Chapitre I
Le langage Python

Le programme d'informatique limite les éléments du langage Python dont la connaissance est exigible des étudiants, et toute utilisation d'autres éléments du langage doit être accompagnée d'une documentation. Dans ce premier chapitre, nous allons passer en revue ces différents éléments ; les titres des sections ainsi que les passages en italique sont extraits du programme officiel.

1. Traits généraux

1.1 Typage dynamique

L'interpréteur détermine le type à la volée lors de l'exécution du code.

Sur un ordinateur, toutes les données manipulées sont représentées par une suite de bits (une quantité qui vaut 0 ou 1) regroupées en octets (= 8 bits). Cette suite de bits est appelée la *valeur* de la donnée. Mais connaître la valeur de la donnée n'est pas suffisant pour déterminer de quel objet il s'agit, car des objets de natures différentes peuvent posséder la même valeur.

Exemple. L'entier `88` et le caractère `'X'` possèdent tous deux la valeur `01011000` dans un codage usuel sur un octet.
C'est pourquoi une donnée est représentée par sa valeur *et par un type* qui décrit la façon dont doit être interprétée cette suite de bits.

```
In [1]: type(88)
Out[1]: <class 'int'>

In [2]: type('X')
Out[2]: <class 'str'>
```

lorsqu'on définit une donnée par l'intermédiaire du clavier, une analyse syntaxique est réalisée lors de l'exécution du code pour déterminer le type de l'objet souhaité, puis sa valeur est calculée.

$$\texttt{2} \xrightarrow{\text{analyse syntaxique}} \texttt{int} \xrightarrow{\text{représentation}} (\texttt{int, 00000010})$$

$$\texttt{'2'} \xrightarrow{\text{analyse syntaxique}} \texttt{str} \xrightarrow{\text{représentation}} (\texttt{str, 00110010})$$

$$\texttt{2.} \xrightarrow{\text{analyse syntaxique}} \texttt{float} \xrightarrow{\text{représentation}} (\texttt{float, 00010000})$$

À l'inverse, lorsqu'un objet doit être affiché sur l'écran, son type permet de déterminer quelle forme lui donner.

$$(\texttt{int, 01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{88}$$

$$(\texttt{str, 01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{'X'}$$

$$(\texttt{float, 01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{1024.}$$

Remarque. Cette détermination « à la volée » permet de ne pas avoir à déclarer le type des paramètres utilisés par une fonction. Néanmoins, pour des raisons aussi bien pédagogiques que de sûreté du code, il est possible de préciser le type des arguments est du résultat des fonctions (c'est ce que fait l'école Centrale dans ses sujets d'informatique). Ainsi,

```
def maFonction(n:int, X:[float], c:str, u) -> (int, numpy.ndarray):
```

signifie que la fonction `maFonction` prend quatre arguments : le premier `n` est un entier, le second `X` est une liste de flottants, le troisième `c` une chaîne de caractères, le type du dernier `u` n'étant pas précisé. Cette fonction renvoie un couple dont le premier élément est un entier et le second un tableau numpy. Néanmoins il ne vous est pas demandé d'utiliser cette syntaxe et vous pouvez utiliser la syntaxe classique :

```
def maFonction(n, X, c, u):
```

1.2 Principe d'indentation

Les impératifs de la programmation structurée nécessitent la définition de blocs d'instructions au sein des structures de contrôles (`def`, `for`, `while`, `if`, ...). Certains langages utilisent des délimiteurs pour encadrer ces blocs d'instructions (des parenthèses en C, des mots-clés en Fortran, etc), mais le langage Python se distingue en utilisant l'*indentation*, qui favorise la lisibilité du code.
Le début d'un bloc d'instructions est défini par un double-point (`:`), la première ligne pouvant être considérée comme un en-tête. Le corps du bloc est alors indenté d'un nombre d'espaces fixes (quatre par défaut), et le retour à l'indentation de l'en-tête marque la fin du bloc.

```
en-tête:
    bloc ........................
    ............................
    d'instructions ..............
```

Il est possible d'imbriquer des blocs d'instructions les uns dans les autres :

```
en-tête 1:
    ............................
    ............................
    en-tête 2:
        bloc ....................
        ........................
        d'instructions ..........
    ............................
    ............................
```

Cette structuration sert entre autre à définir de nouvelles fonctions, à réaliser des tests ou à effectuer des instructions répétitives.

