CHAPITRE

2

STRATÉGIES DE PROGRAMMATION

I - Quelques exemples de problèmes d'optimisation

I.1 - Acheter puis vendre !

On cherche à calculer le gain maximum possible à la Bourse sur une action pendant une période de n jours, en ne faisant qu'une opération d'achat ou de vente d'une seule action. On suppose que les cours quotidiens de cette action sont enregistrés dans une liste d'entiers naturels ($a_i \in \mathbb{N}$) de n éléments ($0 \leqslant i < n$).

Le *gain maximum* est la quantité suivante :

$$\text{gain} = \max_{0 \leqslant i \leqslant j < n} (a_j - a_i).$$

Question 1

En suivant textuellement la définition du gain, écrire une fonction `gain1(a)` qui renvoie, avec une complexité quadratique (par rapport à n), le gain maximal possible sur le cours représenté par le tableau a.

Pour tout $i \in [\![0, n-1]\!]$, on définit le gain courant maximum GC_i comme le gain maximum possible obtenu en vendant son action au temps i, c'est-à-dire :

$$\mathrm{GC}_i = \max_{0 \leqslant k \leqslant i} (a_i - a_k).$$

On vérifie alors que l'on a la relation de récurrence :

$$\mathrm{GC}_i = \max\left\{0 \ ; \ \mathrm{GC}_{i-1} + a_i - a_{i-1}\right\}.$$

Question 2

Écrire une fonction `gain2(a)` qui renvoie, avec une complexité linéaire (par rapport à n), le gain maximum possible sur le cours représenté par le tableau a.

Exercice 1

Modifier les deux fonctions précédentes pour qu'elles renvoient également les deux dates i et j d'achat et de vente de l'action permettant d'obtenir le gain maximum sur le tableau a (avec $j - i$ minimum).

I.2 - Rendu de monnaie et stratégie glouton

Un *algorithme glouton* est un algorithme qui suit le principe de faire, étape par étape, un choix localement optimal, dans l'espoir d'obtenir un résultat global optimal. Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une heuristique gloutonne.

La méthode « usuelle » pour rendre la monnaie est celle de l'algorithme glouton : tant qu'il reste quelque chose à rendre, on choisit la plus grosse pièce que l'on puisse rendre (sans rendre trop). C'est un algorithme très simple et rapide, et on appelle canonique un système de pièces pour lequel cet algorithme donne une solution optimale quelle que soit la valeur à rendre.

On note $\mathrm{S} = [p_0, p_1, \cdots p_i, \cdots p_{n-1}]$ un système de n pièces ou billets de valeurs p_i rangées par ordre croissant. On considère une somme r à rendre à l'aide du système S (il n'y a pas de limitation du nombre de chaque pièce). Une façon de constituer cette somme est donnée par le n-uplet $(m_0, m_1, \cdots m_i, \cdots m_{n-1})$ tel que :

$$r = \sum_{i=0}^{n-1} m_i \cdot p_i$$

On cherche à minimiser le nombre total $\displaystyle\sum_{i=0}^{n-1} m_i$ de pièces rendues.

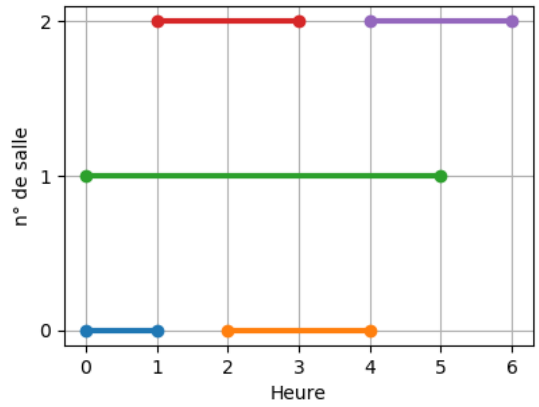
Exercice 2

Écrire la fonction `monnaie_a_rendre (somme_a_rendre, systeme)` qui renvoie une liste des nombres de pièces à rendre la plus proche de `somme_a_rendre` sans la dépasser, dans le système de monnaie `systeme`.
Par exemple, on doit avoir :

```
>>> monnaie_a_rendre(8, [1, 4, 6])
[2, 0, 1]
>>> monnaie_a_rendre(49, [1, 3, 6, 12, 24, 30])
[1, 0, 1, 1, 0, 1]
>>> monnaie_a_rendre(11, [2, 4, 6])
[0, 1, 1]
```

