
TP — Révisions sur la récursivité

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])
```

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.

```
# 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)] == [fibo(n) for n in range(20)])
print(fibo_rapide(100) == 354224848179261915075)
```

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.

```
# 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)
```

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 ».

2. Recherche dichotomique

Question 5. Écrire une fonction récursive `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?

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

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)` ?

```
# 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)
```

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 ?

```
# 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]))
```

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
```

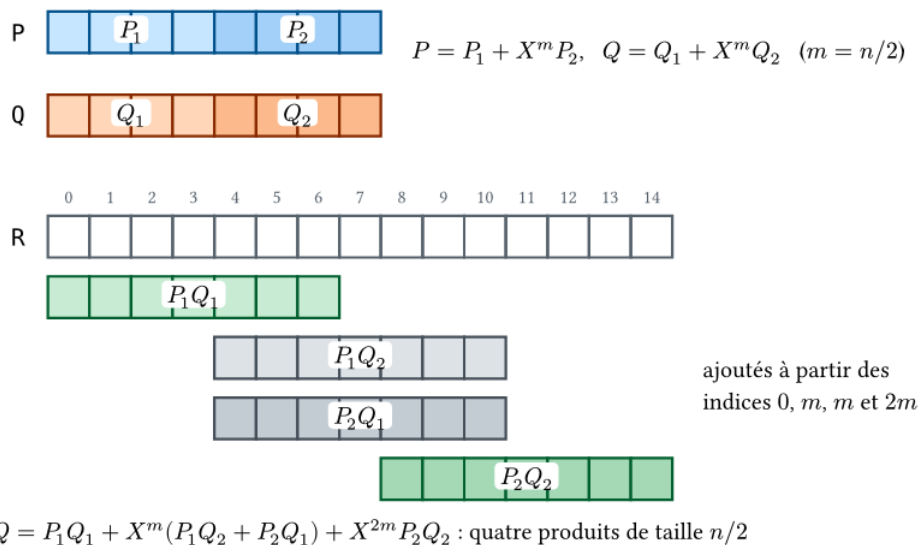
```

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])))

```

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 $2m = n$.



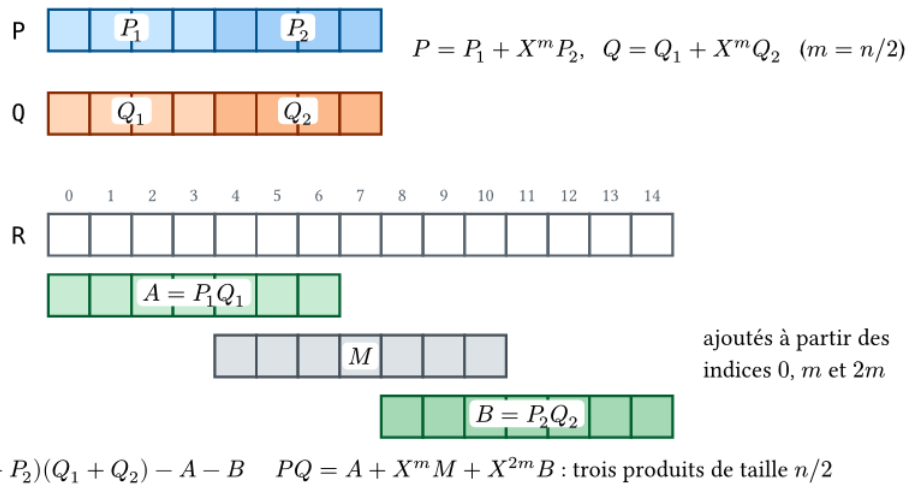
Question 8. Écrire une fonction récursive `produit_dpr(P, Q)` qui prend en arguments deux polynômes de même longueur n , puissance de 2, et renvoie leur produit (à $2n - 1$ coefficients), en effectuant **quatre** appels récursifs sur des polynômes de longueur $n/2$. Établir une relation de récurrence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

```

# 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]))

```

Karatsuba (1960). On pose $A = P_1 Q_1$ et $B = P_2 Q_2$. Comme $(P_1 + P_2)(Q_1 + Q_2) = A + P_1 Q_2 + P_2 Q_1 + B$, le terme du milieu s'obtient par $P_1 Q_2 + P_2 Q_1 = (P_1 + P_2)(Q_1 + Q_2) - A - B$: **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.

```
# 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]))
```

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)
```

4. Triangle de Pascal modulo 2

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

Question 11. Écrire une fonction récursive `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], ...]` : la ligne i est la liste $[\binom{i}{0}\%2, \dots, \binom{i}{i}\%2]$. Quelle est sa complexité ?