1.3 Portée lexicale

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.

```
In [1]: L = [1, 2, 3]

In [2]: def f():
....        L[0] = 0
....        return L

In [3]: f(), L
Out[3]: [0, 2, 3], [0, 2, 3]
```

```
In [1]: L = [1, 2, 3]

In [2]: def f():
....        L = [1, 2, 3]
....        L[0] = 0
....        return L

In [3]: f(), L
Out[3]: [0, 2, 3], [1, 2, 3]
```

```
In [1]: def f():
....        L[0] = 0
....        return L

In [2]: f()
NameError: name 'L' is not defined
```

Figure 1 – Illustration de la portée lexicale.

Considérons les trois exemples de la figure 1. Dans le premier cas, la liste `L` n'est pas définie au sein de la fonction `f`, donc c'est la liste définie dans l'espace global ligne 1 qui est modifiée. Dans le deuxième cas, une liste `L` est définie au sein de la fonction `f` donc c'est elle qui est modifiée par la fonction. Dans le dernier cas, la liste `L` n'est définie ni dans l'espace de la fonction, ni dans l'espace global donc l'interpréteur renvoie une erreur.

1.4 Appel de fonction par valeur

L'exécution de $f(x)$ évalue d'abord x puis exécute f avec la valeur calculée.

Observons le code suivant :

```
In [1]: a = 2

In [2]: def f(x):
....        x = 3
....        return x

In [3]: f(a), a
Out[3]: 3, 2
```

Dans un langage de programmation qui réaliserait un appel de fonction par *variable* plutôt que par *valeur*, le résultat retourné ligne 3 serait `3, 3` (la variable globale `a` aurait été modifiée).

Il convient néanmoins de noter que si l'argument de la fonction est un objet *mutable* (typiquement une liste) alors l'objet en question est effectivement modifié, car la valeur d'un objet mutable est l'adresse où cet objet est stocké en mémoire.

```
In [1]: L = [1, 2, 3]

In [2]: def f(X):
....        X[0] = 0
....        return X

In [3]: f(L), L
Out[3]: [0, 2, 3], [0, 2, 3]
```

2. Types de base

2.1 Opérations sur les entiers (`int`)

les opérations suivantes sont à connaître :

`+` l'addition de deux entiers ;

`-` la soustraction de deux entiers ;

`*` la multiplication de deux entiers ;

`**` l'élévation à la puissance (ne pas confondre avec `^` qui est une opération hors programme sur les entiers) ;

`//` le quotient de la division euclidienne ;

`%` *(avec des opérandes positifs)* le reste de la division euclidienne.

Ces opérations renvoient toujours un résultat de type `int`.

2.2 Opérations sur les flottants (`float`)

les opérations suivantes sont à connaître :

`+` l'addition de deux flottants ;

`-` la soustraction de deux flottants ;

`*` la multiplication de deux flottants ;

`/` la division de deux flottants ;

`**` l'élévation à la puissance.

Ces opérations renvoient toujours un résultat de type `float`.

2.3 Opérations sur les booléens (`bool`)

`not`, `or`, `and` *et leur caractère paresseux.*

Il convient d'expliquer la signification du caractère paresseux des opérateurs `or` et `and` : lors de l'évaluation d'une expression logique de type `(A and B)`, l'expression `A` est d'abord évaluée, *mais l'expression* `B` *n'est évaluée que si le résultat de l'évaluation de* `A` *est égal à* `True`. En effet, si `A` a été évalué à `False`, l'expression `(A and B)` sera elle aussi évaluée à `False`, quelle que soit l'évaluation de `B`.
Pour des raisons analogues, lors de l'évaluation de l'expression `(A or B)` l'expression `A` est d'abord évaluée, puis l'expression `B` *mais uniquement dans le cas où* `A` *aura été évaluée à* `False`.

