

Informatique MP2

Cours



Table des matières

1	Rappels de première année	1
1.1	Les types usuels	1
1.1.1	Les entiers et flottants	1
1.1.2	Les booléens	1
1.1.3	Chaînes de caractères	1
1.1.4	Conversions	2
1.2	Structures de contrôle	2
1.2.1	Tests	2
1.2.2	Les boucles	2
1.3	Fonctions	2
1.3.1	Définitions	2
1.3.2	Autres situations	3
1.3.3	Variables locales et globales	3
1.4	Listes	3
1.4.1	Création	3
1.4.2	Accès aux éléments	4
1.4.3	Ajout d'un élément et concaténation	4
1.4.4	Les n -uplets (tuple)	4
1.4.5	Listes et fonctions	4
1.4.6	Quelques méthodes en complément	5
1.5	Représentation interne des variables en Python	5
1.6	Modules	6
1.6.1	Module math	7
1.6.2	Autres modules usuels	7
2	Algorithmes, complexité et preuves	9
2.1	Algorithmes de première année	9
2.1.1	Algorithmes simples sur les listes	9
2.1.2	Recherche dichotomique dans une liste triée	9
2.1.3	Recherche d'un mot dans une chaîne	10
2.1.4	Analyse numérique	11
2.2	Complexité	11
2.2.1	Généralités	11
2.2.2	Exemple et notations	11
2.2.3	Ordres de grandeur	11
2.3	Preuves de programmes	12
2.3.1	Boucle non conditionnelle : calcul de $n!$	12
2.3.2	Boucle conditionnelle : division euclidienne	12
2.3.3	Exercices	13
3	Piles et récursivité	15
3.1	Listes et tableaux	15
3.1.1	Tableaux homogènes	15
3.1.2	Listes simplement chaînées	15
3.1.3	Listes doublement chaînées	15
3.1.4	En Python	15
3.2	Piles	16
3.2.1	Généralités	16
3.2.2	En Python	16
3.2.3	Pile d'exécution / Pile d'appel	16
3.2.4	Complément : programmation objet	16
3.3	Récursivité	17
3.3.1	Généralités	17
3.3.2	Exemple : calcul de puissances	18
3.3.3	Suite de Fibonacci	18

4 Tris	21
4.1 Généralités	21
4.1.1 Présentation	21
4.1.2 Les tris au programme	21
4.2 Les tris simples	21
4.2.1 Tri par sélection	21
4.2.2 Tri bulle	22
4.2.3 Tri par insertion	22
4.3 Les tris évolués	23
4.4 Tri fusion	23
4.4.1 Principe	23
4.4.2 Tri rapide	24
4.5 Bilan	24
5 Numpy et Matplotlib	25
5.1 Numpy et les tableaux	25
5.1.1 Création de tableaux (ndarray)	25
5.1.2 Caractéristiques d'un tableau	25
5.1.3 Opérations sur les tableaux	25
5.1.4 Sélection, extraction	26
5.1.5 Redimensionnement	26
5.1.6 Nombres aléatoires	26
5.2 Matplotlib	27
5.2.1 Graphe simple	27
5.2.2 Plusieurs courbes en même temps	27
5.2.3 Quelques améliorations	27
6 Résumé des commandes pour l'oral de Centrale	29
6.1 Calcul matriciel	29
6.2 Polynômes	29
6.3 Graphiques	29
6.4 Probabilités	30
6.5 Analyse numérique	30
7 CCP MP	31
7.1 CCP MP 2019	31
7.2 CCP MP 2018	31
7.3 CCP MP 2017	32
7.4 CCP MP 2016	32
7.5 CCP MP 2015	33
8 Divers	35
8.1 Les nombres flottants	35
9 Solutions	37

Chapitre 1 | Rappels de première année

I | Les types usuels

I.1 | Les entiers et flottants



Opérations sur les entiers et flottants

<code>+, -, *</code>	opérations classiques	
<code>//</code>	division entière	
<code>/</code>	division flottante	convertit si besoin
<code>%</code>	modulo (reste de la division euclidienne)	
<code>abs(x)</code>	valeur absolue	int, float, complex
<code>pow(x, y)</code> ou <code>x**y</code>	puissance	avec int, float et complex

I.2 | Les booléens

- Deux valeurs définies : *True* et *False* (correspondent aux entiers 1 et 0)
- opérateurs :
 - `bool1 and bool2` : évalue d'abord `bool1` et si l'expression est vraie, évalue alors `bool2` (appelé évaluation paresseuse - très utilisé lors des tests afin notamment de contrôler qu'une variable prend une valeur acceptable)
 - `bool1 or bool2` : même principe sur l'évaluation, s'arrête dès qu'une condition est vraie
 - `not bool1`

I.3 | Chaînes de caractères

Définition d'une chaîne de caractères

- Une chaîne de caractères est définie sous la forme 'chaîne' ou "chaîne".
- Le caractère d'échappement \ a plusieurs utilisations. Il permet d'écrire sur plusieurs lignes une ligne trop grande mais aussi d'accéder à certains caractères dans les chaînes : pour utiliser les " ou ', pour certains caractères spéciaux (les plus courants sont \n pour le retour à la ligne, \t pour une tabulation) :

```
>>> C="Bonjour !\nLa chaîne est sur plusieurs lignes\nenfin presque"
>>> C
'Bonjour !\nLa chaîne est sur plusieurs lignes\nenfin presque'
>>> print(C)
Bonjour !
La chaîne est sur plusieurs lignes
enfin presque
```

Une dernière méthode pour saisir des chaînes avec des caractères spéciaux : les triples guillemets ou triples apostrophes. On tape alors tout ce qu'on veut, sur plusieurs lignes si l'on souhaite. Cette façon est notamment utilisée pour documenter convenablement une fonction.

Opérations sur les chaînes de caractères



Opérations sur les chaînes

<code>s+t</code>	concaténation des deux chaînes
<code>s*n</code>	création d'une chaîne ou s est copiée n fois
<code>s[i]</code>	éléments en position i - le premier est le 0 (uniquement en lecture)
<code>s[i:j]</code>	tranche entre l'élément i (inclus) et j (exclu)
<code>s[i:j:k]</code>	tranche entre l'élément i (inclus) et j (exclu) avec un pas de k
<code>len(s)</code>	longueur de s



I.4 | Conversions

Il suffit en général d'une commande sous la forme `type(donnees)` :

```
>>> chaîne='123'
>>> int(chaîne)+3
126
>>> float(chaîne)+3
126.0
>>> str(23+12)
'35'
```

II | Structures de contrôle

II.1 | Tests

Syntaxe

```
1 if test1:
2     debut_bloc_1
3     if sous_test1:
4         sous_bloc_interne
5     else:
6         sous_bloc_sinon
7     fin_bloc_1
```

Commentaire

Utilisation de l'évaluation paresseuse : sur un test `if (cond1) and (cond2)`, on détermine d'abord le résultat du premier test. S'il est faux, le second n'est pas évalué. C'est notamment utile lorsqu'on doit vérifier que l'indice dans un tableau est acceptable. Si L est un tableau de taille n : `if (i<n) and (L[i]...)` : permet de s'assurer qu'on n'aura pas un `Index out of range` en évaluant le second test.

II.2 | Les boucles

Boucles inconditionnelles

```
1 for x in iterateur:
```

où `iterateur` est un objet « itérable ». Pour les objets itérables usuels :

- `range(n)` (entier de 0 à $n-1$), `range(a,b)` (entiers de a à $b-1$), `range(a,b,pas)` (entiers $a+k.pas$ strictement inférieurs à b),
- les listes : `x` va décrire chacun des éléments de la liste,
- les chaînes : `x` va décrire chaque caractère de la chaîne



modification du compteur

on peut modifier la valeur de `x` à l'intérieur de la boucle, en revanche au passage suivant, `x` prendra systématiquement la valeur suivante de l'itérateur (si `x` décrit les entiers de 0 à 9 et `x` en est à la valeur 4, même si on modifie `x`, au passage suivant il prendra la valeur suivante à savoir 5)

Boucles conditionnelles (tant que)

```
1 while test:
2     bloc1
```

III | Fonctions

III.1 | Définitions

```
1 def f(x):
2     traitement
3     return(...) # pas obligatoire
```

Il n'y a pas de différence entre fonction (prend des arguments et retourne le résultat) et procédure (bloc de programme qui a une utilité particulière sans retourner de valeurs), donc une fonction n'est pas obligée de renvoyer quelque chose. Elle peut cependant renvoyer n'importe quel type d'objet (et même renvoyer des objets de types différents suivant les valeurs de x)

On peut définir des fonctions prenant plusieurs paramètres en entrée :

```
>>> def f(x,y):
        return x*y
>>> f(3,4)
12
```

III.2 | Autres situations

On peut avoir besoin d'une fonction simple (par exemple pour la transmettre en argument d'une autre) sans avoir envie de lui donner un nom (et la définir avec def). Pour cela, on dispose de l'opérateur lambda

```
>>> f = lambda x:x*x
>>> f(2)
4
>>> (lambda x,y : x+y)(3,4)
7
```

On peut avoir besoin localement d'une fonction dans une autre (avec même éventuellement un argument de la fonction extérieure qui peut être utilisée). Par exemple, on veut manipuler une fonction qui dépend d'un paramètre :

```
1 def essai(n):
2     def f(x):
3         # utilise la valeur de n transmise en argument de essai
4         return 1/(n*n+x*x)
5     ...
```

III.3 | Variables locales et globales

```
>>> def f():
...     a=3
...     print(a)
>>> a=1
>>> f()
3
>>> a
1
```

La variable a affectée dans la fonction f est locale à cette fonction. Elle est même totalement oubliée lorsqu'on quitte la fonction :

En résumé (presque exact), on a

- les arguments passés sont créés en tant que variable locale,
- toute variable affectée dans une fonction est locale
- une fonction cherche d'abord dans les variables locales à la fonction puis dans les variables globales,
- on peut préciser qu'on veut utiliser la version globale d'une variable (et ainsi la modifier globalement) en précisant son statut global dans la fonction (avant de l'utiliser) avec le mot clé global (sous la forme global variable) - c'est évidemment à utiliser plus que le moins possible (on se retrouve avec une variable modifiée sans le savoir... dangereux).

IV | Listes

IV.1 | Création

Une liste est un ensemble ordonné (chaque élément à un numéro d'ordre) d'éléments de types hétérogènes. Elles sont définies entre crochets et les éléments sont séparés par une virgule. On peut créer une liste

- directement : L = [1,2,3]
- par conversion d'un objet itérable : L = list(range(10)), L=list("une chaine de caractères")
- par concaténation de listes : L = L1+L2
- par copie d'une liste L = L1.copy()
- par concaténation multiple L = [0]*5 (donne [0,0,0,0,0] : très utile pour initialiser une liste à un certain nombre d'éléments).
- par compréhension : [fonction(x) for x in iterable]
- par compréhension et filtrage : [fonction(x) for x in iterable if conditions]

Par exemple, on veut construire tous les carrés des éléments plus grand que 2 d'une liste

```
>>> l=[0,4,1,-3,7,8]
>>> [x*x for x in l if x>=2]
[16, 49, 64]
```



IV.2 | Accès aux éléments

La numérotation des éléments d'une liste est exactement la même que pour les chaînes, à la différence que cette fois on peut accéder aux éléments à la fois en lecture et en écriture : si `l` est une liste



Accès aux éléments

<code>l[i]</code>	éléments en position <code>i</code> - le premier est le 0
<code>l[i:j]</code>	tranche entre l'élément <code>i</code> (inclus) et <code>j</code> (exclu)
<code>l[i:j:k]</code>	tranche entre l'élément <code>i</code> (inclus) et <code>j</code> (exclu) avec un pas de <code>k</code>
<code>l[i:]</code>	tranche entre l'élément <code>i</code> (inclus) et la fin
<code>l[:j]</code>	tranche entre le début et l'élément <code>j</code> (exclu)
<code>len(s)</code>	taille de la liste

IV.3 | Ajout d'un élément et concaténation

- Lorsqu'on veut ajouter en fin d'une liste `l`, on peut le faire grâce à l'opérateur `+` :

```
>>> l=[4,5,6]
>>> l+[8]
[4, 5, 6, 8]
>>> l
[4, 5, 6]
```

l'opération `l+[8]` crée une nouvelle liste et ne modifie pas `l`. Si on n'affecte pas le résultat, on a perdu la modification. On peut le réaliser entre deux listes, et même concaténer plusieurs fois une même liste :

```
>>> l=[4,5,6]
>>> m=[0,1]
>>> l+m
[4, 5, 6, 0, 1]
>>> m*3
[0, 1, 0, 1, 0, 1]
>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Avec la méthode `append` de la classe `list`. Cette fois on modifie effectivement le contenu de la liste

