Ces exercices visent à installer et développer une certaine aisance dans l'écriture de code ***Python*** *.* ***Certains sont adaptés à la spécialité mathématiques, quand d'autres excèdent ce niveau :*** *la mention « NSI » identifiera ces exercices. La mention « Tale » pourra également venir préciser le niveau principalement concerné.*

Dans de nombreux exercices, une ou plusieurs « assertions[1] » sont indiquées : *en copiant-collant ces instructions « de validation » après la définition de vos fonctions (sautez une ligne pour plus de lisibilité), vous pourrez vérifier que votre code fonctionne comme attendu sur les exemples qu'elles proposent.* ***Attention :*** *il est souvent impossible de tester l'égalité de quantités numériques* ***: vous devrez donc toujours chercher à proposer de meilleures assertions[2]****.*

« Bonus permanent » en NSI : *toujours chercher à déterminer les préconditions que devraient vérifier les paramètres reçus par les fonctions que vous écrirez !*

ARITHMÉTIQUE

EXERCICE 1 ★☆☆ — *Pair*

Écrire une fonction `entier_est_pair()` qui teste si un nombre entier naturel est pair. Cette fonction prendra en paramètre un entier naturel et renverra le booléen `True` s'il est pair, `False` sinon.

Aide : on utilisera l'opérateur `%` qui renvoie le reste de la division euclidienne d'un entier par un autre. Ainsi, `4 % 2` donne 0 alors que `5 % 2` donne 1.

Validation :

```
assert entier_est_pair(2) == True, "/!\ entier_est_pair()"
assert entier_est_pair(3) == False, "/!\ entier_est_pair()"
```

EXERCICE 2 ★☆☆ — *Impair*

Écrire une fonction `entier_est_impair()` qui teste si un nombre entier naturel est impair. Cette fonction prendra en paramètre un entier naturel et renverra le booléen `True` s'il est impair, `False` sinon.

Aide : on pourra utiliser la fonction définie à l'exercice précédent !

Validation :

```
assert entier_est_impair(2) == False, "/!\ entier_est_impair()"
assert entier_est_impair(3) == True, "/!\ entier_est_impair()"
```

EXERCICE 3 ★☆☆ — *I never give you my number ♫*

Écrire une fonction `donne_chiffres()` qui prend en paramètres un nombre entier ainsi qu'un entier qui déterminera une puissance de 10, et renvoie le chiffre qui correspond à la puissance de 10. On utilisera les instructions `%` et `//` (quotient de la division entière de deux nombres).

Aide : `42 // 10` donne 4 et `42 % 10` donne 2.

Exemple : `donne_chiffres(4059, 3)` renverra `4`, quand `donne_chiffres(4059, 0)` renverra `9`.

Validation :

```
assert donne_chiffres(4059, 3) == 4, "/!\ donne_chiffres()"
```

1. Une assertion permet de vérifier la conformité d'une situation : si la vérification échoue, une erreur survient (on dit qu'une *exception est levée*). Par exemple, il est prudent de s'assurer qu'une quantité est non-nulle avant de calculer son inverse…
2. Il suffit de vérifier que la valeur absolue de l'écart entre le résultat théorique et le résultat approché est assez faible !

EXERCICE 4 ★½☆ — *Diviser (pour régner ou sans régner ?)*

1. Écrire une fonction `diviseurs_naif()` qui prend en paramètre un entier `n` et renvoie l'ensemble de ses diviseurs sous forme de liste PYTHON. On privilégiera une approche basique, « naïve ».

   VALIDATION : `assert diviseurs_naif(20) == [1, 2, 4, 5, 10, 20], "/!\ diviseurs_naif()"`

2. Écrire une fonction `diviseurs_optimise()`, qui sera une version plus efficiente de la précédente. Voici le principe de l'amélioration :

   - il est clair que tout entier n est divisible par 1 et n ;
   - en outre, si on trouve un diviseur a de n, alors il existe un entier b tel que $n = a \times b$. Alors b est nécessairement un diviseur de n !
   - si on commence à explorer les diviseurs a de n en commençant par les « petits » (proches de 1), les diviseurs b associés seront « grands ». Mais à chaque fois qu'on cherche un diviseur a plus grand que le précédent, le diviseur b associé sera, lui, plus petit — et ce, à chaque étape de la recherche. À un certain moment, a finira par être plus grand que b, et il sera inutile de continuer la recherche car les nouvelles valeurs de a correspondront toutes à des diviseurs b déjà trouvés ! La bascule se fera dès que $a > \sqrt{n}$, car $\sqrt{n} \times \sqrt{n} = n$.

     EXEMPLE : $20 = 1\times 20 = 2\times 10 = 4\times 5 = 5\times 4 = \ldots$ et il est inutile d'aller plus loin car $a = 5 > \sqrt{20} \approx 4{,}47$.

   Vous allez donc employer dans le code de cette fonction deux listes PYTHON, désignées par les variables `petits_diviseurs` et `grands_diviseurs` :

   - la 1re contiendra initialement 1, et mémorisera au fur et à mesure les diviseurs a de n tels que $a \leqslant \sqrt{n}$;
   - la 2nde contiendra initialement n et mémorisera progressivement les diviseurs b de n tels que $n = a \times b$.

   La fonction `diviseurs_optimise()` devra renvoyer la liste résultant de la fusion des listes précédentes !

EXERCICE 5 ★☆☆ — *Instinct primal*

Écrire une fonction `entier_est_premier()` qui teste si un nombre entier naturel est premier, c'est-à-dire s'il n'est divisible que par 1 et par lui-même (ni 0 ni 1 ne sont donc premiers). Cette fonction prendra en paramètre un entier naturel et renverra le booléen `True` s'il est premier, `False` sinon.

AIDE : on pourra naturellement recycler l'une des fonctions décrites dans l'exercice précédent.

VALIDATION :
```
assert entier_est_premier(13) == True, "/!\ entier_est_premier()"
assert entier_est_premier(51) == False, "/!\ entier_est_premier()"
```

EXERCICE 6 ★½☆ — *Instinct primal 2, le retour !*

1. Écrire une fonction `premiers()` qui prend en paramètre un entier n et renvoie la liste des n premiers nombres premiers.
2. Écrire une fonction `premiers_inf()` qui prend en paramètre un entier n et renvoie la liste de tous les nombres premiers inférieurs ou égaux à n.

VALIDATION :
```
assert premiers(6) == [2, 3, 5, 7, 11, 13], "/!\ premiers()"
assert premiers_inf(16) == [2, 3, 5, 7, 11, 13], "/!\ premiers_inf()"
```

***BONUS :** utiliser la technique dite de « compréhension de liste » !*

EXERCICE 7 ★☆☆ — *OK, here is my whole number...*

Écrire une fonction `liste_chiffres()` qui renvoie dans une liste chacun des chiffres qui composent le nombre entier passé en paramètre. On utilisera les instructions `%` et `//` (quotient de la division entière de deux nombres).

AIDE : `42 // 10` donne 4 et `42 % 10` donne 2.

VALIDATION : `assert liste_chiffres(4059) == [4, 0, 5, 9], "/!\ liste_chiffres()"`

EXERCICE 8 ★☆☆ — *Criblé de cratères...*

Le crible d'**ÉRATHOSTÈNE** est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné n. L'algorithme procède par élimination : il s'agit de supprimer d'une liste des entiers compris entre 2 et n tous les multiples d'un entier (autres que lui-même). En supprimant tous ces multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier à part 1 et eux-mêmes, et qui sont donc les nombres premiers.

On commence par éliminer les multiples de 2, puis les multiples de 3 restants, puis les multiples de 5 restants, etc. en éliminant à chaque fois tous les multiples du plus petit entier restant. Une optimisation classique consiste à arrêter le processus dès lors que le carré du plus petit entier restant est supérieur au plus grand entier restant, car dans ce cas, tous les nombres non-premiers ont déjà nécessairement été rayés.

Écrire une fonction `erathostene()` qui prend pour unique paramètre un entier naturel n et renvoie la liste des nombres premiers inférieurs ou égaux à n en utilisant l'algorithme d'ÉRATHOSTÈNE.

VALIDATION : `assert erathostene(20) == [2, 3, 5, 7, 11, 13, 17, 19], "/!\ erathostene()"`

EXERCICE 9 ★★☆ — *Nobody's perfect. Well, some are!*

Un entier naturel est dit **parfait** lorsqu'il est égal à la somme de tous ses diviseurs propres autres que lui-même (par exemple 6 est parfait car $6 = 1+2+3$). Écrire une fonction `est_parfait()` qui teste si un entier passé en paramètre est parfait ou non (la fonction retournera naturellement les booléens `True` si c'est le cas et `False` sinon).

VALIDATION :
```python
assert est_parfait(28) == True, "/!\ est_parfait()"
assert est_parfait(5) == False, "/!\ est_parfait()"
```

***REMARQUE :** 496 et 8128 sont d'autres nombres parfaits.*

EXERCICE 10 ★☆☆ — *Parfait fruit de la passion*

Écrire une fonction `liste_parfaits()` qui prend en paramètre un entier n et qui renvoie une liste PYTHON de tous les nombres parfaits inférieurs ou égaux à n.

***BONUS :** utiliser une compréhension de liste !*

EXERCICE 11 Tale ★★★ — *PGCD*

Le **PGCD** (**P**lus **G**rand **C**ommun **D**iviseur) de deux nombres entiers non nuls est le plus grand entier qui les divise simultanément.

EXEMPLE : le PGCD de 20 et de 30 est **10** (leurs diviseurs communs sont 1, 2, 5 et **10**).

L'**algorithme d'EUCLIDE** permet de déterminer le PGCD de deux entiers. En voici le principe :

- si le reste de la division de m par n vaut 0, alors le PGCD des entiers m et n est n ;
- sinon, le PGCD de m et n est égal au PGCD de n et du *reste* de la division entière (on dit aussi « division euclidienne ») de m par n.

EXEMPLES :

- $\text{PGCD}(20;15) = \text{PGCD}(15;5) = 5$ car $20 = 1 \times 15 + 5$;
- $\text{PGCD}(1\,245;123) = \text{PGCD}(123;15)$ car $1\,245 = 10 \times 123 + 15$; puis $\text{PGCD}(123;15) = \text{PGCD}(15;3)$ car $123 = 8 \times 15 + 3$; enfin, $\text{PGCD}(15;3) = 3$ car 3 divise 15 ;
- $\text{PGCD}(28;15) = \text{PGCD}(15;13)$ car $28 = 1 \times 15 + 13$; puis $\text{PGCD}(15;13) = \text{PGCD}(13;2)$ car $15 = 1 \times 13 + 2$; enfin, $\text{PGCD}(13;2) = \text{PGCD}(2;1) = 1$, car $13 = 6 \times 2 + 1$. Dans cette situation, on dit que 28 et 15 sont *premiers entre eux*.

Écrire une fonction `pgcd` qui prenne en paramètres deux entiers positifs m et n retourne leur PGCD.

VALIDATION : `assert pgcd(1245,123) == 3, "/!\ pgcd()"`

***BONUS NSI :** programmer cette fonction à l'aide d'un algorithme récursif.*

EXERCICE 12 ★☆☆ — *Emordnilap*

On dit qu'un entier est **palindromique** si son écriture est symétrique (sa valeur ne change pas, que l'on lise les chiffres qui le composent de la gauche vers la droite ou de la droite vers la gauche). Exemple : 123 321 est un palindrome. Écrire une fonction `est_palindrome()` qui teste si un entier est un palindrome (elle renverra naturellement le booléen `True` si c'est le cas, `False` sinon).

