Informatique

Informatique Semestre 3

Première partie

Complexité des Algorithmes et tri

Définition : complexité. La complexité est le n'ombre d'exécution, d'instruction élémentaires nécessaires pour obtenir la réponse (exacte) au pire cas en fonction de la taille d'entrée.

Algorithme type.

— Entrée : Une suite (une liste) d'objets comparable (entiers, réels, mots,...)

— Sortie : Une suite (une liste) des mêmes objets triés

Définition : opération élémentaires. Une opération élémentaire est une comparaison de deux objets, une addition de deux scalaires etc...

```
def triNaif(L):
    []=Trie
    while len(L)>0:
        m=L[0]
        for e in L:
            if e>m:
                m=e
        Trie=Trie+[m]
    return Trie
```

Calculons la complexité :

— La création de liste et le return ont une complexité constante.

— La boucle while se répète n fois.

— Opération dans la boucle while se répète n fois.

— La boucle for a un nombre contant d'opérations par éléments de L c'est donc une complexité linéaire par rapport à $len(L)$.

Ainsi on a $C = c_1 + c_2 n + c_3(n + (n-1) + ... + 3 + 2 + 1) = c_1 + c_2' n + c_3' n^2 = O(n^2)$.

1 Outils de comparaison.

$T(n) = O(f(n))$. On dit que T est O de f si et seulement si

$$\lim_{n \to +\infty} \frac{T(n)}{f(n)} < \infty$$

c'est à dire $\exists N, \exists c > 0, T(n) \leq cf(n) \forall n \geq N$.

$T(n) = \omega(f(n))$. On dit que T est Ω de f si et seulement si

$$T(n) = \Omega(f(n)) \Leftrightarrow \lim_{n \to \infty} \frac{T(n)}{f(n)} > 0$$

$T(n) = \theta(f(n))$. On dit que T est θ de f si et seulement si

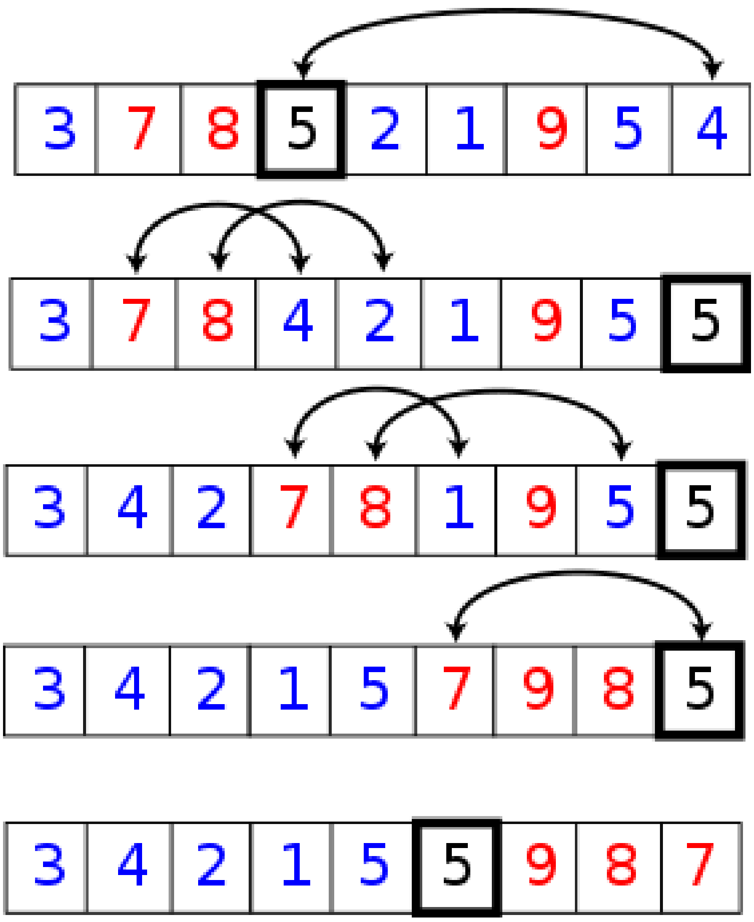
$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$$

On remarque que T est en θ de f si et seulement si T est en Ω et en O de f.