```
>>> l=[4,5,6]
>>> l.append(8)
>>> l
[4, 5, 6, 8]
```

IV.4 | Les n -uplets (tuple)

Il existe un type proche de celui des listes : les tuples (ou n -uplets). On remplace les crochets par des parenthèses lors de leur définition. La grosse différence est que les données ne sont pas modifiables. C'est notamment ce que renvoie un `return` avec plusieurs valeurs de retour.

IV.5 | Listes et fonctions

Pour les explications, lire le paragraphe suivant. Il faut continuer à dissocier la liste et le contenu de la liste dans la fonction appelée. L'exécution de

```
1 def f(liste):
2     liste[1]="modif"
3     # la fonction ne retourne rien
4
5 l=[0,1,2]
6 f(l)
7 print(l)
```

donne à l'affichage

```
[0, 'modif', 2]
```

- La liste `l` est transmise en argument de la fonction `f` sous un identifiant `liste` local à la fonction est créé et pointe au même endroit que la liste `l`.
- L'affectation `liste[1]="modif"` change le contenu de la liste, mais ne modifie pas son identifiant si bien que `liste` pointe toujours au même endroit que `l`


```

1 def f(liste):
2     print("avant", liste)
3     liste=[4,5,6]
4     print("après", liste)

```

```

>>> l=[0,1,2]
>>> f(l)
avant [0, 1, 2]
après [4, 5, 6]
>>> l
[0, 1, 2]

```

IV.6 | Quelques méthodes en complément



méthodes sur les listes

utilisables

<code>l.append(a)</code>	ajoute l'élément <code>a</code> à la fin de la liste <code>l</code>
<code>l.pop(indice)</code>	retourne l'élément d'indice <code>indice</code> et le retire de la liste <code>l</code> . Par défaut, c'est le dernier élément qui est retiré. Si la liste est vide, la méthode renvoie une erreur
<code>l.sort()</code>	trie la liste <code>l</code>

à éviter

<code>l.insert(indice,a)</code>	insert l'élément <code>a</code> à l'indice <code>indice</code> . Si cet indice dépasse la taille de la liste, l'élément est ajouté à la fin
<code>l.extend(iter)</code>	ajoute à la fin de <code>l</code> la liste créée à partir de l'objet itérable <code>iter</code>
<code>l.index(a)</code>	retourne l'indice de l'élément <code>a</code> dans la liste <code>l</code> s'il est présent dans cette liste et une erreur sinon
<code>l.count(a)</code>	compte le nombre d'occurrence de l'élément <code>a</code> dans la liste <code>l</code>

V | Représentation interne des variables en Python

Une partie un peu plus difficile et plus spécifique au langage Python... on va devoir comprendre la différence entre

```

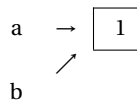
>>> a=1
>>> b=a
>>> a=2
>>> b
1

```

qui semble tout à fait logique. Que se passe-t-il lorsqu'on valide `a=1` : un emplacement mémoire avec l'entier 1 est créé et `a` est un alias qui va pointer vers cet emplacement. On peut le représenter ainsi :



Après `b=a`, on se trouve dans la situation



l'affectation finale `a=2` nous amène à

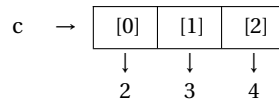


```

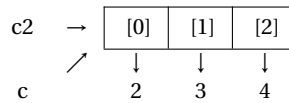
>>> c=[2,3,4]
>>> c2=c
>>> c2[1]=9
>>> c
[2, 9, 4]

```

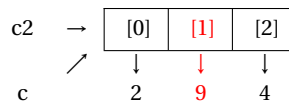
La liste `c` est représentée ainsi :



puis



et enfin



Les deux listes c et c2 ont toujours le même identifiant car on n'a pas modifié la liste c2 mais seulement le lien vers la valeur sur laquelle point le deuxième élément de la liste.

VI | Modules

On peut importer des objets d'un module (un fichier d'instructions Python) de différentes façons et à différents endroits de la mémoire :

```
>>> import math
```

Le module `math` est chargé en mémoire et tout est placé dans « un espace de noms », c'est-à-dire (en simplifié) une zone de mémoire propre qui ne se mélange pas avec les données et variables déjà affectées. Pour accéder à un objet de cet espace de noms, on utilise son préfixe, ici le nom du module chargé :

```
>>> math.sin(0)
0
```

On peut l'importer sous un autre nom

```
>>> import math as m
>>> m.sin(0)
0
```

On peut importer un sous-module d'un module. On le fait fréquemment lorsqu'on utilise de très grosses bibliothèques comme Numpy/-Matplotlib

```
>>> import matplotlib.pyplot as plt
```

le sous-module `pyplot` de `matplotlib` est importé dans l'espace de noms « `plt` ».

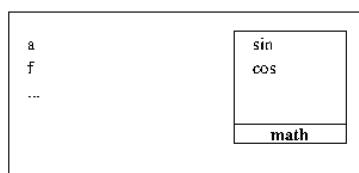
Enfin, on peut directement importer des fonctions ou des modules complets dans l'espace de noms courant

```
>>> from math import *
# ou
# >>> from math import sin,cos,pi
# pour importer seulement ces fonctions
```

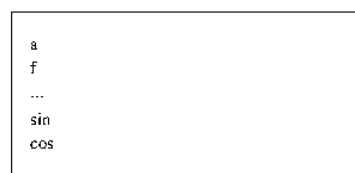
On a alors directement accès aux objets :

```
>>> sin(pi)
1.2246467991473532e-16
```

```
a=1
def f(x)
....
```



import math



from math import *

VI.1 | Module math

- les constantes : `pi`, `e`
- quelques fonctions classiques : `sqrt`, `exp`, `log`, `log10`, ainsi que `ceil`, `floor`, `fabs`, `factorial` (pour un entier)
- les fonctions trigonométriques diverses : `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- les fonctions hyperboliques : `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- quelques fonctions spéciales : `gamma`, `lgamma` ($\ln \circ \Gamma$), `erf` (avec $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$)
- quelques autres plus ou moins utilisées...

Il existe un module assez proche pour traiter le cas des fonctions d'une variable complexe (module `cmath`).

VI.2 | Autres modules usuels



Modules

<code>numpy</code>	pour le calcul numérique essentiellement basé sur le type tableau <code>array</code>
<code>matplotlib</code>	tout ce qui concerne l'affichage de graphiques avec plusieurs sous-modules (notamment <code>matplotlib.pyplot</code>)
<code>deepcopy</code>	permet la copie « profonde » de listes de listes...
<code>os</code>	pour l'interaction avec le système
<code>random</code>	pour l'aléatoire (on utilisera plutôt <code>numpy.random</code>)
<code>time</code>	pour mesurer le temps (obtenir l'heure, le temps d'exécution d'une fonction)



Chapitre 2 | Algorithmes, complexité et preuves

I | Algorithmes de première année

Au programme de première année, les algorithmes du programme sont les suivants :

I.1 | Algorithmes simples sur les listes

ou sur les chaînes de caractères (qui est alors vue comme la liste de ses caractères). On fera bien attention de distinguer les boucles sur les éléments et celles sur les positions des éléments (si on parcourt une liste par `for x in L`, la variable `x` va décrire les éléments de `L` les uns après les autres mais on n'a pas accès à la position de l'élément).

- **test d'appartenance à une liste :**

```
1 def test(L,x):
2     for e in L:
3         if e==x: return True
4         # on quitte la fonction dès qu'on a trouvé l'élément
5     # on est sorti de la boucle, on n'a donc pas trouvé
6     return False
```

ou

```
1 def test(L,x):
2     for i in range(len(L)):
3         if L[i]==x: return True
4     return False
```

Dans la première version on parcourt les *éléments* de la liste `L`, alors que dans le second on utilise un compteur entier `i` et on compare avec l'élément en position `i`

- **position d'un élément dans une liste :**

```
1 def position(L,x):
2     for i in range(len(L)):
3         if L[i]==x: return i
4     return -1
```

- **nombre d'occurrences d'un élément :**

```
1 def nombre(L,x):
2     c = 0 # compteur pour compter le nombre de termes
3     for e in L:
4         if x==e: c+=1
5     return c
```

- **recherche du maximum :**

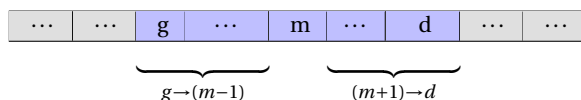
```
1 def maxi(L):
2     M = L[0] # on initialise au premier élément
3     for i in range(1,len(L)):
4         if L[i]>M: M=L[i] # on met à jour si l'élément est plus grand
5     return M
```

- **somme, moyenne des éléments, position du maximum...**

Remarque : on peut parcourir une liste avec les éléments et les positions en même temps : `for i,x in enumerate(L)` décrit la liste `L`, la première variable `i` est un compteur de position et la seconde `x` désigne l'élément actuel. Ce n'est pas au programme, donc à éviter.

I.2 | Recherche dichotomique dans une liste triée

On utilise deux compteurs `g` et `d` (gauche/droite) qui donne la portion de la liste à étudier. Pas de grosse difficulté si ce n'est qu'il faut gérer proprement les compteurs pour éviter les boucles infinies (on exclut l'élément milieu s'il n'est pas bon).



```
1 def dichot(L,x):
2     g = 0
3     d = len(L)-1
4     # g et d sont initialisés aux extrémités
5     while g<=d: # tant qu'il reste des cases à explorer
6         m = (g+d)//2
7         if L[m]==x: return m
8         if L[m]<x: g = m+1 # on explore de m+1 à d
```



```

9     if L[m]>x: d = m-1 # on explore de g à m-1
10    # on est sorti... pas trouvé
11    return -1

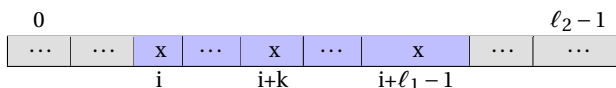
```

C'est très important de choisir $m+1$ ou $m-1$ sinon lorsque $g = d$, on aurait $m = g = d$ et on bouclerait sans fin si on choisit m au lieu de $m-1$ ou $m+1$.

I.3 | Recherche d'un mot dans une chaîne

On doit pouvoir programmer la recherche basique d'un mot dans une chaîne et compter le nombre d'occurrence d'un mot dans une chaîne (avec ou sans recouvrement possible) en testant position par position si le mot est présent.

- **Recherche d'un mot** : on utilise deux boucles imbriquées, la première (compteur i) pour la position du caractère de départ utilisé dans la chaîne et un second compteur (ici k) pour décrire l'avancement dans le mot :



Ce n'est pas la peine de chercher si $i + \ell_1 - 1$ est strictement supérieur à $\ell_2 - 1$ - on fait la recherche tant que $i + \ell_1 - 1 \leq \ell_2 - 1$, c'est-à-dire $i \leq \ell_2 - \ell_1$.

```

1 def recherche(mot, chaîne):
2     """
3     recherche de la première position d'un mot dans une chaîne
4     retourne None si le mot n'apparaît pas
5     """
6     l1=len(mot)
7     l2=len(chaîne)
8     i=0
9     while i<=l2-l1:
10        k=0
11        while (k<l1) and (chaîne[i+k]==mot[k]):
12            # tant qu'on ne dépasse pas et que les caractères correspondent, on avance
13            k+=1
14        if k==l1: # sortie car tous les caractères correspondent
15            return(i)
16        i+=1 # départ suivant
17    return(-1)

```

- **Nombre d'occurrences avec chevauchement** : c'est le plus simple : on parcourt les positions de départ les unes après les autres et on incrémente un compteur à chaque fois qu'on voit le mot. Il suffit de modifier la fonction précédente :

```

1 def occurrence(mot, chaîne):
2     l1=len(mot)
3     l2=len(chaîne)
4     i=0
5     c=0
6     while i<=l2-l1:
7         k=0
8         while (k<l1) and (chaîne[i+k]==mot[k]):
9             # tant qu'on ne dépasse pas et que les caractères correspondent, on avance
10            k+=1
11        if k==l1: # sortie car tous les caractères correspondent
12            c += 1
13        i+=1 # départ suivant
14    return(c)

```

On peut remplacer la boucle `while` par une boucle `for` puisqu'on sait qu'on parcourt toutes les positions les unes après les autres, jusqu'au bout.

- **Nombre d'occurrences sans chevauchement** : si on rencontre le mot on doit se placer après :