Pour comprendre l'intérêt de cette évaluation paresseuse, considérons une fonction qui recherche un élément dans une liste et renvoie l'indice correspondant à sa première apparition :

```
def recherche(x, L):
    i = 0
    while i < len(L) and L[i] != x:
        i = i + 1
    return i
```

Lorsque `x` n'est pas présent dans la liste `L`, cette fonction renvoie la longueur de la liste :

```
In [1]: recherche(2, [1, 1, 1, 1])
Out[1]: 4
```

Inversons maintenant les deux composantes de l'expression booléenne et recommençons l'expérience :

```
def recherche(x, L):
    i = 0
    while L[i] != x and i < len(L):
        i = i + 1
    return i
```

```
In [2]: recherche(2, [1, 1, 1, 1])
IndexError: list index out of range
```

Dans ce deuxième cas de figure, lorsque l'entier `i` prend la valeur 4, la comparaison `L[i] != x` est réalisée en premier et conduit à une erreur puisqu'il n'y a pas de valeur indexée par 4 dans la liste `L`, alors que dans le premier cas de figure, c'est la comparaison `i < len(L)` qui est d'abord évaluée à `False` et permet, grâce au principe de l'évaluation paresseuse, de ne pas évaluer l'expression `L[i] != x`.

2.4 Comparaisons

Sont à connaître les opérations de comparaison suivantes :

`==` l'égalité ;

`!=` la différence ;

`<` strictement inférieur ;

`>` strictement supérieur ;

`<=` inférieur ou égal ;

`>=` supérieur ou égal.

Ces opérations de comparaison renvoient une valeur booléenne.

3. Types structurés

3.1 Structures indicées immuables (chaînes, tuples)

Une structure de donnée est *indicée* lorsqu'on peut accéder à ses éléments individuels par l'intermédiaire de leur indice ; une structure de donnée est *immuable* lorsque ces éléments individuels ne peuvent être modifiés. Deux structures de ce type sont à connaître :

– les chaînes de caractère (le type `str`) qui sont des suites de caractères alphanumériques délimités par des guillemets simples ou doubles. Par exemple `'Le nœud de vipères'` est une chaîne de caractère.

– les tuples (le type `tuple`) ou n-uplets qui sont des suites finies de valeurs séparées par des virgules (si besoin enclos par des parenthèses). Par exemple, `(2, 3, 5, 7, 11)` est un tuple.

Ces structures partagent les opérations suivantes :

– la fonction `len` permet de calculer leur longueur ;

```
In [1]: len('Le nœud de vipères')
Out[1]: 18

In [2]: len((2, 3, 5, 7, 11))
Out[2]: 5
```

– on accède aux éléments individuels avec la syntaxe `[k]` où k est un indice positif valide ;

```
In [3]: 'Le nœud de vipères'[4]
Out[3]: 'œ'

In [4]: (2, 3, 5, 7, 11)[3]
Out[4]: 7
```

– on peut en calculer une tranche avec la syntaxe `[i: j]` qui extrait tous les éléments dont les indices sont compris entre i inclus et j exclus ;

```
In [5]: 'Le nœud de vipères'[3: 7]
Out[5]: 'nœud'
```

– ces structures peuvent êtres concaténées avec l'opérateur `+` et répétées avec l'opérateur `*`.

```
In [6]: 'François' + ' Mauriac'
Out[6]: 'François Mauriac'

In [7]: (1,) * 10
Out[7]: (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

(Notez au passage comment se définit un tuple de longueur 1.)

3.2 Listes

Trois modes de création sont à connaître :

– par *compréhension* en suivant la syntaxe `[expr fox x in s]` ;

– par *duplication* en suivant la syntaxe `[expr] * n` ;

– par `.append` successifs.

La création par duplication présente deux inconvénients : elle ne permet que de créer des listes de valeurs égales et surtout, *elle se révèle incorrecte lorsque cette valeur est de type mutable*. Elle est donc à mes yeux à déconseiller. Pour créer la liste des dix premiers carrés on écrira donc l'une ou l'autre des versions suivantes :

