
TP — Révisions sur la récursivité

— Corrigé

L'objectif de ce TP est de réviser la programmation récursive. On respectera les consignes suivantes :

- on évitera autant que possible les boucles `for` et `while`, en leur préférant des fonctions récursives : une boucle `for` reste permise pour parcourir les **choix possibles** à une étape (les quatre quarts d'un échiquier, les valeurs possibles d'une case de Sudoku, les colonnes d'une ligne...), et quand l'énoncé l'autorise explicitement,
- les listes en compréhension sont autorisées,
- il est au contraire autorisé, et même conseillé, d'écrire des fonctions récursives auxiliaires,
- sauf mention contraire, les fonctions ne doivent pas modifier les listes passées en argument,
- on cherchera dès que possible à obtenir des complexités « raisonnables ».

Après chaque fonction à écrire, une cellule de tests est fournie : une fois la fonction écrite, l'exécuter, elle ne doit afficher que des `True`.

Python limite par défaut à 1000 le nombre d'appels imbriqués. La cellule suivante relève cette limite et importe le module `dessins.py` (à placer dans le même dossier que ce notebook) : l'exécuter avant toute chose. La plupart des fonctions `dessiner_...` de ce module dessinent vos résultats et les **vérifient** : le titre du dessin est vert si le résultat est correct, rouge sinon, et les erreurs sont marquées en rouge.

Sommaire

- Un exemple
- I. Récursivité et « diviser pour régner »
 - 1. Quelques études de complexité classiques (Q1 à Q4)
 - 2. Recherche dichotomique (Q5 et Q6)
 - 3. Multiplication rapide de polynômes (Q7 à Q10)
 - 4. Triangle de Pascal modulo 2 (Q11 et Q12)
 - 5. Pavage par des triominos (bonus) (Q13 et Q14)
 - 6. Tri rapide, sélection et médiane en temps linéaire (Q15 à Q19)
- II. Mémoïsation
 - 7. Échange de shokobons (Q20 et Q21)
 - 8. La suite de Syracuse (Projet Euler 14) (Q22)
 - 9. Chemins dans une grille à trous (Q23 et Q24)
 - 10. Parenthésages et nombres de Catalan (Q25 à Q29)
 - 11. Partitions d'un entier (Projet Euler 76) (Q30 et Q31)
- III. Backtracking
 - 12. Le problème des n reines (Q32 à Q35)
 - 13. Sudoku (Q36 à Q38)
 - 14. Logimages (inspiré de X-ENS 2024) (Q39 à Q45)

```
import sys

sys.setrecursionlimit(10**4)  # au lieu de 1000 par défaut

import numpy as np
from dessins import *
```

Un exemple

Pour calculer la somme des valeurs d'une liste, comparons les quatre codes suivants.

```
def somme_iterative(l):
    S = 0
    for i in range(len(l)):
        S = S + l[i]
    return S
```

```
def somme_recursive_slicing(l):
    if len(l) == 0:
        return 0
    else:
        return l[0] + somme_recursive_slicing(l[1:])
```

```
def somme_recursive_modifie(l):
    if len(l) == 0:
        return 0
    else:
        x = l.pop()
        return x + somme_recursive_modifie(l)
```

```
def somme_recursive_aux(l, j):
    if j == len(l):
        return 0
    else:
        return l[j] + somme_recursive_aux(l, j + 1)

def somme_recursive(l):
    return somme_recursive_aux(l, 0)
```

- `somme_iterative` est itérative : elle ne respecte pas la consigne d'éviter les boucles,
 - `somme_recursive_slicing` est récursive, mais de complexité $O(n^2)$: à chaque appel, `l[1:]` crée une nouvelle liste de taille $n-1$, $n-2$, ...,
 - `somme_recursive_modifie` est récursive et de complexité $O(n)$ (`pop()` sans argument retire le dernier élément en $O(1)$), mais elle **vide** la liste passée en argument,
 - `somme_recursive` est récursive, de complexité $O(n)$, et ne modifie pas la liste : c'est la seule qui respecte toutes les consignes. L'indice `j` de la fonction auxiliaire joue le rôle du *slicing* sans en payer le coût.
-

I. Récursivité et « diviser pour régner »

Rappel. Une fonction récursive s'appelle elle-même avec d'autres arguments. Il faut :

- un ou plusieurs **cas de base** (ou conditions d'arrêt), traités sans appel récursif,
- des appels récursifs sur des arguments qui se rapprochent d'un cas de base, ce qu'on justifie par un **variant** : un entier positif qui décroît strictement à chaque appel (il garantit la **terminaison**),
- pour la **complexité**, une relation de récurrence sur le coût $C(n)$.

Pour la complexité, on rappelle qu'à partir de l'« équation maître » $C(n) = aC(n/b) + O(n^d)$, on peut en déduire les complexités suivantes (cf. cours de première année) :

relation	complexité
$C(n) = C(n-r) + O(1)$	$O(n)$
$C(n) = C(n-r) + O(n)$	$O(n^2)$
$C(n) = qC(n-1) + O(1)$	$O(q^n)$
$C(n) = C(n-1) + C(n-2) + O(1)$	$O(\varphi^n)$, avec $\varphi = \frac{1+\sqrt{5}}{2}$
$C(n) = C(n/b) + O(1)$	$O(\ln(n))$
$C(n) = bC(n/b) + O(n)$	$O(n \ln(n))$
$C(n) = aC(n/b) + O(n)$ avec $a > b$	$O(n^{\log_b a})$

où $r \geq 1$, $q \geq 2$, a et $b \geq 2$ sont des constantes.

Par exemple pour l'écriture d'un entier $n \geq 0$ en base b (avec $b \geq 2$) : le dernier chiffre est $n \% b$, et les précédents sont ceux de $n // b$. On a donc le programme récursif suivant :

```
def ecriture(n, b):
    if n < b:
        # cas de base : un seul chiffre
        return [n]
    else:
        # appel sur n // b < n
        l = ecriture(n // b, b)
        l.append(n % b)
        return l

print(ecriture(2026, 10), ecriture(2026, 2), ecriture(2026, 16), ecriture(0, 2))
```

[2, 0, 2, 6] [1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0] [7, 14, 10] [0]

- *Variant* : l'entier naturel n , qui décroît strictement ($n // b < n$ pour $n \geq b$),
- *Correction* : si $ecriture(n // b, b)$ renvoie l'écriture de $n // b$, en lui ajoutant $n \% b$ on obtient celle de n , car $n = b \cdot (n // b) + (n \% b)$,
- *Complexité* : en notant $C(n)$ le coût de $ecriture(n, b)$, on a l'équation $C(n) = C(n//b) + O(1)$, donc $C(n) = O(\ln(n))$.

1. Quelques études de complexité classiques

Question 1. Quelles sont les complexités des fonctions suivantes ?

```
def fibo_affreux(n):
    if n == 0:
        return 0
```

```

elif n == 1:
    return 1
else:
    return fibo_affreux(n-1) + fibo_affreux(n-2)

```

```

def fibo_aux(n):
    if n == 0:
        return (0, 1)
    else:
        a, b = fibo_aux(n - 1)
        return (b, a + b)

def fibo(n):
    return fibo_aux(n)[0]

```

```

def fibo_argh(n):
    if n == 0:
        return (0, 1)
    else:
        return (fibo_argh(n - 1)[1], fibo_argh(n - 1)[0] + fibo_argh(n - 1)[1])

```

Réponse. On note $C(n)$ le nombre d'opérations élémentaires pour l'argument n (additions en $O(1)$).

- `fibo_affreux` : $C(n) = C(n-1) + C(n-2) + O(1)$. La suite C se comporte comme la suite de Fibonacci elle-même : $C(n) = O(\varphi^n)$ avec $\varphi = \frac{1+\sqrt{5}}{2} \approx 1,6$. Plus précisément, le nombre d'appels est $2F_{n+1} - 1$. Les mêmes valeurs sont recalculées un nombre exponentiel de fois.
- `fibo` : `fibo_aux` fait **un seul** appel récursif, $C(n) = C(n-1) + O(1)$, d'où $C(n) = O(n)$ (et une profondeur de récursion n). Renvoyer le couple (F_n, F_{n+1}) évite de recalculer F_{n-1} .
- `fibo_argh` : même idée que `fibo`, mais le couple est recalculé à chaque utilisation : **trois** appels à `fibo_argh(n-1)`, donc $C(n) = 3C(n-1) + O(1)$ et $C(n) = O(3^n)$, pire encore que `fibo_affreux` ! Il suffit de stocker le résultat dans une variable, comme dans `fibo_aux`.

Question 2. En supposant que la multiplication de deux entiers est de complexité $O(1)$, et en remarquant que la suite de Fibonacci $(F_n)_{n \in \mathbb{N}}$ vérifie

$$\forall n \in \mathbb{N}, \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix},$$

écrire une fonction récursive `fibo_rapide(n)` qui prend en argument un entier naturel n et renvoie F_n , avec une complexité $O(\ln n)$. On représentera les matrices par des listes de listes d'entiers Python, et non par des tableaux numpy : leurs entiers sur 64 bits débordent, et F_{100} serait faux.

Réponse. En notant $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, on a $\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = M^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, donc F_n est le coefficient $(0, 1)$ de M^n . On calcule M^n par **exponentiation rapide** : $M^n = (M^{\lfloor n/2 \rfloor})^2$ si n est pair, et $M^n = (M^{\lfloor n/2 \rfloor})^2 M$ si n est impair.

L'hypothèse « multiplication en $O(1)$ » est discutable ici : F_n s'écrit avec environ $0,7n$ bits, et Python manipule des entiers de taille arbitraire. La complexité $O(\ln n)$ compte des **opérations arithmétiques**, pas des opérations sur les bits.

```

def produit(A, B):
    """Produit de deux matrices 2 x 2."""
    [[a, b], [c, d]] = A
    [[e, f], [g, h]] = B
    return [[a * e + b * g, a * f + b * h],
            [c * e + d * g, c * f + d * h]]

def puissance(A, n):
    if n == 0:
        return [[1, 0], [0, 1]]
    B = puissance(A, n // 2) # un SEUL appel récursif
    C = produit(B, B)
    if n % 2 == 1:
        C = produit(C, A)
    return C

def fibo_rapide(n):
    return puissance([[0, 1], [1, 1]], n)[0][1]

# C(n) = C(n // 2) + O(1), d'où C(n) = O(ln n).
# Écrire produit(puissance(A, n // 2), puissance(A, n // 2)) donnerait
# C(n) = 2 C(n // 2) + O(1), soit O(n) : on perdrait tout le bénéfice.

```

```

# Tests : chaque ligne doit afficher True.
print(fibo_rapide(0) == 0, fibo_rapide(1) == 1, fibo_rapide(2) == 1)
print([fibo_rapide(n) for n in range(20)] == [fibonacci(n) for n in range(20)])
print(fibo_rapide(100) == 354224848179261915075)

```

True True True
True
True

Question 3. Écrire une fonction récursive `pgcd(a, b)` qui prend en arguments deux entiers naturels `a` et `b` non tous deux nuls et renvoie leur `pgcd`.

```

def pgcd(a, b):
    if b == 0:
        return a
    else:
        return pgcd(b, a % b) # Euclide : pgcd(a, b) = pgcd(b, a % b)

```

```

# Tests : chaque ligne doit afficher True.
print(pgcd(12, 18) == 6, pgcd(18, 12) == 6, pgcd(17, 5) == 1)
print(pgcd(7, 0) == 7, pgcd(0, 7) == 7, pgcd(9, 9) == 9) # cas limites
print(pgcd(2**10 * 3**5, 2**4 * 3**8 * 5) == 2**4 * 3**5)

```

True True True
True True True
True

Question 4 (bonus). Déterminer la complexité de votre fonction `pgcd(a, b)` en fonction de `b`, dans le cas où `a < b`.

Indication : on pourra remarquer, et démontrer, que deux termes consécutifs de la suite de Fibonacci constituent le « pire des cas ».

Réponse. Chaque appel coûte $O(1)$, il s'agit de compter les appels, c'est-à-dire les divisions euclidiennes effectuées. Si $a < b$, le premier appel échange les arguments : $\text{pgcd}(a, b)$ appelle $\text{pgcd}(b, a)$. On étudie donc $\text{pgcd}(a, b)$ avec $a > b \geq 1$.

Lemme. Si $a > b \geq 1$ et si $\text{pgcd}(a, b)$ effectue $k \geq 1$ divisions, alors $a \geq F_{k+2}$ et $b \geq F_{k+1}$.

Démonstration, par récurrence sur k .

- $k = 1$: $b \geq 1 = F_2$ et $a > b \geq 1$ donc $a \geq 2 = F_3$.
- Si $\text{pgcd}(a, b)$ effectue $k + 1 \geq 2$ divisions, notons $a = qb + r$ la première. Comme une autre division suit, $r \neq 0$, et $\text{pgcd}(b, r)$ effectue k divisions avec $b > r \geq 1$. Par hypothèse de récurrence, $b \geq F_{k+2}$ et $r \geq F_{k+1}$. Comme $a > b$, on a $q \geq 1$, donc $a \geq b + r \geq F_{k+2} + F_{k+1} = F_{k+3}$. $\square$

Par ailleurs, une récurrence immédiate donne $F_{k+1} \geq \varphi^{k-1}$ pour tout $k \geq 0$ (car $\varphi^2 = \varphi + 1$). Donc $b \geq \varphi^{k-1}$, soit $k \leq 1 + \log_{\varphi} b$.

Conclusion. Pour $a < b$, l'appel $\text{pgcd}(a, b)$ effectue au plus $1 + (1 + \log_{\varphi} a) \leq 2 + \log_{\varphi} b$ divisions : la complexité est $O(\ln b)$.

Cette borne est atteinte à une constante près : pour $k \geq 2$, $F_{k+2} = F_{k+1} + F_k$ avec $0 < F_k < F_{k+1}$, donc l'appel sur (F_{k+2}, F_{k+1}) se ramène à l'appel sur (F_{k+1}, F_k) : de proche en proche, il faut exactement k divisions, et deux termes consécutifs de la suite de Fibonacci réalisent bien le pire des cas.

2. Recherche dichotomique

Question 5. Écrire une fonction récursive $\text{dicho}(L, x)$ qui prend en arguments une liste triée L et une valeur x , et renvoie True si x appartient à L et False sinon, en $O(\ln n)$ où n est la longueur de L . On n'utilisera pas de tranches : pourquoi?

```
def dicho_aux(L, x, g, d):
    """x appartient-il à L[g], ..., L[d - 1] ?"""
    if g >= d:
        return False
    m = (g + d) // 2
    if L[m] == x:
        return True
    elif L[m] < x:
        return dicho_aux(L, x, m + 1, d)
    else:
        return dicho_aux(L, x, g, m)

def dicho(L, x):
    return dicho_aux(L, x, 0, len(L))

# d - g passe à au plus ceil((d - g) / 2) à chaque appel : O(ln n) appels en O(1).
# Avec des tranches L[m + 1:], chaque appel copierait la moitié de la liste :
# n/2 + n/4 + ... = O(n), et on perdrait l'intérêt de la dichotomie.
```

```
# Tests : chaque ligne doit afficher True.
L = [1, 3, 3, 5, 8, 13, 21]
print(dicho(L, 1), dicho(L, 3), dicho(L, 21), dicho(L, 8))
```

```
print(not dico(L, 0), not dico(L, 4), not dico(L, 22), not dico([], 5))
print(dico(list(range(0, 10**6, 2)), 123456))
print(not dico(list(range(0, 10**6, 2)), 12345))
```

True True True True
 True True True True
 True
 True

Question 6. Écrire une fonction récursive `racine_entiere(n)` qui prend en argument un entier naturel n et renvoie la partie entière de $\sqrt{n}$, par dichotomie et **sans calcul flottant**, en effectuant $O(\ln n)$ opérations arithmétiques. Pourquoi éviter `int(n ** 0.5)` ?

```
def racine_aux(n, g, d):
    """Partie entière de la racine de n, sachant que g * g <= n < d * d."""
    if d - g == 1:
        return g
    m = (g + d) // 2          # g < m < d
    if m * m <= n:
        return racine_aux(n, m, d)
    else:
        return racine_aux(n, g, m)

def racine_entiere(n):
    return racine_aux(n, 0, n + 1)  # 0 <= n < (n + 1)^2

# Variant : d - g, entier positif qui décroît strictement (g < m < d),
# et qui passe à au plus ceil((d - g) / 2) à chaque appel : O(ln n) appels.
```

Réponse. Les flottants n'ont que 53 bits de mantisse : pour de grands entiers, `n ** 0.5` est arrondi et `int(n ** 0.5)` peut être faux. Par exemple, pour $n = 10^{30} - 1$, on obtient 10^{15} au lieu de $10^{15} - 1$.

```
# Tests : chaque ligne doit afficher True.
print([racine_entiere(n) for n in range(10)] == [0, 1, 1, 1, 2, 2, 2, 2, 2, 3])
print(racine_entiere(10**30) == 10**15)
print(racine_entiere(10**30 - 1) == 10**15 - 1)
print(racine_entiere(2**100 + 1) == 2**50)
```

True
 True
 True
 True

3. Multiplication rapide de polynômes

Un polynôme $P = a_0 + a_1X + \dots + a_{n-1}X^{n-1}$ est représenté par le **tableau numpy** `np.array([a0, a1, ..., a(n-1)])` de ses n coefficients, du degré 0 au degré $n - 1$. Le produit d'un polynôme à n coefficients par un polynôme à p coefficients en a $n + p - 1$.

Rappels sur numpy (module importé au début du notebook sous le nom `np`) :

— `np.zeros(n)` crée un tableau de n zéros,

- pour deux tableaux P et Q de même longueur, P + Q et P - Q calculent la somme et la différence **coefficient par coefficient**, en $O(n)$,
- la tranche P[i:j] est une **vue** sur les éléments P[i], ..., P[j - 1] : elle ne recopie rien et se crée en $O(1)$, mais modifier la vue modifie aussi P,
- R[i:j] += S ajoute le tableau S (de longueur j - i) aux éléments R[i], ..., R[j - 1],
- A == B compare deux tableaux coefficient par coefficient et renvoie un tableau de booléens : pour savoir si deux tableaux sont égaux, on utilise np.array_equal(A, B).