```

1 def occurrence(mot, chaîne):
2     l1=len(mot)
3     l2=len(chaîne)
4     i=0
5     c=0
6     while i<=l2-l1:
7         k=0
8         while (k<l1) and (chaîne[i+k]==mot[k]):
9             # tant qu'on ne dépasse pas et que les caractères correspondent, on avance
10            k+=1
11        if k==l1: # sortie car tous les caractères correspondent
12            c += 1
13            i = i+l1 # nouveau départ
14        else:
15            i+=1
16    return(c)

```

I.4 | Analyse numérique

Les quatre grandes familles d'algorithmes sont :

- intégration numérique (méthode des rectangles, des trapèzes),
- recherche de zéro : dichotomie et algorithme de Newton,
- résolution de systèmes linéaires (méthode du pivot de Gauss),
- intégration des équations différentielles (méthode d'Euler).

II | Complexité

II.1 | Généralités

Beaucoup de questions se posent pour évaluer la complexité d'un algorithme en fonction des données d'entrée :

- **taille des données d'entrée** : que prend-on en compte ? pour une chaîne de caractères, le nombre de caractères semble convenir. Pour un tableau, on pense à son nombre d'éléments mais ce n'est pas toujours suffisant : si c'est un tableau de chaînes de caractères, doit-on prendre le nombre de chaînes ou le nombre total de caractères ? De même pour un tableau d'entiers (et même pour un entier seul). Si les entiers sont codés sur un nombre fixé de bits, leur nombre suffit. Si les entiers deviennent de taille quelconque, leur taille rentre aussi en jeu. . . .
- **type de complexité** : temporelle (le temps que cela prend en fonction de la taille n des entrées) et/ou spatiale (la place mémoire totale utilisée par l'algorithme). On ne s'intéressera qu'à la complexité temporelle
- **valeur de la complexité** : minimale, maximale ou moyenne ? Si on note D_n l'ensemble des entrées possibles de taille n et $C(d)$ le coût pour $d \in D_n$ de l'algorithme, on a

$$C_{max} = \max \{C(d), d \in D_n\}, C_{min} = \min \{C(d), d \in D_n\} \text{ et } C_{moy} = \sum_{d \in D_n} C(d) \mu(d),$$

où $\mu(d)$ représente la probabilité d'avoir l'entrée d . On regardera essentiellement la complexité maximale (ce qui peut se passer dans le pire des cas) et, lorsque c'est possible, la complexité moyenne (souvent difficile à calculer mais intéressante lorsqu'on doit utiliser l'algorithme un grand nombre de fois).

II.2 | Exemple et notations

def maxi(L) :	cout	nombre
n = len(L)	C_1	1
M = L[0]	C_2	1
for i in range(1,n):	C_3	$n-1$
if L[i]>M:	C_4	$n-1$
M = L[i]	C_5	k
return(M)	C_6	1

Sur cet exemple, le coût total pour une liste à n éléments est $C(n) = C_1 + C_2 + C_6 + (C_3 + C_4) * (n-1) + C_5 * k$ où k varie de 0 à $n-1$. On a un coût sous la forme $C(n) = A + kB + nC$. Comme on ne connaît pas les constantes ni la valeur de k , on ne peut donner qu'une complexité asymptotique. La plupart du temps l'ordre de la complexité suffit. On note (tout est positif ici)

- $C(n) = O(f(n))$ lorsqu'il existe une constante K et $n_0 \in \mathbb{N}$ telle que $C(n) \leq K \cdot f(n)$ pour $n \geq n_0$ (majoration de la complexité asymptotique),
- $C(n) = \Omega(f(n))$ lorsqu'il existe une constante K et $n_0 \in \mathbb{N}$ telle que $C(n) \geq K \cdot f(n)$ pour $n \geq n_0$ (minoration de la complexité asymptotique),
- $C(n) = \theta(f(n))$ lorsqu'il existe des constantes K_1, K_2 et $n_0 \in \mathbb{N}$ telle que $K_1 f(n) \leq C(n) \leq K_2 f(n)$ pour $n \geq n_0$ (encadrement de la complexité asymptotique),
- on peut évidemment utiliser les équivalents si on veut être précis sur une complexité (par exemple le nombre de comparaisons, le nombre d'affectations. . .)

Dans l'exemple précédent, on a $C(n) = \theta(n)$. Au programme, on s'intéresse essentiellement à une majoration, d'où $C(n) = O(n)$ ici.

II.3 | Ordres de grandeur

On peut par exemple mesurer les rapidités de calcul en f1ops (floating point operations per second) - un bon PC actuel étant de l'ordre de 10^{11} f1ops. Prenons seulement 10^9 comme exemple

nom	ordre	$n = 10^2$	$n = 10^3$	$n = 10^6$
logarithmique	$\log n$	immédiat	-	-
linéaire	n	-	-	1 ms
quasi-linéaire	$n \log n$	-	-	20 ms
quadratique	n^2	-	-	15 min
cubique	n^3	-	1s	30 ans
exponentielle	a^n	10^{15} ans avec 2^n	/	/

Pour des algorithmes exponentiels (complexité en 2^n par exemple), on peut difficilement dépasser $n = 30$ ou 40. . . c'est peu.



III | Preuves de programmes

Pour prouver qu'une boucle réalise bien toujours ce qu'on espère :

- on détermine un **invariant de boucle**, c'est-à-dire une propriété qui est vraie à chaque passage de la boucle. En général, cette propriété est la description du contenu des variables importantes lors des différents passages dans la boucle. La propriété peut ou non fait apparaître le nombre de passage dans la boucle.
- on prouve la **terminaison** : on doit sortir de la boucle. C'est immédiat sur une boucle non conditionnelle `for` (nombre d'étapes déterminé) mais parfois plus difficile sur une boucle conditionnelle `while` - la plupart du temps on explicite un variant de boucle : une quantité *entière et strictement décroissante* qui doit rester supérieure à une valeur pour ne pas sortir.
- on prouve la **correction** de la boucle : on vérifie que la sortie correspond à ce qu'on veut

III.1 | Boucle non conditionnelle : calcul de $n!$

Algorithme 1 : Calcul de $n!$

Données : n entier positif ou nul

Sorties : $n!$

début

$p \leftarrow 1$

pour i de 1 à n **faire**

$p \leftarrow p * i$

retourner p

Preuve de l'algorithme :

- si $n = 0$, on ne rentre pas dans la boucle et on renvoie $1 = 0!$. On suppose $n \geq 1$ pour la suite.
- On note p_k le contenu de p à la fin du k -ième passage, avec $p_0 = 1$. On a l'invariant « $p_k = k!$ ». En effet, c'est vrai pour $k = 0$. Si $p_k = k!$ alors au passage suivant dans la boucle, le compteur i prend la valeur $k + 1$ et $p_{k+1} = p_k * (k + 1) = (k + 1)!$.
- la terminaison est immédiate puisque la boucle est une boucle `for` avec n étapes.
- correction : on effectue n passages dans la boucle. À la sortie $p = p_n = n!$.

III.2 | Boucle conditionnelle : division euclidienne

Algorithme 2 : Division euclidienne

Données : a entier positif, b entier strictement positif

Sorties : q, r quotient et reste de la division euclidienne de a par b

début

$q \leftarrow 0$

$r \leftarrow a$

tant que $r \geq b$ **faire**

$r \leftarrow r - b$

$q \leftarrow q + 1$

retourner (q, r)

On note

- q_0 et r_0 la valeur de q et r avant le premier passage dans la boucle
- q_i et r_i la valeur de q et r après le i -ème passage dans la boucle.

Première preuve de l'algorithme :

- invariants : « $q_i = i, r_i = r - i * b$ ».
→ la propriété est vraie pour $i = 0$: on a $q_0 = 0$ et $r_0 = a$.
→ la propriété se conserve : si on a effectué i passages dans la boucle et qu'on rentre dans un passage supplémentaire alors $r_i \geq b$.
On a alors $q_{i+1} = q_i + 1 = i + 1$, ainsi que $r_{i+1} = r_i - b = r - i * b - b = r - (i + 1) * b$.
- terminaison : la suite (r_n) est strictement décroissante car $b > 0$ donc il existe i tel que $r_i < b$.
- correction : si n désigne le nombre de passages dans la boucle, alors $q_n = n, r_n = a - q_n b$ et $r_n < b$ puisqu'on est sorti de la boucle. On a bien $a = b q_n + r_n$ avec $r_n < b$ mais rien ne donne $r_n \geq 0$. Pour cela il faudrait dire qu'on n'était pas sorti avant donc $r_{n-1} \geq b$ et ainsi $r_n \geq 0$... enfin sauf s'il n'y a pas d'étape $n - 1$ car alors... bon on préfère passer à la seconde version en modifiant l'invariant.

Seconde preuve de l'algorithme :

- invariants : « $q_i = i, r_i = r - i * b$ et $r_i \geq 0$ ».
→ la propriété est vraie pour $i = 0$: on a $q_0 = 0$ et $r_0 = a$ et $r_0 \geq 0$.
→ la propriété se conserve : si on a effectué i passages dans la boucle et qu'on rentre dans un passage supplémentaire alors $r_i \geq b$.
On a alors $q_{i+1} = q_i + 1 = i + 1$, ainsi que $r_{i+1} = r_i - b = r - i * b - b = r - (i + 1) * b$ et enfin $r_{i+1} = r_i - b \geq b - b = 0$.



- terminaison : la suite (r_n) est strictement décroissante car $b > 0$ donc il existe i tel que $r_i < b$.
- correction : si n désigne le nombre de passages dans la boucle, alors $q_n = n$, $r_n = a - q_nb$, $r_n \geq 0$ et $r_n < b$ puisqu'on est sorti de la boucle. On a bien $a = bq_n + r_n$ avec $0 \leq r_n < b$.

Troisième preuve de l'algorithme :

- On n'a pas besoin de connaître complètement le contenu de toutes les variables. On propose les invariants : « $a = bq_i + r_i$ et $r_i \geq 0$ ».
→ la propriété est vraie pour $i = 0$: on a $q_0 = 0$ et $r_0 = a$ donc $bq_0 + r_0 = a$ et $r_0 \geq 0$.
→ la propriété se conserve : si on a effectué i passages dans la boucle et qu'on rentre dans un passage supplémentaire alors $r_i \geq b$.
On a alors $r_{i+1} = r_i - b$ et $q_{i+1} = q_i + 1$ d'où $bq_{i+1} + r_{i+1} = b(q_i + 1) + (r_i - b) = bq_i + r_i = a$ et $r_{i+1} \geq 0$ car $r_i \geq b$.
- terminaison : la suite (r_n) est strictement décroissante car $b > 0$ donc il existe i tel que $r_i < b$.
- correction : si n désigne le nombre de passages dans la boucle, alors $bq_n + r_n = a$ et $0 \leq r_n < b$ puisqu'on n'est pas rentré dans la boucle après le passage n .

III.3 | Exercices

▷ Exercice II.1 : Suite de Fibonacci

On définit la suite (u_n) par $u_0 = 1$, $u_1 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$. On propose l'algorithme suivant :

Algorithme 3 : Calcul de u_n

Données : n entier positif ou nul

Sorties : u_n terme d'indice n de la suite de Fibonacci

début

$a \leftarrow 1$

$b \leftarrow 1$

pour i de 1 à n **faire**

$a, b \leftarrow b, a + b$

retourner b

Cet

algorithme calcule-t-il ce qu'il est censé calculer dans toutes les situations ? est-il correct ?

▷ Exercice II.2 : Évaluation d'un polynôme

On se donne un polynôme $P = \sum_{k=0}^n a_k X^k$ et veut l'évaluer en x . On propose deux algorithmes.

- le premier se décompose en deux fonctions :

Algorithme 4 : $expo(x, k)$: calcul de x^k

Données : x réel, k entier positif ou nul

Sorties : la valeur de x^k

début

$p \leftarrow 1$

pour i de 1 à k **faire**

$p \leftarrow p * x$

retourner p

Algorithme 5 : Évaluation d'un polynôme en x

Données : un tableau de coefficient $a = [a_0, a_1, \dots, a_n]$ et un réel x

Sorties : la valeur de $\sum_{k=0}^n a_k x^k$

début

$s \leftarrow 0$

pour i de 0 à n **faire**

$s \leftarrow s + a[i] * expo(x, i)$

retourner s

Justifier que les deux fonctions sont correctes. Déterminer le nombre d'opérations (multiplications et additions) pour cette évaluation. Quel est l'ordre de grandeur ?

- **Algorithme de Hörner** : on remarque que

$$P(x) = (((a_n \times x + a_{n-1}) \times x) + a_{n-2}) \times x \dots + a_1) \times x + a_0,$$

on commence par $s = a_n$, puis $s = s.x + a_{n-1} \dots$. Écrire un algorithme effectuant ce calcul, prouver qu'il est correct et étudier sa complexité.

▷ Exercice II.3 : PGCD

Écrire et justifier un algorithme déterminant le pgcd d de 2 entiers p et q , puis un algorithme calculant une relation de Bezout $d = up + vq$.



Chapitre 3 | Piles et récursivité

Il est naturel d'utiliser les listes Python qui servent à peu près à tout... hélas pas mal de difficultés cachées.

I | Listes et tableaux

On distingue essentiellement deux types d'objets :

- les tableaux dont la taille est fixe
- les listes (dynamiques) dont le nombre d'objets est variable

De plus, les éléments peuvent être homogènes (tous de même type) ou hétérogènes (types différents et donc chaque élément a une taille mémoire différente).

I.1 | Tableaux homogènes

C'est la structure la plus simple : le nombre de cases est fixé et chaque case prend la même place. Cela permet de réserver un espace mémoire suffisant et surtout contigu pour stocker les objets. On a alors les possibilités suivantes :

- accès simple et rapide à l'élément en position k : l'adresse est $\text{debut} + k * \text{taille}$, aussi bien en lecture qu'en écriture
- pas de possibilité d'insertion, d'ajout, de suppression (on peut insérer en décalant les éléments mais il faut alors les déplacer les uns après les autres, éventuellement en supprimant la queue)

les tableaux hétérogènes sont moins intéressants car chaque élément a une taille différente donc il n'y a pas de formule simple pour connaître l'emplacement de l'élément k (si ce n'est d'avancer case par case en fonction de leurs tailles).

I.2 | Listes simplement chaînées

On dispose de l'adresse de début de la liste, chaque élément est séparé en deux parties : la donnée et l'adresse de l'élément suivant.

- on n'a pas de limitation de taille (à part la mémoire), pas de déclaration de taille à faire
- on a facilement accès à l'élément de début
- pour aller à l'élément k , on doit parcourir tous les éléments précédents
- on peut facilement insérer et supprimer un élément en tête de liste
- on peut insérer un élément à une position une fois qu'on est à cette position mais on ne peut pas supprimer au milieu (on n'a pas l'adresse précédente)

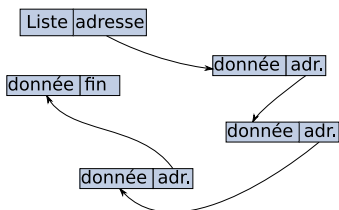


FIGURE 3.1 – Liste chaînée

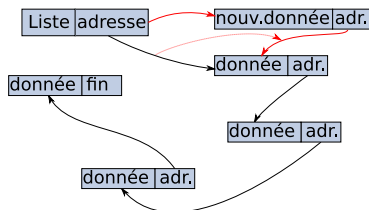


FIGURE 3.2 – Insertion au début

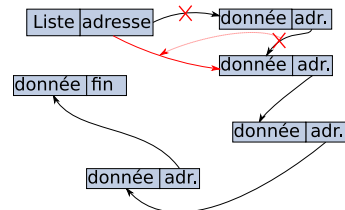


FIGURE 3.3 – Suppression au début

I.3 | Listes doublement chaînées

C'est le même principe mais chaque élément dispose en plus de l'adresse de l'élément précédent (et l'élément précédent le premier élément est le dernier élément : on a alors un cycle).

- on n'a pas de limitation de taille (à part la mémoire), pas de déclaration de taille à faire
- on a facilement accès à l'élément de début et de fin
- pour aller à l'élément k , on doit parcourir tous les éléments précédents
- on peut facilement insérer et supprimer un élément en tête et en queue de liste
- on peut insérer et supprimer un élément à une position une fois qu'on est à cette position

I.4 | En Python

le type `list` en Python est dynamique. L'opération essentielle est d'accéder rapidement aux éléments, ce qui est facile pour les tableaux homogènes. Le choix est le suivant :

- on utilise un tableau d'adresses (ainsi les tailles sont fixes) d'une certaine taille. Un élément est alors une adresse vers l'objet correspondant. Cela permet d'accéder en lecture et écriture à un élément quelconque rapidement



- lorsqu'on veut ajouter un élément en fin de liste : soit il reste des cases vides dans le tableau d'adresses, dans ce cas pas de difficulté ; soit tout le tableau est rempli et dans ce cas on alloue une zone mémoire plus grande, on recopie le tableau dans cette nouvelle zone et on ajoute l'élément (si on ajoute une fois, il y a des chances qu'on ajoute plusieurs fois. Plutôt que d'ajouter une seule case au tableau, on augmente sa taille suffisamment pour des insertions futures - ni trop pour ne pas gaspiller de mémoire, ni trop peu pour ne pas avoir à le refaire souvent). La complexité de l'insertion est alors presque constante (on dit en temps amorti constant).
- Supprimer le dernier élément est facile, supprimer un élément intermédiaire beaucoup plus long car il faut décaler toutes les cases... Pour concaténer deux listes, on en crée une suffisant longue et on copie les adresses (complexité de l'ordre de la taille totale).

II | Piles

II.1 | Généralités

C'est une structure de données dont les opérations sont les suivantes :

- créer une pile vide,
- savoir si la pile est vide,
- ajouter un objet en sommet de pile,
- récupérer et supprimer l'objet en haut de la pile

Les exemples usuels : une pile d'assiettes, l'historique d'un navigateur... On appelle aussi cela des piles LIFO (last in, first out). La structure de données de liste simplement chaînée se prête parfaitement à la réalisation d'une pile.

II.2 | En Python

On se doute bien qu'il doit y avoir un module qui permet de manipuler les piles en Python... on se contentera des listes dynamiques pour les représenter avec comme contraintes

- on ajoute un élément avec la méthode `append` : `L.append(x)`
- on dépile le dernier élément avec `pop` : `x = L.pop()` (retire le sommet de la pile et le renvoie),
- on peut tester si la pile est vide : `L == []` (ou `len(L)==0`)
- éventuellement (ça dépend), on peut lire le sommet de la pile sans le dépiler (mais cela peut se faire avec un dépilement/empilement)
- on interdit tout le reste (retirer un élément quelconque, demander la taille de la liste, modifier un élément en position quelconque)

On pourra éventuellement créer des fonctions correspondantes pour des raisons de clarté :

