TP — Programmation dynamique — Corrigé

Ce TP est une collection de six exercices **indépendants**.

Consignes : on n'utilise que les fonctions et méthodes de l'annexe du programme (en particulier ni `min`, ni `max`, ni `sum`, ni `sorted`), les fonctions ne modifient pas les listes qu'elles reçoivent en argument, et les complexités sont données avec la notation O.

Exécuter d'abord la première cellule de code (le module `dessins.py` doit se trouver dans le même dossier que ce notebook). Elle définit `INFINI = float("inf")` et la fonction `minimum3(a, b, c)`, qui remplace `min` pour trois valeurs.

Sommaire

— 1. Plus grand carré de 1 dans une matrice (Q1 à Q6)
— 2. Distance de Levenshtein (Q7 à Q14)
— 3. Produit d'une chaîne de matrices (Q15 à Q21)
— 4. Sac à dos (Q22 à Q28)
— 5. Bonus : la plus longue descente (Q29 à Q31)
— 6. Bonus : le jeu des pièces (Q32 et Q33)

1. Plus grand carré de 1 dans une matrice

Une matrice de 0 et de 1 à n lignes et p colonnes est représentée par une liste `m` de n listes de p entiers. La case (x, y) est celle de la ligne x et de la colonne y, la case $(0, 0)$ étant en haut à gauche. Le **carré de côté k et de coin supérieur gauche** (x, y) est formé des cases $(x + a, y + b)$ pour $0 \leq a < k$ et $0 \leq b < k$.

On cherche la **taille** (le côté) du plus grand carré ne contenant que des 1. Pour la matrice `M` ci-dessous, c'est 3.

```
M = [[0, 1, 1, 0, 1, 1, 1, 0],
     [1, 1, 1, 0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0, 1, 1, 1],
     [1, 1, 1, 1, 0, 0, 1, 1],
     [0, 1, 1, 0, 1, 1, 1, 1]]
dessiner_matrice(M)
```

matrice 6 x 8, plus grand carré de 1 : à trouver

0	1	1	0	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	0
0	1	1	1	0	1	1	1
1	1	1	1	0	0	1	1
0	1	1	0	1	1	1	1

À la main : recherche exhaustive

Question 1. Écrire une fonction `est_carre_de_uns(m, x, y, k)` qui prend en arguments une matrice `m`, les coordonnées `(x, y)` d'une case et un entier `k` ≥ 1, tels que le carré de côté `k` et de coin supérieur gauche `(x, y)` soit inclus dans la matrice, et renvoie `True` si ce carré ne contient que des 1, `False` sinon. Quelle est sa complexité ?

```
def est_carre_de_uns(m, x, y, k):
    for i in range(x, x + k):
        for j in range(y, y + k):
            if m[i][j] != 1:
                return False
    return True

# Au plus k^2 cases examinées : O(k^2).
```

```
# Tests : chaque ligne doit afficher True.
M = [[0, 1, 1, 0, 1, 1, 1, 0],
     [1, 1, 1, 0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0, 1, 1, 1],
     [1, 1, 1, 1, 0, 0, 1, 1],
     [0, 1, 1, 0, 1, 1, 1, 1]]

print(est_carre_de_uns(M, 0, 4, 3), est_carre_de_uns(M, 2, 1, 3))
print(not est_carre_de_uns(M, 1, 3, 2), not est_carre_de_uns(M, 2, 3, 3))
print(est_carre_de_uns(M, 1, 0, 1), not est_carre_de_uns(M, 0, 0, 1))
print(est_carre_de_uns([[1]], 0, 0, 1), not est_carre_de_uns([[0]], 0, 0, 1))
print(not est_carre_de_uns([[1, 0], [1, 1]], 0, 0, 2))
```

```
True True
True True
True True
True True
True
```

La fonction suivante essaie tous les carrés, par côté croissant.

```
def plus_grand_carre_naif(m):
    n, p = len(m), len(m[0])
    k = 0                       # il existe un carré de 1 de côté k
    trouve = True
    while trouve and k < n and k < p:
        trouve = False          # cherche un carré de 1 de côté k + 1
        for x in range(n - k):
            for y in range(p - k):
                if est_carre_de_uns(m, x, y, k + 1):
                    trouve = True
        if trouve:
            k = k + 1
    return k

print(plus_grand_carre_naif([[0, 1, 1], [1, 1, 1], [1, 1, 1]]))
```

```
2
```

Question 2. Quelle est la complexité de `plus_grand_carre_naif` pour une matrice de taille $n \times n$?

Proposer des améliorations possibles de la fonction `plus_grand_carre_naif`.

Réponse. Au plus n valeurs de k, pour chacune au plus n^2 carrés, chacun testé en $O(k^2) = O(n^2)$: la complexité est $O(n^5)$.

On pourrait améliorer `plus_grand_carre_naif` de plusieurs façons : - s'arrêter dès qu'on trouve un carré de côté k (au lieu de continuer à tester tous les carrés de côté k) (change la complexité dans le meilleur des cas mais pas dans le pire des cas), - tester par valeurs décroissantes de k (au lieu de croissantes) (change la complexité dans le meilleur des cas mais pas dans le pire des cas), - plutôt que d'augmenter (ou diminuer) séquentiellement k, procéder par *dichotomie* sur k, la complexité devient alors $O(n^4 \ln(n))$.

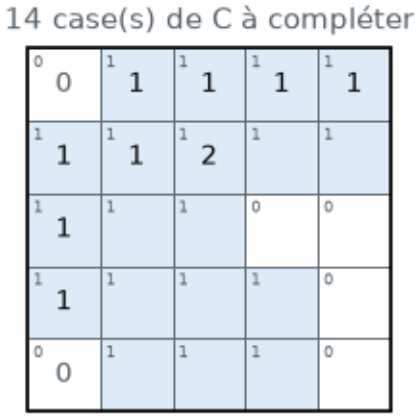
Sous-problèmes et récurrence

Pour chaque case (x, y), on note $C[x][y]$ la taille du plus grand carré de 1 dont (x, y) est le **coin inférieur droit** (et $C[x][y] = 0$ si $m[x][y] = 0$). La réponse est la plus grande valeur du tableau C : il y a un sous-problème par case.

Question 3. Compléter à la main le tableau `C` de la matrice `P` ci-dessous (le petit chiffre en haut à gauche de chaque case est le coefficient de `P`, les cases bleues sont celles qui contiennent un 1).

```
# À compléter à la main : remplacer chaque None par sa valeur, puis exécuter.
P = [[0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1],
     [1, 1, 1, 0, 0],
     [1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0]]

C_main = [[0, 1, 1, 1, 1],
          [1, 1, 2, None, None],
          [1, None, None, None, None],
          [1, None, None, None, None],
          [0, None, None, None, None]]
dessiner_matrice(P, C=C_main)
```




```
P = [[0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1],
     [1, 1, 1, 0, 0],
     [1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0]]

C_main = [[0, 1, 1, 1, 1],
          [1, 1, 2, 2, 2],
          [1, 2, 2, 0, 0],
          [1, 2, 3, 1, 0],
          [0, 1, 2, 2, 0]]
dessiner_matrice(P, C=C_main)
```

tableau C correct : plus grand carré de côté 3

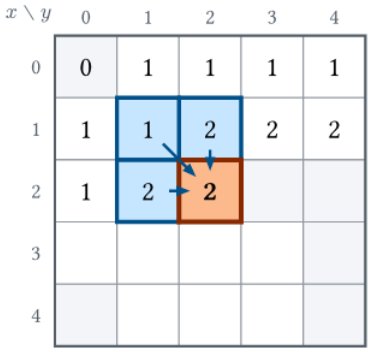
0	1	1	1	1
1	1	2	2	2
1	2	2	0	0
1	2	3	1	0
0	1	2	2	0

Question 4. On suppose que $m[x][y] = 1$. Que vaut $C[x][y]$ si $x = 0$ ou $y = 0$? Sinon, en s'aidant du tableau précédent, exprimer $C[x][y]$ en fonction de $C[x-1][y]$, $C[x][y-1]$ et $C[x-1][y-1]$. *Bonus* : démontrer cette relation par double inégalité, en s'appuyant sur un dessin.

Réponse. Sur la première ligne ou la première colonne, un carré de côté 2 sortirait de la matrice : $C[x][y] = 1$. Sinon,

$$C[x][y] = 1 + \min\left(C[x-1][y],\ C[x][y-1],\ C[x-1][y-1]\right).$$

Dans le tableau, une case lit ses voisines du dessus, de gauche et de la diagonale, ici pour la matrice `P`. La case $(2, 2)$ montre que la diagonale compte : ses voisines du dessus et de gauche valent 2, mais le 0 en $(0, 0)$ l'empêche d'être le coin d'un carré de côté 3, et l'on a $C[2][2] = 2$. Le plus grand carré de `P` a pour coin inférieur droit $(3, 2)$.



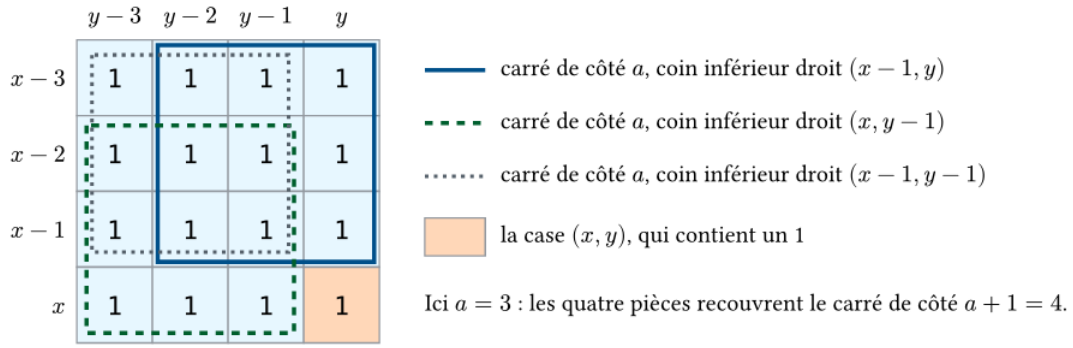
$m[2][2] = 1$:

$C[2][2] = 1 + \min(2, 2, 1) = 2$

cases grises : $m[x][y] = 0$

Démonstration : notons a le minimum des trois voisins.

— « $\geq$ » : chacun des trois voisins est le coin inférieur droit d'un carré de 1 de côté a. Avec la case (x, y), qui contient un 1, ces trois carrés recouvrent le carré de côté $a+1$ de coin inférieur droit (x, y) (figure ci-dessous) : il ne contient que des 1, donc $C[x][y] \geq a+1$.
— « $\leq$ » : soit $k = C[x][y]$. Le carré de 1 de côté k de coin inférieur droit (x, y) contient les carrés de côté $k-1$ dont les coins inférieurs droits sont les trois voisins : chacune des trois valeurs est au moins $k-1$, donc $a \geq k-1$.



