Informatique en CPGE (2018-2019)
Algorithmes de tri

1 Introduction

On considère des données numériques. Trier ces données consiste à les ranger en ordre croissant ou décroissant. Une opération de tri consomme un temps de calcul important sur un ordinateur et il donc nécessaire d'étudier la complexité temporelle des différents algorithmes de tri. On peut évaluer cette complexité dans le "pire des cas", ou dans le "meilleur des cas" ou enfin "en moyenne" et ensuite utiliser l'algorithme qui convient le mieux suivant la situation.

Les algorithmes étudiés sont basés sur des comparaisons successives entre les données et la complexité d'un algorithme a le même ordre de grandeur que le nombre de comparaisons effectuées par cet algorithme.

Il y a $n!$ manières de ranger n données ($n!$ permutations). La première comparaison concerne deux des données a et b de la liste et consiste à poser la question : $a < b$? La réponse permet de diviser les $n!$ manières en deux parties égales. Les comparaisons suivantes, dans le pire des cas, permettront aussi de partager chaque partie en deux parties égales. Donc après k comparaisons, il restera $n!/2^k$ permutations à envisager. Le tri sera terminé lorsqu'il ne restera plus qu'une permutation, soit si $n!/2^k < 2$; ce qui nous donne $2^k > n!/2$, d'où $k \ln 2 > \ln(n!) - \ln 2$. Le nombre de comparaisons k est donc de l'ordre de $\ln(n!)$. Pour n grand, $n! \simeq (n/\mathrm{e})^n\sqrt{2\pi n}$, (formule de Stirling), donc $\ln(n!) \simeq n\ln(n)$.

Conclusion : dans le pire des cas, le nombre de comparaisons est de l'ordre de $n \ln n$.

2 Algorithmes de tri

2.1 Tri par insertion

En anglais : **insertion sort**.

Le tri par insertion d'un tableau à n éléments $[t_0, \ldots, t_{n-1}]$ se fait comme suit : à l'étape numéro i, (i variant de 0 à $n-2$), on suppose que les données d'indice 0 jusqu'à i sont déjà triées et on considère alors la donnée d'indice $i+1$ appelée clé ; on la compare successivement aux données précédentes, en commençant par la donnée d'indice i puis en remontant dans la liste jusqu'à trouver la bonne place de la clé, c'est-à-dire entre deux données successives (qui sont déjà triées), l'une étant plus petite et l'autre plus grande que la clé), ou bien en tout premier si la clé est plus petite que toutes les données précédentes ; au fur et à mesure de ces comparaisons, on décale d'une place vers la droite les données plus grandes que la clé ; on met la clé à la bonne place et à l'issue de cette étape, les données d'indice 0 à $i+1$ sont donc triées.

Algorithme : on se donne une liste de n données ; les indices varient de 0 à $n-1$.

Les variables entières utilisées sont les indices i, k ; la variable clé est du même type que les données de la liste.

```
Pour i variant de 0 à n − 2
    k ← i + 1    (indice de la clé)
    clé ← liste[k]
    Tant que cle < liste[k − 1] et k> 0, faire
        liste[k] ← liste[k − 1]    (décalage des données vers la droite)
        k ← k − 1
    liste[k] ← cle.
```

Exemple de programme :

```
def tri_insertion(liste):
    for i in range(len(liste)-1):
        k = i+1 # indice de la cle
        cle=liste[k]
        while cle<liste[k-1] and k>0:
            liste[k] = liste[k -1]
            k-=1
        liste[k]=cle
    return liste
```

Exemple :
liste : [**8**, 3, 4, 6, 1, 2, 5, 7]
étape 1 : [**3, 8**, 4, 6, 1, 2, 5, 7]
étape 2 : [**3, 4, 8**, 6, 1, 2, 5, 7]
étape 3 : [**3, 4, 6, 8**, 1, 2, 5, 7]
étape 4 : [**1, 3, 4, 6, 8**, 2, 5, 7]
étape 5 : [**1, 2, 3, 4, 6, 8**, 5, 7]
étape 6 : [**1, 2, 3, 4, 5, 6, 8**, 7]
étape 7 : [**1, 2, 3, 4, 5, 6, 7, 8**]