```
1 def est_vide(L):
2     if len(L)==0:
3         return True
4     else:
5         return False
6
7 def empiler(L,x):
8     L.append(x)
9
10 def depiler(L):
11     return L.pop()
```

on pourra également créer une fonction d'affichage.

Remarque : on peut aussi simplement écrire

```
1 def est_vide(L):
2     return (len(L)==0)
```

qui renvoie le résultat booléen du test.

II.3 | Pile d'exécution / Pile d'appel

Lors de l'appel d'une fonction en Python, un ensemble de données est stocké sur une pile (appelée pile d'exécution) : arguments de la fonction, adresse de retour une fois la fonction terminée, de la place pour les variables locales, pour la valeur de retour... cela permet de gérer l'appel à une fonction et aussi l'appel multiple de fonctions.

II.4 | Complément : programmation objet

On peut présenter le principe de la création d'un nouveau type d'objet en Python. Pour cela on crée une nouvelle classe générique avec le mot clé `class`. On disposera alors de cette classe générique et d'objets de ce type (qu'on appelle une instance de la classe) qu'on peut construire de plusieurs façons. On se contentera de créer une pile vide ou à partir d'une liste. Au lieu de créer des fonctions qui prennent un argument qu'on espère être une liste, on va créer ce nouveau type qui va contenir :

- une méthode pour créer un objet de cette classe (le constructeur),
- différentes données qui seront affectées pour chaque instance de la classe (chaque objet) - on peut aussi créer des données communes à la classe, ce qu'on ne fera pas,
- des méthodes (fonctions) qui vont agir sur les instances. On a besoin dans ces fonctions d'avoir une variable qui représente l'objet actuel et pour cela on dispose du mot clé `self`

Un exemple de classe pile :

```

1 class pile():
2     """
3     Une classe pile assez basique, gérée par une liste dynamique
4     """
5     def __init__(self, liste=[]):
6         self.L = liste.copy()
7
8     def est_vide(self):
9         if len(self.L)==0:
10             return True
11         else:
12             return False
13
14     def empiler(self,x):
15         self.L.append(x)
16
17     def depiler(self):
18         try:
19             # on essaie d'exécuter les instructions suivantes
20             x = self.L.pop()
21             return x
22         except IndexError:
23             # si cela donne un message d'erreur IndexError, alors on l'intercepte et on affiche
24             print("Pile vide")
25             return None
26
27     def afficher(self):
28         print("---")
29         l = len(self.L)
30         for i in range(l):
31             print(self.L[l-i-1])
32         print("---")

```

on a :

- la définition de la classe : `class pile()` :
- un constructeur : `def __init__(self, liste=[]):` avec deux arguments. Le premier `self` (toujours - on ne l'utilise pas lorsqu'on appelle la fonction) et le second un paramètre transmis, vide par défaut si on ne transmet rien, qui est la liste servant à initialiser une pile. On l'utilise alors sous la forme

```

>>> P = pile() # crée une pile vide
>>> Q = pile([1,4,2]) # crée une pile contenant 3$ éléments

```

Lorsqu'on l'appelle, on objet de type pile est crée et dans cet objet, une variable L est crée avec le contenu de la pile (on y accède avec `self.L`)

- les méthodes `est_vide`, `empiler`, `depiler` : elles utilisent alors la liste `self.L` (et une dernière méthode pour l'affichage).

```

>>> from pile import *
>>> P = pile()
>>> P.empiler(3)
>>> P.empiler(2)
>>> P.afficher()
---
2
3
---
>>> P.depiler()
2
>>> P.est_vide()
False
>>> P.afficher()
---
3
---

```

Remarque : si on importe le module sous la forme `import pile`, on n'oublie pas qu'il est importé dans l'espace de nom `pile`. Pour une nouvelle pile, on doit utiliser `P = pile.pile()` - ensuite rien ne change

III | Récursivité

III.1 | Généralités

Il arrive fréquemment qu'une fonction s'appelle elle-même avec d'autres valeurs des arguments - cela ne pose pas de problème grâce à la pile d'appels. Le principe générale d'une fonction récursive est la suivante :

```

1 def f(x):
2     # traitement des cas terminaux immédiats
3     if x==...:
4         return val1
5     if x==...:

```



```

6     return val2
7     # autres situations
8     # avec des calculs faisant appel à f(.)
9     return ...

```

III.2 | Exemple : calcul de puissances

On est souvent proche de la définition mathématique pour la fonction. Par exemple pour $a \in \mathbb{R}$ et $n \in \mathbb{N}$, on a

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a * a^{n-1} & \text{si } n \geq 1 \end{cases}$$

ce qui donne

```

1 def puissance(a,n):
2     if n==0:
3         return 1
4     else:
5         return a*puissance(a,n-1)

```

La preuve est simple à réaliser, elle se fait par récurrence sur n . On note $\mathcal{P}(n)$: « pour tout a , puissance(a,n) renvoie a^n ».

1. Pour $n = 0$, puissance(a,n) renvoie 1.
2. Soit $n \in \mathbb{N}^*$ tel que \mathcal{P} est vraie jusqu'au rang $n-1$, alors l'appel à puissance(a,n) rentre dans le else (puisque $n \in \mathbb{N}^*$), et retourne $a * \text{puissance}(a, n-1) = a * a^{n-1} = a^n$.
3. Par récurrence, la fonction est correcte.

Évidemment ce n'est pas toujours aussi simple.

Concernant la complexité : si on note $T(n)$ la complexité du calcul de a^n , on a une relation de récurrence $T(n) = T(n-1) + C$ où C est constant (à peu près...), d'où une suite arithmétique et $T(n) = O(n)$. On peut grandement améliorer cela par différents algorithmes (exponentiation rapide en TD par exemple).

III.3 | Suite de Fibonacci

Première version

On a $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$ pour $n \in \mathbb{N}$. On en déduit une première version :

```

1 def fibo(n):
2     if (n==0) or (n==1):
3         return 1
4     else:
5         return fibo(n-1)+fibo(n-2)

```

Pour la preuve, cela se fait par récurrence comme précédemment. La complexité est plus grande... on va compter le nombre d'additions qu'on réalise. On note A_n ce nombre d'additions pour calculer u_n . On a $A_0 = A_1 = 0$ puis $A_n = 1 + A_{n-1} + A_{n-2}$, ce qui peut se réécrire $(1 + A_n) = (1 + A_{n-1}) + (1 + A_{n-2})$. En posant $a_n = 1 + A_n$, on a $a_0 = a_1 = 1$ et $a_n = a_{n-1} + a_{n-2}$ d'où $a_n = u_n = \theta(\rho^n)$ où $\rho = \frac{1+\sqrt{5}}{2}$. La complexité est exponentielle!!! de plus le nombre d'additions correspond aussi, à peu près, au nombre d'appels de la fonction... la pile d'appel va exploser dès qu'on veut calculer les termes aux delà du 30ème ou 40ème rang.

Seconde version : non récursive

On peut calculer les termes de proche en proche par couple puisqu'on a besoin de deux termes pour avoir le suivant : $(u_n, u_{n+1}) \rightarrow (u_{n+1}, u_{n+2} = u_n + u_{n+1})$:

```

1 def fibo(n):
2     a,b = 1,1
3     for i in range(n):
4         a,b = b,a+b
5     return a

```

La preuve : à la fin du passage k , on a $a_k = u_k$ et $b_k = u_{k+1}$. La complexité est cette fois linéaire.

Troisième version : récursive linéaire

Même principe : si on sait calculer le couple (u_n, u_{n+1}) , on sait facilement calculer le suivant. On crée une fonction fibo(n) qui renvoie le couple (u_n, u_{n+1})

```

1 def fibo(n):
2     if n==0:
3         return (1,1)
4     # on n'a pas le choix, on doit systématiquement renvoyer un couple
5     else:
6         a,b = fibo(n-1)
7         return b,a+b

```

et on récupère le premier élément de la liste (on prouve par récurrence que `fibonacci(n)` renvoie le couple (u_n, u_{n+1})). Si on note A_n le nombre d'additions, on a immédiatement $A_n = A_{n-1} + 1$. La complexité est linéaire.

Troisième version bis

On veut quand même récupérer u_n et pas un couple... on peut le faire en deux fonctions

```
1 def fibo_rec(n):
2     if n==0:
3         return (1,1)
4         # on n'a pas le choix, on doit systématiquement renvoyer un couple
5     else:
6         a,b = fibo_rec(n-1)
7         return b,a+b
8 def fibo(n):
9     u = fibo_rec(n)
10    return u[0]
```

mais cela crée une fonction `fibo_rec` dans l'espace de noms usuels ce qui n'est pas satisfaisant. On va donc cacher cette fonction dans la fonction principale.

