

Lycée Stanislas

■ ■ ■

Cours et TD

d'algorithmique

en

Python

■ ■ ■

A. Camanes

■ Chapitre 1 ■

Introduction à Python

I - Installer Python et se documenter

Nous utiliserons Python version 3.1. Dans les systèmes Linux, Python est généralement déjà installé par défaut. Pour installer Python sous Windows ou Mac, il suffit de télécharger la version 3.1 disponible à l'adresse : <http://www.Python.org/>. Nous utiliserons également l'environnement de développement IDLE.

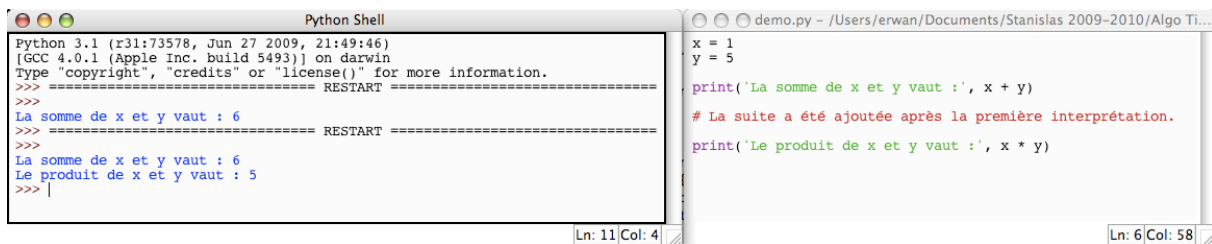
Parmi les nombreuses ressources disponibles sur Internet, on peut citer :

1. l'ouvrage *Apprendre à programmer avec Python* de Gérard Swinnen : <http://inforef.be/swi/python.htm> (chapitres 1 à 7 et chapitre 10).
2. le tutoriel <http://docs.python.org/py3k/tutorial/index.html> (paragraphe 3, 4 et 5).

II - Écrire des programmes

Il est préférable de séparer la *saisie* du programme de son *interprétation*. L'environnement IDLE facilite cette approche.

Pour cela, on ouvre un fichier texte dans une nouvelle fenêtre à côté du terminal. On écrit les instructions, puis on sauvegarde le fichier en choisissant un nom de la forme `nom_du_fichier.py`. Remarquez que l'extension est indispensable pour bénéficier de la mise en couleur des commandes Python. Enfin, on lance l'interprétation dans le terminal (via la touche F5) :



The screenshot shows two windows from the IDLE Python environment. The left window, titled 'Python Shell', displays the Python 3.1 startup message and the execution of a script. The script contains two lines of code: `La somme de x et y vaut : 6` and `Le produit de x et y vaut : 5`. The right window, titled 'demo.py - /Users/erwan/Documents/Stanislas 2009-2010/Algo Ti...', shows the source code of the script. It includes variable assignments `x = 1` and `y = 5`, followed by two `print` statements: `print('La somme de x et y vaut :', x + y)` and `print('Le produit de x et y vaut :', x * y)`. A comment line `# La suite a été ajoutée après la première interprétation.` is also present. The status bars at the bottom of the windows indicate 'Ln: 11 Col: 4' for the shell and 'Ln: 6 Col: 58' for the file editor.

L'encodage des caractères par un ordinateur n'est pas encore unifiée. Pour éviter tout problème, on veillera à proscrire les accents et autres caractères spéciaux.

III - Règles de bonne programmation

Bien programmer, c'est d'abord réfléchir avant d'écrire. Avant de saisir un programme sur l'ordinateur, il faut toujours, en s'armant d'un papier et d'un crayon :

- * Analyser le problème à résoudre.
- * Déterminer les variables nécessaires à sa résolution.
- * Élaborer un premier schéma d'algorithme.

Un programme doit pouvoir être lu et relu. Pour faciliter ces échanges, quelques règles s'imposent :

- * Utiliser des noms de variables et de fonctions *explicites*.
- * *Commenter* les différentes étapes du programme avec le caractère `#`.
- * *Documenter* les fonctions (cf. Partie III).

Enfin, quelques règles de présentation sont à respecter :

- * Placer un espace après une virgule, un point-virgule ou deux-points ;
- * Ne pas placer d'espace avant une virgule, un point-virgule ou deux-points ;
- * Placer un espace de chaque côté d'un opérateur ;
- * Ne pas placer d'espace entre le nom d'une fonction et sa liste d'arguments.
- * Indenter.

■ Chapitre 2 ■

Variables et Fonctions

Affectation			
<code>x = 2</code>			
Opérations			
Addition	+	Soustraction	-
Multiplication	*	Division	/
Fonction puissance	**		
Division			
Partie entière de la division	//	Reste	%
Conversions			
En entier	<code>int()</code>	En flottant	<code>float()</code>
Définition d'une fonction			
<pre>def <nom_de_la_fonction>(<argument_1>,<argument_2>,...,<argument_n>): "Cette ligne explique à quoi sert cette fonction" <instructions></pre>			
Retourner la valeur d'une variable			
<code>return x</code>			

I - Les variables

I.1 - Notion de variable

On peut se représenter la mémoire de Python comme divisée en deux parties :

- * La table des symboles : où sont stockés les *noms* des objets.
À chaque nom d'objet est associé un *type* et une *adresse* dans la table des valeurs.
- * La table des valeurs : où sont stockées les valeurs des objets.
Ce stockage se fait sous forme binaire (suite de bits valant 0 ou 1).

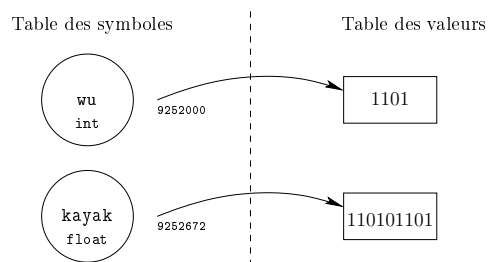


FIGURE 2.1 – Représentation de la table des symboles et de la table des valeurs

I.2 - Affectation

Décomposons le traitement par Python de l'instruction suivante, appelée *affectation* :

`wu = 10 + 1`

- | | |
|---|--|
| <ol style="list-style-type: none"> 1. Python commence par <i>évaluer</i> l'expression <code>10 + 1</code>, et note le <i>type</i> du résultat. 2. Python stocke le résultat obtenu dans la table des valeurs, et note l'adresse utilisée. | <ol style="list-style-type: none"> 3. Python inscrit le <i>nom</i> <code>wu</code> dans la table des symboles, en précisant le type et l'identité associés. |
|---|--|



Le nom d'une variable ou d'une fonction est formé de lettres non accentuées, de chiffres, du caractère tiret bas `_`. Il doit toujours commencer par une lettre.

On accède ensuite aux informations sur la variable :

```
>>> wu    # valeur
11
>>> print(wu) # affiche la valeur (a ne pas utiliser en algorithmique)
11
```

Si on affecte une valeur à une variable dont le nom existe déjà, Python supprime d'abord le nom de la table des symboles puis le réinscrit. Il ne s'agit donc plus de la même variable.

I.3 - Expressions

Une *expression* est formée à partir

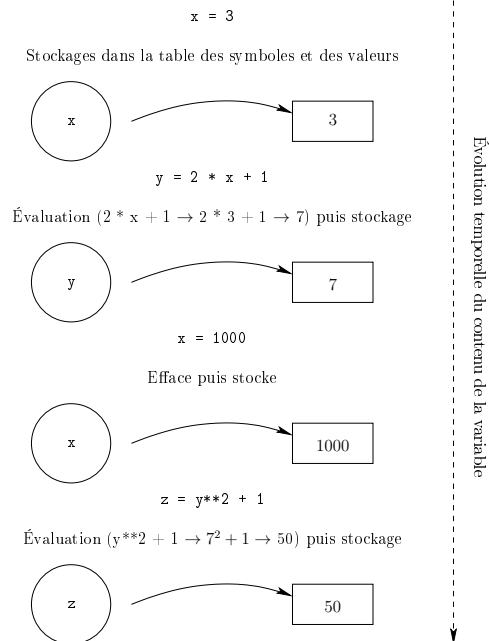
- * de constantes,
- * d'opérateurs,
- * de noms de variables,
- * de noms de fonctions (suivis d'arguments entre parenthèses),
- * ...

Pour évaluer une expression, Python remplace chaque nom de variable par la valeur correspondante, et chaque nom de fonction (suivi d'arguments) par la valeur renvoyée par la fonction. Comme le montre la suite d'instructions suivantes, une expression est toujours évaluée *avant* d'être stockée dans une variable.

Exemple 1.

On représentera l'évolution des différentes variables par un diagramme d'état.

```
>>> x = 3
>>> y = 2 * x + 1
>>> x = 1000
>>> z = y**2 + 1
```



Les *affectations multiples* présentées ci-dessous sont spécifiques au langage Python et ne pourront pas être utilisées dans les autres langages. Toutes les expressions sont évaluées (de gauche à droite) *avant* la première affectation.

Exemple 2.

```
>>> x = 2
>>> y = 3
>>> x, y = 2 * x + 3 * y, 4 * x
>>> x, y
(13, 8)
```

II - Les types numériques

Dans les langages de programmation, à tout objet est associé l'ensemble des valeurs qu'il peut prendre ainsi que les opérations qui peuvent lui être appliquées. Ces caractéristiques sont appelées le *type* de l'objet. Par exemple, un objet peut être un entier, un nombre flottant, une liste... Dans cette partie, nous nous intéressons aux types numériques.

II.1 - Le type int

Ce type est réservé à une variable dont la valeur est un *entier relatif*, stocké en valeur *exacte*. Les valeurs possibles pour une telle variable ne sont limitées que par les capacités de l'ordinateur.



Pour éviter des confusions avec les versions antérieures de Python (où cette notation servait à désigner les entiers écrits en base 8), l'écriture d'un entier ne doit pas commencer par un 0.

II.2 - Le type float

Ce type est réservé à une variable dont la valeur est un *nombre à virgule flottante*. Mathématiquement, ces valeurs correspondent à un sous-ensemble de l'ensemble des nombres décimaux. Ils sont caractérisés par la présence dans leur valeur d'un point `.`, traduction de la virgule en notation anglo-saxonne.