AIDE : la fonction `liste_chiffres()` décrite à l'exercice nº 7 peut être réutilisée. Les élèves de NSI pourront ruser en convertissant l'entier en chaîne de caractères…

VALIDATION :

```
assert est_palindrome(121) == True, "/!\ est_palindrome()"
assert est_palindrome(123) == False, "/!\ est_palindrome()"
```

FONCTIONS : TRACÉ ET RÉSOLUTION D'ÉQUATION

EXERCICE 13 ★★☆ — *Tableau de valeurs et tracé de fonction*

On suppose définie une fonction PYTHON nommée `f`, qui prend un paramètre `x` et ne fait que renvoyer le résultat d'un calcul mathématique correspondant à la seule évaluation d'une expression dépendant de ce paramètre `x`. Par exemple : `def f(x): return x**2+1`. C'est le pendant informatique de la définition d'une fonction mathématique f par l'expression $f(x) = x^2 + 1$.

Écrire une fonction `tableau_valeurs()` prenant 3 paramètres, dans l'ordre suivant :

- `xmin` : la valeur minimale du paramètre `x` précédemment évoqué ;

- `xmax` : la valeur maximale de ce même paramètre `x` ;
- `n` : le nombre total de valeurs **équiréparties** entre `xmin` et `xmax` *incluses* (valeurs sur lesquelles la fonction `f` sera évaluée).

La fonction renverra deux listes Python, la 1re contenant les différentes valeurs de `x`, la 2nde contenant les valeurs correspondantes de `f(x)`.

Validation : avec la fonction `f` définie comme dans l'exemple précédent par $f(x) = x^2 + 1$, on pourra utiliser l'assertion `assert tableau_valeurs(1,2,2) == ([1,2],[2,5]), "/!\ tableau_valeurs()"`.

*Remarque : le code que vous aurez écrit posera probablement des soucis. Essayez par exemple la fonction $f(x) = \frac{1}{x}$ et l'appel `print(tableau_valeurs(-1,1,3))` : si vous obtenez une erreur, c'est normal ! Le problème vient de ce qu'avoir 3 valeurs de x équiréparties entre -1 et 1 implique qu'on cherchera à calculer $f(0)$, ce qui ne se peut pas (diviser par 0 est impossible). Un mécanisme existe pour gérer ces situations, en Python : les **exceptions**.*

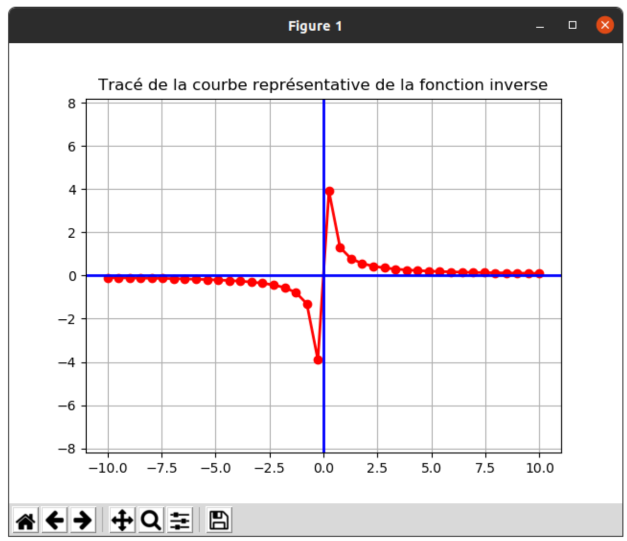
***Bonus :** du tableau de valeurs au graphique ! On peut utiliser la bibliothèque `matplotlib` pour générer un tracé d'une fonction, à l'aide de la fonction `tableau_valeurs()` définie à l'exercice précédent. Saisissez le code ci-dessous et observez ses effets :*

```python
import matplotlib.pyplot as plt

def f(x): return 1/x

def tableau_valeurs(xmin, xmax, n):
    ... # Code à compléter

xs, fxs = tableau_valeurs(-10, 10, 40)
plt.plot(xs, fxs, "or-", linewidth=2)
plt.axis([min(xs), max(xs), min(fxs), max(fxs)])
plt.axis('equal')
plt.axhline(linewidth=2, color='b')
plt.axvline(linewidth=2, color='b')
plt.title("Tracé de la fonction inverse")
plt.grid()
plt.show()
```



Remarque : comme vous pourrez le constater dans la capture d'écran ci-dessus, il reste certains problèmes !

EXERCICE 14 ★☆☆ — *Résoudre une équation par la méthode du « balayage »*

On suppose définie une fonction Python nommée `f`, qui prend un paramètre `x` et ne fait que renvoyer le résultat d'un calcul mathématique correspondant à la seule évaluation d'une expression dépendant de ce paramètre `x`. Par exemple : `def f(x): return x**2-2`. C'est le pendant informatique de la définition d'une fonction mathématique f par l'expression $f(x) = x^2 - 2$.

L'objectif est de trouver une valeur approchée d'une solution de l'équation $f(x) = 0$, à l'aide d'un algorithme dit « de balayage ». Il faut tout d'abord être certain que la fonction f est définie sur un intervalle $[a\,;b]$ qui contient une valeur α telle que $f(\alpha) = 0$. Pour obtenir un encadrement à 10^{-n} près de α, on effectue les opérations suivantes :

- on commence par balayer l'intervalle $[a\,;b]$ avec un pas de 1 (si c'est possible, 0,1 ou 0,01 ou ... si ça ne l'est pas). Cela implique de calculer $f(a)$, $f(a+1)$, $f(a+2)$, etc. On s'arrête dès qu'on a trouvé deux valeurs consécutives x_0 et $x_0 + 1$ pour lesquelles $f(x_0)$ et $f(x_0 + 1)$ sont de signes opposés : la solution α de $f(x) = 0$ appartiendra alors à l'intervalle $[x_0\,;x_0 + 1]$;

- on balaie ensuite l'intervalle $[x_0\,;x_0+1]$ avec un pas de 0,1. On calcule donc $f(x_0)$, $f(x_0+0{,}1)$, $f(x_0+0{,}2)$, etc. et on s'arrête dès qu'on a trouvé une valeur x_1 telle que $f(x_1)$ et $f(x_1+0{,}1)$ sont de signes opposés (on a alors encadré α à 10^{-1} près) ;
- on poursuit le processus en balayant l'intervalle $[x_1\,;x_1+0{,}1]$ avec un pas de 0,01 pour obtenir un encadrement de α à 10^{-2} près. Et ainsi de suite...
- ...jusqu'à ce que l'on ait encadré α à 10^{-n} près.

1. **Version simple** « à pas constant » : on balaiera l'intervalle $[a\,;b]$ par pas d'amplitude constante, égale à 10^{-n}. Écrire une fonction `balayage1()` prenant 3 paramètres : `a`, `b` (les bornes de l'intervalle $[a\,;b]$) et `n` (la puissance de 10 dont l'inverse gouverne la précision souhaitée). La fonction renverra la moyenne arithmétique des valeurs x_n et x_n+10^{-n} telles que $\alpha \in [x_n\,;x_n+10^{-n}]$.
2. **Version optimisée** (cf. méthode décrite ci-avant, « à pas variable »). Écrire une fonction `balayage2()` avec les mêmes paramètres et la même valeur de retour que la fonction précédente. Mais ici, on commencera par trouver un intervalle contenant α inclus dans $[a\,;b]$ en balayant ce dernier avec un pas de 1 (ou de 0,1, ou de 0,01, ou... bref, le pas qui convient à la situation initiale) ; puis on affinera à partir de ce nouvel intervalle en répétant l'opération avec un pas 10 fois plus petit ; et ainsi de suite jusqu'à obtention de la précision souhaitée.

Aide : pour s'assurer que deux quantités sont de signes opposés, on peut tester si leur produit est négatif!

Validation : avec la fonction `f` définie comme dans l'exemple précédent par $f(x) = x^2-2$, on pourra utiliser l'assertion `assert balayage1(1,2,10) == 1.4142135623049998, "/!\ balayage()"`. On en déduit qu'une approximation de $\sqrt{2}$ à 10^{-10} près est 1,4142135623.

EXERCICE 15 ★★☆ — *Résoudre une équation par la méthode de dichotomie*

On suppose définie une fonction Python nommée `f`, qui prend un paramètre `x` et ne fait que renvoyer le résultat d'un calcul mathématique correspondant à la seule évaluation d'une expression dépendant de ce paramètre `x`. Par exemple : `def f(x): return x**2-2`. C'est le pendant informatique de la définition d'une fonction mathématique f par l'expression $f(x) = x^2-2$.

L'objectif est de trouver une valeur approchée d'une solution de l'équation $f(x) = 0$, à l'aide d'un algorithme dit de « **dichotomie** ». Il faut tout d'abord être certain que la fonction f est définie sur un intervalle $[a\,;b]$ qui vérifie les **deux** propriétés suivantes :

- $[a\,;b]$ contient une valeur α telle que $f(\alpha) = 0$;
- $f(a)$ et $f(b)$ sont de signes contraires. Autrement dit : $f(a)\times f(b) < 0$ (petite astuce bien pratique).

Pour obtenir un encadrement à 10^{-n} près de α, on divise par 2 à chaque étape la largeur de l'intervalle contenant α en conservant la « bonne moitié » de l'intervalle, c'est-à-dire celle qui contiendra α : pour l'identifier, on retient la moitié dont les images des bornes sont de signes opposés (cf. l'astuce précédente). On interrompt le processus dès que la largeur de l'intervalle est inférieure à 10^{-n}.

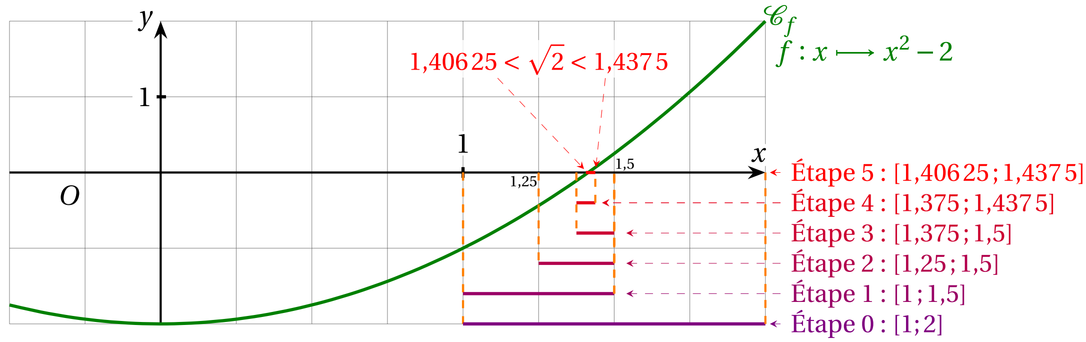
Mathématiquement, cette méthode repose sur la définition de deux suites (a_n) et (b_n), définissant les bornes inférieures et supérieures de l'intervalle, toujours plus réduit, dans lequel se trouve la solution de $f(x) = 0$:

$$a_0 = a \text{ et } b_0 = b \quad \text{puis} \quad \begin{cases} \text{si } f(a_n)\times f\left(\dfrac{a_n+b_n}{2}\right) < 0, \text{ alors } a_{n+1} = a_n \text{ et } b_{n+1} = \dfrac{a_n+b_n}{2}\,; \\ \text{sinon } a_{n+1} = \dfrac{a_n+b_n}{2} \text{ et } b_{n+1} = b_n\,. \end{cases}$$

Exemple : détaillons les premières étapes du processus avec la fonction f définie par $f(x) = x^2-2$, qui s'annule sur l'intervalle $[1;2]$, car $f(\sqrt{2}) = 0$ et $1 < \sqrt{2} < 2$. D'autre part, $f(1) = -1$ et $f(2) = 2$, donc $f(1)\times f(2) < 0$. Cette situation nous conduira donc à déterminer une approximation décimale de $\alpha = \sqrt{2}$.

