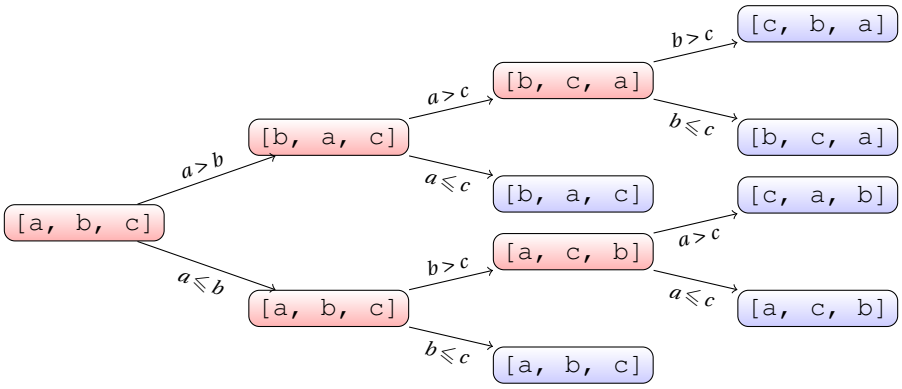
Invariant de sortie : Les i premiers éléments de T sont triés.
Correction : A la fin de l'algorithme, T est bien trié.
Complexité : Au mieux (déjà trié) $\mathrm{O}(n)$, au pire (trié dans l'ordre inverse) $\mathrm{O}(n^2)$.

III DES TRIS RAPIDES

1 Que peut-on faire au mieux ?

On peut montrer que les tris par comparaisons ont tous une complexité au mieux égale à $\mathrm{O}(n \ln n)$.

Si, par exemple, on veut trier `[a, b, c]`, on peut dessiner un arbre représentant toutes les comparaisons possibles d'éléments (arbre de décision) :



Les feuilles de l'arbre sont les $n!$ permutations des éléments.

Le nombre maximal de comparaisons pour obtenir une liste triée est la hauteur h de l'arbre.

Or le nombre de feuilles à la hauteur h d'un arbre binaire étant 2^h, on obtient, avec nos $n!$ feuilles, que $n! \leqslant 2^h$, soit $h \geqslant \log_2(n!)$: un tri par comparaison à une complexité au pire au minimum en $\mathrm{O}(\ln n!)$.

Comme $n! \to +\infty \neq 1$ et $n! \sim \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n$ (formule de Stirling),

$$\ln n! \sim n\ln n + \mathrm{o}(n\ln n) \sim n\ln n.$$

La complexité au pire d'un tri par comparaisons ne peut pas être meilleure que $\mathrm{O}(n\ln n)$.

On peut montrer que c'est encore vrai en moyenne.

Pour les deux tris que l'on va décrire, on suit le paradigme diviser pour régner : on va, pour trier le tableau (ou la liste) :

- le séparer en morceaux,
- trier (récursivement) chacun des deux morceaux,
- recombiner (fusionner) les deux morceaux.

2 Le tri fusion (merge sort)

a Le tri lui-même

Pour le tri fusion, on sépare le tableau en deux morceaux approximativement de même taille, on trie les deux morceaux, puis on les fusionne une fois triés.

Écrire le tri en lui même n'est pas compliqué, c'est la fusion qui demande un peu plus de travail :

```
def tri_fusion(L):
    "renvoie une copie triée de L"
    n = len(L)
    if n in (0, 1):
        return L
    return fusion(tri_fusion(L[:n//2]), tri_fusion(L[n//2:]))
```

Remarque

C'est facile à écrire mais ça coûte cher en espace : recopies des moitiés de tableau à chaque étape. On peut faire mieux en ajoutant en argument les indices de début et de fin de tri (voir exercices).

Une simple récurrence permet de se convaincre que la fonction termine et renvoie le bon résultat si la fusion est correcte.

b La fusion

```
def fusion(L1, L2):
    """renvoie une liste triée contenant les éléments de L1 et L2 supposés
triés."""
    n1, n2 = len(L1), len(L2)
    i1 = i2 = 0 #indices de parcours de L1 et L2
    L = []
    for k in range(n1 + n2): # il faut parcourir complètement L1 et L2
        if i1 < n1 and (i2 == n2 or L1[i1] < L2[i2]):
            L.append(L1[i1])
            i1 += 1
        else:
            L.append(L2[i2])
            i2 += 1
    return L
```

Remarque

Ici aussi, on pourrait être plus économe en mémoire en affectant dès le départ un tableau auxiliaire de même taille que `T` et en le réutilisant à chaque fusion.

