ALGORITHMES DE TRIS

En première année vous avez dû voir que la recherche d'un élément dans un tableau était plus rapide si ce tableau était ordonné. Il est donc naturel de se demander s'il existe une procédure efficace pour trier des données. Nous allons observer différents algorithmes de tri et surtout comparer leurs complexités respectives afin de monter qu'ils ne sont pas équivalents.

On suppose que les éléments à trier sont des entiers (mais les algorithmes proposés sont valables pour n'importe quel type d'éléments muni d'un ordre total). On suppose qu'on trie des tableaux par ordre croissant. On note N le nombre d'éléments à trier.

Pour pouvoir comparer l'efficacité des algorithmes, il faut calculer leurs complexités. Il existe deux types de complexité : la complexité spatiale et la complexité temporelle. Ici on choisit de comparer les nombres d'affectations et de comparaison (opération plus longue).

1 Tri par sélection

C'est le tri dit naïf. Il consiste à recherche le minimum de la liste, et le placer en début de liste puis recommencer sur la suite du tableau.

exemple sur T=[4,12,5,8,9,6,13,3] :

étape 0 : on cherche le minimum sur T[0 :8] et on l'échange avec T[0]	4	12	5	8	9	6	13	3
étape 1 : on cherche le minimum sur ..... et on l'échange avec	3	12	5	8	9	6	13	4
étape 2 :								
...								

écrivons l'algorithme :

```
def imin(T,a,b): #fonction qui renvoie la position de la valeur minimale de T
#entre les positions a et b
    imin=a
    N=len(T)
    for i in range(a+1,b):
        if T[i] < T[imin]:
            imin = i
    return imin

def triselection(T):
    N=len(T)
    for i in range(N-1):
        j=imin(T,i,N)
        T[i],T[j] = T[j],T[i]
    return T
```

Complexités :

Quelle que soit l'étape, on n'a pas besoin d'un autre tableau donc le coût en mémoire de ce tri est constant.

Concernant sa complexité temporelle :
• en comparaisons : pour rechercher le premier minimum on effectue N-1 comparaisons, pour le second N-2, ... le nombre total de comparaisons est de $(N-1)+(N-2)+...+1=\frac{N(N-1)}{2}$ donc en $O(N^2)$
• en affectations : pour chaque échange, on effectue 3 affectations donc 3N, pour chaque recherche de minimum, au maximum N-1, puis N-2, ... donc on retrouve une complexité en $O(N^2)$.

Quel est le « pire » tableau à trier avec cette méthode ?

2 Tri bulle (en TD)

3 Tri par insertion

C'est le tri du joueur de cartes.
Il consiste à insérer successivement chaque élément T[i] dans la portion du tableau T[0 :i] déjà triée. (En Python, T[0 :i] est la portion du tableau T d'indice 0 à l'indice i-1).

exemple sur T=[4,12,5,8,9,6,13,3]. Le tableau T[0 :1]=[4] est déjà trié.

étape 1 : on cherche placer T[1]=12 dans T[0 :1]=[4]	4	12	5	8	9	6	13	3
étape 2 : on cherche placer T[2]=5 dans T[0 :2]=[4,12]	4	5	12	8	9	6	13	3
étape 3 :								
...								

écrivons l'algorithme :

```
def triinsertion(T):
    n=len(T)
    for i in range (1,n):
        valeur=T[i] #valeur a inserer
        j=i-1
        while (j>=0) and (T[j]>valeur):
            T[j+1]=T[j]
            j=j-1
        T[j+1]=valeur
```

Complexités :

Quel est le « pire » tableau à trier avec cette méthode ? le meilleur ?

Une fois de plus, le coût en mémoire de ce tri est constant.