1. On commence par calculer le milieu de l'intervalle $[1\,;2]$: c'est 1,5. A-t-on $\alpha \in [1\,;1{,}5]$ ou $\alpha \in [1{,}5\,;2]$?
   - $f(1) \times f(1{,}5) = -1 \times (1{,}5^2 - 2) = -1 \times (2{,}25 - 2) = -1 \times 0{,}25 = -0{,}25 < 0$.
   - On peut vérifier que $f(1{,}5) \times f(2) = 0{,}25 \times 2 = 0{,}5 > 0$.

   On retient donc le demi-intervalle $[1\,;1{,}5]$: $\alpha \in [1\,;1{,}5]$.



2. On itère le processus en calculant le milieu de $[1\,;1{,}5]$: c'est 1,25. A-t-on $\alpha \in [1\,;1{,}25]$ ou $\alpha \in [1{,}25\,;1{,}5]$?
   - $f(1) \times f(1{,}25) = -1 \times (1{,}25^2 - 2) = -1 \times (1{,}5625 - 2) = -1 \times (-0{,}4375) = 0{,}4375 > 0$.
   - On peut vérifier que $f(1{,}25) \times f(1{,}5) = -0{,}4375 \times 0{,}25 = -0{,}109375 < 0$.

   On retient donc le demi-intervalle $[1{,}25\,;1{,}5]$: $\alpha \in [1{,}25\,;1{,}5]$.
3. On continue jusqu'à ce que l'on ait trouvé un intervalle de largeur inférieure à 10^{-n} qui contienne $\sqrt{2}$.

Écrire une fonction `dichotomie()` prenant 3 paramètres : `a`, `b` (les bornes de l'intervalle $[a\,;b]$) et `n` (la puissance de 10 dont l'inverse gouverne la précision souhaitée). La fonction renverra la moyenne arithmétique des valeurs a_n et b_n telles que $\alpha \in [a_n\,;b_n]$ et $b_n - a_n < 10^{-n}$.

Validation : avec la fonction `f` définie comme précédemment par $f(x) = x^2 - 2$, on pourra utiliser l'assertion `assert dichotomie(1,2,6) == 1.4142136573791504, "/!\ dichotomie()"`.

EXERCICE 16 ★★☆ — *Mystère et boules de code*

1. Que fait la fonction `mystere1()` dont le code est donné ci-dessous ?

```
def mystere1(x):
    a = 0
    b = 1
    while b*b < x:
        a = b
        b = b + 1
    return a,b
```

2. Que fait la fonction `mystere2()` dont le code est donné ci-dessous ?

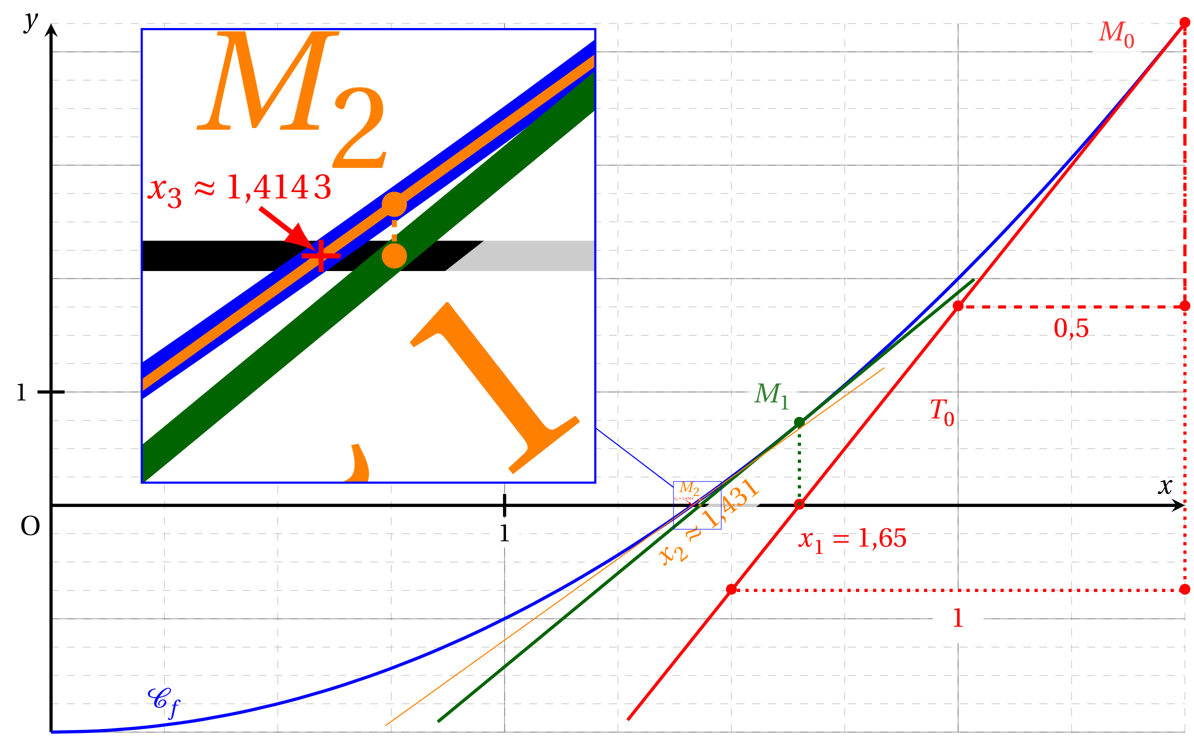
```
def mystere2(x, e):
    a, b = mystere1(x)
    while b-a > e:
        m = (a+b)/2
        if m**2 == x:
            a = b = m
        elif m**2 > x:
            b = m
        else:
            a = m
    return a,b
```

EXERCICE 17 ★★☆ — *Résoudre une équation par la méthode de Newton-Raphson*

Soit une fonction f définie sur un intervalle $[a\,;b]$, contenant une valeur α telle que $f(\alpha) = 0$. Pour déterminer une valeur approchée de α, Isaac Newton, l'un des précurseurs du calcul infinitésimal avec Leibniz, a eu une idée, passée à la postérité sous le nom de « Méthode de Newton » (même si d'autres grands noms des sciences y sont également associés) : aux alentours d'un point donné, une fonction f « ressemble » à sa tangente.

Exemple : la fonction f définie sur $\mathbb{R}$ par $f(x) = x^2 - 2$ admet $\sqrt{2}$ pour solution à l'équation $f(x) = 0$. Voyons concrètement comment on détermine une valeur approchée de $\sqrt{2}$ avec la méthode de Newton…

- On commence au point M_0 appartenant à la courbe $\mathscr{C}_f$ (qui représente la fonction f). L'abscisse x_0 de M_0 est choisie « pas trop loin » de la valeur α telle que $f(\alpha) = 0$: on choisit ici $x_0 = 2{,}5$.
- On détermine l'équation de la tangente T_0 à $\mathscr{C}_f$ en M_0, puis on cherche la valeur de x pour laquelle cette droite T_0 coupe l'axe (Ox) : on note x_1 cette abscisse, et M_1 le point de $\mathscr{C}_f$ correspondant.
- On répète le processus : on cherche l'équation de la tangente T_1 à $\mathscr{C}_f$ en M_1, puis la valeur de x pour laquelle la droite T_1 coupe l'axe des abscisses, qu'on note x_2. Et ainsi de suite…
- On arrête le processus lorsqu'on approche *suffisamment précisément* la solution de l'équation $f(x) = 0$.



Mathématiquement, cette méthode revient à définir par récurrence une suite (x_n) par :

$$x_0 \text{ est une valeur « proche » de } \alpha \quad \text{puis} \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \text{ pour } n \geqslant 1\,,$$

où f' désigne la fonction dérivée de f.

Écrire une fonction `newton()` prenant 2 paramètres : `a` (une valeur « proche » de la solution de l'équation $f(x) = 0$) et un entier `n`. La fonction renverra la valeur x_n obtenue après `n` itérations.

Aide : il faudra gérer le calcul de $f'(x)$. On peut soit définir une nouvelle fonction `df()` qui prend un paramètre `x` et renvoie le résultat du calcul de $f'(x)$ (ce qui implique de redéfinir manuellement `df()` à chaque fois que `f()` change de définition). On pourra aussi se contenter d'une estimation de $f'(x)$ à l'aide d'un taux d'accroissement $\frac{f(x+h)-f(x)}{h}$ établi pour une valeur suffisamment petite de h.

Validation : avec la fonction `f` définie comme précédemment par $f(x) = x^2 - 2$, et en utilisant une fonction auxilliaire pour la dérivée, définie par `def df(x): return 2*x`, on pourra utiliser l'assertion suivante :

`assert newton(2,6) == 1.414213562373095, "/!\ newton()"`.
Vous pourrez vérifier que l'approximation de $\sqrt{2}$ obtenue après seulement 6 itérations de l'algorithme de Newton est précise à moins de 3×10^{-16}, ce qui est tout bonnement excellent !

SUITES ET APPROXIMATIONS

EXERCICE 18 ★☆☆ — *Conjecture de Collatz / Syracuse*

La **conjecture de Collatz** est un énoncé mathématique dont on *ne* sait *pas* ***encore*** s'il est vrai : en voici son énoncé. Prenez un nombre entier positif, et appliquez lui le traitement suivant :

- s'il est pair, vous le divisez par 2 ;
- sinon, vous le multipliez par 3 et ajoutez 1.

Vous obtenez alors un nouveau nombre, sur lequel vous répétez la procédure. Et ainsi de suite... La conjecture s'énonce ainsi : quel que soit l'entier choisi au départ, on finira par obtenir 1. **Aucun mathématicien ne sait prouver sa véracité**, à l'heure actuelle (même si on la pense exacte). L'énoncé du problème est pourtant si simple qu'il est accessible aux élèves dès le milieu de l'école primaire !

Remarque : cette conjecture occupa tant les mathématiciens dans les années 1960 (en pleine guerre froide) qu'une plaisanterie courut selon laquelle ce problème faisait partie d'un complot soviétique visant à ralentir la recherche américaine.

1. Écrire une fonction `collatz()` qui prend en paramètre un entier `n` et retourne le booléen `True` si le processus s'achève (donc si on finit par atteindre la valeur 1). Notez que dans le cas contraire, on n'atteint pas la valeur 1 et le processus ne s'arrête pas : la fonction `collatz()` pourrait alors voir son exécution durer indéfiniment !

   Validation : tous les entiers testés jusqu'ici valident la conjecture de Collatz ! On ignore encore s'il en existe qui ne la valident pas (voir la question ci-dessous)...

