Introduction à Python (pour les maths)

Mickaël Péchaud

2017

Table des matières

Préambule

Vous pourrez retrouver ce poly, son corrigé, ainsi que les fichiers *Python* à l'adresse suivante :
http://mpechaud.fr/formationpython

Pour toutes remarques, questions, merci de vous adresser par mail à l'auteur de ce poly (prenomnom@netc.fr) !

Complexité

Je vais être dans ce poly amené à parler de *complexité* de telle ou telle fonction, ou de tel ou tel algorithme.
Sans rentrer dans les détails (passionnants, mais... difficiles), la complexité d'un algorithme est une mesure de l'ordre de grandeur du nombre d'«opérations élémentaires» (selon les cas : affectation, addition, lecture d'un élément...) nécessaires à son exécution, en fonction de la «taille des données» en entrée.
Dans ce poly, l'entrée sera un tableau (ou liste), et la «taille des données» sera simplement la longueur n du tableau.
La complexité est généralement donnée sous forme d'un $O()$ [1].
Voici les principales *classes de complexité* que nous aurons à manipuler.

complexité	**notation**	**remarque**
Constante	$O(1)$	nombre d'opérations indépendant de la taille de la liste
Logarithmique	$O(\log(n))$ [2]	
Linéaire	$O(n)$	nombre d'opérations au plus proportionnel à la taille de la liste
Quasi-linéaire	$O(n\log(n))$	log est «quasiment» constante en pratique !
Quadratique	$O(n^2)$	
Cubique	$O(n^3)$	on commence souvent à avoir des ennuis en pratique ici...
Exponentielle	$O(a^n)$, où $a > 1$	et ici de gros ennuis...

Quelques ordres de grandeur
Les ordres de grandeur ci-dessous sont donnés en supposant que la constante devant le nombre d'opérations est 1, et que l'on travaille sur un ordinateur capable d'effectuer 10^9 opérations par s.

Complexité	$n = 10^4$	$n = 10^6$	$n = 10^8$
$\mathrm{O}(1)$	1 ns	1 ns	1 ns
$\mathrm{O}(\log n)$	9 ns	14 ns	18 ns
$\mathrm{O}(n)$	10 μs	1 ms	100 ms
$\mathrm{O}(n^2)$	100 ms	15 min	100 jours
$\mathrm{O}(n^3)$	15 min	30 ans	31 millions d'années

Complexité	$n = 10$	$n = 50$	$n = 100$
$\mathrm{O}(2^n)$	1 μs	13 jours	40 000 milliards d'années

1. $f(n) = O(g(n))$ si et seulement si il existe une constante K telle que pour n assez grand on ait $f(n) \leqslant Kg(n)$.
2. Peu importe la base du logarithme utilisée ici, mais en informatique, log désigne généralement $\log_2$

1 Tableaux : algorithmes en taille fixée

1.1 Généralités

Un *tableau* est d'un point de vue abstrait un ensemble ordonné d'éléments, accessibles (en lecture et en écriture) par leur indice.
Les tableaux sont implémentés en *Python* par la structure appelée *liste* (*List*). Dans la suite, j'emploierai donc le terme *Python* de *liste*, même si d'un point de vue algorithmique, il s'agit dans un premier temps d'apprendre à manipuler la structure abstraite de tableau [3].
Nous représenterons une liste `t = [1, 3, 2, -1, 2, 4]` de la façon suivante :

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$
1	3	2	-1	2	4

Le premier indice est 0, et le dernier `len(t)-1`.

Code Python :

```
>>> t=[1, 3, 2, -1, 2, 4]
>>> type(t)
<class 'list'>
>>> t[0]
1
```

Il existe d'autres façons de définir des listes :

Code Python :

```
>>> 10*[0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> [i for i in range(1,10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Une liste se comporte quasiment comme un tuple, à ceci près qu'on peut modifier ses éléments :

Code Python :

```
>>> t=[i**2 for i in range(1,11)]
>>> t
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> len(t)
10
>>> t[4]
25
>>> t[4 :10]
[25, 36, 49, 64, 81, 100]
>>> t[-1] # dernier élément
100
>>> t[0]=5
>>> t
[5, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

3. Le terme *liste* est utilisé en *Python*, car les listes *Python* implémentent également la structure «abstraite» de «liste» : un ensemble ordonné d'éléments qu'il est possible de parcourrir et où il est possible d'ajouter et supprimer des éléments au début ou à la fin

Rem La création d'une liste est de complexité *linéaire* en la taille de la liste.
Obtenir la taille de la liste est une opération en temps *constant* en la taille de la liste.
Accès et écriture dans une liste sont effectuées en temps *constant* – ce qui fait la force de cette structure.

1.2 Premiers algorithmes

1.2.1 Parcours d'une liste

Rappelons les deux principales syntaxes de parcours d'une liste.

Par parcours des indices

for *variable* **in** range(len(l)) :
␣ *bloc d'instructions*

Code Python :

```
def contient_0(l) :
    """teste si la liste passée en argument contient 0"""
    for k in range(len(l)) :
        if l[k]==0 :
            return True
    return False
```

Par parcours direct des éléments de la liste [4]

for *variable* **in** l :
␣ *bloc d'instructions*

Code Python :

```
def contient_0(l) :
    """teste si la liste passé en argument contient 0"""
    for e in l :
        if e==0 :
            return True
    return False
```

En utilisant cette syntaxe, on n'a plus accès aux indices des éléments dans la liste. Si l'on a besoin de cet indice, il faut revenir à la syntaxe précédente [5] :

4. On parle d'«itération» sur la liste.
5. il existe également une syntaxe spécifique à *Python* : `for i,v in enumerate(l) :` qui itère à la fois sur les indices (i) et sur les valeurs (v). Il est théoriquement conseillé de l'utiliser lorsqu'on a besoin des indices pour traiter les éléments de la liste. Pédagogiquement, je préfère utiliser `range`, en remplaçant les v par $l[i]$.

Code Python :

```
def indice_0(l) :
    """renvoie le premier indice où 0 apparait dans l,
       None si la liste ne contient pas 0"""
    for k in range(len(l)) :
        if l[k]==0 :
            return k
    return None
```

Exercice 1 Modifier cette fonction pour qu'elle renvoie le *dernier* indice où 0 apparait dans `l`.

Somme des éléments

Code Python :

```
def somme(l) :
    """renvoie la somme des éléments d'une liste d'entiers ou de flottants"""
```

La complexité de cette fonction est

Code Python :

```
def somme(l) :
    """renvoie la somme des éléments d'une liste d'entiers ou de flottants"""
    s = 0
    for e in l :
        s = s + e
    return s
```

Cette fonction est de complexité linéaire [a].

a. Elle est déjà implémentée en *Python*, avec la commande *sum*.

Calcul du maximum

Code Python :

```
def maximum(l) :
    """renvoie le maximum d'une liste d'entiers ou de flottants"""
```

La complexité de cette fonction est

Code Python :

```
def maximum(l) :
    """renvoie le maximum d'une liste d'entiers ou de flottants"""
    m = l[0]
    for e in l :
        if e > m :
            m = e
    return m
```

Cette fonction est de complexité linéaire [a].

a. Elle est déjà implémentée en *Python*, avec la commande *max*.

Exercice 2 Modifier la fonction `maximum` pour qu'elle renvoie un couple constitué de la valeur maximale et de sa position dans la liste.

Code Python :

```
def maximum(l) :
    """renvoie un couple constitué du maximum d'une liste d'entiers
       ou de flottants, et d'un indice où ce maximum est atteint"""
    (m, i) = (l[0], 0)
    for k in range(len(l)) :
        if l[k] > m :
            (m, i) = (l[k], k)
    return (m, i)
```

Exercice 3 Écrire une fonction `contient_doublons(l)` qui renvoie *True* si la liste d'entiers t contient 2 fois la même valeur, *False* sinon. Quelle est sa complexité ?

Code Python :

```
def contient_doublons(l) :
    """renvoie True si et seulement si la liste passée en
       argument contient au moins 2 fois le même élément"""
    n = len(l)
    for i in range(len(l)) :
        for j in range(i+1, len(l)) :
            if l[i] == l[j] :
                return True
    return False
```

1.2.2 Recherche dans une liste

Recherche d'un élément et de son emplacement

Code Python :

```
def recherche(e, l)
    """renvoie la position d'une occurrence de e dans l si e apparait dans l,
       None sinon"""
```

La complexité dans le pire de cas de cette fonction est

Code Python :

```
def recherche(e, l)
    """renvoie la position d'une occurrence de e dans l si e
       apparait dans l, None sinon"""
    for k in range(len(l)) :
        if l[k] == e :
            return k
    return None
```

La complexité au pire de cette fonction est linéaire.