```
# 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, 0, 0, 1, 1, 1, 1])
```

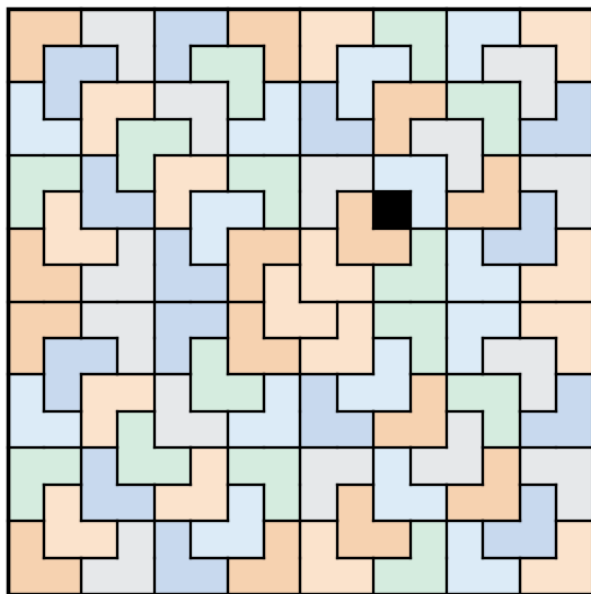
Question 12. Écrire une fonction `afficher_pascal(n)` qui prend en argument un entier $n \geq 1$ et **affiche** avec `print` les lignes 0 à $n - 1$ du triangle de Pascal modulo 2, en représentant chaque coefficient impair par **▲** et chaque coefficient pair par une espace, les caractères d'une ligne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

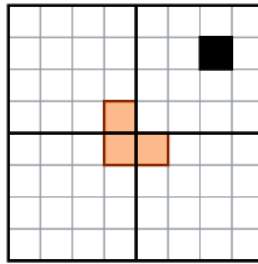
5. Pavage par des triominos (bonus)

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.



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.

```
# 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))
```

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

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 $O(n)$.

```
# 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) == ([], [], []))
```

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 $O(n)$?

```
# 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))
```

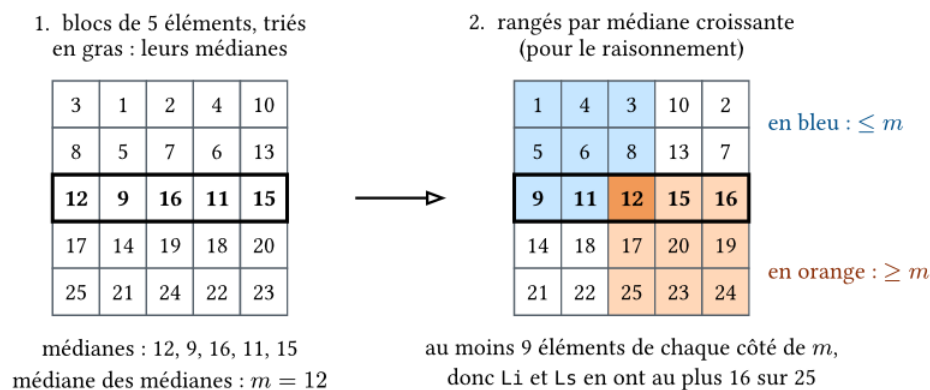
Question 17. Pour $0 \leq k < n$, le k -ième plus petit élément de `L` est l'élément d'indice k de la liste `L` triée. Pour $k = \lfloor n/2 \rfloor$, c'est la **médiane** de `L`. Écrire une fonction récursive `selection(L, k)` qui prend en arguments une liste `L` de longueur n et un entier $k \in [0, n[$, 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?

```
# 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]))
```

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 ?