Calcul de bas en haut

Question 5. Dans quel ordre peut-on remplir le tableau `C` ? Écrire une fonction `tableau_carres(m)` qui prend en argument une matrice `m` et renvoie le tableau `C` (une nouvelle liste de listes, de mêmes dimensions que `m`), puis une fonction `plus_grand_carre(m)` qui prend en argument une matrice `m` et renvoie la taille de son plus grand carré de 1. Quelle est leur complexité ?

```
def tableau_carres(m):
    n, p = len(m), len(m[0])
    C = [[0 for y in range(p)] for x in range(n)]    # un nouveau tableau
    for x in range(n):                               # ligne par ligne,
        for y in range(p):                           # de gauche à droite
            if m[x][y] == 1:
                if x == 0 or y == 0:
                    C[x][y] = 1
                else:
                    C[x][y] = 1 + minimum3(C[x - 1][y], C[x][y - 1], C[x - 1][y - 1])
    return C

def plus_grand_carre(m):
    C = tableau_carres(m)
    meilleur = 0
    for x in range(len(m)):
        for y in range(len(m[0])):
            if C[x][y] > meilleur:
                meilleur = C[x][y]
    return meilleur
```

Réponse. Une case lit ses voisines du dessus, de gauche et de la diagonale : l'ordre de lecture (ligne par ligne, de gauche à droite) convient. Il y a np cases, chacune calculée en $O(1)$: $O(np)$ en temps et en mémoire, contre $O(n^5)$ pour la recherche exhaustive sur une matrice carrée.

```python
# Tests : chaque ligne doit afficher True.
M = [[0, 1, 1, 0, 1, 1, 1, 0],
     [1, 1, 1, 0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0, 1, 1, 1],
     [1, 1, 1, 1, 0, 0, 1, 1],
     [0, 1, 1, 0, 1, 1, 1, 1]]

copie = [ligne[:] for ligne in M]
print(tableau_carres(M) == [[0, 1, 1, 0, 1, 1, 1, 0],
                            [1, 1, 2, 0, 1, 2, 2, 1],
                            [1, 2, 2, 1, 1, 2, 3, 0],
                            [0, 1, 2, 2, 0, 1, 2, 1],
                            [1, 1, 2, 3, 0, 0, 1, 2],
                            [0, 1, 2, 0, 1, 1, 1, 2]])
print(M == copie)       # M n'est pas modifiée
print(plus_grand_carre(M) == 3)
print(plus_grand_carre([[0]]) == 0, plus_grand_carre([[1]]) == 1)
print(plus_grand_carre([[1, 1, 1, 1]]) == 1, plus_grand_carre([[1], [1], [1]]) == 1)
print(plus_grand_carre([[1 for y in range(9)] for x in range(7)]) == 7)
DISQUE = [[1 if (x - 100) ** 2 + (y - 150) ** 2 < 90 ** 2 else 0 for y in range(300)]
          for x in range(200)]
print(plus_grand_carre(DISQUE) == 127)
```

```
True
True
True
True True
True True
True
True
```

```python
# Visualisation : titre vert si le résultat est correct, rouge sinon.
M = [[0, 1, 1, 0, 1, 1, 1, 0],
     [1, 1, 1, 0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0, 1, 1, 1],
     [1, 1, 1, 1, 0, 0, 1, 1],
     [0, 1, 1, 0, 1, 1, 1, 1]]
dessiner_matrice(M, C=tableau_carres(M))
```

tableau C correct : plus grand carré de côté 3

0	1	1	0	1	1	1	0
1	1	2	0	1	2	2	1
1	2	2	1	1	2	3	0
0	1	2	2	0	1	2	1
1	1	2	3	0	0	1	2
0	1	2	0	1	1	1	2

Reconstruction

Question 6. Écrire une fonction `un_plus_grand_carre(m)` qui prend en argument une matrice `m` et renvoie un triplet `(x, y, k)`, où `k` est la taille du plus grand carré de 1 de `m` et `(x, y)` le coin **supérieur gauche** d'un tel carré (et `(0, 0, 0)` si `m` ne contient aucun 1).

```
def un_plus_grand_carre(m):
    C = tableau_carres(m)
    xd, yd, k = 0, 0, 0          # coin inférieur droit et taille du meilleur carré
    for x in range(len(m)):
        for y in range(len(m[0])):
            if C[x][y] > k:
                xd, yd, k = x, y, C[x][y]
    if k == 0:
        return (0, 0, 0)
    return (xd - k + 1, yd - k + 1, k)

# Une case où C est maximal est le coin inférieur droit d'un carré optimal.
```

```
# Tests : chaque ligne doit afficher True.
M = [[0, 1, 1, 0, 1, 1, 1, 0],
     [1, 1, 1, 0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0, 1, 1, 1],
     [1, 1, 1, 1, 0, 0, 1, 1],
     [0, 1, 1, 0, 1, 1, 1, 1]]

x, y, k = un_plus_grand_carre(M)
print(k == 3, (x, y) == (0, 4) or (x, y) == (2, 1))
print(un_plus_grand_carre([[0, 0], [0, 0]]) == (0, 0, 0))
print(un_plus_grand_carre([[0, 0], [0, 1]]) == (1, 1, 1))
print(un_plus_grand_carre([[0, 1, 1], [0, 1, 1], [1, 0, 0]]) == (0, 1, 2))
DISQUE = [[1 if (x - 100) ** 2 + (y - 150) ** 2 < 90 ** 2 else 0 for y in range(300)]
          for x in range(200)]
x, y, k = un_plus_grand_carre(DISQUE)
print(k == 127, est_carre_de_uns(DISQUE, x, y, 127))
```

```
True True
True
True
True
True True
```

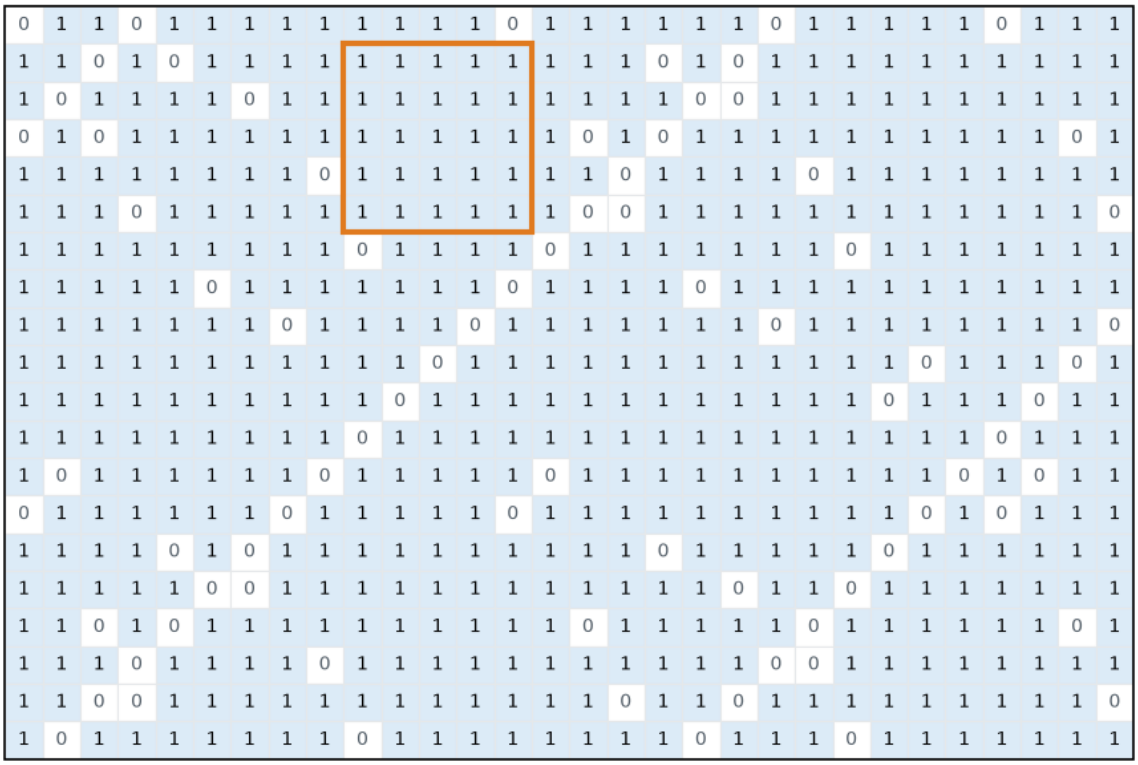
```python
# Visualisation : titre vert si le résultat est correct, rouge sinon.
M = [[0, 1, 1, 0, 1, 1, 1, 0],
     [1, 1, 1, 0, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1, 1, 0],
     [0, 1, 1, 1, 0, 1, 1, 1],
     [1, 1, 1, 1, 0, 0, 1, 1],
     [0, 1, 1, 0, 1, 1, 1, 1]]

dessiner_matrice(M, un_plus_grand_carre(M))
G = [[0 if (x * y + x + 2 * y) % 13 == 0 or (x + y) % 17 == 3 else 1
      for y in range(30)] for x in range(20)]
dessiner_matrice(G, un_plus_grand_carre(G))
```

carré de 1 de côté 3, coin supérieur gauche (0, 4) : maximal

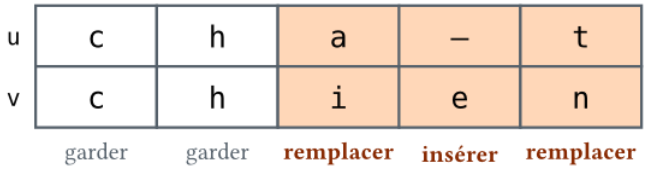
0	1	1	0	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	0
0	1	1	1	0	1	1	1
1	1	1	1	0	0	1	1
0	1	1	0	1	1	1	1

carré de 1 de côté 5, coin supérieur gauche (1, 9) : maximal



2. Distance de Levenshtein

On transforme un mot u en un mot v par des **opérations** qui portent chacune sur une lettre : insérer une lettre, supprimer une lettre, ou remplacer une lettre par une autre (substitution). La **distance de Levenshtein** (ou distance d'édition) entre u et v est le nombre minimal d'opérations nécessaires. Par exemple, la distance entre `chat` et `chien` vaut 3.



On note n et m les longueurs de u et de v.

À la main : un algorithme glouton

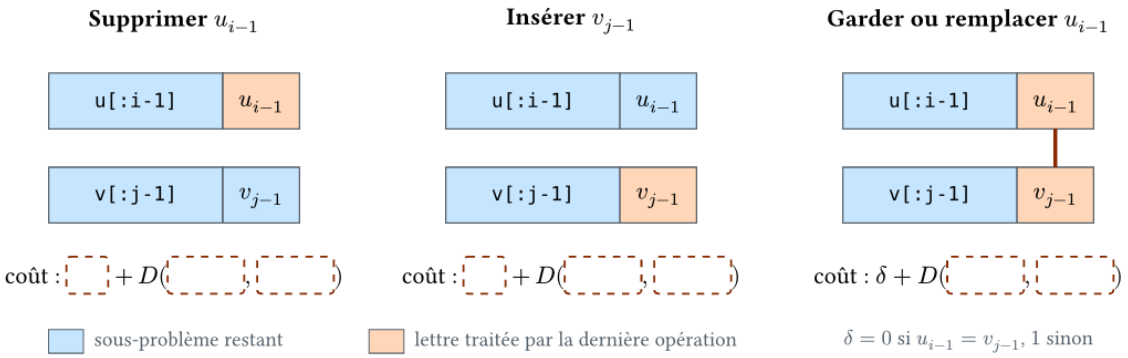
Question 7. Un algorithme glouton compare les deux mots position par position : il remplace la lettre de u quand les deux lettres diffèrent, puis il supprime ou insère les lettres en trop à la fin. Combien d'opérations utilise-t-il pour transformer `chat` en `achat` ? Est-ce optimal ?

Réponse. Il remplace `c` par `a`, `h` par `c`, `a` par `h`, `t` par `a`, puis insère `t` : 5 opérations. Ce n'est pas optimal : il suffit d'insérer `a` en tête, la distance vaut 1.

Sous-problèmes et récurrence

Pour $0 \leq i \leq n$ et $0 \leq j \leq m$, on note $D(i, j)$ la distance entre `u[:i]` (les i premières lettres de u) et `v[:j]` (les j premières lettres de v). La réponse est $D(n, m)$.

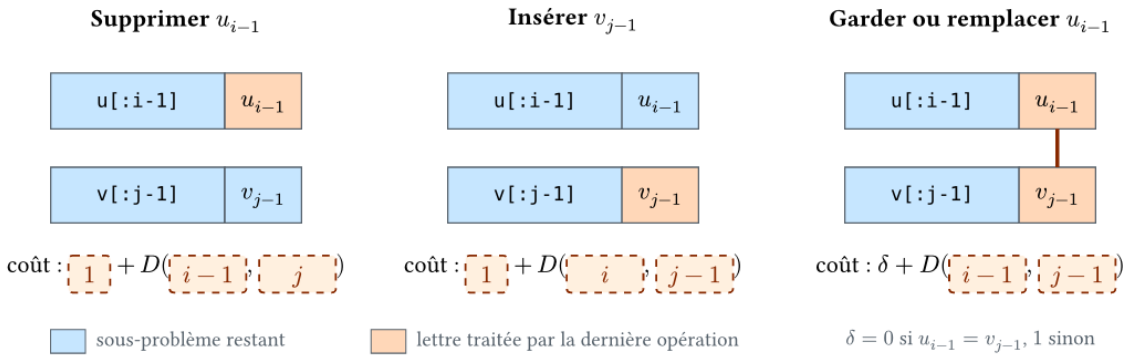
Question 8. Que valent $D(i, 0)$ et $D(0, j)$? Pour $i, j \geq 1$, une transformation de `u[:i]` en `v[:j]` se termine de l'une des trois façons du schéma ci-dessous. Compléter le schéma, puis en déduire la relation de récurrence vérifiée par D, en la justifiant brièvement.