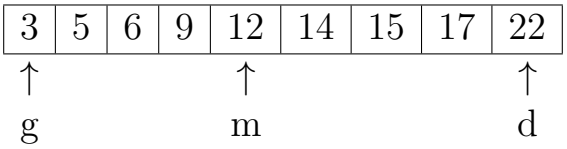
Exercice 4 Modifier cet algorithme pour qu'il renvoie le nombre d'occurrences de `e` dans `l`.

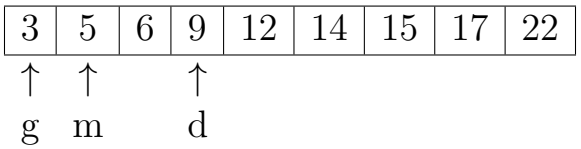
Code Python :

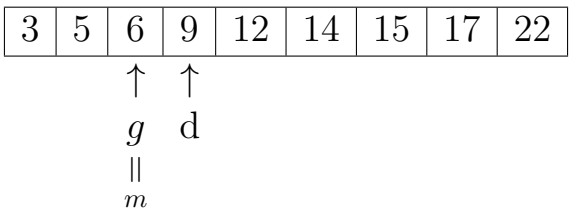
```
def compte(e, l)
    """renvoie le nombre d'occurrences de e dans l"""
    r = 0
    for f in l :
        if f == e :
            r=r+1
    return r
```

Recherche dichotomique dans une liste triée

Exemple : recherche dichotomique de 6 dans la liste `t = [3, 5, 6, 9, 12 14, 15, 17, 22]`.







Code Python :

```
def dichotomie(e,l)
    """renvoie la position d'une occurrence de e dans l
       si e apparait dans l, None sinon"""

    (g,d) =

    while g<=d :

        m =

        if l[m] == e :

            return m

        elif l[m] < e :



        else : # l[m] > e



    return None
```

Code Python :

```
def dichotomie(e,l)
    """renvoie la position d'une occurrence de e dans l
       si e apparait dans l, None sinon"""

    (g,d) = (0, len(l) - 1)

    while g<=d :

        m = (g+d)//2

        if l[m] == e :

            return m

        elif l[m] < e :

            g = m + 1

        else : # l[m] > e

            d = m-1

    return None
```

Notons u_n le nombre de passages dans la boucle nécessaires pour que l'algorithme termine, où $n = d - g + 1$ est la taille de la sous-liste dans lequel on recherche l'élément.
On a
$u_0 = 0,\ u_1 = 1$, et pour tout $n \geqslant 2$, $u_n \leqslant 1 + u_{\lfloor \frac{n-1}{2} \rfloor}$.
On montre alors que $u_n \leqslant \lfloor \log_2(n+1) \rfloor$ [6], donc que la complexité au pire de cet algorithme est logarithmique.

2 Les mains dans le cambouis : l'implémentation des tableaux/listes en machine

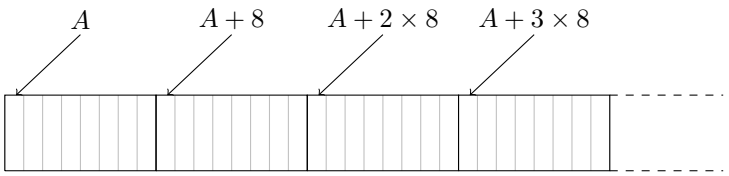
Pour l'instant, nous avons utilisé les listes Python comme des *tableaux*, i.e. une structure indexée (où l'on dispose d'un accès direct aux éléments par le biais d'indices entiers).
On a aussi fréquemment besoin d'opérations modifiant la taille d'une liste (insertion, suppression d'éléments).
Pour comprendre les tenants et aboutissants des ces opérations, il est nécessaire de comprendre comment les listes sont implémentés [7].

2.1 Tableau «naïf»

Dans un certain nombre de langages [8], un tableau est implémenté par un simple bloc mémoire contigü. Dans un tel langage, lorsqu'on déclare un tableau d'entiers (codés sur 8 bits [9]) de taille N, un emplacement mémoire de $N \times 8$ bits est réservé (ou *alloué*).
La variable correspondant au tableau est alors une sorte de pointeur vers le premier élément de ce tableau – c'est-à-dire l'adresse mémoire de celui-ci.
Notons A cette adresse. Le $i^{\text{ème}}$ élément du tableau est alors à l'adresse mémoire $A + 8i$ (c'est pour cette raison que les tableaux sont indicés à partir de 0).

Avantage : on a un accès en lecture et écriture en temps constant, à n'importe quel élément du tableau (on parle d'*accès direct*, ou *random access*).

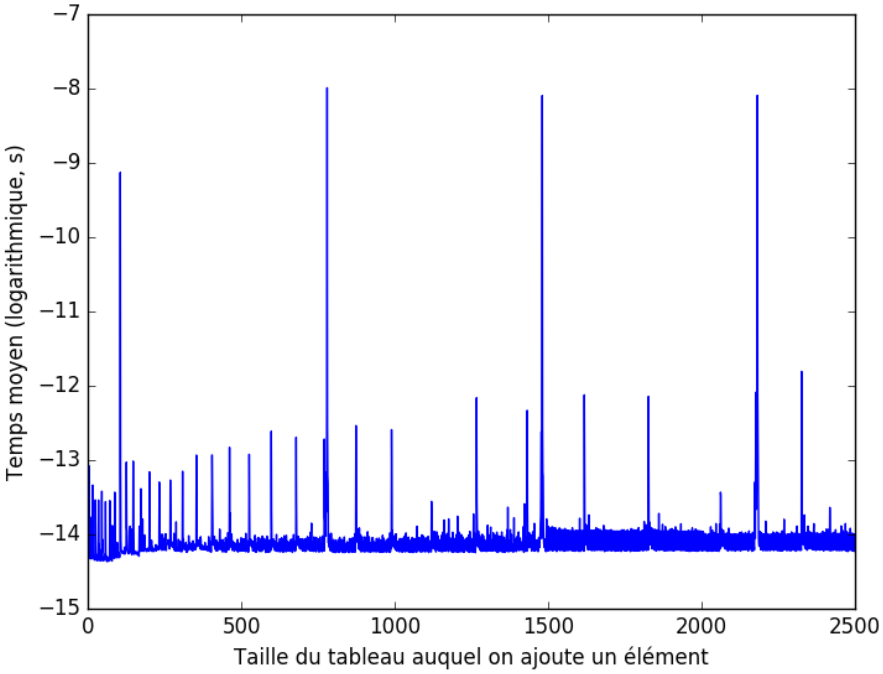
6. Le calcul dans le cas général est un peu technique – le cas où la longueur n de la liste est de la forme $n = 2^p - 1$ est facile à étudier.
7. Anglicisme très utilisé en informatique. Traduit par «mise en œuvre» (inusité).
8. C par exemple, qui n'est pas l'un des moindres...
9. pour l'exemple. En pratique, les entiers sont plutôt codés sur 32 ou 64 bits



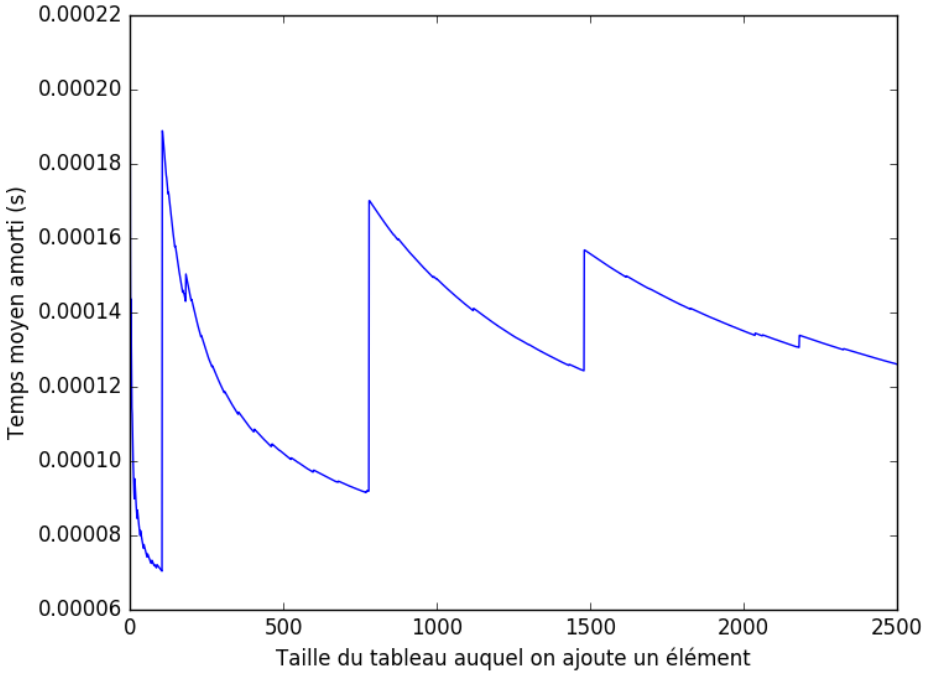
Inconvénient : il s'agit d'une structure *statique* : la taille est fixée. Si on veut agrandir ou raccourcir le tableau, il faut réallouer un bloc mémoire de la bonne taille ailleurs, et recopier tout le tableau dedans (complexité linéaire).