Question 7. Écrire une fonction produit_naif(P, Q) qui prend en arguments deux polynômes P et Q de longueurs quelconques et renvoie leur produit, calculé avec deux boucles imbriquées. Quelle est sa complexité ?

```
def produit_naif(P, Q):
    R = np.zeros(len(P) + len(Q) - 1)
    for i in range(len(P)):
        for j in range(len(Q)):
            R[i + j] = R[i + j] + P[i] * Q[j]
    return R

# len(P) * len(Q) multiplications et additions : O(n^2) pour deux
# polynômes à n coefficients.
```

```
# Tests : chaque ligne doit afficher True.
un, a, b = np.array([1, 1]), np.array([1, 2, 3]), np.array([4, 5])
print(np.array_equal(produit_naif(un, un), [1, 2, 1]))
print(np.array_equal(produit_naif(a, b), [4, 13, 22, 15]))
print(np.array_equal(produit_naif(np.array([2]), np.array([3, 1])), [6, 2]))
```

True
True
True

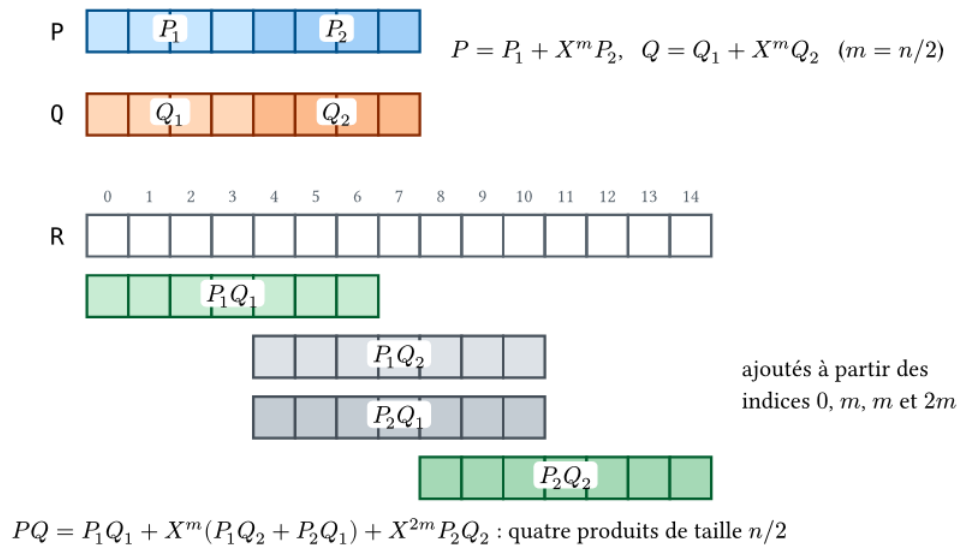
Pour « diviser pour régner », on coupe un polynôme à n coefficients en deux moitiés : $P = P_1 + X^m P_2$ avec $m = n/2$, où P_1 est formé des m premiers coefficients ($P[:m]$) et P_2 des suivants ($P[m:]$). Pour que les moitiés aient toujours la même longueur, on se ramène à deux polynômes de **même longueur n , puissance de 2**, en les complétant par des coefficients nuls avec la fonction fournie ci-dessous : le produit est inchangé, à des coefficients nuls près à la fin, qu'on ne cherchera pas à enlever. Le produit a alors $2n - 1$ coefficients.

```
def completer(P, Q):
    """Complète P et Q par des zéros jusqu'à une même longueur 2^k."""
    N = 1
    while N < len(P) or N < len(Q):
        N = 2 * N
    P2 = np.zeros(N)
    P2[:len(P)] = P
    Q2 = np.zeros(N)
    Q2[:len(Q)] = Q
    return (P2, Q2)

print(completer(np.array([1, 2, 3]), np.array([4, 5])))
```

(array([1., 2., 3., 0.]), array([4., 5., 0., 0.]))

En écrivant $P = P_1 + X^m P_2$ et $Q = Q_1 + X^m Q_2$, on a $PQ = P_1 Q_1 + X^m (P_1 Q_2 + P_2 Q_1) + X^{2m} P_2 Q_2$: quatre produits de polynômes de longueur $n/2$ (donc à $n - 1$ coefficients), ajoutés dans le résultat à partir des indices 0, m et $2m = n$.



Question 8. Écrire une fonction récursive `produit_dpr(P, Q)` qui prend en arguments deux polynômes de même longueur n , puissance de 2, et renvoie leur produit (à $2n - 1$ coefficients), en effectuant **quatre** appels récursifs sur des polynômes de longueur $n/2$. Établir une relation de récurrence sur sa complexité $C(n)$, et la résoudre. Qu'a-t-on gagné par rapport à `produit_naif` ?

```
def produit_dpr(P, Q):
    n = len(P)
    if n == 1:
        return P * Q  # tableau à un coefficient
    m = n // 2
    P1, P2, Q1, Q2 = P[:m], P[m:], Q[:m], Q[m:]  # vues : O(1)
    R = np.zeros(2 * n - 1)
    R[0:n - 1] += produit_dpr(P1, Q1)
    R[m:m + n - 1] += produit_dpr(P1, Q2) + produit_dpr(P2, Q1)
    R[n:2 * n - 1] += produit_dpr(P2, Q2)
    return R
```

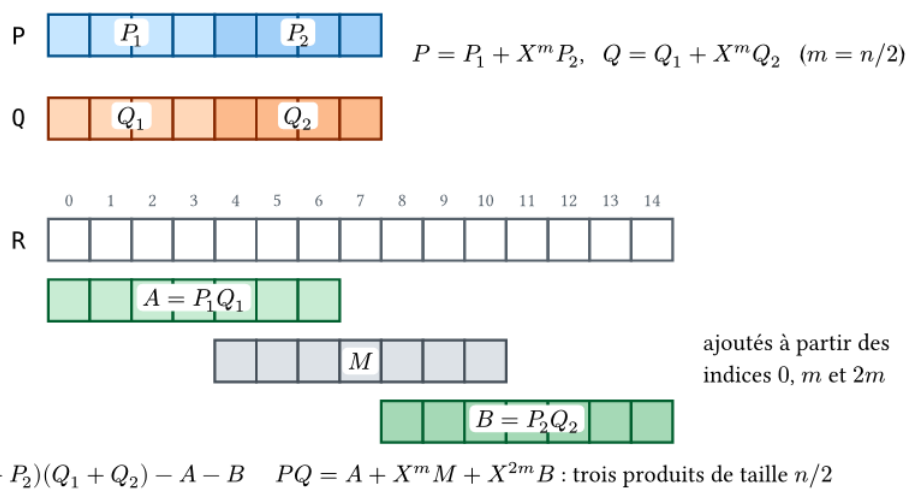
Réponse. Hors appels récursifs, on crée R et l'on fait des additions de tableaux de longueur $O(n)$: $C(n) = 4C(n/2) + O(n)$ (les tranches, qui sont des vues, ne coûtent que $O(1)$). C'est le cas $a = 4 > b = 2$ du rappel : $C(n) = O(n^{\log_2 4}) = O(n^2)$. Sur l'arbre des appels, le niveau j compte 4^j appels de taille $n/2^j$ et coûte $O(4^j \cdot n/2^j) = O(2^j n)$; la somme est dominée par le dernier niveau ($j = \log_2 n$), formé de n^2 multiplications de coefficients. On n'a **rien gagné** : diviser pour régner ne suffit pas, il faut faire moins de sous-problèmes.

```
# Tests : chaque ligne doit afficher True.
a, b = np.array([1, 2]), np.array([3, 4])
print(np.array_equal(produit_dpr(a, b), [3, 10, 8]))
for n in [1, 2, 4, 8, 16, 64]:
    P = np.random.randint(-9, 10, n)
    Q = np.random.randint(-9, 10, n)
    print(np.array_equal(produit_dpr(P, Q), produit_naif(P, Q)))
```

```
P, Q = completer(np.array([1, 1, 1]), np.array([1, -1]))
print(np.array_equal(produit_dpr(P, Q), [1, 0, 0, -1, 0, 0, 0]))
```

True
True
True
True
True
True
True
True

Karatsuba (1960). On pose $A = P_1Q_1$ et $B = P_2Q_2$. Comme $(P_1 + P_2)(Q_1 + Q_2) = A + P_1Q_2 + P_2Q_1 + B$, le terme du milieu s'obtient par $P_1Q_2 + P_2Q_1 = (P_1 + P_2)(Q_1 + Q_2) - A - B$: **trois** produits de polynômes de longueur $n/2$ suffisent.



Question 9. Écrire une fonction récursive `karatsuba(P, Q)` qui prend en arguments deux polynômes de même longueur n , puissance de 2, et renvoie leur produit, en n'effectuant que **trois** appels récursifs.

```
def karatsuba(P, Q):
    n = len(P)
    if n == 1:
        return P * Q
    m = n // 2
    P1, P2, Q1, Q2 = P[:m], P[m:], Q[:m], Q[m:]
    A = karatsuba(P1, Q1)
    B = karatsuba(P2, Q2)
    S = karatsuba(P1 + P2, Q1 + Q2)      # trois appels seulement
    R = np.zeros(2 * n - 1)
    R[0:n - 1] += A
    R[m:m + n - 1] += S - A - B          # P1 Q2 + P2 Q1
    R[n:2 * n - 1] += B
    return R
```

```
# Tests : chaque ligne doit afficher True.
a, b = np.array([1, 2]), np.array([3, 4])
print(np.array_equal(karatsuba(a, b), [3, 10, 8]))
```

```

for n in [1, 2, 4, 8, 16, 64, 256]:
    P = np.random.randint(-9, 10, n)
    Q = np.random.randint(-9, 10, n)
    print(np.array_equal(karatsuba(P, Q), produit_naif(P, Q)))
P, Q = completer(np.array([1, 1, 1]), np.array([1, -1]))
print(np.array_equal(karatsuba(P, Q), [1, 0, 0, -1, 0, 0, 0]))

```

True
 True
 True
 True
 True
 True
 True
 True
 True

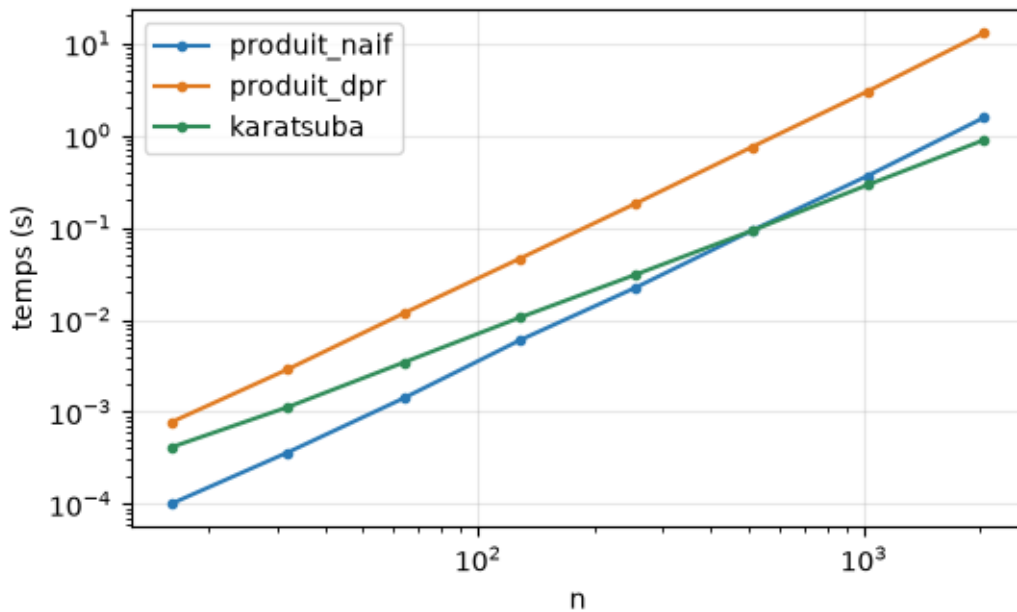
Question 10.

1. Établir une relation de récurrence sur la complexité $C(n)$ de karatsuba, et la résoudre.
2. Si l'on ne compte que les **multiplications** de coefficients, combien karatsuba en effectue-t-elle pour $n = 2^k$? Le fait de compter aussi les additions change-t-il l'ordre de grandeur ?
3. Exécuter la cellule suivante, qui trace les temps de calcul des trois fonctions pour $n = 2^k$ (environ 20 secondes), et commenter.

```

def aleatoire_paire(n):
    """Deux polynômes aléatoires à n coefficients entre 0 et 9."""
    return (np.random.randint(0, 10, n), np.random.randint(0, 10, n))
def naif(PQ):
    return produit_naif(PQ[0], PQ[1])
def dpr(PQ):
    return produit_dpr(PQ[0], PQ[1])
def kara(PQ):
    return karatsuba(PQ[0], PQ[1])
tracer_temps({"produit_naif": naif, "produit_dpr": dpr, "karatsuba": kara},
             [2 ** k for k in range(4, 12)], aleatoire_paire)

```



Réponse.

1. Hors appels récursifs, les additions et différences de tableaux et la création de R coûtent $O(n)$: $C(n) = 3C(n/2) + O(n)$. C'est le cas $a = 3 > b = 2$ du rappel : $C(n) = O(n^{\log_2 3}) \approx O(n^{1,59})$, au lieu de $O(n^2)$.
2. Le nombre $M(n)$ de multiplications vérifie $M(1) = 1$ et $M(n) = 3M(n/2)$, donc $M(2^k) = 3^k = n^{\log_2 3}$: 59 049 multiplications au lieu de 1 048 576 pour $n = 1024$. Compter les autres opérations ne change pas l'ordre de grandeur : le niveau j de l'arbre des appels compte 3^j appels de taille $n/2^j$, et y coûte $O(3^j \cdot \frac{n}{2^j}) = O(n(3/2)^j)$ hors appels récursifs. Ces coûts croissent géométriquement avec j (car $a = 3 > b = 2$), et leur somme est dominée par le dernier niveau, formé des 3^k multiplications : $O(n(3/2)^{\log_2 n}) = O(3^{\log_2 n}) = O(n^{\log_2 3})$. Pour le tri fusion, au contraire ($a = b = 2$), tous les niveaux coûtent autant, d'où le facteur $\ln n$.
3. karatsuba devient plus rapide que produit_naif à partir de quelques centaines de coefficients (0,8 s contre 1,5 s pour $n = 2048$), et l'écart se creuse : en échelle logarithmique, les pentes valent 1,59 et 2. produit_dpr est de loin la plus lente (12 s pour $n = 2048$) : elle a la même complexité que produit_naif, avec le coût des appels et des tableaux en plus. En pratique, on arrête la récursion dès que n est petit (par exemple $n \leq 32$) et l'on utilise alors un produit naïf.

Pour aller plus loin. Un entier écrit en base 10 est la valeur en $X = 10$ du polynôme de ses chiffres : multiplier deux entiers, c'est multiplier deux polynômes, puis propager les retenues. Python utilise d'ailleurs Karatsuba pour multiplier les très grands entiers. La même idée appliquée aux matrices par blocs donne l'algorithme de Strassen (1969) : 7 produits de blocs au lieu de 8, d'où $O(n^{\log_2 7}) \approx O(n^{2,81})$.

4. Triangle de Pascal modulo 2

On s'intéresse à la parité des coefficients binomiaux $\binom{n}{j}$.

Question 11. Écrire une fonction `pascal_mod2(n)` qui prend en argument un

entier naturel n et renvoie la liste des n premières lignes du triangle de Pascal **modulo 2**, c'est-à-dire la liste de listes $[[1], [1, 1], [1, 0, 1], [1, 1, 1, 1], [1, 0, 0, 0, 1], \dots]$: la ligne i est la liste $[(i \choose 0) \% 2, \dots, (i \choose i) \% 2]$. Quelle est sa complexité ?

```
def ligne_suivante(L):
    # Formule de Pascal, calculée modulo 2
    return [1] + [(L[j] + L[j + 1]) % 2 for j in range(len(L) - 1)] + [1]

def pascal_mod2(n):
    if n == 0:
        return []
    elif n == 1:
        return [[1]]
    else:
        lignes = pascal_mod2(n - 1) # liste neuve : on peut la modifier
        lignes.append(ligne_suivante(lignes[n - 2]))
        return lignes

# C(n) = C(n - 1) + O(n), d'où C(n) = O(n^2) : c'est la taille du résultat.

# Tests : chaque ligne doit afficher True.
print(pascal_mod2(0) == [], pascal_mod2(1) == [[1]])
print(pascal_mod2(5) == [[1], [1, 1], [1, 0, 1], [1, 1, 1, 1], [1, 0, 0, 0, 1]])
print(pascal_mod2(20)[19]
      == [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1])
```

True True
True
True

Question 12. Écrire une fonction `afficher_pascal(n)` qui prend en argument un entier $n \geq 1$ et **affiche** avec `print` les lignes 0 à $n - 1$ du triangle de Pascal modulo 2, en représentant chaque coefficient impair par **▲** et chaque coefficient pair par une espace, les caractères d'une ligne étant séparés par une espace et la ligne i étant précédée de $n - 1 - i$ espaces. Par exemple, `afficher_pascal(4)` affiche :

```

      ▲
     ▲▲
    ▲  ▲
   ▲▲▲▲
```

Afficher le triangle pour $n = 32$. Que remarque-t-on ?

```
def motif(L, j):
    """Chaîne représentant L[j], ..., L[len(L) - 1], séparés par des espaces."""
    if L[j] == 1:
        c = "▲"
    else:
        c = " "
    if j == len(L) - 1:
        return c
    else:
        return c + " " + motif(L, j + 1)

def afficher_lignes(lignes, n, i):
```

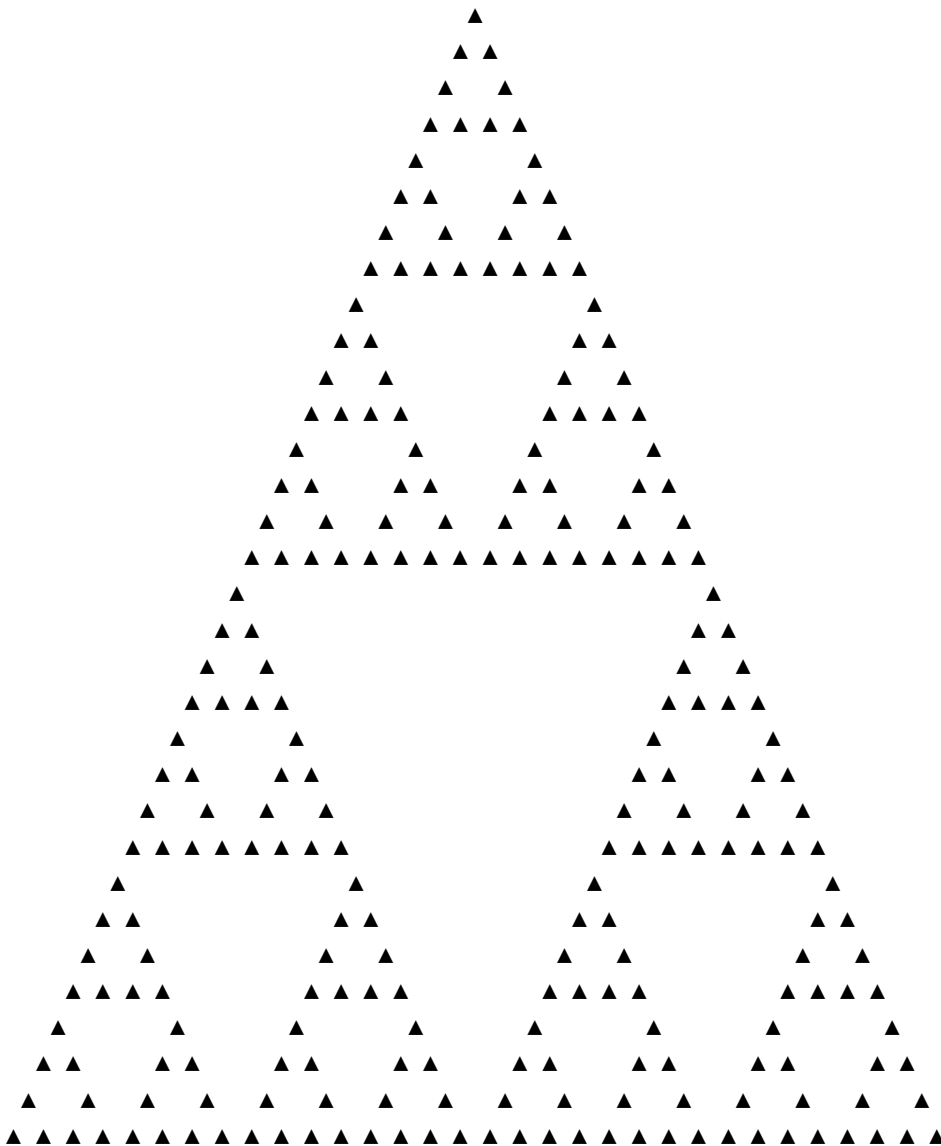
```

"""Affiche les lignes i à n - 1 du triangle."""
if i < n:
    print(" " * (n - 1 - i) + motif(lignes[i], 0))
    afficher_lignes(lignes, n, i + 1)

def afficher_pascal(n):
    afficher_lignes(pascal_mod2(n), n, 0)

afficher_pascal(32)

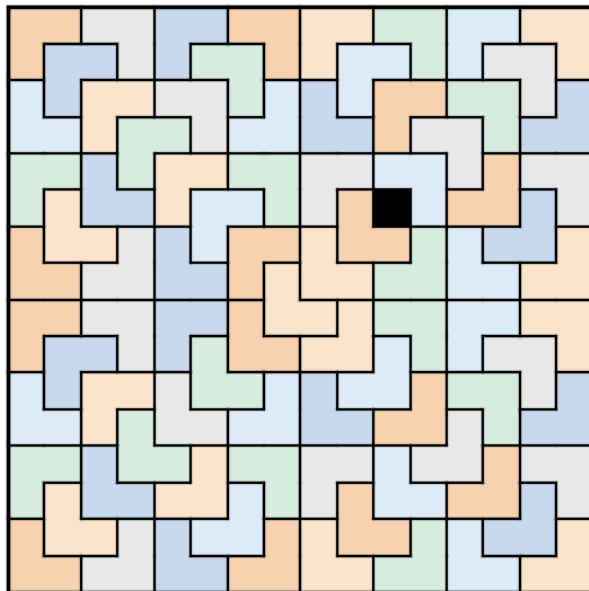
```



Réponse. On voit apparaître le **triangle de Sierpiński** : les lignes 0 à $2^{k+1} - 1$ sont formées d'une copie des lignes 0 à $2^k - 1$, au-dessus de deux copies côte à côte. Explication : dans $(\mathbb{Z}/2\mathbb{Z})[X]$, $(1 + X)^2 = 1 + 2X + X^2 = 1 + X^2$, et par récurrence $(1 + X)^{2^k} = 1 + X^{2^k}$. Pour $0 \leq i < 2^k$, on a donc $(1 + X)^{2^k + i} = (1 + X)^i + X^{2^k}(1 + X)^i$, où le premier terme est de degré $i < 2^k$: les deux termes ne se chevauchent pas. La ligne $2^k + i$ modulo 2 est formée de la ligne i , de $2^k - 1 - i$ zéros, puis de nouveau de la ligne i .