I.3 - Gestion d'un emploi du temps de salles

L'objectif est de construire un emploi du temps d'allocation de salles.
Un emploi du temps est ici une liste de plannings de salles, chaque planning étant lui-même une liste de plages horaires, chaque plage étant une liste `[debut,fin]` de deux entiers naturels tels que debut < fin.



→ voir le document projeté (d'après le cours de M. Grenet de l'Université de Montpellier).

I.4 - Problème du sac à dos

On dispose d'un sac à dos dont la charge utile est limitée à un poids maximal $p_{\max}$ et de n objets $x_0, x_1, \ldots, x_{n-1}$ possédant chacun un poids p_i et une valeur v_i. Le but est de remplir le sac en emportant la valeur maximale sans dépasser le poids limite.

◇ Une première possibilité consiste à utiliser une stratégie de «force brute» : on liste toutes les possibilités et on prend la meilleure. S'il y a n objets, il y a alors 2^n possibilité. C'est inutilisable en pratique.

◇ Une deuxième possibilité consiste à utiliser une stratégie «glouton». Par exemple en classant les objets selon le rapport $\frac{\text{valeur}}{\text{poids}}$ et en donnant systématiquement la priorité à l'objet présentant ce meilleur rapport.

◇ Une autre stratégie, dite de «programmation dynamique» consiste à chercher une relation de récurrence comme dans le premier exemple.

On note $f(k,p)$ la valeur maximale obtenue avec les objets $x_0, \ldots, x_k$ ne dépassant pas le poids p (on recherche donc ici $f(n-1, p_{\max})$. On cherche une relation de récurrence vérifiée par la fonction f.

- Si $k = 0$ alors il y a deux cas à distinguer pour $f(0,p)$:
  - ◇ si $p_0 > p$ alors $f(0,p) = 0$,
  - ◇ si $p_0 \leqslant p$ alors $f(0,p) = v_0$;
- pour $k \neq 0$, l'idée est de relier $f(k,p)$ à $f(k-1,p')$:
  - ◇ si $p_k > p$ alors $f(k,p) = f(k-1,p)$,
  - ◇ si $p_k \leqslant p$ alors $f(k,p) = \max\Big(f(k-1,p), v_k + f(k-1,p-p_k)\Big)$.

II - Rappels sur les algorithmes dichotomiques

II.1 - Recherche dans une liste triée

II.1.a - Le principe de la recherche

On considère une liste a de n nombres, triés par *ordre croissant*. Un nombre x quelconque étant donné, on cherche à savoir si ce nombre est dans la liste a.

On utilise la méthode suivante : on compare x avec l'élément qui est au milieu de la liste dans laquelle on le cherche. Plus précisément si on cherche x dans la sous-liste dont les indices constituent l'intervalle $[\![m,n]\!]$, on comparera x à l'élément d'indice `(n - m)//2`.
Si x lui est égal, alors x appartient à la liste et on s'arrête là (on a trouvé x à la position `(n - m)//2`).
Sinon, si x lui est supérieur, alors comme tous les éléments placés avant $a[(n-m)//2]$ sont plus petits que $a[(n-m)//2]$, x leur est aussi supérieur. On continue alors en cherchant x parmi les éléments situés strictement après $a[(n-m)//2]$.
On procéderait de la même manière si on avait x inférieur à $a[(n-m)//2]$, en cherchant parmi les éléments situés strictement avant $a[(n-m)//2]$.
En procédant ainsi on cherche x dans des listes de plus en plus petites (le nombre d'éléments des listes successives dans lesquelles on cherche x est une suite strictement décroissante). Ainsi, soit on trouve x, soit on finit par le chercher dans une liste vide et la conclusion est alors immédiate : x n'est pas dans la liste.