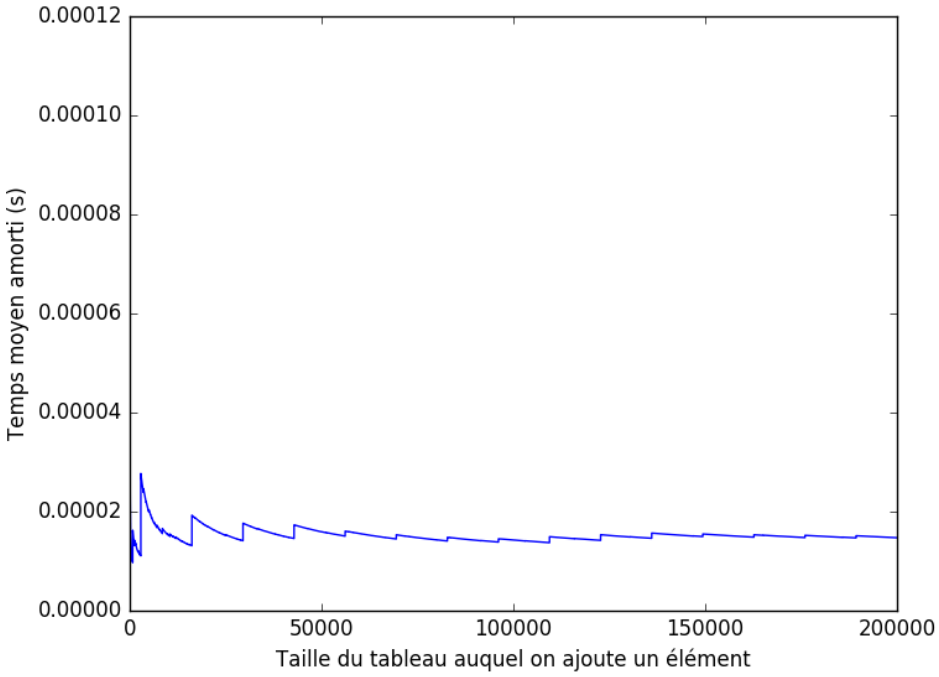
2.2 Tableaux dynamiques

Les tableaux dynamiques permettent de s'affranchir du côté statique des tableaux, tout en conservant la vitesse de lecture et d'écriture.
L'idée est, à la création du tableau, d'allouer un bloc mémoire trop grand. Ainsi, on a la possibilité d'ajouter des éléments à la fin du tableau sans réallocation. On parle de *taille physique* pour la capacité maximale (i.e. la mémoire allouée), et de *taille logique* pour la mémoire utilisée pour effectivement stocker des objets dans le tableau.
Si l'on a trop d'éléments à rajouter – i.e. que la taille logique dépasse la taille physique, alors on procède à une réallocation de mémoire, mais là aussi en prenant une «marge», de façon à ce que les ajouts suivants ne nécessitent pas de réallocation.
Typiquement, la marge consiste à multiplier la taille physique par une constante $a \in]1, 2]$ à chaque fois qu'elle est atteinte.
Les listes *Python* sont implémentées par des tableaux dynamiques (en tout cas dans les versions couramment utilisées).
La première figure ci-dessous montre les temps d'exécutions empiriques obtenus par des ajouts successifs d'éléments à une liste (initialement vide). Noter l'échelle logarithmique en ordonnées.



Les figures suivantes donnent les temps d'exécution amortis – le point d'abscisse x représente le temps moyen pour rajouter les x premiers éléments au tableau. En moyenne, le temps d'ajout d'un nouvel élément est constant, ce qui correspond bien au comportement attendu théoriquement pour un tableau dynamique.





2.3 Conséquences en Python

2.3.1 Ajout en fin de tableau

On peut utiliser la commande *append* pour ajouter un élément à une liste (comme indiqué dans la section sur les tableaux dynamiques).

Code Python :

```
>>> t=[0, 1, 2, 3, 4]
>>> t.append(5)
>>> t
[0, 1, 2, 3, 4, 5]
```

La liste est effectivement *modifié* par cette commande. Le code ci-dessous produit apparemment le même effet, mais il ne se passe pas du tout la même chose en machine !

Code Python :

```
>>> t=[0, 1, 2, 3, 4]
>>> t=t+[5]
>>> t
[0, 1, 2, 3, 4, 5]
```

Ici, à l'exécution de `t=t+[5]`, un nouveau tableau correspondant au membre de droite de l'affectation est créé, et il est ensuite affectée à t. On perd donc du temps (complexité linéaire pour la création du nouveau tableau, au lieu de constante pour un simple ajout en fin de tableau), et de l'espace (2 tableaux cohabitent en mémoire lors de l'exécution de ce code).

2.3.2 Autres opérations

De nombreuses autres opérations sont possibles sur les listes : concaténation, insertion, suppression. Il faut avoir conscience que ces opérations cachent potentiellement en machine un travail important de recopie de tableaux, qui peut prendre du temps [10].

Insertion d'un élément

Code Python :

```
>>> t=[i for i in range(10)]
>>> t
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t.insert(2,-1) #insertion d'un élément
>>> t
[0, 1, -1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Voici par exemple deux petits tests de temps d'exécution pour une insertion en début de liste, qui suggèrent que la complexité de cette opération est linéaire en la longueur de la liste.

10. La norme *Python* ne fixe pas de complexité, mais on peut partir du principe général que toute opération qui modifie la taille d'une liste – autre que l'ajout ou la suppression à la fin de la liste – risque d'avoir une complexité linéaire en la taille de la liste.

Code Python :

```
>>> from time import time

>>> a=10*[0]
>>> c=time() ;a.insert(0,0) ;time()-c
1.9311904907226562e-05

>>> b=10**7*[0]
>>> c=time() ;b.insert(0,0) ;time()-c
0.04149270057678223
```

Concaténation de deux listes

Code Python :

```
>>> [1, 2, 3]+[4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

Complexité : linéaire en la somme des tailles des deux listes.

Remplacement d'une tranche

Code Python :

```
>>> t=[i for i in range(10)]
>>> t
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t[2 :3]=[-1, -2, -3] #remplacement de la tranche 2 :3
>>> t
[0, 1, -1, -2, -3, 3, 4, 5, 6, 7, 8, 9]

>>> t=[i for i in range(10)]
>>> t
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t[2, 2]=[-1, -2, -3] #insertion en position 2 de plusieurs éléments
>>> t
[0, 1, -1, -2, -3, 2, 3, 4, 5, 6, 7, 8, 9]
```

Test d'égalité. Le test d'égalité (avec ==) est également de complexité linéaire dans le cas le pire.

Code Python :

```
>>> from time import time
>>> a=10*[0]
>>> c=time() ;a==a ;time()-c
True
5.626678466796875e-05

>>> b=10**7*[0]
>>> c=time() ;b==b ;time()-c
True
0.08208465576171875
```

Suppression d'un élément `l.remove(e)` supprime la première occurrence de `e` dans la liste `l`.