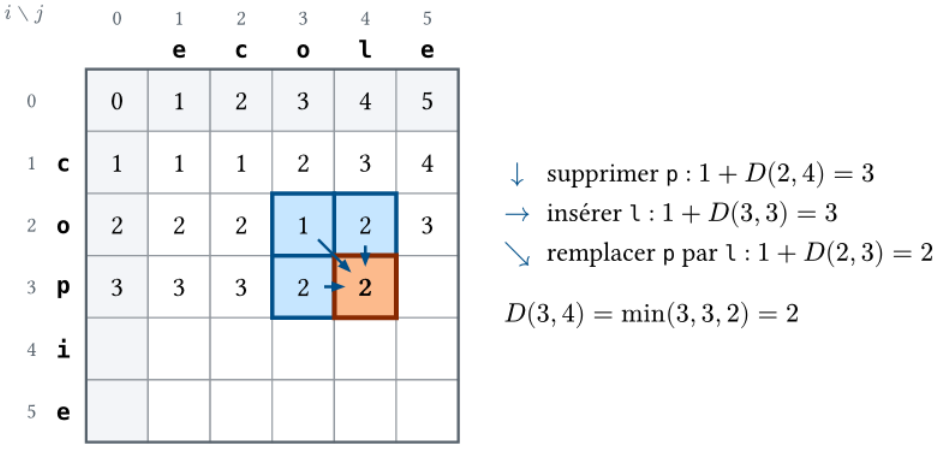
Réponse. $D(i,0) = i$ (supprimer les i lettres) et $D(0,j) = j$ (insérer les j lettres). Pour $i, j \geq 1$,

$$D(i,j) = \min\left(1 + D(i-1,j),\ 1 + D(i,j-1),\ \delta + D(i-1,j-1)\right),$$

où $\delta = 0$ si $u_{i-1} = v_{j-1}$ et $\delta = 1$ sinon. Chaque terme est le coût d'une transformation de `u[:i]` en `v[:j]`, et une transformation optimale se termine de l'une de ces trois façons, ce qui précède étant optimal pour le sous-problème restant.



Dans le tableau, une case lit ses voisines du dessus, de gauche et de la diagonale, ici pour `copie` et `ecole` :



Question 9. Pour `u = "copie"` (en lignes) et `v = "ecole"` (en colonnes), compléter à la main le tableau des $D(i,j)$. Dans quel ordre l'avez-vous rempli ?

```
# À compléter à la main : remplacer chaque None par sa valeur, puis exécuter.

D_main = [[0, 1, 2, 3, 4, 5],
          [1, 1, 1, 2, 3, 4],
          [2, None, None, None, None, None],
          [3, None, None, None, None, None],
          [4, None, None, None, None, None],
          [5, None, None, None, None, None]]
dessiner_tableau_distance("copie", "ecole", D_main)
```

20 case(s) à compléter

i\j	0	1 e	2 c	3 o	4 l	5 e
0	0	1	2	3	4	5
1 c	1	1	1	2	3	4
2 o	2					
3 p	3					
4 i	4					
5 e	5					

```
D_main = [[0, 1, 2, 3, 4, 5],
          [1, 1, 1, 2, 3, 4],
          [2, 2, 2, 1, 2, 3],
          [3, 3, 3, 2, 2, 3],
          [4, 4, 4, 3, 3, 3],
          [5, 4, 5, 4, 4, 3]]
dessiner_tableau_distance("copie", "ecole", D_main)
```

tableau correct : distance(copie, ecole) = 3

i\j	0	1 e	2 c	3 o	4 l	5 e
0	0	1	2	3	4	5
1 c	1	1	1	2	3	4
2 o	2	2	2	1	2	3
3 p	3	3	3	2	2	3
4 i	4	4	4	3	3	3
5 e	5	4	5	4	4	3

Réponse. Une case lit ses voisines du dessus, de gauche et de la diagonale : on remplit ligne par ligne, de gauche à droite (ou colonne par colonne). On trouve $D(5,5) = 3$: insérer `e`, garder `c` et `o`, remplacer `p` par `l`, supprimer `i`, garder `e`.

Mémoïsation

Question 10. Écrire une fonction `levenshtein_memo(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie leur distance, calculée par une fonction récursive mémoïsée (avec un dictionnaire). Quelle profondeur la récursion peut-elle atteindre ?

```
def levenshtein_memo_aux(u, v, i, j, memo):
    if i == 0:                                  # cas de base
        return j
    if j == 0:
        return i
    if (i, j) in memo:                          # déjà calculé
        return memo[(i, j)]
    diagonale = levenshtein_memo_aux(u, v, i - 1, j - 1, memo)
    if u[i - 1] != v[j - 1]:
        diagonale = diagonale + 1
    supprimer = 1 + levenshtein_memo_aux(u, v, i - 1, j, memo)
    inserer = 1 + levenshtein_memo_aux(u, v, i, j - 1, memo)
    memo[(i, j)] = minimum3(supprimer, inserer, diagonale)
    return memo[(i, j)]

def levenshtein_memo(u, v):
    return levenshtein_memo_aux(u, v, len(u), len(v), {})
```

Réponse. Il y a $(n+1)(m+1)$ sous-problèmes, chacun calculé une seule fois, en $O(1)$: $O(nm)$ en temps et en mémoire. Chaque appel diminue $i+j$ d'au moins 1 : la profondeur peut atteindre $n+m$. Pour des mots de quelques centaines de lettres, on dépasse la limite de 1000 appels imbriqués de Python (relevée au début du TP) : le calcul de bas en haut n'a pas ce défaut.

```
# Tests : chaque ligne doit afficher True.

print(levenshtein_memo("chat", "chien") == 3, levenshtein_memo("", "") == 0)
print(levenshtein_memo("chat", "achat") == 1, levenshtein_memo("abc", "") == 3)
print(levenshtein_memo("anticonstitutionnellement", "constitution") == 13)
print(levenshtein_memo("a" * 300, "b" * 300) == 300)
```

```
True True
True True
True
True
```

Calcul de bas en haut

Question 11. Écrire une fonction `tableau_levenshtein(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie le tableau `D` des $D(i,j)$ (une liste de listes), puis une fonction `distance_levenshtein(u, v)` qui renvoie la distance entre `u` et `v` (un entier). Quelles sont les dimensions du tableau ? Quelle est la complexité en temps et en mémoire ?

```python
def tableau_levenshtein(u, v):
    n, m = len(u), len(v)
    D = [[0 for j in range(m + 1)] for i in range(n + 1)]    # (n + 1) x (m + 1)
    for i in range(n + 1):
        D[i][0] = i                     # supprimer les i lettres de u[:i]
    for j in range(m + 1):
        D[0][j] = j                     # insérer les j lettres de v[:j]
    for i in range(1, n + 1):           # ligne par ligne,
        for j in range(1, m + 1):       # de gauche à droite
            diagonale = D[i - 1][j - 1]
            if u[i - 1] != v[j - 1]:
                diagonale = diagonale + 1
            D[i][j] = minimum3(1 + D[i - 1][j], 1 + D[i][j - 1], diagonale)
    return D

def distance_levenshtein(u, v):
    return tableau_levenshtein(u, v)[len(u)][len(v)]
```

Réponse. Le tableau a $n+1$ lignes et $m+1$ colonnes, et non $n \times m$: la ligne 0 et la colonne 0 sont les cas de base. Attention au décalage : la case (i, j) compare les lettres `u[i - 1]` et `v[j - 1]`. Il y a $(n+1)(m+1)$ cases, chacune calculée en $O(1)$: $O(nm)$ en temps et en mémoire.

```python
# Tests : chaque ligne doit afficher True.

print(tableau_levenshtein("chat", "chien") == [[0, 1, 2, 3, 4, 5],
                                               [1, 0, 1, 2, 3, 4],
                                               [2, 1, 0, 1, 2, 3],
                                               [3, 2, 1, 1, 2, 3],
                                               [4, 3, 2, 2, 2, 3]])
print(tableau_levenshtein("", "ab") == [[0, 1, 2]])
print(tableau_levenshtein("ab", "") == [[0], [1], [2]])
print(distance_levenshtein("chat", "chien") == 3)
print(distance_levenshtein("chat", "achat") == 1)
print(distance_levenshtein("", "") == 0)
print(distance_levenshtein("kitten", "sitting") == 3)
print(distance_levenshtein("intention", "execution") == 5)
print(distance_levenshtein("anticonstitutionnellement", "constitution") == 13)
print(distance_levenshtein("a" * 300, "b" * 300) == 300)
```

```
True
True
True
True
True
True
True
True
True
True
```

```python
# Visualisation : titre vert si le résultat est correct, rouge sinon.

dessiner_tableau_distance("chat", "chien", tableau_levenshtein("chat", "chien"))
```

tableau correct : distance(chat, chien) = 3

i\j	0	1 c	2 h	3 i	4 e	5 n
0	0	1	2	3	4	5
1 c	1	0	1	2	3	4
2 h	2	1	0	1	2	3
3 a	3	2	1	1	2	3
4 t	4	3	2	2	2	3

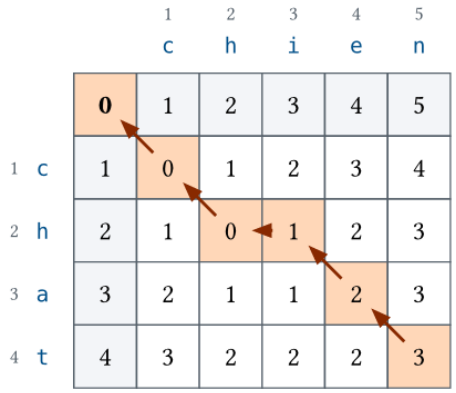
Reconstruction

On décrit une transformation par la liste de ses opérations, lue de gauche à droite : chaque opération est une chaîne de caractères de la forme `"garder c"`, `"remplacer a par i"`, `"insérer e"` ou `"supprimer y"`. Garder une lettre ne coûte rien. La figure du début de l'exercice correspond à la liste

`["garder c", "garder h", "remplacer a par i", "insérer e", "remplacer t par n"]`.

La fonction `appliquer(u, ops)` du module `dessins` applique une telle liste au mot `u` et renvoie le mot obtenu.