```
carres = [x**2 for x in range(1, 11)]
```

```
carres = []
for x in range(1, 11):
    carres.append(x**2)
```

Tout comme pour les tuples et les chaînes de caractères, la fonction `len` renvoie la longueur d'une liste, on accède aux éléments par indice positif valide, on peut en calculer une tranche et réaliser une concaténation de deux listes par l'opérateur `+`.
Cependant, il faut prendre garde au fait que les deux syntaxes `L.append(x)` et `L = L + [x]`, même si elles conduisent au même résultat apparent, *ne sont pas équivalentes*. En effet, la seconde version *recrée* une nouvelle liste augmentée d'un élément là où la première se contente de rajouter un élément à une liste existante. On s'en doute, la deuxième version peut se révéler beaucoup plus coûteuse pour une liste de grande taille. Pour cette raison, on utilisera avec la plus grande parcimonie l'opérateur de concaténation `+` pour les listes.

La principale différence avec les structures de données précédentes est que les listes sont des structures de données *mutables*, dans le sens où il est possible de modifier un élément individuel d'une liste sans avoir à recréer la liste dans son entièreté.
La copie d'une liste `L` se réalise par la méthode `L.copy()` ou par le calcul d'une tranche comprenant l'entièreté de la liste `L[:]`.
Enfin, la méthode `L.pop()` supprime de la liste `L` son dernier élément et le renvoie en valeur de retour. Cette méthode modifie la liste en temps constant, contrairement à une syntaxe de type `L = L[0:len(L)-1]` qui recalculerait l'entièreté de la liste moins son dernier élément (et serait donc bien moins efficiente).

3.3 Dictionnaires (hors programme en 2021-22)

Un dictionnaire présente de nombreuses similitudes avec les listes, si ce n'est qu'au lieu d'accéder aux éléments individuels par le biais d'un indice, on y accède par le biais d'une *clef*.
La création d'un dictionnaire se réalise en suivant la syntaxe `{c1: v1, ... , cn: vn}` où $c_1,\dots,c_n$ sont des clefs (nécessairement deux-à-deux distinctes) et $v_1,\dots,v_n$ les valeurs qui leur sont associées. Ainsi, `{ }` crée un dictionnaire vide.

Si `D` est un dictionnaire et `c` une clef,

– l'expression `c in D` renvoie un booléen indiquant si la clef est présente ou non dans le dictionnaire ;

– `D[c]` renvoie la valeur associée à la clef si celle-ci est présente dans le dictionnaire (et provoque une erreur sinon) ;

– `D[c] = v` crée une nouvelle association si la clef n'est pas présente dans le distionnaire, et modifie l'association précédente sinon.

4. Structures de contrôle

4.1 Instruction d'affectation

Elle se réalise avec `=`. Notons qu'il est possible de *dépaqueter* des tuples, autrement dit utiliser la syntaxe :

```
x, y, z = a, b, c
```

pour affecter simultanément les valeurs a,b,c aux variables x,y,z. Ceci peut s'avérer utile pour permuter le contenu de deux variables en écrivant `x, y = y, x`.

4.2 Instruction conditionnelle

La syntaxe est la suivante :

```
if condition_1:
    bloc d'instructions_1
elif condition_2:
    bloc d'instructions_2
elif condition_3:
    bloc d'instructions_3
...
else:
    bloc d'instructions
```

Seul l'un de ces quatre (ou plus) blocs d'instructions sera réalisé :

- le premier si la condition 1 est réalisée ;
- le deuxième si la condition 1 n'est pas réalisée et la condition 2 réalisée ;
- le troisième si les conditions 1 et 2 ne sont pas réalisées et la condition 3 réalisée ;
- ...
- le dernier si aucune des conditions n'est réalisée.

Notez que les mots-clefs `elif` et `else` sont optionnels : s'ils ne sont pas présents et que la condition 1 n'est pas vérifiée, il ne se passe rien.

4.3 Boucle conditionnelle

Elles suivent la syntaxe :

```
while condition:
    bloc..............
    d'instructions......
```

Le bloc d'instructions est réalisé tant que la condition est vérifiée. Il est néanmoins possible de forcer la sortie d'une boucle à l'aide d'un `return` (exclusivement dans le cadre de la définition d'une fonction) ou plus généralement à l'aide de `break`.

4.4 Énumération

L'instruction

```
for x in X:
    bloc..............
    d'instructions......
```

exécute le bloc d'instructions pour chacun des objets présents dans `X`, ce dernier pouvant être une chaîne de caractère, un tuple, une liste, un dictionnaire ou une itération de type `range(a, b)`.

4.5 Définition d'une fonction

Elle suit la syntaxe :

```
def f(x1, ... , xn):
    bloc..............
    d'instructions......
    return ...
```

En l'absence de `return`, la valeur renvoyée est égale à `None`
Un `return` peut se trouver au sein du bloc d'instructions, auquel cas son exécution interrompt immédiatement l'exécution du code au sein de la fonction.

5. Divers

5.1 Commentaires

Ils sont précédés du caractère `#`.

5.2 Utilisation simple de `print`

Envoie une chaîne de caractère en direction de la sortie standard (par défaut la console dans laquelle s'exécute votre code). À ne surtout pas confondre avec le résultat d'une fonction renvoyé par `return`.

5.3 Importation de modules

On importe l'entièreté d'un module par la syntaxe `import module` ou `import module as alias`. par exemple :