```
1 def fibo(n):
2     def fibo_rec(n):
3         if n==0:
4             return (1,1)
5         else:
6             a,b = fibo_rec(n-1)
7             return b,a+b
8     u = fibo_rec(n)
9     return u[0]
```



Chapitre 4 | Tris

I | Généralités

I.1 | Présentation

On cherche à trier une liste de données (pour une relation d'ordre totale). On peut éventuellement faire la théorie sur des tableaux d'entiers (quitte à associer une clé entière à un objet) et même se limiter à trier un tableau d'entiers de 0 à $n - 1$ si on veut encore plus simplifier). Plusieurs remarques sur les tris :

- certains se font en ne manipulant que le tableau et ses éléments (tri *en place*) et d'autres en créant des tableaux auxiliaires,
- on a certains tris qui sont dits *stables* : lorsque 2 données ont la même clé, elles sont encore dans le même ordre après le tri (par exemple pour faire un tri d'abord sur le prénom puis sur le nom pour conserver l'ordre sur les prénoms lorsqu'on a un même nom)
- la complexité maximale est évidemment importante, mais la complexité moyenne prend son intérêt lorsqu'on doit réaliser beaucoup de tris,
- certains tris sont déterministes (à chaque fois qu'on l'appelle sur un même jeu de données, l'algorithme se déroule de la même façon) ou non-déterministes (ou stochastiques, ou aléatoires) lorsque le déroulement fait apparaître un choix aléatoire (évidemment le résultat est le même en sortie).

I.2 | Les tris au programme

Les tris qu'on va présenter :

- tri par sélection** : on détermine la position du (ou d'un) plus petit élément de la liste et on l'échange avec le premier élément. On recommence avec les $n - 1$ derniers éléments...
- tri bulle** : on fait remonter le plus grand élément par échanges consécutifs puis on recommence avec les $n - 1$ premiers éléments...
- tri par insertion** : on insère les éléments les uns après les autres dans la liste des précédents qu'on a trié
- tri fusion** : on sépare en 2 listes, on les trie et on les fusionne
- tri rapide - quick sort** : on choisit un élément, on sépare en deux listes (les inférieurs et les supérieurs) qu'on trie séparément puis qu'on concatène.

II | Les tris simples

Dans tous ces algorithmes, on a une liste L de taille n d'éléments à trier (en général des entiers, des flottants, des chaînes de caractères).

II.1 | Tri par sélection

Principe

- On parcourt toute la liste pour déterminer la position k du minimum (ou d'un minimum s'il y en a plusieurs),
- on échange les éléments en position k et celui en début de liste,
- on recommence avec la liste des $n - 1$ derniers éléments.

Remarque : on peut évidemment faire avec le maximum et échanger avec le dernier élément.

Algorithme

Algorithme 6 : Tri par sélection

Données : L une liste à trier

Sorties : vide, la liste L est triée sur place

début

```
n ← len(L)
pour i de 0 à n - 2 faire
    p ← i
    pour j de i + 1 à n - 1 faire
        si L[j] < L[p] alors p ← j
    L[p] ↔ L[i]
```

```
1 def tri_select(L):
2     n = len(L)
3     for i in range(n-1):
4         p = i
5         for j in range(i+1, n):
6             if L[j] < L[p]: p = j
7         L[i], L[p] = L[p], L[i]
```

**Commentaires, preuve et complexité**

- pas très difficile... à savoir refaire rapidement (bien réfléchir sur les bornes des boucles)
- pour $i = 0$, on a j qui va de 1 à $n - 1$ et il y a $n - 1$ comparaisons, puis $n - 2$ lorsque $i = 1$... Le nombre total de comparaisons est donc $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$ quelle que soit la liste
- le tri n'est pas stable. Par exemple la liste $[1_1, 1_2, 0]$ est triée en $[0, 1_2, 1_1]$
- pour la preuve, on doit prouver deux invariants, l'un pour la boucle externe et l'autre pour la boucle interne. Pour la boucle externe, on peut prendre, à la fin du passage i :

$$\llcorner L[0] \leq L[1] \cdots \leq L[i-1] \leq L[i], L[i+1], \dots, L[n-1] \gg$$

c'est-à-dire que les i premiers éléments sont inférieurs aux suivants et ils sont triés. Pour la boucle intérieure, on a « pour tout $k \in [i; j]$, $L[p] \leq L[k]$ (le terme en position p est le minimum des termes entre i et j).

II.2 | Tri bulle**Principe**

- On fait remonter le plus grand élément en échangeant $L[i]$ et $L[i+1]$ s'ils sont inversés avec i allant de 0 à $n - 2$,
- on recommence avec la liste des $n - 1$ premiers éléments.

Algorithme**Algorithme 7 : Tri bulle**

Données : L une liste à trier

Sorties : vide, la liste L est triée sur place

début

```

     $n \leftarrow \text{len}(L)$ 
    pour  $i$  de 1 à  $n - 1$  faire
         $p \leftarrow i$ 
        pour  $j$  de 0 à  $n - 1 - i$  faire
            si  $L[j+1] < L[j]$  alors  $L[j+1] \leftrightarrow L[j]$ 
```

```

1 def tri_bulle(L):
2     n = len(L)
3     for i in range(1, n):
4         for j in range(0, n-i):
5             if L[j+1] < L[j]: L[j], L[j+1] = L[j+1], L[j]
```

Commentaires, preuve et complexité

- de nouveau on a toujours $\frac{n(n-1)}{2}$ comparaisons, en revanche on a beaucoup plus d'échanges que pour le tri par sélection
- le tri est stable
- Pour compter le nombre d'échanges : on utilise la notion d'inversion (comme pour les permutations) :
 - le couple (i, j) présente une inversion lorsque $i < j$ et $L[i] > L[j]$.
 - en échangeant deux éléments consécutifs, on supprime une et une seule inversion. À la fin la liste est triée après k échanges ainsi k est le nombre d'inversions de la liste de départ,
 - on peut calculer le nombre moyen d'inversions sur l'ensemble des permutations des entiers de 1 à n : si σ est une permutation avec k inversions, alors la permutation symétrique σ' possède le nombre complémentaire d'inversions à savoir $\frac{n(n-1)}{2} - k$ (le terme $\frac{n(n-1)}{2}$ est le nombre de couples (i, j) avec $i < j$). En regroupant les permutations 2 par 2 de cette manière, on a

$$\frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \text{Inv}(\sigma) = \underbrace{\frac{1}{n!}}_{\text{poids d'une permutation}} \underbrace{\frac{n!}{2}}_{\text{nombre de bicoules}} \underbrace{\frac{n(n-1)}{2}}_{\text{somme des nombres inversions sur un bicouple}} = \frac{n(n-1)}{4}.$$

- pour la preuve... trouver les invariants de boucles

II.3 | Tri par insertion**Principe**

- On commence avec le premier élément,
- on regarde le second élément et on le descend jusqu'à sa place,
- on continue jusqu'au dernier élément.

On va programmer une version « en place ».

Par exemple, sur la liste suivante, les 4 premiers éléments ont été triés. On s'intéresse au cinquième qui vaut 4.

1	8	11	15	4	9	11	17
---	---	----	----	---	---	----	----

1	8	11	4	15	9	11	17
---	---	----	---	----	---	----	----

1	8	4	11	15	9	11	17
---	---	---	----	----	---	----	----

1	4	8	11	15	9	11	17
---	---	---	----	----	---	----	----

En pratique, on le réalise plutôt comme cela : on stocke sa valeur dans une variable, on fait remonter d'un cran les éléments précédents, tant qu'ils valent plus que 4 :

1	8	11	15 →	15	9	11	17
---	---	----	------	----	---	----	----

1	8	11 →	11	15	9	11	17
---	---	------	----	----	---	----	----

1	8 →	8	11	15	9	11	17
---	-----	---	----	----	---	----	----

et enfin on place l'élément conservé au bon endroit

1	4	8	11	15	9	11	17
---	---	---	----	----	---	----	----

Algorithme

```

1 def tri_insert_n(L):
2     n = len(L)
3     for i in range(1,n):           # position de l'élément à insérer
4         x = L[i]                   # on stocke l'élément
5         j = i-1                    # élément à gauche
6         while j>=0 and L[j]>x:      # tant que l'élément est supérieur
7             L[j+1]=L[j]            # on le décale
8             j = j-1                # on passe au précédent
9         L[j+1] = x                 # on place x au bon endroit

```

Commentaires, preuve et complexité

- le nombre maximal de comparaisons est $\frac{n(n-1)}{2}$ (liste triée à l'envers) et le minimal est $n-1$ lorsque la liste est déjà triée. Le nombre moyen semble plus difficile à obtenir.

III | Les tris évolués

Ces deux tris rentrent dans la catégorie des algorithmes « diviser pour régner » (ou « divide and conquer ») : on scinde les données en deux morceaux (si possible les plus équilibrés possibles mais ce n'est pas aussi simple) pour avoir des listes bien plus petites.

IV | Tri fusion

Cet algorithme est important pour le principe de la fusion.

IV.1 | Principe

- on scinde la liste L en deux sous listes L_1 et L_2 (au milieu),
- on trie les deux listes L_1 et L_2 (récursivement avec la condition d'arrêt lorsque la liste comporte au plus un élément),
- on fusionne les deux listes : on dispose d'un compteur de position pour chacune des listes et on a simplement à comparer chacun des éléments en ces positions (on peut voir cela comme deux piles triées - à chaque étape on regarde les deux sommets en prenant le plus petit des 2 piles).

Algorithme

On présente tout d'abord la fusion de deux listes triées (c'est un présupposé) :

```

1 def fusion(L1,L2):
2     n = len(L1)
3     m = len(L2)
4     L = [0]*(m+n)
5     i, j = 0, 0
6     for k in range(m+n):
7         if i==n:                   # on a épuisé la liste L1
8             L[k] = L2[j]
9             j += 1
10        elif j==m:                 # on a épuisé la liste L2
11            L[k] = L1[i]

```



```

12     i += 1
13     elif L1[i] < L2[j]: # les deux listes sont 'non vidées'
14         L[k] = L1[i] # on choisit le plus petit des 2 sommets
15         i += 1
16     else:
17         L[k] = L2[j]
18         j += 1
19     return(L)

```

l'algorithme complet serait alors