Pour reconstruire une liste optimale, on remonte dans le tableau depuis la case (n, m) en cherchant, à chaque case, quel terme du minimum donne sa valeur : c'est ce que montre l'animation ci-dessous pour `chat` et `chien` (sur papier, seule sa dernière image est reproduite). Elle trouve une autre liste optimale que celle de la figure du début.



opérations trouvées, de la dernière à la première :

`remplacer t par n`	$D(4,5) = 1 + D(3,4)$
`remplacer a par e`	$D(3,4) = 1 + D(2,3)$
`insérer i`	$D(2,3) = 1 + D(2,2)$
`garder h`	lettres égales, $D(2,2) = D(1,1)$
`garder c`	lettres égales, $D(1,1) = D(0,0)$

arrivée en $(0,0)$: il reste à renverser la liste

Question 12. Écrire une fonction `operations_levenshtein(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie une liste optimale d'opérations transformant `u` en `v`, obtenue en remontant dans le tableau `D` depuis la case (n, m). Quelle est la complexité de la remontée ?

```python
def operations_levenshtein(u, v):
    D = tableau_levenshtein(u, v)
    i, j = len(u), len(v)
    ops = []                        # opérations, de la dernière à la première
    while i > 0 or j > 0:
        # on cherche quel terme du minimum donne D[i][j]
        if i > 0 and j > 0 and u[i - 1] == v[j - 1] and D[i][j] == D[i - 1][j - 1]:
            ops.append("garder " + u[i - 1])
            i, j = i - 1, j - 1
        elif i > 0 and j > 0 and D[i][j] == 1 + D[i - 1][j - 1]:
            ops.append("remplacer " + u[i - 1] + " par " + v[j - 1])
            i, j = i - 1, j - 1
        elif i > 0 and D[i][j] == 1 + D[i - 1][j]:
            ops.append("supprimer " + u[i - 1])
            i = i - 1
        else:                       # D[i][j] == 1 + D[i][j - 1], avec j > 0
            ops.append("insérer " + v[j - 1])
            j = j - 1
    return ops[::-1]                # liste remise à l'endroit
```

Réponse. Dans la case (i, j), l'un des trois termes de la récurrence donne le minimum : on le retrouve en relisant le tableau, on note l'opération, et on passe à la case du sous-problème, jusqu'à $(0, 0)$. Les opérations sont obtenues de la dernière à la première, d'où le renversement final. À chaque tour, $i + j$ diminue d'au moins 1 : la remontée coûte $O(n + m)$, après le calcul du tableau en $O(nm)$.

```python
# Tests : chaque ligne doit afficher True.

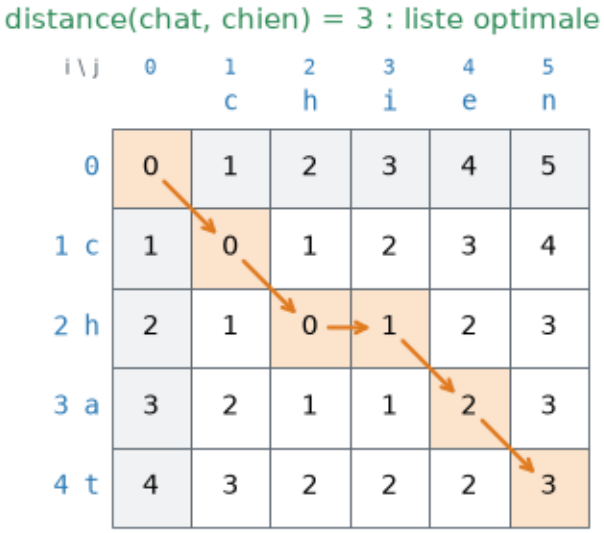
def cout(ops):
    return len([op for op in ops if op[:6] != "garder"])

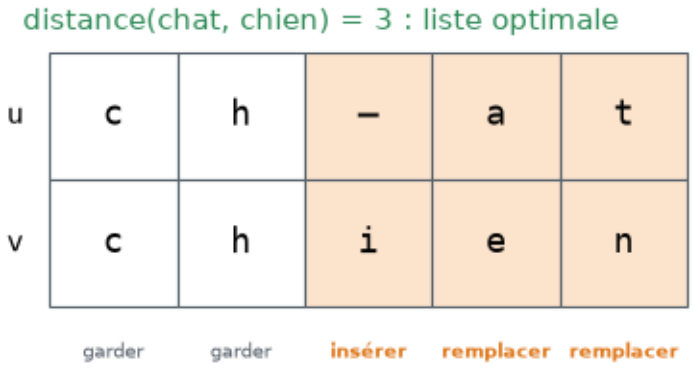
ops = operations_levenshtein("chat", "chien")
print(appliquer("chat", ops) == "chien", cout(ops) == 3)
print(operations_levenshtein("abc", "abc") == ["garder a", "garder b", "garder c"])
print(operations_levenshtein("", "ab") == ["insérer a", "insérer b"])
print(operations_levenshtein("ab", "") == ["supprimer a", "supprimer b"])
print(operations_levenshtein("", "") == [])
for u, v, d in [("chat", "achat", 1), ("kitten", "sitting", 3),
                ("intention", "execution", 5), ("ginette", "interne", 4),
                ("anticonstitutionnellement", "constitution", 13)]:
    ops = operations_levenshtein(u, v)
    print(appliquer(u, ops) == v, cout(ops) == d)
```

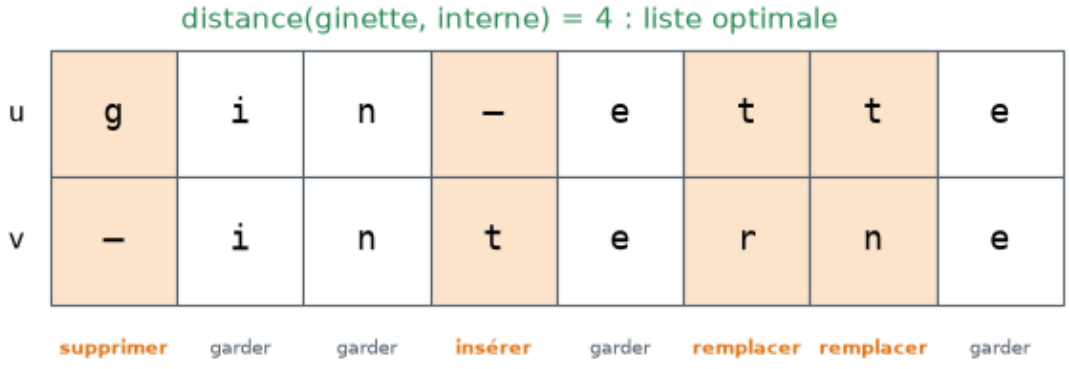
```
True True
True
True
True
True
True True
True True
True True
True True
True True
```

```
# Visualisation : titre vert si le résultat est correct, rouge sinon.

ops = operations_levenshtein("chat", "chien")
dessiner_tableau_distance("chat", "chien", tableau_levenshtein("chat", "chien"), ops)
dessiner_edition("chat", "chien", ops)
ops = operations_levenshtein("ginette", "interne")
dessiner_edition("ginette", "interne", ops)
```







Bonus : mémoire, variante

Question 13 *(bonus)*. Écrire une fonction `levenshtein_2lignes(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie leur distance en n'utilisant qu'une mémoire $O(m)$. Que perd-on ?

```
def levenshtein_2lignes(u, v):
    n, m = len(u), len(v)
    prec = [j for j in range(m + 1)]          # la ligne 0
    for i in range(1, n + 1):
        cour = [0 for j in range(m + 1)]      # la ligne i
        cour[0] = i
        for j in range(1, m + 1):
            diagonale = prec[j - 1]
            if u[i - 1] != v[j - 1]:
                diagonale = diagonale + 1
            cour[j] = minimum3(1 + prec[j], 1 + cour[j - 1], diagonale)
        prec = cour
    return prec[m]
```

Réponse. La ligne i ne lit que la ligne $i-1$ et elle-même : deux listes de longueur $m+1$ suffisent. Le temps reste $O(nm)$, mais on perd la reconstruction, qui a besoin de tout le tableau.

```
# Tests : chaque ligne doit afficher True.

print(levenshtein_2lignes("chat", "chien") == 3, levenshtein_2lignes("", "") == 0)
print(levenshtein_2lignes("", "abc") == 3, levenshtein_2lignes("abc", "") == 3)
print(levenshtein_2lignes("anticonstitutionnellement", "constitution") == 13)
print(levenshtein_2lignes("algorithme" * 50, "logarithme" * 50) == 150)
```

```
True True
True True
True
True
```

Question 14 *(bonus)*. On n'autorise plus que l'insertion et la suppression (remplacer une lettre demande alors deux opérations). Que devient la relation de récurrence ? Écrire une fonction `distance_insertions_suppressions(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie cette distance (5 pour `chat` et `chien`).

```python
def distance_insertions_suppressions(u, v):
    n, m = len(u), len(v)
    D = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(n + 1):
        D[i][0] = i
    for j in range(m + 1):
        D[0][j] = j
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if u[i - 1] == v[j - 1]:
                D[i][j] = D[i - 1][j - 1]           # garder la lettre
            elif D[i - 1][j] < D[i][j - 1]:
                D[i][j] = 1 + D[i - 1][j]           # supprimer u[i - 1]
            else:
                D[i][j] = 1 + D[i][j - 1]           # insérer v[j - 1]
    return D[n][m]
```

Réponse. Les cas de base sont inchangés. Pour $i, j \geq 1$, en notant D' la nouvelle distance,

$$D'(i,j) = \begin{cases} D'(i-1, j-1) & \text{si } u_{i-1} = v_{j-1}, \\ 1 + \min\left(D'(i-1,j),\ D'(i,j-1)\right) & \text{sinon.} \end{cases}$$

Si les dernières lettres sont égales, les garder ne coûte rien et c'est optimal ; sinon, la dernière opération supprime u_{i-1} ou insère v_{j-1}. Les cases lues sont les mêmes qu'à la question 8, la diagonale ne servant que si les deux lettres sont égales. Les lettres gardées forment une plus longue sous-suite commune à u et v (une sous-suite s'obtient en supprimant des lettres, sans changer l'ordre des autres), de longueur ℓ : la distance vaut $n + m - 2\ell$.

```python
# Tests : on ne doit lire que des True.

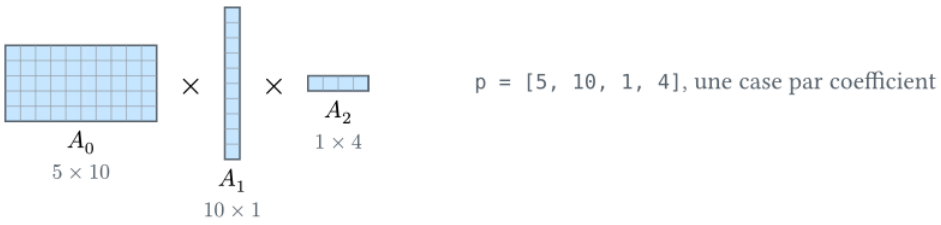
cas = [("chat", "chien", 5), ("chat", "achat", 1), ("", "abc", 3), ("abc", "", 3),
       ("ginette", "interne", 6), ("intention", "execution", 8), ("", "", 0)]
print([distance_insertions_suppressions(u, v) == d for u, v, d in cas])
print(distance_insertions_suppressions("algorithme" * 50, "logarithme" * 50) == 200)
```

```
[True, True, True, True, True, True, True]
True
```

3. Produit d'une chaîne de matrices

Le produit d'une matrice de taille $a \times b$ par une matrice de taille $b \times c$ coûte abc multiplications de coefficients. On veut calculer le produit $A_0 A_1 \cdots A_{n-1}$ de $n \geq 1$ matrices, où A_i est de taille $p_i \times p_{i+1}$: les dimensions sont données par la liste `p` $= [p_0, p_1, \ldots, p_n]$, de longueur $n+1$.

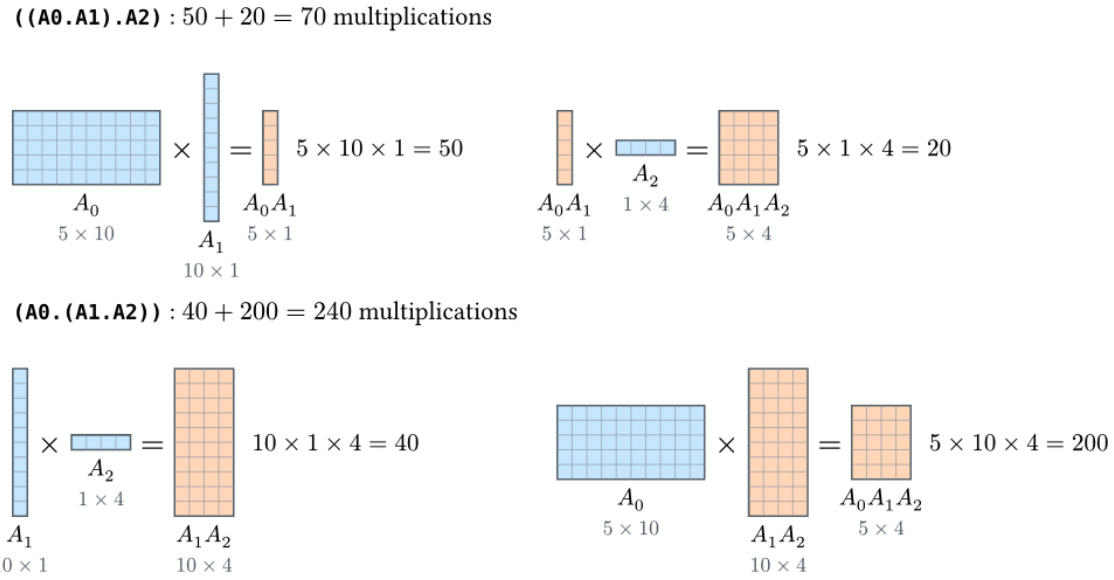
Le produit matriciel est associatif : le résultat ne dépend pas du **parenthésage**, mais le nombre de multiplications, si. On cherche un parenthésage de coût minimal. On le décrit par une chaîne de caractères : `"((A0.A1).A2)"` ou `"(A0.(A1.A2))"` pour trois matrices, `"A0"` pour une seule.