```
import numpy
```

```
import numpy as np
```

Dans le premier cas, chaque fonction du module `numpy` devra être précédée du préfixe `numpy`, dans le second cas du préfixe `np` :

```
In [1]: numpy.cos(numpy.pi)
Out[1]: -1.0
```

```
In [1]: np.cos(np.pi)
Out[1]: -1.0
```

Il est aussi possible de n'importer qu'une seule fonction d'un module donné à l'aide de la syntaxe

`from module import fonction`.

5.4 Manipulation de fichiers texte

La documentation utile de ces fonctions doit être rappelée ; tout problème relatif aux encodages est éludé.
Pour illustrer les différentes manipulations permises sur un fichier texte, nous allons prendre l'exemple d'un fichier CSV (pour *Comma-Separated Values*). Il s'agit d'un fichier texte représentant des données tabulaires sous formes de valeurs séparées par des virgules. Pour notre exemple, nous allons imaginer un fichier intitulé `naissances.csv` dont le contenu est le suivant :

```
Sexe, Prénom, Année de naissance
M, Alphonse, 1932
F, Béatrice, 1964
F, Charlotte, 1988
```

Avant de pouvoir être lu, un fichier texte doit être ouvert à l'aide de l'instruction `open` :

```
fichier = open('naissances.csv', 'r')
```

Le paramètre `'r'` indique que nous voulons accéder à ce fichier en mode lecture (*read*).
La méthode `.read` lit tout le contenu d'un fichier et renvoie une chaîne de caractère unique :

```
In [1]: fichier.read()
Out[1]: Sexe, Prénom, Année de naissance\nM, Alphonse, 1932\nF, Béatrice, 1964\nF,
....    Charlotte, 1988
```

Le caractère `\n` qui apparaît dans la réponse correspond à l'encodage d'un passage à la ligne.

la méthode `.readline()` lit une ligne du fichier et la renvoie sous forme d'une chaîne de caractères. À chaque nouvel appel de la méthode la ligne suivante est renvoyée.

```
In [2]: fichier.readline()
Out[2]: 'Sexe, Prénom, Année de naissance\n'
```

À la fin du fichier, cette méthode renvoie la chaîne de caractères vide `''`, ce qui permet de réaliser une itération conditionnelle d'un fichier. Cependant, il est plus simple dans ce cas d'utiliser une énumération du fichier :

```
In [3]: for ligne in fichier:
....         print(ligne)
M, Alphonse, 1932

F, Béatrice, 1964

F, Charlotte, 1988
```

Notez que la première ligne n'apparaît pas car elle a déjà été lue par l'instruction 2. Notez aussi que le caractère `\n` a bien été interprété comme un passage à la ligne par l'instruction `print`.

La méthode `.readlines()` lit l'entièreté du fichier et renvoie une liste dont les éléments sont les lignes de ce fichier :

```
In [1]: L = fichier.readlines()

In [2]: L
Out[2]: ['Sexe, Prénom, Année de naissance\n', 'M, Alphonse, 1932\n',
         'F, Béatrice, 1964\n', 'F, Charlotte, 1988\n']
```

Si on veut modifier le contenu d'un fichier, il faut l'ouvrir avec l'instruction open mais avec en option `'w'` (pour *write*) pour créer un nouveau fichier (dans ce cas, un fichier existant du même nom serait détruit) ou `'a'` (pour *append*) pour ajouter du texte supplémentaire à un fichier existant, puis utiliser la méthode `.write`. Par exemple, pour ajouter une ligne au fichier CSV donné en exemple il faudrait écrire :

```
In [1]: fichier = open('naissances.csv', 'a')

In [2]: fichier.write('M, Dereck, 2002\n')
```

Dans tous les cas de figure (lecture ou écriture) il est de bon ton de fermer un fichier ouvert une fois le travail terminé :

```
In [3]: fichier.close()
```

Remarque. Pour utiliser les données d'un fichier CSV une fois lues, il faut être capable de séparer ces données textuelles qui pour l'instant sont fournies sous la forme d'une chaîne de caractères. La méthode `.split()` permet de séparer les différents éléments d'une chaîne de caractères :

```
In [1]: L[3]
Out[1]: 'F, Charlotte, 1988\n'