Code Python :

```
>>> a = [1, 0, 1, 2, 3, 2, 1]
>>> a.remove(2)
>>> a
[1, 0, 1, 3, 2, 1]
```

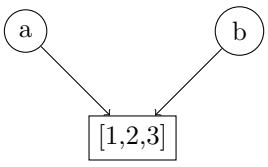
3 Copie, passage en paramètre d'une fonction

3.1 Exemple introductif

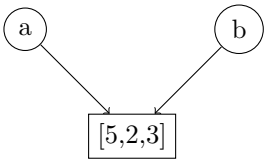
Code Python :

```
>>> a=[1, 2, 3]
>>> b=a          #2
>>> a[0]=5       #3
>>> b
[5, 2, 3]
```

lorsqu'on écrit $b = a$, où a est un tableau, cela n'effectue pas une copie de la liste. b devient simplement un nouveau nom pour désigner la même liste (on parle d'*alias*, ou de *référence*). Cette opération est effectuée en temps *constant*. (Alors qu'une copie «physique» devrait allouer un nouvel espace mémoire et recopier un à un les éléments de la liste dedans – la complexité serait alors linéaire.)
Graphiquement, on peut voir l'instruction $b = \ldots$ comme un simple fléchage entre la variable b et l'objet situé à droite du symbole d'affectation.



(a) Après la ligne 2 : a et b sont deux références pour un même objet en mémoire

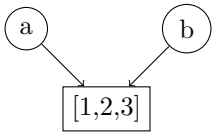


(b) Après la ligne 3

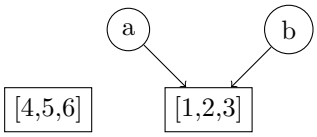
Attention, la situation est différente dans l'exemple suivant :

Code Python :

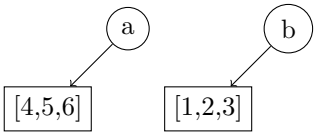
```
>>> a=[1, 2, 3]
>>> b=a              #(2)
>>> a=[4, 5, 6]       #(3)
>>> b
[1,2,3]
```



(a) Après la ligne 2



(b) À la création de [4,5,6] en mémoire



(c) Après la ligne 3

En cas de doute sur ce qui se passe en machine, on peut utiliser la commande `is`. `a is b` renvoie `True` si et seulement si les variables a et b référencent le même objet en mémoire. Cela s'effectue en temps constant – il s'agit essentiellement d'une comparaison d'adresses mémoires, et il est inutile de parcourir les listes.

Code Python :

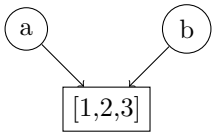
```
>>> a=[1,2,3]
>>> b=a
>>> a is b
True
>>> a=[2,3,4]
>>> a is b
False
>>> a=[1,2,3]
>>> c=[1,2,3]
>>> a is c
False
```

3.2 Ajout d'un élément

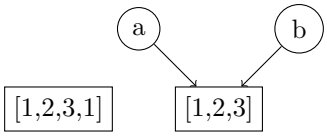
Cela a des conséquences sur la façon d'ajouter un élément à une liste.

Code Python :

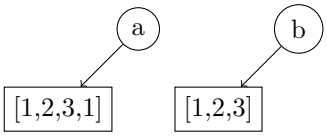
```
>>> a=[1, 2, 3]
>>> b=a             #2
>>> a=a+[1]         #3
>>> b
[1, 2, 3]
```



(a) Après la ligne 2



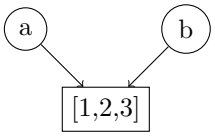
(b) À la création de a+[1] en mémoire



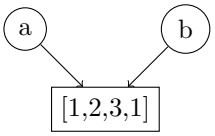
(c) Après la ligne 3

Code Python :

```
>>> a=[1, 2, 3]
>>> b=a            #2
>>> a.append(1)    #3
>>> b
[1, 2, 3, 1]
```



(a) Après la ligne 2



(b) Après la ligne 3

3.3 Copie d'une liste

Si l'on veut *recopier* une liste, il faut utiliser l'une des syntaxes suivantes :

- `b=a[ :]` (extraction de tranche - toute la liste est extraite, et une copie est effectuée).
- `b=copy(a)` (après avoir exécuté `from copy import copy`).
- `b = [e for e in a]`.
- `b = [a[k] for k in range(len(a))]`.

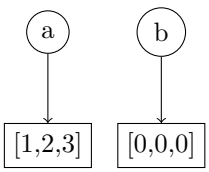
Code Python :

```
>>> a=[1, 2, 3]
>>> b=a[ :]     # copie physique du tableau
>>> a[2]=7
>>> b
[1, 2, 3]
```

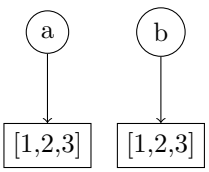
ce qui revient sensiblement au même que de le faire «à la main» :

Code Python :

```
>>> a=[1, 2, 3]
>>> b=[0 for i in range(len(a))]
>>> for i in range(0,len(a)) :        #2
        b[i]=a[i]                    #3
```



(a) Après la ligne 2



(b) Après la ligne 3

Attention, la complexité d'une copie est linéaire, et après la copie, on a deux listes en mémoire. Si l'on a de grandes listes (des images par exemple), la copie est à éviter si possible.

3.4 Passage d'une liste en paramètre d'une fonction

Lorsqu'une liste est passée en argument d'une fonction, il n'est pas recopié, la variable locale correspondante est simplement une référence vers la liste passée en argument. Conséquence : **toute modification d'un élément d'une liste effectuée à l'intérieur d'une fonction est également effectuée à l'extérieur.**
Voici un exemple de fonction qui ne renvoie aucun résultat, mais modifie la liste qui lui est passée en argument [11] :


Code Python :

```
def somme_cumulée(t) :
    for i in range(1,len(t)) :
        t[i]=t[i]+t[i-1]


>>> a=[1, 2, 3, 4, 5, 6]
>>> somme_cumulée(a)
>>> a
[1, 3, 6, 10, 15, 21]
```

Attention, ce comportement est souvent indésirable si on ne l'a pas anticipé... Il est parfois nécessaire d'effectuer une copie de la liste pour y remédier.

4 Tableaux à deux dimensions

Les tableaux à deux dimensions ont de nombreuses applications. Ils peuvent par exemple servir à représenter des **matrices** ou des **images**.
En machine, un tableau à deux dimensions est simplement implémenté par une liste de listes :


Code Python :

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

>>> m=[[(-1)**(k+j) for j in range(3)] for k in range(5)]
>>> m
[[1, -1, 1], [-1, 1, -1], [1, -1, 1], [-1, 1, -1], [1, -1, 1]]

>>> m[1][2]=3
>>> m
[[1, -1, 1], [-1, 1, 3], [1, -1, 1], [-1, 1, -1], [1, -1, 1]]
```

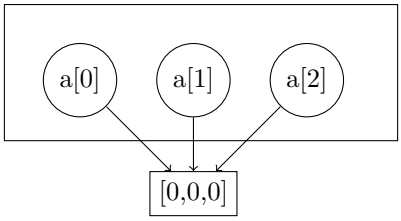
Attention à l'erreur classique suivante :

11. On parle d'«effet de bord» (*side effect*, qui serait mieux traduit par *effet secondaire*) pour tout action d'une fonction qui a une portée non-locale (autre qu'un `return`). Pour l'instant, le seul effet de bord que nous avions vu était l'utilisation d'un `print` à l'intérieur d'une fonction – cas un peu particulier, puisqu'il était impossible de faire quoi que ce soit de ce qui était affiché par le `print` à l'extérieur de la fonction.

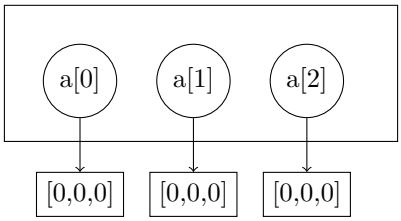
Code Python :

```
>>> a=3*([3*[0]])
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> a[0][0]=1
>>> a
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
```

`3*[0]` créé bien une liste à 3 éléments nuls. En revanche, `3*(3*[0])` n'effectue pas des copies physiques de cette liste. Les trois éléments de a sont des références vers la même liste en machine.
On a, en mémoire, le schéma suivant :



À comparer avec le schéma attendu :



Les tableaux à deux dimensions posent des problèmes supplémentaires par rapport aux copies :

Code Python :

```
>>> from copy import copy

>>> a=[[0 for k in range(3)] for k in range(3)]
>>> b=copy(a)
>>> a[1][2]=1

>>> b
[[0, 0, 0], [0, 0, 1], [0, 0, 0]]
```

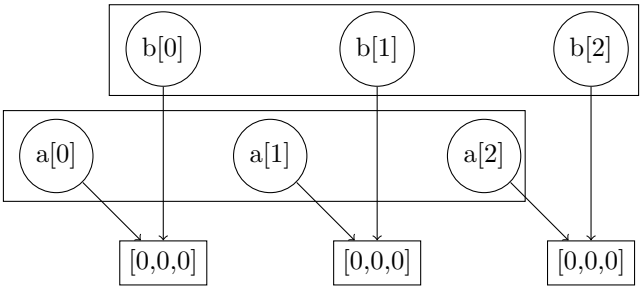


FIGURE 6 : Situation après la copie

Pour y remédier, il faut faire une *copie profonde* (par opposition à la *copie superficielle* effectuée par `copy`) :

Code Python :

```
>>> from copy import deepcopy

>>> a=[[0 for k in range(3)] for k in range(3)]
>>> b=deepcopy(a)
>>> a[1][2]=1

>>> b
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

5 Pour aller plus loin : manipulations avancées avec les listes

Voici quelques mécanismes pour aller plus loin sur les listes.

