Dans l'exemple précédent, on a :

Objet	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Poids	3	8	5	1	6	1	2	6	6
Valeur	1	2	6	3	7	8	2	3	4

Poids maximal = 14.

et le tableau `tab` obtenu est le suivant :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	2	2	2	3	3	3	3
2	0	0	0	1	1	6	6	6	7	7	7	7	7	8	8
3	0	3	3	3	4	6	9	9	9	10	10	10	10	10	11
4	0	3	3	3	4	6	9	10	10	10	11	13	16	16	16
5	0	8	11	11	11	12	14	17	18	18	18	19	21	24	24
6	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24
7	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24
8	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24

En observant les deux dernières lignes du tableau, on peut décider si l'on écarte ou si l'on conserve l'objet x_{n-1}. En réitérant cette analyse de la dernière à la première ligne on obtient alors une composition optimale du sac à dos.
Plus précisément, en partant de la dernière case et en utilisant la formule de récurrence trouvée auparavant, on va trouver un «chemin» qui permet d'arriver à cette valeur optimale. Si l'on est sur la case d'indice (k, p) :

- si $p_k > p$ alors on passe à la case d'indice $(k-1, p)$ et on ne garde pas l'objet k ;
- sinon on distingue encore deux cas :
  - si $f(k,p) = f(k-1,p)$ alors on passe à la case d'indice $(k-1, p)$ et on ne garde pas l'objet k,
  - si $f(k,p) = v_k + f(k-1, p-p_k)$ alors on passe à la case d'indice $(k-1, p-p_k)$ et on garde l'objet k.

  Si les deux derniers cas se produisent en même temps, cela signifie qu'il existe plusieurs solutions optimales ; et on choisit une possibilité quelconque.

Sur l'exemple considéré cela donne :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	2	2	2	3	3	3	3
2	0	0	0	1	1	6	6	6	7	7	7	7	7	8	8
3	0	3	3	3	4	6	9	9	9	10	10	10	10	10	11
4	0	3	3	3	4	6	9	10	10	10	11	13	16	16	16
5	0	8	11	11	11	12	14	17	18	18	18	19	21	24	24
6	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24
7	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24
8	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24

donc les objets conservés sont les objets 2, 3, 4 et 5.

Voici un programme fondé sur cet algorithme.

```
def sac_a_dos_avec_compo(objets, pMax):
    """ entrées :
            objets liste de 2 listes d'entiers (poids et valeurs)
                de même longueur
            pMax entier > 0
        sortie :
            valeur maximale pour un poids total <= pMax
            et liste des objets correspondants
    """
    p, v = objets  # poids et valeurs
    n = len(p)
    tab = [[0 for _ in range(pMax+1)] for _ in range(n)]
    for j in range(pMax+1):
        if j < p[0]:
            tab[0][j] = 0
        else:
            tab[0][j] = v[0]
    for i in range(1, n):
        for j in range(pMax+1):
            if j < p[i]:
                tab[i][j] = tab[i-1][j]
            else:
                x = tab[i-1][j]
                y = v[i] + tab[i-1][j-p[i]]
                tab[i][j] = max(x, y)

    L = []
    i, j = n-1, pMax
    while i > 0:
        if tab[i-1][j] == tab[i][j]:
            i -= 1
        else:
            L.append(i)
            j -= p[i]
            i -= 1
    if tab[0][j] > 0:
        L.append(0)

    L.reverse()

    return tab[n-1][pMax], L
```

Par exemple, avec les listes de poids et valeurs données dans la variable `test` et `pMax=14`, on obtient une valeur maximale de 24.

```
>>> test = [[3, 8, 5, 1, 6, 1, 2, 6, 6], [1, 2, 6, 3, 7, 8, 2, 3, 4]]
>>> sac_a_dos_avec_compo(test, 14)
(24, [2, 3, 4, 5])
```