À la main : un algorithme glouton

Question 15. Pour `p = [5, 10, 1, 4]`, calculer le coût des deux parenthésages `"((A0.A1).A2)"` et `"(A0.(A1.A2))"`. Lequel obtient l'algorithme glouton qui effectue à chaque étape le produit de deux matrices voisines le moins coûteux ?

Réponse. `((A0.A1).A2)` : $5 \times 10 \times 1 = 50$ pour $A_0 A_1$, de taille 5×1, puis $5 \times 1 \times 4 = 20$, soit **70**. `(A0.(A1.A2))` : $10 \times 1 \times 4 = 40$ pour $A_1 A_2$, de taille 10×4, puis $5 \times 10 \times 4 = 200$, soit **240**. L'algorithme glouton commence par $A_1 A_2$ ($40 < 50$) et obtient 240 : il n'est pas optimal.



La fonction `cout_parenthesage(p, chaine)` du module `dessins` calcule le coût d'un parenthésage : on peut l'utiliser pour vérifier.

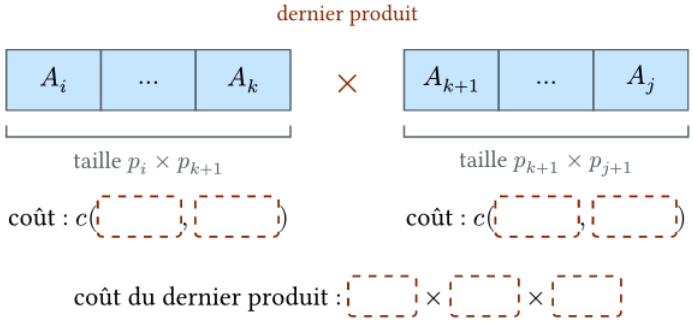
```
p = [5, 10, 1, 4]
print(cout_parenthesage(p, "((A0.A1).A2)"), cout_parenthesage(p, "(A0.(A1.A2))"))
```

```
70 240
```

Sous-problèmes et récurrence

Le nombre de parenthésages croît exponentiellement avec n : les essayer tous est exclu. Pour $0 \leq i \leq j \leq n-1$, on note $c(i,j)$ le nombre minimal de multiplications pour calculer le produit $A_i \cdots A_j$. La réponse est $c(0, n-1)$.

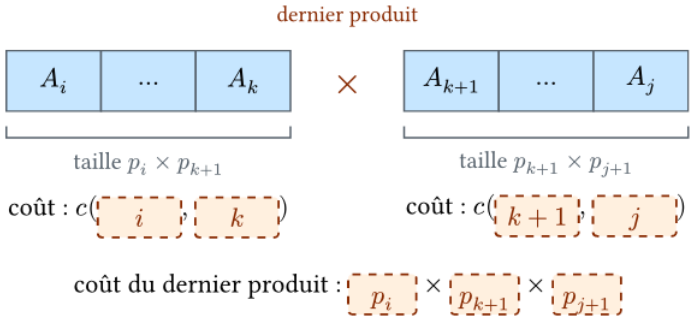
Question 16. Que vaut $c(i,i)$? Pour $i < j$, le dernier produit effectué coupe la chaîne après une matrice A_k, avec $i \leq k < j$. Compléter le schéma, puis en déduire la relation de récurrence vérifiée par c. Pourquoi les produits $A_0 \cdots A_j$ (les préfixes) ne suffisent-ils pas comme sous-problèmes ?



Réponse. $c(i,i) = 0$: il n'y a aucun produit à faire. Pour $i < j$, on essaie toutes les coupures :

$$c(i,j) = \min_{i \leq k < j} \left(c(i,k) + c(k+1,j) + p_i\, p_{k+1}\, p_{j+1} \right).$$

Dans un parenthésage optimal, les deux facteurs sont eux-mêmes parenthésés de façon optimale. Le facteur de droite $A_{k+1} \cdots A_j$ n'est pas un préfixe : les sous-problèmes sont les **intervalles** (i,j), et non les préfixes.



Question 17. Compléter à la main le tableau des $c(i,j)$ pour `p = [2, 3, 1, 4, 2]` (seules les cases $i \leq j$ servent).

```
# À compléter à la main : remplacer chaque None par sa valeur, puis exécuter.

p = [2, 3, 1, 4, 2]
c_main = [[0, None, None, None],
          [0, 0, None, None],
          [0, 0, 0, None],
          [0, 0, 0, 0]]
dessiner_tableau_chaine(p, c_main)
```

6 case(s) à compléter

i \ j	A0 2×3	A1 3×1	A2 1×4	A3 4×2
0 A0 : 2×3	0			
1 A1 : 3×1		0		
2 A2 : 1×4			0	
3 A3 : 4×2				0

```
p = [2, 3, 1, 4, 2]
c_main = [[0, 6, 14, 18],
          [0, 0, 12, 14],
          [0, 0, 0, 8],
          [0, 0, 0, 0]]
dessiner_tableau_chaine(p, c_main)
```

tableau correct : 18 multiplications au minimum

i \ j	A0 2×3	A1 3×1	A2 1×4	A3 4×2
0 A0 : 2×3	0	6	14	18
1 A1 : 3×1		0	12	14
2 A2 : 1×4			0	8
3 A3 : 4×2				0

Réponse. Par exemple $c(0,2) = \min(0+12+2\times3\times4,\ 6+0+2\times1\times4) = \min(36,14) = 14$, et $c(0,3) = \min(0+14+12,\ 6+8+4,\ 14+0+16) = 18$, obtenu pour $k=1$: le parenthésage optimal est `((A0.A1).(A2.A3))`.

Mémoïsation

Question 18. Écrire une fonction `cout_chaine_memo(p)` qui prend en argument la liste `p` des dimensions et renvoie le coût minimal $c(0, n-1)$, calculé par une fonction récursive mémoïsée (avec un dictionnaire). Quelle est sa complexité en temps et en mémoire ?

```python
def cout_chaine_memo_aux(p, i, j, memo):
    if i == j:                                  # cas de base : une seule matrice
        return 0
    if (i, j) in memo:                          # déjà calculé
        return memo[(i, j)]
    meilleur = INFINI
    for k in range(i, j):                       # le dernier produit coupe après A_k
        gauche = cout_chaine_memo_aux(p, i, k, memo)
        droite = cout_chaine_memo_aux(p, k + 1, j, memo)
        cout = gauche + droite + p[i] * p[k + 1] * p[j + 1]
        if cout < meilleur:
            meilleur = cout
    memo[(i, j)] = meilleur
    return meilleur

def cout_chaine_memo(p):
    return cout_chaine_memo_aux(p, 0, len(p) - 2, {})
```

Réponse. Il y a $O(n^2)$ sous-problèmes, et chacun essaie jusqu'à $n-1$ coupures : $O(n^3)$ en temps, $O(n^2)$ en mémoire. Ici le coût d'un sous-problème n'est pas constant : les compter ne suffit pas. L'ordre de calcul n'est pas à trouver, la récursion s'en charge.

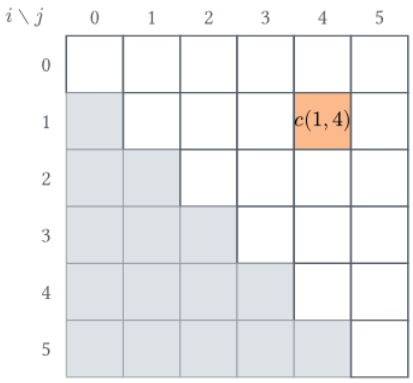
```python
# Tests : chaque ligne doit afficher True.

print(cout_chaine_memo([5, 10, 1, 4]) == 70, cout_chaine_memo([5, 7]) == 0)
print(cout_chaine_memo([2, 3, 4]) == 24, cout_chaine_memo([2, 3, 1, 4, 2]) == 18)
print(cout_chaine_memo([30, 35, 15, 5, 10, 20, 25]) == 15125)
print(cout_chaine_memo([(7 * k) % 23 + 2 for k in range(41)]) == 12932)
```

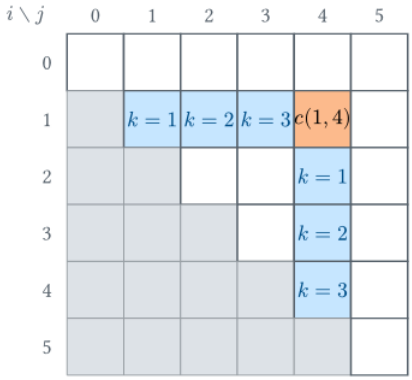
```
True True
True True
True
True
```

Calcul de bas en haut

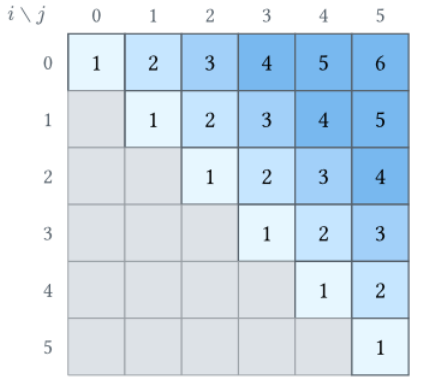
Question 19. La figure ci-dessous représente le tableau pour $n = 6$. Quelles cases faut-il lire pour calculer $c(1, 4)$? Peut-on remplir le tableau ligne par ligne, par i croissant ? Proposer un ordre de remplissage qui convient.



Réponse. $c(1,4)$ lit les cases $c(1,k)$, sur **sa ligne, à gauche**, et les cases $c(k+1,4)$, sur **sa colonne, en dessous**. Remplir par i croissant ne convient pas : la ligne i a besoin des lignes suivantes. Deux ordres conviennent : par **longueur** $j-i+1$ croissante (diagonale par diagonale), ou par i **décroissant** et, sur chaque ligne, par j croissant.



$c(1,4)$ lit sa ligne, à gauche, et sa colonne, en dessous



un ordre possible : par longueur $j-i+1$ croissante, diagonale par diagonale

Question 20. Écrire une fonction `tableau_chaine(p)` qui prend en argument la liste `p` des dimensions et renvoie le tableau `c` des $c(i,j)$, une liste de n listes de longueur n dont seules les cases $i \leq j$ servent, puis une fonction `cout_chaine(p)` qui renvoie le coût minimal $c(0,n-1)$. Quelle est la complexité en temps et en mémoire ?

```
def tableau_chaine(p):
    n = len(p) - 1                                  # nombre de matrices
    c = [[0 for j in range(n)] for i in range(n)]   # c[i][i] = 0
    for longueur in range(2, n + 1):                # longueur j - i + 1 croissante
        for i in range(n - longueur + 1):
            j = i + longueur - 1
            c[i][j] = INFINI
            for k in range(i, j):
                cout = c[i][k] + c[k + 1][j] + p[i] * p[k + 1] * p[j + 1]
                if cout < c[i][j]:
                    c[i][j] = cout
    return c

def cout_chaine(p):
    return tableau_chaine(p)[0][len(p) - 2]
```

