Exercice 1. Affichage dans la console

```
# Figure 1
*****
****
***
**
*
```

```
# Figure 2
*
**
***
****
*****
```

```
# Figure 3
*****
****
***
**
*
**
***
****
*****
```

```
# Figure 4
\         /
 \       /
  \     /
   \   /
    \ /
     V
```

```
# Figure 5
\         /
 \       /
  \     /
   \   /
    \ /
     X
    / \
   /   \
  /     \
 /       \
/         \
```

```
# Figure 6
OOOOOOOO
O O O O
OO  OO
O   O
OOOO
O O
OO
O
```

Figures 1, 2 et 3.

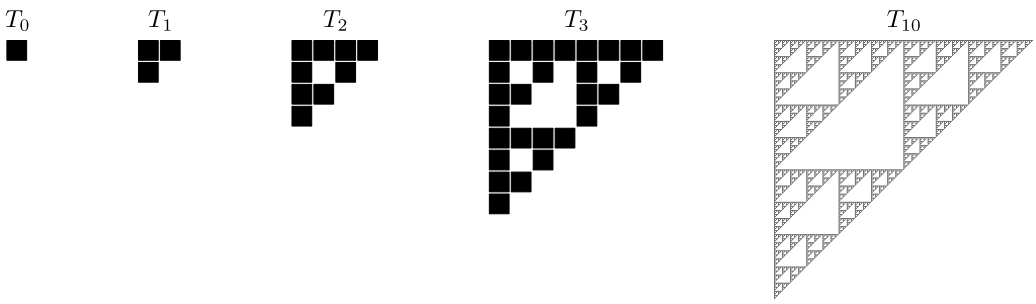
1. Étant donné un entier $n \in \mathbb{N}$, on souhaite afficher dans la console un triangle composé de n lignes. Par exemple, lorsque $n = 5$, on obtient la figure 1.
   (a) Expliquer comment afficher le triangle dans le cas où `n = 0`.
   (b) Supposons `n > 0` et avoir à notre disposition une procédure pour afficher un triangle avec `(n-1)` lignes. Expliquer comment afficher un triangle avec `n` lignes.
   (c) En déduire une fonction récursive de signature « `fig1(n: int) -> Nonetype` » qui affiche un triangle comme dans la figure 1. Vos tests ne doivent pas afficher `None`.
2. Même question pour les figures 2 et 3 (les exemples ci-dessus correspondent à `n = 5`). Pour la figure 3, on écrira une nouvelle fonction sans recourir aux fonction précédentes. Vos tests ne doivent pas afficher `None`.

Figures 4 et 5. Il s'agit maintenant d'écrire une fonction de signature « `affiche_V(n: int) -> Nonetype` » qui affiche un V de taille `n`. Par exemple, le cas `n = 5` donne la figure 4.

3. (a) Écrire la version itérative de cette fonction : « `affiche_V_it(n: int) -> NoneType` ». On vérifiera que le nombre de lignes est correct. **Remarque.** Pour créer une chaîne de caractères contenant un anti-slash, il faut écrire `"\\"` et non `"\"`. De plus, pour afficher deux chaînes de caractères `s1` et `s2`, on utilisera `print(s1 + s2)` et non `print(s1, s2)` car cette deuxième commande ajoute un espace entre `s1` et `s2`.
   (b) En utilisant la remarque 7 du cours, transformer la fonction `affiche_V_it` en une fonction récursive.
4. Sans utiliser la fonction précédente, écrire une fonction récursive qui affiche un X comme dans la figure 5. Vos tests ne doivent pas afficher `None`.

Triangle de Sierpinski. Le triangle de Sierpinski numéro $n \in \mathbb{N}$ noté T_n se définit récursivement :

- T_0 est un unique carré.
- Étant donné T_n, le triangle de Sierpinski T_{n+1} se décompose en quatre blocs :
  - Le bloc en haut à gauche, le bloc en haut à droite et le bloc en bas à gauche contiennent chacun une copie de T_n.
  - Le bloc en bas à droite est vide.