```

1 def tri_fusion(L):
2     if len(L) <= 1: return L
3     n = len(L)
4     L1 = tri_fusion(L[:n//2])
5     L2 = tri_fusion(L[n//2:])
6     return fusion(L1, L2)

```

Complexité

Le fait de couper en 2 récursivement nous incite pour commencer à s'intéresser au cas où n est une puissance de 2. On note C_n la complexité pour trier une liste de n éléments (nombre de tests et d'affectations par exemple). On obtient alors $C_{2^{k+1}} \leq 2C_{2^k} + A2^{k+1}$: le $A2^{k+1}$ correspond à la fusion des deux sous-listes qui est linéaire en le nombre total de termes. Si on note $u_k = \frac{C_{2^k}}{2^k}$, on obtient, en divisant la relation précédente par 2^{k+1} , $u_{k+1} \leq u_k + A$ ce qui donne $u_k \leq u_0 + Ak$ et ainsi $C_{2^k} = O(k2^k)$ ou encore $C_n = O(n \log n)$ lorsque $n = 2^k$. Pour n quelconque, on peut encadrer $2^{k-1} < n \leq 2^k$ et se convaincre raisonnablement que $C_n \leq C_{2^k}$ (par exemple on peut compléter la liste en ajoutant des termes plus grands que tous les autres au bout pour se ramener à une longueur 2^k). On obtient alors $C_n = O(n \log n)$ dans tous les cas.

Amélioration

Pour éviter de passer du temps à recopier des listes notamment lors de la phase de séparation, on peut essayer de réaliser un algorithme modifiant la liste (mais toujours avec une liste intermédiaire dans la fonction) en spécifiant non pas les deux sous listes mais la liste complète et trois entiers $a \leq c < b$ qui désignent les extrémités de la liste à traiter et le point de séparation - on crée une liste temporaire pour y placer les éléments triés qu'on recopie ensuite dans L entre les positions a et b ... à faire en TD.

IV.2 | Tri rapide

Principe

C'est de nouveau un algorithme récursif qui à l'avantage de pouvoir se faire en place.

- on choisit un élément pivot x dans la liste,
- on sépare la liste en L1 celle des éléments strictement inférieurs et L2 contenant ceux qui sont supérieurs ou égaux (on exclut le x choisi afin d'être certain d'avoir des sous-listes de taille strictement inférieure),
- on trie ces 2 sous-listes et on concatène L1, x , L2

Commentaires, complexité

- la première question est de savoir comment choisir le pivot : l'élément de début, de fin, central ou un élément aléatoire. Si les listes sont totalement aléatoires, le fait de choisir le premier ou le dernier (ou le milieu) donnera toujours le même phénomène déterministe avec une complexité du même ordre - de plus lors d'une liste « mal organisée », on peut avoir le risque d'avoir des sous-listes totalement disproportionnées (par exemple si la liste est déjà triée et qu'on choisit l'élément de tête pour trier, on perd le bénéfice du « diviser pour régner »). On peut alors « randomiser » l'algorithme en choisissant un pivot aléatoire. De plus sur une même liste la complexité sera différente à chaque appel avec plus de chance d'obtenir la complexité moyenne.
- On voit assez facilement que la complexité dans le pire des cas est en $O(n^2)$. On montre beaucoup plus difficilement que la complexité moyenne est en $O(n \log n)$.

V | Bilan

tri	complex. moyenne	complex. max	en place	stable	remarque
tri par sélection	$O(n^2)$	$O(n^2)$	oui	non	toujours $\frac{n(n-1)}{2}$ comparaisons
tri bulle	$\frac{n(n-1)}{4}$ échanges	$O(n^2)$	oui	oui	toujours $\frac{n(n-1)}{2}$ comparaisons
tri par insertion	$\sim \frac{n^2}{4}$	$O(n^2)$	oui	oui	complex. sur le nombre de comparaisons
tri fusion	$O(n \log n)$	$O(n \log n)$	non		
tri rapide	$O(n \log n)$	$O(n^2)$	faisable	non	

Chapitre 5 | Numpy et Matplotlib


I | Numpy et les tableaux

I.1 | Création de tableaux (ndarray)

On commence par importer le module numpy en mémoire. Afin de ne pas l'importer dans l'espace de noms principal et afin de conserver une syntaxe pas trop lourde, la plupart du temps on l'importe en tant que np

```
>>> import numpy as np
```


On dispose de différents moyens simples de création pour un tableau à plusieurs dimensions (le type est appelé ndarray = N-dimensional array) :

Création d'un tableau NumPy

array(list)	création à partir d'une liste (de listes si besoin) Python (ou d'un itérable)
arange(debut, fin, pas)	comme range Python
linspace(a, b, n)	subdivision de [a, b] en n points, extrémités comprises - on peut ajouter endpoint=False si on ne veut pas du dernier point
zeros(dim)	tableau de 0 avec les dimensions données (dim est un tuple)
ones(dim)	idem avec des 1
eye(n)	matrice carrée identité de taille n
eye(p, q, n)	matrice de taille p x q avec une diagonale de 1 décalée de n crans
diag(list)	matrice carrée de diagonale list
fromfunction(f, dim)	crée le tableau à partir de la fonction f, avec le format dim

I.2 | Caractéristiques d'un tableau


Quelques propriétés pour un ndarray :

Propriétés d'un ndarray

ndim	nombre de dimension du tableau
shape	les dimensions du tableau
size	le nombre total d'éléments (produit des dimensions)
dtype	le type des éléments du tableau

I.3 | Opérations sur les tableaux

Toutes les opérations se font élément par élément. On dispose des fonctions mathématiques usuelles mais qui ont été programmées pour être vectorisées (elles s'appliquent directement à un tableau ndarray)

Quelques fonctions usuelles

square, sqrt	carré et racine
sum, prod	somme et produit
absolute, fabs	valeur absolue (fabs retourne un flottant)
minimum, maximum, fmin, fmax	min et max
sin, cos, tan	fonctions trigonométriques
arcsin, arccos, arctan	et réciproques
exp, exp2	exponentielles (base e et 2)
log, log10, log2	logarithmes (base e, 10 et 2)



I.4 | Sélection, extraction

On dispose d'une syntaxe assez proche de celle de l'extraction de certaines parties d'un tableau

- **pour un élément**

```
>>> A=np.array([[2,4,5],[6,8,1]])
>>> A[1,2]=0
```

- **sur les lignes**

```
>>> A=np.array([[2,4,5],[6,8,1]])
>>> A[1]
array([6,8,1])
```

- **une colonne**

```
>>> A[:,0]
array([2, 6])
>>> A[:,0].shape
(2,)
# cela donne un tableau à une seule dimension
```

- **comme en Python**

```
>>> A=np.arange(12)
>>> A
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> A[1:9:2]
array([1, 3, 5, 7])
```

Comme en Python, on peut utiliser des valeurs négatives pour le début, la fin (et même le pas).

I.5 | Redimensionnement

Les tableaux ne sont pas toujours de la bonne taille... on a parfois envie de les redimensionner. Cela se réalise grâce à la méthode `reshape` :

```
>>> A=np.arange(12)
>>> A
>>> A.reshape((3,4))
>>> A # A n'est pas modifiée
>>> B=A.reshape((4,3))
>>> B
>>> B[2,2]=-1
>>> A
# A est modifiée car B n'est qu'une vue modifiée dans les dimensions de A
```

Si on veut modifier le tableau, on peut utiliser la méthode `resize` :

```
>>> A=np.arange(9)
>>> A.resize(3,3)
>>> A
```



méthodes de redimensionnement

<code>reshape</code>	renvoie une version redimensionnée sans modifier le tableau
<code>resize</code>	redimensionne effectivement le tableau, ne renvoie rien

I.6 | Nombres aléatoires

On peut tirer aléatoirement des nombres

```
# on charge le module numpy correspondant
>>> import numpy.random as random
>>> random.rand(3,2) # distribution uniforme dans [0,1]
>>> random.randn(3,2) # distribution normale (gaussienne)
```

Dans le tableau suivant, `format` est un entier ou un tuple représentant donnant les dimensions du tableau

Tableaux aléatoires (numpy.random)

<code>rand(d1,d2,...)</code>	distribution uniforme dans $[0, 1]$, matrice à plusieurs dimensions
<code>random(format)</code>	distribution uniforme dans $[0, 1]$, dimension donnée par le tuple <code>format</code>
<code>randn(d1,d2,...)</code>	loi normale centrée réduite, matrice à plusieurs dimensions
<code>randint(max)</code>	entier entre 0 et $max - 1$
<code>randint(min,max,format)</code>	tableau d'entiers entre min et max (non compris)

II | Matplotlib

Matplotlib est un module destiné à produire des graphiques de toute sorte. Il peut fonctionner à partir de tableau Python, mais nous l'utiliserons essentiellement à partir de tableaux Numpy.

II.1 | Graphe simple

on importe le module `matplotlib.pyplot` sous le nom `plt` afin d'avoir accès aux fonctions de façon plus simple

```
>>> import matplotlib.pyplot as plt
```

la fonction la plus simple est la fonction `plot` sous la forme `plot(x,y)` où x est un tableau d'abscisses et y le tableau des ordonnées associées (donc de même taille)

```
>>> import matplotlib.pyplot as plt
>>> x=np.linspace(0,1,21)
>>> y=x*x
>>> plt.plot(x,y)
>>> plt.show()
```

La dernière commande ouvre une fenêtre avec le graphe

II.2 | Plusieurs courbes en même temps

Il suffit de les ajouter sur le même graphe avec `plt.plot` :

```
>>> x=np.linspace(0,2*np.pi,101)
>>> plt.plot(x,np.sin(x))
>>> plt.plot(x,np.sin(2*x))
>>> plt.plot(x,np.sin(3*x))
>>> plt.show()
```

De façon plus condensée, on peut le faire en une seule instruction

```
>>> x=np.linspace(0,2*np.pi,101)
>>> plt.plot(x,np.sin(x),x,np.sin(2*x),x,np.sin(3*x))
>>> plt.show()
```

II.3 | Quelques améliorations

- **Propriétés de la courbe** : lorsqu'on crée un plot, on peut préciser différentes options comme le style de ligne, la couleur.

```
>>> x=np.linspace(0,2*np.pi,21)
>>> plt.plot(x,np.sin(x), 'b-^')
```

le troisième argument décrit le style de courbe

Style simplifié d'une courbe

<code>r,g,b,c,m,y,w,k</code>	couleur (k pour black)
<code>- -- . :</code>	style de la ligne
<code>. , ^ o < > + x</code>	style des marqueurs (voir l'aide pour d'autres)



On peut également préciser les options dans une liste d'options

```
>>> x=np.linspace(0,2*np.pi,101)
>>> plt.plot(x,np.sin(x), color='red', linewidth=3, linestyle="-.", marker="o", markersize=10,
            markeredgewidth=3, markeredgewidth=3)
>>> plt.show()
```

- **zone d'affichage** : on peut modifier la zone d'affichage avec les commandes `xlim` et `ylim` :

```
>>> x=np.linspace(0,2*np.pi,41)
>>> plt.plot(x,np.sin(x))
>>> plt.xlim()
>>> plt.xlim(0,np.pi)
>>> plt.ylim(0,1)
>>> plt.show()
```


Chapitre 6 | Résumé des commandes pour l'oral de Centrale

I | Calcul matriciel

```
1 import numpy as np ; import numpy.linalg as alg
```



Création, manipulation

<code>np.array(list)</code>	création à partir d'une liste (de listes si besoin) Python (ou d'un itérable)
<code>np.linspace(a,b,n)</code>	subdivision de $[a,b]$ en n points, extrémités comprises - on peut ajouter <code>endpoint=False</code> si on ne veut pas du dernier point
<code>np.zeros(dim)</code>	tableau de 0 avec les dimensions données (dim est un tuple)
<code>np.ones(dim)</code>	idem avec des 1
<code>np.eye(n)</code>	matrice carrée identité de taille n
<code>np.eye(p,q,n)</code>	matrice de taille $p \times q$ avec une diagonale de 1 décalée de n crans
<code>np.diag(list)</code>	matrice carrée de diagonale <i>list</i>
<code>np.fromfunction(f,dim)</code>	crée le tableau à partir de la fonction <i>f</i> , avec le format <i>dim</i>
<code>A.shape, A.reshape(dim)</code>	format de la matrice, reformatage de la matrice
<code>np.concatenate((A,B),axis=0)</code>	concaténation vecticale (horizontale avec <code>axis=1</code>)
<code>+,*,np.dot(A,B)</code> ou <code>A.dot(B)</code>	somme, produit (scalaire ou case à case), produit matriciel
<code>A.T</code> ou <code>np.transpose(A)</code>	transposition
<code>alg.matrix_power(A,n)</code>	puissances de A
<code>alg.det(A), alg.trace(A)</code>	déterminant, trace de A
<code>alg.matrix_rank(A)</code>	rang de la matrice
<code>alg.inv(A), alg.solve(A,b)</code>	inverse de A, résolution de $Ax=b$
<code>alg.eigvals(A), alg.eig(A)</code>	valeurs propres, éléments propres (val. et vect.)
<code>np.vdot(u,b), np.cross(u,v)</code>	produit scalaire, vectoriel

II | Polynômes

```
1 from numpy.polynomial import Polynomial
```



Création, opérations

<code>p = Polynomial([-3, 2, 0, 1])</code>	création de $-3 + 2X + X^3$
<code>p(val), p.degree(), p.coef</code>	évaluation, degré, tableau des coefficients
<code>p.deriv(n), p.roots()</code>	polynôme dérivé (n nombre de fois), racines du polynôme
<code>p.integ(1,p0)</code>	primitive avec constante
<code>p.integ(n,tab)</code>	primitives successives avec tableau des constantes
<code>+,*,**,/,%</code>	opérations usuelles

III | Graphiques