5.1 Définition en compréhension, filtre

Nous avons déjà vu ce genre de définitions, semblables aux définitions en compréhension des ensembles mathématiques :

Code Python :

```
>>> [a**2 for a in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Il est également possible d'ajouter une condition, en utilisant la syntaxe suivante :

Code Python :

```
>>> [k**2 for k in range(10) if k % 2 == 0]
[0, 4, 16, 36, 64]
```

Cette syntaxe permet de très facilement *filtrer* une liste, c'est-à-dire d'en extraire les éléments vérifiant telle ou telle propriété. Le code suivant permet par exemple d'extraire les éléments pairs de la liste `t`.

Code Python :

```
>>> t = [1, 3, 4, 5, 2, 7, 5, 8, 6]
>>> t2 = [e for e in t if e % 2 == 0]
[4, 2, 8, 6]
```

La commande `filter` permet de faire la même chose, mais le code obtenu est souvent beaucoup moins lisible :

Code Python :

```
def estpair(x) :
    return x % 2 == 0

t2 = list(filter(estpair, t))
```

ou encore, en utilisant une *fonction anonyme*[12] :

Code Python :

```
t2 = list(filter(lambda x  : x % 2 == 0, t))+.
```

5.2 «Mapping»

Cette même syntaxe permet de «mapper» une fonction sur une liste, i.e. d'appliquer cette fonction à chaque élément d'une liste.

Code Python :

```
>> t = [1, 4, 3, 6, 7]
>> t2 = [2*e for e in t]
>> t2
[2, 8, 6, 12, 14]
```

On peut également utiliser la commande `map` pour faire cela, mais le résultat est ici aussi beaucoup moins lisible...

Code Python :

```
t2 = list(map(lambda x  : 2 * x, t))
```

5.3 zip

La commande `zip` permet d'itérer simultanément sur plusieurs listes en même temps, ce qui peut s'avérer très utile en pratique.

Code Python :

```
>>> list(zip([1, 2, 3],[4, 5, 6]))
[(1, 4), (2, 5), (3, 6)]
```

Code Python :

```
>>> t1 = [1, 2, 3]
>>> t2 = [4, 5, 6]
>>> for a,b in zip(t1,t2) :
>>>     print (a+b)
5
7
9
```

12. c'est-à-dire une fonction sans nom, ce qui est utile par exemple lorsque la fonction n'est destinée à n'être utilisée qu'une fois. On utilise pour cela le mot clef `lambda`

Code Python :

```
>>> t1 = [1, 2, 3]
>>> t2 = [4, 5, 6]
>>> t3 = [a+b for a,b in zip(t1,t2)]
>>> t3
[5, 7, 9]
```

6 Exercices classiques sur les tableaux

1. Écrire une fonction `puissances`, qui prend en argument un entier n, et renvoie la liste consituée des n premières puissances de 2 (en partant de 2^0).

Code Python :

```
def puissances(n) :
    r = []
    p = 1
    for k in range(n) :
        r.append(p)
        p = p * 2
    return r
```

ou bien

```
def puissances(n) :
    return [2**k for k in range(n)]
```

2. Même question, mais on veut la liste dans l'autre sens.

Code Python :

```
def puissancesinv(n) :
    return [2**(n-1-k) for k in range(n)]
```

3. Écrire une fonction `ajouteun`, qui prend en argument une liste, et ajoute 1 à chaque élément de la liste (elle ne doit rien renvoyer).

Code Python :

```
def ajouteun(l) :
    for k in range(len(l)) :
        l[k] = l[k] + 1
```

4. Écrivez et testez une fonction `sommeListes(t1, t2)`, qui prend en argument deux listes de même longueur, et renvoie la liste «somme» des deux (par exemple, `sommeListes([1,2,3],[2,3,-1])` devra renvoyer `[3,5,2]`).

Code Python :

```
def sommeListes(t1, t2) :
    r = []
    for k in range(len(t1)) :
        r.append(t1[k]+t2[k])
    return r
```

5. Écrire une fonction qui prend en argument un entier $n \in \mathbb{N}^*$, et renvoie la matrice identité de taille n (sous forme de listes de listes).

Code Python :

```
def identite(n) :
    r = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n) :
        r[i][i] = 1
    return r
```

6. Écrire une fonction qui prend en argument une matrice (sous forme de liste de listes), et renvoie sa transposée [13].

13. En utilisant les mécanismes plus avancés de la section précédente, on peut utiliser le beaucoup plus compact `list(zip(*l))` – `*` transformant une liste en «séquence» d'arguments que l'on peut passer à une fonction.

Code Python :

```
def transpose(m) :
    n = len(m)
    p = len(m[0])
    r = [[0 for _ in range(n)] for _ in range(p)]
    for i in range(n) :
        for j in range(p) :
            r[j][i] = m[i][j]
    return r
```

7. Écrire une fonction `triangledepascal(n)` qui prend en argument un entier n non-nul, et renvoie, sous forme de listes de listes, les n premières lignes du triangle de Pascal (en partant de la ligne 1). On utilisera la formule de Pascal $\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$ pour calculer les coefficients de proche en proche.
   Pour $n = 4$, elle devra renvoyer
   `[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]`
   qui correspond au tableau suivant :

1			
1	1		
1	2	1	
1	3	3	1

Code Python :

```
def triangledepascal(n) :
    r=[[1]]
    for i in range(1, n) :
        l=[1]
        for j in range(1, i) :
            l.append(r[i-1][j] + r[i-1][j-1])
        l.append(1)
        r.append(l)
    return r
```

8. (a) Écrire une fonction qui prend en argument une liste s d'entiers de 0 à 9 et renvoie la liste de longueur 10 dont la $i^{\text{ème}}$ case contient le nombre d'occurrences de i dans s.

Code Python :

```
def compte(t) :
    c = [0 for _ in range(10)]
    for e in t :
        c[e] = c[e]+1
    return c
```

(b) Écrire une fonction qui prend en argument une liste comme dans la question précédente, et renvoie l'un des entiers qui y apparait le plus de fois.

Code Python :

```
def nombremax(t) :
    c = compte(t)
    r = 0
    for k in range(1, 10) :
        if c[k] >= c[r] :
            r = k
    return r
```

7 Thèmes autour des listes

7.1 Algorithmes de tri

On travaillera dans le fichier `algostri.py`, qui contient des fonctions graphiques permettant de voir une animation des algorithmes de tri.

On s'intéresse dans cette section à des algorithmes qui prennent en entrée une liste de réels ou d'entiers, et le modifient ou renvoient une nouvelle liste contenant les mêmes éléments (avec leur multiplicité) classés par ordre croissant.
Il s'agit d'une tâche algorithmique basique, déjà implémentée en *Python*[14] :

Code Python :

```
>>> a=[1,2,3,1,2,1,6,1,2]
>>> a.sort()
>>> a
[1, 1, 1, 1, 2, 2, 2, 3, 6]
```

7.1.1 Tri à bulles

Le tri à bulles consiste en des *passes* successives. À chaque passe, la liste est parcourue de gauche à droite (dans le sens des indices croissants), et tout couple d'éléments adjacents «dans le mauvais sens» sont intervertis.
On continue jusqu'à ce que la liste soit triée.

1. Écrire et tester une fonction `esttrie(l)`, qui renvoie `True` si et seulement si `l` est trié.

Code Python :

```
def esttrie(l) :
    for k in range(len(l)-1) :
        if l[k] > l[k+1] :
            return False
    return True
```

2. Écrire une fonction `tribulle(l)`, qui effectue un tri à bulles sur une liste `l`.

Code Python :