Réponse. Même complexité que la version mémoïsée : $O(n^2)$ cases, jusqu'à $n-1$ coupures par case, soit $O(n^3)$ en temps et $O(n^2)$ en mémoire, sans pile d'appels.

```
# Tests : chaque ligne doit afficher True.

c = tableau_chaine([2, 3, 1, 4, 2])
print(c[0][3] == 18, c[1][3] == 14, c[0][2] == 14, c[2][2] == 0)
print(tableau_chaine([5, 7]) == [[0]])
print(cout_chaine([5, 10, 1, 4]) == 70, cout_chaine([5, 7]) == 0)
print(cout_chaine([2, 3, 4]) == 24, cout_chaine([10, 100, 5, 50]) == 7500)
print(cout_chaine([30, 35, 15, 5, 10, 20, 25]) == 15125)
P = [(7 * k) % 23 + 2 for k in range(201)]          # 200 matrices
print(cout_chaine(P) == 62606)
```

```
True True True True
True
True True
True True
True
True
```

```
# Visualisation : titre vert si le résultat est correct, rouge sinon.

p = [30, 35, 15, 5, 10, 20, 25]
dessiner_tableau_chaine(p, tableau_chaine(p))
```

tableau correct : 15125 multiplications au minimum

i \ j	A0 30×35	A1 35×15	A2 15×5	A3 5×10	A4 10×20	A5 20×25
0 A0 : 30×35	0	15750	7875	9375	11875	15125
1 A1 : 35×15		0	2625	4375	7125	10500
2 A2 : 15×5			0	750	2500	5375
3 A3 : 5×10				0	1000	3500
4 A4 : 10×20					0	5000
5 A5 : 20×25						0

Bonus : reconstruction

Question 21 *(bonus)*. Écrire une fonction `parenthesage_optimal(p)` qui prend en argument la liste `p` des dimensions et renvoie un parenthésage optimal. Par exemple, `parenthesage_optimal([5, 10, 1, 4])` renvoie `"((A0.A1).A2)"`. On mémorisera pour chaque case un indice k qui donne le minimum, et on pourra utiliser `str(i)`.

```python
def coupures_chaine(p):
    # coupure[i][j] : un indice k qui donne le minimum dans la récurrence de c(i, j)
    n = len(p) - 1
    c = [[0 for j in range(n)] for i in range(n)]
    coupure = [[0 for j in range(n)] for i in range(n)]
    for a in range(n):
        i = n - 1 - a                       # i décroissant
        for j in range(i + 1, n):           # j croissant
            c[i][j] = INFINI
            for k in range(i, j):
                cout = c[i][k] + c[k + 1][j] + p[i] * p[k + 1] * p[j + 1]
                if cout < c[i][j]:
                    c[i][j] = cout
                    coupure[i][j] = k
    return coupure

def parenthesage_optimal_aux(coupure, i, j):
    # Parenthésage optimal de A_i ... A_j.
    if i == j:
        return "A" + str(i)
    k = coupure[i][j]
    return ("(" + parenthesage_optimal_aux(coupure, i, k) + "."
            + parenthesage_optimal_aux(coupure, k + 1, j) + ")")

def parenthesage_optimal(p):
    return parenthesage_optimal_aux(coupures_chaine(p), 0, len(p) - 2)
```

Réponse. On mémorise le choix gagnant de chaque case (ici avec l'autre ordre de remplissage, i décroissant), puis on reconstruit depuis la case $(0, n-1)$: la coupure donne le dernier produit, et les deux facteurs se reconstruisent récursivement. Le calcul reste en $O(n^3)$, la reconstruction fait $2n-1$ appels.

```python
# Tests : chaque ligne doit afficher True.

print(parenthesage_optimal([5, 10, 1, 4]) == "((A0.A1).A2)")
print(parenthesage_optimal([5, 7]) == "A0")
print(parenthesage_optimal([2, 3, 4]) == "(A0.A1)")
print(parenthesage_optimal([2, 3, 1, 4, 2]) == "((A0.A1).(A2.A3))")
p = [30, 35, 15, 5, 10, 20, 25]
print(cout_parenthesage(p, parenthesage_optimal(p)) == 15125)
P = [(7 * k) % 23 + 2 for k in range(41)]            # 40 matrices
print(cout_parenthesage(P, parenthesage_optimal(P)) == 12932)
```

```
True
True
True
True
True
True
```

4. Sac à dos

On dispose de n objets, numérotés de 0 à $n-1$: l'objet i a un poids p_i et un prix v_i, entiers strictement positifs. On veut remplir un sac de capacité b en prenant certains objets, chacun au plus une fois, de façon à maximiser le prix total emporté.

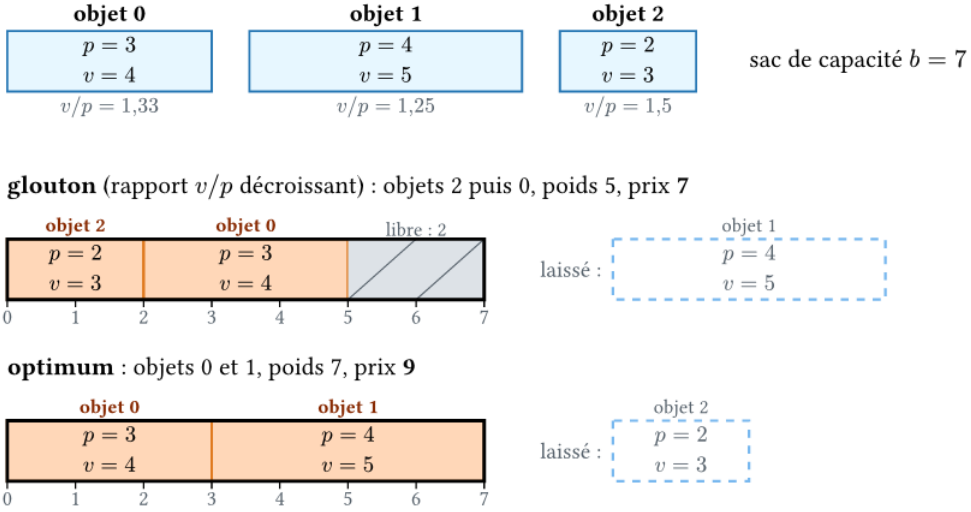
En Python, `objets` est la liste des couples `(poids, prix)`, par exemple `[(3, 4), (4, 5), (2, 3)]`.



À la main : un algorithme glouton

Question 22. Un algorithme glouton parcourt les objets par rapport prix / poids décroissant et prend chaque objet qui tient encore dans le sac. L'appliquer à `objets = [(3, 4), (4, 5), (2, 3)]` et $b = 7$. Est-ce optimal ?

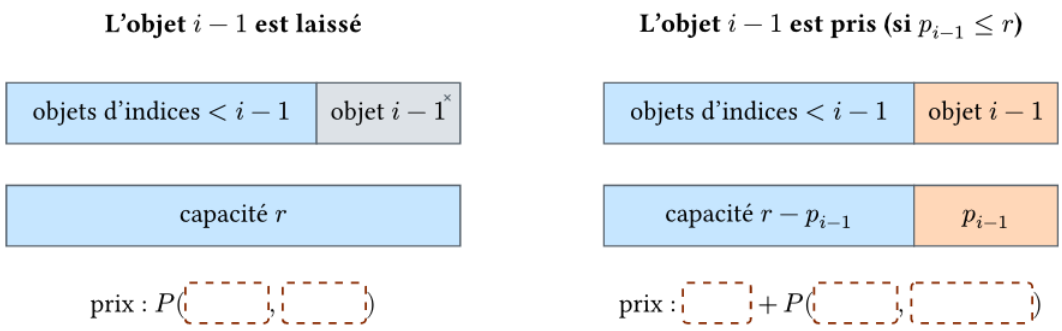
Réponse. Les rapports valent environ 1,33, 1,25 et 1,5 : il prend l'objet 2 (poids 2), puis l'objet 0 (poids total 5), et l'objet 1 ne tient plus. Prix $3+4=7$. Ce n'est pas optimal : les objets 0 et 1 pèsent 7 et valent 9.



Sous-problèmes et récurrence

Pour $0 \leq i \leq n$ et $0 \leq r \leq b$, on note $P(i, r)$ le prix maximal réalisable avec les objets d'indices $< i$ et une capacité r. La réponse est $P(n, b)$.

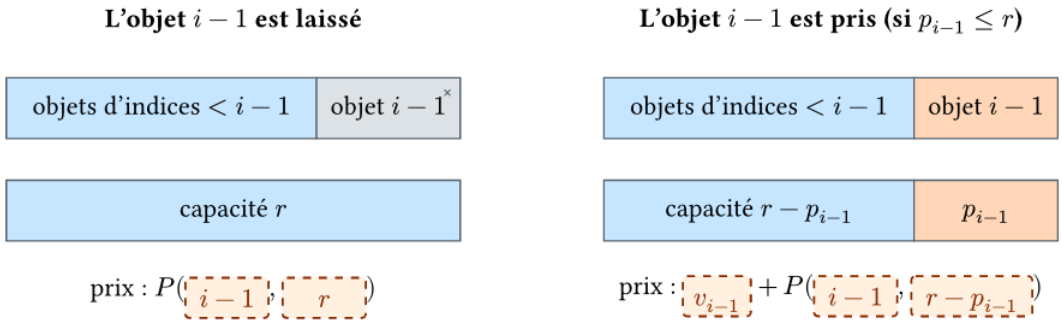
Question 23. Que vaut $P(0, r)$? Pour $i \geq 1$, l'objet $i-1$ est laissé ou pris. Compléter le schéma, puis en déduire la relation de récurrence vérifiée par P. Démontrer que si une solution optimale pour (i, r) prend l'objet $i-1$, alors les autres objets qu'elle prend forment une solution optimale pour $(i-1, r-p_{i-1})$.



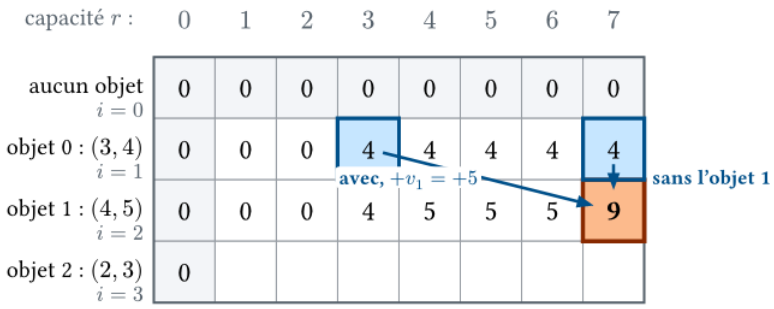
Réponse. $P(0, r) = 0$: aucun objet n'est autorisé. Pour $i \geq 1$,

$$P(i, r) = \begin{cases} P(i-1, r) & \text{si } p_{i-1} > r, \\ \max\left(P(i-1, r),\ v_{i-1} + P(i-1, r-p_{i-1})\right) & \text{sinon.} \end{cases}$$

Démonstration : soit S une solution optimale pour (i, r) qui prend l'objet $i-1$, et S' les autres objets de S. Ils sont d'indices $< i-1$ et pèsent au plus $r-p_{i-1}$: S' est une solution pour $(i-1, r-p_{i-1})$. Si elle n'était pas optimale, il existerait pour ce sous-problème une solution T de prix strictement plus grand, et T complétée par l'objet $i-1$ serait une solution pour (i, r) de prix strictement plus grand que celui de S : c'est absurde. Donc $P(i, r) = v_{i-1} + P(i-1, r-p_{i-1})$ dans ce cas. De même, une solution optimale qui laisse l'objet $i-1$ est optimale pour $(i-1, r)$. Enfin, les deux termes du maximum sont les prix de solutions pour (i, r), d'où l'égalité.