Invariant de sortie : `L` contient les k plus petits éléments de `T1` et `T2`, triés.
Correction : L'algorithme renvoie bien le bon résultat vu l'invariant, les tableaux (listes python) `T1` et `T2` ayant été complètement parcourus.
Complexité : $O(n1+n2)$.

Une version récursive :

```
def fusion(L1, L2):
    """renvoie une liste triée contenant les éléments de L1 et L2 supposés
triés."""
    if L1 == []:
        return L2
    elif L2 == []:
        return L2
    else:
        if L1[-1] > L2[-1]:
            x = L1.pop()
        else:
            x = L2.pop()
        L = fusion(L1, L2)
        L.append(x)
        return x
```

c complexité

Plaçons-nous dans le cas où $n=2^p$.
La complexité temporelle de `tri_fusion` vérifie $T(n)=2T\left(\frac{n}{2}\right)+O(n)$.
Un simple lecture sur un arbre permet de se convaincre que $T(n)=O(n\ln n))$ (dans tous les cas).

Pour le démontrer, il suffit de considérer $u_p=\dfrac{T(2^p)}{2^p}$. Alors $u_p=u_{p-1}+O(1)$, donc $u_p=O(p)$. Comme $n=2^p$, $T(n)=nO(\ln n)=O(n\ln$

Dans le cas général, on a p tel que $2^p \leqslant n \leqslant 2^{p+1}$, et par croissance de la complexité, $T(n)=O\left(p2^p\right)=O(n\ln n)$.

Cette complexité $T(n)=O(n\ln n)$ est optimale en temps mais pas en espace car le tri ne s'effectue pas sur place.

3 Le tri rapide (quicksort)

a Version pas en place

Pour le tri rapide, c'est le partage du tableau qui va être plus compliqué : on choisit un élément appelé **pivot**, puis on trie séparément (récursivement) les éléments respectivement plus petits et plus grand que le pivot. Idéalement, le pivot devrait être médian pour que les deux sous-tableaux à trier soient approximativement de même taille. Comme il est trop coûteux de le rechercher, un bonne idée est de le choisir arbitrairement dans le tableau : le pire cas n'est pas obtenu de manière certaine avec un tableau donné et on peut montrer que si la complexité est la même, l'écart-type est moindre avec un tel pivot, donc on s'éloigne moins de la valeur moyenne de la complexité.

```
from random import randint

def partition(L, i):
    """renvoie un triplet (L1, L2, k) tel que L1 contient les éléments
de L < x, L2 ceux > x et k le nombre d'apparitions de x."""
    L1 = []
    L2 = []
    k = 0
    pivot = L[i]
    for elem in L:
        if elem < pivot:
```

```
            L1.append(elem)
        elif elem > pivot:
            L2.append(elem)
        else:
            k += 1
    return L1, L2, k
```

```
def tri_rapide(L):
    n = len(L)
    if n <= 1:
        return L
    else:
        i = randint(0, n - 1)
        pivot = L[i]
        L1, L2, k = partition(L, i)
        return tri_rapide(L1) + [pivot for _ in range(k)] + tri_rapide(L2)
```

Remarque

L'avantage du tri rapide est qu'il peut être exécuté en place, c'est un peu plus complexe à écrire.

On prouve l'algorithme de partition avec un invariant de boucle et une simple récurrence permet d'obtenir la terminaison et la correction de la fonction de tri.

Pour calculer la complexité, elle dépend essentiellement des tailles des listes après partitionnement :

- Au mieux, elles sont équilibrées à chaque étape, et on obtient comme pour le tri fusion $T(n) = 2T\left(\frac{n}{2}\right) + \mathrm{O}(n)$., soit une complexité en $\mathrm{O}(n \ln n)$.
- Au pire, elles sont complètement déséquilibrées (une de taille 0 et l'autre de taille $n-1$) soit $T(n) = T(n-1) + \mathrm{O}(n)$, ce qui donne $T(n) = \mathrm{O}\left(n^2\right)$.