Pour être stocké, tout nombre flottant est d'abord transformé en nombre binaire. Ensuite, ce dernier est stocké sous la forme $\pm 1.a_1 \dots a_n \cdot 2^{\pm b_1 \dots b_p}$, où $a_1, \dots, a_n, b_1, \dots, b_p \in \{0, 1\}$. Le *signe* est stocké sur 1 bit ; la *mantisse* $1.a_1 \dots a_n$ est stockée sur 52 bits (ce qui offre une précision d'environ 15 chiffres significatifs en base 10) ; l'exposant $\pm b_1 \dots b_p$ est stocké sur 11 bits (y compris son signe). Ainsi, les nombres flottants sont compris entre $2^{-2^{10}}$ et $2^{2^{10}}$.

Exemple 3.

On constate sur cet exemple la différence de précision entre les entiers (définis sans limitation de taille) et les nombres flottants (appartenant à l'ensemble précisé ci-dessus).

```
>>> 2**1024
179769313486231590772930519078902473361797697894230657273430081157732675805500
963132708477322407536021120113879871393357658789768814416622492847430639474124
377767893424865485276302219601246094119453082952085005768838150682342462881473
913110540827237163350510684586298239947245938479716304835356329624224137216
>>> 2.**1024
Traceback (most recent call last):
OverflowError: (34, 'Result too large')
```

La commande `int` permet de transformer un flottant en sa partie entière (de type `int`).



Avant d'être stocké, tout nombre est converti en base 2. Lorsque le développement binaire d'un nombre est infini, ce développement est tronqué. On évitera toute utilisation de test d'égalité entre nombres flottants.

Exemple 4.

Les notations ci-dessous seront introduites dans le Chapitre 3 Partie I.

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> 0.5 + 0.5 + 0.5 == 1.5
True
```

II.3 - Les opérateurs

Les opérateurs entre nombres entiers et nombres flottants sont conformes aux us mathématiques :

- * L'*addition* `+`, la *soustraction* `-`.
- * La *multiplication* `*`, la *division* `/`.
- * La *fonction puissance* `**`.
- * La *partie entière* de la division `//`
- * Le *reste* de la division euclidienne `%`.

L'opérateur de multiplication ne peut être omis. Les opérateurs ont la priorité usuelle, et on utilise les parenthèses pour forcer les priorités. Le type du résultat est déterminé par Python en fonction des types des opérandes.



Si les variables `a` et `b` sont de type `float`, la quantité `a // b` est également de type `float`.

Exemple 5.

```
>>> a = 1234567890123456789 * 1.0
>>> A = 1234567890123456789
>>> A // 123456789012345678
10
>>> a // 123456789012345678
9.0
```

III - Les fonctions

Une *fonction* est un algorithme qui prend des *arguments* en entrée, effectue une séquence d'instructions et renvoie un résultat. Une fonction est considérée par Python comme un objet de type `function`.

Exemple 6.

On appelle triplet pythagoricien tout triplet d'entiers (a, b, c) tel que $a^2 + b^2 = c^2$.

```
def test_pythagore(a,b,c):
    '''renvoie 0 si (a,b,c) est un triplet pythagoricien'''
    e = c**2 - a**2 - b**2
    return e
```

On peut ensuite utiliser cette fonction pour tester si des triplets sont pythagoriciens.

```
>>> test_pythagore(3,4,5)
0
>>> test_pythagore(4,5,6)
-5
```

De manière plus générale, la syntaxe pour définir une fonction est la suivante.

```
def <nom_de_la_fonction>(<argument_1>,<argument_2>,...,<argument_n>):
    "Cette ligne explique à quoi sert cette fonction"
    <instructions>
```



On n'oubliera pas les `:` et l'*indentation* qui sont obligatoires en Python !

La définition d'une fonction

- * commence par le *mot-clé* `def`,
- * suivi du nom de la fonction,
- * et d'une liste entre parenthèses de *paramètres formels*; cette première ligne se termine par des double-points `:`.
- * Les instructions qui forment le corps de la fonction commencent sur la ligne suivante, indentée par quatre espaces (ou une tabulation)
- * La première instruction du corps de la fonction peut être un texte dans une chaîne de caractères; cette chaîne est la chaîne de documentation de la fonction. On peut la visualiser dans un terminal en tapant l'instruction `help(nom_de_la_fonction)`.
- * Le retour à la ligne signale la fin de la fonction



Dès que l'instruction `return` est exécutée (si elle est présente), l'exécution de la fonction se termine; la partie du code écrite après l'instruction `return` n'est jamais exécutée.

IV - Variables globales, Variables locales

Pendant l'exécution d'une fonction `<fct>`, Python crée une table *locale* des symboles, où seront inscrits :

- * les noms des arguments de la fonction `<fct>`;
- * les noms des variables et fonctions créées pendant l'exécution de la fonction `<fct>`.

Au cours de l'exécution, si Python a besoin de la valeur d'une variable `grbl`, il va chercher le nom `grbl` dans la table *locale* des symboles ;

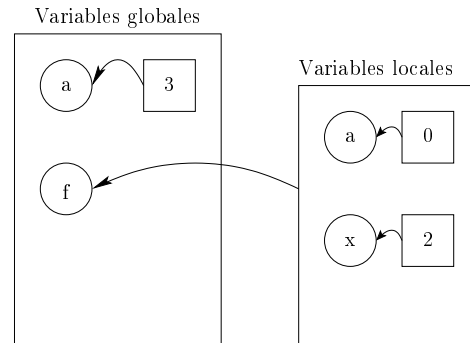
- * si `grbl` ne s'y trouve pas, il consultera la table *globale* ;
- * si `grbl` ne s'y trouve pas non plus, il enverra un message d'erreur.



Ainsi, s'il existe une variable locale et une variable globale portant le même nom, la variable globale ne sera pas accessible pendant l'exécution de la fonction.

Exemple 7.

```
>>> a = 3
>>> def f(x):
...     a = 0
...     return x * a
...
>>> f(2)
0
>>> a
3
```



Il est déconseillé de faire intervenir une variable globale dans une fonction, sauf à la faire passer en argument. En effet, la modification de cette variable globale, après la définition de la fonction, pourrait entraîner une modification de son comportement.

■ Chapitre 3 ■

Les conditionnelles

Comparateurs			
Égalité	==	Différence	!=
Strictement supérieur	>	Strictement inférieur	<
Supérieur ou égal	>=	Inférieur ou égal	<=
Opérateurs booléens			
Non	not		
Ou	or	Et	and
Instruction conditionnelle			
<pre> if <condition_1>: <instructions_1> elif <condition_2>: <instructions_2> elif <condition_3>: <instructions_3> ... else: <instructions_n> </pre>			

I - Le type booléen

Le type `bool` permet de stocker dans une variable la *valeur de vérité* d'une assertion logique. Une variable booléenne ne peut prendre que deux valeurs : `True` (*vrai*) ou `False` (*faux*).



`True` et `False` sont les instances du type `bool`. Ce ne sont donc pas des chaînes de caractères.

On utilise les *opérateurs de comparaison* suivants.

<code>x == y</code>	x est égal à y
<code>x != y</code>	x est différent de y
<code>x > y</code>	x est strictement supérieur à y
<code>x < y</code>	x est strictement inférieur à y
<code>x >= y</code>	x est supérieur ou égal à y
<code>x <= y</code>	x est inférieur ou égal à y



En Python, le symbole `=` est l'opérateur d'affectation, tandis que le symbole `==` est un opérateur de comparaison (ce dernier correspond donc à la relation d'égalité en mathématique).

Pour combiner des variables booléennes, on dispose des trois *opérateurs logiques* qui sont (par ordre de priorité) : `or` (ou), `and` (et), et `not` (non).

Exemple 1.

```

>>> x = 12
>>> (x % 2 == 0) and (x % 3 != 0)
False
>>> (x % 2 == 0) or (x % 3 != 0)
True
>>> not((x % 2 == 0) and (x % 3 != 0))
True

```


II - L'instruction if

Dans certains programmes, on vérifie des conditions avant d'exécuter les instructions. On se sert alors de l'instruction `if`. La forme la plus simple de l'instruction `if` est la suivante :

Exemple 2.

<pre>def racine_carree(a) : if a >= 0 : b = a ** (1/2)</pre>	<pre> return b >>> racine_carree(3) 1.7320508075688772</pre>
---	--

L'expression booléenne après le mot `if` est appelée la *condition*. Si sa valeur est `True`, alors le bloc d'instructions indenté est exécuté. Sinon, Python passe à la suite.

Une seconde forme élaborée de l'instruction `if` permet d'exécuter une ou plusieurs instructions alternative. On utilise alors l'instruction `elif` (qui est la contraction de *else if*). Enfin, l'instruction `else` permet de déterminer les autres cas.

Exemple 3.

<pre>def f(x): if x < -1 : return x + 2 elif x < 1 : return - x else :</pre>	<pre> return x - 2 >>> f(4) 2 >>> f(-.2) 0.2</pre>
--	---

Résumons la structure générale d'un branchement conditionnel.

```
if <condition_1>:
    <instructions_1>
elif <condition_2>:
    <instructions_2>
else:
    <instructions_3>
```

Dans ce cas,

1. La `<condition_1>` est évaluée. Si elle est vraie, les `<instructions_1>` sont exécutées.
2. Si la `<condition_1>` est fausse et la `<condition_2>` est vraie, les `<instructions_2>` sont exécutées.
3. Si la `<condition_1>` et la `<condition_2>` sont fausses, les `<instructions_3>` sont exécutées.



À noter que dès qu'une condition est satisfaite, les instructions associées sont effectuées et le programme sort de la conditionnelle.

Dans une conditionnelle, on peut utiliser plusieurs instructions `elif`. Cependant, il y a au plus une instruction `else`.

Exemple 4.

On veillera à distinguer les deux écritures suivantes.

<pre>def f(x): if x%2 == 0 : a = 0 elif x%3 == 0 : a = 1 elif x%6 == 0 : a = 2 else : a = 3</pre>	<pre> return a >>> f(4) 0 >>> f(9) 1 >>> f(6) 0 >>> f(7) 3</pre>
---	---

```
def g(x):  
    if x%2 == 0 :  
        a = 0  
    if x%3 == 0 :  
        a = 1  
    if x%6 == 0 :  
        a = 2  
    if (x%2!=0 and x%3!=0 and x%6!=0):  
        a = 3  
    return a
```

```
>>> g(4)  
0  
>>> g(9)  
1  
>>> g(6)  
2  
>>> g(7)  
3
```

■ Chapitre 4 ■

Algorithmes itératifs

Liste vide	<code>[]</code>	Type list	
Concaténation	<code>+</code>	Longueur	<code>len(liste)</code>
Extraction	<code>liste[indice]</code>	Ajout d'un élément	<code>liste1.append(a)</code>
Suppression	<code>del(liste[indice])</code>	Extraction d'une partie	<code>liste[debut:fin:pas]</code>
		Copie sans alias	<code>liste1 = liste2[:]</code>
Boucle de répétition			
for <element> in <list>: <instructions>			
Boucle <i>Tant que</i>			
while <condition>: <instructions>			

I - Le type liste

Le type `list` permet de regrouper dans une même variable plusieurs objets (ou *éléments*) rangés dans un ordre déterminé. Une liste peut être de longueur quelconque (dans la limite des capacités de l'ordinateur). Ses éléments peuvent être de types différents. Tous les objets manipulés par Python peuvent être éléments d'une liste.

Pour déclarer une liste, on énumère, entre crochets, des valeurs séparées par des virgules. Une liste vide est notée `[]`.

Exemple 1.

Ces exemples illustrent les notions décrites précédemment. La syntaxe est détaillée par la suite.