II.1.b - Une version itérative

```
def recherche_dicho(x, a):
    """
    Entrée : x (float ou int)
             a (list) une liste de nombres (float ou int)
             supposée triée par ordre croissant
    Sortie : l'indice d'une occurrence de x dans a si x est dans a
             sinon None
    Méthode par dichotomie (séparation de la liste de recherche en
    deux parties égales à une unité près).
    """

    m = 0
    n = len(a) - 1

    while m <= n:
        i0 = (m + n)//2
        if x == a[i0]:
            return i0
        elif x > a[i0]:
            m = i0 + 1
        else:
            n = i0 - 1

    return None
```

II.1.c - L'approche récursive

On reprend l'algorithme de recherche dichotomique et on en propose une version récursive :

```
def dicho1(x, a):
    """
    Entrée : x (float ou int)
             a (list) une liste de nombres (float ou int)
             supposée triée par ordre croissant
    Sortie : True si x est dans a sinon False (bool)
    """

    if len(a) == 0:
        return False

    i0 = len(a)//2
    if x == a[i0]:
        return True
    elif x > a[i0]:
       return dicho1(x, a[i0 + 1:])
    else:
        return dicho1(x, a[:i0])
```

Quel est le problème dans l'implémentation ci-dessus ? Quel est l'avantage de celui ci-dessous ?

```
def dicho2(x, a, m, n):
    """
    Entrée : x (float ou int)
             a (list) une liste de nombres (float ou int)
             supposée triée par ordre croissant
    Sortie : True si x est dans a sinon False (bool)
    Recherche dichotomique récursive.
    Implémentation optimisée conservant la complexité en O(ln(n))
    """

    if m >= n:
        return False

    i0 = (m + n)//2

    if x == a[i0]:
        return True
    elif x > a[i0]:
        return dicho2(x, a, i0 + 1, n)
    else:
        return dicho2(x, a, m, i0)
```

II.2 - Exponentiation rapide

Au lieu d'écrire que $x^n = x \times x^{n-1}$, on va décomposer le calcul suivant deux cas principaux :

$$x^n = \begin{cases} x & \text{si } n = 1 \\ (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x \cdot (x^2)^{\frac{n-1}{2}} & \text{si } n \text{ est impair} \end{cases}$$

Les deuxièmes et troisièmes cas ci-dessus sont les cas principaux car ils peuvent à nouveau se décomposer en d'autres cas. Le premier est un cas d'arrêt de l'algorithme.

```python
def expor(x, n):
    """
    Entrée : x (float), n (int) strictement positif
    Sortie : x**n
    Algorithme d'exponentiation rapide
    """

    p = 1
    y = x
    m = n

    while m != 1:
        if m % 2 == 1:
            p = p*y
        y = y*y
        m = m//2

    return p*y
```

```python
def expor_rec(x, n):
    """
    Entrée : x (float), n (int)
    Sortie : x**n (float)
    Calcul récursif d'exponetiation rapide.
    """

    if n == 0: # cas de base
        return 1
    elif n%2 == 0:
        return expor_rec(x*x, n//2)
    else:
        return x*expor_rec(x*x, n//2)
```

II.3 - Autres exemples de stratégie «diviser pour régner»

La méthode «diviser pour régner» consiste à diviser un problème de taille n en deux sous-problèmes de taille $\frac{n}{2}$ (ou à peu près), puis de résoudre séparément ces problèmes avant de rassembler les résultats pour obtenir la réponse finale.
L'idée, pour avoir des gains de performances importants, est de recommencer récursivement cette méthode sur les sous-problèmes jusqu'à obtenir des cas assez simples pouvant être traités de manière directe :

Algorithme : fonction DR – Diviser pour régner

Données : x
début
  si x est suffisament petit ou simple **alors**
    retourner directement le résultat
  sinon
    Décomposer x en 2 : x_1 et x_2
    $y_1 \leftarrow \text{DR}(x_1)$
    $y_2 \leftarrow \text{DR}(x_2)$
    Rassembler y_1 et y_2 pour trouver la solution y
    retourner y

Il est parfois nécessaire de diviser le problème initial en plus que deux sous-problèmes et il se peut que les sous-problèmes soient choisis de tailles différentes.