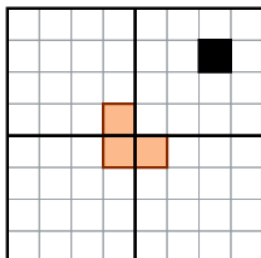
5. Pavage par des triominos (bonus)

Un **triomino** est une pièce en forme de L formée de trois cases. On veut paver par des triominos un échiquier $2^n \times 2^n$ privé d'une case quelconque. Voici par exemple un pavage d'un échiquier 16×16 privé d'une case (en noir).



Question 13. Montrer par récurrence sur n qu'un tel pavage existe toujours, quelle que soit la case retirée. Combien de triominos utilise-t-il ?

Indication : on pourra regarder le schéma suivant.



Réponse. Pour $n = 0$, l'échiquier privé de sa seule case est vide. Pour $n \geq 1$, on découpe l'échiquier en quatre quarts $2^{n-1} \times 2^{n-1}$, la case retirée est dans l'un d'eux. On pose un triomino sur les trois cases centrales qui appartiennent aux trois **autres** quarts (figure). Chaque quart est alors un échiquier $2^{n-1} \times 2^{n-1}$ privé d'une case, que l'on pave par hypothèse de récurrence. Le pavage couvre $4^n - 1$ cases, donc il utilise $\frac{4^n - 1}{3}$ triominos (et $4^n - 1$ est bien divisible par 3, car $4 \equiv 1 \pmod{3}$). La preuve est un algorithme récursif.

Question 14. Écrire une fonction `pavage(n, i, j)` qui prend en arguments un entier naturel n et les coordonnées (i, j) d'une case, et renvoie un pavage de l'échiquier $2^n \times 2^n$ privé de la case (i, j) , sous forme d'une liste de listes G : $G[i][j]$ vaut -1 , et chaque autre case contient le numéro (à partir de 1) du triomino qui la couvre. Quelle est sa complexité ?

Indication : on pourra écrire une fonction récursive auxiliaire `pavage_aux(G, x, y, t, ti, tj, num)` qui pave le carré de coin supérieur gauche (x, y) (ligne x , colonne y) et de côté t privé de la case (t_i, t_j) (déjà couverte, et repérée par ses coordonnées dans la grille G).

entière), en numérotant les triominos à partir de num, et qui renvoie le premier numéro non utilisé.

```
def pavage_aux(G, x, y, t, ti, tj, num):
    if t == 1:
        return num          # la seule case est déjà couverte
    m = t // 2
    coins = [(x, y), (x, y + m), (x + m, y), (x + m, y + m)]
    centres = [(x + m - 1, y + m - 1), (x + m - 1, y + m),
               (x + m, y + m - 1), (x + m, y + m)]
    # numéro du quart contenant la case déjà couverte
    k = 0
    if ti >= x + m:
        k = k + 2
    if tj >= y + m:
        k = k + 1
    trous = []
    for q in range(4):
        if q == k:
            trous.append((ti, tj))
        else:
            # le triomino central couvre ce centre
            G[centres[q][0]][centres[q][1]] = num
            trous.append(centres[q])
    num = num + 1
    for q in range(4):
        (a, b), (u, v) = coins[q], trous[q]
        num = pavage_aux(G, a, b, m, u, v, num)
    return num

def pavage(n, i, j):
    t = 2 ** n
    G = [[0 for b in range(t)] for a in range(t)]
    G[i][j] = -1
    pavage_aux(G, 0, 0, t, i, j, 1)
    return G
```

$C(t) = 4 C(t / 2) + O(1)$: $C(t) = O(t^2)$, linéaire en le nombre de cases.

Tests : chaque ligne doit afficher True.

```
def est_pavage(G, i, j):
    t = len(G)
    cases = {}
    for x in range(t):
        for y in range(t):
            cases.setdefault(G[x][y], []).append((x, y))
    if cases.get(-1) != [(i, j)]:
        return False
    for v, L in cases.items():
        if v != -1:
            xs, ys = [x for x, y in L], [y for x, y in L]
            if len(L) != 3 or max(xs) - min(xs) != 1 or max(ys) - min(ys) != 1:
                return False
    return len(cases) - 1 == (t * t - 1) // 3
```

```

print(pavage(0, 0, 0) == [[-1]], est_pavage(pavage(1, 0, 1), 0, 1))
print(all(est_pavage(pavage(3, i, j), i, j)
            for i in range(8) for j in range(8)))
print(est_pavage(pavage(6, 37, 12), 37, 12))

```

True True

True

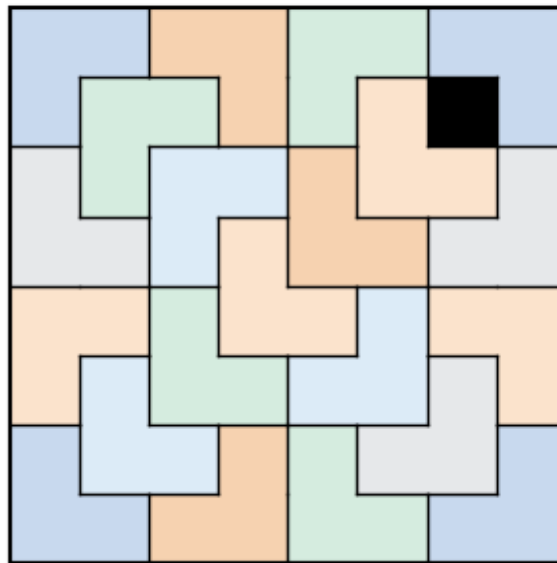
True

```

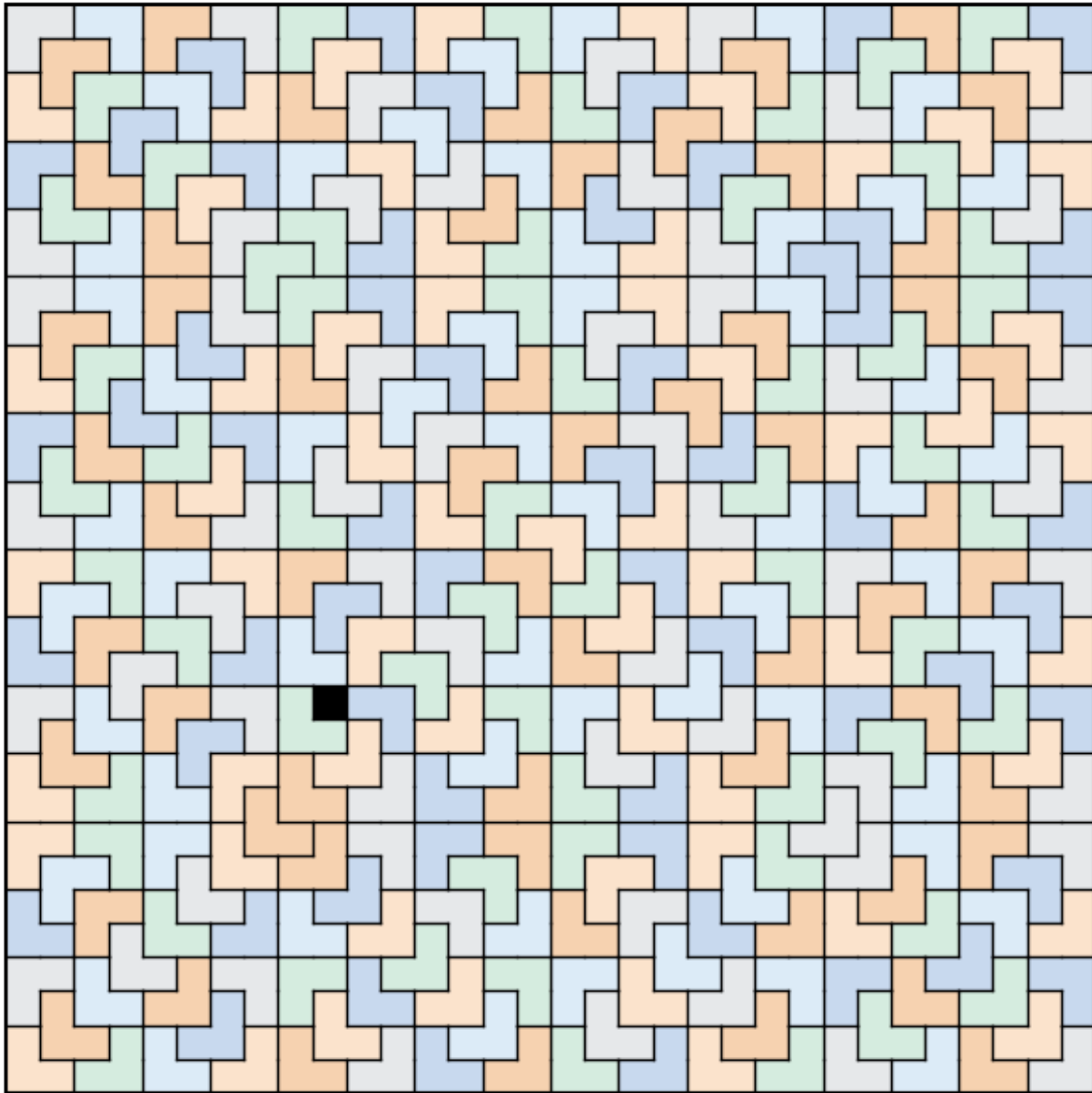
# Visualisation : titre vert si le résultat est correct, rouge sinon.
dessiner_triominos(pavage(3, 1, 6))
dessiner_triominos(pavage(5, 20, 9))

```

21 triominos, pavage valide



341 triominos, pavage valide



6. Tri rapide, sélection et médiane en temps linéaire

Dans cette partie, on s'autorise des boucles `for` dans la fonction `partition` et dans le calcul des médianes des blocs.

Question 15. Écrire une fonction `partition(L, p)` qui prend en arguments une liste `L` et une valeur `p`, et renvoie le triplet (L_i, L_e, L_s) des listes des éléments de `L` respectivement strictement inférieurs, égaux et strictement supérieurs à `p`, dans l'ordre où ils apparaissent dans `L`, en $O(n)$.

```
def partition(L, p):
    Li, Le, Ls = [], [], []
    for x in L:
        if x < p:
            Li.append(x)
```

```

    elif x == p:
        Le.append(x)
    else:
        Ls.append(x)
return Li, Le, Ls

```

```

# Tests : chaque ligne doit afficher True.
print(partition([4, 3, 2, 1, 5], 3) == ([2, 1], [3], [4, 5]))
print(partition([4, 3, 2, 1, 5], 6) == ([4, 3, 2, 1, 5], [], []))
print(partition([2, 7, 2, 1, 2], 2) == ([1], [2, 2, 2], [7]))
print(partition([], 0) == ([], [], []))

```

```

True
True
True
True

```

Question 16. Le **tri rapide** d'une liste L de longueur au moins 2 choisit un **pivot** p (ici le premier élément de L), partitionne L autour de p , trie récursivement Li et Ls , puis renvoie leur concaténation avec Le intercalée.

1. Écrire une fonction récursive `tri_rapide(L)` qui prend en argument une liste L et renvoie une **nouvelle** liste, triée, formée des éléments de L . Justifier sa terminaison.
2. Montrer que sa complexité est $O(n^2)$, et donner une liste pour laquelle elle est effectivement de l'ordre de n^2 . Quelle est la profondeur de récursion de `tri_rapide(list(range(n)))`? Que se passe-t-il quand n dépasse la limite de récursion?
3. Quelle serait sa complexité si le pivot était toujours une médiane de la liste, calculée en $O(n)$?

```

def tri_rapide(L):
    if len(L) <= 1:
        return L.copy()      # nouvelle liste, pas L elle-même
    else:
        Li, Le, Ls = partition(L, L[0])
        return tri_rapide(Li) + Le + tri_rapide(Ls)

```

```

# Tests : chaque ligne doit afficher True.
print(tri_rapide([]) == [])
print(tri_rapide([3]) == [3])
print(tri_rapide([3, 1, 2]) == [1, 2, 3])
L = [5, 3, 8, 3, 1, 9, 2, 8, 8, 0]
print(tri_rapide(L) == [0, 1, 2, 3, 3, 5, 8, 8, 8, 9])
print(L == [5, 3, 8, 3, 1, 9, 2, 8, 8, 0])      # L n'est pas modifiée
from random import randint
R = [randint(0, 1000) for i in range(10**4)]
print(tri_rapide(R) == sorted(R))

```

```

True
True
True
True
True
True

```

Réponse.

1. Le pivot appartient à L , donc L_i n'est pas vide : L_i et L_s sont strictement plus courtes que L , et la longueur est un variant.
2. Partitionner et concaténer coûte $O(n)$, et L_i , L_s sont de longueurs k et ℓ avec $k + \ell \leq n - 1$. En notant $C(n)$ le coût maximal pour une liste de longueur au plus n , on obtient par récurrence $C(n) \leq \beta(n + (n - 1) + \dots + 1) = O(n^2)$, où β est une constante : chaque niveau de l'arbre des appels coûte $O(n)$, et il y a au plus n niveaux. Pour une liste **triée sans doublons**, L_i est toujours vide et L_s de longueur $n - 1$: le coût est $\beta(n + (n - 1) + \dots + 1)$, de l'ordre de n^2 . Pour `list(range(n))`, la profondeur de récursion est n : au-delà de la limite (ici 10^4), on obtient une `RecursionError`, après un calcul déjà quadratique.
3. On aurait, pour une constante α , $C(n) \leq 2C(\lfloor n/2 \rfloor) + \alpha n$, d'où $C(n) = O(n \ln n)$ (comme le tri fusion).

Question 17. Pour $0 \leq k < n$, le k -ième plus petit élément de L est l'élément d'indice k de la liste L triée. Pour $k = \lfloor n/2 \rfloor$, c'est la **médiane** de L . Écrire une fonction récursive `selection(L, k)` qui prend en arguments une liste L de longueur n et un entier $k \in \llbracket 0, n \rrbracket$, et renvoie le k -ième plus petit élément de L . Elle procédera comme le tri rapide (pivot $L[0]$), mais avec **un seul** appel récursif. Quelle est sa complexité dans le pire cas ?

```
def selection(L, k):
    Li, Le, Ls = partition(L, L[0])
    if k < len(Li):
        return selection(Li, k)
    elif k < len(Li) + len(Le):
        return L[0]
    else:
        return selection(Ls, k - len(Li) - len(Le))

# Pire cas (liste triée sans doublons, k = n - 1) : n + (n - 1) + ... = O(n^2),
# et une profondeur de récursion n. En moyenne sur les permutations,
# on peut montrer que la complexité est O(n), mais ce n'est pas garanti.
```

```
# Tests : chaque ligne doit afficher True.
from random import randint
L = [5, 3, 8, 3, 1, 9, 2, 8, 8, 0]
R = [randint(0, 1000) for i in range(10**4)]
print([selection(L, k) for k in range(10)] == [0, 1, 2, 3, 3, 5, 8, 8, 8, 9])
print(all(selection(R, k) == sorted(R)[k] for k in [0, 1, 5000, 9998, 9999]))
```

True

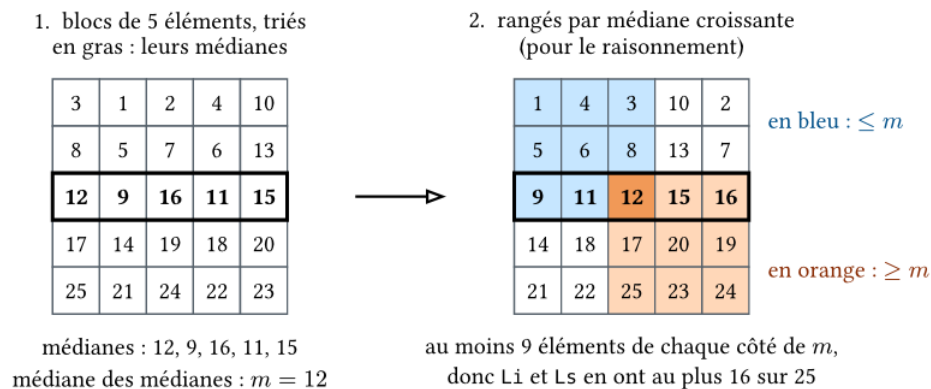
True

Médiane en temps linéaire (bonus)

Objectif : calculer la médiane d'une liste en temps linéaire, sans la trier. Avec `selection`, le calcul de la médiane est quadratique dans le pire cas, car le pivot $L[0]$ peut être proche d'une extrémité. Il faut donc un pivot qui ne soit jamais trop proche des extrémités. L'algorithme de la **médiane des médianes** (Blum, Floyd, Pratt, Rivest et Tarjan, 1973) choisit le pivot ainsi, pour un entier $B \geq 2$ fixé :

- on découpe L en $\lceil n/B \rceil$ blocs consécutifs de B éléments (le dernier pouvant être plus court),
- on calcule la médiane de chaque bloc, en le triant,

- le pivot est la médiane de la liste de ces médianes, calculée **récurivement** par le même algorithme.



L'appel récursif sur L_i ou L_s ne cherche pas une médiane : il faut savoir calculer le k -ième plus petit élément pour tout k .