```
def tribulle(l) :
    while not esttrie(l) :
        for k in range(len(l)-1) :
            if l[k] > l[k+1] :
                (l[k], l[k+1]) = (l[k+1], l[k])
```

14. `sort` est basé sur l'algorithme *Timsort*, qui est une mélange de tri fusion et de tri par insertion.

3. Testez votre fonction sur une liste aléatoire, généré par exemple à l'aide de `l=[random()*100 for _ in range(30)]`

Code Python :

```
  l=[random()*100 for _ in range(30)]
print(l)
tribulle(l)
print(l)
```

4. Insérer la commande `trace(l)` dans votre fonction à chaque modification de `l`, afin de visualiser sous forme d'une animation le déroulement de l'algorithme.

Code Python :

```
def tribulle(l) :
    while not esttrie(l) :
        for k in range(len(l)-1) :
            if l[k] > l[k+1] :
                (l[k], l[k+1]) = (l[k+1], l[k])
                trace(l)
```

On peut montrer que la complexité de cet algorithme est $O(n^2)$.

Rem

- Une variante possible : le tri *boustrophédon*, qui consiste en l'alternance de passes de la gauche vers la droite et de la droite vers la gauche.
- Ces algorithmes sont très mauvais en terme de temps d'exécution, et sont peu utilisés en pratique.

7.1.2 Tri par sélection

Le tri par sélection consiste à trouver le maximum de la liste, le permuter avec le dernier élément de la liste, trouver le deuxième plus grand élément de la liste, le permuter avec l'avant-dernier élément, et ainsi de suite.

1. Écrire et tester une fonction `maximum(l,i)`, qui renvoie l'indice où se trouve l'élément maximal de la liste `l` situé avant l'indice `i` (avant au sens strict).

Code Python :

```
def maximum(l,i) :
    m = 0
    for k in range(i) :
        if l[m] < l[k] :
            m = k
    return m
```

2. Écrire une fonction `triselection(l)`, qui effectue un tri par sélection sur `l`.

Code Python :

```
def triselection(l) :
    n = len(l)
    for k in range(n) :
        m = maximum(l, n-k)
        (l[m], l[n-1-k]) = (l[n-1-k], l[m])
        trace(l)
```

3. Tester, et visualiser l'animation comme pour le tri à bulles.
4. Montrer que la complexité de cet algorithme est $O(n^2)$ (on comptera le nombre de comparaisons).

Le nombre de comparaisons effectuées lors de l'appel de `maximum(l,i)` est i.
Le nombre de comparaisons effectuées par `triselection` sur une liste de longueur n est donc $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$.
La complexité est donc quadratique.

7.1.3 Tri par insertion

Le tri par insertion d'une liste `l` de longueur n se déroule en n passes. À l'issue de la $k^{\text{ème}}$ passe, la sous-liste à k éléments commençant au début de `l` est trié.
Lors de la passe $k+1$, on regarde `l[k+1]`, et on l'insère à la position correcte dans la sous-liste trié situé à sa gauche. Pour ce faire, on peut le permuter successivement avec son voisin de gauche, jusqu'à ce qu'il soit bien positionné.

Liste de départ

3	1	2	-1	6	5

Après la passe 1 (l'élément inséré est en rouge)

1	3	2	-1	6	5

Après la passe 2 (l'élément inséré est en rouge)

1	2	3	-1	6	5

Après la passe 3 (l'élément inséré est en rouge)

-1	1	2	3	6	5

...

1. Implémenter, tester, et visualiser le déroulement du tri insertion.

Code Python :

```
def triinsertion(l) :
    n = len(l)
    for i in range(1,n) :
        k = i
        while l[k] < l[k-1] and k > 0 :
            (l[k], l[k-1]) = (l[k-1], l[k])
            k = k-1
            trace(l)

l=[random()*100 for _ in range(30)]
print(l)
triinsertion(l)
print(l)
```

7.1.4 Tri fusion

Plus difficile !
Le tri fusion est un algorithme récursif :

- Si la liste n'a qu'un élément, il est déjà trié.
- Sinon, on sépare la liste en deux parties de longueurs égales à 1 près.
- On trie récursivement les deux parties avec l'algorithme de tri fusion.
- On fusionne les deux listes triées en une liste triée.

On prendra soin de tester les fonctions écrites au fur et à mesure.

1. Écrire une fonction `separe(l)` qui prend en argument une liste de longueur au moins 2, et renvoie un couple de 2 listes correspondant aux 2 moitiés (à 1 élément près) de la liste `l`.
2. Écrire une fonction `fusionne(l1,l2)` qui prend en argument 2 listes triés de même longueur (à 1 élément près), et renvoie la liste fusionnée triée correspondante, en *temps linéaire* par rapport aux listes de départ.
3. Écrire enfin une fonction `trifusion(l)`, qui prend en entrée une liste l et effectue un tri fusion sur `l` (elle renverra une nouvelle liste en sortie).

Code Python :

```
def separe(l) :
    n = len(l)
    return (l[ :n//2], l[n//2 :])

def fusionne(l1, l2) :
    (i1, i2) = (0, 0)
    r = []
    while i1 < len(l1) or i2 < len(l2) :
        if i2 == len(l2) or (i1 < len(l1) and l1[i1] < l2[i2]) :
            r.append(l1[i1])
            i1 = i1 + 1
        else :
            r.append(l2[i2])
            i2 = i2 + 1
    return r

def trifusion(l) :
    if len(l) > 1 :
        (l1, l2) = separe(l)
        return fusionne(trifusion(l1), trifusion(l2))
    return l

l=[random()*100 for _ in range(30)]
print(l)
trifusion(l)
print(l)
```

NB : on peut montrer que le tri fusion est de complexité quasi-linéaire $O(n\log(n))$, et que cette complexité est optimale pour un algorithme de tri général (n'utilisant que des comparaisons entre les éléments de la liste).

7.2 Jeu de la vie

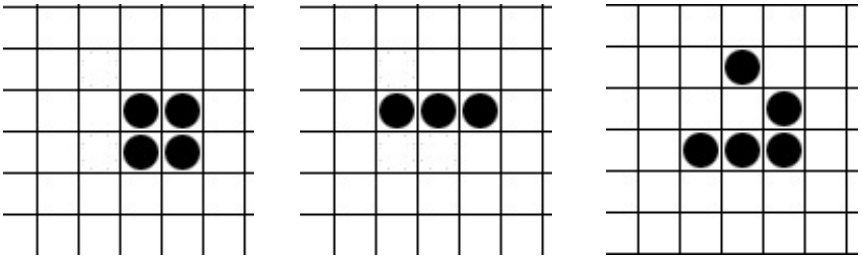
Le *jeu de la vie* est un *automate cellulaire* très étudié [15], dans lequel on simule l'évolution d'une population de cellules virtuelles, évolution qui suit un certain nombre de règles détaillées ci-dessous.
On se place sur une grille de taille $n \times n$. À un moment donné, chaque case peut contenir ou non une cellule.
Le voisinage d'une case est constitué des 8 cases voisines – avec les conventions que les bords de la grille se «recollent» : ainsi les cases voisines de $(0,0)$ sont $(0,1),(1,1),(1,0),(1,n-1),(0,n-1),(n-1,n-1),(n-1,0),(n-1,1)$ [16].
Pour passer d'une génération de cellules à la génération suivante, on utilise les règles suivantes.

- S'il y a une cellule dans une case, et qu'il y a exactement 2 ou 3 cellules dans son voisinage, elle survit. Sinon, elle disparait. (Mort par isolement ou étouffement.)
- Si une case est vide et a exactement 3 cellules dans son voisinage, une cellule y apparait. Sinon, la case reste vide. (Naissance.)

La grille sera représentée par une liste de listes d'entiers. Un coefficient vaudra 1 ou 0 selon qu'il y a ou non une cellule dans la case correspondante.
Dans la suite, vous pourrez directement travailler dans le fichier `jeudelavie.py` fourni.

1. Étudier à la main l'évolution des 3 configurations suivantes (un point fixe, un «clignotant» et un «glisseur») :

15. On trouve facilement des articles et des simulateurs sur internet. Un point de dé2part : https://fr.wikipedia.org/wiki/Jeu_de_la_vie
16. On travaille donc sur un espace de topologie torique.



La premier configuration est fixe (elle ne change pas d'une génération à la suivante).
La seconde «clignote» sur 2 générations.
Pour la troisième, on retrouve la même configuration 3 générations plus tard, mais décalée :
on obtient donc l'illusion d'un déplacement (cette figure est d'ailleurs appelée «glisseur»).

2. En utilisant la commande `random()` du module *random* – qui renvoie un réel choisi uniformément au hasard entre 0 et 1 – compléter la fonction `configuration_initiale(n,p)` de façon à ce qu'elle renvoie une grille $n \times n$ dont chaque case contient indépendamment des autres une cellule avec probabilité p.

Code Python :

