Lycée Polyvalent Régional Dumont d'Urville
CLASSES PREPARATOIRES PTSI / PT
Informatique Pour Tous

ALGORITHMES DE TRI

Partie A : INTRODUCTION

OBJECTIFS

- Savoir rédiger des algorithmes de tris en un pseudo-code ou en code Python
- Savoir calculer la complexité de quelques algorithmes de tris

Le tri est un problème fondamental de l'algorithmique. En général, son but est de préparer des données pour permettre d'y rechercher rapidement une information.

IDEE

Par exemple, dans des données triées, il est facile de :

- chercher rapidement si un élément est présent ou non (recherche dichotomique) ;
- trouver les doublons ;
- trouver une information associée à une donnée (par exemple la recherche d'un numéro dans l'annuaire) ;
- chercher le $k^{ième}$ plus grand élément d'un ensemble.

Partie B : TRI PAR INSERTION

DEF.

Soit une liste L de n éléments. Le tri par insertion consiste à insérer à la $ième$ itération le $ième$ élément à la bonne place par ordre croissant.

B1 Algorithme du tri par insertion

```
1   Pour i allant de 2 à n
2       Insérer l'élément L[i] à sa place dans L[0,i-1]
3   Fin pour
```

Question 1 : Décrire les étapes successives du tri par insertion de $L = [9,6,1,4,8]$:

$i = 2$: On prend 6 et on le place juste avant 9 $(6 < 9)$
$L = [6,9,1,4,8]$
$i = 3$: On prend 1 et on le place juste avant 6 $(1 < 6)$
$L = [1,6,9,4,8]$
$i = 4$: On prend 1 et on le place juste avant 6 $(4 < 6)$
$L = [1,4,6,9,8]$
$i = 5$: On prend 1 et on le place juste avant 6 $(4 < 6)$
$L = [1,4,6,8,9]$

Question 2 : Ecrire en Python la liste d'instructions qui permet de placer un élément à sa place dans une liste de n éléments déjà triés :

```
1   c=0
2   while L[i] > L[c]:
3       c += 1
4   L.insert(c,L.pop(i))
```

pierre.breteau@ac-normandie.fr francois.jouaux@ac-normandie.fr francois.lemeux@ac-normandie.fr

Question 3 : Ecrire en Python, la fonction `TriInsertion(L)` qui prend comme argument une liste `L` de n nombres entiers et qui renvoie la liste `L` triée.

```
1  def TriInsertion(L):
2      n = len(L)
3      for i in range(1,n):
4          c = 0
5          while L[i] > L[c]:
6              c += 1
7          L.insert(c,L.pop(i))
8      return L
```

B2 Complexité du tri par insertion

Dans le meilleur des cas le tableau de valeur est trié à l'envers, de sorte qu'à chaque itération il n'y ait qu'une comparaison à faire. Comme on commence au deuxième élément, il y a donc $n-1$ comparaisons à effectuer. **La complexité est donc linéaire** : $C(n) = O(n)$.

Dans le pire des cas le tableau est déjà trié. Il y a donc une comparaison à effectuer à la première étape, puis 2, puis 3, …, puis n-1. Soit un nombre total de comparaisons : $1 + 2 + 3 + 4 + \cdots + (n-1) = \frac{n.(n-1)}{2}$. **La complexité est donc quadratique** : $C(n) = O(n^2)$.

NOTE On peut aussi montrer que la complexité en moyenne est de classe quadratique lorsque les permutations sont équiprobables. L'efficacité du tri par insertion est excellente lorsque le tableau est déjà trié ou « presque trié » ($C(n) = O(n)$), à condition de prendre la bonne méthode.

DEF. On montre que le nombre de comparaisons est en moyenne de $\frac{n^2}{4}$.

On souhaite vérifier cette propriété :

Question 4 : Modifier la fonction `TriInsertion(L)` pour qu'elle renvoie le nombre de comparaisons d'entiers réalisées.

Question 5 : Définir un protocole pour évaluer le nombre de comparaisons en moyenne. Ecrire un script le mettant en œuvre.

NOTE On pourra construire des listes aléatoires avec la fonction `randint` du module `random`.

Partie C : TRI FUSION

IDEE

La méthode de tri fusion pour un tableau de données est la suivante :
- On coupe en deux parties à peu près égales les données à trier.
- On trie les données de chaque partie par la méthode de tri fusion.
- On fusionne les deux parties en interclassant les données.

Il s'agit donc d'une méthode basée sur le principe de "diviser pour régner".