III.2 - Stratégie récursive

Proposons maintenant une version récursive.

```
def sac_a_dos_memo(objets, pMax):
    """ entrées :
            objets liste de 2 listes d'entiers (poids et valeurs)
                de même longueur
            pMax entier > 0
        sortie :
            valeur maximale pour un poids total <= pMax
    """
    p, v = objets  # poids et valeurs
    n = len(p)
    tab = [[-1 for _ in range(pMax+1)] for _ in range(n)]

    def f(k, pds):
        if tab[k][pds] == -1:
            if k == 0 and p[0] > pds:
                res = 0
            elif k == 0:
                res = v[0]
            elif p[k] > pds:
                res = f(k-1, pds)
            else:
                a = f(k-1, pds)
                b = f(k-1, pds - p[k])
                res = max(a, b + v[k])
            tab[k][pds] = res
        return tab[k][pds]

    return f(n-1, pMax)
```

IV - L'exemple du déplacement sur un damier

On se donne un damier (représenté par exemple par une liste de listes) dont chaque case est affectée d'un poids (*i.e.* d'un entier strictement positif $m(i,j)$ pour la case (i,j)).
On souhaite aller de la case $(0,0)$ vers la case d'indice $(n-1,p-1)$ avec comme seuls déplacements autorisés : sauter d'une case horizontalement «à droite» ou sauter d'une case verticalement «en bas».
Le poids d'un chemin est égal à la somme des poids des cases traversées. Le but est de trouver un chemin de poids minimal.

1. On note $f(i,j)$ le poids minimal pour un chemin reliant la case $(0,0)$ à la case (i,j).

a. Pour quels cas le calcul de $f(i,j)$ est-il direct?

b. Trouver une relation reliant $f(i,j)$, $f(i-1,j)$ et $f(i,j-1)$.

2. Programmer une version itérative pour trouver le poids minimal pour la route reliant les cases $(0,0)$ et $(n-1,p-1)$.

3. Adapter la fonction précédente pour obtenir également un chemin de poids minimal.

4. Proposer une version récursive.

ÉLÉMENTS DE CORRECTION –

1. On note $f(i,j)$ le poids minimal pour un chemin reliant la case $(0,0)$ à la case (i,j).

a. Pour quels cas le calcul de $f(i,j)$ est-il direct ?

Si $i=0$ ou $j=0$ alors le calcul de $f(i,j)$ est direct : en effet, il n'y a qu'un chemin possible pour relier la case d'indice $(0,0)$ à la case (i,j) (route horizontale si $i=0$ et verticale si $j=0$). Ainsi :

- $\forall i \in [\![0,n-1]\!],\ f(i,0) = \sum_{k=0}^{i} m(k,0)$ (d'où $f(i+1,0) = m(i+1,0) + f(i,0)$) ;
- $\forall j \in [\![0,p-1]\!],\ f(0,j) = \sum_{k=0}^{j} m(0,j)$ (d'où $f(0,j+1) = m(0,j+1) + f(0,j)$).

b. Trouver une relation reliant $f(i,j)$, $f(i-1,j)$ et $f(i,j-1)$.

Si i et j sont différents de 0 alors il y a deux moyens pour construire un chemin optimal :

- un chemin optimal de l'origine vers la case $(i-1,j)$, puis un déplacement vertical $(i-1,j) \to (i,j)$ – le poids d'une telle route vaut $f(i-1,j) + m_{i,j}$;
- un chemin optimal de l'origine vers la case $(i,j-1)$, puis un déplacement horizontal $(i,j-1) \to (i,j)$ – le poids d'une telle route vaut $f(i,j-1) + m_{i,j}$.