Remarquons tout d'abord que T_0 est composé d'une seule ligne et que T_{n+1} est composé de deux fois plus de lignes que T_n. Par une récurrence immédiate sur n, le nombre de lignes dans T_n est donc 2^n.

Notre objectif est d'afficher T_n dans la console en remplaçant les cases noires par des O et les cases blanches par des espaces. Par exemple, pour T_3, on obtient la figure 6. L'idée est d'afficher la figure ligne par ligne. On souhaite donc écrire une fonction « `affiche_ligne(n: int, i: int) -> Nonetype` » qui affiche la $\texttt{i}^{\text{ème}}$ ligne de T_n sous l'hypothèse $\texttt{i} \in [\![0, 2^{\texttt{n}} - 1]\!]$.

5. (a) Pour `n = 0`, que doit faire un appel à `affiche_ligne(n, i)` ?
   (b) Soit $n \in \mathbb{N}^*$ et $i \in [\![0, 2^n - 1]\!]$. Supposons savoir afficher n'importe quelle ligne de T_{n-1}. Expliquer comment afficher la ligne i de T_n.
   (c) Écrire la fonction « `affiche_ligne(n: int, i: int) -> Nonetype` » décrite ci-dessus. On pourra utiliser la commande `print("O", end="")` qui affiche un O sans retour à la ligne.
   (d) En déduire une fonction récursive qui prend en entrée un entier n et affiche T_n. Vos tests ne doivent pas afficher `None`.

Exercice 2. Récursivité croisée

La récursivité croisée consiste à écrire deux fonctions `f1` et `f2` qui s'appellent mutuellement. Par exemple :

```
def f1(n):
    """f1(n: int) -> bool"""
    if n == 0:
        return True
    elif n < 0:
        return f2(n+1)
    else:
        return f2(n-1)
```

```
def f2(n):
    """f2(n: int) -> bool"""
    if n == 0:
        return False
    elif n < 0:
        return f1(n+1)
    else:
        return f1(n-1)
```

Déterminer ce que renvoient `f1` et `f2`.

Exercice 3. Manipulations de listes

1. Écrire une fonction récursive « `suppr_doublons(L: list[int]) -> NoneType` » qui prend en argument une liste triée `L` et supprime les éléments en double dans `L`. On pourra utiliser la commande « `del L[k]` » qui supprime l'élément d'indice `k` de `L`. Notez que la liste est triée et que votre fonction modifie la liste en entrée et ne doit rien renvoyer.
2. Écrire une fonction récursive `melange` qui prend en entrée deux listes $\texttt{L}_1$ et $\texttt{L}_2$ et renvoie :

   $$\texttt{L} = [\texttt{L}_1[0], \texttt{L}_2[0], \texttt{L}_1[1], \texttt{L}_2[1], \texttt{L}_1[2], \texttt{L}_2[2], \ldots].$$

   Dans le cas où $\texttt{L}_1$ et $\texttt{L}_2$ ne sont pas de la même longueur, votre programme ajoutera les éléments en surplus à la fin de `L`. Par exemple, `melange([1, 2, 3], [4, 5, 6, 7, 8, 9])` vaut `[1, 4, 2, 5, 3, 6, 7, 8, 9]`.
3. Écrire une fonction récursive « `renverser(L: list[int]) -> list[int]` » qui inverse l'ordre des éléments d'une liste. Par exemple `renverser([1, 2, 3, 4])` vaut `[4, 3, 2, 1]`.

Exercice 4. Génération exhaustive

Dans cet exercice, on dira qu'une liste `L` vérifie la propriété $\mathcal{P}_{\texttt{n}}$ pour un certain $\texttt{n} \in \mathbb{N}$, si :

- ⋆ `len(L) = n`.
- ⋆ Chaque entier de $[\![1\,;\texttt{n}]\!]$ apparaît une et une seule fois dans `L`.
- ⋆ Il n'existe pas d'indice $\texttt{i} \geqslant 0$ tel que $\texttt{L[i]} \leqslant \texttt{L[i + 1]} \leqslant \texttt{L[i + 2]}$.