2. Écrire une fonction `temps_vol_collatz()` qui prend en paramètre un entier `n` et retourne le nombre de fois où le processus est répété avant d'atteindre la valeur 1. On fera l'hypothèse que tout se passera bien (on a vérifié informatiquement que tous les entiers $n < 1{,}25 \times 2^{62}$ satisfont la conjecture) : en effet, si l'entier `n` ne valide pas la conjecture, le nombre de passage dans la boucle sera infini, et la fonction `nb_iter_collatz()` ne verra pas son exécution s'achever !

   Validation : `assert temps_vol_collatz(127) == 46, "/!\ temps_vol_collatz()"`

3. Écrire une fonction `hauteur_vol_collatz()` qui prend en paramètre un entier `n` et retourne la valeur maximale atteinte lors du processus, avant d'atteindre la valeur 1. On fera à nouveau l'hypothèse qu'il y a de bonnes chances que tout se passe bien.

   Validation : `assert hauteur_vol_collatz(127) == 4372, "/!\ hauteur_vol_collatz()"`

4. Écrire une fonction `vol_altitude_collatz()` qui prend en paramètre un entier `n` et retourne le « temps de vol en altitude » : c'est le plus petit numéro d'étape pour lequel le résultat intermédiaire est strictement inférieur à `n`.

   Validation : `assert vol_altitude_collatz(127) == 23, "/!\ vol_altitude_collatz()"`

5. **Spécial NSI :** écrire une fonction `temps_vol_max_collatz()` qui prend en paramètres deux entiers `m` et `n` tels que `m` < `n` et renvoie un **tuple** `(k ; temps_vol_collatz(k))`, où l'entier `k` ∈ `[m; n[` est tel que son `temps_vol_collatz(k)` est le plus grand de l'ensemble des temps de vols de tous les entiers compris entre `m` **inclus** et `n` **exclus** (à la mode Python).

Validation : `assert temps_vol_max_collatz(3, 100) == (97, 118), "/!\ temps_vol_max_collatz()"`

6. **Spécial NSI # 2 :** écrire une fonction `temps_vols_collatz()` qui prend en paramètre un entier `n`. Elle renverra un **dictionnaire** dont les clés seront les différents temps de vols *uniques*, chaque valeur associée étant un **tuple** rassemblant les entiers dont le temps de vol est la clé.
   Aide : on pourra utiliser la méthode `setdefault()` des objets dictionnaires.
   Validation : `assert temps_vols_collatz(10) == {0: (1,), 1: (2,), 7: (3,), 2: (4,)}, "/!\ temps_vols_collatz()"`

Bonus NSI : *diverses améliorations sont possibles. On peut alors chercher à estimer ce qu'elles apportent en matière d'efficacité dans le déroulé du processus.*

- *Collatz lui-même avait remarqué que, lorsqu'un nombre n était impair, le résultat du calcul $3n+1$ était nécessairement pair. De ce fait, on peut raccourcir d'une ligne le code de la fonction `collatz()` en supprimant la clause `else` : si `n` est pair, on peut le diviser par 2, sinon `n` prendra la valeur `3*n+1`, qu'on pourra systématiquement diviser par 2. On peut donc diviser par 2 dans tous les cas !*
- *John Conway a remarqué qu'il arrivait qu'on puisse enchaîner plusieurs divisions par 2. Il a donc proposé que la division par 2 soit effectuée jusqu'à ce qu'elle ne soit plus possible.*

Pour comparer l'efficacité de ces 3 approches (celle décrite en premier et les deux ci-dessus), vous pourrez mesurer le temps d'exécution de chaque appel de fonction sur un grand nombre de valeurs entières, et chronométrer la durée des calculs à l'aide du code suivant :

```
from time import time
t0 = time()       # On enregistre l'heure
...               # On appelle un grand nombre de fois une "fonction collatz"
print("Durée d'exécution :", time()-t0)      # Affichage de la durée des calculs
```

EXERCICE 19 ★☆☆ — *Paradoxe dichotomique* de *Zénon d'Élée*

Zénon (environ 490–430 av. J-C.) se tient à 8 mètres d'un arbre, tenant une pierre : il la lance vers l'arbre. Avant que le caillou n'atteigne l'arbre, il doit franchir la première moitié des 8 mètres. Il faut un certain temps, non nul, à cette pierre pour se déplacer sur cette distance. Ensuite, il lui reste encore 4 mètres à parcourir, dont elle accomplit d'abord la moitié, 2 mètres, ce qui lui prend un certain temps. Puis la pierre avance d'un mètre de plus, progresse après d'un demi-mètre et encore d'un quart, et ainsi de suite, *jusqu'à l'infini*, et à chaque fois avec un temps non nul. Zénon en conclut que la pierre ne pourra pas frapper l'arbre, puisqu'il faudrait pour cela que soit franchie une série infinie d'étapes, ce qui est impossible. Pourtant, l'expérience le prouve aisément : il n'y a guère de difficulté à faire en sorte qu'une pierre heurte un arbre situé à 8 m de distance ! D'où le paradoxe...

Explorons numériquement ce paradoxe, qui peut se modéliser à l'aide d'une suite définie par récurrence (d_n), dont chaque terme représentera la distance parcourue par la pierre en direction de l'arbre à chaque étape :

$$d_1 = 4 \quad \text{et} \quad d_{n+1} = d_n + \frac{4}{2^n}\,.$$

Écrire une fonction `zenon()` qui prend en paramètre un entier `n` et renvoie la valeur du terme d_n associé. Qu'observez-vous en augmentant progressivement les valeurs de n, de 1 à 100 ?

Validation : `assert zenon(20) == 7.999992370605469, "/!\ zenon()"`

Remarque : *une autre formule pour définir la suite (d_n), parfaitement équivalente, est $d_{n+1} = \frac{1}{2}d_n + 4$. Elle découle du raisonnement suivant : d_{n+1}, la distance parcourue par la flèche à l'étape $n+1$, est égale à la distance parcourue à l'étape précédente, d_n donc, augmentée de la moitié de la distance restant à parcourir, soit $\frac{1}{2}(8-d_n)$. Il n'y a plus qu'à simplifier...*

Bonus : *donner les 100 premières valeurs de la suite (d_n) avec une compréhension de liste.*

EXERCICE 20 ★☆☆ — *Héron, héron, petit, pas tapon!*

Comment calculer une approximation décimale de $\sqrt{a}$? La **méthode de HÉRON D'ALEXANDRIE** est un procédé connu depuis le 1er siècle de notre ère (voire avant) qui apporte une réponse à cette question. De nos jours, on la formalise à l'aide d'une suite (x_n) définie par récurrence de la façon suivante :

$$\text{on choisit une valeur strictement positivive } x_0, \text{ puis pour tout entier } n \geqslant 1, x_{n+1} = \frac{x_n + \dfrac{a}{x_n}}{2}.$$

Écrire une fonction `heron()` prenant en 1er paramètre l'entier a, en 2nd paramètre un entier $n \geqslant 1$, qui renverra l'approximation de $\sqrt{a}$ correspondant au terme de rang n de la « suite de HÉRON » définie précédemment.

VALIDATION : `assert heron(2, 4) == 1.4142135623746899, "/!\ heron()"`

REMARQUE : cette méthode est très efficace! 4 étapes suffisent à obtenir une valeur approchée de $\sqrt{2}$ à 2×10^{-12} près.

EXERCICE 21 ★☆☆ — π

Comment **calculer une valeur décimale approchant π** ? Le mathématicien écossais **James GREGORY** (vers 1671), puis l'immense polymathe **Gottfried Wilhelm LEIBNIZ** (vers 1674), démontrèrent et utilisèrent la formule suivante pour y parvenir :

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1} + \cdots \quad \text{d'où} \quad \pi = 4 \times \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1} + \cdots\right).$$

Remarque : *cette formule était en fait déjà connue vers 1400... mais pas en EUROPE! Le mathématicien indien **Madhava de SANGAMAGRAMA** l'a utilisée pour calculer les 11 premières décimales de π.*

Les points de suspension signifient que la somme ne s'arrête pas (jamais!) et qu'il faudrait la continuer *indéfiniment* pour calculer une valeur exacte de π (qui, de ce fait, ne possède pas de valeur décimale!). On peut alors calculer des valeurs approchées de plus en plus proches de la valeur réelle de π, en ajoutant de plus en plus de termes. On observe que cette formule peut être exprimée à l'aide des termes successifs d'une suite (p_n) définie par récurrence :

$$p_0 = 4 \quad \text{et} \quad p_{n+1} = p_n + 4 \times \frac{(-1)^n}{2n+1} \quad \text{pour } n \geqslant 0.$$

Écrire une fonction `pi_14_17()` qui prend en paramètre un entier `n` et renvoie la valeur approchée p_n de π.

VALIDATION : `assert pi_14_17(100000) == 3.1415826535897198 , "/!\ pi_14_17()"`

EXERCICE 22 ★★☆ — *Et π, c'est tout!*

Comment **calculer une valeur décimale approchant π** ? En 1593, la formule suivante est démontrée par le juriste et mathématicien français **François VIÈTE** :

$$\pi = 2 \times \frac{2}{\sqrt{2}} \times \frac{2}{\sqrt{2+\sqrt{2}}} \times \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \times \cdots$$

Les points de suspension signifient que la multiplication ne s'arrête pas (jamais!) et qu'il faudrait la continuer *indéfiniment* pour calculer une valeur exacte de π (qui, de ce fait, ne possède pas de valeur décimale!). On peut alors calculer des valeurs approchées de plus en plus proches de la valeur réelle de π, en multipliant de plus en plus de termes :

On observe que cette formule peut être exprimée à l'aide des termes successifs de **deux** suites (u_n) et (p_n) définies par récurrence (la 1re est une suite auxilliaire, utilisée dans la définition de la 2nde pour approcher π) :

$$u_0 = 0 \quad \text{et} \quad u_{n+1} = \sqrt{2+u_n} \ \text{ pour } n \geqslant 0 \qquad \text{puis} \qquad p_0 = 2 \quad \text{et} \quad p_{n+1} = \frac{2p_n}{u_{n+1}} \ \text{ pour } n \geqslant 0.$$

Écrire une fonction `pi_16()` qui prend en paramètre un entier `n` et renvoie la valeur approchée p_n de π.