Evaluons sa complexité temporelle :
pour le premier placement, on effectue 2 comparaisons ;
pour le second placement, on effectue au mieux 2 comparaisons (si le tableau est déjà trié), au pire 4 comparaisons
...
pour le dernier placement, on effectue au mieux 2 comparaisons (si le tableau est déjà trié), au pire 2(N-1) comparaisons

Le nombre total de comparaisons est donc de :
au mieux de $2(N-1)$ comparaisons donc $O(N)$ temps linéaire si le tableau est trié.
au pire : $2(1+2+...+N-1) = 2\frac{N(N-1)}{2}$ donc en $O(N^2)$, temps quadratique si le tableau est rangé par ordre décroissant.
en moyenne $N^2/2$.

Le nombre d'affectations peut être un peu optimisé. Il sera au mieux égal au nombre de comparaisons donc quadratique.

objectif fin de première heure Question : Pour un tableau k presque trié, c'est à dire que au plus k éléments ne sont pas à leur place ou que les éléments mal rangés sont à une distance au plus k de leur place finale, évaluez le nombre de comparaisons maximal de l'algorithme.

4 Tri Fusion (TD)

Comme le tri rapide, le tri fusion applique le principe du « diviser pour mieux régner ».
Il partage arbitrairement les éléments à trier en deux sous ensembles de même taille (sans chercher à les comparer) afin d'éviter le pire cas du tri rapide où les deux sous ensembles sont de tailles disproportionnées.
Une fois les deux parties triées récursivement, il les fusionne.

exemple sur T=[4,12,8,5,9,6,13,3].

pour trier T[0 :8] on trie T[0 :4] et T[4 :8]	4	12	8	5	9	6	13	3
pour trier T[0 :4] on trie T[0 :2] et T[2 :4]	4	12	8	5				
pour trier T[0 :2] on trie T[0 :1] et T[1 :2]	4	12						
on fusionne T[0 :1] et T[1 :2] triés								
on fusionne T[0 :2] et T[2 :4] triés								
on fusionne T[0 :4] et T[4 :8] triés								

Implémentons cette méthode de tri sous Python :

Nous allons utiliser deux fonctions :
• une fonction fusion qui prend en entrée deux tableaux T1 et T2 supposés triés et renvoie un tableau contenant les mêmes éléments que T1 et T2 rangés par ordre croissant.
• la fonction récursive de tri qui si le tableau contient plus d'un éléments le subdivise en deux sous tableaux de tailles équivalentes et les trie.

```
def fusion(T1,T2):
    n1=len(T1)
    n2=len(T2)
    T=[]
    i=0
    j=0
    while(i<n1 or j<n2):
        if i==n1:
            T.append(T2[j])
            j=j+1
        elif j==n2:
            T.append(T1[i])
            i=i+1
        elif T1[i] < T2[j]:
            T.append(T1[i])
            i=i+1
        else:
            T.append(T2[j])
            j=j+1
    return T

def trirec(T):
    if len(T)<=1:
        return T
    else:
        m=len(T)//2
        T=fusion(trirec(T[0:m]),trirec(T[m:len(T)]))
    return T
```

Etudions la complexité :
Soit $C(N)$ le nombre de comparaisons effectuées par la fonction tri sur un tableau de taille N.
$C(N) = 2C(\frac{N}{2}) + f(N)$ où $f(N)$ désigne le nombre de comparaisons effectuées par fusion.
Cas le plus favorable : tous les éléments d'un des tableaux sont inférieurs à tous les éléments de l'autre sous tableau. On n'effectue vraiment que $f(N) \approx \frac{N}{2}$ comparaisons. On a alors $C(N) \approx \frac{1}{2} N \log_2(N)$.
Cas le moins favorable : il faut examiner tous les éléments, soit $f(N) \approx N - 1$ comparaisons. On a alors $C(N) \approx N \log_2(N)$.

Concernant le nombre $A(N)$ d'affectations, $A(N) = 2A(N/2) + 2N$ dans tous les cas donc $A(N) \approx 2N\log_2(N)$.