```
>>> T = [1, 6, False, print, [3.14, 2.71]]
>>> type(T)
<class 'list'>
>>> type(T[2])
<class 'bool'>
>>> type(T[4][1])
<class 'float'>
```

Les listes peuvent être concaténées (accolées) avec l'opérateur `+`.

Exemple 2.

```
>>> entiers = [0,1,2,3]
>>> suivants = [4,5,6]
>>> tous = entiers + suivants
>>> tous
[0,1,2,3,4,5,6]
```

La commande `len` permet de calculer la longueur d'une liste, i.e. son nombre d'éléments.

Exemple 3.

```
>>> len(tous)
7
```

On peut accéder directement à un caractère d'une liste connaissant sa position dans la liste. Le premier élément d'une liste est en indice 0.

Exemple 4.

```
>>> tous[len(tous)-1]
```

```
6
>>> tous[3] = 5
>>> tous
[0,1,2,5,4,5,6]
```

On peut extraire une sous-liste en déclarant l'indice de début et l'indice de fin, séparés par deux-points. Cette opération est appelée tranchage (*slicing* en anglais).

Exemple 5.

```
>>> tous[1:3]
[1,2]
```

La meilleure façon de se rappeler comment utiliser la technique de tranchage consiste à se représenter une liste comme un saucisson découpé en tranches régulières : on numérote les emplacements des coups de couteaux qu'il a fallu effectuer en partant de la gauche pour trancher le saucisson (l'entame étant entre le 0^e et le 1^{er} coup de couteau). Remarquez également la possibilité de numéroté les tranches négativement.

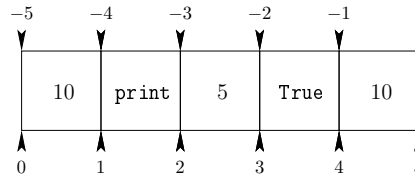


FIGURE 4.1 – Illustration de la technique du tranchage

On peut également donner un troisième indice lors de l'extraction d'une sous-chaîne, qui correspond au *pas* de l'extraction :

Exemple 6.

```
>>> entiers = [0,1,2,3,4,5,6,7,8]
>>> entiers[0:8:2]
[0,2,4,6]
```



Considérons la commande `liste[i:j:k]`.

Si i ou j sont négatifs, alors `len(liste)+i` et `len(liste)+j` leur sont substitués.

Si i et j sont positifs, alors les indices considérés sont les $i + n * k$, où $0 \leq n < \frac{j-i}{k}$.

L'affectation dans des tranches ainsi que la destruction (commande `del`) d'éléments de la liste est aussi possible, et ceci peut changer la taille de la liste.

Exemple 7.

```
>>> a = [4, True, 1., 3.14]
>>> a[1:1]
[]
>>> a[0:3] = [1, 12] ; print(a)      # Remplacer certains éléments
[1, 12, 3.14]
>>> a[1:1] = [3, True] ; print(a)    # En insérer
[1, 3, True, 12, 3.14]
```

Pour ajouter un élément à la fin d'une liste, plusieurs méthodes sont possibles :

Exemple 8.

```
>>> t = [6]
>>> t[len(t):] = [1] ; print(t)     # Affectation dans une tranche
[6, 1]
```

```
>>> t.append(2) ; print(t)      # Méthode .append() de l'objet liste
[6, 1, 2]
>>> t = t + [5] ; print(t)     # Concaténation
[6, 1, 2, 5]
```

La dernière méthode (concaténation) est de nature différente des précédentes. Elle n'utilise pas le caractère modifiable des listes mais crée une nouvelle liste en concaténant la liste initiale et le terme suivant. Cette méthode est donc moins efficace en termes de temps d'exécution et d'utilisation de l'espace mémoire, puisqu'elle nécessite de recopier toute la liste (qui peut être très longue).



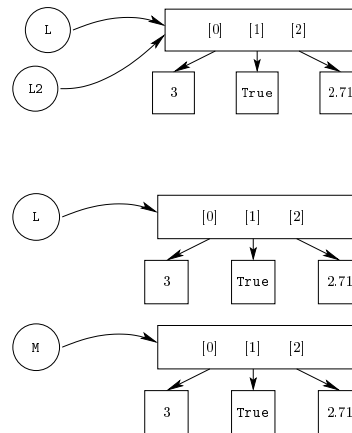
Pour copier la liste L dans la liste M, on utilisera la commande `M = L[:]`.

Exemple 9.

Cet exemple illustre l'utilisation d'alias par Python. On veillera dans les algorithmes à suivre la remarque précédente.

```
>>> L = [3, True, 2.71]
>>> L2 = L
>>> L[1] = False
>>> L2
[3, False, 2.71]

>>> L = [3, True, 2.71]
>>> L3 = L[:]
>>> L3[1] = [ False ]
>>> L3
[3, False, 2.71]
>>> L
[3, True, 2.71]
```



II - La boucle for

Les boucles `for` permettent de réaliser successivement une même action un nombre déterminé de fois.

II.1 - La commande range

La commande `range` permet de créer un objet de type `range` listant une suite arithmétique d'entiers. On s'en servira fréquemment dans des boucles `for`.

Exemple 10.

Remarquons le comportement de la fonction `range`.

```
>>> list(range(1,9))
[1, 2, 3, 4, 5, 6, 7, 8]
```

Plus précisément, la commande `range(debut,fin,pas)` permet de parcourir les entiers compris entre `debut` et `fin-1` avec un pas égal à `pas`.



On prendra garde aux indices de début et de fin !

II.2 - L'itération

La boucle `for` permet d'appliquer un même bloc d'instructions, successivement, à chacun des éléments d'une liste. La syntaxe d'une boucle `for` est la suivante :

```
for <element> in <list>:
    <instructions>
```

Exemple 11.

Pour sommer les entiers pairs compris entre 0 et 6.

```
somme = 0
for x in range(0,7,2) :
    somme = somme + x
```



Comme nous l'avons remarqué précédemment, dans une fonction, la commande **return** termine l'exécution de la fonction. En particulier, le cas échéant, la boucle n'est pas exécutée jusqu'à son terme.

II.3 - Représentation sous forme de tableau

Pour comprendre le fonctionnement des boucles **for**, on s'attachera à réaliser le tableau suivant.

Exemple 12.

```
fact = 1
sum = 0
for i in range(1,6) :
    fact = fact * i
    sum = sum + i
```

i	fact	sum
	1	0
1	1	1
2	2	3
3	6	6
4	24	10
5	120	15

L'exemple plus complexe suivant permet de calculer les 5 premiers termes d'une suite récurrente double appelée suite de Fibonacci définie par $u_0 = u_1 = 1$ et pour tout entier naturel n , $u_{n+2} = u_{n+1} + u_n$.

Exemple 13.

```
a = 1
b = 1
for i in range(2,6) :
    aux = a + b
    a = b
    b = aux
```

i	aux	a	b
		1	1
2	2	1	2
3	3	2	3
4	5	3	5
5	8	5	8

II.4 - Pour aller plus loin

Dans la boucle **for**, l'argument **<list>** n'est pas nécessairement de type **range** mais peut également être de type liste.

Exemple 14.

Les deux fonctions suivantes renvoient toutes deux le minimum de la liste L.

```
def minimum_1(L) :
    m = L[0]
    for element in L :
        if element < m :
            m = element
    return(m)
```

```
def minimum_2(L) :
    l = len(L)
    m = L[0]
    for indice in range(0,l) :
        if L[indice] < m :
            m = L[indice]
    return(m)
```

III - La boucle while

Si on souhaite répéter une séquence d'instructions un nombre de fois non déterminé à l'avance, on utilise une boucle **while**. La syntaxe est la suivante :

```
while <condition>:
    <instructions>
```

Le déroulement de la boucle est le suivant :

- * on évalue la condition (qui doit être une expression booléenne) ;
- * si la condition renvoie la valeur **True**, on effectue les instructions et on revient au début de la boucle ;
- * si elle renvoie la valeur **False**, on sort de la boucle.

Exemple 15.

Écrivons une suite d'instructions qui stocke les termes de la suite de Fibonacci inférieurs à 10.