```
1 import matplotlib.pyplot as plt ; import numpy as np
2 from mpl_toolkits.mplot3d import Axes3D
```

**Tracé 2D**

<code>plt.plot(X,Y,options)</code>	tracé classique
<code>plt.axis('equal')</code>	pour un repère orthonormé
<code>plt.grid()</code>	affichage d'une grille

Pour les tracés en 3D, voir la documentation.

IV | Probabilités

```
1 import numpy.random as rd
```

**Création, manipulation**

<code>rd.randint(a,b,n)</code>	tirage uniforme d'un ou n (facultatif) entiers dans $[a, b[$
<code>rd.randint(a,b,(p,q))</code>	matrice aléatoire d'entiers
<code>rd.random(), rd.random(dim)</code>	loi uniforme sur $[0, 1[$.
<code>rd.binomial(n,p,nombre)</code>	loi binomiale $B(n, p)$
<code>rd.geometric(p,nombre)</code>	loi géométrique de paramètre p
<code>rd.poisson(l,nombre)</code>	loi de Poisson

V | Analyse numérique

```
1 import numpy as np
2 import scipy.optimize as resol
3 import scipy.integrate as integr
4 import matplotlib.pyplot as plt
```

**Résolutions**

<code>resol.fsolve(f,x0)</code>	résolution numérique au voisinage de x_0
<code>resol.root(f,[vals])</code>	idem avec f à plusieurs variables et conditions
<code>integ.quad(f,a,b)</code>	intégration numérique (<code>np.inf</code> pour l'infini)
<code>integ.odeint(f,x0,T)</code>	résolution de $y' = f(y, t)$ avec condition initiale x_0 sur le tableau d'abscisses T

Chapitre 7 | CCP MP

I | CCP MP 2019

Dans cet exercice « Algorithme de décomposition primaire d'un entier » (*Informatique pour tous*), on se propose d'écrire un algorithme pour décomposer un entier en produit de nombres premiers. Les algorithmes demandés doivent être écrits en langage **Python**. On sera très attentif à la rédaction et notamment à l'indentation du code.

On définit la valuation p -adique $[de\ n]$ pour p nombre premier et n entier naturel non nul :

- Si p divise n , on note $v_p(n)$ le plus grand entier k tel que p^k divise n .
- Si p ne divise pas n , on pose $v_p(n) = 0$.

L'entier $v_p(n)$ s'appelle la valuation p -adique de n .

Q.1 Écrire une fonction booléenne `estPremier(n)` qui prend en argument un entier naturel non nul n et qui renvoie le booléen `True` si n est premier et le booléen `False` sinon. On pourra utiliser le critère suivant : un entier $n \geq 2$ qui n'est divisible par aucun entier $d \geq 2$ tel que $d^2 \leq n$, est premier.

Q.2 En déduire une fonction `liste_premiers(n)` qui prend en argument un entier naturel non nul n et renvoie la liste des nombres premiers inférieurs ou égaux à n .

Q.3 Pour calculer la valuation 2-adique de 40, on peut utiliser la méthode suivante :

- 40 est divisible par 2 et le quotient vaut 20.
- 20 est divisible par 2 et le quotient vaut 10.
- 10 est divisible par 2 et le quotient vaut 5.
- 5 n'est pas divisible par 2.

La valuation 2-adique de 40 vaut donc 3.

Écrire une fonction `valuation_p_adique(n,p)` **non récursive** qui implémente cet algorithme. Elle prend en arguments un entier naturel n non nul et un nombre premier p et renvoie la valuation p -adique de n . Par exemple, puisque $40 = 2^3 \times 5$, `valuation_p_adique(40,2)` renvoie 3, `valuation_p_adique(40,5)` renvoie 1 et `valuation_p_adique(40, 7)` renvoie 0.

Q.4 Écrire une deuxième fonction cette fois-ci **récursive** `val_p_adique(n,p)` qui renvoie la valuation p -adique de n .

Q.5 En déduire une fonction `decomposition_facteurs_premiers(n)` qui calcule la décomposition en facteurs premiers d'un entier $n \geq 2$. Cette fonction doit renvoyer la liste des couples $(p, v_p(n))$ pour tous les nombres premiers p qui divisent n .

Par exemple, `decomposition_facteurs_premiers(40)` renvoie la liste `[[2, 3], [5, 1]]`.

II | CCP MP 2018

Les questions sont au milieu du problème, je ne donne que les questions qui servent

On considère l'intégrale de Gauss $I = \int_0^1 e^{-x^2} dx$.

Q.1. Démontrer à l'aide d'une série entière que $I = \sum_{n=0}^{+\infty} \frac{(-1)^n}{(2n+1)n!}$. On pose, pour $n \in \mathbb{N}$, $s_n = \sum_{k=0}^n \frac{(-1)^k}{(2k+1)k!}$.

Q.2. Justifier que pour tout $n \in \mathbb{N}$, on

$$|I - s_n| \leq \frac{1}{(2n+3)(n+1)!}$$

Q.3. Informatique : écrire une fonction récursive `factorielle` qui prend en argument un entier n et renvoie l'entier $n!$.

Q.4. Informatique : en déduire un script qui détermine un entier N tel que $|I - s_N| \leq 10^{-6}$.

Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue. On se donne $n+1$ points x_0, x_1, \dots, x_n dans $[a, b]$, deux à deux distincts. On appelle polynôme interpolateur de f aux points x_i , un polynôme $P \in \mathbb{R}_n[X]$ qui coïncide avec f aux points x_i , c'est-à-dire tel que pour tout $i \in \llbracket 0; n \rrbracket$, $P(x_i) = f(x_i)$.

Existence du polynôme interpolateur

Pour tout entier i de $\llbracket 0; n \rrbracket$, on définit le polynôme l_i de $\mathbb{R}_n[X]$ par :

$$l_i(X) = \prod_{\substack{k=0 \\ k \neq i}}^n \frac{X - x_k}{x_i - x_k}$$

On pose :

$$L_n(f) = \sum_{i=0}^n f(x_i) l_i(X)$$

Q.7. Démontrer que $L_n(f)$ est un polynôme interpolateur de f aux points x_i , puis démontrer l'unicité d'un tel polynôme.

Un tel polynôme est appelé polynôme interpolateur de Lagrange.

**Calcul effectif du polynôme interpolateur de Lagrange**

Q.8. Informatique : si y_0, \dots, y_n sont des réels, le polynôme $P = \sum_{i=0}^n y_i l_i(X)$ est l'unique polynôme de $\mathbb{R}_n[X]$ vérifiant $P(x_i) = y_i$ pour tout

i . Écrire en langage Python une fonction `lagrange` qui prend en arguments x une liste de points d'interpolations x_i , y une liste d'ordonnées y_i de même longueur que x , a un réel, et qui renvoie la valeur de P en a .

Par exemple, si $x = [-1, 0, 1]$ et $y = [4, 0, 4]$, on montre que $P = 4X^2$ et donc $P(3) = 36$. Ainsi `lagrange(x, y, 3)` renverra 36.

Q.9. Informatique : chercher le polynôme interpolateur $P = a_0 + a_1 X + \dots + a_n X^n$ de f aux points x_i revient aussi à résoudre le système linéaire suivant d'inconnues a_0, \dots, a_n :

$$\begin{cases} P(x_0) = f(x_0) \\ \vdots \\ P(x_n) = f(x_n) \end{cases} \iff V \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ \vdots \\ f(x_n) \end{pmatrix}$$

où V est une matrice carrée de taille $n+1$.

Déterminer la matrice V et indiquer la complexité du calcul en fonction de n , lorsque l'on résout ce système linéaire par la méthode du pivot de Gauss.

III | CPP MP 2017**Partie V - Informatique : calcul effectif de la probabilité invariante d'une matrice stochastique strictement positive**

Si A est une matrice stochastique strictement positive, on a établi dans la partie précédente la convergence de la suite $(\mu_n)_{n \in \mathbb{N}}$ associée à la matrice A . Ceci fournit un algorithme de calcul de la probabilité invariante par A . On en propose une implémentation en langage Python. On sera très attentif à la rédaction et notamment à l'indentation du code.

Un vecteur x de \mathbb{R}^p sera représenté en Python par une liste de flottants. Par exemple, le vecteur $x = (1, 2, 3)$ de \mathbb{R}^3 sera représenté par la liste `[1, 2, 3]`. De même, une matrice A sera représentée par une liste dont les éléments sont les lignes de la matrice. Par exemple, la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ sera représentée par la liste `[[1, 2, 3], [4, 5, 6]]`.

Q.29. On exécute le script suivant `A = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]` qui représente la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}.$$

Donner les valeurs renvoyées lorsque l'on exécute `len(A)`, `A[1]` et `A[2][1]`.

Q.30. Écrire une fonction `difference` qui prend en arguments deux vecteurs x et y de même taille et renvoie le vecteur $x - y$. Par exemple si $x = (5, 2)$ et $y = (3, 7)$, `difference(x, y)` renverra `[2, -5]`.

Q.31. Écrire une fonction `norme` qui prend en argument un vecteur $x = (x_1, \dots, x_p)$ et renvoie sa norme infinie $\|x\|_\infty = \max\{|x_i| \mid i \in [1; p]\}$ (on pourra utiliser librement la fonction `abs` qui renvoie la valeur absolue d'un nombre, mais on s'interdit l'utilisation de la fonction `max` déjà implémentée dans Python).

Q.32. Écrire une fonction `itere` qui prend en arguments un vecteur ligne x et une matrice carrée de même taille que x et qui renvoie le vecteur xA . Par exemple si $x = (1, 1)$ et $A = \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$, on a $xA = (5, 7)$ et donc `itere(x, A)` renverra `[5, 7]`.

Q.33. On a vu, dans la **Partie IV**, que si A est une matrice strictement positive, la suite de vecteurs lignes de \mathbb{R}^p associée $(\mu_n)_{n \in \mathbb{N}}$ définie par la relation : $\forall n \in \mathbb{N}, \mu_{n+1} = \mu_n A$ convergeait vers un vecteur μ_∞ indépendant du choix de μ_0 vecteur stochastique.

Écrire une fonction `probaInvariante` qui prend en arguments une matrice stochastique strictement positive A de $M_p(\mathbb{R})$ et un réel $\varepsilon > 0$ et qui renvoie le premier terme μ_k de la suite $(\mu_n)_{n \in \mathbb{N}}$ avec $\mu_0 = \left(\frac{1}{p}, \frac{1}{p}, \dots, \frac{1}{p}\right)$ tel que $\|\mu_k - \mu_{k-1}\|_\infty \leq \varepsilon$. On ne demandera pas à l'algorithme de vérifier que la matrice passée en argument est bien stochastique et strictement positive.

Par exemple, si $A = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} \end{pmatrix}$ et $\varepsilon = 10^{-6}$,

`probaInvariante(A, eps)` renverra `[0.33333396911621094, 0.6666660308837891]`.

IV | CCP MP 2016**EXERCICE I : INFORMATIQUE**

Les algorithmes demandés doivent être écrits en Python. On sera très attentif à la rédaction et notamment à l'indentation du code. Cet exercice étudie deux algorithmes permettant le calcul du pgcd (plus grand diviseurs communs) de deux entiers naturels.

- I.1.** Pour calculer le pgcd de 3705 et 513, on peut passer en revue tous les entiers $1, 2, 3, \dots, 513$ puis renvoyer parmi ces entiers le dernier qui divise à la fois 3705 et 513. Il sera alors bien le plus grand des diviseurs commun à 3705 et 513. Écrire une fonction `gcd` qui renvoie le pgcd de deux entiers naturels non nuls, selon la méthode décrite ci-dessus. On pourra éventuellement utiliser librement l'instruction `min(a, b)` qui calcule le minimum de a et b . Par exemple `gcd(3705, 513)` renverra 57.
- I.2.** L'algorithme d'Euclide permet aussi de calculer le pgcd. Voici une fonction Python nommée `euclide` qui implémente l'algorithme d'Euclide.