On choisit le chemin de poids minimal donc :

$$f(i,j) = m_{i,j} + \min\big(f(i-1,j), f(i,j-1)\big).$$

2. Programmer une version itérative pour trouver le poids minimal pour la route reliant les cases $(0,0)$ et $(n-1,p-1)$.

```
def damier(m):
  n, p = len(m), len(m[0])
  f = [[0 for j in range(p)] for i in range(n)]
  f[0][0] = m[0][0]
  for i in range(n-1):
      f[i+1][0] = m[i+1][0] + f[i][0]
  for j in range(p-1):
      f[0][j+1] = m[0][j+1] + f[0][j]
  for i in range(1, n):
      for j in range(1, p):
          f[i][j] = m[i][j] + min(f[i-1][j], f[i][j-1])
  return f[n-1][p-1]
```

Par exemple, avec :

```
D = [[6, 3, 6, 4], [3, 1, 2, 0], [2, 1, 5, 2], [6, 1, 4, 4], [2, 2, 6, 2]]
```

on obtient comme tableau :

6	9	15	19
9	10	12	12
11	11	16	14
17	12	16	18
19	14	20	20

ce qui donne :

```
>>> damier(D)
20
```

3. Adapter la fonction précédente pour obtenir également un chemin de poids minimal.

```
def damier_bis(m):
    n, p = len(m), len(m[0])
    f = [[0 for j in range(p)] for i in range(n)]
    f[0][0] = m[0][0]
    for i in range(n-1):
        f[i+1][0] = m[i+1][0] + f[i][0]
    for j in range(p-1):
        f[0][j+1] = m[0][j+1] + f[0][j]
    for i in range(1, n):
        for j in range(1, p):
            f[i][j] = m[i][j] + min(f[i-1][j], f[i][j-1])
    c = [(n, p)]
    i, j = n-1, p-1
    while i>0 and j>0:
        if f[i][j] - m[i][j] == f[i-1][j]:
            i -= 1
        else:
            j -= 1
        c.append((i, j))
    if i == 0:
        while j>0:
            j -= 1
            c.append((i,j))
    else:
        while i>0:
            i -= 1
            c.append((i,j))

    c.reverse()

    return f[n-1][p-1], c
```

Toujours avec le même exemple, on a :

```
>>> damier_bis(D)
(20, [(0, 0), (0, 1), (1, 1), (1, 2), (1, 3), (2, 3), (3, 3), (5, 4)])
```

4. Proposer une version récursive.

Voici une première version ; les cas de base correspondent à la première ligne et à la première colonne du tableau.

```
def damier_rec(m, a, b):
    if a == 0 and b == 0:
        return m[0][0]
    if a == 0:
        return m[0][b] + damier_rec(m, 0, b-1)
    if b == 0:
        return m[a][0] + damier_rec(m, a-1, 0)
    gauche = damier_rec(m, a-1, b)
    haut = damier_rec(m, a, b-1)
    return m[a][b] + min(gauche, haut)
```

On a alors :

```
>>> h, l = len(D), len(D[0])
>>> damier_rec(D, h-1, l-1)
20
```

On peut proposer une version avec mémoïsation à l'aide d'un dictionnaire :

```
cases = {}

def damier_rec_memo(m, a, b):
    if (a, b) not in cases:
        if a == 0 and b == 0:
            res = m[0][0]
        elif a == 0:
            res = m[0][b] + damier_rec_memo(m, 0, b-1)
        elif b == 0:
            res = m[a][0] + damier_rec_memo(m, a-1, 0)
        else:
            gauche = damier_rec_memo(m, a-1, b)
            haut = damier_rec_memo(m, a, b-1)
            res = m[a][b] + min(gauche, haut)
        cases[(a, b)] = res
    return cases[(a,b)]
```

Et la même chose avec une fonction qui «contient tout» :

```
def damier_rec_memo_main(m):
    n, p = len(m), len(m[0])
    cases = {}
    def aux(a, b):
        if (a, b) not in cases:
            if a == 0 and b == 0:
                res = m[0][0]
            elif a == 0:
                res = m[0][b] + aux(0, b-1)
            elif b == 0:
                res = m[a][0] + aux(a-1, 0)
            else:
                gauche = aux(a-1, b)
                haut = aux(a, b-1)
                res = m[a][b] + min(gauche, haut)
            cases[(a, b)] = res
        return cases[(a,b)]
    return aux(n-1, p-1)
```

Il suffit alors d'écrire :

```
>>> damier_rec_memo_main(D)
20
```