Exercice 1. Affichage dans la console

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

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

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

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

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

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

Figures 1, 2 et 3.

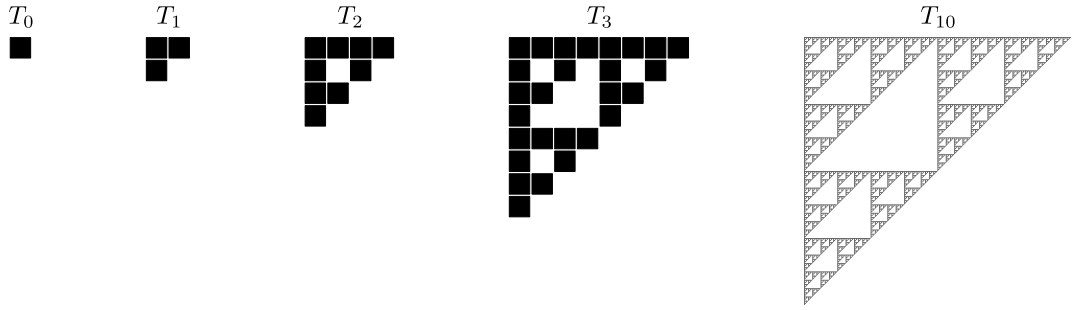
1. Étant donné un entier $n \in \mathbb{N}$, on souhaite afficher dans la console un triangle composé de n lignes. Par exemple, lorsque $n = 5$, on obtient la figure 1.
   (a) Expliquer comment afficher le triangle dans le cas où `n = 0`.
   (b) Supposons `n > 0` et avoir à notre disposition une procédure 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` ». **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 en `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 `i`ème ligne de T_n sous l'hypothèse `i` $\in [\![0, 2^{n} - 1]\!]$.

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 supprimer 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 L_1 et L_2 et renvoie :

   $$L = [L_1[0], L_2[0], L_1[1], L_2[1], L_1[2], L_2[2], \ldots].$$

   Dans le cas où L_1 et 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 $\mathcal{P}_n$ pour un certain `n` $\in \mathbb{N}$, si :

- ⋆ `len(L) = n`.
- ⋆ Chaque entier de $[\![1; n]\!]$ apparaît une et une seule fois dans `L`.
- ⋆ Il n'existe pas d'indice `i` $\geqslant 0$ tel que `L[i]` $\leqslant$ `L[i + 1]` $\leqslant$ `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 $1 \leqslant 4 \leqslant 5$.

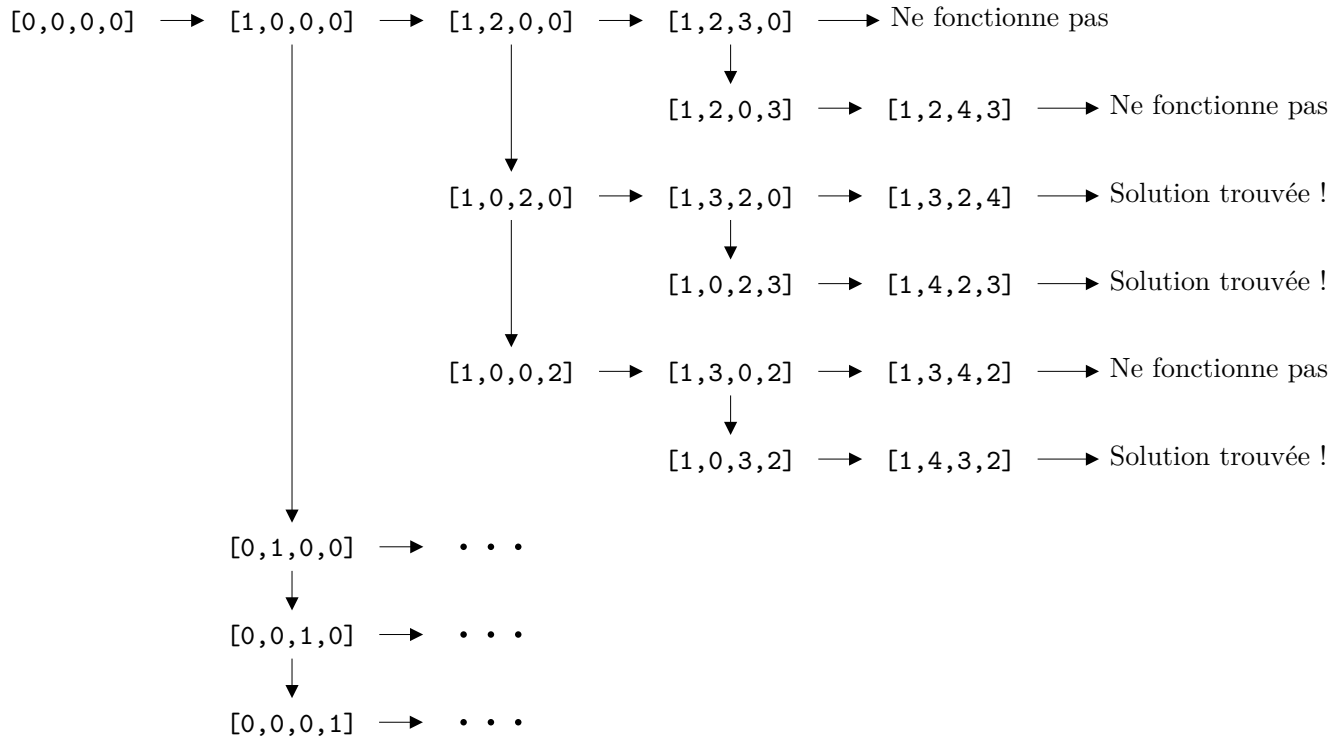
1. Écrire une fonction de signature « `compter_listes_Pn(n: int) -> int` » qui renvoie le nombre de listes vérifiant $\mathcal{P}_n$. Pour cela, vous devez utiliser une fonction récursive qui génère toutes les listes vérifiant $\mathcal{P}_n$. Si besoin, on pourra lire les indications ci-dessous.

