Interrogation d'informatique n°2. Corrigé.

Exercice A. Chemins multicolores

1)

```
def voisins(T,G,C,k) :
    R = [0]*n                       # complexité O(n)
    for x in range(n) :
        if T[x] == 1 :
            for y in G[x] :         # complexité O(m_x)
                if C[y] == k :  R[y] = 1
    return R
```

La complexité est en $O(n + \sum_{x \in A} m_x)$

2)

```
def cheminColore(G,C,s,t,r) :
    T = [0]*n ; T[s] = 1
    for k in range(1,r+2) :  T = voisins(T,G,C,k)
    return (T[t] == 1)
```

3) Notons A_k la partie de S obtenue à l'étape.

Les parties A_k sont nécessairement disjointes, car les villes de A_k sont de couleur k.

La complexité de `voisins` à l'étape k est en $O(n + \sum_{x \in A_k} m_x)$.

Donc la complexité totale est en $\sum_{k=1}^{r+1} O(n + \sum_{x \in A_k} m_x)$.

On a $\sum_{k=1}^{r+1} O(n) = (r+1)n$ et les A_k étant diajoints, on a $\sum_{k=1}^{r+1} \sum_{x \in A_k} m_x \leq \sum_{x \in S} m_x = m$.

Donc la complexité totale est en $O((r+1)n + m)$.

4)

```
def partition(C,r) :
    n = len(C)
    S = [ [] for k in range(r+2)]      # ne pas utiliser [[]]*(r+2)
    for x in range(n) :  S[C[x]].append(x)
    return S
```

5)

```
def voisins2(G,T,S,C,k) :
     for x in S[k-1] :
          if T[x] == 1 :
             for y in G[x] :
                   if C[y] == k :  T[y] = 1
```

6)

```
def cheminColore2(G,C,s,t,r) :
     S = partition(C,r)
     T = [0]*n ; T[s] = 1
     for k in range(1,r+2) :  voisins2(G,T,S,C,k)
     return (T[t] == 1)
```

7) A la fin de l'algorithme, la partie A codée par T est la réunion de toutes les listes A_k.

Notons p_k le nombre de villes de A_k, c'est-à-dire le nombre de villes de couleur k dans A.

Notons s_k le nombre de villes de couleur k.

La complexité de `voisins2` est $O(s_{k-1} + \sum_{x \in A_k} m_x + p_k)$.

En effet, il y a s_{k-1} villes dans `S[k-1]` et p_k affectations dans T dans `voisins2(G,T,S,C,k)`.

La complexité de `cheminColore2` est donc $O(N)$, où $N = \sum_{k=1}^{r+1} \left(s_{k-1} + p_k + \sum_{x \in A_k} m_x \right) \leq n + n + m$.

Donc on obtient bien une complexité en $O(n+m)$.

Exercice B. Mots de Dyck (= expressions bien parenthésées) et permutations de pile

1) On note que les k premiers termes de L contiennent au moins autant de 1 que de -1 ssi $L_1 + ... + L_k \geq 0$.

D'où la fonction :

```
def Dycko(L) :
     n = len(L) ; s = 0
     for k in range(n) :
         s = s + L[k]                    # s représente la somme L_1 + ... + L_k des éléments lus
         if s < 0 :  return(False)
     return (s == 0)                     # si on arrive à la fin du mot, on teste si s est nul
```

2) $L = [1, 1, ...1, -1, -1, ..., -1]$ est associée à $\sigma = (n-1, n-2, ..., 3, 2, 1, 0)$.

$L' = [1, -1, 1, -1, 1, -1, ..., 1, -1]$ est associée à $\sigma' = (0, 1, 2, ..., n-1)$.

3) L'idée est d'utiliser une liste A (variable locale) représentant la pile de l'algorithme.

```
def permut_de_dyck(L) :
    A = [] ; i = 0 ; pile = []
    for x in L :
        if x == 1 :  pile.append(i) ; i = i+1
        else :  L.append(pile.pop())
    return L
```

4) L'idée est d'utiliser $s(\sigma(k)) = k$ en faisant varier k de 0 à $n-1$.

```
def inverse(A) :
    n = len(A) ; B = [0]**n
    for k in range(len(A)) : B[A[k]] = k
```

5) $s(i)$ donne le numéro de sortie de l'élément i.