```
# Initialisation suite
a = 1
b = 1
# Initialisation liste des termes
L = [a]
# Construction
while b <= 10:
    L = L + [b]
    aux = a + b
    a = b
    b = aux
```

b < 10	L	aux	a	b
	[1]		1	1
True	[1, 1]	2	1	2
True	[1, 1, 2]	3	2	3
True	[1, 1, 2, 3]	5	3	5
True	[1, 1, 2, 3, 5]	8	5	8
True	[1, 1, 2, 3, 5, 8]	13	8	13
False				



Il faut veiller à ce que le corps de la boucle contienne une instruction qui change la valeur d'une variable intervenant dans la condition de manière à ce que la boucle termine.

■ Chapitre 5 ■

Algorithmes Récursifs

Un programme récursif est un programme qui fait appel à lui-même. Très souvent un algorithme récursif est lié à une relation de récurrence, permettant de calculer la valeur d'une fonction pour un argument n à l'aide des valeurs de cette fonction pour des arguments inférieurs à n .

Exemple 1 (Calcul de x^n).

Soient $x \in \mathbb{R}$ et $n \in \mathbb{N}$. On peut définir x^n par récurrence à partir des relations :

$$x^0 = 1 \text{ et } x^n = x * x^{n-1} \text{ si } n > 1.$$

```
def puissance(x,n):  
    "calcul récursif de x ** n"  
    if n == 0:  
        return 1  
    else:  
        return x * puissance(x,n-1)  
  
>>> puissance(2,4)  
16
```

```
4 == 0 : False  
Appelle puissance(2, 3)  
3 == 0 : False  
Appelle puissance(2, 2)  
2 == 0 : False  
Appelle puissance(2, 1)  
1 == 0 : False  
Appelle puissance(2, 0)  
0 == 0 : True  
Valeur retournée : 1  
Évaluation de 2 * 1  
Valeur retournée : 2  
Évaluation de 2 * 2  
Valeur retournée : 4  
Évaluation de 2 * 4  
Valeur retournée : 8  
Évaluation de 2 * 8  
Valeur retournée : 16
```



Il faut s'assurer que le cas de base sera atteint en un nombre fini d'étapes.

■ Chapitre 6 ■

Étude d'un algorithme

I - Complexité

Afin de comparer plusieurs algorithmes résolvant un même problème, on introduit des mesures de ces algorithmes appelées *complexités*. On cherchera toujours à écrire l'algorithme dont la complexité est minimale.

- * *Complexité temporelle* : c'est le nombre d'opérations *élémentaires* effectuées par une machine qui exécute l'algorithme.
- * *Complexité spatiale* : c'est le nombre de *positions mémoire* utilisées par une machine qui exécute l'algorithme.

Ces deux complexités dépendent de la machine utilisée mais aussi des données traitées. En pratique, toutes les opérations *élémentaires* n'ont pas le même temps d'exécution : une division demande plus de temps qu'une addition. Pour simplifier, on ne comptera ici que le nombre d'opérations à réaliser (et ce dans le pire des cas) sans tenir compte de leurs différences.

Dans le cas où le nombre d'opérations dépend d'un paramètre n (par exemple la longueur d'une liste), il est intéressant de le majorer par une fonction du type :

- * $K \cdot \ln(n)$. On dit que la complexité est en $O(\ln n)$ et on parle de complexité *logarithmique* (ex : exponentiation rapide).
- * $K \cdot n$. On dit que la complexité est en $O(n)$ et on parle de complexité *linéaire* (ex : recherche du maximum d'une liste).
- * $K \cdot n^2$. On dit que la complexité est en $O(n^2)$ et on parle de complexité *quadratique*.
- * $K \cdot n^\beta$. On dit que la complexité est en $O(n^\beta)$ et on parle de complexité de type *polynomiale*.
- * $K \cdot a^n$. On dit que la complexité est en $O(\exp n)$ et on parle de complexité *exponentielle* (ex : tours de Hanoï).

Exemple 1 (Test de primalité).

Considérons la fonction `est_premier(nombre)` qui permet de déterminer si l'entier naturel `nombre` est premier.

Approche naïve. Le premier algorithme consiste à diviser `nombre` par tous les entiers qui lui sont inférieurs. Si un de ces entiers est un diviseur de `nombre`, `nombre` n'est pas premier.

```
def est_premier(nombre) :
    if nombre == 0 or nombre == 1 :
        return False
    elif nombre == 2 :
        return True
    else :
        for i in range(3, nombre) :
            if nombre % i == 0 :
                return False
    return True
```

Si le nombre vaut N , on effectue au plus N tests. La complexité est linéaire.

Approche moins naïve. Remarquons dans le cas précédent que si `nombre` n'est pas égal à 2, il est inutile de diviser par les nombres pairs.

```
def est_premier(nombre) :
    if nombre == 0 or nombre == 1 :
        return False
    elif nombre == 2 :
        return True
    elif nombre % 2 == 0 :
        return False
    else :
        for i in range(3, nombre, 2) :
            if nombre % i == 0 :
                return False
    return True
```

Si le nombre vaut N , on effectue au plus $N/2$ tests. La complexité est toujours linéaire.

Approche mathématique. Nos connaissances en arithmétique nous assurent que si `nombre` n'est pas premier, un de ses diviseurs est inférieur à $\sqrt{\text{nombre}}$. Le code Python devient alors

```
def est_premier(nombre) :
    if nombre == 0 or nombre == 1 :
        return False
```

```

elif nombre == 2 :
    return True
elif nombre % 2 == 0 :
    return False
else :
    for i in range(3,int(nombre**(1/2))+2,2) :
        if nombre % i == 0 :
            return False
    return True

```

Si le nombre vaut N , on effectue au plus $\sqrt{N}/2$ tests. La complexité est en $O(\sqrt{N})$. Ce dernier programme est donc plus efficace que les précédents.

II - Preuve de bon fonctionnement

Pour la plupart des programmes, on sera amené à *prouver*, sur le papier, qu'un programme a bien dans tous les cas le comportement qu'on attend de lui. Comme on l'a vu dans les paragraphes précédents, il faut en général prouver :

- * **la terminaison** de l'algorithme : vérifier que la condition d'une boucle **while** finit toujours par devenir fausse, ou qu'une fonction récursive finit toujours par atteindre le *cas de base*.
- * **le bon fonctionnement** de l'algorithme : vérifier que le résultat renvoyé est bien celui attendu. Ce type de preuve se rédige de façon différente selon que l'algorithme est itératif ou récursif.

Dans le cas d'un algorithme itératif, le raisonnement sera basé sur un tableau contenant les valeurs successives des variables. Il se décomposera en deux parties :

- * D'une part un raisonnement par l'absurde pour montrer que la boucle finit par s'arrêter ; ce raisonnement est en général basé sur l'idée qu'il n'existe pas de suite d'entiers infinie, strictement décroissante, et minorée (ou strictement croissante et majorée).
- * D'autre part un raisonnement par récurrence finie pour établir les valeurs des variables à la fin (ou au début) de la k -ième itération de la boucle, qui permet en particulier d'obtenir ces valeurs à la fin de la dernière itération.

Exemple 2.

Étudions la fonction suivante (définie sur les entiers) :

```

def fact_iter(n):
    f = 1
    while n > 0:
        f = n * f
        n = n - 1
    return f

```

Prouvons que, pour tout entier naturel n , **fact_iter**(n) est évaluée en temps fini et renvoie $n!$.

Soit $n \in \mathbb{N}$. Pour évaluer **fact_iter**(n), Python crée deux variables locales **n** et **f** ; et leur assigne respectivement les valeurs initiales $n_0 = n$ et $f_0 = 1$.

Si k est un entier naturel non nul tel que la boucle **while** soit parcourue au moins k fois, notons n_k et f_k les valeurs des variables **n** et **f** à la fin de la k -ième itération.

Montrons alors par récurrence finie sur $k \in \llbracket 0, n_0 \rrbracket$ que n_k existe et vaut $n_0 - k$.

Initialisation. n_0 existe et vaut n_0 comme on l'a déjà vu.

Hérédité. Soit $k \in \llbracket 0, n_0 - 1 \rrbracket$ fixé. Supposons que n_k existe (la k -ième itération a donc eu lieu), et que $n_k = n_0 - k$. Comme $k < n$, le booléen $n_k > 0$ vaut **True**. La $(k + 1)$ -ième itération a donc lieu, et n_{k+1} existe. De plus, suite à l'affectation **n** = **n** - 1, on obtient $n_{k+1} = n_k - 1 = n_0 - k - 1$ (n_k est un *invariant* de la boucle).

Conclusion. On a prouvé par récurrence finie que n_{n_0} existe et vaut $n_0 - n_0 = 0$. On en déduit que le booléen $n_{n_0} > 0$ vaut **False**, et que la boucle se termine à l'issue de la n_0 -ième itération.

Ainsi, pour tout $k \in \llbracket 0, n_0 - 1 \rrbracket$, $f_{k+1} = (n_0 - k)f_k$ soit $f_{n_0} = \prod_{k=0}^{n_0-1} (n_0 - k) = n_0!$.

La valeur retournée par la fonction étant la valeur de la variable **f** à la fin de la dernière itération de la boucle **while**, c'est-à-dire f_{n_0} , la fonction renvoie bien $n_0!$.

Dans le cas d'un algorithme récursif, le raisonnement se fera en général par récurrence forte sur l'argument (ou sur un entier mesurant la taille de l'argument). Dans les cas faciles, une récurrence simple suffira.

Exemple 3.

Étudions la fonction suivante :