Conclusion : dans tous les cas, la complexité en comparaisons et en affectations est en $N\log_2(N)$, qui est la complexité optimale du tri « a priori ».

remarque : Pourquoi la complexité en comparaisons d'un algorithme de tri est elle au mieux en $N\log_2(N)$?
On peut visualiser notre problème de tri comme un arbre binaire, chaque noeud représentant une comparaison à effectuer (on part à gauche si la comparaison est positive et à droite sinon).
S'il y a N éléments, il y a N ! comparaisons possibles, donc notre arbre a N ! feuilles.
Une propriété sur les arbres affirme que la hauteur/profondeur d'un arbre à N ! feuilles est au moins de $\log_2 N!$.

5 Tri Rapide

principe :

Le tri rapide s'appuie sur le principe « diviser pour mieux régner ».
On choisit un élément appelé pivot. On sépare les éléments à trier en deux sous ensembles : le premier constitué de tous les éléments plus petits que le pivot, le second constitué des éléments plus grands que le pivot. Puis on trie récursivement chaque sous ensemble. Le tri rapide d'un tableau s'effectue « en place ».

exemple sur T=[4,12,5,8,9,6,13,3] par la méthode du Tri rapide

L'un des enjeux est le choix du pivot : idéalement il est une médiane des données (il sépare ainsi en deux sous ensembles de même taille). Malheureusement, il est impossible de le savoir à l'avance puisque le tableau n'est pas trié. On le choisit donc au hasard !

Implémentons cette méthode de tri sous Python :

Une fois le pivot choisi (la fonction randint(a,b) de la bibliothèque random renvoie un entier compris entre a et b inclus), on a besoin de deux fonctions :
• une fonction de partition qui organise les éléments du tableau à trier autour du pivot et donne la position finale du pivot. Cette fonction nécessitant de procéder à de nombreux échanges de valeurs du tableau, il peut être utile de créer une fonction echange(T,i,j) permettant d'intervertir les contenus des cases i et j du tableau T.
• une fonction de tri qui trie récursivement les deux portions de tableau à gauche et à droite du pivot (d'où l'importance de connaitre sa position).

```
from random import randint

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

def partition(T,g,d):

    #choix du pivot
    m=randint(g,d-1)
    p=T[m]
    #on place le pivot à gauche au depart
    echange(T,g,m)
    m=g

    for i in range (g+1,d): #on considere tous les elements
        if T[i]<p:
            echange(T,i,m+1)
            m=m+1

    echange(T,g,m)#on met le pivot a sa place
    return m

def tri(T,g,d):

    if g<d-1: #si le tableau contient au moins 2 elements
        m=partition(T,g,d)
        tri(T,g,m)
        tri(T,m+1,d)


T=[5,6,2,4,8,3,1,7,12,5,1,3,6,4,2,8,9]
tri(T,0,len(T))
print(T)
```

Etude de la complexité en comparaisons :

On note $C(N)$ le nombre de comparaisons pour trier un tableau de taille N.
L'essentiel des comparaisons vient de la fonction partition qui effectue $d-1-g$ comparaisons à chaque appel.
C'est-à-dire au départ N-1 comparaisons.
Ensuite si le pivot a été bien choisi, il a coupé les données en deux sous ensembles de taille équivalente à $\frac{N}{2}$.
Si le pivot a été mal choisi (si c'est le minimum ou le maximum), on a coupé en un sous ensemble de taille 0 (pas d'élément plus petit ou plus grand que le pivot) et un autre de taille N-1.

• dans le meilleur cas :

$C(N) = N-1+2C(N/2)$ or $C(N/2) = N/2-1+2C(N/4)$
donc $C(N) \approx 2N+4C(N/4)$
en continuant, $C(N) \approx nN+2^nC(N/2^n)$ jusqu'à ce que $N/2^n$ fasse 1 ou moins. A ce moment là, le sous tableau est déjà trié car il ne contient qu'un élément maximum (complexité nulle).
$\frac{N}{2^n} \geqslant 1 \Leftrightarrow N \geqslant 2^n \Leftrightarrow \ln(N) \geqslant n\ln(2) \Leftrightarrow n \leqslant \log_2(N)$

donc $C(N) \approx N\log_2(N)$.