Complexité : la complexité en temps de ce tri est en $\mathcal{O}(n^2)$ dans le pire des cas (cas où la liste est triée dans l'ordre inverse et donc chaque clé doit être comparée à tous les éléments précédents) et en $\mathcal{O}(n)$ dans le meilleur des cas (cas où la liste est déjà triée avant application de l'algorithme) ; on peut vérifier que le tri est en moyenne en $\mathcal{O}(n^2)$. C'est donc un tri simple mais peu efficace.

2.2 Tri rapide

En anglais : **quicksort**.
On dispose d'une liste de n données. Le tri rapide place à l'endroit correct et de manière définitive un élément de la liste appelé pivot, construit une sous-liste "gauche" des éléments inférieurs ou égaux au pivot et une sous-liste "droite" des éléments supérieurs au pivot ; ensuite il procède à un appel récursif sur chacune des deux sous-listes. C'est le principe "diviser pour régner" (en anglais "divide and conquer").

On utilise une fonction partition qui renvoie l'indice du tableau où le pivot est rangé. Pour cela, on s'intéresse aux données qui sont dans la liste entre les indices g et d inclus ; on utilise deux variables entières notées i, j et une variable pivot du même type que les données de la liste.

Algorithme de la fonction "partition(liste, g, d)" :
pivot $\leftarrow$ liste[g]
i $\leftarrow$ g + 1
j $\leftarrow$ d
tant que i $\leq$ j, faire
    tant que liste[i] $\leq$ pivot et i $\leq$ d, faire i $\leftarrow$ i + 1
    tant que liste[j] $>$ pivot, faire j $\leftarrow$ j $-$ 1
    si i < j, alors
        échanger liste[i] et liste[j]
        i $\leftarrow$ i + 1
        j $\leftarrow$ j $-$ 1
échanger liste[g] et liste[j]
renvoyer j.

Application de la fonction partition sur un exemple :

Les données sont les valeurs qui figurent dans la seconde ligne du tableau suivant, la première ligne précise les indices des cases du tableau :

0	1	2	3	4	5	6	7	8	9
8	4	18	11	15	5	17	1	10	7

On applique la fonction partition avec g = 0 et d = 9. La variable pivot vaut liste[0] = 8.

Au premier passage dans la boucle externe, i s'arrête à 2, j reste à 9. On procède à l'échange puisque i < j, et on obtient :

0	1	2	3	4	5	6	7	8	9
8	4	7	11	15	5	17	1	10	18

Ensuite i passe 3 et j à 8.

Au deuxième passage dans la boucle, i reste à 3 et j s'arrête à 7. On procède à l'échange et on obtient :

0	1	2	3	4	5	6	7	8	9
8	4	7	1	15	5	17	11	10	18

Ensuite i passe à 4 et j à 6.

Au troisième passage dans la boucle, i reste à 4 et j s'arrête à 5. On procède à l'échange et on obtient :

0	1	2	3	4	5	6	7	8	9
8	4	7	1	5	15	17	11	10	18

Ensuite i passe à 5 et j passe à 4.

Le test « i < j ? » n'est plus vérifié. On sort de la boucle " tant que i $\leq$ j " ; on échange liste[0] avec liste[j], c'est-à-dire liste[4] ; on obtient :

0	1	2	3	4	5	6	7	8	9
5	4	7	1	8	15	17	11	10	18

La fonction retourne l'indice 4. La valeur 8 est bien à sa place définitive, avant on trouve les données inférieures ou égales à 8 et après, les données supérieures à 8.