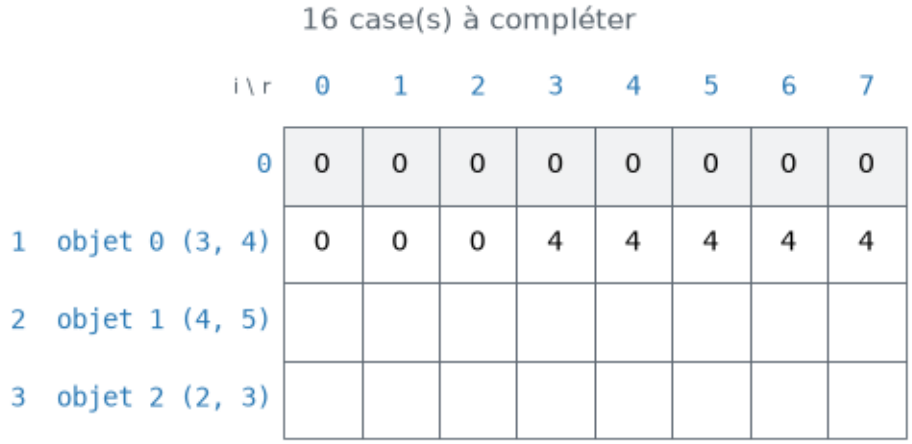
Dans le tableau, une case ne lit que la ligne du dessus : la case située juste au-dessus et, si l'objet tient, celle qui est p_{i-1} colonnes plus à gauche, ici pour les objets de la question 22 :



Question 24. Compléter à la main le tableau des $P(i, r)$ pour `objets = [(3, 4), (4, 5), (2, 3)]` et $b = 7$ (ligne i : objets d'indices $< i$, colonne r : capacité).

```
# À compléter à la main : remplacer chaque None par sa valeur, puis exécuter.

P_main = [[0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 4, 4, 4, 4, 4],
          [None, None, None, None, None, None, None, None],
          [None, None, None, None, None, None, None, None]]
dessiner_tableau_sac([(3, 4), (4, 5), (2, 3)], 7, P_main)
```




```
P_main = [[0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 4, 4, 4, 4, 4],
          [0, 0, 0, 4, 5, 5, 5, 9],
          [0, 0, 3, 4, 5, 7, 8, 9]]
dessiner_tableau_sac([(3, 4), (4, 5), (2, 3)], 7, P_main)
```

tableau correct : prix maximal 9

i\r		0	1	2	3	4	5	6	7
0		0	0	0	0	0	0	0	0
1	objet 0 (3, 4)	0	0	0	4	4	4	4	4
2	objet 1 (4, 5)	0	0	0	4	5	5	5	9
3	objet 2 (2, 3)	0	0	3	4	5	7	8	9

Calcul de bas en haut

Question 25. Écrire une fonction `tableau_sac(objets, b)` qui prend en arguments la liste `objets` et la capacité `b`, et renvoie le tableau `P` des $P(i, r)$ (une liste de listes), puis une fonction `sac(objets, b)` qui renvoie le prix maximal $P(n, b)$. Quelle est la complexité en temps et en mémoire ?

```
def tableau_sac(objets, b):
    n = len(objets)
    P = [[0 for r in range(b + 1)] for i in range(n + 1)]     # P[0][r] = 0
    for i in range(1, n + 1):
        poids, prix = objets[i - 1]
        for r in range(b + 1):
            P[i][r] = P[i - 1][r]                             # sans l'objet i - 1
            if poids <= r and prix + P[i - 1][r - poids] > P[i][r]:
                P[i][r] = prix + P[i - 1][r - poids]          # avec
    return P

def sac(objets, b):
    return tableau_sac(objets, b)[len(objets)][b]
```

Réponse. Le tableau a $n + 1$ lignes et $b + 1$ colonnes, la ligne i ne lit que la ligne $i - 1$: on remplit ligne par ligne. $(n + 1)(b + 1)$ cases en $O(1)$ chacune : $O(nb)$ en temps et en mémoire.

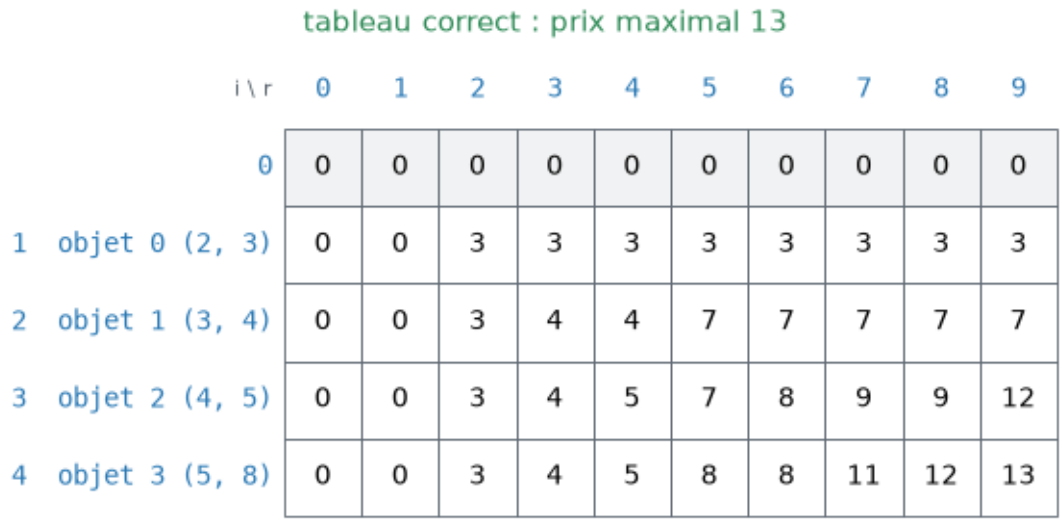
```
# Tests : chaque ligne doit afficher True.

print(tableau_sac([(3, 4), (4, 5), (2, 3)], 7) == [[0, 0, 0, 0, 0, 0, 0, 0],
                                                   [0, 0, 0, 4, 4, 4, 4, 4],
                                                   [0, 0, 0, 4, 5, 5, 5, 9],
                                                   [0, 0, 3, 4, 5, 7, 8, 9]])
print(tableau_sac([], 3) == [[0, 0, 0, 0]], tableau_sac([(2, 5)], 0) == [[0], [0]])
print(sac([(3, 4), (4, 5), (2, 3)], 7) == 9, sac([(2, 3), (1, 4), (4, 4)], 5) == 8)
print(sac([(2, 3), (3, 4), (4, 5)], 5) == 7, sac([(5, 1), (5, 2)], 4) == 0)
objets = [((13 * k) % 29 + 1, (17 * k) % 31 + 1) for k in range(60)]
print(sac(objets, 200) == 604)
```

```
True
True True
True True
True True
True
```

```
# Visualisation : titre vert si le résultat est correct, rouge sinon.

objets = [(2, 3), (3, 4), (4, 5), (5, 8)]
dessiner_tableau_sac(objets, 9, tableau_sac(objets, 9))
```

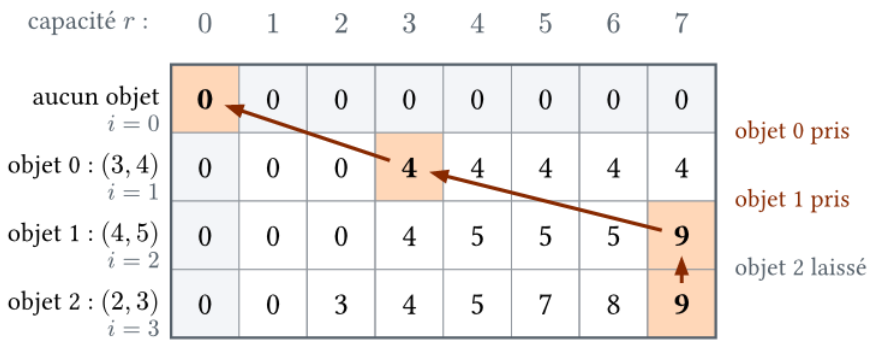


Reconstruction

Question 26. Écrire une fonction `sac_objets(objets, b)` qui renvoie la liste des indices des objets d'une solution optimale, dans l'ordre croissant, en remontant dans le tableau `P` depuis la case (n, b). Par exemple, `sac_objets([(3, 4), (4, 5), (2, 3)], 7)` renvoie `[0, 1]`. Quelle est la complexité de la remontée ?

```
def sac_objets(objets, b):
    n = len(objets)
    P = tableau_sac(objets, b)
    choisis = []
    r = b
    i = n
    while i > 0:                          # de la dernière ligne vers la première
        if P[i][r] != P[i - 1][r]:        # l'objet i - 1 a été pris
            choisis.append(i - 1)
            r = r - objets[i - 1][0]
        i = i - 1
    return choisis[::-1]                  # dans l'ordre des indices
```

Réponse. Si $P(i,r) = P(i-1,r)$, laisser l'objet $i-1$ est optimal, sinon le maximum a été obtenu en le prenant, et l'on poursuit avec la capacité $r - p_{i-1}$. Un test par ligne : la remontée coûte $O(n)$, après le calcul du tableau en $O(nb)$.



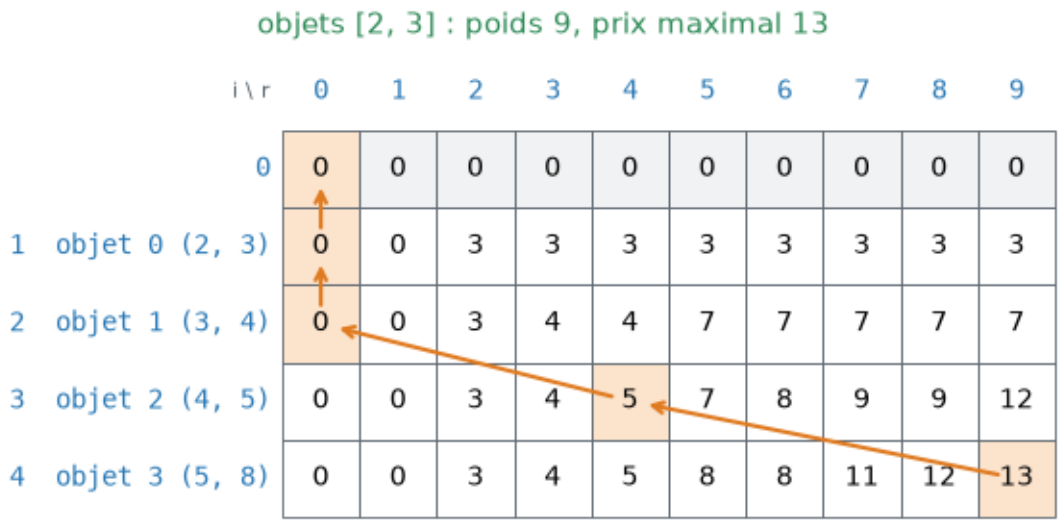

```
# Tests : chaque ligne doit afficher True.

print(sac_objets([(3, 4), (4, 5), (2, 3)], 7) == [0, 1])
print(sac_objets([(2, 3), (1, 4), (4, 4)], 5) == [1, 2])
print(sac_objets([], 3) == [], sac_objets([(5, 1), (5, 2)], 4) == [])
print(sac_objets([(2, 3), (3, 4), (4, 5), (5, 8)], 9) == [2, 3])
objets = [((13 * k) % 29 + 1, (17 * k) % 31 + 1) for k in range(60)]
choisis = sac_objets(objets, 200)
poids_total, prix_total = 0, 0
for i in choisis:
    poids_total = poids_total + objets[i][0]
    prix_total = prix_total + objets[i][1]
print(poids_total <= 200, prix_total == 604)
```

```
True
True
True True
True
True True
```

```
# Visualisation : titre vert si le résultat est correct, rouge sinon.

objets = [(2, 3), (3, 4), (4, 5), (5, 8)]
dessiner_tableau_sac(objets, 9, tableau_sac(objets, 9), sac_objets(objets, 9))
```



Bonus : mémoïsation, mémoire

Question 27 *(bonus)*. Écrire une fonction `sac_memo(objets, b)` qui renvoie $P(n, b)$, calculé par une fonction récursive mémoïsée (avec un dictionnaire). Quel avantage a-t-elle sur le tableau quand la capacité est grande et les objets peu nombreux ?

```
def sac_memo_aux(objets, i, r, memo):
    if i == 0:                                  # cas de base : aucun objet
        return 0
    if (i, r) in memo:                          # déjà calculé
        return memo[(i, r)]
    poids, prix = objets[i - 1]
    meilleur = sac_memo_aux(objets, i - 1, r, memo)             # sans l'objet i - 1
    if poids <= r:
        avec = prix + sac_memo_aux(objets, i - 1, r - poids, memo)
        if avec > meilleur:
            meilleur = avec
    memo[(i, r)] = meilleur
    return meilleur

def sac_memo(objets, b):
    return sac_memo_aux(objets, len(objets), b, {})
```

Réponse. Elle ne calcule que les couples (i, r) atteints depuis (n, b) en retranchant des poids : au plus 2^{n-i} capacités pour la ligne i. Dans le dernier test (14 objets, $b = 10^9$), elle calcule 1146 sous-problèmes, alors que le tableau aurait 15 milliards de cases.

```
# Tests : chaque ligne doit afficher True.

print(sac_memo([(3, 4), (4, 5), (2, 3)], 7) == 9, sac_memo([], 3) == 0)
print(sac_memo([(2, 3), (1, 4), (4, 4)], 5) == 8, sac_memo([(5, 1), (5, 2)], 4) == 0)
objets = [((13 * k) % 29 + 1, (17 * k) % 31 + 1) for k in range(60)]
print(sac_memo(objets, 200) == 604)            # 60 objets
objets = [(100000007 * (k % 5 + 1) + 1000 * k * k, (31 * k) % 17 + 5)
          for k in range(14)]
print(sac_memo(objets, 10**9) == 76)          # 14 objets très lourds
```

```
True True
True True
True
True
```

Question 28 *(bonus)*. Écrire une fonction `sac_une_liste(objets, b)` qui renvoie $P(n, b)$ en n'utilisant qu'une seule liste de longueur $b + 1$. Dans quel sens faut-il la parcourir ? Que perd-on ?

```
def sac_une_liste(objets, b):
    P = [0 for r in range(b + 1)]         # la ligne i - 1, qui devient la ligne i
    for i in range(1, len(objets) + 1):
        poids, prix = objets[i - 1]
        for a in range(b - poids + 1):
            r = b - a                     # de b à poids
            if prix + P[r - poids] > P[r]:
                P[r] = prix + P[r - poids]
    return P[b]
```

Réponse. La ligne i ne lit que la ligne $i-1$, dans des colonnes plus petites : en parcourant r de b à p_{i-1}, la case `P[r - poids]` contient encore la valeur de la ligne $i-1$ quand on la lit. Mémoire $O(b)$ au lieu de $O(nb)$, mais on ne peut plus reconstruire la solution.