Par exemple, `[3; 4; 1; 5; 2]` vérifie $\mathcal{P}_5$, mais `[3; 1; 4; 5; 2]` ne vérifie pas $\mathcal{P}_5$ car $1 \leqslant 4 \leqslant 5$.

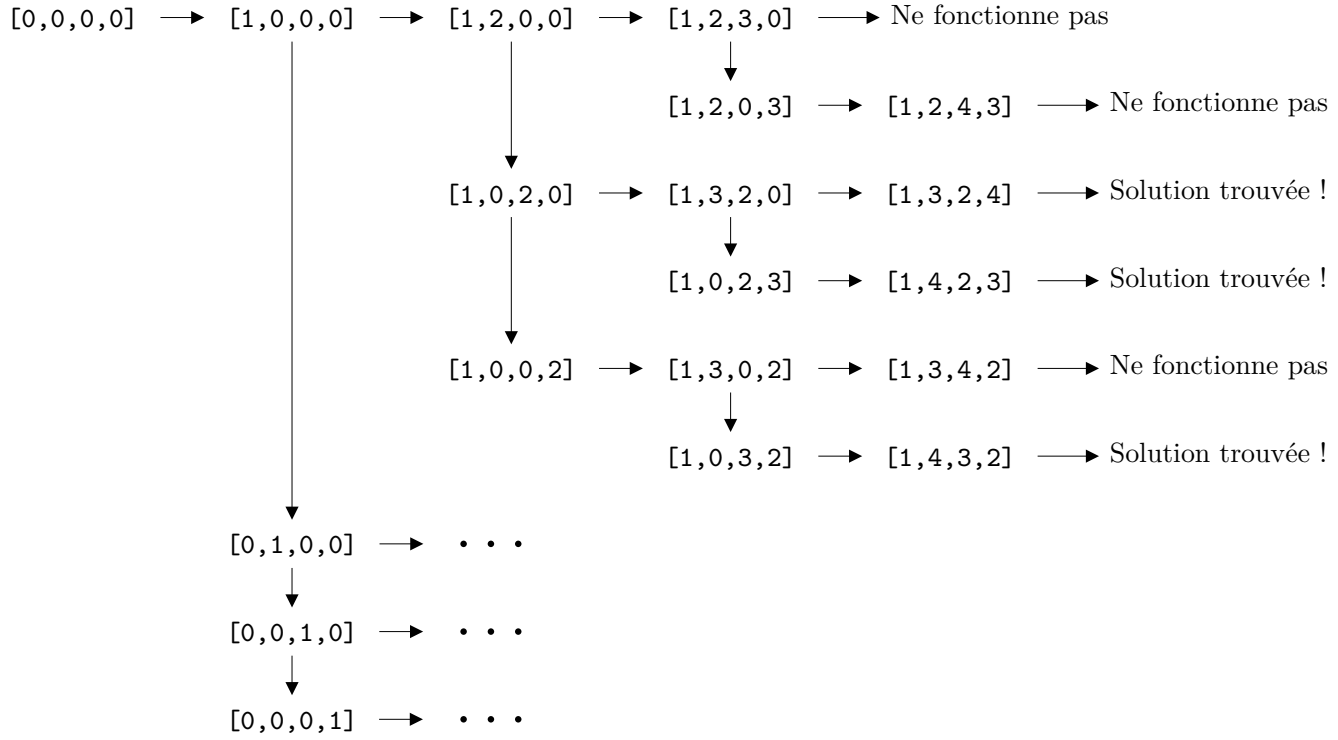
1. Écrire une fonction de signature « `compter_listes_Pn(n: int) -> int` » qui renvoie le nombre de listes vérifiant $\mathcal{P}_{\texttt{n}}$. Pour cela, vous devez utiliser une fonction récursive qui génère toutes les listes vérifiant $\mathcal{P}_{\texttt{n}}$. Si besoin, on pourra lire les indications ci-dessous.