```
# 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)
```

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

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

1. Montrer qu'au moins $\lceil B/2 \rceil (\lceil n/B \rceil / 2 - 1)$ 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 $(1 - \frac{\lceil B/2 \rceil}{2B})n + \lceil B/2 \rceil$ 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 $n > B$,

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

3. On admet qu'on peut négliger les parties entières et les constantes additives, c'est-à-dire remplacer cette inégalité par $T(n) \leq T(\frac{n}{B}) + T(\beta_B n) + \gamma_B n$, 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.

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

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

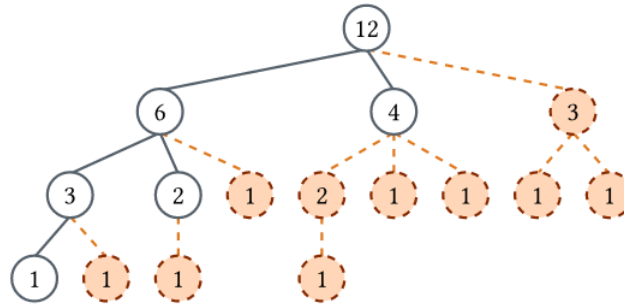
Question 20. Écrire une fonction récursive `shokobons_naif(n)` qui prend en argument un entier naturel n et renvoie $f(n)$.

```
# 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)
```

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:
            exchange = (shokobons_memo(n // 2, memo)
                        + shokobons_memo(n // 3, memo)
                        + shokobons_memo(n // 4, memo))
            if exchange > n:
                memo[n] = exchange
            else:
                memo[n] = n
    return memo[n]

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

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

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` ?

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

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

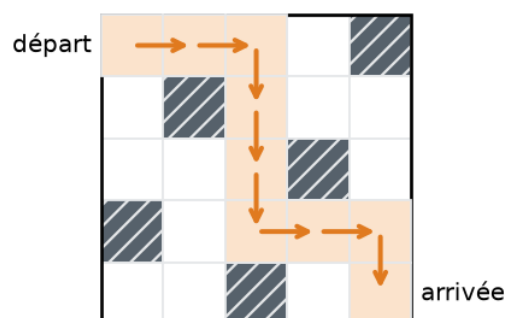
Par exemple, la suite issue de 13 est 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 : elle compte 10 termes jusqu'au premier 1 inclus. La **conjecture de Syracuse** affirme que la suite atteint 1 quel que soit n . Elle a été vérifiée jusqu'à plus de 10^{20} , mais personne ne sait la démontrer.

Question 22. Écrire une fonction `plus_longue(N)` qui prend en argument un entier $N \geq 2$ et renvoie l'entier $n \in \llbracket 1, N-1 \rrbracket$ 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).

```
# 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)
```

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)
```

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.

```
# 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]]
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)
```

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 ?

```
# 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([T]) == 0)
print(nb_chemins(VIDE13) == 2704156)
print(nb_chemins(VIDE30) == 30067266499541040)
print(nb_chemins(G30) == 11856863193075)
```

```
# 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)
```

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 exemple, "(()())" est bien parenthésé, mais ni ")(", ni "(()".

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`.

```
# Tests : chaque ligne doit afficher True.
print(est_bien_parenthese(""))
print(est_bien_parenthese("("))
print(est_bien_parenthese("(()())"))
print(not est_bien_parenthese(")()"))
print(not est_bien_parenthese("(())"))
print(not est_bien_parenthese("()()()"))
```

Question 26. Écrire une fonction `parenthesages(n)` qui prend en argument un entier naturel `n` et renvoie la liste de tous les mots bien parenthésés contenant `n` parenthèses ouvrantes et `n` fermantes.

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

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.

```
# 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)
```

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 $O(n)$.

```
# 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("[()])"))
```

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.

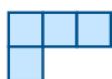
```
# Tests : chaque ligne doit afficher True.
print(sorted(parenthesages2(1)) == ["()", "[]"])
print([len(parenthesages2(n)) for n in range(8)]
      == [1, 2, 8, 40, 224, 1344, 8448, 54912])
print(all(est_bien_parenthese2(s) for s in parenthesages2(5)))
```

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 $n = 60$.

```
# 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)
```

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 ?

```
# 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)
```