Si l'on note C(n) la complexité requise pour traiter un problème de taille n avec la méthode «diviser pour régner» alors :

$$\mathrm{C}(n) = a\mathrm{C}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + b\mathrm{C}\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n);$$

où :

- a et b sont des coefficients qui peuvent varier suivant les algorithmes. Par exemple,
  - ◇ $a = b = 1$ s'il faut traiter les deux sous-problèmes de manière indépendante,
  - ◇ $a = 1$ et $b = 0$ s'il ne faut s'occuper que du premier sous-problème,
  - ◇ $a + b = 1$ s'il ne faut s'occuper que d'un sous-problème, sans savoir lequel *a priori* ;

  On a toujours $a + b \geqslant 1$.
- $f(n)$ représente le coût pour séparer le problème et recombiner les deux résultats.

II.3.a - L'exemple de la multiplication des grands entiers

On veut multiplier deux entiers comportant chacun n chiffres : $x = a_{n-1}\dots a_2a_1a_0$ et $y = b_{n-1}\dots b_2b_1b_0$. En posant la multiplication comme appris à l'école primaire :

$$\begin{array}{rl} & a_{n-1}\dots a_2a_1a_0 \\ \times & b_{n-1}\dots b_2b_1b_0 \\ \hline \end{array}$$

on s'aperçoit que l'on va effectuer n^2 multiplications simples $(a_i \times b_j)$ – et aussi quelques additions (celles-ci étant beaucoup plus simples pour le processeur, on les laisse de côté dans l'évaluation de la complexité).
On cherche maintenant à découper le problème. Si n est pair (*i.e.* $n = 2m$), on peut écrire :

$$\begin{cases} x = 10^m a + b \\ y = 10^m c + d, \end{cases} \quad \text{avec } a, b, c, d \text{ quatre nombres entiers à } m \text{ chiffres.}$$

Le produit xy s'écrit alors :

$$10^{2m}ac + 10^m(ad + bc) + bd.$$

On doit donc évaluer quatre produits : ac, ad, bc, bd ; soit $4 \times m^2 = n^2$ multiplications simples. On n'y gagne rien. Mais en étant plus malin, on peut poser :

$$p = ac, \qquad q = bd, \qquad r = (a + b)(c + d),$$

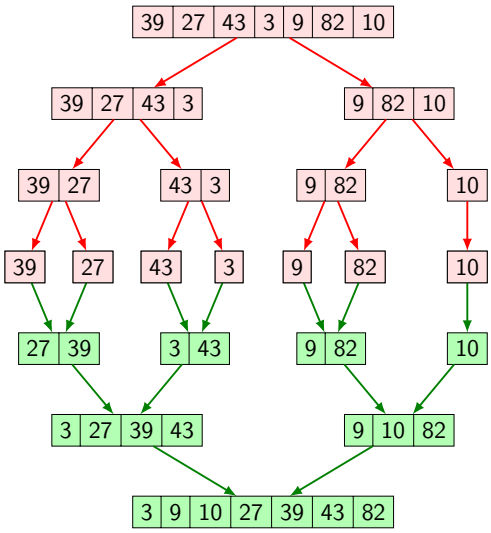
on a alors :

$$xy = 10^{2m}p + 10^m(r - p - q) + q.$$

Il n'y a plus que trois produits à évaluer ! On a gagné 25% de temps de calcul ! (Cette affirmation est un peu rapide, car il y a des calculs supplémentaires pour découper les nombres et rassembler les résultats, mais l'idée est là).
De plus, on peut recommencer ce découpage pour calculer ces trois produits… On arrive en définitive à une complexité en $\mathrm{O}\left(n^{\log_2 3}\right)$.