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On part d'une liste `L` de taille `n` ne contenant que des 0, que l'on va compléter petit à petit. Pour cela, on écrit une fonction auxiliaire `aux` prenant en entrée un entier $k \in [\![1; n]\!]$, un indice $i \in [\![0, n-1]\!]$, ainsi que la liste `L` en construction :

- ⋆ `aux` suppose que les entiers $1, 2, \dots k-1$ ont déjà été placés dans `L` et essaye d'y placer k.
- ⋆ `aux` suppose qu'on a déjà essayé de placer k en position $0, 1, \dots, i-1$ et essaye de placer k en position i.
- ⋆ Si k peut être placé en position i sans contredire la propriété $\mathcal{P}_{\texttt{n}}$, alors :
  - La fonction place k en position i, puis complète `L` grâce à un appel récursif. Cet appel récursif renvoie r_1 le nombre de solutions dans lesquelles k est placé en position i dans `L`.
  - Grâce à un autre appel récursif, la fonction complète `L` en plaçant k à une position $\geqslant i+1$. Cet appel récursif renvoie r_2 le nombre de solutions dans lesquelles k est placé en position i' dans `L` avec $i' \geqslant i$.

  Les solutions générées par les deux appels récursifs étant distinctes deux à deux, il suffit de renvoyer $r_1 + r_2$.
- ⋆ Sinon (k ne peut pas être placé en position i), on essaye de placer k à une position $\geqslant i+1$ grâce à un appel récursif.

Voici un schéma représentant l'évolution de `L` au cours de l'exécution :



Exercices à rendre au plus tard le 26/01/2025 à 20h

Exercice 5. 0 + 0 = la tête à Toto

Cet exercice est issu du site France-ioi :

`http://www.france-ioi.org/algo/task.php?idChapter=513&idTask=509`

Comme vous le savez, $0+0=0$. On pourrait aussi dire $0=(0+0)$. Dans ce cas, on peut aussi aller un peu plus loin, et puisque 0 vaut $(0+0)$, remplacer les 0 de $(0+0)$ par leur valeur, et obtenir :

$$0 = ((0+0)+(0+0))$$

Rien n'empêche de continuer et d'écrire :

$$0 = (((0+0)+(0+0))+((0+0)+(0+0)))$$

1. Écrire une fonction récursive de signature « `tete_a_toto(n: int) -> str` » qui prend en entrée un entier N et renvoie la chaîne de caractères indiquant la valeur de 0, en ayant remplacé N fois les zéros à droite de l'égalité "0 = 0" par leur valeur "(0 + 0)".

   Exemple 1. Sur l'entrée 0, votre fonction doit renvoyer `"0"`.

   Exemple 2. Sur l'entrée 1, votre fonction doit renvoyer `"(0 + 0)"`.

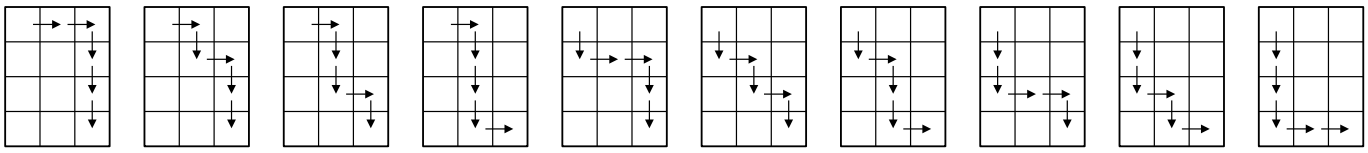
   Exemple 3. Sur l'entrée 2, votre fonction doit renvoyer `"((0 + 0) + (0 + 0))"`.