Plus précisement, $s(i)$ est le nombre d'éléments extraits de la pile avant que i n'en soit extrait.

Par exemple, $s(0) = 2$ ssi 0 est sorti de la pile en troisième position (après $\sigma(0)$ et $\sigma(1)$).

6) L'idée est de simuler à nouveau la pile et à chaque étape de comparer l'élément $\sigma(k)$ avec l'élément t situé en haut de pile :

- si $\sigma(k)$ est l'élément en haut de pile, on le supprime de la pile, on incrémente k et on ajoute -1 à L
- sinon, on ajoute i à la pile, on incrémente i et on ajoute $+1$ à L.

Remarque : Initialement, la pile est vide, i et k valent 0.

On utilise une fonction auxiliaire

```
def tete(pile) :   # renvoie l'élément en tête de pile (et renvoie -1 si la pile est vide)
    if pile :  return pile[len(pile)-1]
    else :  return -1

def dyck_de_permut(A) :
    n = len(A) ; i = 0 ; k = 0 ; pile = [] ;  L = []
    while k < n
        if tete(pile) == A[k] :
            L.append(-1) ; pile.pop() ; k = k+1
        else :
            L.append(1) ; pile.append(i) ; i = i+1
```

La complexité est en $O(n)$: Le nombre d'opérations effectués correspond au nombre de $+1$ ou de -1 ajoutés,

donc il vaut $2n$ puisqu'on obtient finalement un mot de Dyck !

7)

```
def test(A) :
    return Dycko(dyck_de_permut(A))
```

La complexité est en $O(n)$ car il en est de même des fonctions `dyck_de_permut` et `Dycko`.

Exercice C. Tri Shell

1) La commande `assert` permet de s'assurer que p est un entier naturel non nul.

Les lignes 04 à 09 consistent à effectuer un tri par insertions sur le sous-tableau des termes d'indice congru à r modulo p, c'est-à-dire le sous-tableau $[x_r, x_{r+p}, x_{r+2p}, x_{r+3p}, ...]$.

Ce tri est effectué pour toute valeur de $r \in \{0, 1, ..., p-1\}$.

2) En particulier, `tri(A,1)` est le tri par insertions classique (dans ce cas, r prend 0 comme unique valeur).

Le nombre d'échanges effectués est $N = \sum_{i=0}^{n-1} m(i) \leq \sum_{i=0}^{n-1} i = \dfrac{1}{2}n(n-1)$, donc de l'ordre de $\frac{1}{2}n^2$.

Cette valeur est atteinte lorsque les éléments de A sont classés par ordre décroissant.

3) Le nombre d'échanges effectués par `tri(A,2)` est donc au plus

$$\frac{1}{2}n_0(n_0-1) + \frac{1}{2}n_1(n_1-1), \text{ où } n_0 = \left\lfloor \frac{n}{2} \right\rfloor \text{ et } n_1 = \left\lceil \frac{n}{2} \right\rceil, \text{ donc de l'ordre de } \frac{1}{4}n^2$$

Une fois les deux sous-tableaux $[x_0, x_2, x_4, ...]$ et $[x_1, x_3, x_5, ...]$ triés, on obtient un tableau A vérifiant :

$$m(i) \leq \left\lceil \frac{i}{2} \right\rceil .$$

En effet, après `tri(A,2)`, tous les x_j, où $j < i$ est de même parité que i, vérifient $x_j \leq x_i$.

Donc le nombre d'échanges effectués ensuite par `tri(A,1)` est donc au plus

$$\sum_{i=0}^{n-1} \left\lceil \frac{i}{2} \right\rceil, \text{ qui est de l'ordre de } \frac{1}{2}\left(\frac{n}{2}\right)^2 = \frac{1}{8}n^2$$

Ainsi, au total, `tri(A,1) ; tri(A,2)` effectue environ $\dfrac{3n^2}{8}$ échanges, donc moins que $\dfrac{n^2}{2}$.

La méthode est ainsi plus efficace (dans le pire cas) car les éléments qui doivent être avancés dans la liste progressent plus rapidement vers leur position si on utilise d'abord des pas de 2 (au lieu de pas de 1).

Remarque culturelle :

En effectuant `tri(A,p)` successivement pour tous les entiers p de la forme $2^a 3^b$ pris dans l'ordre décroissant, on obtient une complexité $O(n(\log n)^2)$, qui est la meilleure complexité connue dans un tri Shell.