II.3.b - L'exemple du tri fusion

◇ L'idée du tri fusion est de séparer le tableau en deux sous-tableaux, de trier (récursivement) ces deux tableaux, puis de fusionner ces tableaux triés pour obtenir le résultat final.



On fait donc les étapes suivantes :

- on coupe la liste en deux sous-listes de longueurs (à peu près) égales ;
- on trie les deux sous-listes ;
- on fusionne les deux sous-listes triées en une liste triée.

◇ Programmons cela en Python. On commence par la fonction fusionnant les deux sous-listes triées :

```
def fusion(s, t):
    """ entrée : deux tableaux triés
        sortie : un tableau trié constitué par les éléments de s et t
    """
    l = []
    i, j = 0, 0                     # indices de parcours des deux listes
    while i < len(s) or j < len(t):
        if i == len(s):             # liste s intégralement traitée
            l.append(t[j])
            j += 1
        elif j == len(t):           # liste t intégralement traitée
            l.append(s[i])
            i += 1
        else:                       # listes s et t encore exploitables
            if s[i] < t[j]:
                l.append(s[i])
                i += 1
            else:
                l.append(t[j])
                j += 1
    return(l)
```

On utilise deux indices `i` et `j` et, à chaque tour de boucle, on fait avancer soit l'un soit l'autre indice. L'algorithme termine car `i+j` est strictement croissant.

```
>>> a = [1,3,89,102]
>>> b = [0,2,6,8,9,10,42]
>>> fusion(a, b)
[0, 1, 2, 3, 6, 8, 9, 10, 42, 89, 102]
```

On programme alors la fonction de tri :

```
def triFusion(L):
    n = len(L)
    if n <= 1:      # on finit par considérer une liste vide ou à 1 élément
        return L
    else:
        m = n//2
        l1 = triFusion(L[:m])
        l2 = triFusion(L[m:])
        return(fusion(l1, l2))
```

Voici une autre version, quelle différence y a-t-il ?

```
def fusionne(T, debut, milieu, fin):
    aux = [0]*(fin-debut+1)
    p1 , p2 = debut, milieu+1
    for i in range(fin-debut+1):
        if p2 == fin+1 or (p1 <= milieu and T[p1] <= T[p2]) :
            aux[i] = T[p1]
            p1 += 1
        else:
            aux[i] = T[p2]
            p2 +=1
    for i in range(debut, fin+1):
        T[i] = aux[i-debut]

def tri(T, debut, fin):
    if debut < fin:
        milieu = (debut + fin)//2
        tri(T, debut, milieu)
        tri(T, milieu+1, fin)
        fusionne(T, debut, milieu, fin)

def trifusion(T):
    tri(T, 0, len(T)-1)
```

Pour étudier la complexité dans le pire des cas de cet algorithme, on se focalise sur le nombre de comparaisons nécessaires.
Pour la fonction `fusion` : les longueurs des deux tableaux à fusionner ne diffèrent qu'au plus de 1 et le pire des cas est celui où il faut intercaler alternativement des éléments de chacun des sous-tableaux. Si la taille du tableau fusionné est n, cela peut donc nécessiter $(n-1)$ comparaisons entre éléments du tableau.
La relation de récurrence sur la complexité $C(n)$ s'écrit pour tout $n \in \mathbb{N}^*$:

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1).$$

On prouve alors que $C(n) = O(n \ln n)$.

