Interrogation d'informatique n°4. Corrigé

Exercice A

1) a) La ligne à compléter est :

```
05                q = q//2
```

`binaire(6,6)` renvoie `[0,1,1,0,0,0]`.

b)
```
def liste(n) :
    return [binaire(k,n) for k in range(2**n)]
```

2) a) `liste(2)` renvoie `[[0,0],[0,1],[1,0],[1,1]]`

b) Lorsque $n = 2$, les valeurs prises par `tab_partiel` sont successivement

`[-1,-1]`, `[0,-1]`, `[0,0]`, `[0,1]`, `[1,1]`, `[1,0]`, `[0,1]`

Remarque : La valeur indiquée en grad correspond à l'élément d'indice k.

c) Le tableau `tab_partiel` est modifié au fur et à mesure des appels récursifs. S'il n'était pas modifié, toutes les valeurs de `liste` pointeraient vers le même tableau (et dont la valeur est celle associée à l'appel récursif en cours).

Exercice B. Problème du sac-à-dos

1)
```
def depaq(L) :
    return [c[0] for c  in L] , [c[1] for c  in L]
```

2) *Remarque culturelle* : En revanche, si on pouvait *fractionner* les objets, la stratégie gloutonne serait optimale : elle consiste à classer les objets selon la valeur du taux de rentabilité valeur/poids, puis à remplir tant que cela est possible le sac-à-dos par les objets les plus rentables puis à placer la fraction correspondante au premier objet restant afin d'achever de remplir le sac-à-dos.

a)
```
def glouton(L,C) :
    s = 0 ; reste = C ; i = 0 ; n = len(L) ; p,v = depaq(L)
    for i in range(n) :
        if p[i] <= reste :
           s = s + v[i] ; reste = reste - p[i]
    return s
```

Attention au risque de dépassement de tableau : on utilise (`i` $<$ `n`) et l'évaluation paresseuse.

b) La complexité est en $O(n)$, et donc en $O(n \ln n)$ si on tient compte du tri initial.

3) a)
```
def somme(v,a) :
    s = 0
    for i in range(n) :  s = s + v[i]*a[i]
    return s
```

b)
```
def successeur(a) :
```

```
i = 0
while a[i] == 1 :     # on peut aussi ajouter i<len(a)
     a[i] == 0 :  i = i+1
a[i] == 1
```

Remarques : il s'agit ici d'une procédure : on ne renvoie aucune valeur.

Il n'y a pas de risque de dépassement de tableau, car la liste a contient au moins un 0.

c) `def optimal(L,C) :`

```
    n = len(L) ; a = [0]*n ; v_max = 0
         # lorsque a = [0,...,0], on a somme(v,a) = 0
    for k in range(2**n-1) :
         successeur(a)
         if somme(v,a) > v_max and somme(p,a) <= C : v_max = s
    return v_max
```

Remarque : Dans `successeur`, on ne peut pas prendre $a = [1, 1, ..., 1]$ comme argument.

Il faut donc veiller à ce que `a` ne prenne pas cette valeur, d'où la boucle de longueur `2**n-1`.

On pourrait éviter ce problème en gérant le cas de dépassement de tableau dans `successeur`.

Remarque : On peut aussi utiliser une boucle de la forme `while a != [1]*n`

4) a) $M[0,k] = 0$ et pour $0 \leq i < n$, $M[i+1,k] = \begin{cases} \max(M[i,k], M[i,k-p_i] + v_i) & \text{si } p_i \leq k \\ M[i,k] & \text{si } p[i] > k \end{cases}$

b) `def opt_dyn(L,C) :`

```
     n = len(L) ; M = [[0 for k in range(C+1)] for i in range(n+1)]
     for i in range(n) :
          for k in range(C+1) :
               M[i+1][k] = M[i,k]
               if p[i] <= k and M[i][k-p[i]]+v[i] > M[i+1][k] :
                    M[i+1][k] = M[i][k-p[i]]+v[i]
     return M[n][C]
```

En récursif : Pour programmer en récursif, il faut utiliser la mémoïsation via un dictionnaire (ou une liste).

```
def opt_dyn(L,C) :
     M = {} ; n = len(L)      ### M dictionnaire de mémoïsation
     for k in range(C+1) :  M[(0,k)] = 0
     def aux(i) :      ### ici 0 ≤ i < n
          # il s'agit ici de définir les M[(i+1,k)] en fonction des M[(i,*)]
     aux(n-1)
     return M[(n,C)]
```

Remarque : Il est essentiel d'utiliser un seul dictionnaire, donc défini dans la fonction principale et non dans la fonction auxiliaire récursive.

c) Il convient de partir de la case $M[n][C]$. Le dernier objet est sélectionné ssi $M[n-1][C] < M[n][C]$.

Dans ce cas, on se ramène au cas de $M[n-1][C-p_{n-1}]$.

Sinon, on est ramené au cas de $M[n-1][C]$. On peut ensuite itérer le procédé.

```
def sac(M) :
    n = len(M)-1 ; k = len(M[0])-1 ; L = []
    for i in range(n-1,-1,0) :
        if M[i,C] < M[i+1,C] :
            L.append(i) ; k = k - L[i][0]
```

La complexité est en $O(n)$.

5) a)
```
def opt_back(L,i,C) :
        n = len(L)
        if i == n :  return 0    # test d'arrêt, s'il n'y a aucun objet
        (p,v) = L[i]
        if C < p :      # la branche ne peut pas contenir i, mais seulement des objets j > i
             return opt_back(L,i+1,C)
               ### on a ainsi élagué la branche contenant i
        else :           # la branche peut contenir i ou des objets j > i
             w = v + opt_back(L,i+1,C-p)
             return max(w,opt_back(L,i+1,C))
```

Remarque : Ici, le coût est trop élevé du fait des redondances.

Pour avoir la complexité souhaitée, il faut utiliser la mémoïsation :

```
memo = {}    ###  on définit
def opt_back(L,i,C) :
        n = len(L)
       if (i,C) in memo :  return memo[(i,C)]
        if i == n :
            memo[(i,C)] = 0 ; return 0
        (p,v) = L[i]
        if C < p :
             memo[(i,C)] = opt_back(L,i+1,C)
                    # on a ainsi élagué la branche contenant i
        else :           # la branche peut contenir i ou des objets j > i
             w = v + opt_back(L,i+1,C-p)
             memo[(i,C)] = max(w,opt_back(L,i+1,C))
       return memo[(i,C)]
```