Informatique — MPSI/PCSI

Correction du TP n° 8

Exercice 8.1 (Tri à bulles)

1. En partant de la fin, on compare tous les couples

```
def TriBulles(L):
    n = len(L)
    for a in range(n-1):
        for k in range (1,n-a):
            if L[n-k] < L[n-k-1]:
                L[n-k], L[n-k-1] = L[n-k-1], L[n-k]
```

2. Dans le meilleur des cas, c'est-à-dire lorsque la séquence est triée, il faut $O(n)$ opérations. Dans le pire des cas, c'est-à-dire lorsque la liste est triée en sens inverse, il faut

$$(n-1)+(n-2)+\cdots+1=\sum_{k=1}^{n-1}=\frac{n(n-1)}{2}$$

décalages, soit $O(n^2)$ opérations.

Exercice 8.2 (Tri par insertion)

1. À partir de la description du tri donnée, il vient l'implémentation suivante.

```
def TriInsertion(L):
    for k in range(1,len(L)):
        c=L[k]
        j=k
        while j>0 and c<L[j-1]:
            L[j]=L[j-1]
            j-=1
        L[j]=c
```

2. La complexité au pire est obtenue lorsque la séquence, supposée de longueur n, est initialement en ordre décroissant et que le k-ième élément $(0 \leqslant k \leqslant n-1)$ nécessite k décalages. Il vient alors :

$$\sum_{k=1}^{n-1} k = \frac{n\,(n-1)}{2}$$

d'où une complexité au pire en $\mathcal{O}(n^2)$. La complexité au mieux est obtenue pour une séquence triée et donc $(n-1)$ tests d'où une complexité au mieux en $\mathcal{O}(n)$.

3. L'invariant de boucle utilisé est que la séquence $L_k = \{a_0, a_1, \ldots, a_k\}$ est triée.

   Initialisation Quand $k=1$, L_0 se compose d'un unique élément a_0 et est donc triée.

   Conservation Au cours de la k-ième insertion, $1 \leqslant k \leqslant n-1$, on a au début de l'itération L_{k-1} triée. On insère a_k dans cette séquence triée de sorte que l'on ait en fin d'itération L_k triée.

   Terminaison La boucle `for` prend fin lorsque k dépasse $n-1$ et donc vaut n. Ainsi, L_{n-1} correspond à la version intégralement triée de la séquence initiale.

4. Pour permuter deux éléments d'une séquence, il suffit de sauvegarder de façon temporaire un des deux objets pour réaliser la permutation. Tenant compte du fait que l'on agit sur la liste elle-même (pointeur), il n'est pas nécessaire de renvoyer quoi que ce soit.

```
def Permuter(L,i,j):
    tmp=L[i]
    L[i]=L[j]
    L[j]=tmp
```

La version suivante utilise le *swap* natif des objets en python.

```
def Permuter(L,i,j):
    L[i],L[j]=L[j],L[i]
```

5. Pour définir la fonction `Inserer`, on commence par remarquer que l'on travaille sur une séquence L de longueur $k+1$ où les k premiers éléments sont triés. Il faut donc trouver dans cette sous-séquence et en partant de la fin, la position de l'élément à insérer. Avec l'expression donnée, il vient :

```
def Inserer(L,k):
    if k>0 and L[k]<L[k-1]:
        Permuter(L,k,k-1)
        Inserer(L,k-1)
```

6. Pour définir une version récursive du tri par insertion, il suffit d'exploiter l'invariant de boucle, partant de $k=1$ et s'arrêtant à $k=n$.

```
def TriInsRec(L,k=1):
    if k<len(L):
        Inserer(L,k)
        TriInsRec(L,k+1)
```

Exercice 8.3 (Tri par sélection)

1. On commence par définir une fonction `imin` qui renvoie l'indice du minimum d'une séquence à partir de l'indice k :

```
def imin(L,k):
    for i in range(k+1,len(L)):
        if L[i]<L[k]:
            k=i
    return(k)
```

puis on parcourt la séquence en permutant le k-ième élément avec le minimum des $n-k+1$ éléments suivants.

```
def TriSelection(L):
    for k in range(len(L)):
        Permuter(L,k,imin(L,k))
```

2. La complexité est ici constante pour toute séquence de longueur n. Positionner le k-ième élément $(0 \leqslant k \leqslant n-2)$ nécessite $n-k+1$ tests pour trouver le minimum. Il vient alors :

$$\sum_{k=1}^{n-1} k = \frac{n\,(n-1)}{2}$$

d'où une complexité au pire en $\mathcal{O}(n^2)$.