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 :



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 0, 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)))"`.

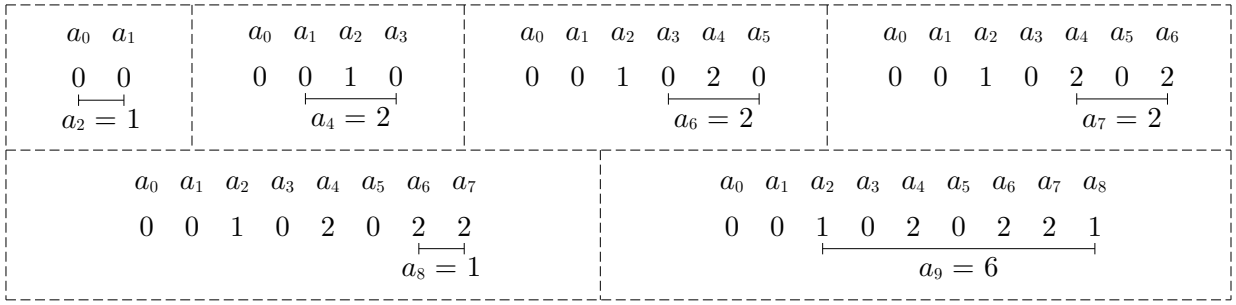
Exercice 6. Suite de Van Eck

Consignes.

- ⋆ Dans cet exercice, vous n'êtes pas obligés d'écrire des fonctions récursives (c'est même déconseillé pour la plupart des questions).
- ⋆ Pensez à tester vos fonctions et à laisser les tests dans votre fichier source. Vous perdrez des points si les tests n'apparaissent pas.
- ⋆ Des indications sont données à la fin du sujet. Essayez dans un premier temps de résoudre l'exercice sans ces indications.
- ⋆ Chaque question comporte également un test facultatif. Si vous souhaitez que votre programme puisse exécuter les tests facultatifs, vous devez concevoir une méthode plus efficace que celle des indications.

Énoncé. La suite de Van Eck $(a_n)_{n\in\mathbb{N}}$ est une suite infinie d'entiers naturels dont les dix premiers termes sont $0, 0, 1, 0, 2, 0, 2, 2, 1, 6$. Cette suite est définie récursivement : $a_0 = 0$ et pour tout $n \geqslant 1$:

- Si a_{n-1} n'apparaît pas dans $L = [a_0, a_1, \ldots, a_{n-2}]$, alors $a_n = 0$.
- Sinon, $a_n = n - m - 1$ où a_m est la dernière apparition de a_{n-1} dans L.



1. Calculer les termes $a_{10}, a_{11}, \ldots, a_{20}$ de la suite de Van Eck.
2. Écrire une fonction `vanEck` qui prend en entrée un entier n et renvoie a_n. Vérifier que $a_{20\,000} = 7$. **Facultatif :** vérifier que $a_{10\,000\,000} = 5\,522\,779$.

Conjectures et théorèmes

Le but de cette partie est de vérifier numériquement plusieurs conjectures et théorèmes concernant la suite de Van Eck.

Théorème 1. *La suite de Van Eck comporte une infinité de* 0*.*

Pour vérifier le théorème 1, on fixe un entier $k \in \mathbb{N}^*$ et on cherche à déterminer l'indice $z(k)$ tel que $a_{z(k)}$ est le $k^{\text{ème}}$ zéro de la suite de Van Eck. Par exemple :

$$z(1) = 0, \qquad z(2) = 1, \qquad z(3) = 3, \qquad z(4) = 5, \qquad z(5) = 10.$$

3. Écrire une fonction `kemeZero` qui prend en entrée un entier $k \in \mathbb{N}^*$ et renvoie $z(k)$. Votre fonction déclenchera une erreur si $k \leqslant 0$. Vérifier que $z(5000) = 31\,268$. **Facultatif :** vérifier que $z(1\,000\,000) = 7\,930\,437$.