III - Programmation dynamique

III.1 - L'exemple du calcul des coefficients binomiaux

◇ La formule du triangle de Pascal permet le calcul de $\binom{n}{p}$ à l'aide de deux calculs plus simples :

$$\forall (n,p) \in (\mathbb{N}^*)^2, \quad \binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}.$$

Cette formule suggère une programmation récursive du calcul :

```
def binomial(n, p):
    """ entrée : deux entiers n et p
        sortie : entier égal au coefficient p parmi n
    """
    if p < 0 or p > n:
        return 0
    if p == 0 or p == n:
        return 1
    return binomial(n-1, p-1) + binomial(n-1, p)
```

On constate un problème de vitesse de calcul :

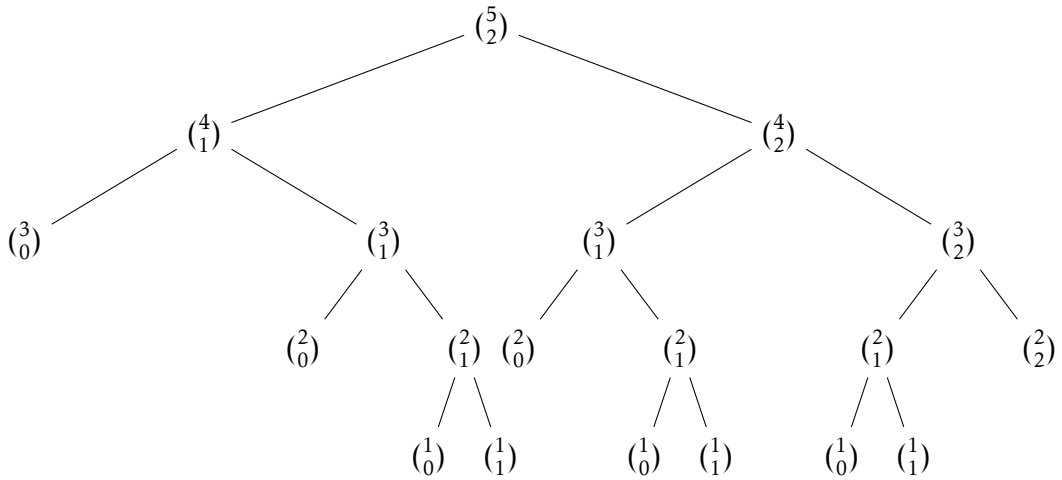
```
>>> from time import time
>>> t = time(); binomial(22, 11); time()-t
705432
0.3385660648345947
>>> t = time(); binomial(25, 13); time()-t
5200300
2.622504711151123
```

Comme le résultat est obtenu en additionnant des 1, la complexité pour calculer $\binom{n}{p}$ est en $O\left(\binom{n}{p}\right)$.

Exemple

Justifions que, par cette méthode, le calcul de $\binom{2n}{n}$ est de complexité exponentielle en n.

L'arbre suivant illustre le fait que l'on calcule plusieurs fois les mêmes coefficients.



Pour calculer $\binom{5}{2}$, on va donc évaluer $\binom{4}{1}$ et $\binom{4}{2}$; et pour trouver ces deux termes, on va à chaque fois avoir besoin du coefficient $\binom{3}{1}$.

La fonction ramène le calcul d'un coefficient binomial au calcul de deux coefficients inférieurs cependant, et contrairement à ce qu'il se passe dans une approche du type *diviser pour régner*, ici les deux sous-problèmes ne sont pas indépendants.

Ces calculs redondants ne sont pas anecdotiques : pour calculer $\binom{20}{10}$, le coefficient $\binom{2}{1}$ va être évalué presque 50 000 fois ! Il faut donc mettre en place un système pour ne pas perdre du temps à recalculer ce qui a déjà été déterminé.

◇ On va palier ce problème de temps avec une autre stratégie. On commence par résoudre les «plus petits» problèmes et on exploite leurs solutions pour résoudre des problèmes de plus en plus grands. Concrètement, on a va utiliser un tableau de nombres initialisé avec des zéros et on va calculer les coefficients de proche en proche.

```
def binomial_iter(n, p):
    t = [[0 for _ in range(p+1)] for _ in range(n+1)]
    for k in range(n+1):
        t[k][0] = 1
        if k <= p:
            t[k][k] = 1
    for i in range(2, n+1):
        for j in range(1, p+1):
            t[i][j] = t[i-1][j-1] + t[i-1][j]
    return t[n][p]
```

La complexité pour le calcul de $\binom{n}{p}$ est désormais en $O(np)$ ce qui améliore considérablement la situation ! Il y a également un coût en espace puisque l'on crée un tableau de taille $(n+1)\times(p+1)$. Ce dernier point pourrait néanmoins être amélioré en ne conservant qu'une ligne à chaque étape.