```
def fact_rec(n):
    if n == 0:
        return 1
    else:
        return n * fact_rec(n-1)
```

Montrons par récurrence sur $n \in \mathbb{N}$ que l'expression `fact_rec(n)` est évaluée en temps fini et renvoie $n!$.

Initialisation. Comme le booléen `0 == 0` vaut `True`, `fact_rec(0)` est évalué en temps fini et renvoie 1, c'est-à-dire $0!$.

Hérédité. Soit $n \in \mathbb{N}$. Supposons que l'expression `fact_rec(n)`, évaluée en temps fini, renvoie $n!$.

Lors de l'évaluation de `fact_rec(n + 1)`, Python commence par évaluer le booléen `n + 1 == 0`, qui vaut `False` puisque $n \geq 0$. Il est donc amené à évaluer l'expression `(n + 1) * fact_rec(n)`. Or, par hypothèse de récurrence, `fact_rec(n)` est évalué en temps fini et renvoie $n!$, donc `(n + 1) * fact_rec(n)` est évalué en temps fini et vaut $(n + 1)!$.

On a ainsi prouvé que `fact_rec(n + 1)`, évalué en temps fini, renvoie $(n + 1)!$.

Conclusion. On a prouvé par récurrence que la fonction `fact_rec` se termine et renvoie le résultat attendu, quel que soit l'argument (entier naturel) qu'elle reçoit.

III - L'utilisation du Débogueur

IDLE possède un débogueur embarqué. Pour l'utiliser, il suffit de suivre la démarche suivante.

1. Dans le menu **Debug** de l'interpréteur, choisir l'option **Debugger**. Une ligne `[DEBUG ON]` s'affiche alors dans le terminal.
2. Dans la fenêtre **Debug Control** qui vient de s'ouvrir, vérifier que les options **Stack**, **Source**, **Locals**, **Globals** sont validées.
3. Lancer un programme dans le terminal.
4. Le programme s'exécute alors pas à pas lorsque vous cliquez sur le bouton **Step**.
Dans la fenêtre principale s'affiche la ligne du programme qui est en cours d'exécution.
Dans les fenêtres **Locals** et **Globals** s'inscrivent les variables qui sont stockées en mémoire ainsi que la valeur qui leur est affectée.
5. Le bouton **Go** permet de quitter le mode pas à pas et de lancer le programme jusqu'à ce qu'il soit terminé... ou qu'une erreur survienne.

■ Chapitre 7 ■

Mémento Python

Instructions et fonctions primitives	
=	Affectation de variable
# <commentaire>	Commentaire (non lu par Python)
Définition de nouvelles fonctions	
def <nomdefonction> :	Définition d'une fonction
return	Renvoie un résultat et sort de la fonction
Opérations arithmétiques sur les entiers et les flottants	
+, -, *, **, /	Addition, soustraction, multiplication, exponentiation, division
//, %	Partie entière de la division, reste de la division euclidienne
abs	valeur absolue
Méthodes relatives aux listes	
[]	Liste vide
s + t	Concaténation de s et t
s[i]	Élément numéro i de s, l'origine étant 0
s[i:j]	tranche de s de i à j
s[i:j:k]	tranche de s de i à j avec un pas k
len(s)	longueur de s
s[i] = x	l'élément d'indice i dans s est remplacé par x
s[i:j] = t	la tranche de s de i à j est remplacée par le contenu de t
del s[i:j]	Suppression des éléments s[i:j]
s.append(x)	Ajout de l'élément x à la fin de la liste s
range(début,fin,pas)	Construction d'un objet contenant les éléments de la suite arithmétique
Booléens	
True, False	Booléens
not, or, and	Opérateurs
==, <=, >=, <, >, !=	Opérateurs de comparaison
Outils de contrôle d'exécution	
if <conditions1> : <instructions1>	Instruction conditionnelle
elif <conditions2> : <instructions2>	
else : <instructions3>	
for k in <list> : <instructions>	Boucle de répétition
while <condition> : <instructions>	Boucle conditionnelle

■ T.D. 1 ■

Découverte de Python

I - Variables

Exercice 1. (✎) Déterminer les valeurs des variables à l'issue des lignes de commandes suivantes.

1. <pre>a1 = 1 i = 1 ai = 3 i = 3 ai = 6</pre>	3. <pre>a = 4 b = 2 c = b b = a</pre>	5. <pre>x = 5 y = 2 * x + 3 x = 100</pre>
2. <pre>a = 2 b = 4 c = a b = a c = b - a</pre>	4. <pre>a = 4 b = 2 c = b b = a a = c</pre>	6. <pre>a = 3 b = 4 x = a a = b b = x</pre>

Exercice 2. (✎) Expliquer pourquoi les lignes de commandes suivantes ne fonctionnent pas.

1. <pre>a = 4 b = 2 2 = a b = 3</pre>	2. <pre>b = 2 b = a</pre>
---	---

Exercice 3. (✎) Donner les valeurs finales des variables x et y.

1. Affectation classique <pre>x = 19 x = x + 2 y = x * 2</pre>	2. Affectation très pythonique. <pre>x = 19 x, y = x + 2, x * 2</pre>
--	---

Exercice 4. (Échange des contenus de deux variables, ♡) On suppose que les variables x et y ont pour valeurs respectives les entiers 3 et 4. On souhaite échanger le contenu de ces deux variables.

1. Proposer une méthode qui utilise une variable auxiliaire appelée tmp.
2. On exécute la séquence d'instructions suivante : `x = x + y`; `y = x - y`; `x = x - y`. Quel est le contenu des variables x et y en fin de séquence?

II - Fonctions

Exercice 5. (✎) Expliquer pourquoi les programmes suivants ne fonctionnent pas.

1. <pre>rad_deg def(angle) : return (angle * 180 / 3.1)</pre>	3. <pre>def vol(r,l) : aire = 3.14 * r**2 volume = aire * l return volume</pre>
2. <pre>def vol(l,r) : volume = 3.14 * l^2 aire = volume * r return aire</pre>	4. <pre>def vol(r,l) : return (pi*r**2*l)</pre>

Exercice 6. (✎) Donner la valeur de la variable `x` à l'issue de ces lignes de code.

```
x = 6
def fct1(y) :
    x = 3 * y + 2
    return x
    x = 5

def fct2(y) :
    return 2 * fct1(y)**2 + 2 * y

x = fct1(x) + 2 * fct2(x) - 3
```

Exercice 7. (→) Pour chacun des programmes suivants, déterminer les valeurs de `mult_3(4)` puis de `N`. Lesquels de ces programmes permettent de retourner la multiplication par 3 ?

1.

```
N = 3
def mult_3(x) :
    N = 4
    return (N * x)
```

2.

```
N = 3
def mult_3(N) :
    return (N * N)
```

3.

```
def mult_3(x) :
    N = 3
    return (N * x)
```

4.

```
N = 3
def mult_3(x) :
    return (N * x)
N = 4
```

Exercice 8. (♡)

1. Écrire une procédure `hms(N)` qui reçoit un nombre `N` de secondes, le convertit en heures / minutes / secondes et retourne ces trois valeurs (on utilisera les opérateurs de division euclidienne).
2. Transformer la quantité 4529 secondes.

Exercice 9. (→)

1. Écrire une fonction `mult(a)` qui prend comme argument un flottant `a` et retourne une fonction qui permet de multiplier tout nombre flottant par `a`.
2. Quel est le type de `mult(7)` ? Calculer `mult(7)(3)` .

■ T.D. 2 ■

Conditionnelles

I - Échauffement

Exercice 1. (✎) Pour chacune des fonctions `f` suivantes, déterminer la valeur de `f(4)`.

1.

```
def f(x) :  
    if x % 2 == 0 :  
        a = 0  
    elif x % 4 == 0 :  
        a = 1  
    else :  
        a = 3  
    return a
```

2.

```
def f(x) :  
    if x % 2 == 0 :  
        a = 0  
    else x % 4 == 0 :  
        a = 1  
    return a
```

3.

```
def f(x) :  
    if x % 2 == 0 :  
        a = 0  
    if x % 4 == 0 :  
        a = 1  
    return a
```

4.

```
def f(x) :  
    if x % 2 = 0 :  
        return 0  
    if x % 4 = 0 :  
        return 1
```

Exercice 2. (✎) Écrire une fonction `ordre(x, y, z)` qui renvoie `True` si `x` divise `y` et `y` divise `z` et `False` sinon.

II - Petits programmes

Exercice 3. (✎)

1. Écrire une fonction `max2(x,y)` qui renvoie le plus grand des deux nombres `x` et `y`.

2. En déduire une fonction `max3` qui calcule le maximum de 3 nombres.

On dispose en Python d'une fonction prédéfinie `max` qui calcule le maximum d'un nombre quelconque d'arguments. Il est bien sûr défendu de l'utiliser pour cet exercice !

Exercice 4. (✎) Écrire une fonction `racines(a,b,c)` prenant comme argument trois réels `a`, `b` et `c` et renvoyant la liste des racines réelles du trinôme aX^2+bX+c . Si le trinôme ne possède pas de racine réelle, la fonction renverra `False`.

On utilisera la fonction racine carrée : $x \mapsto x^{1/2}$.

Exercice 5. (✎)

1. Écrire une fonction booléenne `bissextile(annee)` qui permet de tester si une année est bissextile.

On rappelle que les années bissextiles reviennent tous les 4 ans, sauf les années séculaires, si celles-ci ne sont pas multiples de 400. Ainsi, 1900 n'était pas une année bissextile, alors que 2000 l'était.

2. Écrire la même fonction en utilisant uniquement des opérateurs logiques (sans branchement conditionnel).

Exercice 6. (✎) Écrire une fonction booléenne `date(jour,mois,annee)` qui teste si un ensemble de trois chiffres correspond à une date. Faites attention aux années bissextiles !

On pourra utiliser le test `in` d'appartenance à une liste.

■ T.D. 3 ■

Itératif / Récursif

Pour la plupart des programmes suivants, on s'attachera à évaluer leur complexité ainsi qu'à prouver leur bon fonctionnement.

I - Boucles for

Exercice 1. (✎) Donner le contenu de la liste L à l'issue des lignes de commandes suivantes.

```
L = []
n = 3
for i in range(-1, n):
    if i != 0:
        L.append(i)

for i in range(1, 13, 2*n):
    for j in range(n):
        L.append([i, j])

for i in range(1, n+1):
    for j in range(i):
        if j != n//2:
            L.append([i, j])

for i in range(1, 13, 2*n):
    for j in range(0, i, 2):
        for k in range(2, j, 1):
            b = i > j > k
            if b:
                L.append([i,j,k])
```

Exercice 2. (✎) Quel est le résultat renvoyé par la fonction suivante ?