• dans le pire cas :

$C(N) = N-1+C(N-1) = N-1+N-2+C(N-2)...$
donc $C(N) = \frac{N(N-1)}{2} \approx \frac{N^2}{2}$
La complexité est quadratique.

donc $C(N) \approx N\log_2(N)$.

• en moyenne : On admet que la complexité moyenne est en $2N\log_2(N)$.

Etudions la comparaison en affectations :

dans la fonction Tri elle même il y a au plus une affectation. L'essentiel des affectations se produit dans la

fonction Partition.

Les deux premières affectations peuvent être supprimées afin d'optimiser le code en terme d'affectation mais on ne peut éviter :
• de faire le premier échange pour mettre le pivot à gauche (2 affectations)
• de procéder à un échange chaque fois qu'une valeur plus grande que le pivot est trouvée, en incrémentant m.
• de placer le pivot à sa place à la fin s'il n'y est pas.

Il y a autant d'échanges que d'incrémentations de m : soit entre 1 et $(d-g)$.

• dans le meilleur cas :
Le meilleur cas en terme d'affectation est donc quand il n'y a pas d'incrémentation de m, c'est à dire que le pivot reste tout à gauche. Sauf que dans ce cas, cela signifie que nous avons pris comme pivot le minimum et donc tres mal partitionné notre tableau (coupé en 0 éléments inférieurs et N-1 éléments supérieurs). Il y a alors seulement l'echange initial soit deux affectations. Mais il va falloir le faire N fois. La complexité est donc linéaire en affectations (de l'ordre de 2N) mais elle est alors quadratique en terme de comparaisons.

• dans le cas le plus favorable pour les comparaisons :
Le pivot se situe *au milieu*. Il y a donc environ $\frac{d-g}{2}$ échanges soit $d-g$ affectations. Au départ $d-g=N$, puis on a deux sous tableaux de taille $\frac{N}{2}$... jusqu'à ce que $\frac{N}{2^n}$ soit inférieur à 1. On retrouve une complexité en $N\log_2(N)$.

• dans le pire cas :
Le pire cas pour les affectations correspond aux cas où le pivot est situé à l'extrême droite (il est la valeur maximum) car toutes les valeurs précédentes auront été échangées soit $d-g$ échanges. De plus, la partition est alors peu efficace (N-1 valeurs inférieures au pivot et 0 supérieure). On a alors une complexité quadratique en affectation et en comparaisons.

• en moyenne :
on admet que la complexité moyenne en terme d'affectation est en $2N\log_2(N)$.

Résumons :

opérations	meilleur cas pour les affectations	meilleur cas pour les comparaisons	moyenne	pire cas
comparaisons	N^2	$N\log_2(N)$	$2N\log_2(N)$	$\frac{N^2}{2}$
affectations	$2N$	$N\log_2(N)$	$2N\log_2(N)$	N^2

remarques :
• un cas peut être le pire ou le meilleur, tout dépend du point de vue. Il est donc important de faire attention aux priorités souhaitées.
• Dans les faits, une comparaison est une opération plus complexe et donc plus chronophage pour l'ordinateur.
• Un autre facteur limitant sera également le nombre d'appel à une fonction récursive (limitée à priori à 1000 en Python).

Tableau des comparaisons en terme de temps (en s) :

N	1000	10 000	100 000
Tri par selection	0.079	5.82	565.94
Tri Bulle	0.16	14.51	1476
Tri rapide	0.013	0.09	excede le nb de recursions autorisées
Tri fusion	0.011	0.05	0.69

idées de TP :
tri bulle estimation de la complexité

tri par insertion linéaire sur les tableaux presque triés ?
optimiser le tri fusion : lorsque tous les éléments du tableau de gauche sont inférieurs au tableau de droite !