Question 18. Écrire une fonction `selection_mom(L, k, B)` qui prend en arguments une liste L de longueur n , un entier $k \in \llbracket 0, n \rrbracket$ et un entier $B \geq 2$, et renvoie le k -ième plus petit élément de L . Elle procédera comme `selection`, mais avec ce choix de pivot, calculé par une fonction auxiliaire `pivot_mom(L, B)` qui prend en arguments une liste L et l'entier B et renvoie la médiane des médianes de ses blocs. Les listes d'au plus B éléments seront triées directement avec `tri_rapide`. En déduire une fonction `mediane(L)` qui prend en argument une liste non vide L et renvoie sa médiane, avec $B = 5$. Pourquoi a-t-on autorisé une boucle pour calculer les médianes des blocs ?

```
def pivot_mom(L, B):
    medianes = []
    for j in range((len(L) + B - 1) // B):      # ceil(n / B) blocs
        bloc = tri_rapide(L[B * j : B * j + B])
        medianes.append(bloc[len(bloc) // 2])
    return selection_mom(medianes, len(medianes) // 2, B)

def selection_mom(L, k, B):
    if len(L) <= B:
        return tri_rapide(L)[k]
    p = pivot_mom(L, B)
    Li, Le, Ls = partition(L, p)
    if k < len(Li):
        return selection_mom(Li, k, B)
    elif k < len(Li) + len(Le):
        return p
    else:
        return selection_mom(Ls, k - len(Li) - len(Le), B)

def mediane(L):
    return selection_mom(L, len(L) // 2, 5)

# Une fonction récursive parcourant les blocs aurait une profondeur n / B,
# trop grande dès que n / B dépasse la limite de récursion.
# Ici, la profondeur est O(ln n) pour tout B >= 2 (question suivante) :
# list(range(10**5)) ne pose plus de problème.
```

```
# Tests : chaque ligne doit afficher True.
from random import randint
L = [5, 3, 8, 3, 1, 9, 2, 8, 8, 0]
R = [randint(0, 1000) for i in range(10**4)]
print(mediane(L) == 5, mediane([7]) == 7, mediane(R) == sorted(R)[5000])
print([selection_mom(L, k, 5) for k in range(10)]
      == [0, 1, 2, 3, 3, 5, 8, 8, 8, 9])
print(all(selection_mom(R, k, B) == sorted(R)[k]
          for k in [0, 1, 5000, 9998, 9999] for B in [2, 3, 5, 7, 31]))
print(selection_mom(list(range(10**5)), 12345, 5) == 12345)
print(selection_mom(list(range(10**5, 0, -1)), 99999, 5) == 10**5)
```

True True True

True

True

True

True

```
# Illustration de la question suivante (pas de vérification ici).
from random import randint
dessiner_blocs([randint(0, 99) for i in range(45)], 5)
```

$n = 45, B = 5$, pivot $m = 50$
au moins 15 éléments $\leq m$ (bleu) et 14 éléments $\geq m$ (orange) garantis, sans comparer à m

4	17	30	11	40	3	15	42	15
7	32	37	16	42	38	46	49	51
16	37	43	43	50	53	59	60	63
67	71	44	57	59	66	74	90	82
70	89	63	85	76	89	82	93	83

Question 19. On note m le pivot choisi par `pivot_mom` pour une liste de longueur $n > B$.

1. Montrer qu'au moins $\lceil B/2 \rceil$ ($\lceil \lceil n/B \rceil / 2 \rceil - 1$) éléments de L sont inférieurs ou égaux à m , et de même pour les éléments supérieurs ou égaux. En déduire que L_i et L_s ont au plus $\left(1 - \frac{\lceil B/2 \rceil}{2B}\right)n + \lceil B/2 \rceil$ éléments.
2. On note $T(n)$ le coût maximal de `selection_mom` pour une liste de longueur au plus n . Justifier qu'il existe une constante $\gamma_B > 0$ telle que, pour $n > B$,

$$T(n) \leq T\left(\left\lceil \frac{n}{B} \right\rceil\right) + T\left(n - \left\lceil \frac{B}{2} \right\rceil \left(\left\lceil \frac{\lceil n/B \rceil}{2} \right\rceil - 1\right)\right) + \gamma_B n.$$

3. On admet qu'on peut négliger les parties entières et les constantes additives, c'est-à-dire remplacer cette inégalité par $T(n) \leq T\left(\frac{n}{B}\right) + T(\beta_B n) + \gamma_B n$, avec $\beta_B = 1 - \frac{\lceil B/2 \rceil}{2B}$. Montrer que si $\frac{1}{B} + \beta_B < 1$, alors $T(n) = O(n)$.
4. Pour quelles valeurs de B cette condition est-elle satisfaite ?

5. Comparer les temps d'exécution de `selection` et `selection_mom` (avec $B = 5$) sur une liste aléatoire et sur `list(range(800))`, puis ceux de `selection_mom` pour différentes valeurs de B . Commenter. On pourra utiliser `tracer_temps`, comme pour la multiplication de polynômes.

Réponse.

1. La médiane d'un bloc trié de longueur b est l'élément d'indice $\lfloor b/2 \rfloor$: elle est supérieure ou égale à $\lfloor b/2 \rfloor + 1$ éléments du bloc et inférieure ou égale à $b - \lfloor b/2 \rfloor = \lceil b/2 \rceil$ d'entre eux, soit au moins $\lceil B/2 \rceil$ dans les deux cas pour un bloc complet. De même, m est supérieur ou égal à au moins $\lceil \lceil n/B \rceil / 2 \rceil$ médianes de blocs, et inférieur ou égal à au moins $\lceil \lceil n/B \rceil / 2 \rceil$ d'entre elles. En écartant le dernier bloc, peut-être incomplet, au moins $\lceil B/2 \rceil (\lceil \lceil n/B \rceil / 2 \rceil - 1)$ éléments sont inférieurs ou égaux à m : ils ne sont pas dans `Li`. Comme $\lceil \lceil n/B \rceil / 2 \rceil \geq \frac{n}{2B}$, `Li` a au plus $n - \lceil B/2 \rceil (\frac{n}{2B} - 1) = \beta_B n + \lceil B/2 \rceil$ éléments. Le raisonnement est symétrique pour `Li`.
2. Trier les $\lceil n/B \rceil$ blocs coûte $O(B^2)$ par bloc, soit $O(Bn)$, et la partition coûte $O(n)$: en tout $\gamma_B n$. S'y ajoutent l'appel récursif sur les $\lceil n/B \rceil$ médianes, puis au plus un appel sur `Li` ou `Li`, de longueur majorée au point 1.
3. Montrons par récurrence forte que $T(n) \leq \lambda n$, avec $\lambda = \max(T(1), \dots, T(B), \frac{\gamma_B}{1 - 1/B - \beta_B})$. C'est vrai pour $n \leq B$, pour $n > B$,

$$T(n) \leq \lambda \frac{n}{B} + \lambda \beta_B n + \gamma_B n = ((\frac{1}{B} + \beta_B) \lambda + \gamma_B) n \leq \lambda n,$$

par le choix de λ . Les deux appels portent sur des tailles de somme $(\frac{1}{B} + \beta_B) n < n$: le coût de chaque niveau de l'arbre des appels décroît géométriquement.

4. $\frac{1}{B} + \beta_B < 1 \iff \frac{\lceil B/2 \rceil}{2B} > \frac{1}{B} \iff \lceil B/2 \rceil > 2 \iff B \geq 5$. Pour $B = 5$, $\frac{1}{5} + \frac{7}{10} = \frac{9}{10}$. Pour $B = 3$ ou $B = 4$, la somme vaut exactement 1 : chaque niveau de l'arbre des appels coûte $O(n)$, et on obtient seulement $O(n \ln n)$. Pour $B = 2$, la somme vaut $\frac{1}{2} + \frac{3}{4} > 1$: on n'obtient aucune borne linéaire.
5. Sur une liste aléatoire, `selection` est environ 10 fois plus rapide : la constante cachée dans le $O(n)$ de `selection_mom` est grande. Sur une liste triée, `selection` est quadratique (et lève une `RecursionError` dès que n dépasse la limite de récursion), alors que `selection_mom` reste linéaire. Pour $B \geq 5$, les temps varient peu, $B = 3$ est un peu plus lent (de l'ordre de 1,5 fois), ce qui illustre le point 4. Avec 10^4 éléments, les temps sont de l'ordre de 10 ms et donc bruités : mieux vaut mesurer sur 10^5 éléments. En pratique, on choisit plutôt un pivot **aléatoire**, ce qui donne une complexité linéaire **en moyenne** quelle que soit la liste.

```
import time
from random import randint
R = [randint(0, 10**9) for i in range(10**5)] # valeurs distinctes

for liste in [R, list(range(800))]:
    n = len(liste)
    debut = time.time()
    selection(liste, n // 2)
    milieu = time.time()
    selection_mom(liste, n // 2, 5)
    fin = time.time()
    print(n, "éléments :", milieu - debut, "s contre", fin - milieu, "s")

for B in [3, 5, 7, 15, 31]:
    debut = time.time()
    selection_mom(R, len(R) // 2, B)
```

```
print("B =", B, ":", time.time() - debut, "s")
```

100000 éléments : 0.009896278381347656 s contre 0.1492297649383545 s

800 éléments : 0.010574817657470703 s contre 0.0014204978942871094 s

B = 3 : 0.23727154731750488 s

B = 5 : 0.14792156219482422 s

B = 7 : 0.12425017356872559 s

B = 15 : 0.1188809871673584 s

B = 31 : 0.12149286270141602 s

Remarque : pourquoi ne pas écrire une version **en place**, qui permute les éléments d'une seule liste au lieu de créer Li, Le, Ls et les blocs ?

- La complexité temporelle est la même, $O(n)$: créer ces listes ne coûte pas plus que les parcourir pour partitionner.
- La mémoire supplémentaire reste $O(n)$ dans les deux cas : la version avec listes utilise $n + \beta_B n + \beta_B^2 n + \dots = O(n)$, et une version en place doit d'abord **copier** L pour ne pas la modifier. On a mesuré un pic de mémoire de 16 Mo pour la version avec listes contre 8 Mo pour la version en place, pour $n = 10^6$.
- En Python, la version en place est environ **5 fois plus lente** (11 s contre 2,3 s pour $n = 10^6$). Les append sont exécutés par l'interpréteur en C, alors que la gestion des indices et des échanges est écrite en Python.
- Le code en place est nettement plus délicat : il faut une partition en trois zones par échanges, et ranger les médianes des blocs au début de la liste.

La version **en place** prend son intérêt dans un langage compilé, ou quand la mémoire est vraiment limitée.

II. Mémoïsation

Rappel. Quand une fonction récursive est appelée de nombreuses fois avec les mêmes arguments, on range chaque résultat calculé et on le relit au lieu de le recalculer : c'est la **mémoïsation**.

7. Échange de shokobons

Un professeur d'ITC propose à ses élèves l'échange suivant. Un élève qui possède un paquet de n shokobons peut l'échanger contre **trois** paquets de $\lfloor n/2 \rfloor$, $\lfloor n/3 \rfloor$ et $\lfloor n/4 \rfloor$ shokobons, qu'il peut à leur tour échanger, ou bien garder ses n shokobons. Par exemple, un paquet de 12 shokobons s'échange contre des paquets de 6, 4 et 3 shokobons, soit 13 shokobons. Le nombre maximal de shokobons qu'on peut obtenir à partir d'un paquet de n shokobons vérifie

$$f(0) = 0 \quad \text{et} \quad f(n) = \max(n, f(\lfloor n/2 \rfloor) + f(\lfloor n/3 \rfloor) + f(\lfloor n/4 \rfloor)) \quad \text{pour } n \geq 1.$$

Question 20. Écrire une fonction récursive `shokobons_naif(n)` qui prend en argument un entier naturel n et renvoie $f(n)$.

```
def shokobons_naif(n):
    if n == 0:
        return 0
    echange = (shokobons_naif(n // 2) + shokobons_naif(n // 3)
               + shokobons_naif(n // 4))
    if echange > n:
        return echange
    return n

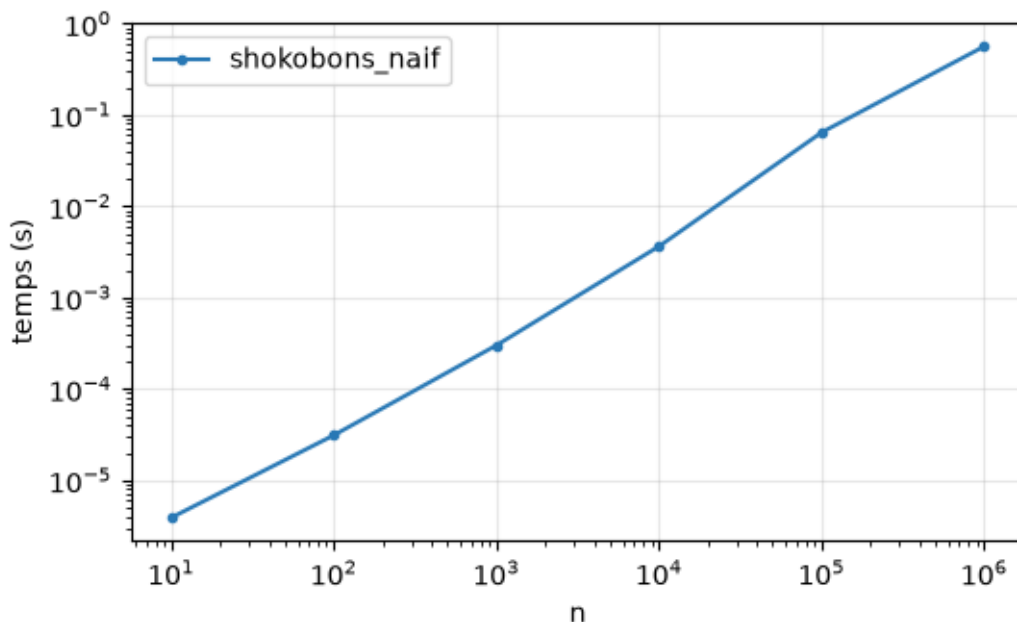
# Tests : chaque ligne doit afficher True.
print(shokobons_naif(0) == 0)
print(shokobons_naif(11) == 11)
print(shokobons_naif(12) == 13)
print(shokobons_naif(100) == 120, shokobons_naif(10**4) == 16615)
```

True
True
True
True True

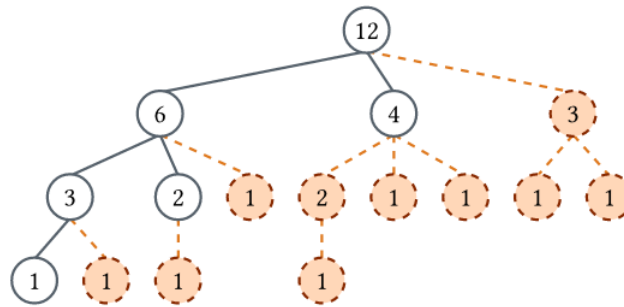
Exécuter la cellule suivante, qui trace le temps de calcul de `shokobons_naif(n)` pour n allant de 10 à 10^6 (quelques secondes).

```
def identite(n):
    return n

tailles = [10 ** k for k in range(1, 7)]
tracer_temps({"shokobons_naif": shokobons_naif}, tailles, identite)
```



En échelle logarithmique, la courbe est presque une droite, de pente $\alpha \approx 1,08$: le temps de calcul est de l'ordre de n^α , où α est la solution de $2^{-\alpha} + 3^{-\alpha} + 4^{-\alpha} = 1$ (le nombre d'appels $A(n)$ vérifie $A(n) = 1 + A(\lfloor n/2 \rfloor) + A(\lfloor n/3 \rfloor) + A(\lfloor n/4 \rfloor)$). Pour $n = 10^9$, il faudrait une vingtaine de minutes. Pour comprendre pourquoi, voici l'arbre des appels de `shokobons_naif(12)` :



en pointillé : valeur déjà calculée ailleurs dans l'arbre (les appels sur 0 ne sont pas dessinés)

Les mêmes valeurs sont calculées plusieurs fois : $f(3)$ deux fois, $f(2)$ deux fois, $f(1)$ neuf fois. Pour $n = 10^6$, la fonction fait près de 10 millions d'appels. Pourtant, elle n'est appelée que sur les entiers de la forme $\lfloor n/(2^i 3^j) \rfloor$, qui sont peu nombreux. On mémorise donc chaque valeur $f(k)$ calculée dans un dictionnaire `memo` dont les clés sont les entiers k :

```
def shokobons_memo(n, memo):
    if n not in memo:
        if n == 0:
            memo[n] = 0
        else:
            echange = (shokobons_memo(n // 2, memo)
                      + shokobons_memo(n // 3, memo)
                      + shokobons_memo(n // 4, memo))
            if echange > n:
                memo[n] = echange
            else:
                memo[n] = n
    return memo[n]

def shokobons(n):
    return shokobons_memo(n, {})

print(shokobons(12), shokobons(10**6), shokobons(10**9))
```

13 2566393 4243218150

Question 21. Commenter le code de `shokobons_memo` : que contient le dictionnaire `memo` ? À quoi sert le test `if n not in memo` ? Pourquoi `shokobons` crée-t-elle un dictionnaire vide, et pourquoi le passe-t-on en argument au lieu de le recréer à chaque appel ? Combien d'appels à `shokobons_memo` (appel initial compris) fait `shokobons(12)`, et combien d'entre eux font un calcul plutôt qu'une lecture dans `memo` ?

Réponse. `memo` associe à chaque entier k déjà rencontré la valeur $f(k)$. Le test `if n not in memo` détecte un argument jamais vu : on calcule alors $f(n)$, une seule fois, et on le range. Sinon, on relit directement la valeur rangée, sans aucun appel récursif. Le dictionnaire doit être **partagé** par tous les appels issus d'un même calcul : c'est pourquoi on le passe en argument (un dictionnaire recréé à chaque appel serait toujours vide). `shokobons` le crée vide au départ. Pour $n = 12$, seules les valeurs 12, 6, 4, 3, 2, 1 et 0 sont calculées : 19 appels en tout, dont 12 qui relisent `memo`, au lieu de 49. Pour $n = 10^9$, `memo` ne contient que 242 clés, et il y a 724 appels, au lieu de plus de 10^{10} pour la version naïve.

Ici, les arguments sont quelques entiers dispersés entre 0 et n : on les range dans un

dictionnaire (une liste de longueur $n + 1$ serait impossible à créer pour $n = 10^9$). Quand les arguments sont au contraire **tous** les entiers de 0 à n , une **liste** de longueur $n + 1$ initialisée à None (« pas encore calculé ») suffit, et l'on teste `if memo[k] is None`. Quand il y a deux arguments entiers (i, j) , on utilise de même une **grille** (une liste de listes) `memo[i][j]`, ou un dictionnaire dont les clés sont des tuples (une liste ne peut pas être une clé) : `if (i, j) not in memo`.

Toute la difficulté est de choisir **ce qu'il faut mettre dans la clé** : tous les arguments dont dépend le résultat, et seulement eux. Le prochain TP, de programmation dynamique, prolongera ces idées.

8. La suite de Syracuse (Projet Euler 14)

Pour un entier $n \geq 1$, la **suite de Syracuse** issue de n est définie par $u_0 = n$ et

$$u_{k+1} = \begin{cases} u_k/2 & \text{si } u_k \text{ est pair,} \\ 3u_k + 1 & \text{sinon.} \end{cases}$$

Par exemple, la suite issue de 13 est 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 : elle compte 10 termes jusqu'au premier 1 inclus. La **conjecture de Syracuse** affirme que la suite atteint 1 quel que soit n . Elle a été vérifiée jusqu'à plus de 10^{20} , mais personne ne sait la démontrer.