```
def f(n) :
    u = 1
    for x in range(1,n) :
        u = u * x
    return u
```

Exercice 3. (✎) On suppose que les listes manipulées dans cet exercice sont des listes de nombres.

1. Écrire une fonction `somme(liste)` qui retourne la somme des éléments de `liste`.
2. Écrire une fonction `moyenne(liste)` qui retourne la moyenne des éléments de `liste`.
3. Écrire une fonction `carre(liste)` qui retourne la liste des éléments de `liste` élevés au carré.
4. Écrire une fonction `pair(liste)` qui retourne la liste des éléments pairs de `liste`.

Exercice 4. (♡) Écrire une fonction `max1(L)` (resp. `min1(L)`) qui, étant donnée une liste L de nombres, retourne le plus grand (resp. plus petit) élément de L.

Exercice 5. Écrire une fonction `en_commun(l1,l2)` qui renvoie la liste des éléments de la liste l1 qui se retrouvent dans l2.

Exercice 6. (♡) Écrire une fonction `occurences(mot,lettre)` qui retourne le nombre d'occurences de l'entier `lettre` dans la liste d'entiers `mot`.

Exercice 7. (♥) Écrire une fonction `suite_ab(n)` qui, pour tout entier naturel n , renvoie une valeur approchée du n -ième élément de la suite définie par $u_n = \frac{a_n}{b_n}$, où

$$\begin{cases} a_0 = 3 \\ b_0 = 2 \\ a_{n+1} = a_n + 2b_n \\ b_{n+1} = a_n + b_n \end{cases}$$

Calculer le 10-ième puis le 100-ième élément de cette suite. Que conjecturez-vous ?

II - Boucles while

Exercice 8. (♣) Écrire une fonction `seuil_som_car(M)` qui, pour tout entier M , renvoie le plus petit entier n tel que $1^2 + 2^2 + \dots + n^2 \geq M$.

Exercice 9. (♣) 1. Écrire une fonction `fact(n)` qui étant donné un entier n renvoie $n!$.

2. En déduire une fonction `est_fact(n)` qui détermine si un entier n s'écrit sous la forme $n = k!$, où k est un entier naturel.

Exercice 10. (♥) 1. Écrire, à la main, une fonction `inverse(mot)` qui reçoit une liste, la recopie en inversant l'ordre des éléments et renvoie le résultat.

2. En déduire une fonction booléenne `palindrome(mot)` qui teste si une liste est un palindrome.

3. Écrire une fonction `palindrome_rapide(mot)` qui parcourt le mot au plus une fois pour tester si `mot` est un palindrome.

Exercice 11. (♥) Écrire une fonction `trouve(mot, lettre)` qui renvoie l'indice de la première occurrence de `lettre` dans la liste `mot` et renvoie `False` si `lettre` n'apparaît pas dans `mot`.

Exercice 12. (♣) Écrire une fonction `zero(liste)` qui reçoit une liste, la recopie en insérant des 0 entre les éléments, et renvoie le résultat. Par exemple, `zero([1,2,3])` devra renvoyer `[1,0,2,0,3]`.

Exercice 13. (Syracuse, ♥) La suite de Syracuse est définie par $u_0 \in \mathbb{N}^*$ et par la relation de récurrence :

$$u_{n+1} = \begin{cases} 1 & \text{si } u_n = 1 \\ \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Écrire une fonction `syr2(u0)` prenant pour argument `u0` et renvoyant le nombre minimum d'itérations nécessaires pour aboutir à 1.

On ne sait pas à ce jour s'il existe un réel u_0 pour lequel cette suite n'atteint jamais 1. On ne cherchera pas à prouver que ce programme termine...

III - Récursivité

Exercice 14. (♣) Écrire une fonction récursive `fibonacci_rec(n)` qui prend en entrée un entier n et retourne le terme de rang n de la suite de Fibonacci.

Exercice 15. (♣) Écrire une fonction récursive `palindrome_rec(mot)` qui prend comme argument une liste `mot` et retourne `True` si cette liste est un palindrome, `False` sinon.

Exercice 16. (Exponentiation rapide, ♥) Étant donné un réel positif a et un entier n , remarquer que

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair,} \\ a \cdot \left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair.} \end{cases}$$

Écrire une fonction récursive `puissance_rapide(a,n)` qui utilise cette remarque pour calculer a^n . Évaluer la complexité de cet algorithme.

■ T.D. 4 ■

Arithmétique

Exercice 1. (Algorithme d'Euclide)

1. En utilisant l'algorithme des soustractions successives, écrire une fonction `modulo(a,b)` qui retourne le reste de la division euclidienne de **a** par **b**.
2. En utilisant l'algorithme d'Euclide, écrire une fonction `pgcd(a,b)` qui retourne le plus grand commun diviseur de deux entiers naturels **a** et **b**.
3. À l'aide de la fonction précédente, écrire une fonction `pgcd_liste(L)` qui reçoit une liste de nombres entiers et renvoie leur plus grand commun diviseur.

Exercice 2. (Liste de nombres premiers) Écrire un programme `premiers(N)` qui prend en argument un entier naturel **N** et qui retourne la liste des **N** plus petits nombres premiers.

Exercice 3. (Décomposition en produit de facteurs premiers) Écrire une fonction `decomp_premiers(N)` qui prend en entrée un entier naturel **N** et retourne la liste de ses facteurs premiers.

Exercice 4. (Nombres parfaits) On dit qu'un nombre n est parfait si la somme de ses diviseurs vaut $2 \cdot n$, i.e. $n = \sum_{d|n, d \neq n} d$.

1. Écrire une fonction booléenne `parfait(n)` permettant de vérifier si **n** est parfait.
2. Écrire une fonction `tous_les_parfaits(N)` renvoyant la liste de tous les nombres parfaits inférieurs ou égaux à **N**.

Exercice 5. (Longueur d'un nombre)

1. Écrire une fonction `long(n)` qui étant donnée l'écriture décimale de l'entier naturel $n = \overline{a_p \dots a_0}$ retourne l'entier p ?
2. Écrire une fonction `int_list(n)` qui transforme l'entier naturel $n = \overline{a_p \dots a_0}$ en la liste $[a_p, \dots, a_0]$.

Exercice 6. (Décomposition binaire)

1. Écrire une fonction `binaire(n)` qui calcule la décomposition binaire de tout nombre entier **n** et renvoie le résultat sous forme de liste de 0 et de 1.
2. Écrire une fonction `decimal(n)` qui permet de retrouver l'écriture décimale de l'entier **n**.

■ T.D. 5 ■

Tris

Tri par sélection

Le *tri par sélection* est une des méthodes de tri les plus simples. On commence par rechercher le plus grand élément du tableau que l'on va mettre à sa place : c'est à dire l'échanger avec le dernier élément. Puis on recommence avec le deuxième élément le plus grand, etc. . .

1. Écrire une fonction `maxi(l, n)` qui prend en argument une liste `l` et un entier naturel non nul `n` et qui retourne le maximum et son indice parmi les `n` premiers éléments de `l`.
2. En utilisant la fonction `maxi`, écrire une fonction `tri_selec` qui trie une liste selon la méthode de tri par sélection.
3. Pour une liste de taille `n`, combien de comparaisons effectue votre algorithme? Combien d'échanges effectue votre algorithme dans le pire cas?

Tri par insertion

Le *tri par insertion* est généralement le tri que l'on utilise pour classer des documents : on commence par prendre le premier élément à trier que l'on place en position 1. Puis on insère les éléments dans l'ordre en plaçant chaque nouvel élément à sa bonne place.

Pour procéder à un tri par insertion, il suffit de parcourir la liste : on prend les éléments dans l'ordre. Ensuite, on les compare avec les éléments précédents jusqu'à trouver la place de l'élément qu'on considère. Il ne reste plus qu'à décaler les éléments du tableau pour insérer l'élément considéré à sa place dans la partie déjà triée.

Par exemple, si on veut trier la liste `[10,1,5,19,3,3]`, on obtient successivement