Validation : `assert pi_16(20) == 3.1415926535850938, "/!\ pi_16()"`

EXERCICE 23 ★★☆ — *π, la revanche !*

Comment **calculer une valeur décimale approchant π** ? Leonhard Euler démontre au 18e siècle la formule suivante :

$$\pi = 2 \times \left(1 + \frac{1}{3} + \frac{1\times 2}{3\times 5} + \frac{1\times 2\times 3}{3\times 5\times 7} + \frac{1\times 2\times 3\times 4}{3\times 5\times 7\times 9} + \cdots\right)$$

Les points de suspension signifient que la somme ne s'arrête pas (jamais !) et qu'il faudrait la continuer *indéfiniment* pour calculer une valeur exacte de π (qui, de ce fait, ne possède pas de valeur décimale !). On peut alors calculer des valeurs approchées de plus en plus proches de la valeur réelle de π, en ajoutant de plus en plus de termes dans la somme entre parenthèses :

$$p_0 = 2\times 1 = 2 \qquad p_1 = 2\times\left(1+\frac{1}{3}\right) = \frac{8}{3} \approx 2{,}667 \qquad p_2 = 2\times\left(1+\frac{1}{3}+\frac{1\times 2}{3\times 5}\right) = \frac{44}{15} \approx 2{,}933$$

$$p_3 = 2\times\left(1+\frac{1}{3}+\frac{1\times 2}{3\times 5}+\frac{1\times 2\times 3}{3\times 5\times 7}\right) = \frac{64}{21} \approx 3{,}048 \qquad p_4 = 2\times\left(1+\frac{1}{3}+\frac{1\times 2}{3\times 5}+\frac{1\times 2\times 3}{3\times 5\times 7}+\frac{1\times 2\times 3\times 4}{3\times 5\times 7\times 9}\right) \approx 3{,}098$$

Remarque : *à l'aide d'une formule semblable, Euler aurait réussi à calculer les 20 premières décimales de π en moins d'une heure. Songez qu'il aura fallu plus de 20 ans de calcul au mathématicien amateur anglais William Shanks pour achever le calcul en 1873 de 707 décimales de π… dont seules les 527 premières étaient exactes !*

On observe que cette formule peut être exprimée à l'aide des termes successifs de **deux** suites (u_n) et (p_n) définies par récurrence (la 1re est une suite auxilliaire, utilisée dans la définition de la 2nde pour approcher π) :

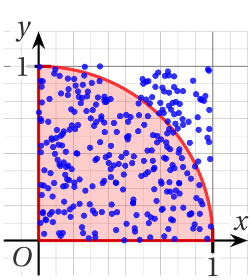
$$u_1 = \frac{1}{3} \quad \text{et} \quad u_{n+1} = u_n \times \frac{n+1}{2n+3} \ \text{ pour } n \geqslant 0 \qquad \text{puis} \qquad p_0 = 2 \quad \text{et} \quad p_{n+1} = p_n + 2\times u_{n+1} \ \text{ pour } n \geqslant 0.$$

Écrire une fonction `pi_18()` qui prend en paramètre un entier `n` et renvoie la valeur approchée p_n de π.

Validation : `assert pi_18(20) == 3.1415922987403384, "/!\ pi_18()"`

EXERCICE 24 ★☆☆ — *Le fils du retour de la vengeance de π !*

Les **méthodes de Monte-Carlo** permettent d'estimer la valeur approchée de grandeurs en utilisant le hasard : on va s'en servir pour approcher une fois de plus la valeur de π.



Le schéma ci-contre montre un quart du disque de centre O et de rayon 1, qui a donc une surface de $\frac{\pi}{4}$. L'idée est de tirer *aléatoirement* les coordonnées d'un point $M(x\,;y)$, avec $0 \leqslant x \leqslant 1$ et $0 \leqslant y \leqslant 1$. M sera dans le quart de disque lorsque $x^2+y^2 \leqslant 1$, d'après le théorème de Pythagore. La probabilité que le point M appartienne au quart de disque sera donc égale à $\frac{\pi}{4}$ (quotient de la surface du quart de disque par celle du carré dans lequel il est inscrit, et qui vaut 1).

En tirant au hasard un grand nombre de positions pour M, le rapport du nombre de points dans le disque au nombre de tirages donnera donc une approximation de $\frac{\pi}{4}$!

Écrire une fonction `montecarlo_pi()` qui prend en paramètre un entier `n` et renvoie la valeur approchée de π obtenue après `n` tirages.

VALIDATION : la nature même de cette méthode fait qu'il est **impossible** de fournir une assertion qui la valide ! En effet, les points M sont tirés au hasard, il est donc impossible de prévoir combien, parmi les tirages effecutés, appartiendront au quart de disque...

REMARQUE : si cette méthode est très inefficace pour approcher π, elle présente en revanche des avantages certains dans des situations plus complexes.

EXERCICE 25 — NSI ★☆☆

Mais c'est de π en π !

Comparer l'efficacité des différentes méthodes permettant de calculer des approximations décimales de π. Vous pourrez mesurer le temps d'exécution de chaque fonction pour obtenir un même nombre de décimales de π exactes (commencer par un petit nombre !), et chronométrer la durée des calculs à l'aide du code suivant :

```
from time import time
t0 = time()      # On enregistre l'heure
...              # On appelle un grand nombre de fois une "fonction collatz"
print("Durée d'exécution :", time()-t0)      # Affichage de la durée des calculs
```

EXERCICE 26 ★★☆

Une histoire de lapins pas crétine...

La **suite de FIBONACCI** est une suite d'entiers (F_n) vérifiant les propriétés suivantes :

$$F_0 = F_1 = 1 \quad \text{et} \quad F_n = F_{n-1} + F_{n-2} \text{ pour } n \geqslant 2\,.$$

Historiquement, on la voit apparaîte en EUROPE dans un problème récréatif posé par Leonardo FIBONACCI en 1202 (mais un mathématicien Indien avait semble-t-il déjà établi cette formule dès le 8e siècle).

1. Écrire une fonction `fibonacci()` qui prend en paramètre un entier `n` et renvoie la valeur de F_n.
   VALIDATION : `assert fibonacci(16) == 987, "/!\ fibonacci()"`
2. Écrire une fonction `liste_fibo()` qui prend en paramètre un entier `n` et renvoie une liste PYTHON contenant toutes les valeurs de F_k pour $k \leqslant$ `n`.
   VALIDATION : `assert liste_fibo(4) == [1, 1, 2, 3, 5], "/!\ liste_fibo()"`

***BONUS NSI :** cet exercice est un grand classique dans l'introduction aux algorithmes récursifs. Étudier en autonomie cette portion de l'article de Wikipedia consacré à la suite de FIBONACCI. Puis mettre en place un mécanisme de mémoïsation comme décrit dans cet autre article de Wikipedia.*

EXERCICE 27 ★☆☆

« La parole est d'argent, le silence est d'or » — un nombre aussi !

Le **nombre d'or** est $\varphi = \dfrac{1+\sqrt{5}}{2}$. C'est le rapport qui existe *nécessairement* entre deux longueurs a et b telles que $a > b$ et qui vérifient en sus :

$$\frac{a+b}{a} = \frac{a}{b}$$ (le quotient de la somme des longueurs par la plus grande égale le quotient de la plus grande longueur par la plus petite).

Ce rapport particulier est souvent appelé « divine proportion ». Si sa présence dans les constructions humaines antique est au minimum sujette à controverse, on peut pourtant l'observer dans la nature (notamment dans les fleurs de tournesol, les écailles des pommes de pin, le pavage de Penrose, modèle plausible pour les quasi-cristaux [3]). Au 20e siècle, il a même été utilisé par certains pour défendre des positions racistes 😱 !

Il existe un lien étroit entre le nombre d'or et la suite de Fibonacci (voir l'exercice précédent) : le rapport entre deux termes consécutifs de cette suite se rapproche du nombre d'or lorsqu'on augmente l'indice de ces termes. En langage mathématique, si l'on note (F_n) la suite de Fibonacci :

$$\varphi = \lim_{n\to\infty} \frac{F_{n+1}}{F_n}\,.$$

On peut également démontrer que le terme général de la suite de Fibonacci est $F_n = \frac{1}{\sqrt{5}}\left(\varphi^n - \left(-\frac{1}{\varphi}\right)^n\right)$.

1. La suite récurrente (a_n) définie par $a_0 = 2$ et $a_{n+1} = 1 + \frac{1}{a_n}$ donne des valeurs qui se rapprochent inexorablement de φ lorsque n augmente : **on dit que la suite (a_n) converge vers φ.**
   Écrire une fonction `nbdor1()` qui prend en paramètre un entier `n` et renvoie la valeur de a_n.
   Validation : `assert nbdor1(8) == 1.6181818181818182, "/!\ nbdor1()"`
2. La suite récurrente (b_n) définie par $b_0 = 2$ et $b_{n+1} = \sqrt{b_n + 1}$ converge également vers φ.
   Écrire une fonction `nbdor2()` qui prend en paramètre un entier `n` et renvoie la valeur de b_n.
   Validation : `assert nbdor2(8) == 1.618064196086926, "/!\ nbdor2()"`
3. La suite récurrente (c_n) définie par $c_0 = 2$ et $c_{n+1} = \frac{c_n^2 + 1}{2c_n - 1}$ converge *elle aussi* vers φ.
   Écrire une fonction `nbdor3()` qui prend en paramètre un entier `n` et renvoie la valeur de c_n.
   Validation : `assert nbdor3(4) == 1.618033988749989, "/!\ nbdor3()"`
4. Réutiliser la fonction `liste_fibo()` créée à l'exercice précédent pour coder une fonction `nbdor4()` qui prend en paramètre un entier `n` $\geqslant 1$ et renvoie la valeur de $\frac{F_n}{F_{n-1}}$ où F_n désigne le terme d'indice n de la suite de Fibonacci.
   Validation : `assert nbdor4(8) == 1.6153846153846154, "/!\ nbdor4()"`
5. Comparer l'efficacité de ces différentes méthodes pour approcher le nombre d'or.

EXERCICE 28 — Tale ★½☆ — … *car le logarithme ne paie rien* 😊 *!*

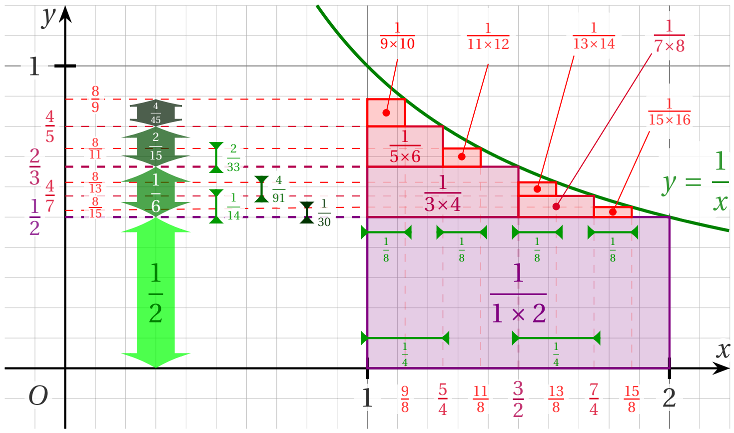
En mathématiques, on définit $\ln 2$ (le *logarithme népérien de 2*) comme étant l'aire [4] délimitée par l'axe (Ox), les droites d'équations $x = 1$ et $x = 2$ et la courbe d'équation $y = \frac{1}{x}$ (c'est une portion d'une des deux branches de l'hyperbole correspondant à la fonction inverse). Le schéma ci-dessous illustre la façon dont le linguiste et mathématicien anglais du 17e siècle William Brouncker trouve une valeur approchée de $\ln 2$.

L'idée est d'approcher l'aire recherchée en ajoutant des rectangles de largeur divisée par 2 à chaque étape, et de hauteur adaptée de sorte que le coin supérieur droit de chaque rectangle soit situé sur l'hyperbole.

3. Citons Wikipedia : « *les pavages de Penrose ne seraient restés qu'un joli divertissement mathématique si n'avaient été découverts, en 1984, des matériaux présentant une structure fortement ordonnée comme celle des cristaux mais non périodique : les quasi-cristaux. Les pavages non périodiques, en particulier ceux de Penrose, s'avérèrent alors un modèle plausible de ces étranges matériaux. Cette découverte illustra à nouveau ce que Roger Penrose lui-même avait déjà remarqué en 1973, à propos d'un sujet de relativité générale :* "on ne sait jamais vraiment quand on perd son temps" ».