3. Pour définir une version récursive de l'algorithme de tri par sélection, il suffit d'exploiter l'invariant de boucle qui est que, à la k-ième itération, $L_k \leftrightarrow \min\limits_{k\leqslant j<n}(L_j)$ et s'arrêter à L_{n-1}.

```
def TriSelRec(L,k=0):
    if k<len(L)-1:
        Permuter(L,k,imin(L,k))
        TriSelRec(L,k+1)
```

4. En procédant avec les maximums, il vient :

```
def imax(L):
    i=0
    for k in range(1,len(L)):
        if L[k]>L[i]:
            i=k
    return(i)
```

et de façon impérative

```
def TriSelection(L):
    n=len(L)
    for k in range(n):
        Permuter(L,n-k-1,imax(L[:n-k]))
```

ou, de façon récursive

```
def TriSelRec(L,k=0):
    n=len(L)
    if k<n-1:
        Permuter(L,n-k-1,imax(L[:n-k]))
        TriSelRec(L,k+1)
```

Exercice 8.4 (Tri par partition-fusion)

1. Remarquant que le principe de la fusion consiste simplement à comparer les premiers éléments des deux séquences puis à appeler récursivement la fonction elle-même jusqu'à ce qu'une des deux séquence soit vide, il vient le code suivant.

```
def Fusion(M,N):
    if M==[]:
        return(N)
    elif N==[]:
        return(Fusion(N,M))
    elif M[0]<=N[0]:
        return([M[0]]+Fusion(M[1:],N))
    else:
        return(Fusion(N,M))
```

2. Pour diviser une séquence de longueur n en deux, on utilise le *slicing* et le quotient de $n/2$. Si la séquence contient 0 ou 1 élément, c'est qu'elle est triée. Il vient le code suivant.

```
def TriFusion(L):
    n=len(L)
    if n<2:
        return(L)
    else:
        m=n//2
        return(Fusion(TriFusion(L[:m]),TriFusion(L[m:])))
```

3. La complexité en temps de l'algorithme de partition-fusion dépend :
   — du nombre de comparaisons de la fonction `Fusion`, soit $n-1 = O(n)$ pour 2 séquences contenant n éléments en tout ;
   — du nombre d'appels récursifs de la fonction `Fusion` pour réaliser :

$$\begin{aligned} &\left\lfloor \frac{n}{2} \right\rfloor \text{ fusions de 2 éléments} \\ +&\left\lfloor \frac{n}{4} \right\rfloor \text{ fusions de 4 éléments} \\ +&\dots \\ +&\left\lfloor \frac{n}{2^k} \right\rfloor = 1 \text{ fusion de } 2^k \text{ éléments} \end{aligned}$$

   où $k \simeq \log_2(n)$. Il vient alors par somme et au pire :

$$\sum_{i=1}^{\log_2(n)} \left\lfloor \frac{n}{2^i} \right\rfloor 2^i = n \sum_{i=1}^{\log_2(n)} 1 = n \log_2(n) = O\left(\ln(n)\right)$$

   La complexité est donc constante et vaut $O\left(n \ln(n)\right)$. Attention, la complexité en espace est en $O(n)$ puisqu'on crée une nouvelle liste de même longueur que la liste à trier et cela peut poser problème pour de très grandes listes.

Exercice 8.5 (Tri rapide)

1. On partitionne la liste `L[i:j+1]` suivant le pivot `L[i]` et on renvoie la nouvelle position du pivot.

```
def Partitionnement(L,i,j):
    v=L[i]
    a=i
    for k in range(i+1,j+1):
        if L[k]<v:
            a+=1
            Permuter(L,a,k)
    Permuter(L,a,i)
    return(a)
```

2. Pour le tri rapide, on se sert de la fonction `Partitionnement` par appels récursifs. Il vient le code suivant.

```
def TriRapide(L, i=0, j=None):
    if j==None:
        j=len(L)-1
    if i<=j:
        k = Partitionnement(L,i,j)
        TriRapide(L,i,k-1)
        TriRapide(L,k+1,j)
```

3. La complexité en temps de l'algorithme de tri rapide dans le pire des cas, c'est-à-dire pour une liste triée dans l'ordre décroissant, vaut $O(n^2)$. Dans le meilleur des cas, le pivot est placé au milieu de la séquence et la complexité correspond à celle du tri par partition-fusion. Dans tous les cas, la complexité en espace est diminuée.

Exercice 8.6 (Tri par comptage)

1. Tirant profit du fait que les éléments de `L` sont des entiers, ils peuvent servir d'indices pour la séquence des occurrences `C` initialisée avec N zéros. Le code suivant convient.

```
def comptage(L,N):
    C = [0 for k in range(N)]
    for e in L:
        C[e]+=1
    return(C)
```

2. Une fois construit l'histogramme, il suffit d'affecter successivement le bon nombre d'occurrences des entiers de $[\![0, N-1]\!]$ à `L` pour ne pas augmenter la complexité spatiale.

```
def TriComptage(L,N):
    i=0
    for e,n in enumerate(comptage(L,N)):
        for j in range(n):
            L[i]=e
            i+=1
```

3. Avec 2 boucle de n termes, la complexité en temps de cet algorithme est linéaire, soit en $O(n)$. La complexité en espace est en $n+N$. Son seul inconvénient porte sur la nature très restrictive de ses données d'entrée.

* *
*