Programme de la fonction partition :

```
def partition(liste,g,d):
    pivot=liste[g]
    i=g+1
    j=d
    while i<=j:
        while i<len(liste) and liste[i]<=pivot:
            i=i+1
        while liste[j]>pivot:
            j=j-1
        if i<j:
            liste[i],liste[j]=liste[j],liste[i]
            i=i+1
            j=j-1
    liste[g],liste[j]=liste[j],liste[g]
    return j
```

On peut maintenant écrire l'algorithme récursif du tri rapide pour un tableau dont les indices sont compris entre g et d. On appellera tri_rapide(0, n − 1) pour trier des données se trouvant entre les indices 0 et n − 1 d'un tableau à n éléments.

Procédure tri_rapide(liste, g, d) (j est une variable) :
si g < d, alors
    j ← partition(liste, g, d)
    tri_rapide(liste, g, j − 1)
    tri_rapide(liste, j + 1, d)

Sur notre exemple, il reste à appliquer le tri rapide entre les indices 0 et 3 puis entre les indices 5 et 9.
L'application de tri_rapide(0, 3) appelle partition(0, 3) qui conduit au tableau :

0	1	2	3	4	5	6	7	8	9
4	1	5	7	8	15	17	11	10	18

et renvoie la valeur 2. On applique alors tri_rapide(0,1), qui appelle partition(0,1) et conduit au tableau

0	1	2	3	4	5	6	7	8	9
1	4	5	7	8	15	17	11	10	18

et retourne la valeur 1, puis tri_rapide (0, 0) et tri_rapide (3, 3) qui ne font rien ; le tri rapide entre les indices 0 et 3 est terminé.
L'application de tri_rapide(5, 9) fait appel à partition(5, 9) qui conduit au tableau :

0	1	2	3	4	5	6	7	8	9
1	4	5	7	8	11	10	15	17	18

et retourne l'indice 7. Ensuite, tri_rapide(5, 6) puis, tri_rapide(8, 9) qui conduisent au tableau

0	1	2	3	4	5	6	7	8	9
1	4	5	7	8	10	11	15	17	18

L'algorithme est terminé.

Programme du tri rapide :

```
def tri_rapide(liste,g,d):
    if g<d:
        j=partition(liste,g,d)
        tri_rapide(liste,g,j-1)
        tri_rapide(liste,j+1,d)
    return liste
```

Complexité : la complexité en temps de la fonction partition est linéaire en la longueur de la liste considérée.

Le pire des cas est celui où le pivot vient toujours à une extrémité de la sous-liste allant de l'indice g à l'indice d pour chaque appel à la fonction partition. C'est le cas si la liste est déjà triée. On fait alors appel à la fonction partition pour des listes de longueur n, puis n − 1, puis n − 2, . . . , puis 2 ; la somme des complexités de ces fonctions donne un algorithme en $\mathcal{O}(n^2)$.

Le meilleur des cas est celui où la liste est toujours partagée en deux sous-listes de même longueur. On a alors :

— un appel à la fonction partition sur une liste de longueur n,
— deux appels à la fonction partition sur des listes de longueurs environ n/2,
— quatre appels à la fonction partition sur des listes de longueurs environ n/4,
— etc

D'où une complexité de l'ordre de n pour la liste de longueur n, une complexité totale de l'ordre de n pour les deux sous-listes de longueur n/2, une complexité totale de l'ordre de n pour les quatre sous-listes de longueur n/4, etc.

Comme la longueur est divisée par deux quand on passe de n à n/2, puis de n/2 à n/4, on a en tout une complexité de l'ordre de $n \times \ln n$.

On peut montrer que la complexité en moyenne est aussi de l'ordre de $n \times \ln n$.