```
def configuration_initiale(n,p) :
    m=[[0 for x in range(n)] for x in range(n)]
    for i in range(n) :
        for j in range(n) :
            if random()<p :
                m[i][j]=1
    return m
```

3. Compléter la fonction `nbvoisins(g,i,j)` afin qu'elle renvoie le nombre de cellules dans le voisinage de la case (i, j) sur la grille `g`.

Code Python :

```
def nbvoisins(m,i,j) :
    s=0
    n=len(m)

    for k in range(-1,2) :
        for l in range(-1,2) :
            if k !=0 or l !=0 :
                s=s+m[(i+k) % n][(j+l) % n]
    return s
```

4. Compléter la fonction `configuration_suivante(g)` afin qu'elle renvoie une nouvelle grille calculée en avançant d'un pas de temps à partir de la grille `g`. Attention, les règles décrites ci-dessus doivent s'appliquer simultanément à toutes les cases !

Code Python :

```
def configuration_suivante(m) :
    n=len(m)
    r=[[0 for x in range(n)] for x in range(n)]
    for i in range(n) :
        for j in range(n) :
            if m[i][j]==1 and nbvoisins(m,i,j) in [2,3] :
                r[i][j]=1
            if m[i][j]==0 and nbvoisins(m,i,j) == 3 :
                r[i][j]=1
    return r
```

5. Tester enfin la fonction `jeudelavie(n,p)` fournie.

7.3 Autour des images

7.3.1 Les images en Python

Il existe plusieurs bibliothèques Python permettant de manipuler les images. Nous allons utiliser «Python Image Library» (PIL pour les intimes). Voici les commandes qui vous seront utiles :

- `from PIL.Image import *` importe une fois pour toute les fonctions suivantes.
- `im=open('chemin/vers/fichier.jpg')` charge une image.
- `im.save('chemin/vers/fichier.jpg')` sauvergarde une image.
- `im.show()` affiche une image.
- `im.width` et `im.height` renvoient la largeur et la hauteur de l'image en pixels.
- `r=new("L",(im.width,im.height),"black")` créé une nouvelle image noire, de largeur et hauteur indiquées.
- `c=Image.getpixel(im,(x,y))` renvoie la valeur du pixel de coordonnées entières (x, y) de l'image im. Pour une image en niveaux de gris, le résultat renvoyé est un entier compris entre 0 (noir) et 255 (blanc).
- `Image.putpixel(im, (x, y), c)` met à c (un entier entre 0 et 255) le pixel (x, y) de l'image im.

7.3.2 Débruitage

Dans toute la suite, on ne «s'embêtera pas trop» avec les bords de l'image...

On considère une image bruitée (`benchbruite.jpg`, `geckobruite.jpg` ou `oiseaubruite.jpg`).
On propose les méthodes suivantes pour la débruiter.

Filtre moyen on remplace la valeur de chaque pixel par la moyenne des valeurs des 9 pixels adjacents (en comptant le pixel lui-même).

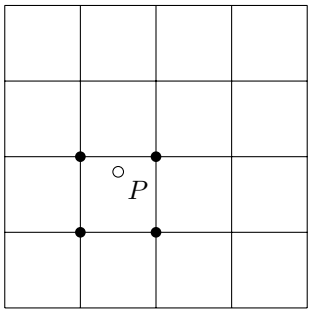
> Les corrections sont dans le fichier *corrimages.py*.

Filtre médian on remplace la valeur de chaque pixel par la médiane des valeurs des 9 pixels adjacents (en comptant le pixel lui-même). (Pour trier une liste, ou pourra utiliser la partie sur les algorithmes de tri, ou bien la commande `l.sort()`.)

Implémenter et tester ces 2 méthodes sur les différentes images (pour le filtre moyen, on pourra essayer de l'appliquer plusieurs fois d'affilée).

7.3.3 Interpolation linéaire

Il est parfois nécessaire d'accéder au niveau de gris d'une image en un point P de coordonnées non-entière (designé par le cercle dans l'image ci-dessous). On va alors dire que ce niveau de gris est une combinaison linéaire des niveaux de gris des 4 plus proches points à coordonnées entières (les cercles pleins), en se débrouillant pour que le poids d'un point entier soit d'autant plus important que P est proche de ce point (on utilisera les coordonnées barycentrique de P par rapport aux 4 points voisins).



Écrire la fonction `linear(im,x,y)` correspondante (en particulier, si x et y sont entiers, elle devra renvoyer la valeur exacte du niveau de gris en (x, y)...).

7.3.4 Application à la rotation d'image

On souhaite écrire une fonction `rotation(im,theta)`, qui prend en argument une image et un angle, et renvoie l'image déduite de `im` par rotation d'angle `theta` par rapport au centre de `im`.
L'image r renvoyée sera de même taille que l'image de départ. Pour chaque pixel p de l'image r :

- on calcule son image p' par la rotation d'angle $-\theta$;
- si p' sort des bords de l'image, on laisse p en noir ;
- sinon, le niveau de gris de p est celui de p' dans l'image initiale.

Coder deux versions de cette fonction :

- l'une où l'on prend pour niveau de gris de p le niveau de gris du point entier le plus proche de p ;
- l'autre où l'on utilise une interpolation linéaire pour déterminer ce niveau de gris.

Comparer les résultats obtenus (on pourra commencer par regarder ce que cela donne sur `bench.jpg`).

FIGURE 7 : Zoom sur des résultats donnés par la rotation naïve, et celle utilisant l'interpolation linéaire. Noter l'effet d'«aliasing» produit par la rotation naïve.

7.4 Initiation aux objets et classes, exemples d'applications en Mathématiques

Dans cette partie, on introduit au travers de trois exemples mathématiques la composante «programmation orientée objet» du langage *Python*, dont une compréhension basique est utile pour aborder un certain nombre de modules *Python*, ou pour se lancer dans des projets un peu plus complexes : la programmation orientée objet permet en effet d'écrire du code très lisible et modulaire. Il s'agit d'une couche essentielle de nombreux langages actuels.

7.4.1 Objets, classes

En programmation orientée objet» on manipule des entités appelées «objets» appartenant à certaines «classes». Une classe est une nouvelle structure de données (personnalisée, ou importée via un module), qui va supporter un certains nombres d'opérations.
Voici un exemple minimal de classe pour décrire des vecteurs du plan :

Code Python :

```
class Vecteur :
    def __init__(self,x,y) :
        self.x=x
        self.y=y

    def norme(self) :
        return math.sqrt(self.x**2+self.y**2)
```

et une utilisation possible :

Code Python :

```
>>> v=Vecteur(2,3)
>>> v.x
2
>>> v.y=4
>>> v.norme()
4.47213595499958
```

Détaillons un peu :

- À l'intérieur d'une classe, on trouve un certain nombre de fonctions, appelées *méthodes*. Leur premier argument est toujours `self`, qui désigne l'objet sur lequel la méthode est appelée. Si o est un objet, on appelera une méthode sur cet objet à l'aide de la syntaxe `o.methode(arguments...)`, et non pas `methode(o,arguments...)`.
- `__init__` est une méthode un peu particulière – appelée *constructeur*. C'est elle qui est appelée lors de la création d'un objet (via `Vecteur(2,3)` dans notre exemple). Ici, elle se borne à affecter les valeurs passées en argument aux deux variables `x` et `y` attachées à l'objet (on parle d'*attributs*).
- `norme` est une méthode qui renvoie la norme du vecteur en question.

En *Python*, une variable contenant un objet est une référence vers cet objet, ce qui explique que les objets aient le même type de comportements que les listes que nous avons vu au début de ce poly.

Code Python :