```
10, 1,5,19,3,3
1,10, 5,19,3,3
1,5,10, 19,3,3
1,5,10,19, 3,3
1,3,5,10,19, 3
1,3,3,5,10,19
```

4. Écrire une procédure `insertion(l,n)` qui prend en argument une liste `l` dont on suppose que les `n` premiers éléments sont triés et qui insère l'élément `l[n]` à sa place parmi les `n` premiers éléments de `l`.
5. Écrire une fonction `tri_insert(l)` qui trie une liste selon la méthode du tri par insertion.
6. Quel est le nombre de comparaisons effectuées par votre tri? Quel est le nombre d'échange dans le pire des cas?

Tri rapide

Le *tri rapide* est une des méthodes de tri les plus rapide. L'idée est de prendre un élément quelconque (par exemple le premier) que l'on appelle *pivot* et de le mettre à sa place définitive en plaçant tous les éléments qui sont plus petits à sa gauche et tous ceux qui sont plus grands à sa droite. On recommence ensuite le tri sur les deux sous-listes obtenues jusqu'à ce que la liste soit triée.

7. Écrire une procédure `separation(l)` qui prend en argument une liste `l` et qui retourne la liste des éléments de `l` inférieurs ou égaux au pivot `l[0]` et la liste des éléments de `l` strictement supérieurs au pivot `l[0]`.
8. Écrire un algorithme récursif `tri_rapide(l)` triant une liste en utilisant l'algorithme.
9. Combien de comparaisons fait votre algorithme dans le pire des cas?

Échangeurs de Polynômes

Échangeurs

Dans ce problème on s'intéresse à des polynômes à coefficients réels qui s'annulent en 0. Un tel polynôme P s'écrit donc $P(x) = a_1x + a_2x^2 + \dots + a_mx^m$. Le but de ce problème est d'étudier la position relative autour de l'origine de plusieurs polynômes de ce type.

Dans tout le problème, les polynômes sont représentés par des tableaux (non vides) de nombres flottants de la forme $[a_1; a_2; \dots; a_m]$. Le nombre a_m peut être nul, par conséquent un polynôme donné admet plusieurs représentations sous forme de tableau. Dans l'énoncé, ces tableaux sont indexés à partir de 1 et les éléments du tableau de taille m sont donc indexés de 1 à m .¹

Dans tout ce problème, **vrai** et **faux** sont des termes génériques utilisés pour désigner les booléens du langage choisi. Les élèves remplaceront ces termes par les mots-clefs correspondant dans la rédaction de leurs réponses.

Pour les élèves rédigeant en Maple, le terme « fonction » désigne indifféremment une fonction ou une procédure.

Le problème est découpé en deux parties qui peuvent être traitées de manière indépendante. Cependant, la **Partie II** utilise les notions et notations introduites dans la **Partie I**.

Partie I - Permutation de n polynômes

1. Afin de se familiariser avec cette représentation des polynômes, écrire une fonction **evaluation** qui prend en arguments un polynôme P , représenté par un tableau, et un nombre flottant v , et qui renvoie la valeur de $P(v)$.

Nous commençons notre étude par quelques observations. On remarque que le comportement au voisinage de l'origine est décrit par le premier monôme a_kx^k dont le coefficient a_k est non nul (les coefficients $a_1, \dots, a_{k-1}$ étant donc tous nuls). On remarque alors les deux règles suivantes au voisinage de l'origine :

- * Si la valuation k est paire, le graphe du polynôme reste du même côté de l'axe des abscisses.
- * Si la valuation k est impaire, le graphe du polynôme traverse l'axe des abscisses.

2. Justifier mathématiquement l'assertion précédente.

3. Écrire une fonction **valuation** qui prend en argument un polynôme P et renvoie sa valuation. Par définition, cette fonction renverra 0 si P est le polynôme nul.

On s'intéresse maintenant aux positions relatives autour de l'origine des graphes de deux polynômes donnés.

4. Écrire une fonction **difference** qui prend en arguments deux polynômes P et Q (dont les tailles peuvent être différentes) et qui renvoie la différence des polynômes $P - Q$.

5. Écrire une fonction **compare_neg** qui prend en arguments deux polynômes P et Q et qui renvoie :

- * un entier négatif si $P(x)$ est plus petit que $Q(x)$ au voisinage de 0^- ,
- * 0 si les deux polynômes P et Q sont égaux,
- * un entier positif si $P(x)$ est plus grand que $Q(x)$ au voisinage de 0^- .

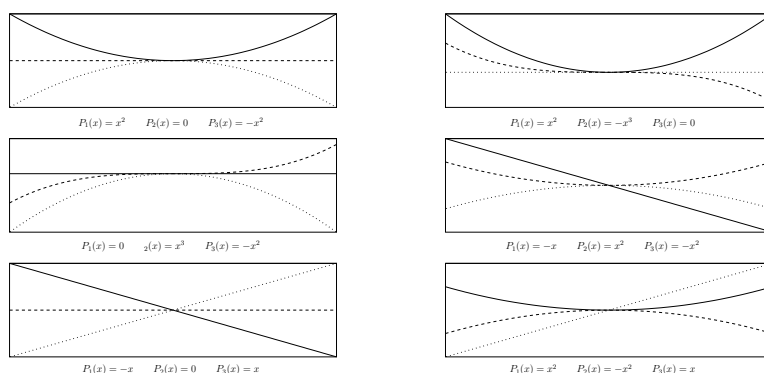
*On admettra sans démonstration que l'ordre usuel sur le résultat de la fonction **compare_neg** définit une relation d'ordre.*

6. Écrire une fonction **tri** qui prend en argument un tableau t contenant n polynômes deux à deux distincts et qui les trie en utilisant la fonction **compare_neg**, de telle sorte que l'on ait $t[1](x) > t[2](x) > \dots > t[n](x)$ pour x négatif et assez petit.

*On supposera par la suite que les tableaux de polynômes $\mathbf{t}$ considérés ont, via la fonction **tri**, été triés de telle sorte que l'on ait $t[1](x) > t[2](x) > \dots > t[n](x)$ pour x négatif et assez petit.*

Enfin, passons à l'étude des graphes de trois polynômes. Les figures ci-après montrent les positions relatives de trois polynômes P_1 , P_2 et P_3 autour de l'origine, avec la légende suivante : P_1 est en trait plein, P_2 en tirets et P_3 en pointillés.

1. Les élèves composant en Python utiliseront les conventions de numérotation des tableaux propres à ce langage dans la rédaction de leurs réponses.



Le choix de ces polynômes est fait pour qu'à chaque fois les inégalités $P_1(x) > P_2(x) > P_3(x)$ soient vérifiées pour x légèrement négatif. Maintenant, observons les positions relatives de ces graphes pour x légèrement positif. On remarque que l'ordre des courbes est permuté : on passe de l'ordre $P_1(x) > P_2(x) > P_3(x)$ à un autre ordre. La donnée des trois polynômes P_1 , P_2 et P_3 définit donc une unique permutation π de $\{1, 2, 3\}$ (i.e. une fonction bijective de $\{1, 2, 3\}$ dans $\{1, 2, 3\}$) telle que $P_{\pi(1)}(x) > P_{\pi(2)}(x) > P_{\pi(3)}(x)$, pour x positif et assez petit. On note que les six permutations de $\{1, 2, 3\}$ sont possibles, comme le montrent les six exemples ci-dessus.

De manière générale, on dit qu'une permutation π de $\llbracket 1, n \rrbracket$ permute les polynômes $P_1, P_2, \dots, P_n$ si et seulement si :

$$P_1(x) > P_2(x) > \dots > P_n(x) \text{ pour } x \text{ négatif assez petit} \\ \text{et } P_{\pi(1)}(x) > P_{\pi(2)}(x) > \dots > P_{\pi(n)}(x) \text{ pour } x \text{ positif assez petit.}$$

Ce qui était vrai pour trois polynômes ne l'est plus à partir de quatre polynômes : il existe des permutations qui ne permutent aucun ensemble de polynômes $P_1, P_2, \dots, P_n$.

Dans la suite, une permutation p de $\llbracket 1, n \rrbracket$ sera représentée par un tableau d'entiers de taille n $[a_1, \dots, a_n]$, de telle sorte que $p(i) = a_i$.

7. Écrire une fonction `verifier_permute` qui prend en argument une permutation p de $\llbracket 1, n \rrbracket$ et un tableau t de même taille et renvoie le booléen `vrai` si p permute les n polynômes $t[1], \dots, t[n]$ contenus dans t , et `faux` sinon. On pourra s'aider d'une fonction `compare_pos`, similaire à la fonction `compare_neg`, pour comparer deux polynômes pour x positif assez petit.

Partie II - Échangeurs de n polynômes

Dans la suite, nous dirons qu'une permutation π de $\llbracket 1, n \rrbracket$ est un échangeur s'il existe n polynômes $P_1, \dots, P_n$ tels que π permute ces polynômes. Nous allons maintenant écrire des fonctions qui répondent aux questions suivantes : Une permutation π est-elle un échangeur ? Peut-on dénombrer les échangeurs ? Peut-on énumérer les échangeurs ?

Une condition nécessaire et suffisante pour qu'une permutation soit un échangeur est la suivante : une permutation π de $\llbracket 1, n \rrbracket$ est un échangeur si et seulement si il n'existe aucuns entiers a, b, c, d tels que $n \geq a > b > c > d \geq 1$ et

$$\pi(b) > \pi(d) > \pi(a) > \pi(c) \text{ ou } \pi(c) > \pi(a) > \pi(d) > \pi(b). \quad (13.1)$$

8. Écrire une fonction `est_echangeur_aux` qui prend en argument une permutation p de $\llbracket 1, n \rrbracket$ et un entier d tel que $1 \leq d \leq n$ et qui renvoie le booléen `vrai` s'il n'existe aucuns entiers a, b et c tels que $n \geq a > b > c > d$ et vérifiant (13.1) et le booléen `faux` sinon.

9. En utilisant la fonction `est_echangeur_aux`, écrire une fonction `est_echangeur` qui prend en argument une permutation p de $\llbracket 1, n \rrbracket$ et renvoie `vrai` si π est un échangeur et `faux` sinon.

On admet sans démonstration que la relation de récurrence suivante permet de compter le nombre $a(n)$ de permutations de $\llbracket 1, n \rrbracket$ qui sont des échangeurs.

$$a(1) = 1, a(n) = a(n-1) + \sum_{i=1}^{n-1} a(i) \cdot a(n-i).$$

10. Écrire une fonction `nombre_echangeurs` qui prend un entier n en argument et renvoie le nombre d'échangeurs $a(n)$.

Enfin, les deux questions suivantes ont pour but d'énumérer tous les échangeurs de $\llbracket 1, n \rrbracket$.

11. Écrire une fonction **caler** qui prend en arguments un tableau t de taille n et un entier v et qui renvoie un nouveau tableau u de taille $n + 1$ tel que

$$\begin{cases} u[1] = v \\ u[i] = t[i - 1] & \text{si } t[i - 1] < v \text{ et } 2 \leq i \leq n + 1 \\ u[i] = 1 + t[i - 1] & \text{si } t[i - 1] \geq v \text{ et } 2 \leq i \leq n + 1 \end{cases}$$

L'algorithme que nous allons utiliser pour énumérer les échangeurs de $\llbracket 1, n \rrbracket$ consiste à énumérer successivement les échangeurs de $\llbracket 1, k \rrbracket$ pour tout k de 1 à n dans un tableau t de taille $a(n)$. Si on suppose qu'un tableau t contient les m échangeurs de $\llbracket 1, k \rrbracket$ entre les cases $t[1]$ et $t[m]$, on peut en déduire les échangeurs de $\llbracket 1, k + 1 \rrbracket$ de la manière suivante : pour tout entier v entre 1 et $k + 1$ et tout entier i entre 1 et m , on décale (à l'aide de la fonction **caler**) l'échangeur $t[i]$ avec v puis on teste si le résultat est un échangeur (avec la fonction **est_echangeur_aux**).

12. Écrire une fonction **enumerer_echangeurs** qui prend un entier n en argument et renvoie un tableau contenant les $a(n)$ échangeurs de $\llbracket 1, n \rrbracket$. On pourra utiliser un second tableau pour stocker temporairement les nouveaux échangeurs.

■ T.D. 7 ■

Codage / Décodage

ave cesar (zudbdrzq)

On cherche à crypter un texte t de longueur n composé de caractères en minuscules non accentuées (soit 26 lettres différentes) représentés par des entiers compris entre 0 et 25 ($0 \leftrightarrow a, 1 \leftrightarrow b, \dots, 25 \leftrightarrow z$). Nous ne tenons pas compte des éventuels espaces. Ainsi, un texte t de n caractères est un tableau de n cases dont chacune des cases est un chiffre compris entre 0 et 25.