   Exemple 4. Sur l'entrée 3, votre fonction doit renvoyer :

   ```
   "(((0 + 0) + (0 + 0)) + ((0 + 0) + (0 + 0)))"
   ```

Exercice 6. Déplacement sur une grille

Dans cet exercice, vous n'êtes pas obligés d'utiliser des fonctions récursives. Soient $\texttt{n}, \texttt{m} \in \mathbb{N}^*$ deux entiers. Une fourmi se déplace sur une grille avec `n` lignes et `m` colonnes en utilisant uniquement des pas vers le bas et vers la droite. Son but est de partir du coin supérieur gauche de la grille et d'atteindre le coin inférieur droit. Par exemple, pour `n = 4` et `m = 3`, voici tous les chemins possibles :



L'objectif de l'exercice est de calculer le nombre de chemins différents que peut emprunter la fourmi. D'après les schémas ci-dessus, le nombre de chemins est 10 lorsque `n = 4` et `m = 3`.

1. Écrire une fonction `nb_chemins` qui prend en entrée n et m, et renvoie le nombre de chemins différents que peut emprunter la fourmi pour se rendre de la case en haut à gauche vers la case en bas à droite. Vérifier que pour $n = 34$ et $m = 56$, le nombre de chemins est d'environ 1.68×10^{24}. Si vous n'y arrivez pas, vous pouvez lire les indications ci-dessous
2. **Facultatif.** En fait, il existe une formule explicite pour le nombre de chemins faisant intervenir les coefficients binomiaux. Donner le nombre de chemins que peut emprunter la fourmi pour se déplacer de la case en haut à gauche vers la case en bas à droite, sous la forme d'un coefficient binomial. Justifier.

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On numérote les `n` lignes de la grille par un entier $\texttt{i} \in [\![\texttt{0}, \texttt{n} - \texttt{1}]\!]$: l'indice de la ligne du haut est `i = 0` et l'indice de la ligne du bas est $\texttt{i} = \texttt{n} - \texttt{1}$. De même, on numérote les `m` colonnes de la grille par un entier $\texttt{j} \in [\![\texttt{0}, \texttt{m} - \texttt{1}]\!]$: l'indice de la colonne de gauche est `j = 0` et l'indice de la colonne de droite est $\texttt{j} = \texttt{m} - \texttt{1}$. Pour tout $\texttt{i} \in [\![\texttt{0}, \texttt{n} - \texttt{1}]\!]$ et tout $\texttt{j} \in [\![\texttt{0}, \texttt{m} - \texttt{1}]\!]$, on note $\texttt{C}_{\texttt{i},\texttt{j}}$ le nombre de chemins entre la case d'indice `(0,0)` et la case d'indice `(i,j)`. Notez que $\texttt{C}_{\texttt{0},\texttt{0}} = \texttt{1}$ car le seul chemin pour aller de la case `(0,0)` vers la case `(0,0)` consiste à faire 0 pas. De plus, d'après l'exemple donné au début de l'exercice, $\texttt{C}_{\texttt{3},\texttt{2}} = \texttt{10}$.

3. (a) Soit $\texttt{i} \in [\![\texttt{0}, \texttt{n} - \texttt{1}]\!]$. Que vaut $\texttt{C}_{\texttt{i},\texttt{0}}$?
   (b) Soit $\texttt{j} \in [\![\texttt{0}, \texttt{m} - \texttt{1}]\!]$. Que vaut $\texttt{C}_{\texttt{0},\texttt{j}}$?
   (c) Soient $\texttt{i} \in [\![\texttt{1}, \texttt{n} - \texttt{1}]\!]$ et $\texttt{j} \in [\![\texttt{1}, \texttt{m} - \texttt{1}]\!]$. Exprimer $\texttt{C}_{\texttt{i},\texttt{j}}$ en fonction de $\texttt{C}_{\texttt{i}-\texttt{1},\texttt{j}}$ et $\texttt{C}_{\texttt{i},\texttt{j}-\texttt{1}}$.

Afin de calculer le nombre de chemins possibles, nous allons utiliser une technique appelée ***programmation dynamique***. Le principe est de créer une liste de listes « `C: list[list[int]]` » de taille `n` dont toutes les sous-listes sont de taille `m`. Initialement, `C` ne contient que des `None`, puis ses cases sont remplies à l'aide des formules de la question 3.

4. Écrire la fonction `nb_chemins` demandée dans la question 1.