Question 22. Écrire une fonction `plus_longue(N)` qui prend en argument un entier $N \geq 2$ et renvoie l'entier $n \in [1, N - 1]$ dont la suite de Syracuse est la plus longue (le plus petit en cas d'égalité). Elle doit répondre en quelques secondes pour $N = 10^6$: quel est alors le résultat ? On justifiera les choix faits (récursivité, mémoïsation, boucle éventuelle).

```
import time

def longueur_memo(n, memo):
    """Nombre de termes de la suite issue de n (memo contient {1: 1})."""
    if n not in memo:
        if n % 2 == 0:
            memo[n] = 1 + longueur_memo(n // 2, memo)
        else:
            memo[n] = 1 + longueur_memo(3 * n + 1, memo)
    return memo[n]

def plus_longue(N):
    memo = {1: 1}          # le cas de base est dans le dictionnaire
    meilleur = 1
    record = 1
    for n in range(2, N):
        l = longueur_memo(n, memo)
        if l > record:
            meilleur = n
            record = l
    return meilleur

debut = time.time()
print(plus_longue(10**6))
print(time.time() - debut, "secondes")
```

837799

1.2145118713378906 secondes

Réponse. On calcule la longueur de la suite issue de n par une fonction récursive : 1 si $n = 1$, et 1 de plus que la longueur de la suite issue du terme suivant sinon. Sa terminaison pour tout n **est** la conjecture de Syracuse : il n’y a pas de variant évident, puisque $3n + 1 > n$.

- **Mémoïsation** : la suite issue de 13 passe par 10, dont la longueur a déjà été calculée. On range les longueurs dans un dictionnaire (les termes rencontrés sont des entiers dispersés, jusqu’à environ $5,7 \times 10^{10}$ pour $N = 10^6$: une liste serait impossible). Sans mémoïsation, le calcul pour $N = 10^6$ prend une dizaine de secondes, avec mémoïsation, environ une seconde, et le dictionnaire contient plus de deux millions de clés.
- **Boucle** : on parcourt les entiers de 2 à $N - 1$ avec une boucle. Une récursion sur n aurait une profondeur de 10^6 , bien au-delà de la limite de récursion (ici 10^4).
- **Résultat** : 837 799, dont la suite compte 525 termes.

Le gain de la mémoïsation est moins spectaculaire que pour les shokobons : la fonction ne fait qu’un appel récursif, et la mémoïsation évite seulement de reparcourir les fins de suites déjà rencontrées.

```
# Tests : chaque ligne doit afficher True.
print(plus_longue(2) == 1, plus_longue(10) == 9, plus_longue(100) == 97)
print(plus_longue(20) == 18)          # 18 et 19 : 21 termes chacun
print(plus_longue(10**5) == 77031)
```

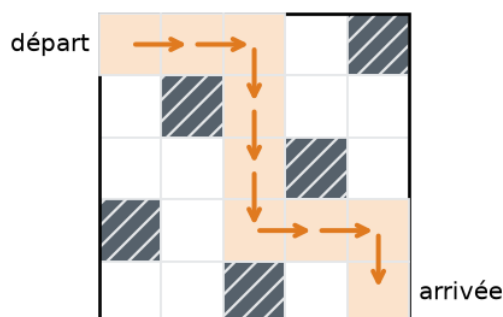
True True True

True

True

9. Chemins dans une grille à trous

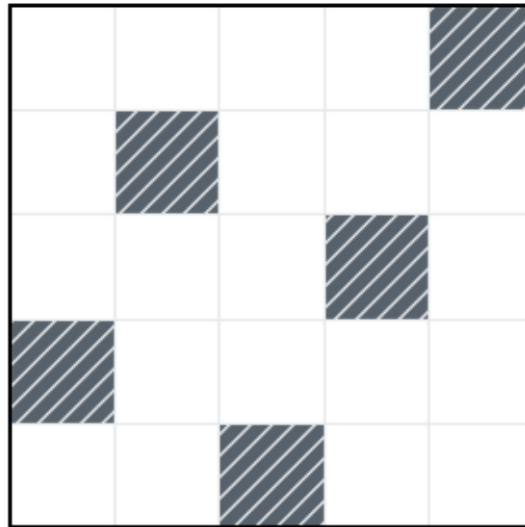
Une grille de n lignes et p colonnes est donnée par une liste de n listes de p booléens : $G[i][j]$ vaut **True** si la case (i, j) est un **trou**, et **False** si elle est libre. Un **chemin** part de la case $(0, 0)$ en haut à gauche, ne se déplace que d’une case vers la **droite** ou vers le **bas**, et ne passe par aucun trou. Voici un exemple de grille et de chemin.



```
F, T = False, True    # T : trou
P = [[F, F, F, F, T],
      [F, T, F, F, F],
      [F, F, F, T, F],
```

```
[T, F, F, F, F],
[F, F, T, F, F]]
dessiner_grille_trouee(P)
```

grille 5 x 5, 5 trou(s)



Question 23. Écrire une fonction récursive `nb_chemins_naif(G, i, j)` qui prend en arguments une grille `G` et les coordonnées (i, j) d'une case, et renvoie le nombre de chemins de $(0,0)$ à (i, j) dans `G`. Mesurer son temps de calcul sur une grille de 13×13 cases sans trou, et expliquer pourquoi il est si long.

```
def nb_chemins_naif(G, i, j):
    if G[i][j]:
        return 0
    elif i == 0 and j == 0:
        return 1
    else:
        total = 0
        if i > 0:
            # on arrive par le haut
            total = total + nb_chemins_naif(G, i - 1, j)
        if j > 0:
            # on arrive par la gauche
            total = total + nb_chemins_naif(G, i, j - 1)
        return total
```

Réponse. La fonction obtient son résultat en additionnant des 1 : elle fait au moins autant d'appels qu'il y a de chemins. Sans trou, il y a $\binom{24}{12} = 2\,704\,156$ chemins de $(0,0)$ à $(12,12)$ (choisir les 12 déplacements vers le bas parmi 24), soit plusieurs millions d'appels et environ une seconde. Les mêmes cases sont recalculées un très grand nombre de fois.

```
# Tests : chaque ligne doit afficher True.
F, T = False, True
P = [[F, F, F, F, T],
     [F, T, F, F, F],
```

```

    [F, F, F, T, F],
    [T, F, F, F, F],
    [F, F, T, F, F]]
print(nb_chemins_naif(P, 4, 4) == 8)
print(nb_chemins_naif(P, 0, 4) == 0)
print(nb_chemins_naif(P, 2, 2) == 2)
print(nb_chemins_naif([[F, F], [F, F]], 1, 1) == 2)
print(nb_chemins_naif([[F, T], [T, F]], 1, 1) == 0)

```

True
True
True
True
True

Question 24. Écrire une fonction `nb_chemins(G)` qui prend en argument une grille `G` et renvoie le nombre de chemins de $(0,0)$ jusqu'à la case en bas à droite, en mémorisant les résultats. Quelle est sa complexité ?

```

def nb_chemins_memo(G, i, j, memo):
    """memo[i][j] : nombre de chemins jusqu'à (i, j), ou None."""
    if memo[i][j] is None:
        if G[i][j]:
            # un trou
            memo[i][j] = 0
        elif i == 0 and j == 0:
            memo[i][j] = 1
        else:
            total = 0
            if i > 0:
                total = total + nb_chemins_memo(G, i - 1, j, memo)
            if j > 0:
                total = total + nb_chemins_memo(G, i, j - 1, memo)
            memo[i][j] = total
    return memo[i][j]

def nb_chemins(G):
    n, p = len(G), len(G[0])
    memo = [[None for j in range(p)] for i in range(n)]
    return nb_chemins_memo(G, n - 1, p - 1, memo)

```

Réponse. Les arguments qui varient sont les coordonnées (i, j) , toutes dans $\llbracket 0, n \rrbracket \times \llbracket 0, p \rrbracket$: on mémorise dans une **grille** `memo` de même taille que `G`, initialisée à `None`. Un dictionnaire de clés (i, j) conviendrait aussi, mais la grille est plus simple et plus rapide. Chaque case est calculée une fois, en $O(1)$ hors appels récurrents : $O(np)$ pour une grille $n \times p$. La grille `memo` ne dépend que de `G` : `nb_chemins` en crée une neuve à chaque appel, car des valeurs calculées pour une grille seraient fausses pour une autre.

```

# Tests : chaque ligne doit afficher True.
F, T = False, True      # T : trou
P = [[F, F, F, F, T],
      [F, T, F, F, F],
      [F, F, F, T, F],
      [T, F, F, F, F],
      [F, F, T, F, F]]

```

```

VIDE13 = [[False for j in range(13)] for i in range(13)]
VIDE30 = [[False for j in range(30)] for i in range(30)]
G30 = [[(7 * i + 3 * j) % 11 == 0 and 0 < i + j < 58 for j in range(30)]
        for i in range(30)]
print(nb_chemins(P) == 8, nb_chemins([[True]]) == 0)
print(nb_chemins(VIDE13) == 2704156)
print(nb_chemins(VIDE30) == 30067266499541040)
print(nb_chemins(G30) == 11856863193075)

```

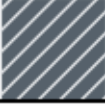
True True
True
True
True

```

# Visualisation : titre vert si le résultat est correct, rouge sinon.
F, T = False, True      # T : trou
P = [[F, F, F, F, T],
      [F, T, F, F, F],
      [F, F, F, T, F],
      [T, F, F, F, F],
      [F, F, T, F, F]]
def f(i, j):
    # chemins jusqu'à (i, j) = chemins dans la sous-grille des lignes 0..i
    # et des colonnes 0..j
    return nb_chemins([ligne[:j + 1] for ligne in P[:i + 1]])
dessiner_grille_trouee(P, f)

```

8 chemins jusqu'à l'arrivée

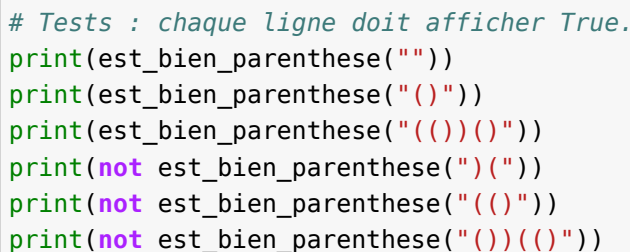
1	1	1	1	
1		1	2	2
1	1	2		2
	1	3	3	5
0	1		3	8

10. Parenthésages et nombres de Catalan

Un **mot de parenthèses** est une chaîne formée des caractères (et). Il est **bien parenthésé** s'il est vide, ou s'il s'écrit "(" + u + ")" + v avec u et v bien parenthésés. Par

Question 25. Écrire une fonction `est_bien_parenthese(s)` qui prend en argument un mot de parenthèses `s` et renvoie `True` s'il est bien parenthésé et `False` sinon, en $O(n)$ où n est la longueur de `s`.

Réponse. Principe : on lit le mot de gauche à droite en tenant à jour sa **hauteur**, le nombre de (lues moins le nombre de) lues. Un mot est bien parenthésé si et seulement si sa hauteur reste positive ou nulle et vaut 0 à la fin : c'est un chemin qui monte à chaque (, descend à chaque), ne passe jamais sous 0 et revient à 0. La décomposition "((" + u + ")" + v correspond au premier retour du chemin à la hauteur 0. La fonction auxiliaire reçoit l'indice i et la hauteur h après lecture des i premiers caractères : un seul appel par caractère, d'où $O(n)$.



32

ouvrantes et n fermantes.

```
def parentheses_aux(debut, o, f, liste):
    if o == 0 and f == 0:
        liste.append(debut)
    if o > 0:
        parentheses_aux(debut + "(", o - 1, f, liste)
    if f > 0: # on peut fermer si la hauteur f - o est strictement positive
        parentheses_aux(debut + ")", o, f - 1, liste)

def parentheses(n):
    liste = []
    parentheses_aux("", n, n, liste)
    return liste

# Aucune impasse : chaque appel produit au moins un mot.
# Chaque mot est construit en 2n appels : complexité  $O(n^2 C_n)$ 
# en comptant le coût des concaténations ( $C_n$  mots, cf. question suivante).
```

```
# Tests : chaque ligne doit afficher True.
print(parentheses(0) == [""], parentheses(1) == ["()"])
print(sorted(parentheses(3))
      == ["((()))", "(()())", "()(())", "()()()"])
print([len(parentheses(n)) for n in range(9)]
      == [1, 1, 2, 5, 14, 42, 132, 429, 1430])
```

True True

True

True

Question 27. On note C_n le nombre de mots bien parenthésés à n paires de parenthèses (le n -ième **nombre de Catalan**). Justifier que

$$C_0 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}, \quad C_{n+1} = \sum_{k=0}^n C_k C_{n-k}.$$

Écrire une fonction `catalan(n)` qui prend en argument un entier naturel n et renvoie C_n , avec $O(n^2)$ opérations arithmétiques.

Réponse. Un mot bien parenthésé non vide s'écrit de manière **unique** `"(" + u + ")" + v`, où `"(" + u + ")"` est le plus court préfixe non vide de hauteur nulle. Si le mot a $n + 1$ paires, u en a $k \in \llbracket 0, n \rrbracket$ et v en a $n - k$. Inversement, tout tel couple (u, v) donne un mot bien parenthésé à $n + 1$ paires. D'où la formule.

Calculée naïvement par récursion, la formule recalcule les mêmes C_k un nombre exponentiel de fois : on mémorise. Chaque C_m est alors calculé une seule fois, en $O(m)$ opérations, soit $O(n^2)$ au total.

```
def catalan_memo(n, memo):
    if memo[n] is None:
        s = 0
        for k in range(n):
            s = s + catalan_memo(k, memo) * catalan_memo(n - 1 - k, memo)
        memo[n] = s
    return memo[n]
```

```
def catalan(n):
    memo = [None for k in range(n + 1)]    # arguments de 0 à n : une liste
    memo[0] = 1
    return catalan_memo(n, memo)
```

```
# Tests : chaque ligne doit afficher True.
print([catalan(n) for n in range(10)]
      == [1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862])
print(catalan(30) == 3814986502092304)
print(catalan(50) == 1978261657756160653623774456)
```

True

True

True

Question 28. On utilise maintenant deux types de parenthèses, () et []. Un mot est bien parenthésé s'il est vide, ou s'il s'écrit "(" + u + ")" + v ou "[" + u + "]" + v avec u et v bien parenthésés. Ainsi "([])[]" est bien parenthésé, mais pas "([)]". Écrire une fonction `est_bien_parenthese2(s)` qui prend en argument un tel mot s et renvoie True s'il est bien parenthésé et False sinon, en $O(n)$.

```
def fermante(c):
    if c == "(":
        return ")"
    else:
        return "]"

def lire(s, i):
    if i == len(s) or s[i] == ")" or s[i] == "]":
        return i                # mot vide
    j = lire(s, i + 1)          # le mot u, après l'ouvrante s[i]
    if j == -1 or j == len(s) or s[j] != fermante(s[i]):
        return -1
    return lire(s, j + 1)       # le mot v, après la fermante s[j]

def est_bien_parenthese2(s):
    return lire(s, 0) == len(s)
```

```
# Chaque caractère est lu une fois : O(n). La pile d'appels joue le rôle
# de la pile des ouvrantes en attente de leur fermante.
```

```
# Tests : chaque ligne doit afficher True.
print(est_bien_parenthese2(""))
print(est_bien_parenthese2("() []"))
print(est_bien_parenthese2("([]) [() ()]"))
print(not est_bien_parenthese2("]"))
print(not est_bien_parenthese2("([)]"))
print(not est_bien_parenthese2("(("))
print(not est_bien_parenthese2("]"))
print(not est_bien_parenthese2("() ]"))
print(not est_bien_parenthese2("[ ( ]"))
```

True

True

True
True
True
True
True
True
True

Question 29. Combien y a-t-il de mots bien parenthésés à n paires avec deux types de parenthèses? Écrire une fonction `parenthesages2(n)` qui prend en argument un entier naturel n et renvoie la liste de tous ces mots, et vérifier.

Réponse. $2^n C_n$: en oubliant les types, on obtient un mot bien parenthésé à une seule sorte de parenthèses, et chacune des n paires peut ensuite prendre l'un des deux types, indépendamment des autres. (On peut aussi adapter la question sur les nombres de Catalan : $E_{n+1} = 2 \sum_{k=0}^n E_k E_{n-k}$.)

```
def parenthesages2_aux(debut, o, ouvertes, liste):
    if o == 0 and len(ouvertes) == 0:
        liste.append(debut)
    if o > 0:
        parenthesages2_aux(debut + "(", o - 1, ouvertes + "(", liste)
        parenthesages2_aux(debut + "[", o - 1, ouvertes + "[", liste)
    if len(ouvertes) > 0:
        # la dernière ouvrante ouverte est la première fermée
        k = len(ouvertes) - 1
        fin = debut + fermante(ouvertes[k])
        parenthesages2_aux(fin, o, ouvertes[:k], liste)

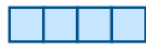
def parenthesages2(n):
    liste = []
    parenthesages2_aux("", n, "", liste)
    return liste
```

```
# Tests : chaque ligne doit afficher True.
print(sorted(parenthesages2(1)) == ["()", "[ ]"])
print([len(parenthesages2(n)) for n in range(8)]
      == [1, 2, 8, 40, 224, 1344, 8448, 54912])
print(all(est_bien_parenthese2(s) for s in parenthesages2(5)))
```

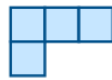
True
True
True

11. Partitions d'un entier (Projet Euler 76)

Une **partition** d'un entier $n \geq 0$ est une écriture de n comme somme d'entiers strictement positifs, sans tenir compte de l'ordre des termes (les **parts**). On la représente par la liste décroissante de ses parts. Par exemple, 4 a cinq partitions : [4], [3, 1], [2, 2], [2, 1, 1] et [1, 1, 1, 1]. L'entier 0 a une seule partition, la liste vide. On note $p(n)$ le nombre de partitions de n . On représente souvent une partition par un **tableau de Young** : une ligne de cases par part, de longueur la valeur de la part. Voici les cinq partitions de 4.



[4]



[3, 1]



[2, 2]



[2, 1, 1]



[1, 1, 1, 1]

Question 30. On note $p(n, m)$ le nombre de partitions de n dont toutes les parts sont inférieures ou égales à m . Justifier que, pour $1 \leq m \leq n$,

$$p(n, m) = p(n, m - 1) + p(n - m, m),$$

et préciser les cas de base. Écrire une fonction récursive `nb_partitions_naif(n)` qui prend en argument un entier naturel n et renvoie $p(n) = p(n, n)$, et mesurer son temps de calcul pour $n = 50$ et $n = 60$.