Par exemple, le texte `ecolepolytechnique` est représenté par le tableau suivant où la première ligne représente le texte et la seconde les entiers correspondants.

e	c	o	l	e	p	o	l	y	t	e	c	h	n	i	q	u	e
4	2	14	11	4	15	14	11	24	19	4	2	7	13	8	16	20	4

1. Écrire la fonction `max( $\ell, n$ )` qui prend en argument un tableau de chiffres ℓ de longueur n ; et qui retourne l'indice i pour lequel $\ell[i]$ est maximum.
2. Donner la complexité de la fonction `max( $\ell, n$ )` en fonction de la longueur n du texte ℓ .
3. Écrire la fonction `frequencies( $t, n$ )` qui prend en argument le tableau t de longueur n ; et qui retourne un tableau de longueur 26 dont la case d'indice i contient le nombre d'apparitions du nombre i dans t (pour $i \in \llbracket 0, 25 \rrbracket$).
4. Donner la complexité de la commande `frequencies( $t, n$ )` en fonction de la longueur n du texte t .
5. Écrire la fonction `pgcd( $a, b$ )` qui calcule le plus grand commun diviseur des deux entiers a et b par soustractions successives.

Codage de César

Ce codage est le plus rudimentaire que l'on puisse imaginer. Il a été utilisé par Jules César (et même auparavant) pour certaines de ses correspondances. Le principe est de décaler les lettres de l'alphabet vers la gauche de une ou plusieurs positions. Par exemple, en décalant les lettres de 1 position, le caractère **a** se transforme en **z**, le **b** en **a**, ..., le **z** en **y**. Le texte `avecésar` devient donc `zudbdrzq`.

6. Que donne le codage du texte `maitrecorbeau` en utilisant un décalage de 5?
7. Écrire la fonction `codageCesar( $t, n, d$ )` qui prend en arguments le tableau t , sa longueur n et un entier d ; et qui retourne un tableau de même longueur que t contenant le texte t décalé de d positions.
8. Écrire de même la fonction `decodageCesar( $t, n, d$ )` qui prend les mêmes arguments mais qui réalise le décalage dans l'autre sens.

Pour réaliser ce décodage, il faut connaître la valeur du décalage. Une manière de la déterminer automatiquement est d'essayer de deviner cette valeur. L'approche la plus couramment employée est de regarder la fréquence d'apparition de chaque lettre dans le texte crypté. En effet, la lettre la plus fréquente dans un texte suffisamment long en français est la lettre **e**.

9. Écrire la fonction `decodageAutoCesar( $t, n$ )` qui prend en argument un tableau t de longueur n représentant le texte crypté; et qui retourne le texte supposé d'origine (en calculant le décalage pour que la lettre **e** soit la plus fréquente dans le message).

Le codage Vigenère

Au XVIème siècle, Blaise de Vigenère a modernisé le codage de César très peu résistant de la manière suivante. Au lieu de décaler toutes les lettres du texte de la même manière, on utilise un texte clé qui donne une suite de décalages.

Prenons par exemple la clé `concours`. Pour crypter un texte, on code la première lettre en utilisant le décalage qui envoie le **a** sur le **c** (la première lettre de la clé). Pour la deuxième lettre, on prend le décalage qui envoie le **a** sur le **o** (la seconde lettre de la clé) et ainsi de suite. Pour la huitième lettre, on utilise le décalage de **a** vers **s**, puis, pour la neuvième, on reprend la clé à partir de sa première lettre. Sur l'exemple `ecolepolytechnique` avec la clé `concours`, on obtient (la première ligne donne la clé, la deuxième le message à coder et la troisième le message crypté)

c	o	n	c	o	u	r	s	c	o	n	c	o	u	r	s	c	o
e	c	o	l	e	p	o	l	y	t	e	c	h	n	i	q	u	e
g	q	b	n	s	j	f	d	a	h	r	e	v	h	z	i	w	s

10. Donner le codage du texte **becunfromage** en utilisant la clé de codage **jean**.

11. Écrire la fonction `codageVigenere(t, n, c, k)` qui prend comme arguments un tableau t de longueur n représentant le texte à crypter, et un tableau d'entiers c de longueur k donnant la clé servant au codage ; et qui retourne un tableau de longueur n contenant le texte crypté.

Les questions qui suivent sont facultatives. N'entamez leur traitement que si vous avez répondu à toutes les questions précédentes.

Maintenant, on suppose disposer d'un texte t' assez long crypté par la méthode de Vigenère et on veut retrouver le texte t d'origine. Pour cela, on doit trouver la clé c ayant servi au codage. On procède en deux temps : 1) détermination de la longueur k de la clé c , 2) détermination des lettres composant c .

La première étape est la plus difficile. On remarque que deux lettres identiques dans t espacées de $\ell \times k$ caractères (où ℓ est un entier et k est la longueur de la clé) sont codées par la même lettre dans le texte crypté t' . Mais cette condition n'est pas suffisante pour déterminer la longueur k de la clé c puisque des répétitions peuvent apparaître dans t' sans qu'elles existent dans t . Par exemple, les lettres **t** et **n** sont toutes deux codées par la lettre **h** dans le texte crypté à partir de **ecolepolytechnique** avec **concours** comme clé. Pour éviter ce problème, on recherche les répétitions non pas d'une lettre mais de séquences de lettres dans t' puisque deux séquences de lettres répétées dans t dont les premières lettres sont espacées par $\ell \times k$ caractères, sont aussi cryptées par deux mêmes séquences dans t' .

Dans la suite de l'énoncé, on ne considère que des séquences de longueur 3 en supposant que toute répétition d'une séquence de 3 lettres dans t' provient exclusivement d'une séquence de 3 lettres répétée dans t . Ainsi, la distance séparant ces répétitions donne des multiples de k . La valeur de k est alors obtenue en prenant le plus grand commun diviseur de tous ces multiples. Si le nombre de répétitions est suffisant, on a de bonnes chances d'obtenir la valeur de k . On suppose donc que cette assertion est vraie.

12. Écrire la fonction `pgcdDistancesEntreRepetitions(t', n, i)` qui prend en argument le texte crypté t' de longueur n et un entier $i \in \llbracket 0, n-3 \rrbracket$ qui est l'indice d'une lettre de t' et qui retourne le plus grand commun diviseur de toutes les distances entre les répétitions de la séquence de 3 lettres $[t'[i], t'[i+1], t'[i+2]]$ dans la suite du texte $[t'[i+3], t'[i+4], \dots, t'[n-1]]$. Cette fonction retourne 0 s'il n'y a pas de répétition.

13. Écrire la fonction `longueurCle(t', n)` qui prend en argument le texte crypté t' de longueur n et qui retourne la longueur k de la clé de codage.

14. Donner une idée de l'algorithme permettant de retrouver chacune des lettres de la clé. (Il s'agit de décrire assez précisément l'algorithme plutôt que d'écrire le programme).

15. Écrire la fonction `decodageVigenereAuto(t', n)` qui prend en argument le tableau t' de longueur n représentant le texte crypté et qui retourne le texte t d'origine. (On n'hésitera pas à recopier des parties de texte dans des tableaux intermédiaires).

■ T.D. 8 ■

Quelques exercices en plus

Exercice 1.

1. Écrire une fonction `fahrenheit(celsius)` qui convertit une température de degrés Celsius en degrés Fahrenheit et renvoie le résultat (on rappelle que $f = \frac{9}{5}c + 32$).
2. Améliorer la fonction précédente pour qu'elle arrondisse automatiquement le résultat au centième de degré Fahrenheit inférieur.
3. Combien font 232.8 degrés Celsius en degrés Fahrenheit ?

Exercice 2. Réécrire, à l'aide d'une boucle `for`,

1. la fonction `somme(n)` qui renvoie la somme des n premiers entiers.
2. la fonction `som_car(n)` qui renvoie la somme des carrés des n premiers entiers.
3. Écrire une fonction `valid_somme_car(n)` qui permet de vérifier sur les n premiers entiers que `som_car(n)` renvoie bien $\frac{n(n+1)(2n+1)}{6}$.

Exercice 3. Écrire une fonction `N2(n,m)` qui étant donné deux entiers n et m renvoie la liste des couples d'entiers de $[0,n] \times [0,m]$.

Exercice 4. Écrire une fonction `est_presente(syllabe,mot)` qui teste si une suite de chiffres `syllabe` est présente dans une liste `mot`.

Exercice 5.

1. Écrire une fonction booléenne `colineaire(u,v)` qui teste la colinéarité de deux vecteurs u et v du plan, donnés comme des listes de coordonnées cartésiennes.
2. Améliorer la fonction `colineaire(u,v)` pour qu'elle accepte deux vecteurs de même taille n , pour tout $n \geq 1$.

Exercice 6. (Algorithme d'Euclide)

1. En utilisant l'algorithme des soustractions successives, écrire une fonction récursive `modulo(a,b)` qui retourne le reste de la division euclidienne de a par b .
2. En utilisant l'algorithme d'Euclide, écrire une fonction récursive `pgcd(a,b)` qui retourne le plus grand commun diviseur de deux entiers naturels a et b .

Exercice 7. (Résolution d'une équation par dichotomie, $\Rightarrow$) Soit $f : [a,b] \rightarrow \mathbb{R}$ une fonction continue sur le segment $[a,b]$ telle que $f(a) \cdot f(b) < 0$. D'après le théorème des valeurs intermédiaires, l'équation $f(x) = 0$ admet au moins une solution dans l'intervalle $[a,b]$. On suppose également que f strictement monotone sur $[a,b]$. Ainsi, il y a unicité de la solution.

1. Écrire un algorithme non récursif `dicho(f,a,b,eps)` permettant de calculer par dichotomie un intervalle de longueur `eps` contenant la solution.
2. Écrire un algorithme récursif `dicho_rec(f,a,b,eps)` permettant de calculer par dichotomie un intervalle de longueur `eps` contenant la solution.
3. Donner une valeur approchée du point fixe de la fonction cosinus à 10^{-7} près.

On importera le module `math` pour définir la fonction cosinus.

Exercice 8. Écrire une fonction `Hanoi(n,tige1,tige2)` en Python qui prend comme arguments le nombre n de disques de la tour, la tige `tige1` de départ et la tige `tige2` d'arrivée (choisies parmi les trois tiges A, B et C) et qui retourne une liste d'instructions pas à pas qui permettra au manipulateur de transporter la tour disque par disque (on numérottera les disques $1, \dots, n$ par ordre de taille croissante).

Exercice 9. (Puissance finale) Soient n et a deux entiers non nuls. On dit que n est une puissance finale de a si le nombre n termine le nombre a^n . Par exemple, $2^{36} = 68719476736$ et 36 est une puissance finale de 2.

1. Écrire une fonction booléenne `finale(n,a)` permettant de vérifier si n est une puissance finale de a .
2. Écrire une fonction `toutes_les_finales(N,a)` renvoyant la liste de toutes les puissances finales de a inférieures ou égales à N .

Exercice 10. Un nombre est dit tricolore si son carré s'écrit uniquement avec des nombres carrés parfaits non nuls, i.e. 1, 4, 9.

1. Écrire une fonction booléenne `tricolore(n)` permettant de vérifier si n est tricolore.
2. Écrire une fonction `tous_les_tricolores(N)` renvoyant la liste de tous les entiers tricolores inférieurs ou égaux à N .