Interrogation d'informatique n°5. Corrigé

Exercice A

```
01.     def somme(L) :
02.         if isinstance(x,int) :  return x    ###   test d'arrêt
03.         s = 0
04.         for M in L : s = s + somme(M)
05.         return s
```

Exercice B

1)
```
def inverse(A) :
    n = len(A) ; B = [ [] for k in range(n)]
    for i in range(n) :
        for j in A[i] :  B[j].append(i)
    return B
```

2) a) (i) étape 1 : ligne 02 ; étape 2 : lignes 03 à 06 ; étape 3 : lignes 07 à 16 ; étape 4 : ligne 17

(ii) Le booléen `test` vaut `True` s'il reste au moins un élément non étiqueté, c'est-à-dire s'il existe au moins un sommet i tel que $H[i]$ vaut -1.

(iii) 14. `if H[j] == -1 or H[j] == k :  flag = False`

Pour avoir $H[i] = k$, une condition nécessaire et suffisante est :
- que i n'ait pas été étiqueté à l'étape $(k-1)$, c'est-à-dire qu'au début de l'étape k, $H[i]$ vaille -1
- que tous les précédesseurs j de i soient étiquetés et admettent des hauteurs $< k$.

Or, durant le déroulement de l'étape k, toutes les étiquettes sont $\leq k$ (celles de valeurs k viennent d'être attribuées).
Il convient donc d'exclure les sommets dont au moins un des prédécesseurs admet une étiquette -1 ou k.
Le drapeau `flag` devient justement `False` si l'un des prédécesseurs vérifie cette condition.

b) A chaque itération en k, on parcourt tous les sommets et toutes les arêtes dont l'extrémité n'a pas encore été etiquetée (c'est-à-dire les arêtes $j \to i$ avec $H[i] = -1$). La complexité en $O(n(n+m))$.

Remarque : Pour donner un exemple où cette complexité est atteinte, on considère le graphe acyclique de hauteur n tel que pour tous $0 \leq i < j \leq n$, on a $i \to j$. On a ainsi $m = \frac{1}{2}n(n-1)$.
Et à la k-ième étape, les k premiers sommets ont été étiquetés et le nombre d'opérations effectuées à la k-ième étape est $n + \sum_{i=k}^{n-1} i$. D'où au total une complexité en $O(n^3) = O(nm)$.
En effet, $\sum_{0 \leq i \leq k < n} i = \sum_{i=0}^{n-1} i(n-i) = \frac{1}{2}n^2(n-1) - \frac{1}{6}n(n-1)(2n-1) \sim \frac{1}{6}n^3$.

c) Il suffit de modifier les lignes 02, 05 et 15 et 18 :

```
02.      ### On ajoute l'instruction L = []
05.           if B[i] == [] :  H[i] = 0 ; L.append(i)
```

```
15.             if flag :  H[i] = k ; L.append(i)
18.       return L
```

3) a)
```
def kahn(A) :
    ### Calcul des degrés entrants :
    n = len(A) ; degre = [0]*n
    for i in range(n) :
        for j in A[i] :  degre[j] += 1
    ### Calcul des sommets de degré entrant 0 :
    pile = []     # pile des éléments à traiter
    for i in range(n) :
         if degre[i] == 0 :  pile.append(i)
    ### Ordonnancement :
    L = []
    while pile :
         i = pile.pop() ; L.append(i)
         for j in A[i] :
              degre[j] = degre[j] - 1
              if d[j] == 0 :  pile.append(j)
    return L
```

b) La complexité est linéaire en $O(n+m)$. chaque sommet est ajouté une seule fois à la pile ; chaque arête est visitée une seule fois (les arêtes $i \to j$ sont visitées lorsque i est dépilé).

4) a) $K[i]$ est la longueur **maximale** des chemins issus de i, appelé **profondeur** de i dans le graphe des tâches A. On le prouve par induction : si la propriété est vraie pour les successeurs de i dans le graphe des tâches, elle est vraie pour i.

Remarque : La complexité est ici linéaire en $O(n+m)$: en effet chaque sommet est traité une première fois, permettant d'attribuer une valeur positive à $K[i]$, lequel vaut initialement -1, et ensuite le nombre d'appels à `traite(i)` correspond au nombre de prédécesseurs de i.

b) On obtient les sommets dans l'ordre inverse de celui souhaité.
En effet, si $i \to j$, alors la tâche i est traitée après la tâche j. D'où :

```
def f(A) :
    n = len(A) ; K = [-1]*n ; L = []       ### ligne modifiée
    def traite(i) :
         if K[i] == -1 :
              m = 0
```

```
            for j in A[i] :
                traite(j) ; m = max(m,1+K[j])
            K[i] = m ; L.append(i)      ### ligne modifiée
    for i in range(n) :  traite(i)
    return reversed(L)                  ### ligne modifiée
```

avec

```
def reversed(L) :
    return L[i] for i in range(len(L)-1,-1,-1)
```

c) **La hauteur d'un sommet est sa profondeur dans le graphe inversé**. D'où :

```
def g(A) :
    return f(inverse(A))
```

d)
```
def f(A) :
    n = len(A) ; K = [-2]*n ; pile = []
    while pile :
        i = pile.pop()
        if K[i] == -2 :    # on ajoute les prédécesseurs de i en tête de pile
            K[i] == -1 ; pile.append(i)
            for j in A[i] :  pile.append(j)
        if K[i] == -1 :
        # les prédécesseurs ont tous été traités ; on enlève i de la pile et on calcule H[i]
            m = 0
            for j in B[i] :  m = max(m,1+K[j])
            K[i] = m
        # sinon, K[i] a déjà été calculé : on ne fait rien et on enlève i de la pile.
    for i in range(n) :  pile.append(i)
    return K
```