Réponse. Une partition de n en parts au plus égales à m ou bien ne contient pas la part m (c'est une partition en parts au plus égales à $m - 1$), ou bien la contient : en retirant une part m , on obtient une partition de $n - m$ en parts au plus égales à m , et réciproquement. Cas de base : $p(0, m) = 1$ (la partition vide), $p(n, 0) = 0$ pour $n > 0$, et $p(n, m) = p(n, n)$ si $m > n$.

```
def p_naif(n, m):
    """Nombre de partitions de n en parts au plus égales à m."""
    if n == 0:
        return 1
    elif m == 0:
        return 0
    elif m > n:
        return p_naif(n, n)
    else:
        return p_naif(n, m - 1) + p_naif(n - m, m)

def nb_partitions_naif(n):
    return p_naif(n, n)

# Le nombre d'appels est au moins p(n) : environ 0,1 s pour n = 50,
# 0,5 s pour n = 60, et il est multiplié par 5 tous les 10.
```

```
# Tests : chaque ligne doit afficher True.
print([nb_partitions_naif(n) for n in range(11)]
      == [1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42])
print(nb_partitions_naif(30) == 5604)
```

True

True

Question 31. Écrire une fonction `nb_partitions(n)` qui prend en argument un entier naturel n et renvoie $p(n)$, en mémorisant les valeurs $p(n', m')$. Quelle est sa complexité ? Le problème 76 du Projet Euler demande le nombre de façons d'écrire 100 comme somme d'au moins deux entiers strictement positifs : que vaut-il ?

```
def p_memo(n, m, memo):
    """memo[n][m] : nombre de partitions de n en parts <= m, ou None."""
    if memo[n][m] is None:
```

```

    if n == 0:
        memo[n][m] = 1
    elif m == 0:
        memo[n][m] = 0
    elif m > n:
        memo[n][m] = p_memo(n, n, memo)
    else:
        memo[n][m] = p_memo(n, m - 1, memo) + p_memo(n - m, m, memo)
    return memo[n][m]

def nb_partitions(n):
    # les arguments (n', m') vérifient n' <= n et m' <= n : une grille suffit
    memo = [[None for j in range(n + 1)] for i in range(n + 1)]
    return p_memo(n, n, memo)

# Au plus (n + 1)^2 couples (n', m'), chacun calculé une fois en O(1)
# hors appels récursifs : O(n^2).

# Tests : chaque ligne doit afficher True.
print([nb_partitions(n) for n in range(61)]
      == [nb_partitions_naif(n) for n in range(61)])
print(nb_partitions(200) == 3972999029388)

```

True

True

Réponse. $p(100) - 1 = 190\,569\,291$: on retire la partition $[100]$, qui n'a qu'une part.

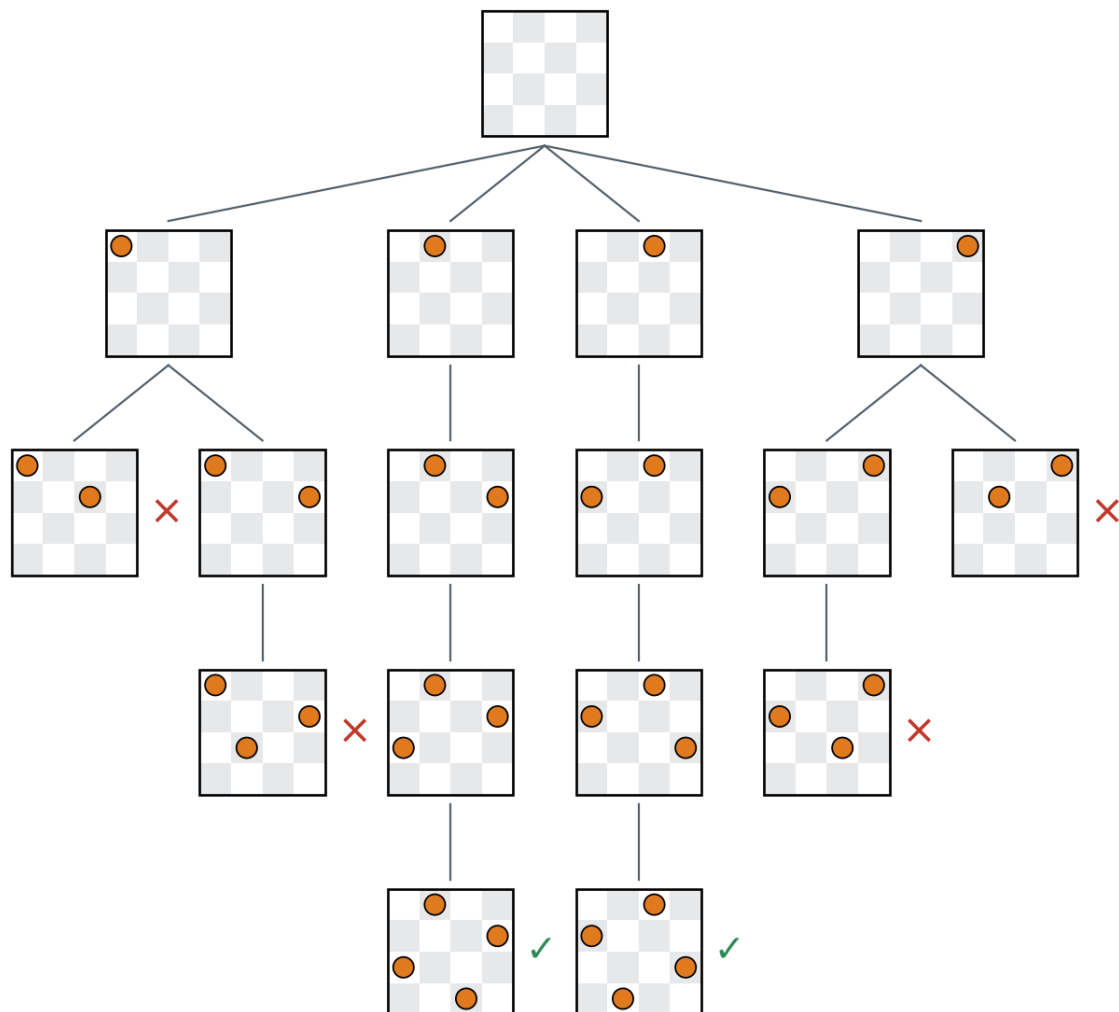
III. Backtracking

Rappel. On construit une solution par une suite de choix. Une fonction récursive reçoit une solution partielle :

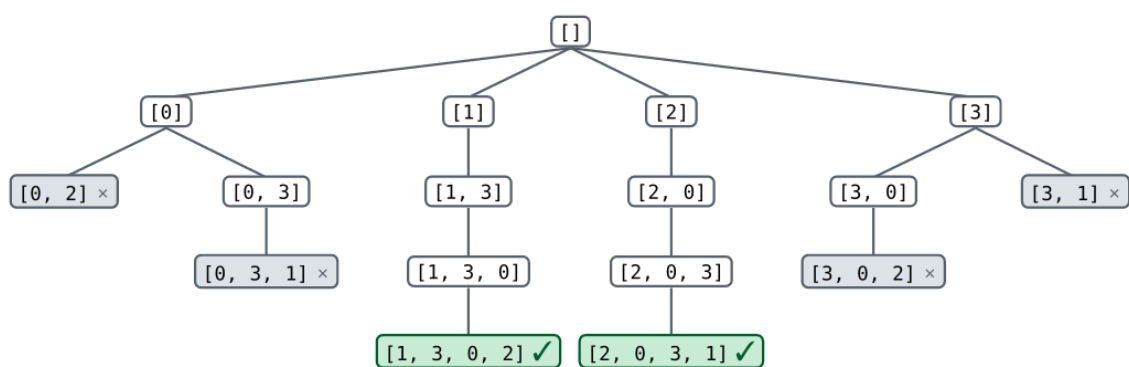
- si elle est complète, on l'enregistre (ou on s'arrête, si une seule solution suffit),
- sinon, pour chaque choix **compatible** avec elle, on fait ce choix, on appelle récursivement la fonction, puis on **annule** le choix avant d'essayer le suivant (c'est inutile si l'on a transmis une copie modifiée, comme `pos + [c]`).

Quand aucun choix ne convient, la fonction se termine sans rien trouver et l'on revient au choix précédent : c'est le **backtracking**. On parcourt ainsi en profondeur l'arbre des choix, sans jamais prolonger une solution partielle incompatible : on **élague** les branches qui ne mènent à aucune solution.

Exemple : placer n reines sur un échiquier $n \times n$ sans que deux d'entre elles soient sur une même ligne, une même colonne ou une même diagonale. Il y a exactement une reine par ligne : on place la reine de la ligne 0, puis celle de la ligne 1, et ainsi de suite, en n'essayant à chaque ligne que les colonnes où la nouvelle reine n'est en prise avec aucune des précédentes. Si aucune colonne ne convient, on revient à la ligne précédente pour y essayer la colonne suivante. Voici l'arbre des choix pour $n = 4$: chaque nœud est l'état de l'échiquier, une croix marquant une impasse.



Pour programmer cette recherche, il est inutile de stocker tout l'échiquier : puisqu'il y a une reine par ligne, un placement des reines sur les k premières lignes est entièrement décrit par la liste `pos` de longueur k , où `pos[i]` est la colonne de la reine de la ligne i . Le même arbre, avec cette représentation :

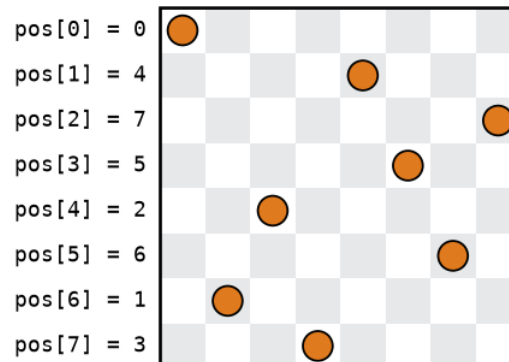


× : aucune colonne ne convient pour la ligne suivante, on revient en arrière ✓ : solution

Pour compter les appels d'une fonction récursive, on pourra utiliser une liste `cpt = [0]` définie en dehors de la fonction, et l'incrémenter au début de chaque appel par `cpt[0] = cpt[0] + 1`.

12. Le problème des n reines

La fonction `dessiner_echiquier(pos, n)` du module `dessins` dessine un placement sur un échiquier $n \times n$, et relie en rouge les reines en prise.



Question 32. Écrire une fonction `compatible(pos, c)` qui prend en arguments un placement `pos` de k reines et une colonne c , et renvoie `True` si l'on peut placer une reine en colonne c sur la ligne k sans qu'elle soit en prise avec les reines déjà placées, et `False` sinon, en $O(k)$.

```
def compatible_aux(pos, c, i):
    """La reine (k, c) est-elle compatible avec celles des lignes i à k - 1 ?"""
    k = len(pos)
    if i == k:
        return True
    elif pos[i] == c or pos[i] - c == k - i or c - pos[i] == k - i:
        return False
    else:
        return compatible_aux(pos, c, i + 1)

def compatible(pos, c):
    return compatible_aux(pos, c, 0)
```

```
# Tests : chaque ligne doit afficher True.
print(compatible([], 0), compatible([0], 2), compatible([1, 3], 0))
print(not compatible([0], 0))
print(not compatible([0], 1))
print(not compatible([1, 3], 2))
print(not compatible([1, 3, 0], 1), not compatible([2], 1))
print(compatible([0, 2], 4))
```

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

Question 33. Écrire une fonction `nb_solutions(n)` qui prend en argument un entier $n \geq 1$ et renvoie le nombre de façons de placer n reines sur un échiquier $n \times n$.

```
def nb_solutions_aux(pos, n):
    if len(pos) == n:
        return 1
    s = 0
    for c in range(n):
        # élagage : on ne prolonge que les placements valides
        if compatible(pos, c):
            s = s + nb_solutions_aux(pos + [c], n)
    return s

def nb_solutions(n):
    return nb_solutions_aux([], n)
```

```
# Tests : chaque ligne doit afficher True.
print([nb_solutions(n) for n in range(1, 11)]
      == [1, 0, 0, 2, 10, 4, 40, 92, 352, 724])
```

True

Question 34. Écrire une fonction `nb_solutions_appels(pos, n)` qui prend en arguments un placement `pos` et un entier `n`, et renvoie le couple (nombre de solutions qui prolongent `pos`, nombre d'appels), où le nombre d'appels compte **tous** les appels de la fonction, appel initial compris : c'est le nombre de nœuds de l'arbre des choix (17 pour $n = 4$, voir la figure du rappel). Pour $n = 8$, comparer ce nombre au nombre 8^8 de placements d'une reine par ligne, et au nombre $8!$ de placements d'une reine par ligne et par colonne.

```
def nb_solutions_appels(pos, n):
    """Renvoie le couple (nombre de solutions, nombre d'appels)."""
    if len(pos) == n:
        return (1, 1)
    s, a = 0, 1
    for c in range(n):
        if compatible(pos, c):
            x, y = nb_solutions_appels(pos + [c], n)
            s, a = s + x, a + y
    return (s, a)
```

```
# Tests : chaque ligne doit afficher True.
print(nb_solutions_appels([], 4) == (2, 17))
print(nb_solutions_appels([], 8) == (92, 2057))
```

True

True

Réponse. 2057 appels, contre $8! = 40\,320$ permutations et $8^8 = 16\,777\,216$ placements d'une reine par ligne. L'élagage abandonne un placement partiel dès qu'il est invalide, et coupe ainsi des sous-arbres entiers de l'arbre des placements. La complexité reste au moins exponentielle (on sait que le nombre de solutions croît plus vite que toute exponentielle), mais le gain est considérable : environ 0,7 s pour $n = 11$.

Question 35. Écrire une fonction `une_solution(n)` qui prend en argument un entier $n \geq 1$ et renvoie **une** solution (la liste `pos`), ou `None` s'il n'y en a pas. Elle doit s'arrêter **dès qu'une solution est trouvée**, sans explorer le reste de l'arbre. Dessiner une solution pour $n = 8$ et pour $n = 20$.

```
def une_solution_aux(pos, n):
    if len(pos) == n:
        return pos
    for c in range(n):
        if compatible(pos, c):
            s = une_solution_aux(pos + [c], n)
            if s is not None:
                return s
    return None

def une_solution(n):
    return une_solution_aux([], n)
```

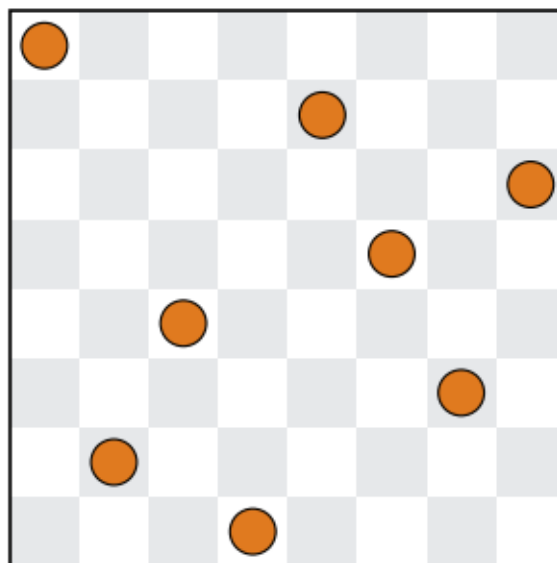
```
# Tests : chaque ligne doit afficher True.
def valide(s, n):
    return (len(s) == n and sorted(s) == list(range(n))
            and all(s[j] - s[i] != j - i and s[i] - s[j] != j - i
                    for i in range(n) for j in range(i + 1, n)))

print(une_solution(2) is None)
print(une_solution(3) is None)
print(une_solution(1) == [0])
print(all(valide(une_solution(n), n) for n in range(4, 16)))
```

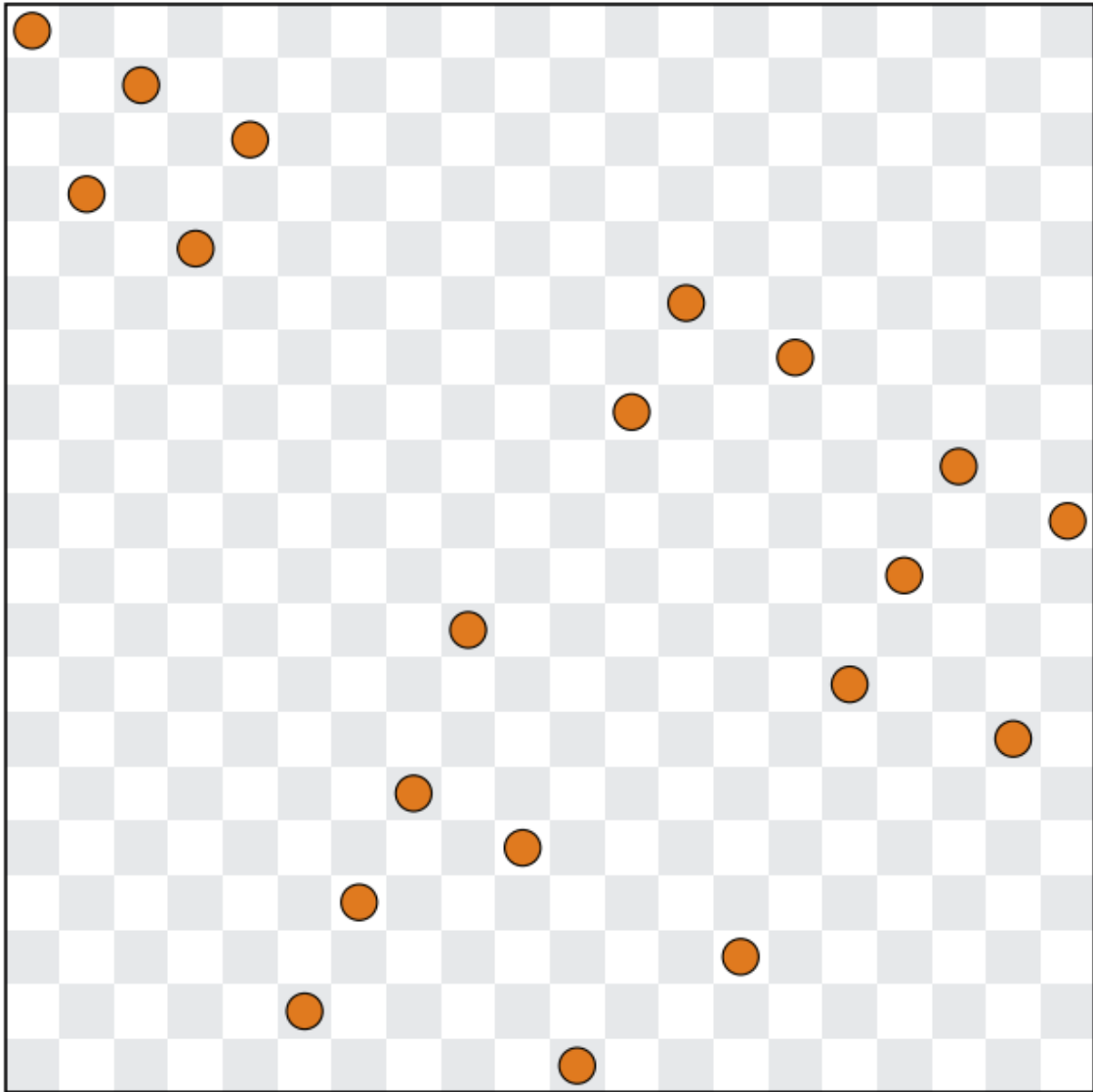
True
True
True
True

```
# Visualisation : titre vert si le résultat est correct, rouge sinon.
dessiner_echiquier(une_solution(8))
dessiner_echiquier(une_solution(20)) # quelques secondes
```

8 reines, aucune n'est en prise



20 reines, aucune n'est en prise



13. Sudoku

Une grille de Sudoku est représentée par une liste de 9 listes de 9 entiers, la valeur 0 indiquant une case vide. Il faut la compléter avec des chiffres de 1 à 9 de sorte que chaque ligne, chaque colonne et chacun des 9 carrés 3×3 contienne une seule fois chaque chiffre. La fonction `dessiner_sudoku(G, initiale)` du module `dessins` dessine la grille `G`, avec en gras les chiffres de la grille `initiale`, et marque en rouge les chiffres en conflit.