Voici enfin un programme utilisant des listes supplémentaires et les possibilités de Python sans utiliser la fonction "partition" :

```
def tri_rapide(liste):
    if liste==[]:
        return liste
    liste_g=[]
    liste_d=[]
    for i in range(1,len(liste)):
        if liste[i]<=liste[0]:
            liste_g.append(liste[i])
        else:
           liste_d.append(liste[i])
    return tri_rapide(liste_g)+[liste[0]]+tri_rapide(liste_d)
```

2.3 Tri par fusion

En anglais "**merge sort**", inventé par John von Neumann en 1945.

Le tri par fusion utilise le principe "diviser pour régner". L'algorithme se décrit simplement de manière récursive :

— si la liste a 0 ou 1 élément, elle est déjà triée,
— si la liste a plus d'un élément, on la partage en deux liste et on applique le tri fusion sur chacune des deux listes
— on fusionne les résultats.

Pour fusionner deux listes triées, on compare les éléments de chacune des deux listes et on déplace le plus petit dans une nouvelle liste. Quand une des deux liste est vide, on déplace les éléments restants de la seconde liste.

Exemple :

liste1	liste2	résultat
[3,5,8,9]	[1,2,6,10]	[ ]
[3,5,8,9]	[2,6,10]	[1]
[3,5,8,9]	[6,10]	[1,2]
[5,8,9]	[6,10]	[1,2,3]
[8,9]	[6,10]	[1,2,3,5]
[8,9]	[10]	[1,2,3,5,6]
[9]	[10]	[1,2,3,5,6,8]
[ ]	[10]	[1,2,3,5,6,8,9]
[ ]	[ ]	[1,2,3,5,6,8,9,10]

Programme de la fonction fusion :

```
def fusion(liste1,liste2):
    liste=[]
    i,j=0,0
    while i<len(liste1)and j<len(liste2):
        if liste1[i]<=liste2[j]:
            liste.append(liste1[i])
            i+=1
        else:
            liste.append(liste2[j])
            j+=1
    while i<len(liste1):
        liste.append(liste1[i])
        i+=1
    while j<len(liste2):
        liste.append(liste2[j])
        j+=1
    return liste
```

Programme du tri par fusion :

```
def tri_fusion(liste):
    if len(liste)<2:
        return liste[:]
    else:
        milieu=len(liste)//2
        liste1=tri_fusion(liste[:milieu])
        liste2=tri_fusion(liste[milieu:])
        return fusion(liste1,liste2)
```

Complexité : la complexité en temps de cet algorithme est en $\mathcal{O}(n \ln n)$ dans le meilleur des cas, dans le pire des cas et donc en moyenne.

Attention, la complexité en espace est en $\mathcal{O}(n)$ puisqu'on crée une nouvelle liste de même longueur que la liste à trier et cela peut poser problème pour de très grandes listes.

3 Application

Recherche de la médiane : pour déterminer la médiane d'une liste de nombres, on commence par trier cette liste dans l'ordre croissant. Ensuite si la longueur de la liste est un nombre impair n, la médiane est l'élément d'indice (n-1)/2, sinon la médiane est la moyenne des deux éléments d'indices respectifs (n-2)/2 et n/2.

4 Le tri en Python

L'algorithme de tri utilisé en Python est appelé **timsort**, inventé par Tim Peters en 2002. C'est un algorithme dérivé du tri fusion et du tri par insertion, avec l'idée principale que dans un ensemble de données, une partie est déjà triée et qu'il faut en tirer un avantage. La complexité de cet algorithme est en $\mathcal{O}(n)$ dans le meilleur des cas, en $\mathcal{O}(n \ln n)$ dans le pire des cas et en $\mathcal{O}(n \ln n)$ en moyenne.

La méthode **sort** prend une liste en argument et modifie cette liste. La fonction **sorted** prend un objet itérable en argument et renvoie un nouvel objet trié.

```
t=[3,5,6,4]
print(sorted(t))  # affiche [3,4,5,6]
print(t)  # affiche [3,5,6,4] (t n'est pas modifié)
t.sort()
print(t)  # affiche [3,4,5,6]
```