Algorithmes de tris

Algorithme		Complexité		sur place	stable
	meilleur	moyen	pire		
Tri par sélections	$O(n^2)$		$O(n^2)$	$\times$	$\times$
Tri par insertions	$O(n)$	$O(n^2)$	$O(n^2)$	$\times$	$\times$
Tri fusion	$O(n\log n)$		$O(n\log n)$		$\times$
Quicksort (simple)	$O(n)$	$O(n\log n)$	$O(n^2)$	$\times$	
Tri par tas	$O(n)$	$O(n\log n)$	$O(n\log n)$	$\times$	

On dit qu'un algorithme de tri fonctionne *sur place* s'il n'utilise pas de structure auxiliaire.
On dit qu'il est *stable* s'il préserve l'ordre relatif des éléments égaux.

Remarque : La plupart des langages de programmation utilisent le tri par insertions pour les séquences de petite taille; le tri rapide (amélioré), le tri fusion ou le tri par tas pour les séquences de grande taille. Par exemple Timsort qui combine tris par fusion et par insertion.

1) Tri par sélections : On met successivement en bonne position les plus petits éléments du tableau : à la i-ième, étape, on a les i plus petits éléments placés dans l'ordre : $A[0] \leq A[1] \leq ... \leq A[i-1]$, et on cherche le plus petit des éléments restants, qu'on met ensuite à sa place à l'aide d'un échange. La complexité est en $O(n^2)$.

Ecrire une procédure `triSelect(A)` qui effectue ce tri en modifiant A de sorte à en trier les éléments.

Solution :

```
triSelect(A) :
     n = len(A)
     for i in range(n) :
          k = i
          for j in range(i+1,n) :
               if A[j]<A[k] :  k = j
          A[i],A[k] = A[k],A[i]
```

2) Tri rapide par fusions (= tri d'une liste par fusions de listes triées)

On coupe la liste L en deux (de même longueur, à un près), on trie chaque moitié et on les fusionne.

Il faut $O(m)$ opérations pour fusionner deux listes triées de taille m. La complexité est en $O(n\log n)$.

Ecrire une fonction `triFusion(L)` qui renvoie la liste composée des éléments de L classés par ordre croissant.

On pourra utiliser (cf annexe) la fonction `fusion(L,M)` supposée connue qui fusionne deux listes triées.

Solution :

```
def triFusion(L) :
     n = len(L)
     if n <= 1 :  return L
     m = n//2
     return fusion(triFusion(L[0,m]),triFusion(L[m,n]))
```

3) Tri par insertions (= insertions successives dans des sous-tableaux triés)

a) Ecrire une procédure `triInsert(A)`qui trie un tableau par insertions successives, consistant à mettre le i-ième élément en bonne position parmi les éléments qui le précédent dans le tableau et qui sont triés par ordre croissant.

Solution :

```
def triInsert(A) :
     n = len(A)
     for i in range(n) :
          k = i
          while k>0 and A[k-1] > A[k] :
               A[k-1],A[k] = A[k],A[k-1] ;  k = k-1
```

```
    return A
```

b) *Variante pour les listes* : On suppose connue (cf annexe) une procédure `insert(x,L)` qui insère un élément x dans une triée L en $O(n)$ opérations.
Ecrire une fonction `triInsert2(L)` qui par insertions successives à une liste initialement vide construit (et renvoie) la liste composée des éléments de L classés par ordre croissant.

Solution :

```
def triInsert(L) :
    M = []
    for x in L : insert(x,M)
    return M
```

4) Tri rapide par pivots (QuickSort)

a) Une fois appliquée une partition par rapport à un pivot, il suffit de trier les deux parties qui sont de part et d'autre du pivot. On suppose connue (paragraphe b)) une procédure-fonction `pivot(A,i,j)` qui prenant $A[i]$ comme pivot, modifie les termes du tableau $A[i:j]$ en plaçant les termes inférieurs au pivto avant les termes supérieurs au pivot, met le pivot en place dans le sous-tableau $A[i:j]$, et renvoie sa position.

Ecrire une procédure récursive `triPivotAux(A,i,j)` qui trie le sous-tableau $A[i:j]$.
Solution :

```
def triPivotAux(A,i,j) :
    if j-i <= 1 :  return None
    k = pivot(A,i,j) ; triPivotAux(A,i,k) ; triPivotAux(A,k+1,j)
```

b) Deux implémentations possibles de la partition (cf annexe) :

- on utilise un tableau auxiliaire : on lit les éléments de A et on les place à gauche ou à droite dans un nouveau tableau selon qu'ils sont inférieurs ou supérieurs au pivot.

- implémentation sur place : on procède par des échanges en utilisant deux curseurs i et j qui partent des deux extrémités du tableau. L'algorithme s'arrête lorsque $i \geq j$.

c) *Remarque culturelle* : La complexité est $O(n \log n)$ en moyenne et $O(n^2)$ dans le cas le pire, et elle peut être améliorée dans les pires cas par un choix judicieux des pivots.

5) Tri à bulles

Il est basé sur des comparaisons entre éléments consécutifs :

Solution :

```
def triBulles(A) :
    n = len(A)
    for i in range(n) :
        for j in range(n-1,i,-1) :        # on a i < j ≤ n − 1
            if A[j]<A[j-1] :   A[j],A[j-1] = A[j-1],A[j]
```

L'étape i permet de mettre en bonne position le i-ième plus petit élément du tableau.