Dans cette partie, on s'autorise les boucles pour parcourir la grille : la récursivité sert au backtracking. Les fonctions de résolution **modifient** la grille qu'on leur passe.

grille à compléter

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

complétion fausse (conflits en rouge)

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

complétion correcte

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

```
FACILE = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
           [6, 0, 0, 1, 9, 5, 0, 0, 0],
           [0, 9, 8, 0, 0, 0, 0, 6, 0],
           [8, 0, 0, 0, 6, 0, 0, 0, 3],
           [4, 0, 0, 8, 0, 3, 0, 0, 1],
           [7, 0, 0, 0, 2, 0, 0, 0, 6],
           [0, 6, 0, 0, 0, 0, 2, 8, 0],
           [0, 0, 0, 4, 1, 9, 0, 0, 5],
           [0, 0, 0, 0, 8, 0, 0, 7, 9]]
```

```
DIFFICILE = [[8, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 3, 6, 0, 0, 0, 0, 0],
              [0, 7, 0, 0, 9, 0, 2, 0, 0],
              [0, 5, 0, 0, 0, 7, 0, 0, 0],
              [0, 0, 0, 0, 4, 5, 7, 0, 0],
              [0, 0, 0, 1, 0, 0, 0, 3, 0],
              [0, 0, 1, 0, 0, 0, 0, 6, 8],
              [0, 0, 8, 5, 0, 0, 0, 1, 0],
              [0, 9, 0, 0, 0, 0, 4, 0, 0]]
```

```
dessiner_sudoku(FACILE)
```

pas de conflit, 51 case(s) vide(s)

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

Question 36. Écrire une fonction `possible(G, i, j, v)` qui prend en arguments une grille `G`, les coordonnées (i, j) d'une case et un chiffre `v`, et renvoie `True` si l'on peut écrire `v` dans la case (i, j) , c'est-à-dire si `v` n'apparaît ni dans la ligne i , ni dans la colonne j , ni dans le carré 3×3 contenant la case, et `False` sinon.

```
def possible(G, i, j, v):
    for k in range(9):
        if G[i][k] == v or G[k][j] == v:
            return False
    a, b = 3 * (i // 3), 3 * (j // 3)      # coin du carré 3 x 3
    for x in range(a, a + 3):
        for y in range(b, b + 3):
            if G[x][y] == v:
                return False
    return True
```

```
# Tests : chaque ligne doit afficher True.
FACILE = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
           [6, 0, 0, 1, 9, 5, 0, 0, 0],
           [0, 9, 8, 0, 0, 0, 0, 6, 0],
           [8, 0, 0, 0, 6, 0, 0, 0, 3],
           [4, 0, 0, 8, 0, 3, 0, 0, 1],
           [7, 0, 0, 0, 2, 0, 0, 0, 6],
           [0, 6, 0, 0, 0, 0, 2, 8, 0],
           [0, 0, 0, 4, 1, 9, 0, 0, 5],
           [0, 0, 0, 0, 8, 0, 0, 7, 9]]
G = FACILE
print(possible(G, 0, 2, 1), possible(G, 0, 2, 4), possible(G, 4, 4, 5))
print(not possible(G, 0, 2, 5))
print(not possible(G, 0, 2, 8))
print(not possible(G, 0, 2, 9))
print(not possible(G, 0, 2, 7))    # 7 : seulement dans la ligne
print(not possible(G, 4, 4, 7))    # 7 : seulement dans la colonne
```

True True True
 True
 True
 True
 True
 True

Question 37. Écrire une fonction `resoudre(G)` qui prend en argument une grille `G`, la complète (en la modifiant) et renvoie `True` si c'est possible, et qui renvoie `False` sinon.

```
def resoudre_aux(G, c):
    if c == 81:
        return True
    i, j = c // 9, c % 9
    if G[i][j] != 0:                      # case donnée par l'énoncé
        return resoudre_aux(G, c + 1)
    for v in range(1, 10):
```

```

    if possible(G, i, j, v):
        G[i][j] = v
        if resoudre_aux(G, c + 1):
            return True
    G[i][j] = 0    # on efface avant de revenir en arrière : sinon la case
    return False   # garderait une valeur qui fausserait les tests possible

def resoudre(G):
    return resoudre_aux(G, 0)

```

```

# Tests : chaque ligne doit afficher True.
FACILE = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
          [6, 0, 0, 1, 9, 5, 0, 0, 0],
          [0, 9, 8, 0, 0, 0, 0, 6, 0],
          [8, 0, 0, 0, 6, 0, 0, 0, 3],
          [4, 0, 0, 8, 0, 3, 0, 0, 1],
          [7, 0, 0, 0, 2, 0, 0, 0, 6],
          [0, 6, 0, 0, 0, 0, 2, 8, 0],
          [0, 0, 0, 4, 1, 9, 0, 0, 5],
          [0, 0, 0, 0, 8, 0, 0, 7, 9]]
DIFFICILE = [[8, 0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 3, 6, 0, 0, 0, 0, 0],
             [0, 7, 0, 0, 9, 0, 2, 0, 0],
             [0, 5, 0, 0, 0, 7, 0, 0, 0],
             [0, 0, 0, 0, 4, 5, 7, 0, 0],
             [0, 0, 0, 1, 0, 0, 0, 3, 0],
             [0, 0, 1, 0, 0, 0, 0, 6, 8],
             [0, 0, 8, 5, 0, 0, 0, 1, 0],
             [0, 9, 0, 0, 0, 0, 4, 0, 0]]

def est_solution(G, initiale):
    """G est-elle complète, valide, et conforme aux chiffres de initiale ?"""
    for i in range(9):
        for j in range(9):
            if initiale[i][j] != 0 and G[i][j] != initiale[i][j]:
                return False
    lignes = [G[i] for i in range(9)]
    colonnes = [[G[i][j] for i in range(9)] for j in range(9)]
    carres = [[G[3 * a + x][3 * b + y] for x in range(3) for y in range(3)]
              for a in range(3) for b in range(3)]
    return all(sorted(z) == list(range(1, 10))
              for z in lignes + colonnes + carres)

for initiale in [FACILE, DIFFICILE]:
    G = [ligne.copy() for ligne in initiale]
    print(resoudre(G) and est_solution(G, initiale))

# la case (0, 8) ne peut recevoir aucun chiffre
G = [[1, 2, 3, 4, 5, 6, 7, 8, 0], [0, 0, 0, 0, 0, 0, 0, 0, 9]]
G = G + [[0 for j in range(9)] for i in range(7)]
print(not resoudre(G))

```

True

True

True

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

```
DIFFICILE = [[8, 0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 3, 6, 0, 0, 0, 0, 0],
             [0, 7, 0, 0, 9, 0, 2, 0, 0],
             [0, 5, 0, 0, 0, 7, 0, 0, 0],
             [0, 0, 0, 0, 4, 5, 7, 0, 0],
             [0, 0, 0, 1, 0, 0, 0, 3, 0],
             [0, 0, 1, 0, 0, 0, 0, 6, 8],
             [0, 0, 8, 5, 0, 0, 0, 1, 0],
             [0, 9, 0, 0, 0, 0, 4, 0, 0]]
G = [ligne.copy() for ligne in DIFFICILE]
resoudre(G)
dessiner_sudoku(G, DIFFICILE)
```

grille complète et valide

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

Question 38. Plutôt que de traiter les cases dans l'ordre, on peut choisir à chaque étape la case vide qui a le **moins de chiffres possibles**, et revenir en arrière dès qu'une case vide n'en a aucun. Écrire une fonction `resoudre_mieux(G)`, de même argument et de même résultat que `resoudre`, qui suit cette stratégie. Comparer, pour la grille `DIFFICILE`, le nombre d'appels récursifs **et** le temps de calcul des deux méthodes : le résultat peut surprendre, l'expliquer.

La grille `ANTI` ci-dessous a été conçue pour piéger la première méthode : ne pas lancer `resoudre` dessus (plusieurs minutes), mais seulement `resoudre_mieux`.

```
ANTI = [[0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 3, 0, 8, 5],
        [0, 0, 1, 0, 2, 0, 0, 0, 0],
        [0, 0, 0, 5, 0, 7, 0, 0, 0],
        [0, 0, 4, 0, 0, 0, 1, 0, 0],
        [0, 9, 0, 0, 0, 0, 0, 0, 0],
        [5, 0, 0, 0, 0, 0, 0, 7, 3],
```

```
[0, 0, 2, 0, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 4, 0, 0, 0, 9]]
```

```
def candidats(G, i, j):
    liste = []
    for v in range(1, 10):
        if possible(G, i, j, v):
            liste.append(v)
    return liste

def resoudre_mieux(G):
    # recherche de la case vide ayant le moins de candidats
    meilleure = None
    for i in range(9):
        for j in range(9):
            if G[i][j] == 0:
                c = candidats(G, i, j)
                if meilleure is None or len(c) < len(meilleure[2]):
                    meilleure = (i, j, c)
    if meilleure is None:
        # plus de case vide : grille complète
        return True
    i, j, c = meilleure
    for v in c:
        # si c est vide, retour en arrière immédiat
        G[i][j] = v
        if resoudre_mieux(G):
            return True
    G[i][j] = 0
    return False
```

Tests : chaque ligne doit afficher True.

```
FACILE = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
           [6, 0, 0, 1, 9, 5, 0, 0, 0],
           [0, 9, 8, 0, 0, 0, 0, 6, 0],
           [8, 0, 0, 0, 6, 0, 0, 0, 3],
           [4, 0, 0, 8, 0, 3, 0, 0, 1],
           [7, 0, 0, 0, 2, 0, 0, 0, 6],
           [0, 6, 0, 0, 0, 0, 2, 8, 0],
           [0, 0, 0, 4, 1, 9, 0, 0, 5],
           [0, 0, 0, 0, 8, 0, 0, 7, 9]]

DIFFICILE = [[8, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 3, 6, 0, 0, 0, 0, 0],
              [0, 7, 0, 0, 9, 0, 2, 0, 0],
              [0, 5, 0, 0, 0, 7, 0, 0, 0],
              [0, 0, 0, 0, 4, 5, 7, 0, 0],
              [0, 0, 0, 1, 0, 0, 0, 3, 0],
              [0, 0, 1, 0, 0, 0, 0, 6, 8],
              [0, 0, 8, 5, 0, 0, 0, 1, 0],
              [0, 9, 0, 0, 0, 0, 4, 0, 0]]

ANTI = [[0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 3, 0, 8, 5],
         [0, 0, 1, 0, 2, 0, 0, 0, 0],
         [0, 0, 0, 5, 0, 7, 0, 0, 0],
         [0, 0, 4, 0, 0, 0, 1, 0, 0],
```

```

[0, 9, 0, 0, 0, 0, 0, 0, 0],
[5, 0, 0, 0, 0, 0, 0, 7, 3],
[0, 0, 2, 0, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 4, 0, 0, 0, 9]]
def est_solution(G, initiale):
    """G est-elle complète, valide, et conforme aux chiffres de initiale ?"""
    for i in range(9):
        for j in range(9):
            if initiale[i][j] != 0 and G[i][j] != initiale[i][j]:
                return False
    lignes = [G[i] for i in range(9)]
    colonnes = [[G[i][j] for i in range(9)] for j in range(9)]
    carres = [[G[3 * a + x][3 * b + y] for x in range(3) for y in range(3)]
              for a in range(3) for b in range(3)]
    return all(sorted(z) == list(range(1, 10))
              for z in lignes + colonnes + carres)
for initiale in [FACILE, DIFFICILE, ANTI]:
    G = [ligne.copy() for ligne in initiale]
    print(resoudre_mieux(G) and est_solution(G, initiale))

```

True

True

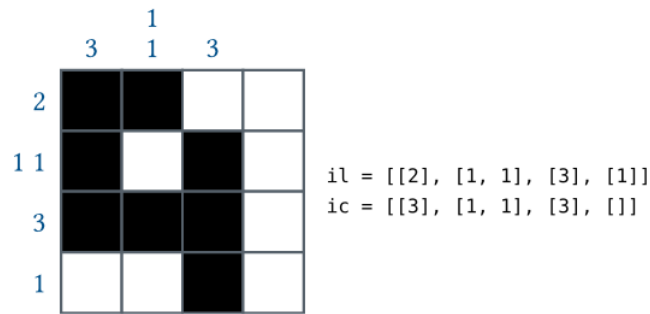
True

Réponse. Pour DIFFICILE, résoudre fait environ 72 000 appels et résoudre_mieux 10 000 : 7 fois moins. Mais chaque appel de résoudre_mieux recalcule les candidats de toutes les cases vides, et son temps total est **plus long** : environ 2 s, contre 0,2 s pour résoudre. Le choix de la case ne paie que si l'arbre naïf est vraiment grand : pour ANTI, résoudre fait 88 millions d'appels (4 min), contre 45 000 appels (8 s) pour résoudre_mieux. On gagnerait encore en tenant à jour les chiffres déjà utilisés dans chaque ligne, colonne et carré, comme pour les reines dans les compléments du TP.

14. Logimages (inspiré de X-ENS 2024)

Un **logimage** est une grille de n_l lignes et n_c colonnes dont il faut noircir certaines cases. Pour chaque ligne et chaque colonne, on donne la liste des longueurs des **blocs** de cases noires consécutives, dans l'ordre. Une grille est représentée par une liste de listes de 0 (case blanche) et de 1 (case noire), et les indications par deux listes de listes il (lignes) et ic (colonnes). La fonction `dessiner_logimage(G, il, ic)` du module `dessins` dessine une grille avec ses indications, en rouge celles qui ne sont pas respectées.

On s'autorise les boucles pour parcourir les lignes et les colonnes.



Question 39. Écrire une fonction `blocs(L)` qui prend en argument une liste `L` de 0 et de 1 et renvoie la liste des longueurs des blocs de 1 consécutifs de `L`. Par exemple, `blocs([1, 1, 0, 1, 0, 0, 1, 1, 1])` renvoie `[2, 1, 3]`.

```
def blocs(L):
    resultat = []
    courant = 0 # longueur du bloc en cours
    for x in L:
        if x == 1:
            courant = courant + 1
        elif courant > 0:
            resultat.append(courant)
            courant = 0
    if courant > 0:
        resultat.append(courant)
    return resultat
```

```
# Tests : chaque ligne doit afficher True.
print(blocs([1, 1, 0, 1, 0, 0, 1, 1, 1]) == [2, 1, 3])
print(blocs([]) == [])
print(blocs([0, 0]) == [])
print(blocs([1]) == [1])
print(blocs([0, 1, 1, 0]) == [2])
print(blocs([1, 0, 1]) == [1, 1])
```

True
True
True
True
True
True

Question 40. Écrire une fonction `verifie(G, il, ic)` qui prend en arguments une grille `G` et des indications `il` et `ic`, et renvoie `True` si `G` respecte les indications et `False` sinon.

```
def verifie(G, il, ic):
    for i in range(len(il)):
        if blocs(G[i]) != il[i]:
            return False
    for j in range(len(ic)):
        if blocs([G[i][j] for i in range(len(il))]) != ic[j]:
            return False
    return True
```

```
# Tests : chaque ligne doit afficher True.
G = [[1, 1, 0], [0, 1, 1], [1, 0, 1]]
print(verifie(G, [[2], [2], [1, 1]], [[1, 1], [2], [2]]))
print(not verifie(G, [[2], [2], [1, 1]], [[1, 1], [2], [1]]))
print(not verifie(G, [[2], [1], [1, 1]], [[1, 1], [2], [2]]))
```

True

True

True

Question 41. Écrire une fonction `solutions_naif(il, ic)` qui prend en arguments des indications `il` et `ic` et renvoie la liste de toutes les grilles qui les respectent, en énumérant **toutes** les grilles possibles case par case : la case numéro k est $(k // n_c, k \% n_c)$. Quelle est sa complexité? Peut-on l'utiliser pour une grille 10×10 ?

Indication : on pourra écrire une fonction récursive auxiliaire `solutions_naif_aux(k, grille, il, ic, liste)` qui essaie les deux valeurs possibles de la case numéro k dans grille, et ajoute à liste les solutions trouvées. Attention à ce qu'on ajoute à liste.

```
def solutions_naif_aux(k, grille, il, ic, liste):
    nl, nc = len(il), len(ic)
    if k == nl * nc:
        if verifie(grille, il, ic):
            # copie complète : grille va encore être modifiée
            liste.append([ligne.copy() for ligne in grille])
    else:
        i, j = k // nc, k % nc
        for x in range(2):
            grille[i][j] = x
            solutions_naif_aux(k + 1, grille, il, ic, liste)

def solutions_naif(il, ic):
    grille = [[0 for j in range(len(ic))] for i in range(len(il))]
    liste = []
    solutions_naif_aux(0, grille, il, ic, liste)
    return liste

# 2^(nl nc) grilles, chacune vérifiée en O(nl nc) : O(nl nc 2^(nl nc)).
# Pour 10 x 10, 2^100 ~ 10^30 grilles : hors de portée.
# Erreurs relevées par le jury d'X-ENS 2024 : ajouter grille sans copie
# (ou avec une copie superficielle grille.copy()), oublier le facteur nl nc.
```

```
# Tests : chaque ligne doit afficher True.
il, ic = [[2], [1, 1], [3], [1]], [[3], [1, 1], [3], []]
S = solutions_naif(il, ic) # ce logimage a deux solutions
print(sorted(S) == [[0, 1, 1, 0], [1, 0, 1, 0], [1, 1, 1, 0], [1, 0, 0, 0],
                    [1, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 0], [0, 0, 1, 0]])
print(len(solutions_naif([[1], [1]], [[1], [1]])) == 2)
```

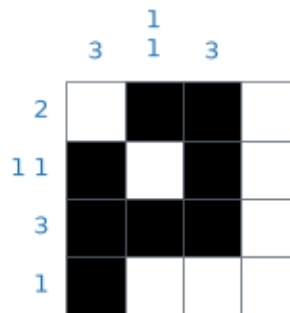
True

True

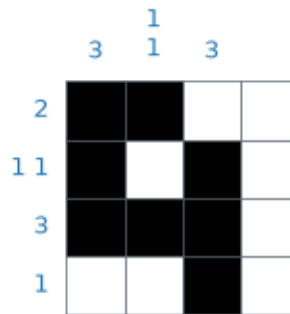
```
# Visualisation : titre vert si le résultat est correct, rouge sinon.
il, ic = [[2], [1, 1], [3], [1]], [[3], [1, 1], [3], []]
```

```
for G in solutions_naif(il, ic):
    dessiner_logimage(G, il, ic)
```

toutes les indications sont respectées



toutes les indications sont respectées



On procède maintenant **ligne par ligne** : on n'essaie pour chaque ligne que les listes compatibles avec son indication, et on revient en arrière dès que le début d'une colonne contredit l'indication de cette colonne.

Question 42. Écrire une fonction récursive `lignes_possibles(ind, nc)` qui prend en arguments une indication `ind` et un entier `nc`, et renvoie la liste de toutes les listes de longueur `nc` dont les blocs sont donnés par `ind`. Par exemple, `lignes_possibles([2, 1], 5)` renvoie (dans un ordre quelconque) `[1, 1, 0, 1, 0]`, `[1, 1, 0, 0, 1]` et `[0, 1, 1, 0, 1]`.