In [2]: L[3].split(',') # l'argument sert à préciser le séparateur utilisé
Out[2]: ['F', ' Charlotte', ' 1988\n']
```

Notez qu'il resterait à convertir le dernier argument en un objet de type entier et à enlever un espace devant le prénom.

5.5 Assertion

un `assert` est une aide au débogage qui, à l'entrée d'une fonction, vérifie si certaines conditions sont vérifiées. Pour qu'une fonction ne puisse s'appliquer qu'à un entier positif (par exemple pour définir la fonction factorielle), on écrira :

```python
def fact(n):
    assert isinstance(n, int)
    assert n >= 0
    f = 1
    for k in range(2, n+1):
        f = f * k
    return f
```

```
In [1]: fact(5)
Out[1]: 120

In [2]: fact(-5)  # l'argument n'est pas positif
AssertionError

In [3]: fact(5.)  # l'argument n'est pas entier
AssertionError
```

6. Exercices

Exercice 1

On appelle *espace binaire* d'un entier naturel n toute séquence consécutive de `0` délimités par deux `1` dans la décomposition en base 2 de n. Par exemple, le nombre 529 possède deux espaces binaires de longueurs respectives 3 et 4 car $529 = (1000010001)_2$. En revanche, 32 ne possède pas d'espace binaire puisque $32 = (100000)_2$.
Rédiger une fonction `espacemax` qui prend pour argument un entier naturel et renvoie la longueur du plus grand espace binaire présent dans n s'il en existe, et la valeur 0 sinon.

Exercice 2

On appelle *rotation* d'un tableau t le fait de décaler tous les éléments d'une place vers la droite, à l'exception du dernier qui est placé en première place. Par exemple, la rotation du tableau $[1,2,3,4]$ est le tableau $[4,1,2,3]$.

a. Rédiger une fonction `rotation(t)` qui renvoie un *nouveau* tableau égal à la rotation du tableau initial.

b. Rédiger une fonction `rotation2(t)` qui *modifie* le tableau t pour le remplacer par sa rotation.

c. Rédiger une fonction `rotation3(k, t)` qui renvoie un nouveau tableau égal à k rotations du tableau t.

Exercice 3

À partir d'un tableau t d'entiers naturels, rédiger une fonction `entierManquant(t)` qui renvoie le plus petit entier naturel absent du tableau. Par exemple, pour $t = [1,3,7,6,4,1,2,0]$ cette fonction devra renvoyer l'entier 5.

Exercice 4

Un tableau non vide t de n entiers relatifs étant donné, on cherche la valeur maximale de la quantité $\Delta_k = (t[0]+\cdots+t[k]) - (t[k+1]+\cdots+t[n-1])$ lorsqu'on fait varier k dans $[\![0, n-2]\!]$.

a. Rédiger une fonction `delta(k, t)` qui calcule la quantité Δ_k et en déduire une fonction `equilibre(t)` qui résout le problème posé. Évaluer la complexité temporelle de cette dernière fonction.

b. Trouver une fonction `equilibre2(t)` qui résout ce problème en temps linéaire.

Exercice 5

Une petite grenouille se trouve face à une rivière. Initialement située sur une des deux rives (position 0), elle veut se rendre sur la rive opposée (position $x+1$) mais ne peut réaliser que des sauts d'une unité. Heureusement pour elle, des feuilles tombent sur la surface de la rivière et peuvent lui permettre de passer de feuille en feuille.

On donne un tableau t composé de n entiers représentant les feuilles qui tombent : $t[k]$ représente la position où une feuille tombe à l'instant k. L'objectif est de trouver le moment le plus précoce où la grenouille pourra passer d'une rive à l'autre, c'est-à-dire la date où toutes les positions de 1 à x seront couvertes par une feuille.
Rédiger une fonction `grenouille(x, t)` qui résout ce problème. par exemple, pour $x = 5$ et $t = [1, 3, 1, 4, 5, 3, 2, 4]$ cette fonction devra renvoyer l'entier 6.
Évaluer la complexité temporelle et spatiale de votre fonction.

Exercice 6

Un tableau contient un nombre impair d'entiers positifs. Chacun de ces entiers est présent un nombre pair de fois, à l'exception d'un seul.
Rédiger une fonction `impair(t)` qui prend pour argument un tel tableau et renvoie cet unique entier présent un nombre impair de fois. Analysez la complexité de votre algorithme.