Important : Le tri à bulles dans sa première version ressemble au tri par sélections (invariants de boucle), mais procède par permutations d'éléments consécutifs comme dans l'algorithme de tri par insertions. L'algorithme utilise $O(n^2)$ échanges d'éléments dans tous les cas contrairement au tri par insertions dont le coût est $O(n)$ dans les meilleurs cas. Une optimisation courante du tri à bulles (qui n'est pas possible pour le tri par sélections) consiste à pouvoir interrompre le tri dès qu'un parcours des éléments dans la boucle interne est effectué sans permutation. En effet, cela signifie que le tableau entier est trié. Ainsi, le tri à bulles est en $O(n)$ opérations dans le meilleur cas.

Solution optimisée :

```
def triBulles(A) :
    n = len(A)
    for i in range(n) :
        flag = True
```

```
        for j in range(n-1,i,-1) :
            if A[j]<A[j-1] :
                A[j],A[j-1] = A[j-1],A[j]
                flag = False
        if flag :  return None
```

6) Tris par tas

On part de la liste des éléments à trier.
On crée un tas contenant ces éléments ordonnés selon leur valeur.
Cette création se fait en $O(n \log n)$, et même en $O(n)$ si on procède par dichotomie.
On récupère la liste triée en supprimant dans le tas les éléments un par un.
La complexité est en $O(n \log n)$.

7) Tris dans le cas d'informations connues sur les valeurs à trier

a) Tri par comptage

On suppose que les valeurs à trier appartiennent à un ensemble fini $E = \{x_0, ..., x_{p-1}\}$.
On crée en $O(n + p)$ alors un tableau de comptage M tel que $M[i] = \text{card}\{j \mid A[j] = i\}$.
On peut ensuite recomposer la liste triée en $O(n + p)$ opérations.
D'où un coût total en $O(n + p)$ opérations, particulièrement intéressant lorsque $p << n \log(n)$.

b) Tri par baquets

On suppose que les valeurs à trier appartiennent à un intervalle $E = [a, b]$.
On partitionne alors l'intervalle en p sous-intervalles E_0, ... , E_{p-1} de même longueur.
On crée en $O(n + p)$ alors une table H telle que $H[i]$ est la liste des j tels que $A[j] \in E_i$.
Puis on trie chaque liste $H[i]$ dont on note n_i la longueur, et enfin on concatène les listes obtenues.
En utilisant des tris quadratiques, la complexité totale est en $O(n + p) + O\left(\sum_{i=0}^{p-1} n_i^2\right) + O(n + p)$.

Dans le cas idéal où les $H[i]$ ont même longueur $n_i = \dfrac{n}{p}$, la complexité vaut $O\left(\dfrac{n^2}{p} + p\right)$.
La méthode est donc particulièrement intéressante lorsque les éléments du tableau se répartissent bien dans les différents "baquets" $H[i]$ et lorsque p est assez grand, idéalement de l'ordre de grandeur de n.

8) Annexes

Question **2)** :

```
def fusion(L,M) :
    n = len(L) ; m = len(M)
    S = [] ;  i = 0 ; j = 0
    while i<n and j<m :
        if L[i]<M[j] :  S.append(L[i]) ; i = i+1
        else :  S.append(L[j]) ; j = j+1
    for k in range(i,n) :  S.append(L[k])
    for k in range(j,m) :  S.append(M[k])
    return S
```

Question **3)** :

```
def insert(x,L) :
     k = len(L) ; L.append(x)     #  ainsi, L[k] vaut x
     while k>=0 and L[k-1]>L[k] :        # on fait descendre x à sa place
         L[k-1],L[k] = L[k],L[k-1]
```

Question **4)** :

```
def pivot(A,i0,j0) :     # on suppose j0 - i0 > 0, c'est-à-dire le sous-tableau n'est pas vide
    B = [0]*(j0-i0) ; x = A[i0] ; i = i0 + 1 ; j = j0 - 1
    for k in range(i0+1,j0) :
        if A[k] <= x :  B[i] = A[k] ; i = i + 1
```

```
        if A[k] > x :  B[j] = A[k] ; j = j - 1
            #   invariants : pour i0 < k < i, on a B[k] ≤ x ; et pour j < k < j0, on a B[k] > x
            #   à la sortie, i = j ; il reste à mettre le pivot A[i0] à la bonne place :
    B[i] = x
            #  puis il faut modifier le tableau initial A
    for k in range(i0,j0) :  A[k] = B[k]
    return i

def pivot2(A,i0,j0) :
    x = A[i0] ; i = i0 + 1 ; j = j0
    while i < j  :      #    il vaut mieux faire une seule opération à la fois :
        if A[i] <= x :  i = i + 1
        elif A[j] > x :  j = j - 1
        else :  A[i],A[j] = A[j],A[i]
            #   invariants  de boucle : pour k < i, on a A[k] ≤ x ; et pour k > j, on a A[k] > x
            #   à la sortie, i = j ; il reste à mettre le pivot A[i0] à la bonne place :
    if A[i] < x :  A[i0],A[i] = A[i],A[i0]
    elif A[i] > x :  A[i0],A[i-1] = A[i-1],A[i0]
            # dans la précédente instruction, il se peut que i soit égal à i0 + 1
    return i
```