```
def euclide(a,b):
    """Données: a et b deux entiers naturels
    Résultat: le pgcd de a et b, calculé par l'algorithme d'Euclide"""
    u=a
    v=b
    while v !=0:
        r=u%v
        u=v
        v=r
    return u
```

Écrire une fonction « récursive » `euclide_rec` qui calcule le pgcd de deux entiers naturels selon l'algorithme d'Euclide.

- I.3.** On note $(F_n)_{n \in \mathbb{N}}$ la suite des nombres de Fibonacci définie par :

$$F_0 = 0, F_1 = 1, \quad \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n.$$

- I.3.a.** Écrire les divisions euclidiennes successivement effectuées lorsque l'on calcule le pgcd de $F_6 = 8$ et $F_5 = 5$ avec la fonction `euclide`.
- I.3.b.** Soit $n \geq 2$ un entier. Quel est le reste de la division euclidienne de F_{n+2} par F_{n+1} ? On pourra utiliser librement que la suite $(F_n)_{n \in \mathbb{N}}$ est strictement croissante à partir de $n = 2$. En déduire, sans démonstration, le nombre u_n de divisions euclidiennes effectuées lorsque l'on calcule le pgcd de F_{n+2} et F_{n+1} avec la fonction `euclide`.
- I.3.c.** Comparer pour n au voisinage de $+\infty$, ce nombre u_n , avec le nombre v_n de divisions euclidiennes effectuées pour le calcul du pgcd de F_{n+2} et F_{n+1} par la fonction `gcd`. On pourra utiliser librement que F_n est équivalent au voisinage de $+\infty$, à $\phi^n / \sqrt{5}$ où $\phi = (1 + \sqrt{5})/2$ est le nombre d'or.
- I.4.** Écrire une fonction `fibo` qui prend en argument un entier naturel n et renvoie le nombre de Fibonacci F_n . Par exemple, `fibo(6)` renverra 8.
- I.5.** En utilisant la fonction `euclide`, écrire une fonction `gcd_trois` qui renvoie le pgcd de trois entiers naturels. Par exemple, `gcd_trois(18, 30, 12)` renverra 6.

V | CCP MP 2015

EXERCICE I : INFORMATIQUE

Les algorithmes demandés doivent être écrits en Python. On sera très attentif à la rédaction et notamment à l'indentation du code.

Voici, par exemple, un code Python attendu si l'on demande d'écrire une fonction nommée `maxi` qui calcule le plus grand élément d'un tableau d'entiers :

```
1 def maxi(t):
2     """Données: t un tableau d'entiers non vide
3     Résultat: le maximum des éléments de t"""
4     n = len(t) # la longueur du tableau t
5     maximum = t[0]
6     for k in range(1,n):
7         if t[k] > maximum:
8             maximum = t[k]
9     return maximum
```

L'instruction `maxi([4,5,6,2])` renverra alors 6.

- I.1.** Donner la décomposition binaire (en base 2) de l'entier 21.
On considère la fonction `mystere` suivante :

```
1 def mystere(n, b):
2     """Données: n > 0 un entier et b > 0 un entier
3     Résultat: ....."""
4     t = [] # tableau vide
5     while n > 0:
6         c = n % b
7         t.append(c)
8         n = n // b
9     return t
```

On rappelle que la méthode `append` rajoute un élément en fin de liste. Si l'on choisit par exemple `t = [4,5,6]`, alors, après avoir exécuté `t.append(12)`, la liste `t` a pour valeur `[4,5,6,12]`.

Pour $k \in \mathbb{N}^*$, on note c_k , t_k et n_k les valeurs prises par les variables `c`, `t` et `n` à la sortie de la k -ème itération de la boucle `while`.



I.2. Quelle valeur est renvoyée lorsque l'on exécute `mystere(256, 10)` ?

On recopiera et complétera le tableau suivant, en ajoutant les éventuelles colonnes nécessaires pour tracer entièrement l'exécution.

k	1	2	...
c_k			...
t_k			...
n_k			...

I.3. Soit $n > 0$ un entier. On exécute `mystere(n, 10)`. On pose $n_0 = n$.

I.3.a Justifier la terminaison de la boucle `while`.

I.3.b On note p le nombre d'itérations lors de l'exécution de `mystere(n, 10)`. Justifier que pour tout $k \in \llbracket 0; p \rrbracket$, on a $n_k \leq \frac{n}{10^k}$. En déduire, une majoration de p en fonction de n .

I.4. En s'aidant du script de la fonction `mystere`, écrire une fonction `somme_chiffres` qui prend en argument un entier naturel et renvoie la somme de ses chiffres. Par exemple, `somme_chiffres(256)` devra renvoyer 13.

I.5. Écrire une version récursive de la fonction `somme_chiffres`, on la nommera `somme_rec`.



Chapitre 9 | Solutions

► Exercice II.1 :

On commence par trouver l'invariant à prouver :

- au passage 0 : $a_0 = 1 = u_0$ et $b_0 = 1 = u_1$,
- à la fin du passage 1 : $a_1 = b_0 = u_1$ et $b_1 = a_0 + b_0 = u_0 + u_1 = u_2$.

On conjecture que l'invariant de boucle est : « à la fin du passage k , on a $a_k = u_k$ et $b_k = u_{k+1}$ ».

- l'invariant est vérifié aux rangs 0 et 1.
- si l'invariant est correct à la fin du passage k , alors $a_{k+1} = b_k = u_{k+1}$ et $b_{k+1} = a_k + b_k = u_k + u_{k+1} = u_{k+2}$. L'invariant est vrai au rang $k+1$.

Pour la terminaison : on a une boucle `for` avec exactement n passages. En sortie, on a $a = a_n = u_n$ et $b = b_n = u_{n+1}$. Le retour de b est donc mauvais. Il vaut mieux renvoyer a (on renvoie b avec seulement $n-1$ passages de boucle... attention dans ce cas aux premières valeurs de n . Si $n=0$ ou $n=1$ alors on renvoie 1 ce qui est, par chance, la bonne valeur dans les deux situations).

► Exercice II.2 :

Version 1

Le premier algorithme est très proche de la preuve de `factoriel`. On montre l'invariant $\mathcal{P}(i)$: « $p_i = x^i$ ».

Pour le second algorithme, on commence par examiner le contenu de s .

- avant de rentrer dans la boucle : $s_0 = 0$,
- à la fin du passage 1 : on a $k=0$ et $s_1 = s_0 + a[0]x^0 = a_0$,
- à la fin du passage 2 : on a $k=1$ et $s_2 = s_1 + a[1]x^1 = a_0 + a_1x$,

On utilise alors l'invariant suivant : à la fin du passage k , on a $s_k = \sum_{j=0}^{k-1} a_j x^j$. On fera attention de bien dissocier le numéro du passage et la valeur du compteur i . En effet au passage k , on a $i = k-1$.

- initialisation : au passage 0 ou 1, l'invariant est correct.
- on suppose que l'invariant est correct à la fin du passage k . On passe au passage $k+1$. On a alors $i = k$ et

$$s_{k+1} = s_k + a[i]x^i = s_k + a[k]x^k = \sum_{j=0}^k a_j x^j.$$

L'invariant se propage.

On sort à la fin du passage $n+1$ et on obtient en sortie $s = s_{n+1} = \sum_{j=0}^n a_j x^j$.

Algorithme de Hörner

On propose cet algorithme :

Algorithme 8 : Évaluation d'un polynôme en x

Données : un tableau de coefficient $a = [a_0, a_1, \dots, a_n]$ et un réel x

Sorties : la valeur de $\sum_{k=0}^n a_k x^k$

début

```
s ← 0
n ← len(a) - 1
pour i de 0 à n faire
  s ← s * x + a[n - i]
retourner s
```

- on a $s_0 = 0$, $s_1 = a_n$, $s_2 = a_n x + a_{n-1}$.
- on prouve l'invariant $s_k = \sum_{j=n-k+1}^n a_j x^{j-n+k-1}$ où (dans l'autre sens de sommation), $s_k = \sum_{j=0}^{k-1} a_{n+j-k+1} x^j$. Le second est peut-être plus simple à écrire mais le premier plus facile à prouver.
- pour $k=1$, la somme donne $\sum_{j=n}^n a_j x^{j-n} = a_n$.
- pour $k=2$, la somme donne $\sum_{j=n-1}^n a_j x^{j-n+1} = a_{n-1} + a_n x$.
- si l'invariant est vrai à la fin du passage k . Au passage $k+1$, on a $i = k$ et

$$s_{k+1} = x s_k + a[n-k] = x \left(\sum_{j=n-k+1}^n a_j x^{j-n+k-1} \right) + a_{n-k} = \sum_{j=n-k+1}^n a_j x^{j-n+(k+1)-1} + a_{n-k} = \sum_{j=n-k}^n a_j x^{j-n+k}$$

car le terme ajouté est bien $a_j x^{j-n+k}$ pour $j = n-k$ (c'est-à-dire a_{n-k}).

- à la fin du passage $n+1$, on a $s_{n+1} = \sum_{j=0}^n a_j x^j$.

**Complexité**

On compte le nombre de multiplications et d'additions (sans compter les initialisations) :

- Pour la fonction `expo(x, k)` : exactement k multiplications.
- Pour l'évaluation naïve : $(1+0) + (1+1) + (1+2) + \dots + (1+n) = 1+2+\dots+(n+1) = \frac{(n+1)(n+2)}{2}$ opérations (pour ajouter $a_k x^k$, on a $(1+k)$ opérations). Cela donne une complexité en $O(n^2)$.
- Pour l'algorithme de Hôrner : $2(n+1)$ opérations et ainsi une complexité en $O(n)$.

► Exercice II.3 :**PGCD**

Assez simple, on sait que $a \wedge b = b \wedge r$ où r est le reste de la division euclidienne de a par b . Cela donne l'algorithme et la preuve :

```

1 def pgcd(a,b):
2     """ pgcd de a,b avec a,b entiers naturels """
3     while b>0:
4         a,b = b, a%b
5     return a

```

Pour la preuve, on note de façon usuelle a_k et b_k les contenus de a et b à la fin du passage k . On a $a_0 = a$ et $b_0 = b$. On prouve l'invariant $a_k \wedge b_k = a \wedge b$. Avec les relations $a_{k+1} = b_k$ et $b_{k+1} = r_k$ reste de la division de a_k par b_k . Alors $a_{k+1} \wedge b_{k+1} = b_k \wedge r_k = a_k \wedge b_k = a \wedge b$. Pour la terminaison : la suite b_k est une suite d'entiers positifs, strictement décroissante puisque tant qu'on est dans la boucle b_k est strictement positif donc $b_{k+1} < b_k$. En sortie, on a $b_n = 0$ (on pourrait ajouter l'invariant $b_k \geq 0$ mais c'est évident... du moment que $b_0 \geq 0$ quand même) et $a_n \wedge b_n = a_n = a \wedge b$.

Relation de Bezout

On note r_k le reste k dans l'algorithme avec $r_0 = a$ et $r_1 = b$. On a une relation $r_{k-1} = q_k r_k + r_{k+1}$ ou encore $r_{k+1} = r_{k-1} - q_k r_k$ (q_k est le quotient de la division euclidienne de r_{k-1} par r_k . Si on écrit $r_k = u_k a + v_k b$ (ils sont tous dans l'idéal engendré par a et b ... ou bien on le montre simplement par récurrence), on obtient les relations

$$u_{k+1} = u_{k-1} - q_k u_k \text{ et } v_{k+1} = v_{k-1} - q_k v_k.$$

On calcule de proche en proche les 6 valeurs $r_{k-1}, r_k, u_{k-1}, u_k, v_{k-1}, v_k$ dans les variables R, RR, U, UU, V, VV (en général on utilise des majuscules pour des constantes, pas pour des variables... mais ici pour des raisons de clarté, c'est-à-dire pour dissocier la suite des restes (r_k) et le contenu de R aux différentes étapes, on utilise cette distinction) :

```

1 def bezout(a,b):
2     U,V = 1,0 # r0 = a = 1.a+0.b
3     UU,VV = 0,1 # r1 = b = 0.a+1.b
4     R,RR = a,b
5     while RR>0:
6         q = R // RR
7         reste = R % RR
8         R,RR = RR, reste
9         U,UU = UU, U-q*UU
10        V,VV = VV, V-q*VV
11    return R,U,V

```

```

>>> bezout(14070,55965)
(105, 179, -45)
>>> 179*14070-45*55965
105

```

On prouve les invariants, à la fin du passage k :

$$R_k = r_k, RR_k = r_{k+1}, U_k = u_k, UU_k = u_{k+1}, V_k = v_k \text{ et } VV_k = v_{k+1}.$$

On pourrait simplement remplacer les quatre derniers par $R_k = aU_k + bV_k$ et $RR_k = aUU_k + bVV_k$.

- Au passage 0, on a $R_0 = a = r_0$ et $RR_0 = b = r_1$ et les valeurs les autres termes correspondent
- Si l'invariant est vrai au passage k , alors au passage $k+1$ on a
 - $R_{k+1} = RR_k = r_{k+1}$ et $RR_{k+1} = R_k \% RR_k = r_k \% r_{k+1} = r_{k+2}$,
 - $U_{k+1} = UU_k = u_{k+1}$ et $UU_{k+1} = U_k - q_k UU_k = u_k - q_k u_{k+1} = u_{k+2}$,
 - idem avec V et VV .
- La suite (RR) est strictement décroissante, positive donc la boucle termine.
- En sortie R et RR sont les deux derniers restes avec $RR = 0$ - tout est fait pour qu'on ait $R_k = aU_k + bV_k$, notamment en sortie.