Rappel. On rappelle que le nombre d'ordres possibles pour une liste de n élément vaut $n!$.

2 Quick Sort

Principe. .




```
def quickSort(L, début=0, fin=None):
    if fin==None:
        fin=len(L)-1
    if début>=fin:
        return
    pivot=L[randint(début, fin)]
```

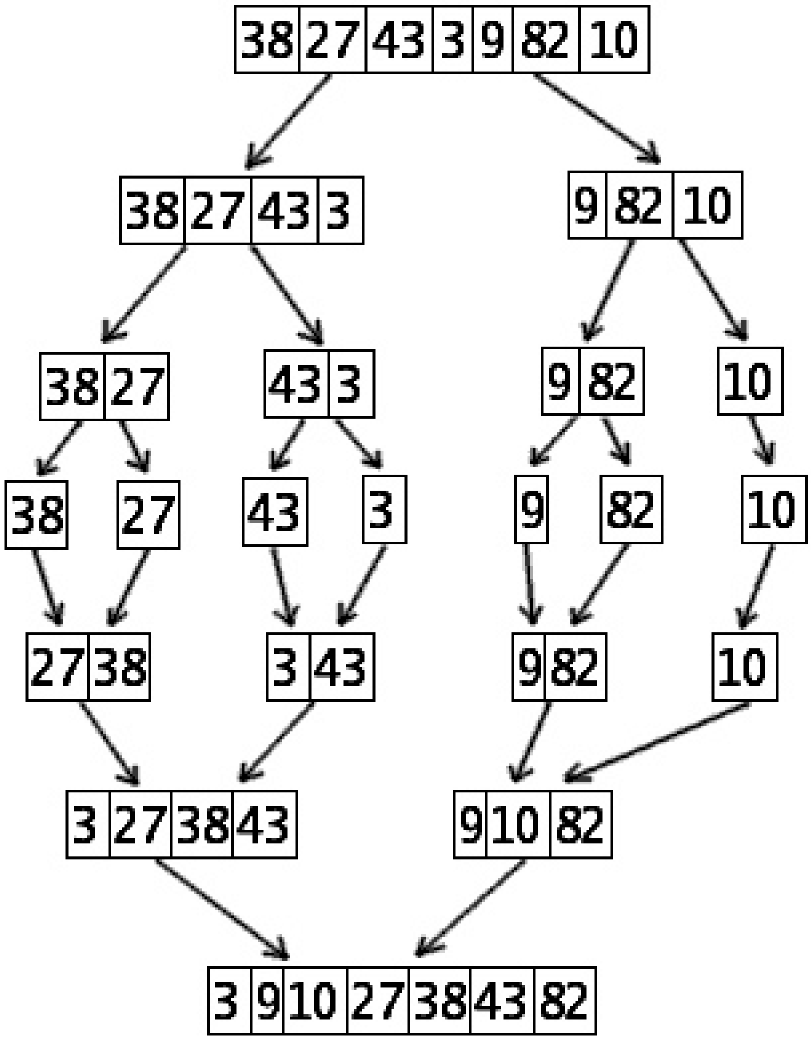
```
seuil=partitionner(L,début,fin  ,pivot)
quickSort(L,début,seuil-1)
quickSort(L,seuil,fin)
```

```
def partitionner(L,début,fin,pivot):
    i=début
    j=fin
    while i<j:
        while L[i]<pivot:
            i=i+1
        while L[j]>pivot:
            j=j-1
        if i<j:
            a=L[i]
            L[i]=L[j]
            L[j]=a
    return i
```

Réalisation.