Conjecture 1. *Tous les entiers naturels apparaissent dans la suite de Van Eck.*

Soit $n \in \mathbb{N}$ un entier. On souhaite déterminer le plus petit entier $k_n \in \mathbb{N}$ tel que k_n n'apparaît pas dans l'ensemble $A_n = \{a_0, a_1, a_2, \ldots, a_n\}$. Par exemple, $k_9 = 3$ car c'est le plus petit entier qui n'apparaît pas dans :

$$A_9 = \{0, 0, 1, 0, 2, 0, 2, 2, 1, 6\}$$

4. Écrire une fonction `minAbsents` qui prend en entrée un entier $n \in \mathbb{N}$ et renvoie k_n. Vérifier que `minAbsents(20_000)` vaut `379`. **Facultatif :** vérifier que `minAbsents(10_000_000)` vaut `53791`.

Conjecture 2. *Pour tout $n \in \mathbb{N}^*$, on pose $b_n = a_n/n$. On note également c_n le maximum de l'ensemble $\{b_1, b_2, \ldots, b_n\}$. Alors $\lim\limits_{n\to+\infty} c_n = 1$.*

5. Écrire une fonction `getC` qui prend en entrée un entier $n \in \mathbb{N}^*$, et renvoie c_n. vérifier que `getC(30_000)` vaut `0.963346...` **Facultatif :** vérifier que `getC(10_000_000)` vaut `0.990816...`

Conjecture 3. *Soit $A = \Big\{(k_1, k_2) \in \mathbb{N}^2 : (k_1, k_2) \neq (1, 1)$ et $k_2 \neq k_1 + 1\Big\} \cup \Big\{(0, 1)\Big\}$. Alors pour tout $(k_1, k_2) \in A$, il existe n tel que $a_n = k_1$ et $a_{n+1} = k_2$.*

6. Écrire une fonction de signature « `premiereOccurrence(k1: int, k2: int) -> int` » qui prend en entrée $k_1 \in \mathbb{Z}^2$, $k_2 \in \mathbb{Z}^2$, et renvoie le plus petit entier n tel que $a_n = k_1$ et $a_{n+1} = k_2$. Si $(k_1, k_2) \notin A$, votre fonction déclenchera une erreur. Vérifier que `premiereOccurrence(17, 382)` vaut `27635`. **Facultatif :** vérifier que `premiereOccurrence(31,1502)` vaut `9465773`.

Théorème 2. *Pour tout $(i, j) \in \mathbb{N}^2$, si $i \neq j$, $a_i \neq 0$ et $a_j \neq 0$ alors $i - a_i \neq j - a_j$.*

7. (a) En utilisant uniquement une compréhension de listes, écrire une fonction `listeDiff` qui prend en entrée une liste quelconque `[x0, x1, x2, ..., xn]`, et renvoie une liste contenant tous les `(i - xi)` où `xi` $\neq$ `0` et `i` $\in [\![$`0, n`$]\!]$.

   (b) En déduire une fonction `verifTheo2` qui prend en entrée un entier $n \in \mathbb{N}$ et renvoie :

   - `True` si $i - a_i \neq j - a_j$ pour tous les couples $(i, j) \in [\![0, n]\!]^2$ tels que $i \neq j$, $a_i \neq 0$ et $a_j \neq 0$.
   - `False` sinon.

   Dans cette question, vous n'avez pas le droit de considérer que le théorème 2 est connu. Vérifier que `verifTheo2(10_000)` vaut `True`. **Facultatif :** vérifier que `verifTheo2(10_000_000)` vaut `True`.

Indications (essayez d'abord de répondre aux questions sans lire ce qui suit)

2. (a) Écrire une fonction « `termeSuivant(L: list[int]) -> int` » qui prend en entrée une liste de la forme `[a0, a1, ..., an-1]` et renvoie `an`.

   (b) À l'aide de la fonction précédente, écrire une fonction « `listeVanEck(n: int) -> list[int]` » qui renvoie la liste `[a0, a1, ..., an]`.

(c) En déduire la fonction `vanEck`.

3. Si on utilise la fonction `listeVanEck`, on obtiendra une fonction trop lente. Il faut donc utiliser la fonction `termeSuivant`.
4. Écrire une fonction intermédiaire « `listePresents(n: int) -> list[bool]` » qui renvoie une liste `P` de taille `n+1` telle que `P[i]` vaut `True` si et seulement si `i` apparaît dans `listeVanEck(n)`. Pour que la fonction soit suffisamment rapide, vous ne devez parcourir la liste `listeVanEck(n)` qu'une seule fois.
6. Écrire une fonction intermédiaire « `estDansA(k1: int ,k2: int) -> bool` » qui indique si le couple `(k1,k2)` appartient à A.