C1 Principe : tri d'un jeu de cartes

Soit un paquet de n cartes :
- On divise le paquet en deux parties sensiblement égales
- Chacun de ces paquets est de nouveau divisé en deux parties
- Quand chacun des paquets ne contient plus qu'une seule carte, on fusionne les paquets deux à deux en ordonnant correctement les cartes

La méthode est donc récursive. La seule difficulté consiste en la fusion des deux paquets triés.

C2 Application au tri de listes

Soient deux listes $L1$ et $L2$ préalablement triées par ordre croissant. Le principe de la fusion consiste à comparer le premier élément de chaque liste et à ne conserver que le plus petit dans la liste fusionnée L. On recommence jusqu'à ce que l'une des deux listes soient vides. On termine par ajouter le contenu de la liste non vide à L.

C2.1 Algorithme de la fusion

Soient deux listes `L1` et `L2` préalablement triées par ordre croissant que l'on souhaite fusionner dans la liste `L`.

```
Tant qu'une des deux listes n'est pas vide
     Si le premier élément de L1 est plus petit que le premier élément de L2.
          Retirer le premier élément de L1 et l'ajouter à L
          Sinon, retirer le premier élément de L2 et l'ajouter à L
Si L1 n'est pas vide
     Elle est ajoutée à L
     Sinon L2 est ajoutée à L
Renvoyer L
```

C2.2 Script Python de la fusion

Question 6 : Ecrire en Python le script de la fonction `fusion(L1,L2)`.

```
def fusion(L1,L2):
    L=[]
    while len(L1) != 0 and len(L2) != 0:
        if L1[0] < L2[0]:
            L.append(L1.pop(0))
        else:
            L.append(L2.pop(0))
    if len(L1) != 0:
        L += L1
    else:
        L += L2
    return L
```

C2.3 **Réalisation du tri**

```
1  def TriFusion(L):
2      if len(L)<=1:
3          return L
4      m=len(L)//2
5      L1=TriFusion(L[m:])
6      L2=TriFusion(L[:m])
7      L=fusion(L1,L2)
8      return L
```

C3 Complexité du tri fusion

C3.1 **Complexité**

Pour deux listes de longueur $n1$ et $n2$ le nombre de comparaisons à réaliser pour l'opération de fusion est au pire $n1 + n2 - 1$. Soit une complexité linéaire pour l'opération de fusion.

Soit n la longueur de la liste L à trier et $C(n)$ la complexité de l'algorithme de tri fusion pour la trier. Le fonctionnement de l'algorithme permet d'écrire une relation de récurrence de la forme $C(n) = 2.C\left(\frac{n}{2}\right) + n - 1$

DEF. On peut montrer (et on admettra) que dans ce cas, la complexité temporelle est en :

$$C(n) = O(n.\ln(n))$$

L'algorithme du tri fusion est optimal du point de vue de la complexité temporelle.

NOTE En revanche, contrairement au tri par insertion, chaque appel de la fonction `fusion(L1,L2)` créé temporairement une liste de taille `len(L1)+len(L2)`.

Dans le cas où la mémoire est une ressource limitée, on évitera l'utilisation de cet algorithme.

C3.2 **Comparaison avec le tri par insertion**

Question 7 : Modifier les codes des deux fonctions `TriInsertion(L)` et `TriFusion(L)` de manière à ce qu'ils renvoient le temps mis pour trier la liste.

Question 8 : Ecrire un script qui permet de comparer le temps moyen mis par chaque algorithme pour trier une liste de 10 000 entiers.

Partie D : TRI RAPIDE (QUICKSORT)

IDEE Le tri rapide a été inventé en 1960 par Tony Hoare.
C'est une méthode récursive qui s'appuie sur le principe "diviser pour régner" :

- on choisit un pivot (aléatoirement ou pas) parmi tous les éléments à trier,
- on réorganise tous les éléments de sorte que toue les éléments inférieurs soient à gauche du pivot et les éléments supérieurs à droite
- trier chaque sous-ensemble (droite et gauche) en y choisissant un nouveau pivot
- lorsque chaque sous-ensemble ne contient qu'un élément, la liste est triée

Question 9 : Application au tri de la liste $L = [5,8,7,3,2,9,6,4,1,8]$

5	8	7	3	2	9	6	4	1	8
3	2	4	1	5	8	7	9	6	8

3	2	4	1		5		8	7	9	6	8
2	1	3	4		5		7	6	8	8	9

2	1		3		4		5		7	6		8		8	9
1	2		3		4		5		6	7		8		8	9

1		2		3		4		5		6		7		8		8		9

Il reste à concaténer toutes les listes ne comprenant qu'un seul élément :