Cependant, on peut montrer qu'en moyenne, la complexité est $\mathrm{O}(n \ln n)$.

b Version en place

```
from random import randint


def echange(T, i, j):
    T[i], T[j] = T[j], T[i]


def partition(T, debut, fin, ipivot):
    """Modifie T[debut:fin] pour que pivot = T[ipivot] s'y trouve en position
finpp tel que tous les éléments de T[debut:finpp] sont < pivot et ceux de
T[finpp:fin] sont >= pivot."""
    pivot = T[ipivot]
    if ipivot != debut: # pivot placé au début
        echange(T, ipivot, debut)

    finpp = debut
    for i in range(debut + 1, fin):
        # les éléments rencontrés de début + 1 à finpp sont < pivot
        if T[i] < pivot:
            finpp += 1
            echange(T, finpp, i)

    if finpp != debut:
        echange(T, finpp, debut) # on remet le pivot à se place, en finpp

    return finpp
```

```
def tri_rapide(T):
    "tri en place de T"

    def tri_aux(T, debut, fin):
        "tri en place de T[debut:fin]"

        if debut < fin:
            ipivot = randint(debut, fin - 1)
            ipivot = partition(T, debut, fin, ipivot)
            # pivot est à sa place ipivot, on trie récursivement
            # T[debut : ipivot] et T[ipivot + 1:fin] de tailles
            # strictement plus petites.

            tri_aux(T, debut, ipivot) # ipivot non compris !
            tri_aux(T, ipivot + 1, fin)

    tri_aux(T, 0, len(T))
```

c Version minimaliste

Et une version pas optimale, pas en place, mais sans fonction auxiliaire (merci Python !) :

```
def tri_rapide(L):
    "tri rapide pas en place de L"
    if len(L) < 2 :
        return L
    else:
        pivot = L.pop()
        avant = [element for element in L if element < pivot]
        apres = [element for element in L if element >= pivot]
        L.append(pivot) # Si on ne veut pas vider L !
        return tri_rapide(avant) + [pivot] + tri_rapide(apres)
```

IV EXERCICES

Exercices

Ex 1 – Écrire le tri sélection : le principe est de déterminer le plus petit élément, le mettre en première position, et recommencer avec les éléments restants. Prouver sa correction et calculer sa complexité. En donner une version récursive.

Ex 2 – Écrire une version récursive (toutes les fonctions !) du tri par insertion.

Ex 3 – Écrire un tri par insertion dichotomique (la recherche de l'emplacement d'un élément parmi ceux déjà triés se fait par dichotomie). Calculer sa complexité.

Ex 4 – Écrire un tri d'un tableau de n entiers dans $[\![0, N[\![$ en temps linéaire en n et N.

Ex 5 – Écrire le tri bulle. Le principe est le suivant : on parcourt le tableau d'un bout à l'autre en échangeant deux éléments successifs s'ils ne sont pas dans le bon ordre, et on recommence jusqu'à ce qu'aucun échange n'ait eu lieu. Donner un invariant de boucle (se voit facilement sur un exemple) et la complexité.

Ex 6 – Écrire le cocktail sort : c'est le même que le tri bulle, mais on parcourt alternativement dans un sens puis dans l'autre.

Ex 7 – Écrire le tri fusion en utilisant un tableau auxiliaire de taille n et rien d'autre (le reste s'exécute sur place). Il sera initialisé au départ, avec la fonction de fusion qui prendra en argument le tableau `T`, les indices `debut`, `milieu`, `fin` tels qu'il faille fusionner `T[debut:milieu]` et `T[milieu:fin]` supposés triés, et le tableau `aux` et qui remplira `aux` de `debut` à `fin` par les valeurs triées de `T[debut:fin]`.

Ex 8 – Une idée classique pour accélérer le tri rapide est de faire un tri insertion lorsque le nombre d'éléments à trier est petit (par exemple inférieur à 5). Modifier le tri rapide pour mettre en place cette idée.

Ex 9 – On considère la fonction suivante :

```python
def tri_aux(T, i, j):
    if T[i] > T[j - 1]:
        T[i], T[j - 1] = T[j - 1] = T[i]
    if j - i > 2:
        k = (j - i) // 3
        tri_aux(T, i, j - k)
        tri_aux(T, i + k, j)
        tri_aux(T, i, j - k)

def tri(T):
    tri_aux(T, 0, len(T))
```

Justifier qu'il s'agit d'un algorithme de tri. Déterminer une relation de récurrence vérifiée par sa complexité $T(n)$, puis la déterminer lorsque $n = 3^p$. Conclusion ?