Il y a deux autres inconvénients :

- on calcule de nombreux coefficients pour rien ;
- on perd en lisibilité par rapport à la version récursive.

On peut concilier les deux aspects (la concision de la programmation récursive et l'efficacité du procédé itératif) en utilisant une méthode de ***mémoïsation*** : on va stocker les valeurs déjà calculées et ne faire un appel récursif que lorsque le calcul est utile.
Pour stocker les valeurs déjà calculées, on peut utiliser à nouveau une liste de listes mais la structure de dictionnaire est particulièrement adaptée puisqu'elle permet d'utiliser des clés de la forme `(n,p)`.

```
d = {}

def binomial_memo(n, p):
    if (n, p) not in d:
        if p < 0 or p > n:
            res = 0
        elif p == 0 or p == n:
            res = 1
        else:
            res = binomial_memo(n-1, p-1) + binomial_memo(n-1, p)
        d[(n, p)] = res
    return d[(n, p)]
```

Voici une autre version pour éviter la variable `d` en dehors de la fonction :

```
def binomial_memo_bis(n, p):
    d = {}
    def aux(m, q):
        if (m, q) not in d:
            if q < 0 or q > m:
                res = 0
            elif q == 0 or q == m:
                res = 1
            else:
                res = binomial_memo(m-1, q-1) + binomial_memo(m-1, q)
            d[(m, q)] = res
        return d[(m, q)]
    return aux(n, p)
```

III.2 - Les principes de la programmation dynamique

L'idée essentielle est que toute étape d'une solution optimale est elle-même optimale, c'est le *principe d'optimalité de Bellman*.

La résolution d'un problème par programmation dynamique consiste donc déterminer un algorithme fondé sur une ***relation de récurrence*** puis à programmer en stockant les résultats des sous-problèmes afin d'éviter des calculs inutiles.

Pour cette programmation, il y a deux stratégies.

- Une démarche itérative.
  Il s'agit de :
  - trouver une relation de récurrence reliant la solution du problème aux solutions des sous-problèmes ;
  - définir un tableau (ou un dictionnaire) de taille adéquate et l'initialiser avec les valeurs triviales ;
  - résoudre les sous-problèmes de taille de plus en plus grande (boucles utilisant les formules de récurrence) ;
  - lire la solution dans le tableau (ou la récupérer si la lecture n'est pas directe).
- Une démarche récursive avec mémoïsation.
  À chaque appel récursif, on vérifie si la valeur est déjà connue et, sinon, on la calcule et on la stocke.

III.3 - L'exemple du problème du sac à dos

On dispose d'un sac à dos dont la charge utile est limitée à un poids maximal $p_{\max}$ et de n objets $x_0, x_1, \ldots, x_{n-1}$ possédant chacun un poids p_i et une valeur v_i. Le but est de remplir le sac en emportant la valeur maximale sans dépasser le poids limite.

On a déjà évoqué l'idée de procéder par «force brute» : on liste toutes les possibilités et on prend la meilleure. S'il y a n objets, il y a alors 2^n possibilités. C'est inutilisable en pratique.

On a également évoqué l'idée d'utiliser une stratégie «glouton», par exemple en classant les objets selon le rapport $\frac{\text{valeur}}{\text{poids}}$ et en donnant systématiquement la priorité à l'objet disponible présentant ce meilleur rapport.

On s'intéresse dans ce qui suit à la mise en place d'une stratégie de programmation dynamique.