3 Merge sort

Principe. .




```
def fusion(L1,L2):
    i=0
    j=0
    L=[]
    while i<len(L1) and j<len(L2):
        if L1[i]<L2[j]:
            L.append(L1[i])
            i=i+1
        else:
            L.append(L2[j])
            j=j+1
    while len(L1)>i:
        L.append(L1[i])
        i=i+1
    while len(L2)>j:
        L.append(L2[j])
        j=j+1
```

```
    return L

def casser(L):
    L_c=[]
    i=0
    while len(L)>0:
        x=L.pop(i)
        L_c=L_c+[[x]]
    return L_c

def mergeSort(L):
    cpt=0
    L=casser(L)
    while len(L)>2:
        x=L.pop(0)
        y=L.pop(1)
        L.append(fusion(x,y))
    if len(L)==2:
        x=L.pop(0)
        y=L.pop(0)
        return fusion(x,y)
```

Analyse de la complexité. On cherche la complexité de mergesort définie par la complexité à trier 2 listes à $n/2$ éléments et la complexité à fusionner 2 éléments on a donc :

$$T(n) = 2T(n/2) + c.n = 2(2T(n/4) + c(n/2)) + cn = 4T(n/4) + 2cn$$

Par récurrence on obtient

$$T(2^k) = 2^k T(1) + kc2^k = nT(1) + cnlog_2(n)$$

Donc $T(n) = O(nlog_2(n))$.

4 Master Theorem

Quelle est la complexité d'une procédure définie ainsi :

```
p(entrée de taille n) :
        si n<n_0:
                trouver la réponse directement
        sinon
```

```
décomposer l'entrée en a objets, chacun de taille n/b
appeler p() pour chacun de ces objets
combiner les résultats partiels pour trouver la réponse à retourner
```

La complexité d'un tel algorithme est la somme de la complexité de l'appel de p récursivement et de combiner les résultats partiels.

Master Theorem. Soit $c_0 = log_b(a)$

— Si $g(n) = O(n^c)$ avec $c < c_0$ alors $T(n) = \theta(n^{c_0})$
— Si $g(n) = \Omega(n^c)$ avec $c > c_0$, alors $T(n) = \theta(g(n))$
— Sinon : $g(n) = \theta(n^{c_0}, log^k(n))$ et
  — Si $k > -1$ alors $T(n) = \theta(n^{c_0}, log^{k+1}(n))$
  — Si $k = -1$ alors $T(n) = \theta(n^{c_0}, log(n))$
  — Si $k < -1$ alors $T(n) = \theta(n^{c_0})$

Deuxième partie

Les graphes

Généralités. Un graphe $G = (V, E)$ où V est le nombre de sommet et E le nombre d'arête, est tel que chaque arrête relie deux sommets. Une arrête reliant un sommet à lui même est une boucle. Si deux arrêtes sont parallèles alors se sont des arêtes reliant les mêmes sommets.
Si e relier u et v, on dit $e = uv$ est incidente à u et v, u et v sont voisins ou adjacents, u et v sont les extrémités de e.

Définition : degré. Le degré d'un sommet u est le nombre d'incidences auxquelles ils participent. Dans un graphe simple c'est différentes arrêtes incidentes, différents voisins.

Définition : chaîne. Une chaîne reliant uev dans un graphe G est une suite $u = u_0, e_1, u_1, ..., e_k, (u_k = v)$ tel que $\forall i = 1...k$, $e_i = u_{i-1}u_i \in E(G)$.

Définition : connexe. Un graphe est connexe si pour tout u, v n'importe quel sommet il existe une chaîne les reliant.
Un graphe non connexe se décompose en composantes connexes qui sont les sous graphes maximales connexes.

Remarque. La somme des degrés est égal à deux fois le nombre d'arêtes

$$\sum deg(V) = 2.|E(G)|$$

d'où la somme des degrés est toujours paire, le nombre de sommets de degré impair est pair.

Définition : cycle. Un cycle est une suite d'arête reliant un sommet à lui même sans passer deux fois au même sommet ou la même arrête.

Définition : arbre. Un arbre est un graphe connexe et sans cycle.