Indication : choisir le nombre de 0 placés avant le premier bloc, puis placer les blocs suivants récursivement. On pourra calculer d'abord la longueur minimale d'une liste contenant les blocs `ind`.

```
def place_min(ind):
    """Longueur minimale d'une liste dont les blocs sont ind."""
    if len(ind) == 0:
        return 0
    elif len(ind) == 1:
        return ind[0]
    else:
        return ind[0] + 1 + place_min(ind[1:])
```

```
def lignes_possibles(ind, nc):
    if len(ind) == 0:
        return [[0 for k in range(nc)]]
    resultat = []
    for s in range(nc - place_min(ind) + 1):
        debut = [0 for k in range(s)] + [1 for k in range(ind[0])]
        if len(ind) > 1:
            debut = debut + [0] # séparateur obligatoire
        for fin in lignes_possibles(ind[1:], nc - len(debut)):
            resultat.append(debut + fin)
    return resultat
```

```
# Tests : chaque ligne doit afficher True.
print(sorted(lignes_possibles([2, 1], 5))
      == [[0, 1, 1, 0, 1], [1, 1, 0, 0, 1], [1, 1, 0, 1, 0]])
print(lignes_possibles([], 3) == [[0, 0, 0]])
print(lignes_possibles([3], 3) == [[1, 1, 1]])
print(len(lignes_possibles([1], 6)) == 6)
print(len(lignes_possibles([1, 1, 1], 7)) == 10)
print(all(blocs(l) == [2, 1, 3] and len(l) == 12
          for l in lignes_possibles([2, 1, 3], 12)))
```

True
True
True
True
True
True

Question 43. Écrire une fonction `prefixe_compatible(p, ind)` qui prend en arguments une liste `p` de 0 et de 1 et une indication `ind`, et renvoie `True` si `p` peut être le **début** d'une liste dont les blocs sont donnés par `ind`, et `False` sinon. Autrement dit, les blocs **terminés** de `p` doivent être exactement les premiers blocs de `ind`, et un éventuel bloc **en cours** (si `p` finit par 1) ne doit pas dépasser le bloc correspondant de `ind`. Par exemple, `[1, 0, 1, 1]` est compatible avec `[1, 3]` mais pas avec `[1, 1]`, et `[1, 1, 0]` n'est pas compatible avec `[3]`.

```
def prefixe_compatible(p, ind):
    b = blocs(p)
    if len(b) > len(ind):
        return False
    if len(p) > 0 and p[len(p) - 1] == 1:
        # le dernier bloc n'est peut-être pas terminé
        k = len(b) - 1
        return b[:k] == ind[:k] and b[k] <= ind[k]
    else:
        return b == ind[:len(b)]
```

```
# Tests : chaque ligne doit afficher True.
print(prefixe_compatible([1, 0, 1, 1], [1, 3]))
print(not prefixe_compatible([1, 0, 1, 1], [1, 1]))
print(not prefixe_compatible([1, 1, 0], [3]))
```

```

print(prefixe_compatible([], [2]))
print(prefixe_compatible([0, 0], [2]))
print(prefixe_compatible([1, 1], [3, 1]))
print(not prefixe_compatible([1, 0, 1], [1]))

```

True
 True
 True
 True
 True
 True
 True

Question 44. Écrire une fonction `resoudre_logimage(il, ic)` qui prend en arguments des indications `il` et `ic` et renvoie la liste de toutes les solutions, par backtracking ligne par ligne. Résoudre le logimage 10×10 dont les indications `IL` et `IC` sont données dans la cellule de tests, et compter le nombre d'appels récurrents.

```

def resoudre_logimage_aux(grille, possibles, il, ic, liste):
    i = len(grille) # nombre de lignes déjà placées
    for j in range(len(ic)): # élagage sur les débuts de colonnes
        if not prefixe_compatible([grille[k][j] for k in range(i)], ic[j]):
            return
    if i == len(il):
        # les débuts de colonnes ne suffisent plus : un bloc peut manquer
        if verifie(grille, il, ic):
            # grille + [ligne] est une liste neuve, mais ses lignes sont
            # partagées avec possibles : il ne faut pas les modifier ensuite.
            liste.append(grille)
        return
    for ligne in possibles[i]:
        resoudre_logimage_aux(grille + [ligne], possibles, il, ic, liste)

def resoudre_logimage(il, ic):
    possibles = [lignes_possibles(ind, len(ic)) for ind in il]
    liste = []
    resoudre_logimage_aux([], possibles, il, ic, liste)
    return liste

```

Réponse. En comptant les appels (par exemple avec une liste `cpt` à un élément), on trouve 10 499 appels pour ce logimage 10×10 , contre $2^{100} \approx 10^{30}$ grilles pour la méthode naïve. Les logimages de la question suivante, plus difficiles, demandent quelques centaines de milliers d'appels.

```

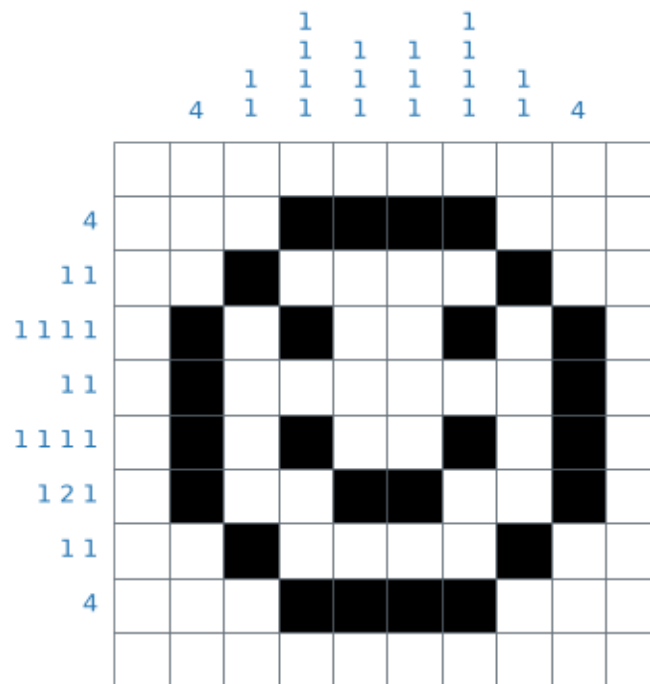
# Tests : chaque ligne doit afficher True.
IL = [[], [4], [1, 1], [1, 1, 1, 1], [1, 1],
       [1, 1, 1, 1], [1, 2, 1], [1, 1], [4], []]
IC = [[], [4], [1, 1], [1, 1, 1, 1], [1, 1, 1],
       [1, 1, 1], [1, 1, 1, 1], [1, 1], [4], []]
S = resoudre_logimage(IL, IC)
print(len(S) == 1, verifie(S[0], IL, IC))
il, ic = [[2], [1, 1], [3], [1]], [[3], [1, 1], [3], []]
print(sorted(resoudre_logimage(il, ic)) == sorted(solutions_naif(il, ic)))
print(resoudre_logimage([[1]], [[1], [1]]) == [])

```

True True
True
True

```
# Visualisation : titre vert si le résultat est correct, rouge sinon.
IL = [[], [4], [1, 1], [1, 1, 1, 1], [1, 1],
       [1, 1, 1, 1], [1, 2, 1], [1, 1], [4], []]
IC = [[], [4], [1, 1], [1, 1, 1, 1], [1, 1, 1],
       [1, 1, 1], [1, 1, 1, 1], [1, 1], [4], []]
for G in resoudre_logimage(IL, IC):
    dessiner_logimage(G, IL, IC)
```

toutes les indications sont respectées



Question 45 (bonus). Les indications de trois logimages mystères sont données ci-dessous. Les résoudre avec `resoudre_logimage`, vérifier que chacun a une seule solution, et dessiner les solutions avec `dessiner_logimage`. Quels animaux reconnaît-on ?

```
# Trois animaux mystères (19 x 18, 19 x 19 et 16 x 21)
IL_1 = [[1], [2], [2], [3, 1], [2, 2], [2, 3], [5, 3], [6, 5], [3, 4, 2, 2],
         [10, 2, 2], [12, 2, 1], [14, 2], [3, 10, 3], [2, 10, 2], [9, 1],
         [7, 2], [6, 1], [5, 2], [5, 1]]
IC_1 = [[2], [2, 4], [3, 5], [3, 4], [3, 6], [9], [2, 6], [10], [12], [14],
         [11], [2, 10], [3, 9], [4, 8], [1, 2, 6], [2, 3], [2, 8],
         [2, 2, 1, 1]]

IL_2 = [[1, 1, 1], [1, 1, 1], [9], [13], [3, 3, 3], [3, 1, 3],
         [2, 2, 1, 2, 2], [3, 2, 1, 2, 3], [3, 3, 3], [4, 5, 4], [7, 7],
         [6, 5, 6], [6, 5, 6], [2, 4, 4, 2], [3, 11, 3], [4, 4], [4, 4],
         [5, 5], [11]]
IC_2 = [[9], [12], [9, 4], [3, 5, 3], [2, 5, 3], [1, 2, 5, 2],
```

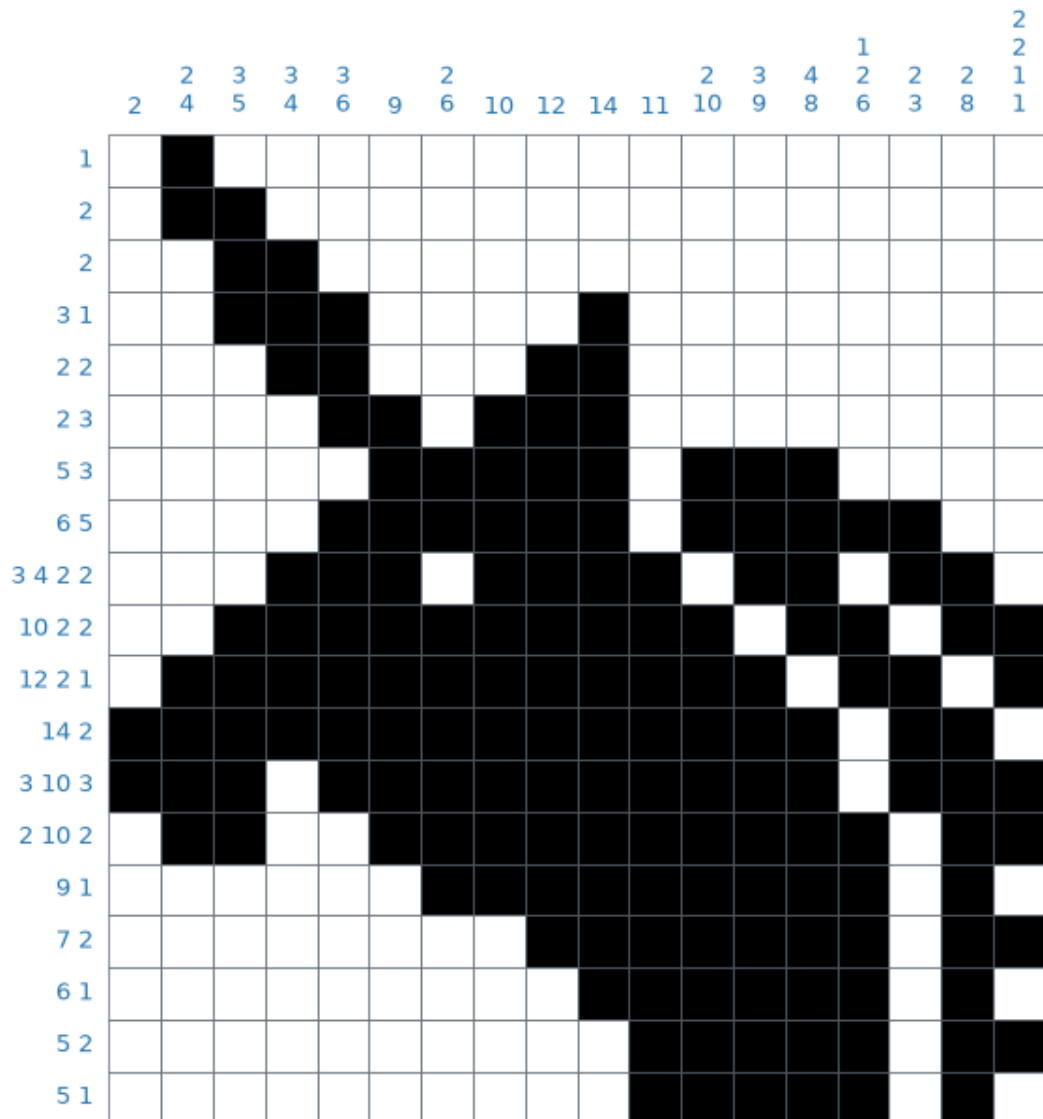
```
[3, 2, 1, 2, 2], [2, 2, 1, 2, 1, 1], [3, 2, 2, 1, 1], [10, 2, 1, 1],
[3, 2, 2, 1, 1], [2, 2, 1, 2, 1, 1], [3, 2, 1, 2, 2], [1, 2, 5, 2],
[2, 5, 3], [3, 5, 3], [9, 4], [12], [9]]
```

```
IL_3 = [[1, 1], [2, 2], [3, 3], [3, 3], [6], [1, 4, 2], [7, 4], [7, 5],
        [4, 7], [14], [13], [8], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1],
        [2, 2, 2, 2]]
IC_3 = [[1], [2], [4, 2], [7], [3, 3], [7, 1], [16], [6, 4], [4, 7], [3, 1],
        [3], [3, 1], [7], [4], [3, 4], [3, 1], [4], [5], [4], [4], [3]]
```

```
for il, ic in [(IL_1, IC_1), (IL_2, IC_2), (IL_3, IC_3)]:
    S = resoudre_logimage(il, ic)
    print(len(S), "solution(s)")
    for G in S:
        dessiner_logimage(G, il, ic)
```

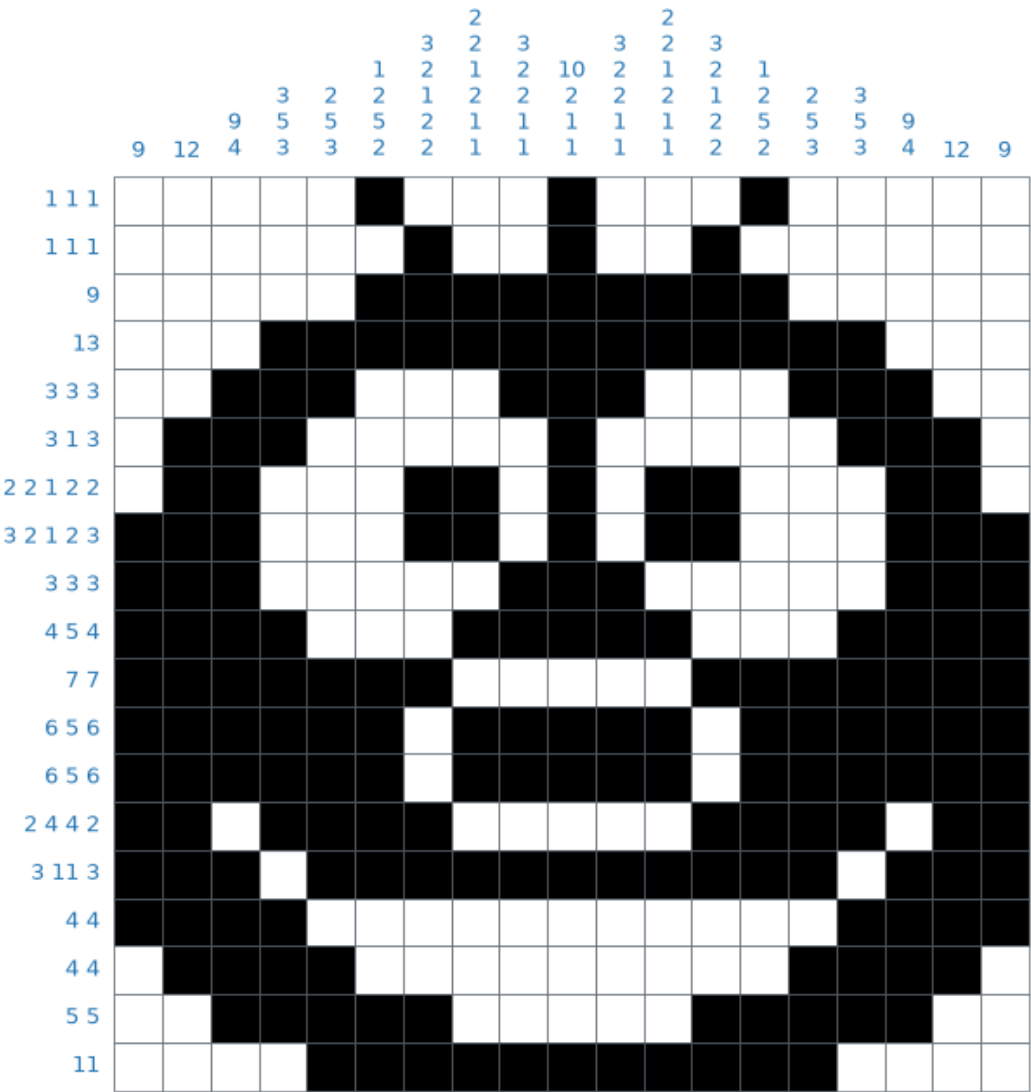
1 solution(s)

toutes les indications sont respectées



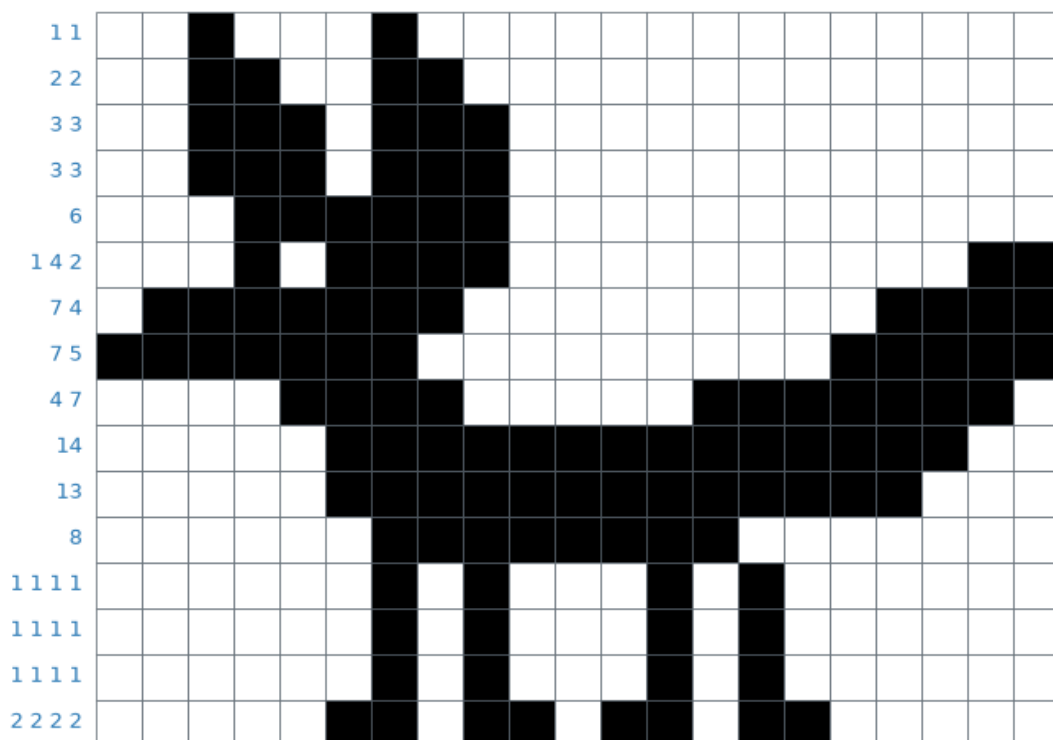
1 solution(s)

toutes les indications sont respectées



1 solution(s)

1 2 2 7 3 1 16 4 7 1 3 1 7 4 4 1 4 5 4 4 3



Réponse. Une licorne (vue de profil : corne, oreille, crinière), la Petite Taupe (vue de face : trois poils, grands yeux, gros nez, bouche ouverte) et le renard du Petit Prince (grandes oreilles, queue touffue). Chaque résolution prend quelques secondes, les indications ont été calculées et l'unicité vérifiée par le script `logimage_inverse.py`.