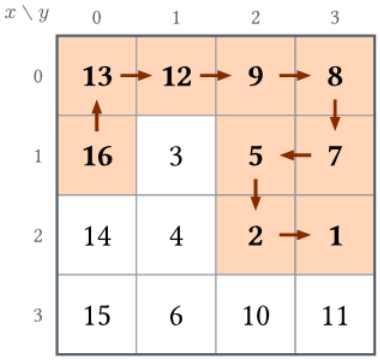
```
# Tests : chaque ligne doit afficher True.

print(sac_une_liste([(3, 4), (4, 5), (2, 3)], 7) == 9, sac_une_liste([], 3) == 0)
print(sac_une_liste([(2, 3), (1, 4), (4, 4)], 5) == 8)
print(sac_une_liste([(5, 1), (5, 2)], 4) == 0)
print(sac_une_liste([(2, 5)], 7) == 5)        # un objet est pris au plus une fois
objets = [((13 * k) % 29 + 1, (17 * k) % 31 + 1) for k in range(60)]
print(sac_une_liste(objets, 200) == 604)
```

```
True True
True
True
True
True
```

5. Bonus : la plus longue descente

Une grille `h` de n lignes et p colonnes (une liste de n listes de p entiers) donne l'altitude de chaque case d'un domaine skiable. Une **descente** est une suite de cases dans laquelle chaque case est voisine de la précédente (au-dessus, en dessous, à gauche ou à droite) et d'altitude **strictement plus petite**. Sa longueur est son nombre de cases. On cherche la longueur d'une plus longue descente.



une plus longue descente, de longueur 9 :

16, 13, 12, 9, 8, 7, 5, 2, 1

```
H = [[13, 12, 9, 8],
     [16, 3, 5, 7],
     [14, 4, 2, 1],
     [15, 6, 10, 11]]
```

Question 29. Un skieur glouton part de la case la plus haute de la grille `H` et rejoint à chaque étape la plus haute de ses voisines plus basses, pour perdre le moins d'altitude possible. Quelle descente obtient-il ? Est-elle de longueur maximale ?

Réponse. Il part de 16, dont les voisines plus basses sont 13, 3 et 14 : il rejoint 14, puis 4, puis 3, et s'arrête, car les voisines de 3 sont toutes plus hautes. Sa descente est de longueur 4, alors que celle de la figure, qui part de la même case, est de longueur 9.

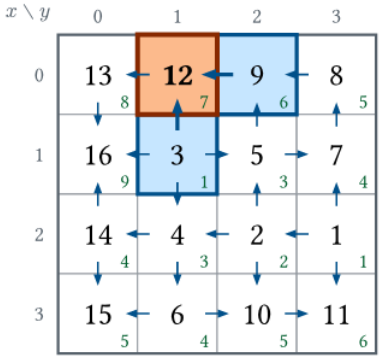
Question 30. On note $L(x, y)$ la longueur d'une plus longue descente **partant** de la case (x, y). Exprimer $L(x, y)$ en fonction des valeurs de L sur certaines voisines de (x, y). Quelles cases la case $(0, 1)$ de `H` lit-elle ? Et la case $(3, 2)$? Peut-on remplir le tableau L ligne par ligne, de gauche à droite ? En le balayant dans un autre sens fixé à l'avance ? Pourquoi n'y a-t-il pourtant aucune dépendance circulaire ?

Réponse. Une descente partant de (x, y) se poursuit par une descente partant d'une voisine plus basse, s'il y en a une :

$$L(x, y) = 1 + \max\{L(x', y') \mid (x', y') \text{ voisine de } (x, y) \text{ et } h[x'][y'] < h[x][y]\},$$

ce maximum valant 0 quand aucune voisine n'est plus basse. Les cases lues dépendent des **altitudes** : la case $(0, 1)$ lit ses voisines de droite et du dessous, la case $(3, 2)$ celles du dessus et de gauche. Aucun balayage fixé à l'avance ne convient donc, par lignes ou par colonnes : la première demande de traiter d'abord ce qui est à droite et en dessous, la seconde ce qui est au-dessus et à gauche. Il n'y a pourtant pas de dépendance circulaire : chaque case ne lit que des cases strictement plus basses. Remplir les cases par altitude croissante conviendrait, mais il faudrait d'abord les trier.

Toutes les lectures de la grille `H` :



$$L(0,1) = 1 + \max(L(0,2), L(1,1))$$
$$= 1 + \max(6,1) = 7$$

une flèche par lecture, de la case lue vers la case qui la lit ; en vert, $L(x,y)$

Question 31. Écrire une fonction `plus_longue_descente(h)` qui prend en argument une grille `h` et renvoie la longueur d'une plus longue descente, calculée par une fonction récursive mémoïsée (avec un dictionnaire dont les clés sont les couples `(x, y)`). Quelle est sa complexité ? Pourquoi la mémoïsation est-elle ici plus commode qu'un tableau rempli par deux boucles ?

```
def descente_aux(h, x, y, memo):
    if (x, y) in memo:                          # déjà calculé
        return memo[(x, y)]
    n, p = len(h), len(h[0])
    meilleure = 0                               # aucune voisine plus basse
    for (dx, dy) in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        a, b = x + dx, y + dy
        if a >= 0 and a < n and b >= 0 and b < p and h[a][b] < h[x][y]:
            longueur = descente_aux(h, a, b, memo)
            if longueur > meilleure:
                meilleure = longueur
    memo[(x, y)] = 1 + meilleure
    return memo[(x, y)]

def plus_longue_descente(h):
    memo = {}
    maxi = 0
    for x in range(len(h)):
        for y in range(len(h[0])):
            longueur = descente_aux(h, x, y, memo)
            if longueur > maxi:
                maxi = longueur
    return maxi
```

Réponse. Il y a np sous-problèmes, chacun calculé une seule fois en lisant au plus quatre voisines : $O(np)$ en temps et en mémoire. La récursion calcule les cases dans un ordre qui convient sans qu'on ait à le déterminer (c'est un parcours en profondeur du graphe des dépendances), alors qu'un calcul de bas en haut demanderait de trier les cases par altitude. En contrepartie, la profondeur de la récursion peut atteindre np.

```python
# Tests : chaque ligne doit afficher True.

print(plus_longue_descente(H) == 9, plus_longue_descente([[5]]) == 1)
print(plus_longue_descente([[3, 3], [3, 3]]) == 1)
print(plus_longue_descente([[1, 2, 3, 4]]) == 4)
serpent = [[20 * x + y for y in range(20)] for x in range(20)]
for x in range(1, 20, 2):
    serpent[x] = serpent[x][::-1]               # une ligne sur deux est renversée
print(plus_longue_descente(serpent) == 400)     # elle visite toutes les cases
grande = [[(37 * x * x + 91 * y * y + 13 * x * y) % 101 for y in range(60)]
          for x in range(60)]
print(plus_longue_descente(grande) == 8)
```

```
True True
True
True
True
True
```

6. Bonus : le jeu des pièces

Des pièces de valeurs $v_0, \dots, v_{n-1}$ sont alignées. Deux joueurs A et B retirent à tour de rôle une pièce à l'**une des deux extrémités** de la rangée, A commençant. Chacun cherche à maximiser la somme de ses pièces.

Question 32. Sur la rangée $8, 15, 3, 7$, que gagne A si chaque joueur prend toujours la plus grande des deux pièces disponibles ? Que gagne-t-il s'il commence par prendre 7 ?

Réponse. A prend 8, B prend 15, A prend 7, B prend 3 : A gagne 15 et B gagne 18. Si A commence par 7, B prend 8 ou 3 et libère 15 dans les deux cas : A gagne $7 + 15 = 22$.

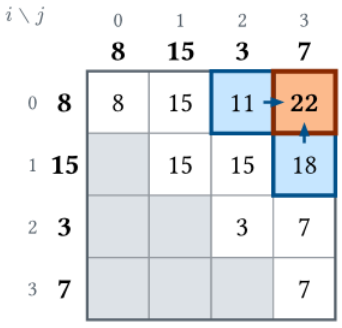
Question 33. Pour $0 \le i \le j \le n-1$, on note $G(i,j)$ le gain du joueur qui commence sur la rangée $v_i, \dots, v_j$ lorsque les deux joueurs jouent au mieux, et $S(i,j) = v_i + \cdots + v_j$. Écrire une fonction `gain_max(v)` qui prend en argument une liste non vide `v` et renvoie le gain $G(0, n-1)$ du joueur A, en $O(n^2)$.

```
def gain_max(v):
    n = len(v)
    G = [[0 for j in range(n)] for i in range(n)]
    S = [[0 for j in range(n)] for i in range(n)]      # S[i][j] = v[i] + ... + v[j]
    for a in range(n):
        i = n - 1 - a                          # i décroissant
        G[i][i] = v[i]
        S[i][i] = v[i]
        for j in range(i + 1, n):              # j croissant
            S[i][j] = S[i][j - 1] + v[j]
            # l'adversaire commence sur le reste : on lui laisse le moins possible
            if G[i + 1][j] < G[i][j - 1]:
                G[i][j] = S[i][j] - G[i + 1][j]        # prendre v[i]
            else:
                G[i][j] = S[i][j] - G[i][j - 1]        # prendre v[j]
    return G[0][n - 1]
```

Réponse. $G(i,i) = v_i$. Si le joueur prend v_i, son adversaire commence sur $v_{i+1}, \dots, v_j$ et y gagne $G(i+1, j)$: il reste au joueur $S(i,j) - G(i+1,j)$. De même s'il prend v_j :

$$G(i,j) = S(i,j) - \min\left(G(i+1,j),\ G(i,j-1)\right).$$

Les sous-problèmes sont des intervalles, comme pour la chaîne de matrices : même tableau triangulaire, mêmes ordres de remplissage, mais chaque case n'en lit que deux, celle du dessous et celle de gauche (ici pour la rangée $8, 15, 3, 7$), d'où $O(n^2)$ en temps et en mémoire.



$S(0,3) = 8 + 15 + 3 + 7 = 33$

$G(0,3) = 33 - \min(18, 11) = 22$

```python
# Tests : chaque ligne doit afficher True.

print(gain_max([8, 15, 3, 7]) == 22, gain_max([4]) == 4, gain_max([1, 2]) == 2)
print(gain_max([3, 9, 1, 2]) == 11, gain_max([2, 2, 2, 2]) == 4)
print(gain_max([1, 100, 1, 1, 1]) == 3, gain_max([2, 1, 2]) == 3)
print(gain_max([(7 * k) % 13 + 1 for k in range(300)]) == 1065)     # 300 pièces
```

```
True True True
True True
True True
True
```