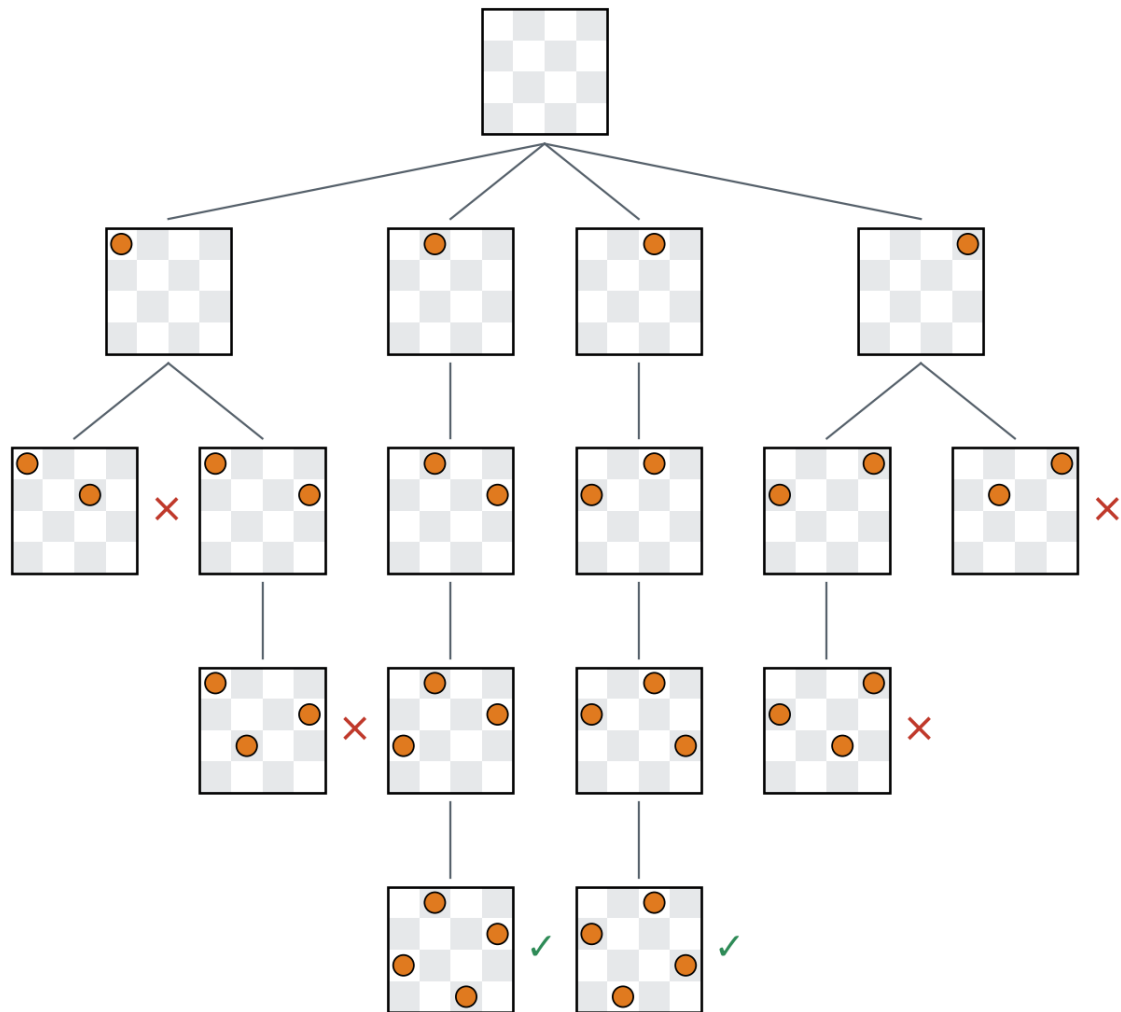
III. Backtracking

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

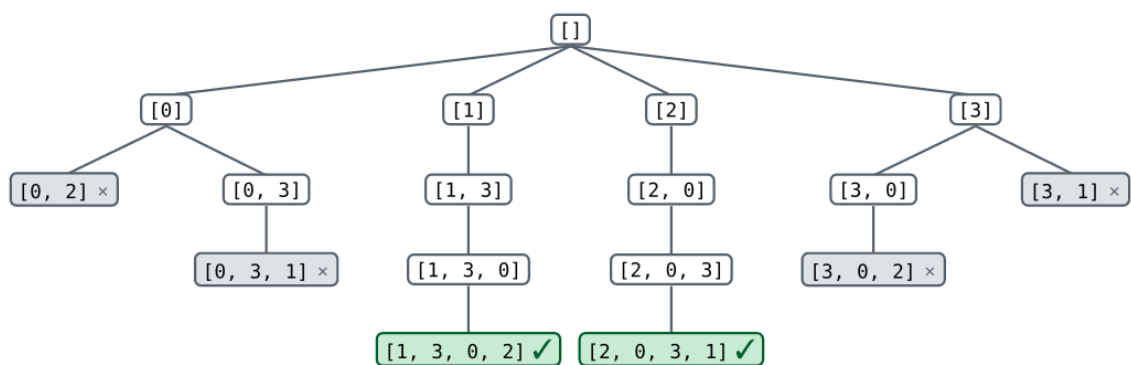
- si elle est complète, on l'enregistre (ou on s'arrête, si une seule solution suffit),
- sinon, pour chaque choix **compatible** avec elle, on fait ce choix, on appelle récursivement la fonction, puis on **annule** le choix avant d'essayer le suivant (c'est inutile si l'on a transmis une copie modifiée,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

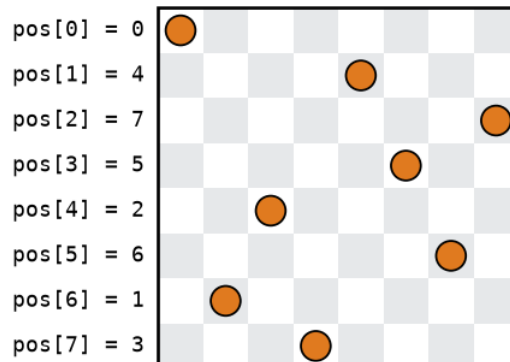


× : aucune colonne ne convient pour la ligne suivante, on revient en arrière ✓ : solution

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 $O(k)$.

```
# 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))
```

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 $n \times n$.

```
# 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])
```

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.

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

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 $n = 20$.

```
# 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)))
```

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

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									complétion fausse (conflits en rouge)									complétion correcte								
5	3			7					5	3	8	6	7	4	9	1	2	5	3	4	6	7	8	9	1	2
6				1	9	5			6	7	2	1	9	5	3	4	8	6	7	2	1	9	5	3	4	8
	9	8						6	1	9	8	3	4	2	5	6	7	1	9	8	3	4	2	5	6	7
8				6				3	8	5	9	7	6	1	4	2	3	8	5	9	7	6	1	4	2	3
4				8		3		1	4	2	6	8	5	3	7	9	1	4	2	6	8	5	3	7	9	1
7				2				6	7	1	3	9	2	4	8	5	6	7	1	3	9	2	4	8	5	6
	6						2	8	9	6	1	5	3	7	2	8	4	9	6	1	5	3	7	2	8	4
				4	1	9		5	2	8	7	4	1	9	6	3	5	3	4	5	2	8	6	1	7	9