4. En mathématiques, ces calculs d'aires relèvent de ce qu'on appelle le *calcul intégral*, qui est l'une des deux branches du *calcul infinitésimal* avec le *calcul différentiel* — ce dernier menant à la notion de *fonction dérivée*. Le calcul intégral implique souvent l'opération réciproque de la dérivation : le calcul des *primitives*.

$$\frac{1}{4}\times\frac{1}{14}=\frac{1}{56}=\frac{1}{7\times 8}$$
$$\frac{1}{4}\times\frac{2}{15}=\frac{1}{30}=\frac{1}{5\times 6}$$
$$\frac{1}{2}\times\frac{1}{6}=\frac{1}{12}=\frac{1}{3\times 4}$$
$$1\times\frac{1}{2}=\frac{1}{1\times 2}$$



$$\frac{1}{8}\times\frac{1}{30}=\frac{1}{240}=\frac{1}{15\times 16}$$
$$\frac{1}{8}\times\frac{4}{91}=\frac{1}{182}=\frac{1}{13\times 14}$$
$$\frac{1}{8}\times\frac{2}{33}=\frac{1}{132}=\frac{1}{11\times 12}$$
$$\frac{1}{8}\times\frac{4}{45}=\frac{1}{90}=\frac{1}{9\times 10}$$

Mathématiquement, cela revient à définir une suite $(u_n)_{n\geqslant 1}$ *explicitement* par $u_n=\sum_{k=1}^{n}\frac{1}{(2k-1)2k}$.

1. Établir une définition par récurrence de la suite $(u_n)_{n\geqslant 1}$: $u_1=\ldots$ et $u_{n+1}=u_n+\ldots$.
2. Compléter le code ci-contre et expérimentez-le.
3. Écrire une fonction `lst_va_ln2()` qui prend en paramètre un entier `p` et renvoie sous forme de liste PYTHON les termes de la suite $(u_n)_{n\geqslant 1}$ tels que $\ln 2-u_n>10^{-p}$.

```python
def brouncker(n):
    assert n > 0, "n doit être > 0"
    u = 0
    for k in range(...):
        u = ...
    return u
```

EXERCICE 29 ★★☆ — euuuu...

Le nombre e désigne la base du logarithme népérien (ou naturel). Il a été découvert à la fin du 17e siècle par plusieurs grands noms des sciences (Gottfried LEIBNIZ, Christian HUYGENS, Jacob BERNOULLI).

1. La suite $(u_n)_{n\geqslant 1}$ de terme général $u_n=\left(1+\frac{1}{n}\right)^n$ se rapproche du nombre e lorsque n devient grand.
   Écrire une fonction `e1()` qui prend un paramètre `n` et renvoie le terme de rang `n` de la suite $(u_n)_{n\geqslant 1}$.
   VALIDATION : `assert e1(100000) == 2.7182682371922975, "/!\ e1()"`
2. En mathématiques, on prouve l'égalité :
$$e=1+\frac{1}{1}+\frac{1}{1\times 2}+\frac{1}{1\times 2\times 3}+\frac{1}{1\times 2\times 3\times 4}+\cdots=\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\cdots$$
   REMARQUE : *la notation $n!$ désigne la factorielle de l'entier n. C'est par définition le produit de tous les entiers inférieurs ou égaux à n : $n!=1\times 2\times 3\times\cdots\times(n-1)\times n$. On lira $4!$ « factorielle 4 ». Notons que l'on a $0!=1$!*
   On peut donc approcher le nombre e au moyen de deux suites récurrentes (a_n) et (b_n) définies par :
$$a_0=1\quad\text{et}\quad a_{n+1}=(n+1)\times a_n\qquad\text{puis}\qquad b_0=1\quad\text{et}\quad b_{n+1}=b_n+\frac{1}{a_{n+1}}.$$
   Écrire une fonction `e2()` qui prend un paramètre `n` et renvoie le terme de rang `n` de la suite (b_n).
   VALIDATION : `assert e2(8) == 2.7182539682539684, "/!\ e2()"`

REMARQUE : fin 2020, l'approximation décimale de e compte plus de 31 000 milliards de décimales (David CHRISTIE dut faire tourner un ordinateur durant près de 54 jours pour obtenir ce résultat impressionnant !).

EXERCICE 30 — Tale ★★★ — *Coefficients binomiaux*

En mathématiques, le coefficient binomial $\binom{n}{k}$ (à prononcer « k parmi n ») donne, pour un ensemble comptant n éléments ($n \in \mathbb{N}$), le nombre de sous-ensembles à k éléments ($k \in \mathbb{N}$ vérifiant $k \leqslant n$). Lorsqu'on répète n fois à l'identique une expérience aléatoire ayant 2 issues possibles (traditionnellement appelées succès et échec), $\binom{n}{k}$ donne le nombre de façons *différentes* d'obtenir k succès au cours des n répétitions (on représente cette situation à l'aide d'un arbre probabilisé appelé schéma de BERNOULLI).

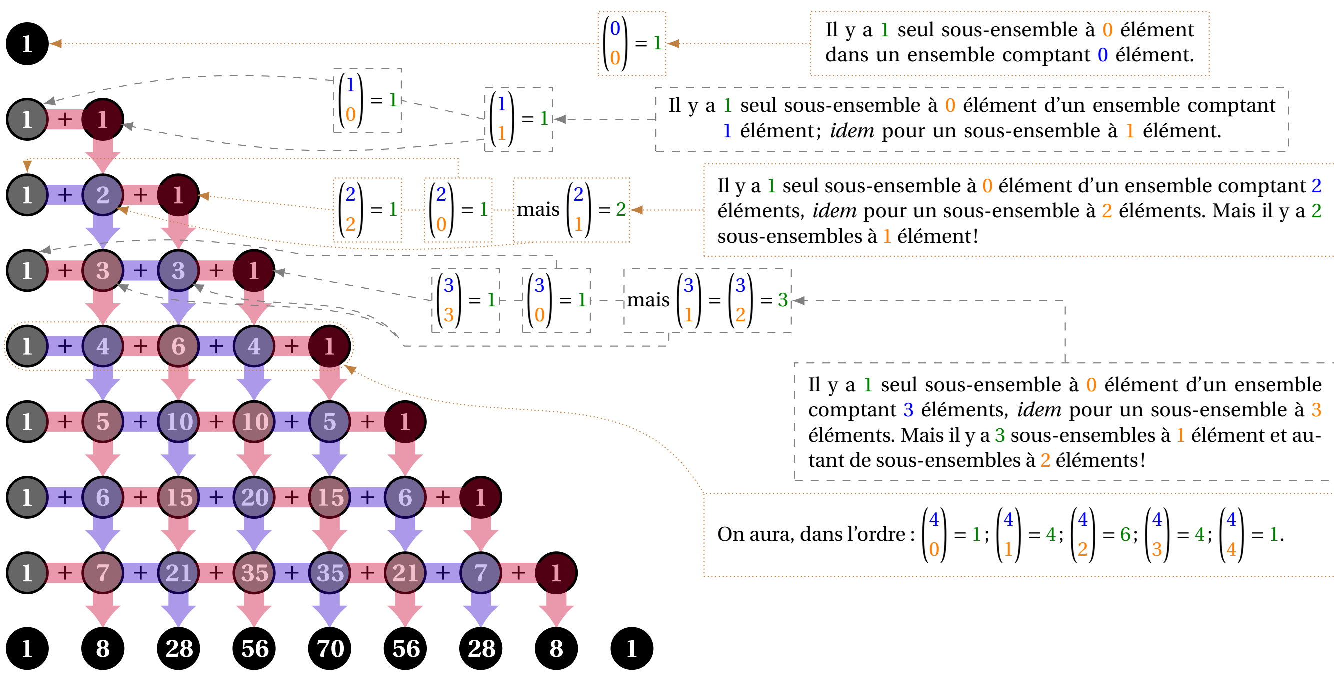
Une formule explicite, impliquant la notion de *factorielle* (cf. exercice précédent), en permet le calcul direct :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}\,.$$

Cette formule est néanmoins malpratique à utiliser dans un contexte de calcul numérique, en raison de la taille des nombres impliqués (qui induirait des erreurs en raison des approximations nécessaires). On peut utiliser une autre technique de calcul, basée sur la **relation de PASCAL** :

$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}$$

Cette relation préside à la création du triangle de PASCAL (son appellation occidentale, il porte des noms différents dans d'autres régions du monde) :



Écrire une fonction `binom()` qui prend en paramètres les entiers `n` puis `k` et renvoie la valeur de $\binom{n}{k}$. Cette fonction exploitera *impérativement* la relation de PASCAL.

AIDE : la relation de PASCAL implique qu'on n'a besoin, pour connaître les coefficients binomiaux d'une ligne, que de connaître les coefficients binomiaux de la ligne *précédente*. Mieux : pour calculer $\binom{n}{k}$, on n'a même besoin que de connaître les $k+1$ *premiers* coefficients binomiaux de la ligne précédente. On utilisera donc deux variables `ligne` et `ligne_suivante` désignant des listes, convenablement initialisées; chaque boucle permettant de déterminer une nouvelle (portion de) ligne du triangle de PASCAL se terminera par l'instruction `ligne = ligne_suivante[:]` (cette syntaxe particulière est *indispensable* : elle copie le *contenu* de la variable `liste_suivante`; l'instruction `ligne = ligne_suivante` aurait un effet tout autre : les deux variables désigneraient la *même* liste, ce qui aurait des conséquences pénibles!)

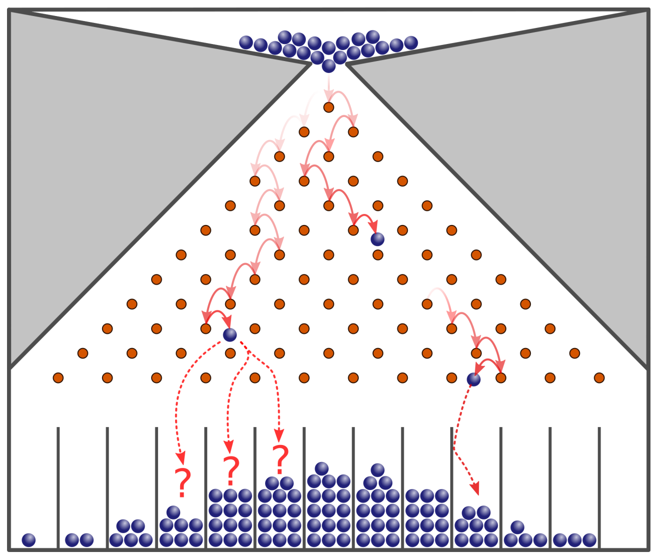
REMARQUES :

- *il existe un lien entre les coefficients binomiaux, le triangle de PASCAL, le jeu dit des « Tours de Hanoï » et la fractale nommée « Triangle de Sierpińksi ». On trouve sur Internet une excellente vidéo de Benoît RITTAUD qui aborde ces sujets : http://video.math.cnrs.fr/la-tour-de-hanoi/.*
- *il existe aussi un lien entre le triangle de PASCAL et la suite de FIBONACCI! Les sommes des coefficients alignés sur les « diagonales ascendantes » forment les termes de la suite de FIBONACCI.*

EXERCICE 31 — Tale ★★★ — *Simulation d'une **planche de GALTON***

La planche de GALTON est un dispositif inventé par sir Francis GALTON : un réservoir accueillant des billes est situé au dessus d'un ensemble de clous plantés de telle sorte qu'une bille lâchée sur la planche rebondit sur les clous, et passe soit à droite soit à gauche de chaque clou rencontré. En bas de la structure, les billes sont rassemblées dans divers compartiments.