```
>>> v=Vecteur(2,3)
>>> u=v
>>> v.x=4
>>> u.x
4
```

7.4.2 Calculs exacts dans $\mathbb{Q}$

Le fichier `rationnel.py` contient une classe qui permet d'implémenter les nombres rationnels avec une précision arbitraire.
Une fois celui-ci placé dans un l'un des répertoires des modules (dont on obtient la liste à l'aide de `import sys ; print(sys.path)` ;[17]), on peut l'importer à l'aide de la commande `from rationnel import *`.

Exemple d'utilisation de ce module (Méthode de Héron). On considère un entier $a \in \mathbb{N}^*$, et la suite $(u_n) : \begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}, u_{n+1} = \dfrac{1}{2}(u_n + a/u_n) \end{cases}$.
On peut montrer qu'il s'agit d'une suite à termes rationnels, qui converge très rapidement [18] vers $\sqrt{a}$. Pour $a = 2$ par exemple, on peut montrer que u_{10} donne une valeur approchée de $\sqrt{2}$ avec 400 décimales exactes.
En utilisant le module *rationnels*, on peut utiliser le code suivant pour obtenir les 400 premières décimales de $\sqrt{2}$:

Code Python :

```
from rationnel import *

def heron(a,n) :
    """calcule le n eme terme de la suite de Héron,
    où u0=a est un entier naturel non-nul"""
    u=Rationnel(1)          # créé un nouveau rationnel = 1
    for _ in range(n) :
        u=Rationnel(1,2)*(u+Rationnel(a)/u)
          # Rationnel(1,2) correspond à 1/2
    return u

r=heron(2,10)
r.affiche(400)
```

```
1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415727350138462309122970249248360558507372126441214970999358314132226659275055927557999505011527820605714701095599716059702745345968620147285174186408891986095523292304843087143214508397626036279952514079896872533965463318088296406206152583523950547457502877599617298355752203375318570113543746034084988471
```

Quelques explications :

- `Rationnel(n)` créé un nouvel objet «nombre rationnel» correspondant à l'entier n.
- `Rationnel(n,m)` créé un nouvel objet «nombre rationnel» correspondant à $\dfrac{n}{m}$, où n et m sont deux entiers relatifs.
- On peut utiliser les opérations arithmétiques usuelles entre nombres rationnels [19].
- `r.affiche(d)` affiche l'écriture décimale du rationnel r, tronquée à `d` chiffres après la virgule.

Description du module Nous allons simplement détailler les éléments utilisés dans l'exemple ci-dessus.

On commence par déclarer la nouvelle classe :

17. si l'on n'a pas accés en écriture à ces répertoires, on peut toujours modifier le «path» à l'aide de la commande `sys.path.append("chemin complet vers le répertoire contenant rationnel.py")`
18. quadratiquement
19. Il serait possible de simplifier la syntaxe afin de pouvoir directement ajouter des entiers et des rationnels – cela nécessiterait d'étoffer un peu la classe décrite ci-après.

Code Python :

```
class Rationnel :
```

On écrit ensuite le *constructeur* (toujours nommé `__init__`). Ce sera la fonction appelée lorsqu'on veut créer un nouveau nombre rationnel, via `Rationnel(a)`, ou `Rationnel(a,b)`.
Le principal travail de ce constructeur est d'initialiser les deux *attributs* correspondant au numérateur et au dénominateur. Ces deux attributs sont des entiers, et donc ont un nombre de chiffres arbitrairement grand.

Code Python :

```
def __init__(self,a,b=1) :
    """constructeur - code appelé lors de la construction d'un nouveau rationnel"""
    self.numerateur=a
    if b==0 :
        raise ValueError('Un dénominateur ne peut pas être nul')
    self.denominateur=b
```

(À noter également : le `b=1`, qui donne une valeur par défaut à l'argument, et le `raise`, qui permet de lever une exception en cas d'erreur.)

On écrit ensuite une fonction de conversion en chaîne de caractères (toujours nommé `__str__`), qui sera par exemple utilisée par `print` lorsqu'on voudra afficher un rationnel. Le rationnel est mis au passage sous forme irréductible, via la méthode `simplifie`.

Code Python :

```
def simplifie(self) :
    """met le nombre rationnel sous forme irréductible"""
    g=gcd(self.numerateur,self.denominateur) # gcd=pgcd
    self.numerateur//=g
    self.denominateur//=g
    if self.denominateur<0 :
        self.denominateur*=-1
        self.numerateur*=-1

def __str__(self) :
    """conversion en chaine de caractères (utilisée par print)"""
    self.simplifie()
    return str(self.numerateur)+'/'+str(self.denominateur)
```

On peut également définir une autre fonction d'affichage, permettant d'obtenir l'écriture décimale avec un certain nombre de chiffres significatifs. Seuls des entiers sont manipulés, les calculs sont donc exacts.

Code Python :

```
def affiche(self,nbchiffres) :
    """affichage tronqué à nbchiffres après la virgule"""
    puiss=10**nbchiffres
    t=self.numerateur*puiss//self.denominateur
    (ent,dec)=(t//puiss,t%puiss)
    print(str(ent),'.',str(dec),sep='')
```

On a ensuite besoin de définir les opérations $+, *, /\ldots$ pour ce nouveau type d'objets. Cela se fait grâce aux fonctions `__add__`, `__mul__`, `__truediv__`.

Code Python :

```
def __add__(self,other) :
    """addition"""
    return Rationnel(self.numerateur*other.denominateur
                     +other.numerateur*self.denominateur,
                     self.denominateur*other.denominateur)

def __mul__(self,other) :
    """produit"""
    return Rationnel(self.numerateur*other.numerateur,
                     self.denominateur*other.denominateur)

def __truediv__(self,other) :
    """division avec le signe /"""
    return Rationnel(self.numerateur*other.denominateur,
                     self.denominateur*other.numerateur)
```

Vous trouverez d'autres méthodes complétant cette classe dans le fichier `rationnel.py`.

7.4.3 Écriture d'une classe pour les polynômes

Écrire une classe `Polynome`. Son unique attribut sera une liste, représentant les coefficients par degrés croissants. Elle devra en particulier implémenter :

- une méthode `degré` ;
- une méthode permettant de tester l'égalité de 2 polynômes (`__eq__`), qui sera appelée automatiquement lors d'un test avec d'égalité avec `==`. Attention aux problèmes d'arrondis (on ne peut pas tester l'égalité de 2 flottants avec `==`), et au fait qu'il peut y avoir des 0 en trop à la fin de la liste représentant un polynôme...
- une méthode permettant d'évaluer un polynôme en un réel x (appelée `__call(self,x)__`, ce qui permettra de l'appeler avec la syntaxe naturelle `P(x)`) ;
- une méthode `__add__` permettant d'ajouter 2 polynômes ;
- une méthode `__mul__` permettant de multiplier 2 polynômes ;
- (plus difficile) des méthodes `__floordiv__` et `__mod__`, permettant de calculer les quotients et restes de la division euclidienne de 2 polynômes (à l'aide des commandes `P//Q` et `P%Q`).

7.4.4 Écriture d'une classe pour les matrices 2×2

Écrire une classe `Matrices` [20]. Son attribut sera un tableau à 2 dimensions, représentant les coefficients de la matrice.
Elle devra en particulier implémenter :

- une fonction (extérieur à la classe) renvoyant la matrice identité de taille 2, et une autre renvoyant la matrice nulle.
- une méthode permettant de savoir si une matrice est nulle ;
- une méthode permettant de savoir si une matrice est diagonale ;
- une méthode calculant la somme de deux matrices ;
- une méthode calculant le produit de deux matrices ;
- une méthode renvoyant le déterminant de la matrice ;

20. Le module *numpy* fournit une class *matrix*, a priori plus efficace que ce que nous pouvons coder naïvement.

- une méthode testant si la matrice est inversible. La tester avec quelques matrices non-triviales pour éviter des surprises...
- une méthode permettant de calculer l'inverse d'une matrice.

Plus ambitieux :

- Écrire une classe pour des matrices de n'importe quelle taille !
- Écrire une classe pour des matrices creuses – c'est-à-dire une matrice avec «beaucoup de 0». Une façon classique de représenter ce type de matrice est d'avoir 3 listes :
  — une pour les valeurs non-nulles ;
  — une pour les numéros de lignes des valeurs non-nulles ;
  — une pour les numéros de colonnes des valeurs non-nulles.

  Par exemple, la matrice $\begin{pmatrix} 1 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & -3 & 0 \end{pmatrix}$ sera représentée par les 3 listes `[1,2,-3]`, `[0,0,2]` et `[0,2,1]`.

A Très courte bibliographie

Avec un peu plus de pratique, pour aller plus loin dans le langage – et pour coder plus proprement, on pourra se référer à *Effective Python*, de Brett Slatkin (Pearson, Addison-Wesley).