On note $f(k,p)$ la valeur maximale obtenue avec les objets $x_0, \ldots, x_k$ en ne dépassant pas le poids p. Le but est donc d'obtenir la valeur $f(n-1, p_{\max})$.
On cherche une relation de récurrence vérifiée par la fonction f.

- Si $k = 0$ alors il y a deux cas à distinguer pour $f(0,p)$:
  - si $p_0 > p$ alors $f(0,p) = 0$,
  - si $p_0 \leqslant p$ alors $f(0,p) = v_0$;
- pour $k \neq 0$, l'idée est de relier $f(k,p)$ à $f(k-1,p')$:
  - si $p_k > p$ alors $f(k,p) = f(k-1,p)$,
  - si $p_k \leqslant p$ alors $f(k,p) = \max\Big(f(k-1,p), v_k + f(k-1,p-p_k)\Big)$.

III.3.a - Stratégie itérative (ascendante)

```
def sac_a_dos(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 = [[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)

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

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(test, 14)
24
```

Cherchons maintenant à obtenir, non seulement la valeur maximale, mais également la liste des objets concernés.

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 l'on en choisit une.

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.3.b - 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)
```

III.4 - 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.
Par exemple, avec le damier suivant, le chemin de poids minimal est représenté avec les cases grisées :

3	9	4	3	2	6	7
1	6	9	1	3	5	8
9	2	5	4	6	4	9
6	2	8	1	8	2	7
3	6	3	7	8	9	7

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.

Exercice 3

La *distance d'édition*, ou *distance de Levenshtein*, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à une autre.
Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes :

— suppression de la lettre `i` : `polynomial` → `polynomal` ;

— remplacement du `n` par un `g` : `polynomal` → `polygomal` ;

— remplacement du `m` par un `n` : `polygomal` → `polygonal` ;

Donc la distance d'édition entre ces deux mots est au plus égale à 3, et on peut aisément se convaincre qu'il n'est pas possible de faire mieux.

Nous allons calculer la distance d'édition entre deux mots $a = a_1a_2\dots a_m$ et $b = b_1b_2\dots b_n$ en généralisant le problème, c'est-à-dire en définissant la distance d'édition $d(i,j)$ entre les mots $a_1a_2\dots a_i$ et $b_1b_2\dots b_j$.

Les cas suivants peuvent se rencontrer pour passer, de façon optimale, de $a_1a_2\dots a_i$ à $b_1b_2\dots b_j$:

— a_i a été supprimé : il y a donc une étape de plus que pour passer de $a_1a_2\dots a_{i-1}$ à $b_1b_2\dots b_j$ d'où
$$d(i,j) = d(i-1,j)+1;$$

— b_j a été ajouté : il y a donc une étape de plus que pour passer de $a_1a_2\dots a_i$ à $b_1b_2\dots b_{j-1}$ d'où
$$d(i,j) = d(i,j-1)+1;$$

— a_i a été remplacé par b_j : il y a donc une étape de plus que pour passer de $a_1a_2\dots a_{i-1}$ à $b_1b_2\dots b_{j-1}$ d'où
$$d(i,j) = d(i-1,j-1)+1$$
sauf si $a_i = b_j$ auquel cas il y a autant d'étapes que pour passer de $a_1a_2\dots a_{i-1}$ à $b_1b_2\dots b_{j-1}$ d'où
$$d(i,j) = d(i-1,j-1).$$

On en déduit :

$$d(i,j) = \begin{cases} \min(d(i-1,j)+1,\ d(i,j-1)+1,\ d(i-1,j-1)+1) & \text{si } a_i \neq b_j \\ \min(d(i-1,j)+1,\ d(i,j-1)+1,\ d(i-1,j-1)) & \text{si } a_i = b_j \end{cases}$$

Les conditions initiales sont : $d(i,0) = i$ et $d(0,j) = j$.

Programmer en python la fonction `dist(mot1,mot2)` donnant la distance d'édition entre `mot1` et `mot2` puis préciser sa complexité.