1	2	3	4	5	6	7	8	8	9

D1 Partition

D1.1 **Algorithme de la partition**

On suppose qu'on choisit le premier élément à trier comme pivot.

```
On choisit le premier élément de L comme pivot
On initialise le curseur à c=1 (premier élément après le pivot)
Pour tous les éléments qui suivent le pivot
     Si l'élément d'indice i est inférieur au pivot
          On l'échange avec l'élément d'indice c et on fait c=c+1
          Sinon on ne fait rien
On échange le pivot avec le terme d'indice c-1
```

NOTE A chaque fois qu'un terme plus petit que le pivot est trouvé, le compteur est incrémenté de 1. A la fin de la partition, le curseur c contient l'indice du premier terme supérieur au pivot, c'est pourquoi on place le pivot à la position $c-1$.

NOTE L'algorithme de tri rapide étant récursif, la partition devra être réalisable sur une sous-liste de la liste L à trier. On définit cette sous liste L' par les indices g et d :

$$L' = L[g:d]$$

NOTE En Python, la liste $L[g:d]$ correspond à la sous-liste de L contenant les éléments d'indice g inclus à d exclu. Le terme $L[d]$ ne fait donc pas partie de $L[g:d]$.

D1.2 **Script Python de la partition**

Question 10 : Ecrire le code de la fonction `partition(L,g,d)` qui prend en argument une liste d'entiers L et deux indices g et d ($g < d$) qui définissent une sous liste $L[g:d]$ à partitionner. Cette fonction renverra la position du pivot dans la liste L après partition.

```
def partition(L,g,d):
    c = 1
    pivot = L[g]
    for i in range(1, len(L[g:d])):
        if L[g+i] < pivot:
            L[g+i],L[g+c] = L[g+c],L[g+i]
            c += 1
    L[g+c-1],L[g] = L[g],L[g+c-1]
    return g+c-1
```

D2 Tri

Question 11 : Etablir le code Python d'une fonction `TriRapideRec(L,g,d)` qui prend en argument une liste d'entiers L et les deux indices g et d qui définissent la sous-liste à trier. Cette fonction doit effectuer le tri rapide de $L[g:d]$.

```
def TriRapideRec(L,d,g):
    if len(L[g:d])<=1:
        return L[g:d]
    p=partition(L,g,d)
    return TriRapideRec(L,g,p)+[L[p]]+TriRapideRec(L,p+1,d)


```

Question 12 : Ecrire le code Python de la fonction `TriRapide(L)` qui prend en argument une liste d'entiers non triés et qui la renvoie triée.

```
1  def TriRapide(L):
2      return TriRapideRec(L,0,len(L)-1)
3
4
5
```

D3 Complexité du tri rapide

Supposons une liste de longueur n. Pour la partitionner il faut réaliser $n-1$ comparaisons avec le pivot.

- **Dans le meilleur des cas**, la partition génère deux listes de même longueur $\frac{n-1}{2}$, la relation de récurrence sur la complexité est alors

$$C(n) = 2.C\left(\frac{n-1}{2}\right) + n - 1$$

La complexité est alors quasi linéaire en $O(n.\ln(n))$.

- Dans le pire des cas, la liste est déjà triée. On réalise $n-1$ comparaisons à la première itération, puis $n-2$, puis $n-3$ soit un total de :

$$C(n) = \frac{n.(n-1)}{2}$$

La complexité est alors quadratique.

IDEE

On est souvent confronté à des listes déjà triées ou partiellement triées, la meilleure des solutions consisterait à prendre une valeur "moyenne" des éléments de la liste.
Un bon compromis consiste à choisir aléatoirement le pivot.

Pour cela, il suffit de modifier le début de la fonction partition :

```
1  import random as rd
2
3  def partition(L,g,d):
4      i=rd.randint(g,d-1) #tirage d'un entier entre g et d exclu
5      L[g],L[i]=L[i],L[g] #interversion du pivot et du premier élément
6      pivot=L[g] #initialisation du curseur
7
8
9
10
11
12
13
14
15
```

D4 Comparaison aux autres méthodes

Question 13 : Modifier le code de la fonction `TriRapide` pour qu'elle renvoie également le temps mis pour trier la liste.

Question 14 : Ecrire un script qui permet de comparer le temps moyen mis par chaque algorithme pour trier une liste de 10 000 entiers.