Chaque compartiment correspond donc à une issue d'une « expérience binomiale » (répétition d'une expérience de BERNOULLI). On observe que la répartition des billes dans les compartiments correspond à une courbe de GAUSS (on dit aussi « courbe en cloche »). Et cela est d'autant plus vrai que le nombre de rangées augmente : on dit que « la loi binomiale converge vers la loi normale ».

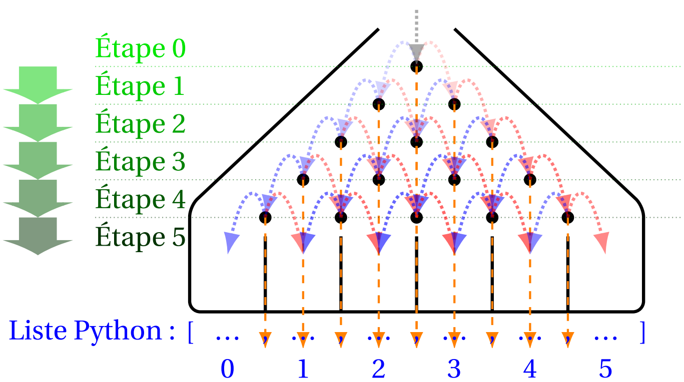


Écrire une fonction `galton()` prenant 2 paramètres entiers non nuls : dans l'ordre, le nombre `p` de rangées de clous, puis le nombre `n` de billes. Elle renverra une liste PYTHON contenant les fréquences d'apparition des billes dans chaque compartiment à l'issue de l'expérience.

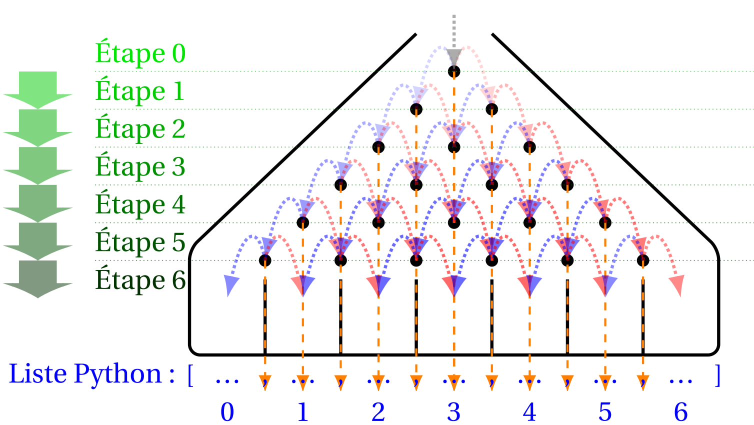
AIDE : c'est un exercice difficile. Voici plusieurs éléments susceptibles de vous aider, quitte à vous guider. Ignorez-les si vous êtes friands de défis à relever ☺!

1. Le paramètre `p` gouverne le nombre de *rangées* de clous, par suite il y aura `p+1` compartiments. La liste mémorisant des résultats sera donc de longueur `p+1`. Vous la désignerez par la variable `res`, et elle aura donc `p+1` emplacements numérotés de `0` à `p`.
2. Une façon de transposer l'expérience de GALTON dans le monde numérique consiste à se dire que la position initiale d'une bille (au début du processus, c'est-à-dire « à l'étape 0 ») est « au milieu de la liste », à l'emplacement numéroté `p/2`. **Problème :** `p/2` n'est pas forcément entier (tout dépend de la parité de `p` : s'il est impair, `p/2` est un nombre décimal!). Mais ça n'est en fait pas vraiment un souci (voir schémas ci-dessous). En effet :
   - chaque évolution d'une bille dans la planche de GALTON peut être vue comme un déplacement d'une demi-position vers la gauche ou vers la droite (en PYTHON, on pourra utiliser l'instruction `choice([-.5, .5]));`

- si p est impair, on aura un nombre impair de déplacements dans la grille, ce qui se traduira par la somme d'un nombre impair de $-\frac{1}{2}$ ou de $\frac{1}{2}$. Vous pourrez vous convaincre que cette somme aura alors une partie décimale valant 0,5 qui, combinée à une position initiale ayant également une partie décimale valant 0,5, donnera systématiquement, au final, une position entière, qui sera en plus obligatoirement comprise entre `0` et `p`. On n'aura donc ainsi **aucun** souci à l'exécution, à la fin du processus, d'une instruction telle que `res[position] = res[position]+1` qui ajoutera une bille à l'emplacement de la liste qui convient !
- si p est pair, un raisonnement analogue permet de conclure qu'il n'y aura pas davantage de souci.



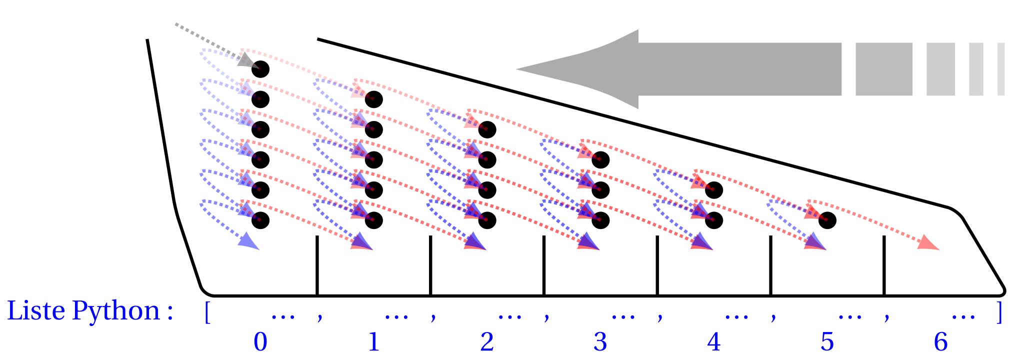
Cette planche de Galton à 5 rangées de clous comporte 6 réceptacles, qu'on assimile à une liste Python de 6 emplacements (numérotés de 0 à 6). À l'étape 0, une bille est lâchée : relativement à la liste Python, on peut considérer que sa position est $\frac{5}{2} = 2{,}5$. À l'étape 1, elle rebondit sur un clou, ce qui l'amène à se diriger vers la gauche ou vers la droite : elle évolue donc soit vers la position no 2 (car $2 = 2{,}5 - 0{,}5$) soit vers la position no 3 (car $3 = 2{,}5 + 0{,}5$). Et ainsi de suite…



Cette planche de Galton à 6 rangées de clous comporte 7 réceptacles, qu'on assimile à une liste Python de 7 emplacements (numérotés de 0 à 7). À l'étape 0, une bille est lâchée : relativement à la liste Python, on peut considérer que sa position est $\frac{6}{2} = 3$. À l'étape 1, elle rebondit sur un clou, ce qui l'amène à se diriger vers la gauche ou vers la droite : elle évolue donc soit vers la « position no 2,5 » (car $2{,}5 = 3 - 0{,}5$) soit vers la « position no 3,5 » (car $3{,}5 = 3 + 0{,}5$). Et ainsi de suite…

3. Une autre façon de modéliser numériquement l'expérience de Galton consiste à imaginer que les billes sont initialement toutes « **au <u>début</u> de la liste** », à l'emplacement numéroté `0`. On considérera alors qu'à chaque étape, les deux possibilités d'évolution d'une bille sont :
   - ne pas bouger (au lieu de rebondir vers la gauche) ;
   - se déplacer d'un cran vers la droite.

   Cela revient à ne compter sur p évolutions que les d évolutions vers la droite. Visuellement, la progression suivrait alors une grille en forme de triangle de Pascal (cf. exercice précédent). On peut aussi se figurer les choses en imaginant déformer « par cisaillement » la planche de Galton (la base restant fixe) :



Validation : la nature même de cette expérience fait qu'il est **impossible** de fournir une assertion qui valide la fonction `galton()` ! En effet, le chemin de chaque bille est déterminé au hasard, il est donc impossible de prévoir la fréquence d'apparition des billes dans chaque compartiment...

Remarque : *avec notre regard du 21e siècle, on dira pudiquement de Francis Galton que c'est un personnage qui « sent le soufre » ! En effet, c'est un des pères de l'eugénisme (doctrine visant à sélectionner les « meilleurs » individus d'une population et à éliminer les autres) : il a contribué à la recherche de moyens de sélectionner systématiquement et scientifiquement « l'élite » de l'humanité (ou plutôt du Royaume-Uni).*

EXERCICE 32 — Tale NSI ★★★ — *Loi géométrique*

Dans le domaine des probabilités, la loi géométrique est une loi de probabilité discrète impliquant des épreuves de Bernoulli. Rappels :

- une *loi de probabilité* décrit le comportement d'un phénomène aléatoire, c'est-à-dire dont la réalisation dépend du hasard (historiquement, l'étude des jeux de hasard a ouvert ce champ des mathématiques) ;
- un phénomène est qualifié de *discret* lorsque le nombre de résultats possibles est fini (ou au plus *dénombrable*, ce qui signifie que ses résultats sont « numérotables » à l'aide d'entiers, quand bien même il faudrait faire appel à l'infinité d'entiers de l'ensemble $\mathbb{N}$ pour y parvenir).
  Remarque : *certaines situations amènent à rencontrer des lois de probabilité ayant une infinité* non dénombrable *d'issues. Ainsi, lorsqu'on répète* indéfiniment *un jeu de pile ou face, la distribution des fréquences d'apparition de la face pile s'approche d'une courbe dite « courbe de Gauss » ou « courbe en cloche », associée à la loi normale (cf. l'exercice précédent et http://youtu.be/LGfGZmi3mYY?t=150).*
- une épreuve de Bernoulli de paramètre p (nombre réel compris entre 0 et 1) est une expérience aléatoire (c'est-à-dire soumise au hasard) comportant deux issues, qu'on appelle par convention *succès* (associé à la probabilité p) et *échec* (associé à la probabilité $q = 1 - p$).

On peut définir la loi géométrique de deux façons différentes, qui reposent néanmoins sur la notion de schéma de Bernoulli (c'est-à-dire sur la loi binomiale) qui consiste à répéter n fois, de façon indépendante, une expérience ayant deux issues — qu'on qualifie simplement de « succès » et « échec », chaque succès ayant une probabilité p de survenir.

Dans une loi binomiale, on compte le nombre de succès apparus au cours des n répétitions de l'expérience, et on en déduit les probabilités d'avoir k succès (k désignant n'importe quel entier compris entre 0 et n).

Dans la loi géométrique :

- soit on compte le nombre d'épreuves de Bernoulli nécessaires pour obtenir le **premier succès** (ce nombre k est alors un entier valant au moins 1), et on calcule les probabilités associées ;
- soit on compte le nombre d'échecs avant l'apparition du **premier succès** (qui vaut au moins 0).

Ces deux définitions sont naturellement équivalentes.

Remarque : *lorsqu'on s'intéresse à la loi géométrique, le nombre n d'épreuves de Bernoulli à réaliser n'est pas connu à l'avance : il peut être aussi élevé que nécessaire.*

Soit X une variable aléatoire suivant la loi géométrique : les valeurs qu'elle peut prendre sont 1, 2, 3, etc. On montre alors que la probabilité que $X = k$ (pour $k = 1, 2, 3, \ldots$) est $P(X = k) = q^{k-1}p$.