				8				7	3	4	5	2	8	6	1	7	9	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)
```

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.

```
# 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
```

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.

```
# 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))

```

```

# 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)

```

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 récurifs **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]]

```

```

# 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)

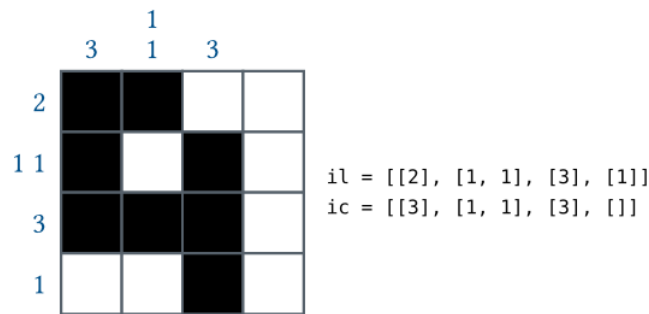
for initiale in [FACILE, DIFFICILE, ANTI]:
    G = [ligne.copy() for ligne in initiale]
    print(resoudre_mieux(G) and est_solution(G, initiale))

```

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]`.

```
# 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])
```

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.

```
# 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]]))
```

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.

```
# 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)
```

```
# 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)
```

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`.

```
# 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)))
```

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]`.

```
# 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]))
```

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 récursifs.

```
# 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]]) == [])
```

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)
```

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]]
```