Écrire une fonction `loi_geom()` prenant 2 paramètres : un décimal `p` $\in]0;1[$, donnant la probabilité d'un succès dans l'expérience *élémentaire* qui sera répétée jusqu'à ce qu'un succès advienne ; et un entier `n` indiquant le nombre de répétitions de l'expérience. Cette fonction renverra deux listes Python :

- **la 1re donnera les différents nombres de répétitions de l'expérience *élémentaire* rangés par ordre croissant ;**
- **la 2nde contiendra les fréquences correspondantes.**

EXEMPLE : prenons p = 0,5 (l'expérience élémentaire correspond au lancer d'une pièce de monnaie équilibrée) et n = 1 000. Alors `ks, fs = loi_geom(0.5, 1000)` pourra donner des résultats semblables à :

- `[1, 2, 3, 4, 5, 6, 7, 8, 10]` comme 1re liste (ici désignée par `ks`);
- `[0.497, 0.239, 0.133, 0.062, 0.035, 0.018, 0.009, 0.006, 0.001]` comme 2nde liste (ici mémorisée *via* la variable `fs`).

Cela signifie que les simulations couronnée de succès dès la 1re tentative sont au nombre de 497 (soit une fréquence d'apparition de 0,497) ; que 239 simulations ont nécessité 2 tentatives avant d'obtenir un succès (soit une fréquence d'apparition de 0,239) ; que la fréquence d'apparition des simulations ayant nécessité 3 tentatives avant d'obtenir un succès est 0,133 (soit 133 simulations sur les 1 000 effectuées) ; etc.

AIDE : il faudra recourir à un dictionnaire PYTHON, car on ne peut pas savoir à l'avance quels seront les résultats de chaque simulation (on ne peut anticiper sur le nombre de répétitions de l'expérience *élémentaire* avant qu'elle ne se conclue par un succès) !

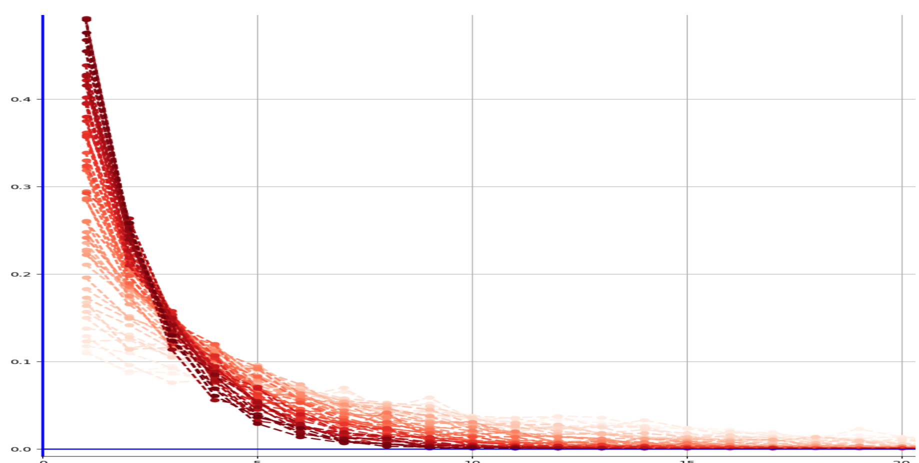
VALIDATION : la nature même de cette expérience fait qu'il est **impossible** de fournir une assertion qui valide la fonction `loi_geom()` ! Le hasard, toujours le hasard...2

BONUS NSI :

1. *vous pourrez afficher vos résultats avec* `matplotlib`*;*
2. *vous pourrez superposer plusieurs distributions de fréquences pour différentes valeurs de* `p`*. Il vous faudra utiliser des fonctionnalités additionnelles de* `matplotlib` *pour un rendu agréable :*

```
import matplotlib as mpl
norm = mpl.colors.Normalize(vmin=.1, vmax=pmax)
cmap = mpl.cm.ScalarMappable(norm=norm, cmap=mpl.cm.Reds)
...
while p ... :
    ks, fs = loi_geom1(p, n)
    plt.plot(fs, fs, "o--", linewidth=2, color=cmap.to_rgba(p))
```

Voici ce que vous pourriez alors obtenir :

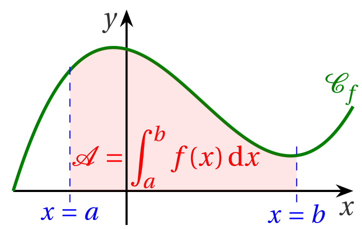


REMARQUE : *le problème du « collectionneur de vignette »* (coupon collector *en anglais) est un exemple de situation impliquant la loi géométrique. Le voici résumé : un collectionneur cherche à réunir N objets (images différentes pour compléter un album — footballeurs, personnages de fiction, etc. — , cartes d'un jeu quelconque, ...). Problème : à l'achat, le numéro de la vignette est inconnu (l'objet convoité est, par exemple, glissé au hasard dans des paquets de céréales). La question est : « combien faut-il faire d'achats pour avoir la collection complète ? » L'étude de ce problème et de ses généralisations* *trouve de nombreuses applications « dans la vraie vie » (en ingénierie des télécommunications — détection des attaques par déni de service ; en électronique, avec le problème des défauts de cache dans les processeurs ; en biologie, où il sert à estimer le nombre d'espèces animales). Ce problème était déjà mentionné en 1812 par Pierre-Simon de Laplace dans Théorie analytique des probabilités (page 195).*

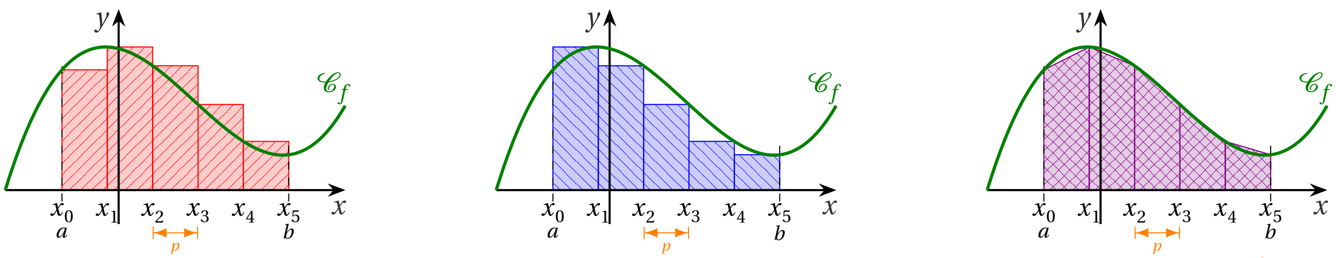
EXERCICE 33 — Tale ★★☆ — *On se fait l'intégrale ?*

L'ingénierie repose largement sur les outils mathématiques essentiels que sont les calculs de longueur, d'aire et de volume. De tout temps, on a cherché à développer ces outils : au 2e millénaire av. J.-C., les Babyloniens et les Égyptiens savaient déjà calculer l'aire d'un triangle. Au 3e siècle av. J.-C., **Archimède** découvrait la formule donnant l'aire sous une parabole. Mais c'est l'arrivée du calcul différentiel et intégral, au 17e siècle, qui a enfin fourni les clés du calcul de l'aire d'une région de forme quelconque (mais aussi celles du calcul de la longueur d'une courbe et celui du volume d'un solide). On a par la suite réalisé que les probabilités regorgeaient également de calculs d'intégrales. Bref : *l'intégration est un outil scientifique* ***fondamental*** *!*

Pour faire simple, on appelle « intégrale de a à b d'une fonction f définie sur $[a\,;b]$ » l'aire $\mathscr{A}$ délimitée par l'axe des abscisses, les droites d'équations $x = a$ et $x = b$, et la courbe $\mathscr{C}_f$ représentant la fonction f. On note cette quantité $\displaystyle\int_a^b f(x)\,\mathrm{d}x$.



Pour en obtenir une approximation, on peut employer une méthode dite « des rectangles » ou la méthode « des trapèzes ». Leur principe est décrit par les illustrations ci-dessous :



Mathématiquement, avec un nombre n de rectangles (ou de trapèzes), ayant tous la même largeur $p = \dfrac{b-a}{n}$:

$$\mathscr{A} \approx p\sum_{k=0}^{n-1} f(a+kp) \approx p\sum_{k=1}^{n} f(a+kp) \approx p\sum_{k=0}^{n-1}\frac{f(a+kp)+f\big(a+(k+1)p\big)}{2}\,.$$

En notant $x_0 = a$, $x_1 = a+p$, $x_2 = a+2p$, ..., $x_{n-1} = a+(n-1)p$, $x_n = a+np = b$, ces formules se ré-écrivent :

$$\mathscr{A} \approx p\times\big(f(x_0)+f(x_1)+\cdots+f(x_{n-1})\big) \approx p\times\big(f(x_1)+f(x_2)+\cdots+f(x_n)\big)$$

$$\text{ainsi que } \mathscr{A} \approx p\times\left(\frac{f(x_0)+f(x_1)}{2}+\frac{f(x_1)+f(x_2)}{2}+\cdots+\frac{f(x_{n-1})+f(x_n)}{2}\right).$$

On suppose définie une fonction Python nommée `f`, qui prend un paramètre `x` et ne fait que renvoyer le résultat d'un calcul mathématique correspondant à la seule évaluation d'une expression dépendant de ce paramètre `x`. Par exemple : `def f(x): return x**2+1`. C'est le pendant informatique de la définition d'une fonction mathématique f par l'expression $f(x) = x^2+1$.

1. **Écrire une fonction `integrale_rect()` prenant 4 paramètres. Dans l'ordre :**
   - **`a` : la borne inférieure de l'intervalle $[a\,;b]$ sur lequel on cherche à approcher $\displaystyle\int_a^b f(x)\,\mathrm{d}x$ (la**

fonction `f` devra impérativement être définie sur $[a\,;b]$**) ;**

- `b` **: la borne supérieure de l'intervalle** $[a\,;b]$ **(mêmes précisions que ci-avant) ;**
- `n` **: un entier non nul donnant le nombre de rectangles à utiliser dans l'approximation ;**
- `nature` **: une chaîne de caractères valant `"gauche"` ou `"droite"`. Elle gouvernera la façon dont les rectangles sont définis :**
  - `"gauche"` **: le 1er rectangle aura pour aire** $p \times f(x_0)$ **et le dernier** $p \times f(x_{n-1})$ **;**
  - `"droite"` **: le 1er rectangle aura pour aire** $p \times f(x_1)$ **et le dernier** $p \times f(x_n)$**.**

Cette fonction renverra le nombre décimal résultant de la somme des aires des *rectangles*, selon l'une ou l'autre des méthodes « des rectangles » exposées précédemment.

2. **Écrire une fonction `integrale_trap()` prenant 3 paramètres (identiques à ceux de la question précédente — à l'exception du dernier, ici inutile). Cette fonction renverra le nombre décimal résultant de la somme des aires des *trapèzes* (selon la méthode « des trapèzes », donc).**

VALIDATION : on suppose définie la fonction `def f(x): return x**2+1`. Vous pourrez utiliser les assertions :

```
assert integrale_rect(-1, 2, "gauche") == 5.595000000000001, "/!\ integrale_rect()"
assert integrale_rect(-1, 2, "droite") == 6.495, "/!\ integrale_rect()"
```

REMARQUE : *la méthode « des trapèzes » donne comme résultat la moyenne des deux méthodes « des rectangles ». Elle revient à considérer, sur chaque sous-intervalle de* $[a\,;b]$ *d'amplitude* p*, que la fonction* f *« ressemble » à une droite : on parle d'interpolation linéaire. D'autres techniques d'interpolation existent !*